Stanley Wei
Yu - W25

CS 118: Computer Network Fundamentals

Contents

The Internet

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
    - Fiber, copper, radio, satellite, etc.
    - Has associated transmission rate/bandwidth (BW) - bits/sec
- ***Internet Service Providers*** (**ISP**)
    - Have regional & global ISPs; local networks have routers connected to regional ISP routers

The Internet is a "network of networks"

Internet protocols define how to send & receive packets (ex: HTTP, TCP, IP)

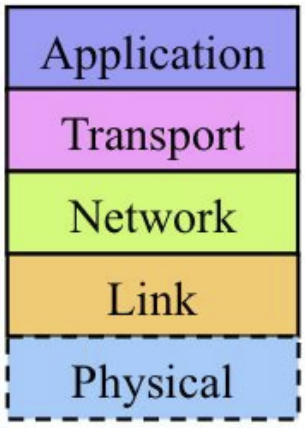
- Dictate message formats, order of messages sent/received, and actions taken on packet transmission & receipt
- Internet protocol standards: from RFCs, IEEE standards, W3C, etc.

Internet built on multiple layers of protocols (called the ***Internet protocol stack***)

1. **Application layer**: Support data exchange between application processes
    a. Ex: SMTP, HTTP, DNS
2. **Transport layer**: Handle delivery of packets (esp. ensuring reliability)
    a. Also handles ***multiplexing*** within a host (keeping up multiple "conversations" within a single Internet access point)
    b. Ex: TCP, UDP
3. **Network layer**: Define how formatted packets are forwarded from source to destination
    a. Outline which jumps are taken at every step
    b. Ex: IP
4. **Link layer**: Define how data is transferred between directly connected network elements (e.g. host → router)
    a. Ex: Ethernet, WiFi
5. *Physical layer*: Physical movement of data/information

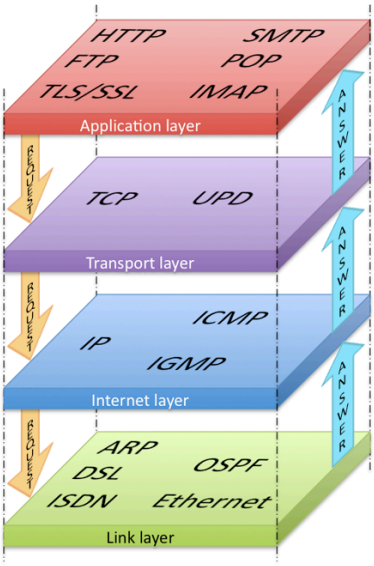
Protocol "layers": each layer encapsulates the previous layer's message/packet within a larger packet

- Application data only looks at the application header; each subsequent layer adds a new header on top of previous layer
    - Headers added top → bottom (application → link layer)
- During delivery: headers are removed in reverse order, bottom → top
    - Each layer only looks at its own header, then passes the rest of the packet to the next layer



Example: For online browsing:

1. **Application layer**: HTTP handles data transfer from web server to local host
    a. Assumes network can send data to any hosts on Internet; ignores how data is sent, assumes data is sent successfully
2. **Transport layer**: TCP/UDP deliver application layer packets to & from client, server
3. **Network layer**: IP forwards packets from source to destination
    a. Data may need to travel between multiple different routers (e.g. WiFi → campus router → local ISP → regional ISP → web server)
4. **Link & physical layers**: Controls physical movement of data/information



Computer Networks

Routers handle **packet switching** ("*statistical multiplexing*"): Dividing larger chunks of data into smaller packets that can be sent

- Via ***store-and-forward***: packet switch temporarily buffers up packets, forwards to host once full packet is received
    - Senders send packets as soon as link is available; receiver stores and forwards
    - Cons: results in some amount of delay; packets may be dropped when queue is full
- *Packet-switching* (at each jump: take best path) vs *circuit switching* (predefined path, reserved resources)

Measurements of Network Performance

- **Throughput** (*bits/sec = bps*): Measures the bandwidth of a single link, point-to-point
    - Is more difficult to measure in multi-access scenarios (multiple devices connected to same router, competing for bandwidth)
- **Packet losses**
    - Wired links: Packet loss due to transmission errors, congestion
    - Wireless: Higher bit error rate (compared to wired)
        - Have to contend with: limited transmission rate, host mobility (high variance in number of hosts sharing same wireless channel)

4 sources of network delay:

1. *Node processing*: need to check bit errors, determine output link
    a. Generally ignored, negligible
2. **Transmission delay**: Packet length / rate (link bandwidth)
    a. Represents amount of time needed to "push" the entire packet into a link
    b. *Recall*: Due to store-and-forward, the entire packet must arrive at a node before it can start moving to the next link
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
    a. Can only upload one packet at a time

Throughput: rate (bits/sec, e.g.) at which bits are transferred sender → receiver

- Instantaneous (rate at a given point in time) vs average throughput (over a longer period)
- Throughput throughput multiple connections bottlenecked by least-throughput connection/pipe

Utilization of a link: % of time spent transmitting

- Formula (Stop-and-Wait): (transmission time) / (transmission time + 1 RTT)

The Application Layer

Client-server application communication

- Socket API - use port numbers to let client process identify server process with which to communicate
- Socket: analogous to a door
    - Can tie (bind) socket to a pair (local IP address, port number)
- Other aspects
    - Client uses website name to find server-side IP address via DNS
    - Port number either assigned by OS (client), or via defined standards (server)
        - Ex: web has standard port numbers by service: web requests sent to port 80, FTP requests to port 21, and email requests to port 25

- Building an Internet application
    - Application process - create application process at two hosts/end systems
        - Each process creates Internet socket via socket API
    - Need to select transport service for socket: UDP vs TCP, commonly
        - TCP: connection-oriented (requires setup), reliable transfer + rate control
            - Rate control: flow control & congestion control (throttle sender when receiver is overloaded)
            - No security
        - UDP: connectionless, but unreliable & no rate control of messages
        - TCP vs UDP
            - Some apps (e.g. file transfer) require 100% reliability; others (e.g. audio/video) can tolerate some loss
            - Some apps (e.g. games) require low delay to be effective
            - Some apps (e.g. multimedia) require a minimum amount of throughput; other apps (“elastic apps”) do not
            - Security: Neither TCP nor UDP provides security
    - Define application-layer protocol
        - Types of messages exchanged (e.g. request, response), message syntax, message semantics, rules for when & how to send & respond to messages
        - Open protocols: public (defined in RFCs) & interoperable; e.g. HTTP, SMTP

        - Proprietary protocols (e.g. Skype)

Web page - consists of base HTML file + several referenced objects

- Objects: OtherHTML file, media, etc.
- Objects addressable by Universal Resource Locator (URL); application process specifies how to access

HTTP

HTTP: Web application-layer protocol

- Uses a ***client/server model***: browser requests/receives/displays Web objects; Web server sends objects in response to requests
- HTTP runs over TCP
    - Client creates socket (TCP connection) to server (default: port 80), uses to communicate
- Is "**stateless**" - server maintains no information about past requests
    - Stateful protocols - more complex, need to handle state synchronization between client & server (e.g.)

Protocol versions: HTTP/1.0 and HTTP/1.1

- HTTP/1.0: no persistent connection
    - 3 types of request: GET, POST, HEAD
- HTTP/1.1: persistent connection
    - Several additional forms of request

HTTP Versions

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

- To improve, can open multiple TCP connections in parallel
- Total delay: 2 * # objects RTT
    - 1 RTT for establishing connection; 1 RTT for data transfer

(***Persistent***) **HTTP/1.1**: Use a single TCP connection to send multiple objects

- Use KeepAlive field in header to ask server to stay connected
- Delay: (# objects + 1) RTT
    - +1: RTT for establishing connection
- Additional improvement:
    - Previously: client issues next request after previous response has been received
        - Total delay: 1 RTT per object
    - **Pipelining**: client sends requests as soon as it sees a referenced object
        - Total delay: 1 RTT (establish connection) + 1 RTT (retrieve index file) + data transfer time

HTTP/2 - solves performance issues in HTTP/1.1

- Modern Web applications generally composed of lots of small objects
    - 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)
        - HTTP/1.1 workaround: open multiple parallel connections
    - HTTP/1.1 has large ASCII header with repetitive information; has to send in full with every query
- Binary encoding instead of ASCII for header
- Uses single TCP connection between browser & server: each HTTP request a stream, multiplexed streams in priority order
    - Header compression - within a TCP connection, after sending a request, a subsequent request will only send elements of the header that have changed (all others: assumed unchanged)
        - Server, browser keep a header table until connection closes
- Request-response pairs encoded in a stream (stream = virtual channel; carries frames in both directions)
    - Frame: basic unit of communication; distinguish header & data frames
    - Message: HTTP request or response; may be encoded in 1 or multiple frames
- Streams have different priority, can interleave/shuffle by dividing into frames
    - Can transmit smaller objects before larger ones
    - Priority determined by clients (e.g. load stylesheets before media elements)
    - If one
- Server push
    - Motivation: In HTTP/1.1, have to send one request for each object
        - Multiple objects → multiple requests (frames)
    - HTTP/2: 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
        - Associated assets sent automatically in subsequent responses without needing to be requested by the client
        - If client moves to a different page, client can know to stop sending assets from previous page
- Limitation: HTTP/2 only mitigates HOL blocking at HTTP layer

    - Uses vanilla TCP → a single client-server TCP connection may still experience HOL blocking (if TCP packet losses occur & TCP starts packet recovery e.g.)

HTTP/2 → HTTP/3

- HTTP: no security over TCP, recovering from packet losses stalls object transmissions (causes HOL blocking)
- Adds security, per-object error and congestion control (more pipelining) over UDP
    - Switching TCP to UDP solves HOL blocking within a single connection

HTTP/2

- Header compression: while browser & server are connected, only contain the header elements of a request that have been changed from previous request
    - Ex: if previous & current requests were both GET, don't send GET portion of new/current request
    - Browser, server keep header table until TCP connection closes
- Binary encoding
- Server push: if a single object (e.g. "home.html") refers to a number of other objects (e.g. media elements), server will automatically send those objects
    - HTTP/1.1: client has to request those objects itself; HTTP/2: done server-side
    - Server first returns main object + promises regarding other objects; later returns responses with referred objects

(*) Caching

Scaling web services - popular websites receive large amounts of requests; may exceed capacity of a single web server

- Can cache popular contents & serve user requests from cache
- Configure each browser to send web requests to a proxy server (cache)
    - Cache: if a requested object is in cache, returns the object; otherwise, cache fetches the object from the server, returns to client, & saves a copy
- Alt: add a local cache on local area network (LAN)
    - Modern web: local cache hits tend to be small, reusable components (e.g. JavaScript libraries)
    - Reduces utilization of access link to origin server → reduces delay

    - Complications: refreshing stale cache contents, secure HTTP (HTTPS) connections (encrypted contents → caching no longer works)
        - Stale cache: HTTP conditional GET to reload cache copies based on time since last refresh
        - HTTPS - caching done by content distribution networks/CDNs (ex: CloudFlare, Fastly), rather than local ISP
            - Browser connects to CDN server via HTTPS; websites pay CDN providers, share crypto keys with them

Cookies

test

HTTP GET/response is stateless, but user/server want to be able to maintain state (beyond the current connection) (to recover from & complete multi-step exchanges across multiple connections, e.g.)

→ websites, client browser use cookies to maintain state between sessions

- Cookie initially issued by server
    - Cookie is first placed in a response header; client can include that cookie in request headers later to use that cookie
        - Ex: for user identification, without having to login on subsequent visits
    - Server can specify additional parameters for a cookie (e.g. maximum lifetime)
- Cookie: client keeps cookie file stored, associates a website/domain with a value
    - Client includes value in request headers to that website
        - Server can change behavir based on cookie value
    - Each cookie: a key-value pair
- Cookie: usefulnss vs privacy
    - Usefulness: can use for convenience, make recommendations
    - Privacy: websites can use cookies to learn a user's online behavior
- 3rd-party cookies: advertisers can use to obtain user info across multiple sites
    - When a page loads an advertisement (linking to an external site - advertiser's website), advertisement website returns a cookie with advertisement response

    - On future instances of seeing that advertisement website (same domain): advertiser will receive cookie, can use to track user

HTTP/2

- User, server state maintained via cookies
    - Cookies: key-value pairs
    - Recall: HTTP GET/response interaction is stateless
    - Delete cookies → lose user state

- HTTPS: Developed to provide security to transactions via encryption
    - With vanilla TCP, cookies can be intercepted

The Domain Name System (DNS)

Motivation: In application layer (e.g. HTTP), need to identify processes

- HTTP: Use IP address & port number
- Issue: IP address + port number not user-friendly for humans

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

- DNS protocol: uses query-reply pattern (name query → IP address response), like HTTP
- Is an application layer protocol
    - May use with either TCP or UDP on transport layer; typically, TCP (more reliable)
        - May also be used over UDP for speed and simplicity; most DNS servers support both protocols
        - DNS queries/replies typically fairly small compared to most packets

Domain names: hierarchical, separated by dots

- A node's domain name identifies its position in DNS **name space**
    - No theoretical length limit, but limited to 255 characters in practice
- Hierarchy: (root; .) → edu → ucla → cs (cs.ucla.edu), e.g.
    - Read left-to-right, but parsed right-to-left on management side
    - Note: DNS namespace completely independent from topological network connectivity, purely semantic

Scaling up name-address lookup: 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 (e.g. edu → ucla.edu)
        - Lowest level server (cs.ucla.edu) will provide the true mapping from name to IP address
    - Have separate IPv4 vs IPv6 name servers
- Making a query

    - On network: have caching resolvers/local DNS servers, acting as both client and server
        - Hosts send domain name queries to caching resolver; caching resolver sends queries to DNS server
            - Historically: host talks to caching resolver via stub resolver
        - Caching resolver caches results, sends back to hosts
    - DNS query protocol used by local DNS servers to query authoritative servers
        - Modern-day: have some public cache resolvers

DNS namespace governance: ICANN organization manages root name servers

- Also assigns and delegates top-level domains/TLDs (e.g. ".com")
- 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
    - Top-level domains: have generic TLDs (e.g. ".com", ".org") and country code TLDs managed by countries (e.g. ".us", ".kr")

DNS name servers

- Authoritative servers: often have multiple authoritative name servers for a single domain
    - Ideally placed in different networks (for redundancy)
- Root name servers: 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 contracts registrars (e.g. GoDaddy [US], CoolOcean [India]) to sell domain names to registrants

- Many registrars; recently: cloud providers
- Registrars update DNS registry (organization managing DNS namespace; ex: Public Interest Registry); registry updates internal domain database; domain database pushes change to other domain name servers
    - Registrars submits change requests to registry on behalf of registrant

All DNS data stored in domain database as resource records/RRs

- RR: contains 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
    - Stub resolver: configured with IP address of caching resolver(s)
    - 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

The DNS Protocol

- dig - tool for exploring DNS

Namespace allocation vs delegation

- Allocation: purely conceptual, only affects namespace
- Delegation: concerns operation responsibility of a subdomain
    - Creates a zone (administration unit)
    - Subdomain zones can be administered by same entity as, or independently of, a parent → more scalable
- Domain (from allocation, determined by namespace structure) vs zone (from delegation, determined by administration)
    - Administration hierarchy need not perfectly reflect namespace hierarchy

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

- Message header: identification (16 bit #) + various flags
    - Flags: query or reply, recursion desired/available, reply is authoritative
- Body: questions + answers (as RRs)

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 offer widely-distributed caching services; results in lower latency/faster loading for users
    - Content owners pay for caching service
- Process:
    - User queries DNS to get web server IP, sets up TCP connection to it
    - Web server outsources its DNS record to CDN provider (CDN network server) via CNAME (server domain → CDN IP); user HTTP requests go to CDN server
        - With HTTPS: web server shares crypto key with CDN provider

CDN: commonly use 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

Transport vs network layer

- Network layer - delivers packets source → destination via hops between hosts
- Transport layer - logical pipe between adjacent hosts

UDP

TCP and UDP

- TCP - connection-oriented
    - Reliable, delivers byte streams (in order)
    - Has other functions: congestion & flow control, e.g.
- UDP: only provides multiplexing/demultiplexing
    - Unreliable, delivers datagrams

Multiplexing & demultiplexing

- Multiplexing: sender gathers data from multiple application processes, sends them as a whole under a single message (with associated header)
- Demultiplexing: receiver delivers received segments to correct app layer process

Demultiplexing - host receives IP packet

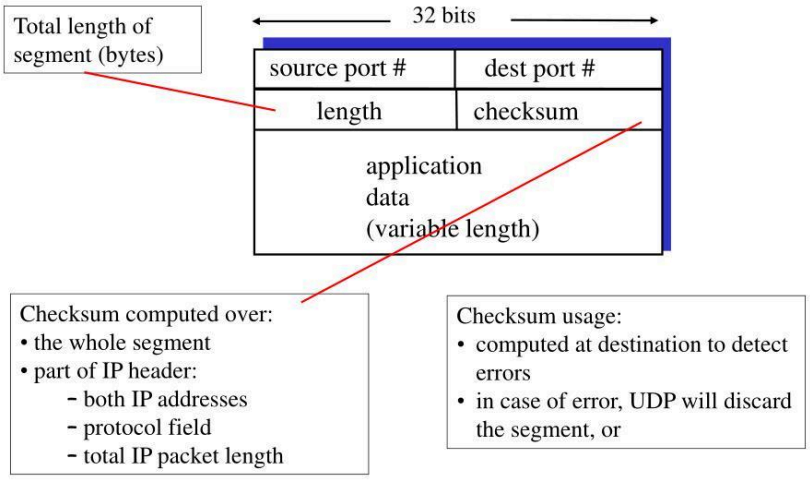
- IP packet contains transport layer data segment + application data
    - Data segment contains source, destination IP addresses
- Connectionless (UDP): when sending a packet, specifies source, destination address
- Connection-oriented (TCP): server creates separate sockets for each client
    - TCP sockets identified by 4-tuple of source IP address + port number, and destination IP + port number
        - Host uses 4-tuple (stored in segment) to direct to appropriate socket
    - Server creates “file descriptor” for each socket

UDP

- Unreliable - segment may be lost/duplicated/delievered out of order
    - For reliable transfer, can add reliability at application layer
- Connectionless - no handshaking between sender & receiver; each UDP segment handled independently of others
- UDP usages: DNS, streaming, general loss-tolerant & rate-sensitive applications
- Very small header:
    - Source & dest port; length + checksum
        - Checksum computed over pseudo header + UDP header, data

    - Pseudo-header: helps reliable data transfer
- No reliability guarantees + no congestion control

UDP segment structure



Principles of Reliable Data Transfer

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

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

Stop-and-Wait with NACK: 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) retransmission: sender sends up to N unacknowledged packets

- N: flow control window size, parameter (exchanged in initial handshake)
- 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

Selective repeat: sender sends up to N unacked packets

- Receiver acknowledges each correctly received packet; acknowledges & buffers out-of-order packets
    - Can release out-of-order packets when missing packets are received
- Sender maintains timer for first unack'ed packet; when timer expires, retransmits only that single unack'ed packet
- Flow control window: as before

TCP

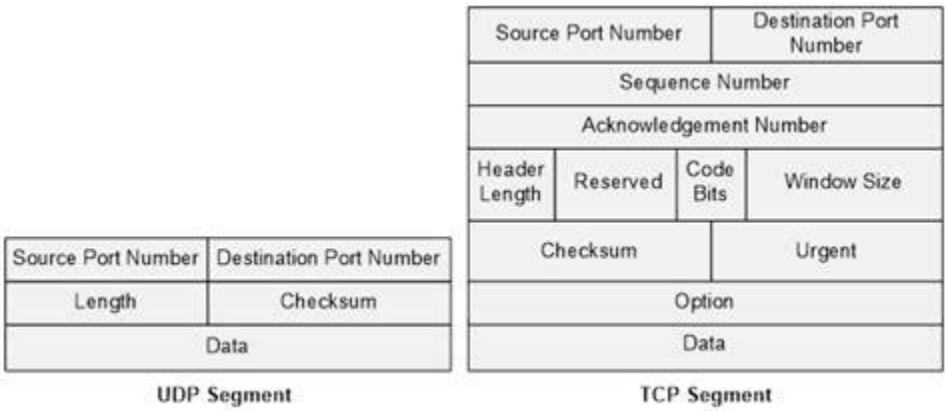
TCP: one timer, selective repeat, cumulative ACK

- Point-to-point: creates virtual pipe between 2 processes
    - Returns a socket (as 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
    - No message boundaries enforced by TCP within byte stream; framing 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 transmission speed to avoid overloading pipe

TCP segment format

- Same as UDP: source + dest port #, checksum; at end: application data
    - Checksum - includes TCP header + pseudo header
        - Pseudo header: source + dest IP, zero, protocol, TCP segment length
- Sequence number, acknowledgment number (count # of bytes)
    - Sequence number: represents sequence # of first byte in payload
        - Gives offset within current byte stream; have to break up a byte stream into multiple messages → sequence # encodes offset
    - Ack number: next expected byte to receive
- Receiver window size: represents buffer capacity of receiver
    - Dynamically adjustable, even after connection has been established
- Various flags; e.g. ACK, SYN, FIN, Reset
    - Connection management: SYN/setup, FIN/finish, Reset
        - Reset: represents connection reset/refused (e.g. bad authentication)
- Option fields [variable length]: used heavily
- (No longer used: pointer to urgent data [for slow computers])
- Note: no details w.r.t. congestion control

-



TCP logic: sequence & acknowledgment numbers

- Store 2 “sequence numbers”: 1 for byte stream A→B, 1 for byte stream B→A
    - A sends A→B byte stream offset as sequence number in its messages; returns next expected offset for B→A in its acknowledgments
    - 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 uses cumulative ACK

TCP connection management

- Set up connection (handshake): each end informs other of its initial data byte sequence number value
    - Use to set initial sequence number
- Close connection: each end informs other of its final data byte sequence number value
- Connection is aborted upon receiving an RST segment
    - Cases for sending an RST:
        - Receiving TCP segment of unknown connection
        - TCP retransmission count hits upper-bound
        - Need to reject a new connection request or close an existing TCP connection due to resource limitations

TCP connection setup: Via 3-way handshake:

1. Client sends TCP SYN segment to server [via connect()]
    - SYN flag - 1, no data
    - Specifies client's initial seq # (selected randomly)
2. Server receives SYN [via listen()], replies with ACK & SYN control segment
    - SYN, ACK flags are set to 1; ACK # is set as received sequence # +1
    - Server specifies its own initial sequence # (selected randomly)
    - Upon receiving: 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: server sees connection established

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

1. Host A sends TCP FIN control segment to Host B [via close()]
    - FIN flag set to 1; no data
2. Host B receives FIN, replies with ACK
    - Regular AACK, ACK A's FIN
3. When B finishes sending all data and is ready to close, it sends a FIN segment [via close()]]
4. A receives FIN, replies with ACK
    a. B receives ACK → B closes its connection
    - Note: host A never knows if host B receives ACK; simply closes its own connection after "long enough" without receiving retransmitted FIN
        - "Long enough": 2x max segment lifetime, e.g.

Connection reset

- Motivation: Maintaining a TCP connection requires resources
- Upon TCP connection setup request: system sets up TCP control block (TCB) to track TCP connection state
    - Identified by source + destination addresses, source + destination ports
    - 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

Boundary between TCP options and data

- Data offset field in segment: 4-bit header length
    - Data offset: where payload data starts
- Options: up to 40 bytes
    - Header length of 8 → option length of 40 byte
- Have blank field between data offset and flags
- Flags: ACK, SYN/FIN/RST; also: U/urgent, P/push (no longer actively used)

Flow control window

- Flow control: prevent sender from overrunning receiver by tranmsmitting too much dat too fast
- Receiver: informs sender of amount of free buffer space
    - Carried in RcvWindow field of TCP header with every arriving segment [sent by receiver]; can change 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
    - When timer expires, retransmit starting from S
- How many segments to retransmit? Depends on control windows
    - Receiver flow control window, rwnd
    - Congestion control window, cwnd
    - Number that can be retransmitted: min[cwnd, rwnd]
- Setting up retransmission timer
    - Don't want to timeout too early (causes unneeded resending) or too late - need some strategy to determine when to set timer

        - TCP sets timer based on estimated RTT plus a safety margin DevRTT, based on parameters alpha, beta
            - SampleRTT: based on historic time gap between most recent ACKed message send and ACK
            - SRTT: estimated “smoothed” RTT
                - SRTT = (1-alpha) * SRTT + alpha * SampleRTT
                - Initiailized at SampleRTT
            - DevRTT: estimated RTT deviation
                - DevRTT = (1-beta) * DevRTT + beta * |SRTT - SampleRTT|
            - RTO: retransmission timeout
                - RTO = SRTT + 4 * DevRTT
- Setting initial RTO: manually configured, based on guesses (in practice)
    - Current practice: initial RTO = 1 sec
- Q: what to do in case of retransmissions?
    - Taking measurements: delay between first transmission & final ACK, or last retransmission and final ACK?
    - Can’t just not measure
    - 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 data transmission (i.e. that didn’t require retransmission)

TCP: 3 duplicate ACK

TCP fast retransmit

- By default: RTO set to relatively long value
- Can detect lost segments via duplicate ACKs
    - Segment lost → next arrival at receiver OOO; when segment arrives OOO, receiver can immediately send ACK indicating seq # of next byte it is expecting
        - Segment lost → ACK # will be at same seq # [of lost segment] for several messages in a row
- Sender receives 3 duplicate for same sequence # → assume segment with seq # was lost
    - → Fast retransmit, start retransmitting without waiting for timer to expire

        - Only retransmit the one segment

TCP: delayed ACK

- TCP connection carries traffic in both directions → ACKs are piggybacked on data segments
    - For one-way data flow: receiver sends ACK after receiving every segment → double 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 ACK for both within a single packet; otherwise, send ACK for S1
- Issue: causes a little delay in RTT calculation for RTO
    - Upon OOO arrival, immediately send ACK indiciatign seq # of next expected back
    - Upon arrival of segment that partially or completely fills a gap, immediately send ACK if segment starts at lower end of the gap

Principles of Congestion Control

Types of Congestion

- Network congestion: Network achieves maximum possible throughput -> long delays (potentially unbounded queuing delay)
    - 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
    - Known loss case: sender only retransmits if a packet is known to be lost
    - Duplicates: sender may time out prematurely and retransmit (if ACKs are dropped, e.g.) -> duplicates are delivered
- Unneeded/superfluous retransmissions: when multiple copies of same packets go through overloaded links, reducing effective throughput
    - Packet dropped -> any "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
    - Heavier congestion -> more retransmissions

Congestion control: necessary to avoid congestio ncollapse

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 self automatically
    - More advanced schemes: also estimate via other variables (e.g. delays, delay changes)
- Sender limits: LastByteSent-LastByteAcked <= cwnd
- Adjust cwnd size based on network traffic load - infer by observed packet losses
    - Designed in two phases:
        - Slow start: set cwnd size to 1 segment initially; start slow, but rapidly increase cwnd

        - Congestion avoidance: slowly but continuously increase cwnd size
    - Use slow-start threshold/ssthresh to define boundary between 2 phases
        - Cwnd < sstresh -> slow-start phase, increase cwnd quickly
        - Cwnd >= sstresh -> in congestion avoidance phase, increase cwnd by 1 segment/RTT

TCP slow start - want to gauge pipeline size quickly

- Set cwnd = 1 MSS (max segment size, in bytes)
- Send cwnd-allowed segments
- If receive an ACK:
    - Cwnd = cwnd + 1
- Eventually: if a packet times out (indicating network congestion)
    - Set ssthresh = cwnd / 2
    - Set cwnd = 1 MSS & return to step

Default: initialize ssthresh to flow control window size

- Congesstion avoidance: increase cwnd by one packet per RTT
    - Continue until network limit is reached; if a packet times out, reset cwnd = 1 and restart entire process

TCP fast retransmit: detect packet loss by duplicate ACKs (indicating OOO transmission) -> when sender receives 3 duplicate ACKs, retransmit lost packet without waiting for timeout

- Restart timer upon retransmitting packet
- Congestion Avoidance: Additive Increase, Multiplicative Decrease (AIMD)
    - Objective: cautiously probe for unused resources, quickly recover from overshoot (without having to reset cwnd to single segment & return to slow-start)
        - in steady state, sender gently probes for unused resources
        - Goal: always stay in congestion avoidance phase (after initial slow-start phase), avoid needing
    - Send cwnd-allowed segments
        - If all sent segments in last RTT time period are ACKed: cwnd += 1 segment
            - if receives an ACK, cwnd += MSS^2 / cwnd
        - Else if 3 dup ACKs: cwnd = ssthresh = cwnd / 2

        - (Else if timeout: cwnd = 1 segment)

Issue: May be that we send cwnd packets, but receive 3 duplicate ACKs -> cwnd becomes cwnd/2, already inflight packets fall out of cwnd

- **Fast Recovery**: inflate cwnd by # duplicate ACKs received (without allowing any new packets to be sent)
    - Alternate interpretation: "skip" including 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 in network/not inflight

TCP fast recovery

- Don't perform slow-start/congestion avoidance cwnd increases within fast recovery (i.e. during packet loss recovery); reinitiate once back to normal
    - Duplicate ACK arrives -> packet out of network -> increase cwnd by 1 segment (cwnd inflation)

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

- Throughput: W / RTT
    - Immediately after loss: window -> W/2, throughput -> W / 2RTT
- On average (rough estimate): 0.75 W / RTT

TCP congestion control - fairness

- Fairness: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity
    - TCP congestion control is all done on sender end
    - 2 connections in congestion avoidance -> equal bandwidth sharing (oscillate between majorities)

History of TCP

- Originally: TCP, IP one protocol
- 1974: 3-way handshake
- 1978: TCP, IP split into TCP/IP
- 1983: ARPAnet switches to TCP/IP

- 1986: Internet starts seeing congestion collapses
- 1987-1988: Van Jacobson - congestion control for TCP (TCP-Tahoe)
- 1990: Van Jacobson - fast retransmit and fast recovery (TCP-Reno)

Congestion control - can do better than loss-based congestion detection

- Network traffic can be in one of 3 states: underutilized (no queue), over-utilized (queues form), saturated (queues full, packet loss occurs)
- Loss-based control systems: probe upward to saturated point, then try to backtrack to assumed underutilized state to let queues drain
- 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
    - Set a 1-bit flag within packet, indicating congestion
    - Note: want to change sender's cwnd, but only receiver receives the flagged packet
        - Goal: receiver recognizes congestion bit, communicates with sender

Early Congestion Notification/ECN

- ECN-capable hosts/routers set ECT (0 or 1 bits) in IP header
    - Requires supporting router; ECT = ECN Capable Transport
- When router is overloaded: set 2 ECN bits to 11
- TCP receiver: set "ECN-Echo" (ECE) flag in ACK packet going to sender
- TCP sender: cut cwnd to half (congestion avoidance))

QUIC

Motivation: Make HTTP run faster

- TCP: 40+ years old; requirements change over time
- QUIC: from past lessons in transport protocol design

TCP vs QUIC

TCP:

- Connection management:
    - Identification via source & dest address + port
    - Connection setup, teardown done via agreement on first & final byte sequence number on both ends
- Congestion control: latched onto same window mechanism used for flow control
    - Notion of congestion control was discovered after TCP written, implemented as a virtual window on top of TCP

Limitations of HTTP/2 & TCP

- HTTP/2: Only solves HOL blocking at the HTTP level; relies on TCP for transport
- TCP:
    - HOL blocking can occur within a connection (due to packet loss, e.g.)
    - Connection ID tied to IP addresses → not good for mobile nodes
    - Congestion control tangled with window flow control for reliable delivery
        - Fast Recovery needed to fix congestion control window getting stuck
    - Connection setup delay: connection setup requires TCP setup (1 RTT) + TLS setup (1 RTT)

QUIC

- Transport connection management: connection ID is a pair of random numbers
    - Connection IDs are independent of IP address & port number → can continue a QUIC connection over client address, port # changes
- Secure connection setup bundled together with connection setup
    - Requires 1 handshake instead of 2
- Structured data delivery: streams & frames (inherited from HTTP/2)

        - Frames are packaged into next outgoing QUIC packet based on packet (streams are multiplexed based on priority)
            - Priority
- Decoupled congestion control from reliable data delivery
    - Congestion control: count QUIC packets
    - Flow control on data byte in each stream

QUIC terminology

- Connection: conversation between 2 QUIC endpoints
    - Multiple streams running within a single QUIC connection
- Stream: single- or bi-directional byte stream, delivered within a QUIC connection
    - Streams identified by unique stream ID (established by either end)
    - Analogous to a single TCP connection
    - Frames: basic unit in QUIC
        - Analogous to a single TCP data segment
        - Multiple types (e.g. stream frame, ACK frame)
            - Stream frame - contains data segment
                - QUIC stream frame segmented by app (vs TCP - chopped up by TCP)
        - Each stream can carry multiple frames; frame size must not exceed QUIC packet size
- Transmission: UDP/TCP technology very well-established → QUIC packages sent within UDP/TCP to leverage existing infrastructure
    - QUIC packet: structured UDP payload
        - UDP header immediately follows UDP header
        - One UDP datagram encapsulates one or multiple QUIC packets; one QUIC packet → one or multiple frames
    - Packet format: connection ID, packet number, protected payload

QUIC

- Resilience to IP address/port# change
    - An endpoint that receives packets containing source IP, port not seen before → can start sending new packets with those as dest IP, port

        - Note: must set firewall to recognize QUIC application format, let it through
            - QUIC packets: start with a 0 bit always
    - Packets from different source address may be reordered → use packet with highest packet # to determine which path to use
        - Susceptible to attack - can intercept packets within network, replay from different location (with higher packet #) → highest packet # will be from new location
            - Solution (pass validation): create a new frame with random number; to ACK, send random number back
                - Run every time address changes (1 RTT)
                - Endpoints validate that its peer can receive (& decode) packets at new address before sending additional data

QUICK packet: containers of frames

- Data steam frame: contains stream ID, sequence #
    - Stream frames in same packet may belong to different streams
- ACK frame: acknowledges received QUIC packets
    - Indirectly ACKs all stream frames sent in the packet

Packet #: each QUIC packet assigned monotonically increasing packet number

- Packet # increased by 1 for every QUIC packet sent
- Used for loss detection - receiver acknowledges all received packets by their numbers
    - In case of lost packet: retransmit contents as a new packet, with a brand new packet # (different from the lost packet)
        - Separates flow, congestion control

Multiplexing without HOL blocking

- Within one connection
- Flow control
    - Stream flow control: same as TCP window flow control procedure
    - Connection flow control: adds together window sizes of all streams

QUIC: reliable data delivery

- Stream frame: contains data segment
- ACK frame: contains cumulative ACK for largest QUIC packet # received + selective ACKs for all received packets
- No packet retransmission
    - Lost stream frames are put into future outgoing packets for retransmission

RTT measurement

- Include ACK delay in ACK frame
    - ACK delay: time period from receiving data to sending ACK frame
- Separate packet acknowledgment from data reliability control
    - TCP can only use cumulative ACK to omeasure travel time

Stream frames

- User data uniquey identified by stream ID, sequence #
- Each frame: starts with frame type byte, followed by addl. Type-dependent fields

ACK frame contains

- Largest packet # Pn received
- Packet #s of all recently received QUIC packets, encoded in ACK blocks
    - Each ACK block acknowledges contiguous range of received packets
    - Challenge: how many ACK blocks to include? (QUIC frame: fixed size)
        - Omit older ranges (smallest packet #s) until ACK frame fits within single QUIC packet
    - For bidirectional data flow: when packet containing ACK frame Fa acknowledged, receiver can stop acknowledging packets <= largest acknowledged in Fa
- ACK delay: Time delay between Pn reception time & Fack sending time
- Number of ECN (explicit congestion notification - carried in IP packets) if any

Loss detection

- QUIC receiver sends ACK frames to inform sender of all received packets
- ACK frame acknowledges QUIC packets
    - Can be carried in any packet going th eother direction

2 types of loss detection (2 thresholds for considering a packet P lost)

1. Packet number-based: difference between largest ACKed packet # & a packet P's sequence # exceeds some number
2. Time-based: difference between largest ACKed packet #, P's transmission time

Probe timeout: when a PTO timer expires, sender must send at least one ACK-eliciting packet

- Should carry new data when possible; may retransmit unack'ed data if new data is unavailable, or flow control does not permit new data to be sent
- May send up to 2 datagrams containing ACK-eliciting packets (to avoid expensive consecutive PTO expiration due to a single lost datagram)
- Note: PTO timer expiration does nto indicate packet loss; do not mark unACKed packets as lost yet
    - Multiple PTO expirations → indicate persistent congestion

Retransmitting frames

- Stream frame ID: stream ID + sequence #
- Sender keeps mapping between packet #, stream frame ID
- When receive ACK for packet #, sender marks all frames carried in that packet ACKed
- If a frame is deemed lost → put into transmission queue by priority order

Congestion control

- Still based on detecting packet losses
    - Can use ECN if available
- Still uses window-based congestion control, similar to TCP
- QUIC's improvement: congestion control via packet #, decoupled from stream flow control window
    - QUIC packet #: used by congestion control
    - Stream frame sequence #: used by flow control window & reliable delivery

TLS

- Runs over relib
- 2 layers: Record Protocol & Handshake Protocol

HTTP/3:

- Lower protocol changes: TCP+TLS → UDP+QUIC
- Streams, flow-control function moved to QUIC
- Parallel request transmission supported by QUIC