<u>Internet</u>

Terminology:

- ***Hosts/end systems***: Systems accessing the Internet
    - Hosts run *network applications* - send & receive data packets
- ***Routers***: Packet switches inside a network
- ***Communication links*** refer to how hosts, local & ISP routers, etc. are connected
    - Has associated transmission rate/bandwidth (BW) - bits/sec
- ***Internet Service Providers***: regional & global; local network routers connect to regional ISP routers

Internet protocol layers:

1. **Application layer**: Support data exchange between applications (ex: SMTP, HTTP, DNS)
2. **Transport layer**: Handle delivery of packets, reliability (ex: TCP, UDP)
    a. ***Multiplexing*** within a host (keeping up multiple "convos" in a single Internet access pt.)
3. **Network layer**: Define how formatted packets are forwarded from src → dest (Ex: IP)
    a. *Packet-switching* (at each jump: take best path) vs *circuit switching* (predefined path, reserved resources)
4. **Link layer**: Define how data is transferred between directly connected network elements
5. *Physical layer*: Physical transfer of data

Each layer encapsulates the previous layer's message/packet within a larger packet; removed in reverse order

<u>Computer Networks</u>

Internet: routers do packet switching, statistical multiplexing (dividing larger chunks of data into smaller packets)

- ***Store-and-forward***: packet switch temporarily buffers up packets, forwards to host once full packet is received

Network performance: throughput & packet loss

- **Throughput** (*bits/sec = bps*): Measures bandwidth of a link, point-to-point; hard for multi-access
    - Multiple connections: bottlenecked by least-throughput connection/pipe
- Packet loss: due to transmission errors & congestion (wired) + mobility - change in # hosts (wireless)

4 sources of network delay**:**

1. *Node processing*: need to check bit errors, determine output link; generally ignored, negligible
2. **Transmission delay**: Packet length / rate (link bandwidth)
    a. Due to store-and-forward: entire packet must arrive at a node before it can move on
3. **Propagation delay**: Distance of physical link / propagation speed
    a. Determined by physical propagation speed in medium (e.g. cable)
4. **Queuing**: # packets in queue * transmission time of each packet (only one can be transmitted at a time)

<u>HTTP</u>

HTTP: Web application-layer protocol (TCP/UDP as transport layer service)

- Client-server model (browser = client, Web server = server)
- Is **stateless**: server maintains no information about past requests (→ less complex; no synch issues)

HTTP versions:

1. **HTTP/1.0**: No persistent connection; at most one object sent over one TCP connection

a. 3 types of request: GET, POST, HEAD
b. <u>Delay</u>: 2 * (# objects) RTT (1 RTT for establishing connection; 1 RTT for data transfer)
    i. To improve: open multiple connections in parallel

2. Persistent **HTTP/1.1**: Use a single TCP connection to send multiple objects
    a. Use KeepAlive field in header to ask server to stay connected
    b. <u>Delay</u>: (# objects + 1) RTT [+1 is RTT for establishing connection]
    c. By default: client issues next request after previous response received (1 RTT per object)
        i. **Pipelining**: client sends requests as soon as it sees a referenced object
            1. <u>Delay</u>: 1 RTT (connect) + 1 RTT (retrieve index file) + data transfer time [+1 RTT: objects]
3. **HTTP/2**: Solves performance issues in HTTP/1.1
    a. HTTP/1.1 has large ASCII header with repetitive information; has to send in full with every query
        i. <u>HTTP/2</u>: ASCII → binary header; ***header compression*** (on subsequent requests: only send the header fields that changed, all other fields assumed unchanged/same as previous; both browser & server keep header table while connection is active)
    b. HTTP/1.1 downloads one-by-one in strict sequential order → requests for large file/dynamic computation will block all following requests (***head-of-line blocking***)
        i. HTTP/2: Request-response pairs encoded in a ***stream*** (stream = bidirectional virtual channel);
            1. ***Frame***: basic unit of communication, distinguish header & data frames (one ***message*** - HTTP request or response - may be encoded in 1 or multiple frames)
        ii. Streams have different priority, can interleave/shuffle by dividing into frames
            1. Can transmit smaller objects before larger ones; priority determined by clients (e.g. load stylesheets before media elements)
        iii. HTTP/2 prevents HOL blocking at HTTP-level; but not at TCP level (HTTP/2 = vanilla TCP; can still HOL block within a TCP connection; due to packet losses → packet recovery, e.g.)
    c. ***Server push***: If an asset (e.g. .html index page) requests additional assets; rather than just giving asset, server returns response with asset + promises to send associated assets
        i. Assets sent automatically in later responses without needing to be requested by the client
        ii. If client moves to a different page, client can know to stop sending assets from previous page
4. **HTTP/3**: Adds security, per-object error and congestion control (more pipelining) over UDP
    a. Switching TCP → UDP solves HOL blocking within a single connection

Cookies: HTTP stateless, but user/server want to be able to maintain state between sessions → use ***cookies***

- **Cookies**: key-value pairs, initially issued by server (put in response header); client keeps cookie file stored, associates a website/domain with a value → client includes value in request headers to that website
    - Server can specify additional parameters for a cookie (e.g. maximum lifetime)
- *3rd-party cookies* to obtain user info across multiple site: when a page loads an advertisement (linking to an external site - advertiser's website), advertisement website returns a cookie with advertisement response
    - On seeing ad. website (same domain) later: advertiser receives cookie, can use to track user

__**DNS**__

Domain Name System (**DNS**): service mapping domain names to IP addresses

- DNS protocol: uses query-reply pattern (name query → IP address response), like HTTP
- Is an application-layer protocol; may use either TCP or UDP for transport (typically: TCP)

Domain names: hierarchical, separated by dots; ex (cs.ucla.edu): (root, .) → edu → ucla → cs

- A node's domain name identifies its position in DNS **name space**
    - No theoretical length limit, but limited to 255 characters in practice

DNS mapping stored as distributed/federated database: each zone (e.g. edu, ucla.edu, cs.ucla.edu) managed by a ***domain zone server*** (called "***authoritative server***"), responsible for records in that domain

- To look up, start by sending domain name query to root zone server; at each level: zone server will forward query to next matching domain zone server (lowest level server: true mapping name → IP address)

Making a query: hosts send domain name queries (historically: via stub resolver) to local ***caching resolvers***/DNS servers, caching resolver sends queries to DNS server; caches results, sends back to hosts

- DNS query protocol used by local DNS servers to query auth. servers; modern-day: public cache resolvers

Namespace governance: ICANN organization manages root servers, assigns and delegates top-level domains/TLDs

- TLDs: generic TLDs (e.g. ".com", ".org") + country code TLDs managed by countries (e.g. ".us", ".kr")
- TLD operators run TLD name servers, allocate 2nd level domain names (e.g. edu → ucla.edu); at each level: a domain owner assigns domains on next level

Often have multiple authoritative name servers for a domain; ideally in different networks (for redundancy)

- Root domain file published on 13 authoritative root DNS servers, administered by various volunteer organizations; root domain file: contains names and IP addresses of TLD authoritative servers

TLD operators contract ***registrars*** (e.g. GoDaddy [US], CoolOcean [India]) to sell domain names to registrants

- Registrars submit change requests to ***DNS registry*** (organization managing DNS namespace; ex: Public Interest Registry) on behalf of registrant
- Registry updates internal domain database; domain database pushes change to other domain name servers

All DNS data stored in domain database as ***resource records***/**RRs**; contain name, type, class, TTL/lifetime, and value

- Lifetime: how long a RR should live in cache before needing to be refreshed; higher levels (of name server) typically have longer TTLs than lower levels
- Referrals: each zone will have a corresponding ***glue RR*** connecting it with its parent; glue RR is stored in both that zone's zone files, and its parent zone files [glue RR (in parent) stores address of the child zone server]

DNS resolution: app first calls DNS to translate name to IP address, then connects to address directly

- System calls: getaddrinfo(), gethostbyname()
- Caching resolver has IP address of root servers hardcoded
    - Caches addresses of every zone server visited at every step of the DNS lookup to reuse in future queries; can use to bypass lookup process, not need to lookup namespaces

DNS data is coded in ***resource record*** (**RR**): name + type + class + TTL + RL + RDATA

- Name: 1-byte length value + list of labels (string, variable length)
- Type: A, AAAA, NS, etc. = IPv4 address, IPv6 address, authoritative name server, etc.

        - MX: mail server, CNAME: canonical name (like symlink), TXT: text record
        - TXT: commonly stores various/arbitrary data, leveraging DNS database
    - Class: protocol family (nowadays: only IN = Internet used)
    - TTL: cache lifetime; set by DNS operators in master file
    - RDATA: interpretation depends on RR type; variable length
        - Ex: IP address (A, AAAA), preference order (MX), DNS server name (NS), real DNS name (CNAME)

The DNS protocol: client-server based, DNS query & reply over UDP/TCP

DNS stores stores multiples of same name, class, type in multiple RRs

- **RRset**: made of all RRs with same name, class, and type

Using DNS for **Content Distribution Networks** (CDN)

- HTTP caching: CDN providers can cache resources (like caching resolver) for lower latency/faster loading
- Web server outsources its DNS record to CDN provider (CDN network server) via CNAME (mapping server domain → CDN IP); user HTTP requests go to CDN server
    - With HTTPS: web server shares crypto key with CDN provider
- DNS for load-balancing: CDNs have multiple servers; guesses geolocation from query IP address → use to determine best CDN server to use; based on IP address + load on each server, sends different DNS address
    - Assign short TTL for final IP address result to ensure refresh

The Transport Layer

TCP: connection-oriented, reliable; format: delivers ***byte streams*** (in order)

- Has other functions: congestion & flow control, e.g.

UDP: connectionless, only provides multiplexing/demultiplexing, unreliable; format: delivers ***datagrams***

Multiplexing & demultiplexing (for applications)

- ***Multiplexing***: sender gathers data from multiple applications, sends as a whole under single message (with associated header); ***demultiplexing***: receiver delivers received segments to correct app layer process

<u>Demultiplexing</u> (transport) - host receives ***IP packet*** containing transport layer data segment + application data

- Data segment contains source, destination IP addresses
- Connectionless (UDP): Only specify source, destination address
- Connection-oriented (TCP): Server creates separate sockets for each client; identified by 4-tuple of <u>source IP address + port number, and destination IP + port number</u> (stored in data segment)
    - Server creates file descriptor for each socket; host uses 4-tuple to direct to appropriate socket

UDP: for DNS, streaming, general loss-tolerant & rate-sensitive applications

- ***Unreliable***: segment may be lost/duplicated/delivered OOO (for reliable transfer, add at application layer)
- ***Connectionless***: no handshake between sender & receiver; each UDP segment handled independently of others
- <u>Header format</u>: source & destination IP address; length + checksum (for error checking)
    - Checksum computed over pseudo header + UDP header, data

Reliable Data Transfer

Stop-and-Wait: sender sends packet, sets retransmission timer, then waits for an **ACK** from the receiver

- Each packet assigned a sequence number (1 bit: 0 or 1); sender resends after retransmission timer times out
- ACK = acknowledgment packet; receiver: if packet has bit error, does nothing → sender resends

Stop-and-Wait with NACK: Only if B receives a packet with a bit error, sends an ACK with sequence number of last correctly received packet; duplicate ACK treated as negative-ACK by sender → sender retransmits

***Go-Back-N*/GBN**: Sender sends up to N unacknowledged packets; receiver keeps track of next expected packet (based on sequence number), acknowledges it once received

- *Sender*: Sets timer for oldest unack'ed packet, retransmits all unack'ed packets within window upon timing out; *Receiver*: Only keeps track of single variable (expected sequence #), discards out-of-order packets
- N: flow control window size, parameter (exchanged in initial handshake)

Selective Repeat: Sender sends up to N unacked packets (flow control window, as before); receiver acknowledges each correctly received packet; acknowledges & buffers out-of-order packets

- Sender keeps timer for first unack'ed packet; when timer expires, retransmits only that single unack'ed packet
- Receiver: can release buffered out-of-order packets when missing packets are received

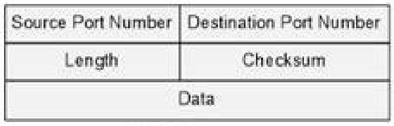
<u>TCP</u>

TCP: one timer, selective repeat (sort of), cumulative ACK

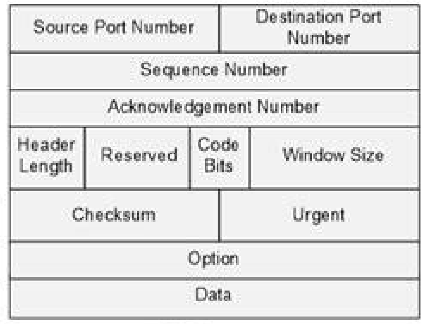
- ***Point-to-point***: creates virtual pipe between 2 processes (returns a socket file descriptor to each process)
- ***Connection-oriented***: sets up a connection before data transmission, tears down connection after finishing
    - New connection → new socket; connection initiation & termination requires action from both sides
- Provides bidirectional + reliable byte stream delivery
    - Note: No message boundaries made by TCP in byte stream, must be done by application layer (e.g.: via HTTP frames)
- ***Flow-controlled***: prevents sender from overwhelming receiver
- ***Congestion control***: mitigates traffic overload inside network; controls send speed to avoid overloading pipe

<u>TCP segment format:</u>

- Common with UDP: Source + dest port #, checksum; at end: application data
    - Checksum - includes TCP header + pseudo header (source + dest IP, zero, protocol, segment length)
- ***Sequence number, acknowledgment number*** (count # of bytes)
    - SEQ #: represents sequence # of first byte in payload; ACK #: next expected byte to receive
    - Sequence number gives offset within current [A->B] byte stream; have to break up a byte stream into multiple messages → sequence # encodes offset (ACK #: given for B->A byte stream)
- ***Receiver window size***: represents buffer capacity of receiver
    - Dynamically adjustable, even after connection has been established
- Data offset field in segment: 4-bit header length, indicates data offset (where payload data starts)
- Have blank field between data offset and flags
- Various flags; e.g. ***ACK***; connection management: ***SYN, FIN, Reset***
- Option fields [variable length, up to 40 bytes]: used heavily; header length 8 -> 40 byte options
- No longer used: pointer to urgent data [for slow computers]; note: no explicit congestion control field



UDP Segment



TCP Segment

Sequence & acknowledgment numbers: TCP uses cumulative ACK

- Store 2 "sequence numbers": 1 for byte stream A→B, 1 for byte stream B→A
  - A sends A→B byte stream offset as its sequence #; returns next expected offset B→A in its ACKs
  - B sends B→A offset as its SEQ, returns next expected A→B for its ACKs
- Next expected byte: computed as previous SEQ + size of previous message's data

TCP connection setup: Via 3-way handshake:

1. Via ***connect()***: Client sends TCP SYN segment (SYN flag - 1, no data) to server
   - Specifies client's initial seq # (selected randomly)
2. Via ***listen()***: Server receives SYN, replies with ACK & SYN control segment
   - SYN, ACK flags=1; ACK # is received SEQ # +1; server specifies its own initial seq # (selected randomly)
   - Upon receiving ACK: client sees connection established
3. Client host sends an ACK packet (ACK flag set to 1; ACK number set as received sequence number +1)
   - Packet may carry data; upon receiving ACK packet, server sees connection established

Closing a TCP connection: EItherend can initiate the closure of its end at any time

1. Via ***close()***: Host A sends TCP FIN control segment (FIN flag 1, no data) to Host B
2. Host B receives FIN, replies with ACK [acknowledges A's FIN]
3. When B finishes sending all of its data and is ready to close, it sends a FIN segment via close()
4. A receives FIN, replies with ACK; B receives ACK → B closes its connection

- Host A never knows if host B receives ACK; simply closes its own connection after "long enough" (2x max segment lifetime, e.g.) without receiving retransmitted FIN

TCP connection reset: On TCP connection setup: system sets up ***TCP control block*** (**TCB**) to track connection state

- TCB identified by source + destination addresses, source + destination ports
  - TCP connection state: receiver flow control window size, sequence # (oldest sent but unacked, latest sent + unacked), segments that arrived out of order, etc.
- Resetting: if TCP receives a non-SYN segment, but cannot find corresponding TCB → replies with RST
  - Receiver of RST aborts connection, all data on connection considered lost
  - Possible causes: due to bit errors, or by attacks
- Other causes: retransmit count hits UB, one side needs to reject/close connections due to resource limitations

TCP flow control: want to prevent sender from overrunning receiver by transmitting too much data too fast

- TCP: receiver informs sender of amount of free buffer space using RcvWindow field of TCP header in every message receiver -> sender; updates value RcvWindow over time/dynamically

- Sender: keeps amount of transmitted, unACKed data to be no more than most recently received RcvWindow

TCP loss detection and recovery

- TCP: one retransmission timer on earliest sent, but unACKed segment S
  - If S is ACKed, restart timer on next unACKed segment; if timer expires, then retransmit starting from S
- # segments to retransmit: min[cwnd, rwnd] (min of receiver flow control, congestion control windows)
- Timer value: TCP sets timer based on estimated RTT plus a safety margin DevRTT (+ params alpha, beta)
  - SampleRTT: time gap between most recent ACKed message send and ACK
  - SRTT: estimated "smoothed" RTT, SRTT = (1-alpha) * SRTT + alpha * SampleRTT (initial: SampleRTT)
  - DevRTT: estimated RTT deviation, DevRTT = (1-beta) * DevRTT + beta * |SRTT - SampleRTT|
  - RTO: retransmission timeout, RTO = SRTT + 4 * DevRTT (initial: 1 sec)
- Retransmissions: less clear when to measure (can take delay between first transmission & final ACK, or last retransmission and final ACK, etc.) -> Karn's Algorithm: in case of retransmission, do not take RTT sample (don't update SRTT/DevRTT), just double retransmission timeout value after each timeout
    - Take RTT measures again upon next successful no-retransmit data transmission

TCP Fast Retransmit: Can detect lost segments via duplicate ACKs, without waiting entire duration of RTO timer

- Segment lost → next arrival at receiver will be OOO; when segment arrives OOO, receiver can immediately send ACK indicating seq # of next byte it is expecting
- Sender receives 3 duplicate for the same sequence # in a row → assume segment with seq # was lost → Fast retransmit, retransmit the one lost segment without waiting for timer to expire

TCP: **Delayed ACK**

- TCP connection carries traffic in both directions → ACKs can be piggybacked on data segments; for one-way data flow, receiver sending (just) an ACK after receiving every segment doubles packet count across Internet
- ***Delayed ACK***: After connection setup, upon receiving a data segment S1, wait a bit to see if another segment S2 will arrive; if yes, send the ACK for both within a single packet; otherwise, just send an ACK for S1
- Issue: causes a little delay in RTT calculation for RTO -> upon OOO arrival, immediately send ACK indicating expected seq #; upon arrival of segment that increases expected seq #, immediately send ACK

Congestion Control

Congestion cases:

- Network congestion: Network achieves maximum possible throughput (too much data sent into network)
  - From too many sources sending data too fast into network at same time
- Congested packet buffer: Buffer fills -> packets are dropped at the router, sender needs to retransmit
  - Duplicates: sender may time out and retransmit if ACKs dropped -> duplicates are delivered
- Unneeded/superfluous retransmissions: multiple copies of the same packet go through overloaded links, reducing effective throughput (packet drop -> "upstream transmission capacity" used for that packet is wasted)

Congestion collapse: occurs when packets are constantly being retransmitted & dropped

- Causes effective load/throughput to stop increasing with offered load, begin decreasing

TCP congestion control: adds congestion control window (**cwnd**) on top of flow control window

- **End host adaptation**: don't rely on network help, try to estimate network state based on packet losses & adjust cwnd automatically [more advanced schemes: also estimate via other variables (e.g. delays, delay changes)]
- Sender limits: LastByteSent-LastByteAcked <= min(cwnd, rcvwin)

Two phases of TCP congestion control (which phase: depends on slow start threshold ***ssthresh***):

1. **Slow start** (cwnd < sstresh): Initialize cwnd = 1 segment, then rapidly increase cwnd
2. **Congestion avoidance** (cwnd >= sstresh): slowly but continuously increase cwnd size
3. Default: initialize ssthresh to flow control window size

Slow start: initialize cwnd = 1 MSS (max segment size, in bytes); want to gauge pipeline size quickly

- Send cwnd-allowed segments; if receive an ACK, increment <u>cwnd = cwnd + 1</u>
- If a packet times out (indicates congestion): set ssthresh = cwnd / 2, set cwnd = 1 MSS, return to step 2

Congestion Avoidance: [Additive Increase, Multiplicative Decrease (AIMD)] - want to cautiously probe for unused resources, quickly recover from overshoot (hopefully: avoid having to reset cwnd to 1 & return to slow-start)

- Send cwnd-allowed segments
    - If the previous (cwnd) packets were ACKed: <u>cwnd += 1 segment</u> (in effect: +1/RTT)
    - If 3 dup ACKs: <u>cwnd = ssthresh = cwnd / 2</u>; if timeout: cwnd = 1 segment, return to slow start

Issue: May be that when cwnd is decreased, already inflight packets (seen from dup ACKs) fall out of the cwnd

- **Fast Recovery**: Inflate cwnd by # duplicate ACKs received (without allowing any new packets to be sent)
    - "Skip" ACK'ed packets in cwnd count; cwnd "deflates" once loss has been recovered
- Cwnd is limit on # of packets inside network (want to count inflight packets); already received -> not inflight
    - Duplicate ACK arrives -> packet out of network -> increase cwnd by 1 segment (cwnd inflation)
- Resume slow-start/congestion avoidance once fast recovery ended

TCP throughput as a function of window size W (ignore slow start) and RTT

- cwnd=W: W / RTT; immediately after loss: cwnd/=2 => throughput = W/2RTT; average (roughly): 0.75 W/RTT

3 states for network: underutilized (no queue), over-utilized (queues form), saturated (queues full, packet loss occurs)

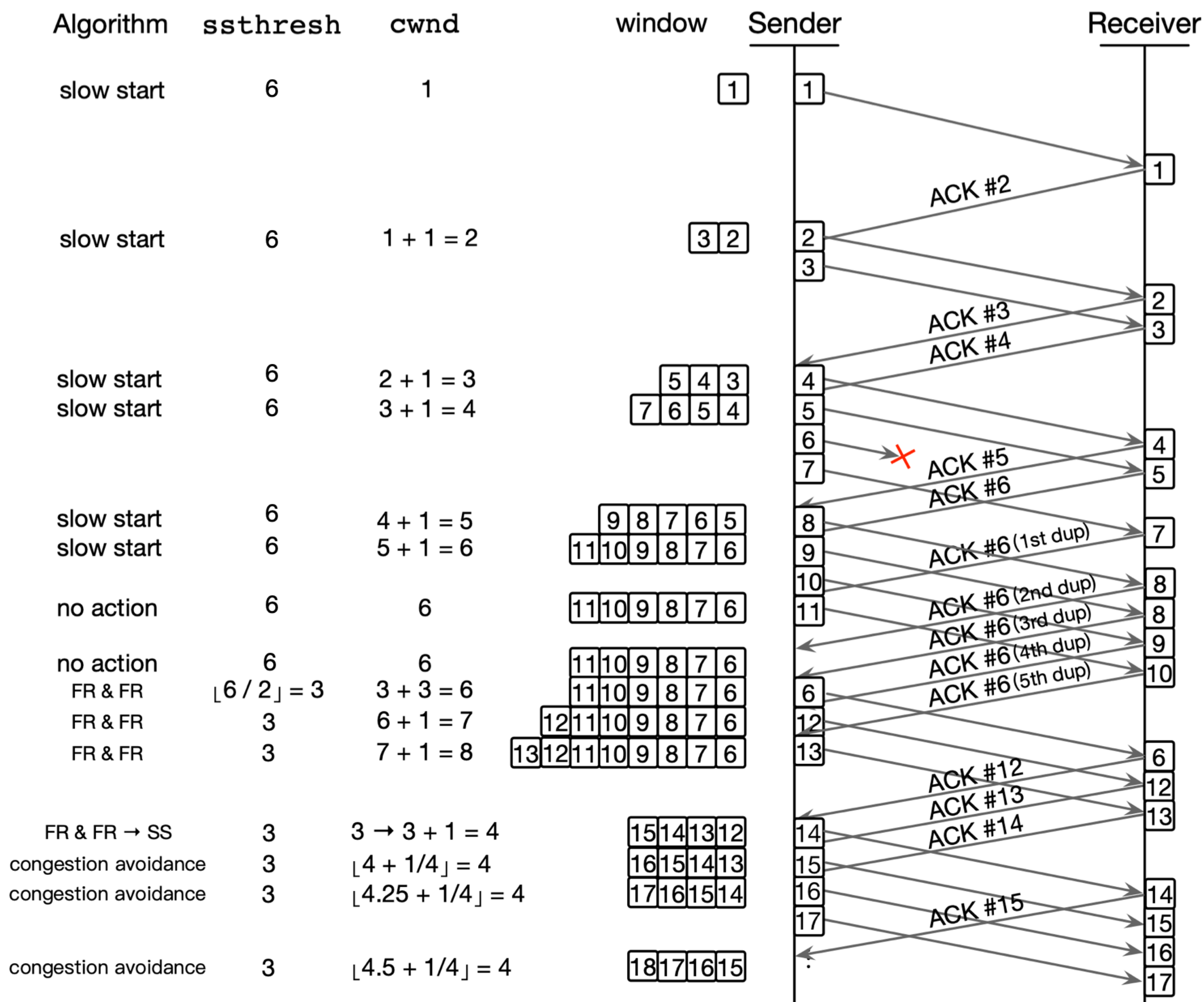
- Loss-based control systems (TCP congestion control): probe upward to saturated point, then try to backtrack to assumed underutilized state to let queues drain <- suboptimal
- Optimal control: at point of state change from under- to overutilized (before reaching saturation point)

Two approaches to congestion control

- *End-to-end*: no explicit feedback from network; hosts infer congestion from observed loss/delay
- *Network-assisted*: routers provide feedback to end hosts (via 1-bit flag within packet, indicating congestion)

Early Congestion Notification/**ECN**: ECN-capable hosts/routers set ECT (0 or 1 bits) in IP header; when router is overloaded: set 2 ECN bits to 11 (note: requires supporting router; ECT = ECN Capable Transport)

- Receiver: set "ECN-Echo" (ECE) flag in ACK packet to sender; sender: cut cwnd in half (congestion avoidance)



When ACK #15 is received, ssthresh = 3 and cwnd = 4

The Socket API

Relevant calls: *socket(), recv(socket file desc., receive buffer), send(socket file desc., send buffer, 0)*

- **Client**: [TCP] *connect(socket file desc., server address)*
- **Server**: *bind(file desc., server address)*, [TCP] *listen(file desc.)*, [TCP] *accept(file desc., &client address)*

Set non-blocking: *fcntl(file desc., F_SETFL, flags | O_NONBLOCK);*

<u>The Network Layer</u>

Routers: examines header fields of all IP datagrams passing through it, moves datagrams from input to output ports to transfer along path; are middle parts of end-to-end path; two functions: **routing** and **forwarding**

- **Routing**: Fill in router's forwarding table (FIB) with best path to every destination
- **Forwarding**: Use datagram dest. address as input to FIB lookup to determines best next hop

Routing & forwarding: **data plane** (forwarding: local function) vs **control plane** (routing: network-wide logic)

<u>IP</u>

IP: Stateless, provides datagram delivery source -> dest; fragments & reassembles transport layer packets as needed

IP datagram format: Header + options (if any) + data (variable length; typically a TCP/UDP segment)

IP header format: (20-60 bytes, depending on options)

- Version #, header length (4 bytes), type of service, total length (incl. payload - variable length)
- *Fragmentation/reassembly fields*: 16-bit identifier, flags, fragment offset (8 bytes)
- ***Time to live*** ("hop limit"): represents max # of remaining IP hops (guards against forwarding loops)
- Protocol (indicates upper-layer protocol to deliver data to; TCP vs UDP, e.g.) + IP header checksum
- Source, destination IP address

In-network fragmentation & reassembly: Each link layer protocol (e.g. WiFi, Bluetooth) has its own max transmission unit (max link-layer payload) -> if datagram is too large for next link's MTU, router fragments IP packet (chop up payload)

- Each fragment a separate IP datagram -> reassembled at destination host; receiving host sets timer on first fragment/segment -> if all fragments are received, pass to transport layer; otherwise, drop the packet
- Fragmentation details: Header contains packet identifier (same for all segments in IP packet), fragment bit, fragment offset (like TCP offset); recalculate IP header total length, checksum, offset when fragmenting

IP address: unique 32-bit identifier assoc. with each network (host/router) <u>interface</u> (one device may have multiple)

- Network interface: connection between host/router and physical link (can also be virtual: Docker)

IP address ranges: allocated hierarchically (ISPs/large sites from RegInterRegistries; customer subblocks from ISPs)

- Router FiBs map dest. address ranges (address blocks) to output links; each router interface is a subnet
- Addressing is hierarchical: ISP routers receive traffic for their address range, forward downstream

Classless Interdomain Routing (CIDR): divide IP addresses into (network portion) + ("free space" - suballocatable)

- Address format: a.b.c.d/x; x is # bits in network portion of address (varying lengths)
- Subnetting: allocate additional # of bits to a subnet organization -> new address: a.b.c.d/y

IP packet forwarding: router finds longest-matching prefix (binary, in FiB) for destination address, uses that entry

Subnets: device interfaces that can physically reach each other without passing through an intervening router

- Local network operator connects their subnets with a router; for global reachability: assign address range to each subnet + assign (to hosts) addresses within their subnet's address range (configure subnet mask)
- Reserved: 255.255.255.255/32 ("broadcast"); last address of the network (223.1.1.255 for 223.1.1.0/24) local network broadcast address; first address of network: network address; 0.0.0.0: "default route" (not in FiB)
- Subnets determined by physical connectivity; each subnet has address block, not all addr blocks have a subnet

DHCP

Host configuration: IP host must be configured with following information to be able to send/receive data:

- Data: IP addr. of an interface, subnet mask, default router's IP addr [HTTP: DNS caching resolver IP address(es)]
- Can be hard-coded by sysadmin in configuration file, or (today) obtained via DHCP ("auto config")

DHCP: New host sends DHCP discovery (each server must have one DHCP server/proxy)

- DHCP server(s) respond with DHCP offer(s) [w/ configured params]; client picks one, sends DHCP request
- DHCP server sends address (DHCP ack) to confirm offer, adds host to local database

DHCP implementation: Over UDP (since DHCP server address unknown)

- All messages (discover, offer, request, ack) sent to 255.255.255.255 [broadcast]
- Client's source address always 0.0.0.0 (in discover, request); server source address is a valid IP (server's)
- Network configuration is "leased" for a given time period; can be "renewed" (client resends request)

NAT

IPv4 addresses (32 bits) not enough to handle all possible IP addresses -> solution: use ***Network Address Translation***

Idea: treat port number as part of namespace (on top of IP addr.); associate private address space to a port number

NAT: network address & port translation (note: new socket->new port)

- Every host within a private network has its own private IP address (LAN address)
- NAT keeps **NAT translation table**: maps (LAN/private address, port #) to (WAN/public address, port # pair)
    - Preventing NAT translation table overflow: keep a timer on each table entry
- Local host sends packet to some destination -> NAT replaces packet source address (local address) with public address; all packets leaving private network (via router) are assigned the same source IP address
- Reply arrives with source addr + port # as dest. -> NAT replaces packet dest addr with LAN addr + port #

NAT problems: increased complexity; single point of failure (router); limited scalability due to port #, NAT address block limitations; cannot run services inside a NAT box; applications have to worry about NAT traversability

No services: NAT creates new entries based on outgoing traffic -> no public addr for outside servers until entry is made

- *Static port forwarding*: statically configure NAT to forward incoming connection requests at some port to server
- Alt. – *Universal Plug and Play Protocol (UPnP)*: Allow a host behind a NAT to learn public IP address, add/remove port mappings dynamically (with lease times) [only works on a single layer of NAT!]
- Alt. – *Application-layer relaying*: NATed client app establishes connection to public relay, external client connects to relay -> relay bridges packets between two endpoints [can use even under multiple NAT layers]

IP-in-IP (***IP tunnelling***): Encapsulate an IP datagram within a larger IP packet; outside IP packet deals with public addresses, internal IP datagram has private addresses (can also use for security)

- IP node receives packet, sees dest. addr is own addr -> removes header, forwards payload within own network

IPv6 packet format: address length: <u>32 -> 128 bits</u> (much larger address space)

- Header: Type of service -> traffic class, TTL -> hop limit, protocol -> next header, added flow label field
- Moved fragmentation, IP options out of base header; no checksum (moved to link layer)

IPv6 header format: Fixed-length 40-byte header (length field excludes header)

- Version (4 bits), priority (8 bits), flow label (20 bits)
- Payload length (1 byte), next header (1 byte), hop limit (1 byte)

- Source, destination addresses (16 byte ea.)
- (Options: outside of basic header, indicated by "next header" field -> no need for header length field)

Nowadays: Tunnel IPv6 packets through IPv4 networks; routers keep both IPv4, IPv6 FIB

- Router receives IPv6 packet, but IPv6 FIB points to IPv4 address -> encapsulate IPv6 datagram in IPv4 packet

The Application Layer

Two ways to structure network routing/control plane: (i) Per-router control (traditional): routers run routing protocol to set up forwarding table; (ii) (Logically) centralized control - more recent, use software-defined networking (SDN)

Route computation algorithm: Given a graph (***network graph abstraction***: connected set of nodes/routers, + edges/links with positive costs), find the least-cost path from a given node to all other nodes in the graph

- Algorithms: ***link-state*** (Djikstra, needs global knowledge) vs ***distance-vector*** (Bellman-Ford, local knowledge)

Routing protocols: Lay out how routers communicate, implement knowledge transfer within a network

- Define format of routing information exchanges + defines computation upon receiving routing updates
- Network topology changes over time -> need to continuously update all routers with latest changes

Routing Algorithms

Link-State (***Djikstra***): Given a complete network topology graph, each node computes the least-cost paths from itself to all other nodes; populates forwarding table with next hop of best path to each destination

- Is an iterative algorithm: after k iterations, a node knows the best paths to the k closest destinations
- Notation: D(N) [distance start -> node N], p(N) [node directly before N in the best path start -> N]
- Output is a tree rooted at start node; algorithm complexity: O(n^2) -> O(nlogn) [optimized]
- If link cost is dynamic, then need to keep recomputing routing -> may see oscillations; packets may be forwarded back & forth/looped -> propagation time increases, affects upstream layers (increases RTT)

Distance-Vector (***Bellman-Ford***): Each node only needs to know, from each direct neighbor, its list of distances to all destinations; each node computes its own shortest paths locally based on inputs from neighbors

- Bellman-Ford: dx(y) cost of least-cost path to y -> d_x(y) = min{c(x, v) + d_v(y)} [min over all neighbors v of x]
- Algorithm only needs to know distance to neighbors + neighbors' shortest paths (routing costs to destinations)

Distance-Vector Algorithm: Start node x initializes link costs to neighbors c(x, v)

- X maintains distance vector D_x = [d_x(y) : for each y in network N], sends D_x to all of its neighbors
- X receives D_v from each neighbor v, calculates new vector D_x'; if D'x(y) < Dx(y), update next hop to y
    - If D_x -> D_x' results a change, send an update to each of its neighbors + update own table
- If link costs change: check table for updates, send updates to neighbors if table changed

Distance-vector protocol: Local iterations caused by local link cost change or D_v update message from neighbor

- Each node notifies neighbors only when its DV changes; iterative, distributed
- Async.: nodes need not iterate at the same time, can propagate changes through network over time

Count-to-Infinity Problem: If a link is broken, non-neighboring nodes may report shortest-cost path relying on that node -> node on link increases cost based on other nodes' estimate, other nodes increments, node on link increments

- Continues indefinitely, costs go to infinity; meanwhile: packets will ping-pong in between (***routing loop***)

Count-to-Infinity: Due to each node having only local information

- ***Split horizon***: If a node B reaches d via C, B tells C nothing about node D
    - Issue: Cannot solve larger routing loops (not between neighbors); not effective in nonlinear topologies
- ***SH with poison reverse***: If a node A goes through C to reach D, tells C that its distance to D is infinite
    - Prevents routing loops involving only two routers, but 3+ can still have loops (*mutual deception*)
- ***Path-vector routing***: Each node announces its entire route/path to every destination (used in BGP)

In practice: Link-state within autonomous systems, distance-vector between autonomous systems

Link-state vs distance vector: compare (i) message overhead, (ii) time to convergence

- Distance-vector: Update messages can be large (linear with # destinations), but travels over one link only; each node only knows distances to other destinations; slow convergence, link breakage issues (Count-to-Inf.)
- Link-state: Each node broadcasts its distance to each neighbor to entire network -> small updates, but to more destinations; each node learns entire topology map; on link breakage, update topology -> nodes recompute

Routing protocols: Monitor link, neighbor nodes' statuses; send updates as needed to mitigate packet losses

- On link breakage: explicitly broadcast topology update (L.S.) vs distance vector updates (D.V.)

<u>OSPF</u>

Global Internet: an interconnection of a large number of ***autonomous systems*** (ASes)

- Each AS assigned a unique 4-byte autonomous system number/ASN
- Transit ASes: Internet service providers (hierarchy among ASes/ISPs: tier 1, 2, 3 ISPs, e.g.)
    - May also offer connectivity to user networks (not ASes)
- Stub AS: end user networks (e.g. campuses); may connect to multiple service providers (*multihoming*)

Internet routing: 2-level hierarchy (intra- and inter-AS)

- Intra-AS: within an AS -> *<u>intra-domain routing protocols</u>* [interior gateway protocols]: RIP, **OSPF**, e.g.
    - **Open Shortest Path First** (**OSPF**): intra-AS, link-state routing [RIP: distance-vector]
- Inter-AS: between ISPs, between stub & transit ASes -> *<u>inter-domain routing protocols</u>*: **only BGP**
    - **Border Gateway Protocol** (**BGP**): inter-AS, path-vector routing

Issue: what if a router in one AS receives a datagram with destination in a different AS?

- Inter-domain routing learns of destinations reachable through bordering ASes, propagates reachability to all routers in current AS; figure out best ***gateway router*** [connected to a router in a different AS] to use
- In router: forwarding table relies on both intra-, inter-AS routing algorithm information; jointly fill in each router's forwarding table: intra-AS for internal destinations, inter-AS + intra-AS for external destinations

Separation of intra-, inter-AS routing (reasons)

- Policy: (Inter-AS) admin wants control over how its traffic is routed (+ who passes through its network/AS)
- Scale: hierarchical routing saves table size -> reduced update traffic; performance

OSPF: Link-state; OSPF messages sent as IP packet (no specification of transport protocol) with protocol ID 89

- Each node sends a Hello to each neighbor periodically, monitors link + neighbor nodes' statuses
- Each node broadcasts ***link-state advertisement*** (routing update) to entire network; done periodically, or if the status of any neighbor/link changes (handling routing packet losses: via hop-by-hop ack)
- Every node broadcasts local piece of topology graph -> can combine to piece together a complete graph; link costs can be asymmetric (graph is directed; represent as a table)

Link-state routing: figures out how to reach the router which can reach the destination/prefix

- Each subnet is allocated a specific IP address block (= address prefix), connected to 1+ routers

Link-state protocol: need reliable flooding; via (i) sequence #, (ii) timer ack, (iii) unique ID: router ID + sequence number

- Router ID either manually configured, otherwise defaults to the highest IP address of the router

Link-state advertisement/LSA packet structure:

- LS age (TTL: LSA's lifetime), LS type, ID of the node that created the LSA (*link state ID*)
- Advertising router (could be same as link-state ID or different; depends on LS type)
- LS sequence number, LS checksum, length (sequence #: increase every time)
- *Content*: List of direct neighbors + link cost to each of them

OSPF: When neighboring routers discover each other or the first time: exchange link-state database

- Failure detection: Nodes periodically send Hello messages to neighbors; no hello message after certain period -> failure, send updated LSA; otherwise, send LSA at certain intervals

Link-state routing daemon: routing daemon running at each router; sends Hellos/LSA messages

- Upon receiving a new LSA: replay to all neighbors (same LSA contents in new OSPF datagram with new source, broadcast addresses), process LSA to update topology graph & recompute shortest paths
- Each router stores the most recent LSA from all others; decrement TTL of stored LSAs, discard when TTL=0

Reliable flooding: Each node replays received new LSA to all neighbors, except the sender; receive ACK from neighbor, otherwise retransmit the LSA (reliable delivery); use the link state ID + sequence # in an LSA to detect duplicates

- Router crashes & restarts: send request to ask neighbors to summarize database -> afterward: send requests to get LSAs from neighbors (use to restore sequence number)

Hierarchical OSPF (for large domains): Divide network into local areas, connected by larger top-level backbone area

- Internal areas run internal OSPF, backbone runs backbone OSPF -> can keep LSDB small
- ***Border gateway routers*** [area border routers; included in both backbone, local area]: summarize distances to destinations in own area, advertise to backbone; store distances to its area's internal routers + distances to all other area border routers, propagates area border router distances to its internal area's routers
- Local routers: flood & route LS in local area only; forward packets to outside via area border router
- Can add/remove areas easily; new area OSPF only affects current area, and only area border router communicates with backbone; other areas don't care until area border routers are told

<u>BGP</u>

Path-vector algorithm: Variation of distance-vector; announces whole network path to avoid loops

BGP: Only inter-domain routing protocol (between ASes); provides each AS a means to:

- Advertise its own IP address prefixes to the rest of the Internet
- Obtain IP address prefix reachability info from neighboring OSes and propagate the reachability info to all routers internal to the AS
    - An AS may choose not to allow other ASes to take a particular route through it
- Determine "good" routes to use for learned reachability to destination prefix and policy
- Propagate a proper set of of the externally learned routers to selected neighbors; an AS advertises routes to destination network prefixes [route = prefix + attributes], promise to forward packets to those prefixes

iBGP vs **eBGP**: BGP sessions between routers in same AS (***iBGP***) vs gateway routers in 2 different ASes (***eBGP***)

- Gateway routers use iBGP to distribute new prefix info to all routers in its AS; routers learn of new prefix, create corresponding entry in local forwarding table
- eBGP: Each side may advertise reachability to some prefix to the other

BGP: Between neighboring/connected routers, over TCP (port 179)

- OPEN message: opens TCP connection to remote BGP peer, authenticates sending BGP peer
- UPDATE: advertises a new path, or withdraws an old one
- KEEPALIVE: keeps connection alive in absence of UPDATEs + ACKs OPEN request
- NOTIFICATION: reports errors in received BGP updates; also used to close connection

BGP path attributes:

1. ***AS-PATH***: a list of ASes, through which prefix advertisement has passed
    a. For receiving router: indicates list of ASes that can be used as a path to that prefix
2. ***NEXT-HOP***: indicates specific internal AS [gateway] router that leads to next-hop AS
    a. Names outgoing interface of eBGP router; can be multiple links from one AS to a neighboring AS
    b. Next-hop changes between eBGP peers, but not iBGP; local preference injected while iBGP
3. ***Local-Preference***: policy preference in path selection
    a. Border routers inject local-preference into received BGP updates; used by internal routers in path selection (e.g. deciding which AS to pass through to reach a certain destination)

BGP routing policies: Import policy (which paths to keep vs drop?), route selection, export policy (which paths/destinations to advertise to neighbors?)

BGP route selection: decide best hop by local preference > shortest AS path > lowest internal cost > *other criteria*

- Local preference: manually configured value according to AS policies (note: paths may be asymmetric!)
- “Lowest internal group cost” (*hot potato routing*): route with shortest path within sender AS to gateway

Internet AS interconnects: hierarchical structure; tier 1 ISPs (all peers, full-mesh connected with each other) > regional ISPs (customers of tier 1, may be peers of each other) > customer stub networks (often multihomed)

BGP export policy in routing advertisements

- ***“No valley” routing policy***: a provider passes all prefixes to its customer ASes; a customer does not pass prefixes between providers
    - Provider announces customer prefixes to peers & provider, everything to customers
    - Do not announce {peers, provider} to peers or providers
- Once you go downhill (or horizontal) you can't go uphill; once you go downhill you can't go horizontal either
- Customer is “***multi-homed***” [attached to multiple provider networks]; does not advertise, to any provider, any route that it learned from a different provider (should not forward traffic from one provider to another)
- A provider only propagates customers’ routes to peers, passes all prefixes to its customer ASes; does not pass prefixes that are not its clients’ to other providers

Datagram delivery: unicast (announce IP block from some location), broadcast (different broadcast scopes), multicast (send to a group of recipients), anycast (announce IP block from multiple locations <- multiple potential paths)

BGP operates on trust (BGP hijacking: an AS can broadcast a certain prefix to re-route traffic)

The Link Layer

(**Data**) **Link layer**: below network layer, transfers packets between physically connected nodes (i.e. routers, hosts)

- Link layer addressing: via **MAC addresses** (***Medium Access Control***)
- Link types: simplex, half-duplex, full-duplex [*half-duplex*: multi-access links (e.g. Ethernet, WiFi)]
- Functions: data framing (marking start & end of a data chunk), error detection, channel access protocols

Link layer implemented in adaptor (***network interface card***/**NIC**) or on a chip; combines hardware/software/firmware

- Ethernet card: implements link & physical layer, e.g.; attached to host's system buses
- Communication: sender encapsulates IP packet in frame, adds error checking bits, follows access control protocol to send frame; receiver checks for errors; if ok, extracts datagram and passes to upper layer

Data frames: Link layer term for a block of data; contains a header field and data field (+ optional middle ***trailer field***)

Byte-oriented framing protocol: delineate data frames with a byte of special bit sequence 0111111110

- ***Byte stuffing***: sender adds/"stuffs" extra 01111110 byte after each appearance of 01111110 in data stream
- Receiver: If single 126, take as flag byte; otherwise, if multiple in a row, discard 1st and take rest in data stream

Error detection: Append EDC (*error detection and correction bits*) to data field

- Issue: not 100% reliable, may miss some errors; larger field -> better detection, correction
- ***Cyclic redundancy check*** (CRC): better mathematical means of error detection, widely used in practice

Multi-Access Protocols

Multiple-access links and protocols: sharing a single transmission medium can lead to collisions (i.e. >=2 parties speaking at intersecting times) -> ***collisions***: receivers cannot decode frames

- Multi-access protocols "coordinate" when a node can speak; hard vs soft coordination
- Ideally: Given a broadcast channel with a rate R bits/sec -> if M nodes can each send at rate R/M

3 classes of solutions: ***channel partitioning***, "***taking turns***", ***random-access protocols***

Channel partitioning: Divide channel into pieces (by time/frequency/code, e.g.)

- ***Time division multiple access***/**TDMA**: access to channel in "rounds"; each station gets a fixed-length slot (packet transmission time) in each round; unused slots go idle
- ***Frequency division multiple access***/**FDMA**: Channel spectrum divided into frequency bands, each station assigned a fixed frequency band

"**Taking turns**": on-demand channel allocation

- **Polling**: master node asks slave nodes to transmit in turn (concerns: overhead, latency, single point of failure)
- Alt. – **Token passing**: one token message is passed from one node to the next sequentially; whoever gets the token can send one data frame, then pass token to next node
    - Concerns: latency, single point of failure (token); master station generates token

Random access protocols: Lets a node transmit at full channel data rate R; no a-priori coordination among nodes

- Random-access protocols need to specify how to detect and recover from collisions
- **ALOHA**: If a node has data, send whole frame immediately; in case of collision, retransmit with probability p
    - Any other node tries to transmit while node is sending -> collision
    - Pure ALOHA frequency: at most 1/2e = 0.18
- **Slotted ALOHA**: Divides time into equal-size slots; a node transmits only at beginning of next slot

        - If no collision, node can send new frame in next slot; otherwise, if collision, retransmit in every subsequent slot with probability p until success; Assumes clocks in all nodes are synchronized
        - Slotted ALOHA efficiency: at most 1/e = 0.37
    - ***Carrier sense MA*****/CSMA**: listen before transmitting; if the channel is busy, wait until it is sensed to be idle
        - Idle behavior varies between different types of CSMA: Retry immediately (1-persistent), retry immediately with probability p (P-persistent), retry after a random interval (non-persistent)
        - Collisions still possible - chance of collision increases with distance between nodes
    - **CSMA/CD** (**Collision Detection**): compare transmitted with received signals; abort collided transmissions
        - Efficiency: Given Tprop = max propagation delay between any 2 nodes + Trans time to transmit a maximum-sized frame, **eff = 1/(1 + 5Tprop/Ttrans)**; eff -> 1 as Tprop->0, Trans->inf

(**Shared-Cable**) **Ethernet CSMA/CD algorithm**: NIC receives datagram from network layer, creates frame

- If NIC senses channel idle, starts transmission [1-persistent: if channel is busy, wait until idle, then transmit]
- If NIC transmits entire frame without detecting another transmission -> done
- If NIC detects another transmission while transmitting, abort and send jam signal for a short time period
- After aborting, NIC enters ***binary exponential backoff***: after mth collision, NIC chooses a value K at random from 0, 1, 2,..., 2^m - 1; NIC waits K slots, returns to step 2 (more collisions -> longer backoff intervals)
    - 1 slot = transmission time for 512 bits

Ethernet frame structure: preamble -> dest address -> source address -> type/length -> data -> CRC

- ***Preamble***: 8 bytes - 7 bytes with pattern 10101010, followed by 1 byte with pattern 10101011
    - Used to synchronize receiver, sender clock rates
- ***(MAC) addresses***: 6 bytes each; if received frame destination address matches NIC address, or is broadcast address, adaptor passes data to network layer protocol (otherwise, discards frame))
- ***Type***: 2 bytes, indicates higher-layer protocol (more recently: changed to "length", type field -> data part)
- ***Data***: 46-1500 bytes; encapsulates IP datagram, among other contents
- ***CRC***: 4 bytes, used for error detection; added by sender, checked by receiver (if error, drop frame)

Medium access control (MAC) addresses:

- Ethernet, WiFi: 48-bit MAC addresses; unique MAC address for each interface on LAN
- Hard-coded into adaptor, typically; software-settable in some cases
    - Blocks: assigned to vendors (e.g. Apple) by IEEE -> adaptors: assigned by vendor from its block
- Special addresses: *broadcast address* (all 1s), *group addresses*: 01-80-C2-00-00-00 to 01-80-C2-FF-FF-FF

IEEE controls MAC address allocation (analogy: MAC address is SSN; IP address is postal address):

- Adaptor manufacturers buy MAC address blocks from IEEE (ensures uniqueness)
- MAC address is flat -> portability; LAN (local area network) card can move from one LAN to another

Wireless channel characteristics/challenges: Decreased signal strength over time, interference signals from other sources, multipath propagation (bounces, e.g., may cause multiple arrival times)

- Other problems: ***Hidden terminal*** (two nodes A, C see a node B, but not each other; try to send simultaneously -> collision), ***signal attenuation*** (A, C see but cannot hear each other due to interference at an intervening B)

Ethernet switches (alternative to shared-cable): link-layer device, behaves as a host on each connected LAN

- Speaks Ethernet protocol at each interface; does not understand IP datagrams
- Transparent: hosts are unaware of presence of switches
- ***Store-and-forward***: examine each incoming frame's MAC address, forward to dest LAN if dest host is on a different LAN (can forward two frames with different output interfaces simultaneously)
    - Buffers frames temporarily if next LAN is busy

Switch keeps <u>forwarding table</u>: given a destination MAC address, specifies which specific interface to forward to
- Previously: routers know global topology via OSPF; here: switch does not have knowledge of topology
- Forwarding table entries: MAC address of host, interface to reach that MAC address, TTL

Can build a forwarding table via ***self-learning***; upon receiving a data frame:
- Look at source MAC address; if matching table entry found for source MAC in forwarding table, reset TTL; otherwise, add source MAC to forwarding table
- Look at destination MAC address;
    - If matching table entry found: if frame arrives on interface to be used for forwarding, then drop frame; otherwise, forward frame to interface indicated by entry
    - Otherwise, flood: forward to all other interfaces besides incoming (with MAC addresses unchanged)
- Interconnecting switches: self-learning to learn how to forward

Routers vs switches: both store-and-forward
- Router fills forwarding table via routing protocols; switches, via self-learning
- Router (network layer) vs switch (link layer); at link layer, router looks like any other host

Switches vs hubs
- Hubs: pure amplification; does not understand link layer, physical layer only; upon receiving a frame, will broadcast it to all other interfaces simultaneously; simulates other interfaces sharing a bus
- Switches: buffers frame, then forwards (store-and-forward); does not broadcast/flood unless dest MAC not found; utilizes buffering -> can connect to LANs of different speed

Switches: advantages & limitations
- Advantages: transparent (no change to hosts); isolates collision domains -> higher total maximum throughput; store-and-forward/buffering -> can connect Ethernets of different speeds; no configuration needed
- Limitations: constrained topology (can only work in tree structure); all inter-segment traffic concentrated on a single tree; all multicast traffic forwarded to all LANs

Routers: Support arbitrary topologies
- Issues: requires IP address configuration; more complex packet processing than switches
    - Advanced settings: need to worry about who OSPF is talking to

<u>LANs</u>

MAC addresses: 48 bits
- Used locally to get frame from one interface to another physically-connected interface [same subnet]
- Burned into NIC (for most LANs); can sometimes overwrite via software
- One machine may have multiple MAC addresses (NICs); routers: one MAC addr for each subnet contact

Connecting IP, link layers

- Node A sends IP packet to B -> A looks up IP address, compares subnet mask of source, destination IP addresses; if same, A and B are on the same subnet; otherwise, must communicate via router
    - A, B on same subnet -> A sends to B via link layer protocol; puts IP packet within link layer frame
- Given B's IP address, how to find B's MAC address? -> Keep a routing table IP->MAC
    - Lookup MAC for destination IP or router's IP, depending on whether packet can be sent directly; use to send link layer frame
    - Router/node receives frame; removes Ethernet header, finds IP dest address: if IP is self, deliver to transport; if IP is not self and node is router, repeat previous steps

Address Resolution Protocol (**ARP**): Every IP node (host & router) on LAN runs ARP to build an ARP table

- Table entry stores IP address, MAC address, entry TTL (every time an entry is looked up, reset TTL value)
    - Soft-state design: information deletes itself after a certain time unless it is refresh

ARP discovery: Initially: A wants to send datagram to B, has no ARP table entry -> host A broadcasts ARP request containing B's IP address (source MAC address: its own; destination MAC address: all 1s)

- All nodes on LAN receive ARP query, add A's info to their ARP table; B replies to A with ARP response (unicast, direct B->A; MAC address B->A); A receives B's reply, adds B's entry into its ARP table
    - B's reply: nodes passed-through along the way add B's entry to their ARP table
    - Hubs: hub will broadcast unicast to all connected interface -> all connected interfaces will learn

Routing to another subnet: **addressing**

- Assume: A knows B's IP address, IP + MAC address of 1st hop router R; IP configuration (4 components: IP address, subnet mask, DNS resolver) hardcoded
    - May need to use ARP discovery beforehand to obtain MAC address of router R
- A creates IP datagram with IP source A, destination B; creates link-layer frame
    - Link layer destination: R's MAC address (uses ARP to obtain router's MAC address)
- Router determines outgoing interface, passes datagram with IP source A, dest B to link layer
    - Creates link-layer frame containing A-to-B IP datagram; frame dest address: B's MAC address
    - Transmits link-layer frame to other subnet (note: this is a different datagram than sent by A!)