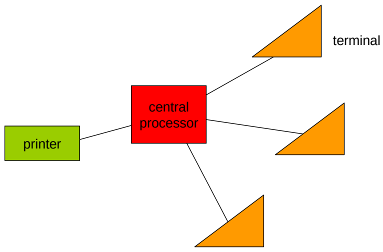
Data Communication Networks

Lecture 1

Saad Mneimneh
Computer Science
Hunter College of CUNY
New York

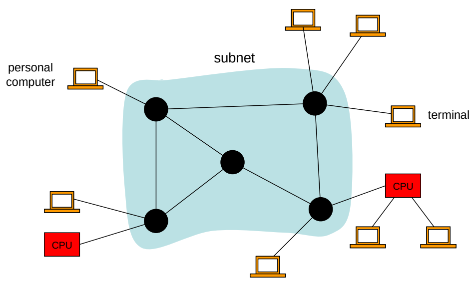
Primitive forms of networks

- Early societies: smoke signals
- 1800s: Telegraphy, messages manually encoded into strings of symbols and manually transmitted, and relayed if needed
  - e.g. Morse code, short and long marks V: ...—
- 1950s - 1960s: Central computers, time-sharing



Early networks

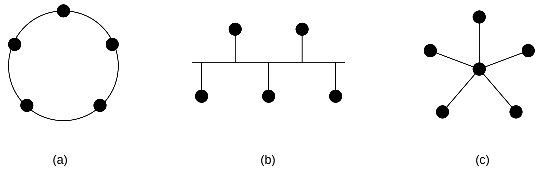
- First large scale computer networks connecting geographically distributed systems
  - e.g. ARPATNET, TYMNET



- Instead of having a computer as center of network, subnet becomes central
  - set of nodes (these are computers in their own right)
  - various pairs of nodes are connected by links

Topology

- The placement of links in a subnet is often called Topology
- The previous topology is somewhat arbitrary, which is typical of Wide Area Networks
  - Wide Area Network: WAN, covering more than a metropolitan area
  - Local Area Network: LAN, covering few square kilometers at most
    - e.g. building, department
- In contrast to WANs, LANs have more structured topologies



(a) Ring, (b) Bus, (c) Star

A falvor of network protocols

- For a network to function properly and provide useful communication, it is important to establish rules by which network activity is conducted
- Such rules are known as protocol
- One possible problem is the coordination of message transmission
  - e.g. if all computers
    - insist on transmitting their own messages
    - without relaying other messages

    then no communication is achieved
- We will look at two protocols
  - Token Ring
  - CSMA/CD (Ethernet)

Token Ring (IBM 1970)

- Machines are on a ring topology
- Messages are transmitted and relayed in **only one direction**
- Each message contains information about sender and receiver
- When a message reaches its destination, the machine keeps a copy of it, but continues relaying it
- When a message reaches back to sender, the sender knows it must have been delivered and stops relaying it

But ... Machines can still be selfish!

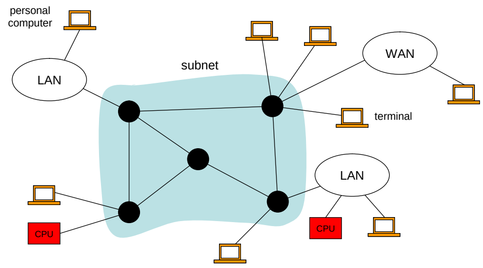
- A unique pattern called **token** is passed around the ring
- A machine can grab the token, i.e. stops relaying it
- Only a machine with the possession of the token can transmit its own message
- All other machines just act as relays
- When a machine sees its own message, it releases the token

CSMA/CD (Ethernet)

- CSMA/CD Carrier Sense Multiple Access with Collision Detection
- Machines are on a bus topology
- Each message is broadcast on the bus (MA)
- Every machine keeps those messages that are addressed to it
- To transmit, each machine (CS):
  - senses the bus
  - waits until the bus is silent
  - begins transmission
- If another machine begins transmitting, both "detect a clash" (CD) and pause for a **random** period, then retry

WANs and LANs

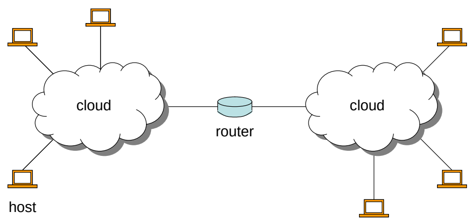
- In 1980s, many networks started to connect to each other



- Practically, a network of networks is more complicated than just a network of nodes
  - not all networks are created equal ;)
  - networks evolve differently
  - the have different conventions and control algorithms
    - e.g. packet format, message forwarding mechanisms, routing
- Bridges and gateways/routers are used to connect different networks

Network in modern terminology

- With LANs, WANs, and routers, network connectivity occurs at many different levels
- We define a network recursively
  - two or more nodes connected by a physical link
    - point-to-point
    - multiple access (like Token Ring or Ethernet)
  - two or more networks connected by a node (router/gateway)



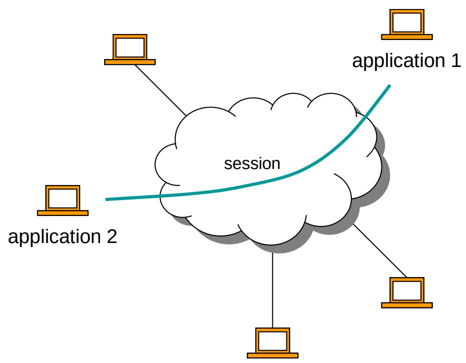
- Nodes inside a cloud implement the network function or the switching function

Messages

- So far we have been deliberately using the term "message" as a unit of communication
- **Q**: What really constitutes a unit of communication?
- **A**: It depends!
  - user: receiving partial message is worthless
  - subnet/computer: in most networks, messages are broken into smaller chunks, called *packets*, for effective transmission (**later on this**)
  - link: a message is just a sequence of bits
- Moral of the story: Distinguish between a message and its representation
- A message carries a specific information that does not change; however, the message may undergo several transformations as it travels from sender to receiver
  - broken into packets (and reassembled)
  - compressed/decompressed
  - encrypted/decrypted

Sessions

- Messages between two users occur as a sequence in a larger transaction, such sequence or transaction is known as a *session*
- We can think of a session as a logical channel over which application level processes are communicating



- What service a session should provide?

Sessions (cont.)

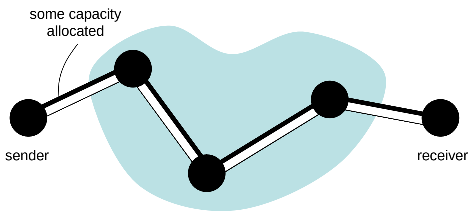
- The service provided depends on the application:
  - Should we allow messages to be lost?
  - Should we receive messages in order?
- Regardless of different guarantees, we may put sessions in two categories
  - When a setup procedure is required to initiate the session, the session is called *a connection*
  - If no such setup is required, and each message is treated independently, the session is *connectionless*
- The way messages are forwarded in the network (**switching technique**) affects the properties of a session

Switching techniques

- Circuit switching
- Store-and-forward (packet switching)
  - datagram
  - virtual circuit

Circuit switching

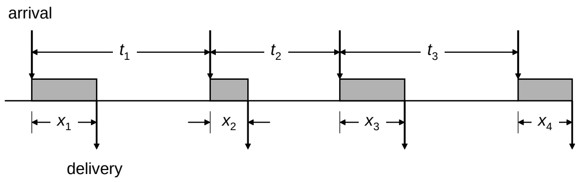
- When the session is initiated (connection), it is allocated a given rate λ in bits per second bps
- A path is then chosen from source to destination
- Each link on the path allocates the desired rate λ of its capacity C



- If capacity is used, future sessions are blocked (e.g. phone network)

Circuit switching (cont.)

- Advantages:
  - dedicated resource
  - fixed delay
  - guaranteed delivery
- Disadvantages:
  - typically in data networks, sessions are low duty factor, i.e. bursty
  - sessions tend to have a short burst of activity followed by long inactive period
  - circuit switching wastes allocated rate during idle times



Circuit switching (cont.)

Let

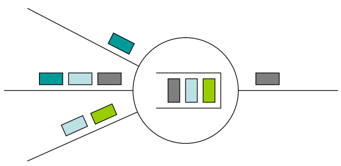
$$\begin{aligned} \lambda = & \quad \text{message arrival rate } (1/\lambda = E[t_i]) \\ r = & \quad \text{allocated rate} \\ L = & \quad \text{expected message length } (L = E[X_i]) \end{aligned}$$

Then the expected delay over a link is L/r.

- If $L/r << 1/\lambda$, the link is under-utilized (previous figure).
- If the allowable delay is T, then we must have $L/r < T$. But if $T << 1/\lambda$, then $L/r << 1/\lambda$.
- Sessions for which $T << 1/\lambda$ are called bursty.

Store-and-forward
(packet switching)

- To overcome the inefficiency of circuit switching, most data networks use what is known as *statistical multiplexing*:
  - links are shared among sessions over time on a demand basis
  - to ensure that every session gets a turn to transmit, messages are broken into limited size blocks known as **packets**
  - one packet is transmitted at a time using the full capacity of the link
  - a packet may need to use a link that is not available, because another packet is being **forwarded** $\Rightarrow$ must be **stored** in a queue



- Althoug queueing delays are hard to manage and control, it can be shown that using links on a need basis often reduces delays in networks
  - e.g. issues such as buffer overflow arise $\Rightarrow$ congestion control algorithms are needed

Store-and-forward (cont.)

Store-and-forward

- Datagram (connectionless)
- Virtual circuits (connection-oriented)

Datagram

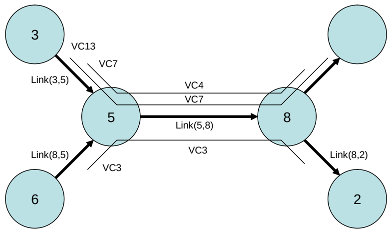
- Route is chosen on a packet by packet basis
- Different packets may follow different routes
- Packets may arrive out of order

Example: IP (Internet Protocol)

- Since no path is established, the source and destination addresses (e.g IP addresses) must be included in each packet (header), i.e. $2\lceil \log n \rceil$ bits overhead (where n is the number of nodes)
- The Internet mostly implements this switching technique

Virtual circuit

- Packets of a session follow the same path
- Route is chosen at the start of the session
- Each link can be visualized as being shared by a set of "virtual channels" VCs
- When the session is set up, a path is established by assigning, one each link of the path, one unused VC



- Each node maintains a mapping of VCs

Node 5 table
(3,5) VC13 → (5,8) VC7
(3,5) VC7 → (5,8) VC4
(6,5) VC3 → (5,8) VC3

Virtual circuit (cont.)

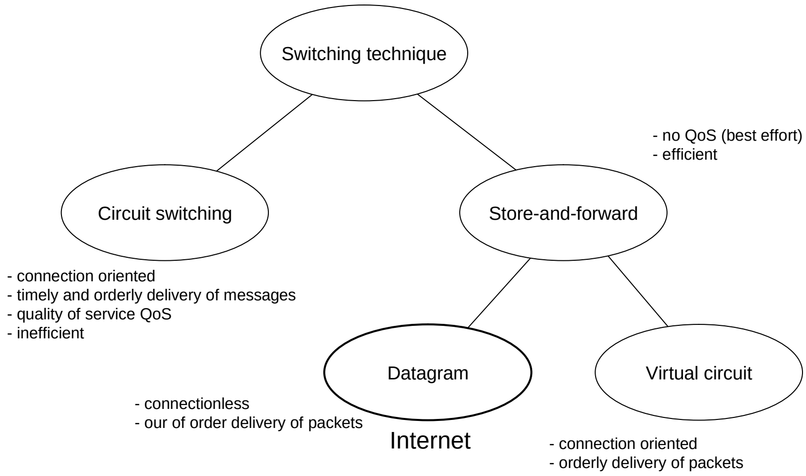
- Global addresses needed to establish virtual circuit when session start
- Once established, local VC numbers can be used for routing
- Worst case we have $n(n-1)$ sessions, and

$$\lceil \log n(n-1) \rceil < \lceil \log n^2 \rceil = \lceil 2 \log n \rceil \leq 2 \lceil \log n \rceil$$

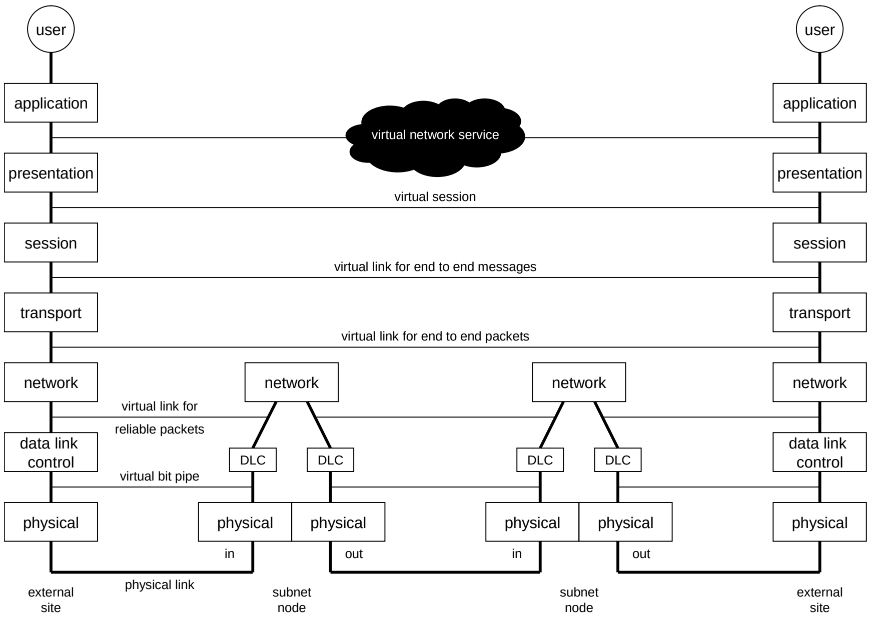
bits overhead is needed to store VC numbers; however, generally virtual circuit requires less overhead in packet header

Example: X.25 and ATM

Switching summary



A network is like an onion
(ISO OSI and IETF standard)

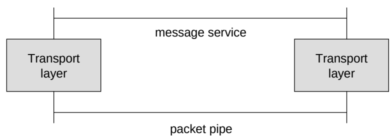


Top layers

- **Application layer**: This is the application that is used to access the network. Each application performs something specific to the user needs, e.g. browsing the web, transferring files, sending email, etc...
- **Presentation layer**: The main functions of the presentation layer are data formats, data encryption/decryption, data compression/decompression, etc...
- **Session layer**: Mainly deals with access rights in setting up sessions, e.g. who has access to particular network services, billing functions, etc...
- There is not a strong agreement about the definition of these three top layers. Usually the focus is on the Transport layer, the Network layer, and the DLC layer.

Transport layer

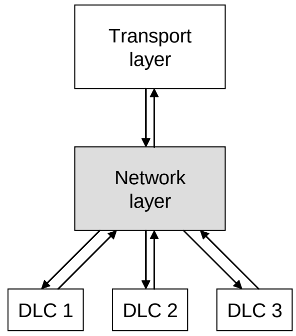
- The network layer provides end-to-end packet pipe to the transport layer
- The transport layer provides end-to-end message service to higher layers



- Functions of transport layer include:
  - Break messages into packets and reassemble packets into messages (packets of suitable size to network)
  - Resequence packets at destination to retrieve correct order (e.g. Datagram)
  - Achieve end-to-end reliable communication in case network is not reliable, recover from errors and failures (arbitrary networks can join the Internet!)
  - Flow control
- E.g. TCP, UDP

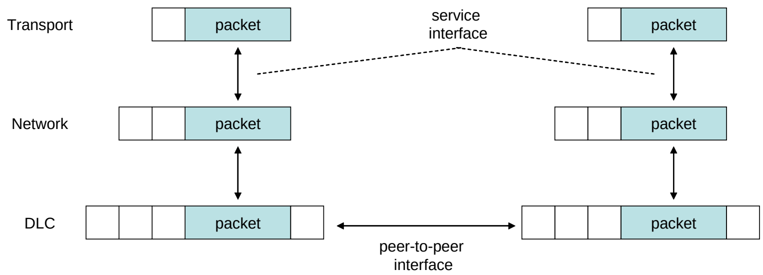
Network layer

- The network layer accepts incoming packets from the transport layer and transit packet from the DLC layer
- The network layer routed each packet to the proper outgoing DLC or to transport layer (if destination)
- Typically, the network layer adds its own header to the packet received from the transport layer to accomplish the **routing** function
  - e.g. source/destination, VC number



Header and Trailer

- Each layer/protocol provides a service to the upper layer/protocol
- Peer processes/protocols communicate information through headers



- The DLC adds a trailer for error detection and correction

DLC

- DLC is responsible for error-free transmission of packets across a **single link**
- The goal to ensure that every packet is
  - delivered once,
  - only once,
  - without errors,
  - and in order
- To accomplish this task, DLC adds its own header/trailer
  - e.g. header may contain sequence numbers to ensure delivery of packets in order
- The packet thus modified is called a *frame*

Physical layer

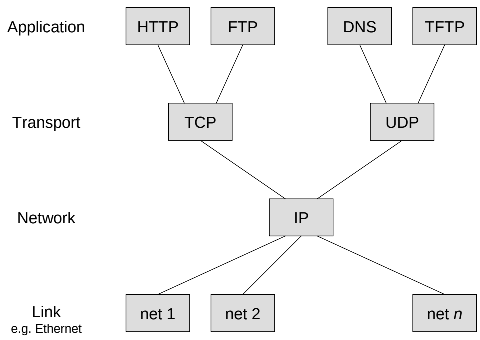
- Responsible for actual transmission of bits over a link
- Delays
  - Propagation delay: time it takes for signal to travel from one end of link to another = distance/speed of light
  - Bandwidth: number of bits that can be transmitted over a period of time, i.e. bits per second (bps)
  - Latency of packet =

    $$\text{Propagation delay} + \text{size of packet/Bandwidth} + \text{Queuing delay}$$

  - RTT = Round Trip Time for exchanging small messages ≈ 2(Propagation delay + Queuing)
- Errors
  - signal experiences power loss
  - noise
  - simple channel model: Binary Symmetric Channel $p\{0 \to 1\} = p\{1 \to 0\} = p$.
  - in practice errors are bursty

The hourglass shape
(an alternative view)

- Internet architecture does not imply strict layering
  - many times there are multiple protocols provided at a given layer in the system, giving an hourglass shape to the architecture
  - An application is free to bypass the defined transport layers and to directly use IP



- IP serves as focal point (common method for exchanging packets among networks)
  - many transport protocols lie above IP (define services)
  - many networks technologies lie below IP

Network software

- Most network protocols are implemented in software (that's one of the main reason for the Internet's success)
  - All computer systems implement their protocols as part of the operating system
  - The operating systems exports the network functionality as a Network Application Programming Interface (API)
- Some API is widely accepted and supported: the *socket interface*
  - the main abstraction of the socket interface is the *socket*
  - socket: a point where the local application process attaches to the network
  - the API specifies ways to
    - create sockets
    - attach the socket to the network
    - send/receive through the socket
    - close the socket

Create a socket

```
int socket (int domain, int type, int protocol)
```

- **domain**: specifies protocol family
  - PF_INET denotes the Internet family
  - PF_UNIX denotes the Unix pipe family
  - PF_PACKET denotes direct access to the network interface, i.e. bypass TCP
- **type**: specifies the semantics of the communication
  - SOCK_STREAM denotes a byte stream
  - SOCK_DGRAM denotes a message oriented service
- **protocol**: identifies specific protocol to be used, in our case we can simply ignore it because the combination PF_INET and SOCK_STREAM implies TCP
- The return value is an identifier of the newly created socket

Server

```
int bind(int socket, sockaddr * address, int addr_len)
int listen(int socket, int backlog)
int accept(int socket, sockaddr * address, int * addr_len)
```

- The **bind** operation binds the socket to the network address, a data structure that includes
  - IP address
  - TCP port (a port is used to identify the process)
- The **listen** operation defines how many connections can be pending on a given socket
- The **accept** operation is a blocking operation that does not return until a remote participant has established a connection
  - it returns a new socket that corresponds to just this connection
  - the address argument contains the remote participant's address

Client

```
int connect(int socket, sockaddr * address, int addr_len)
```

- The connect operation does not return until TCP has successfully established a connection
  - address contains the remote participants (server) address
  - client typically does not care which port it uses, OS simply selects an unused one
- Once a connection is established, the client can use the following to send and receive data:

```
int send(int socket, char * message, int msg_len, int flags)
int recv(int socket, char * buffer, int buf_len, int flags)
```

Example server

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;
const int max_pending = 1;
const int max_len = 256;

int main() {

  sockaddr_in address; //address
  sockaddr_in client_address; //client address
  char message[max_len];

  int s;
  int new_s;
  int len;

  //build address
  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = htonl(INADDR_ANY);
  address.sin_port = htons(port);

  //setup passive open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }
```

Example server (cont.)

```
  //bind socket to address
  if (bind(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in bind";
    return 0;
  }

  if (listen(s, max_pending) < 0) {
    cout<<"error in listen";
    return 0;
  }

  //wait for connection, then receive message
  socklen_t size;
  while (1) {
    if ((new_s = accept(s, (sockaddr *)&client_address, &size)) < 0) {
      cout<<"error in accept";
      return 0;
    }
    while (len = recv(new_s, message, sizeof(message), 0))
        cout<<message<<"\n";
    close(new_s);
  }
}
```

Example client

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

using std::cout;
using std::cin;

const unsigned short int port = 5432;

int main() {
  int s;
  sockaddr_in address;

  bzero((char *)&address, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_port = htons(port);
  address.sin_addr.s_addr = htonl(INADDR_ANY); //my IP address


  //active open
  if ((s=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    cout<<"error in socket";
    return 0;
  }
```

Example client (cont.)

```
  //connect
  if (connect(s, (sockaddr *)&address, sizeof(address)) < 0) {
    cout<<"error in connect";
    close(s);
    return 0;
  }

  char message[256];
  while(cin.getline(message, 256, '\n')) {
    if (strlen(message) == 0)
      break;
    send(s,message,strlen(message)+1,0);
  }
  close(s);
}
```