# CS118: Computer Network Fundamentals

Professor Lu

Spring 2020

## Contents

# CS118: Computer Network Fundamentals

---

## Overview

---

- **computer networks** allow for interaction and communications between computers
  - requires certain hardware and software components
  - involves standardized **network protocols**
    - protocols are complex, with different layers, eg. application, transport, or link layers
    - used by developers for **network programming**
    - eg. the **TCP** and **IP** protocol suite used in the today's internet
- the **internet** is a global network for computers
  - hierarchal, has global, regional, and local levels
    - managed by different **internet service providers (ISP)**
  - *nuts and bolts* view:
    - **hosts** are the end systems running various network apps
      - billions of connected computing devices
      - clients *and* servers
    - **communication links**, eg. fiber, copper, radio
      - wired or wireless
      - each has an associated transmission rate and bandwidth
      - different types of connections, eg. phone-wireless, phone-base, router-router, router-server
    - **routers** and **switches**
      - deals with transferring **packets** ie. chunks of data
      - act as the in-between between hosts and do not run network apps
  - the **network edge** is made up of the hosts, access networks, and various physical media
  - the **network core** acts as a backbone that deals with actually transferring the data
    - consists of interconnected routers and the packet/circuit switching method used

### Access Networks

---

- **digital subscriber line (DSL)**:
  - uses the *existing* dedicated *telephone* line to connect to a central **DSL access multiplexer (DLSAM)**
    - **splitter** sends data on the DSL line through internet and voice on the DSL line to telephone net
    - DLSAM is handled by an ISP
  - requires a dedicated hardware device called a **DSL modem**
  - downstream transmission rate is usually *much faster* than the upstream transmission rate
    - based on user patterns, users typically download much more than they upload
- **cable network**:
  - alternatively, use the *television* line
  - *similarities* with DSL:
    - data and TV is *split* and transmitted at different frequencies over a shared cable distribution network
    - requires hardware device called **cable modem**
    - connected to a central **cable modem termination system (CMTS)** or **cable headend**
    - CMTS is handled by an ISP
    - asymmetric transmission rate
  - unlike DSL, multiple homes are connected via the cable network to the ISP's cable headend
    - access network is shared, instead of having dedicated access to the central office as with DSL
- **home network**:
  - a *lower* hierarchy of networks
  - within the home, a **wireless access point** is connected to the DSL or cable modem
    - various devices can *wirelessly* connect to the access point
    - speed of access point is slower than a direct *wired* connection
      - speed also dependent on the wifi card of the device connecting to the access point
- **enterprise access network** or **Ethernet**:
  - uses a special hardware device called an **Ethernet switch**
  - connected with ISP through some institutional link and router
  - allows for *much higher* possible transmission rates
  - end systems typically connect into Ethernet switch, eg. WiFi router and PC
- **wireless access networks**:

- shared access networks that connect end systems to routers *wirelessly*
- wireless **local area network (LAN)** can reach within a building (100 ft)
  - supports up to 450 Mbps rate
  - eg. 802.11 b/g/n
- **wide-area wireless access** coverage is almost universal (10's km)
  - provided by a cellular operator
  - much slower, between 1 and 10 Mbps
  - eg. 4G, 5G, LTE

### Physical Media

---

- data is *physically* transferred using **bits** that propagate between transmitter/receiver pairs
- a **physical link** lies between the transmitter and receiver
  - eg. common **twisted pair** with two insulated copper wires
- **guided media**:
  - signals propagate through *solid* media, eg. copper, fiber, coax
  - coax cable is made of concentric conductors, allows for bidirectionality
    - supports multiple channels, **hybrid fiber coax (HFC)**
  - fiber optic cable is a glass fiber carrying light pulses to represent bits
    - allows for extremely high-speed operation
    - *immune* to electromagnetic noise
- **unguided media**:
  - signals carried freely through electromagnetic spectrum
    - no physical wire
  - has issues of reflection, obstruction, inteference
  - eg. LAN, wide-area, satellite

### Network Core

---

- the **network core** is a mesh of interconnected routers
  - its role is to send **packets** or chunks of data between hosts
- two key *functions*:
  - **forwarding** relays packets from a router's input to the appropriate router output
  - **routing** determines the source-destination route taken by packets
    - these routes are computed locally and proactively, and are stored

within the router

- key *technologies*:
  - **packet switching**:
    - hosts *break* application-layer messages into packets
    - packets are forwarded between routers, across links, from source to destination
      - packets *hop* through a certain number of intermediate nodes
    - each packet is transmitted *back-to-back*, not simultaneously, allowing for **full link capacity** transferrence
      - sending packets takes time (L bits) / (transmission rate R bits/sec)
    - entire packet must arrive before it can be transmitted (**store and forward**)
      - thus, the *end-to-end* delay is therefore *scaled* to the number of hops the packet must make
  - alternatively, **circuit switching**:
    - used in traditional telephone networks
    - no packets, switching granularity is in terms of **circuits**
    - resources/circuits are *dedicated* for a particular call
    - reservation-based, no sharing of an in-use circuit
    - circuits are *released* on call completion
  - *sharing between users* with circuit switching:
    - with **frequency division multiplexing (FDM)**, split up frequency domain
    - alternatiely, with **time division multiplexing (TDM)**, use time slices and time sharing
- why is packet switching used by the internet over circuit switching?
  - circuit switching is less **robust**, in that if a part of a circuit fails, it may break the entire network
    - on the other hand, with packet switching, the network infrastructure is maintained even if some routers go down
  - packet switching also allows for *more users* to use the network
    - many users will be *idle* for a percentage of their time on the network
    - eg. with a 1 Mbs link, and each user using 100 Kbs and active 10% of the time:
      - this user pattern is an example of *bursty data*
      - for circuit switching, can only support up to 1 Mbs / 100 Kbs = 10 users at a time (*dedicated* circuits)
      - for packet switching, can support 35 users with a probability that > 10 are active that is less than 0.0004
  - the probability that x users are active is: $P(N,x) = \binom{N}{x} p^x (1-p)^{N-x}$

    * in order to afford a certain number of users, the probability that more than the threshold number of users are active at the same time should be extremely small
  - however, excessive **congestion** is still possible with packet switching:
    * packet delay and loss may occur when the network becomes overloaded with active users
      · packets may have to jump more links in order to alleviate network congestion
    * thus, certain protocols are needed for reliable data transfer and congestion control
  - ie. circuit switching uses *reserved* resources and allows for consistent service, while packet switching uses *on-demand* allocation and less guaranteed service

**Packet Delay, Loss, and Throughput**

---

- if the arrival rate to a link *exceeds* the transmission rate for a time:
  - packets will **queue**, and await transmission
    * the **queuing delay** is the time waited in the buffer before transmitted
    * *different* from **transmission delay**, the total amount of time to transmit all bits of a packet
  - packets can then be **lost** or dropped if the memory buffer for the queue fills up
- thus **packet delay** overall has multiple sources:
  - **processing delay** from checking bit errors and determining output link
  - **queuing delay** from awaiting transmission, depends on congestion
    * as (L bits * a average arrival rate) / R rate approaches 1, queuing delay becomes large
    * above 1, the average delay becomes infinite
  - **transmission delay** is how long it takes to push out all bits of the packet, depends on packet size
    * L bits / R rate
  - **propagation delay** is the time for a bit to actually travel to another router
    * d length / s speed
- the `traceroute` program provides delay measurement from source to destination
  - send three probe packet that reaches each router along the path
  - measures time interval between transmission and reply
- handling **packet loss**:
  - when a packet is lost, the source must slow its transmission, and also re-

transmit the lost packet
    * different *response* for different *applications*:
        · eg. for video streaming, the media will buffer and prioritize lower delay and allow dropping of some packets
        · eg. for emails and communications, delay is not as important as data integrity
  – the exact response is dictated by different transmission protocols eg. TCP
- the **throughput** is the rate at which bits are transferred between sender and receiver
  – can be *instantaneous* or *average*
  – often constrained by the slowest **bottleneck link** in the network

### The Internet

---

- the **internet** is built as a network of networks
- given *millions* of access ISPs, how should they be connected to one another?
  1. pairwise connections, ie. connect each ISP to every other
  – fully distributed and requires $O(n^2)$ connections
  – this solution doesn't scale
  2. connect each ISP to a *global* transit ISP
  – full centralized solution
  – this global ISP becomes a *bottleneck* as all traffic passes through it
  3. use *multiple* global ISPs
  – a natural byproduct of a single global ISP from competition
  – each only serves a subset of its local networks
  – requires **peering links** and **internet exchange points (IXP)** between the global ISPs
    * IXP are managed by a third party
    * note that these are less of a bottleneck since global ISPs want to minimize user interaction with another ISP
  4. *hiearchical* structure
  – this is the current structure of the internet
  – at a lower level, several access ISPs are connected to a global ISP through a **regional net**
  – creates a **hierarchy** from access ISPs, to regional nets, to global ISPs
  – another unique level is the **content provider network** eg. Google that brings services and content directly to end users, bypassing the hiearchy
  – this structure is motiviated more by business concerns than technical concerns

# Appendix

---

### Network Programming

---

- *application layer*:
  - applications using the network, eg. client-server model
  - **client** initiates communication and awaits the server's response
  - **server** responds to requests
    - discoverable by clients
      - always running, waiting for client connections
    - processes requests and sends replies
      - requests can be processed *concurrently*, *sequentially*, or some *hybrid*
  - however, client and server are not always disjoint
    - eg. server can be a client to another server
- *transport layer*:
  - responsible for actually providing communication services (in conjunction with lower layers)
  - outlined by **protocols** such as TCP and UDP
- **transmission control protocol (TCP)**:
  - full-duplex byte stream
  - has guarantees for *reliable* data transfer:
    - deliveries are completed
    - data is ordered, with no duplicates
  - allows for regulated flow for flow and congestion control
- **user data protocl (UDP)**:
  - variable length datagram transfer
  - very basic transmission service
  - no reliability, order, or delivery guarantee
  - no flow or congestion control
- **socket programming**:
  - a **socket** is an endpoint on the network
    - tuple of **IP address** and **port number**
  - socket programming helps to build the communication tunnel between application and transport layer
  - multiple types of sockets, eg. stream sockets, datagram sockets, and raw sockets for different protocols

- basic steps for working with TCP sockets:
  - create service
  - establish TCP connection
  - send and receive data
  - close TCP connection
- TCP server:
  - create a `socket`
  - `bind` socket to an address
  - `listen` for clients
  - `accept`, blocked until a connection from client
  - `read` and `write` data
- TCP client:
  - create a `socket`
  - `connect` to a server address
  - `read` and `write` data

### Socket Programming API

---

`int socket(int domain, int type, int protocol)` creates a socket:

- returns socket descriptor or -1 and sets `errno`
- `domain` is the protocol family, eg. `PF_INET`, `PF_INET6`
- `type` is communication style, eg. `SOCK_STREAM` for TCP and `SOCK_DGRAM` for UDP
- `protocol` is the specific protocol within family, usually 0

`int bind(int sockfd, struct sockaddr* myaddr, int addrlen)` binds a socket to a *local* address:

- returns 0 or -1 and sets `errno`
- `sockfd` is the socket file descriptor
- `myaddr` is a structure including the IP address and port number
- `sockaddr` and `sockaddr_in` structures are the same size
  - typically, use `sockaddr_in` and cast to `socketaddr`
- `addrlen` is `sizeof(struct sockaddr_in)`

`sockaddr` and `sockaddr_in`:

```
struct sockaddr {
  short sa_family;
  char sa_data[14];
```

```c
};

struct sockaddr_in {
  short sin_family;          // protocol family, eg. AF_INET (same as PF_INET)
  ushort sin_port;           // port number
  struct in_addr sin_addr;   // IP address
  unsigned char sin_zero[8]; // buffer for having same size as sockadddr
};

struct in_addr { // used for IPv4 only
  uint32_t sin_port;
};
```

`int listen(int sockfd, int backlog)` waits for connections:

- returns 0 or -1 and sets `errno`
- `sockfd` is the socket file descfiptor
- `backlog` is number of connections program can serve simultaneously

`int accept(int sockfd, struct sockaddr* client_addr, int* addrlen)` accepts a new connection:

- return client's socket file descriptor or -1 and sets `errno`
- `sockfd` is the socket file descriptor for server
- `client_addr` to be filled in with IP address and port number of client
- `addrlen` to be filled in with size of the `client_addr`
- a new socket is being cloned from the client
  - if there are no incoming connections to accept, there are multiple blocking modes
    * non-block: `accept` can return -1
    * blocking: `accept` operation is added to the wait queue

`int connect(int sockfd, struct sockaddr* server_addr, int addrlen)` connects a socket to a *remote* address:

- return 0 or -1 and sets `errno`
- `sockfd` is the socket file descriptor to be connected
- `server_addr` is the IP address and port number of the server

- `addrlen` is `sizeof(struct sockaddr_in)`

`int write(int sockfd, char* buf, size_t nbytes)` and `int read(int sockfd, char* buf, size_t nbytes)` read and write data from a TCP stream:

- returns number of bytes processed or -1
  - 0 if socket is closed
- `sockfd` is the socket file desriptor
- `buf` is a data buffer
- `nbytes` is the number of bytes to process
  - max to read, or desired number to send

`int close(int sockfd)` closes a socket:

- returns 0 or -1
- sockfd is no longer valid

### Utilities

- note that **byte ordering** matters when transferring data between host systems
  - little endian vs. big endian
  - hosts may use different orderings, but the **network byte order** is always big endian
  - `ntohl` performs net-to-host long translation
  - `ntohs` performs net-to-host short translation
  - `htonl` performs host-to-net long translation
  - `htons` performs host-to-net short translation
  - thus, the port number and IP address in the API address structures should always be converted
    - eg. `servaddr.sin_port = htons(servport)`
- other utilities are provided for working with network addresses:
  - `struct hostent* gethostbyname(const char* hostname)` translates a host name to an IP address
  - `struct hostent* gethostbyaddr(const char* addr, size_t len, int family)` translates an IP address to host name
  - `char* inet_ntoa(struct in_addr inaddr)` translates IP dddres to a dotted-decminal string
  - `int gethostname(char* name, size_t namelen)` reads the local host's name
  - `in_addr_t inet_addr(const char* str)` translates dotted-decimal string to IP address *in* network byte order
    - `int inet_aton(const char* str, struct in_addr* inaddr)` same translation, different format

`hostent` structure:

```
struct hostent {
  char*  h_name;      // canonical host name
  char** h_aliases;   // list of aliases, last element is NULL
  int    h_addrtype;  // address type, eg. AF_INET
  int    h_length;    // length of addresses, eg. 4 for IPv4
  char** h_addr_list; // list of IP addresses
  // h_addr is an alias for h_addr_list[0]
};
```