Stanley Wei

Yu - W25

CS 118: Computer Network Fundamentals

## Contents

[**Contents 1**](#_nocxxpp7u799)

[**The Internet 2**](#_r735b9lcl832)

[Computer Networks 4](#_g8a5sw8tc03j)

[**The Application Layer 6**](#_2aean9jifus8)

[HTTP 8](#_luh9wjemfmjs)

[DNS 13](#_myouxtbeyxkc)

[**The Transport Layer 18**](#_mgmtlfjxlvj5)

[UDP 19](#_79dls8tuunz4)

[Reliable Data Transfer 21](#_olyh82rs7xf7)

[TCP 22](#_df1mzzg4tlqk)

[Congestion Control 28](#_jlkkzvocbjp9)

[(∗) QUIC 32](#_ih8g356rzlbv)

[**The Network Layer 37**](#_ikih0hh1pzod)

[IP 39](#_jf737zs5tdcs)

[DHCP 45](#_fdy7vj8cthjj)

[Network Address Translation 46](#_dm0fegqxklcx)

[IPv6 48](#_gu0w0282o667)

[(∗) Network Security 50](#_8c9iqmpj5zfg)

[**The Application Layer 55**](#_dlutabs2bko1)

[Routing Protocols 55](#_aqjeyab7z017)

[OSPF 60](#_btg41seibbmb)

[BGP 66](#_2z0dobwplrd8)

[**The Link Layer 70**](#_ehj7nyty8bba)

[Multi-Access Protocols 72](#_fuapc5rkyxj9)

[Ethernet 73](#_45796h33vs70)

[(∗) 802.11 (WiFi) 78](#_2bs2zwdcb3fs)

[LANs 80](#_g8ttjlxzhw2b)

## 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

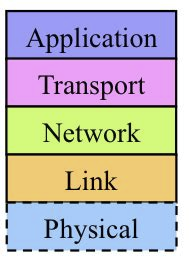
The Internet is a “network of networks”

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

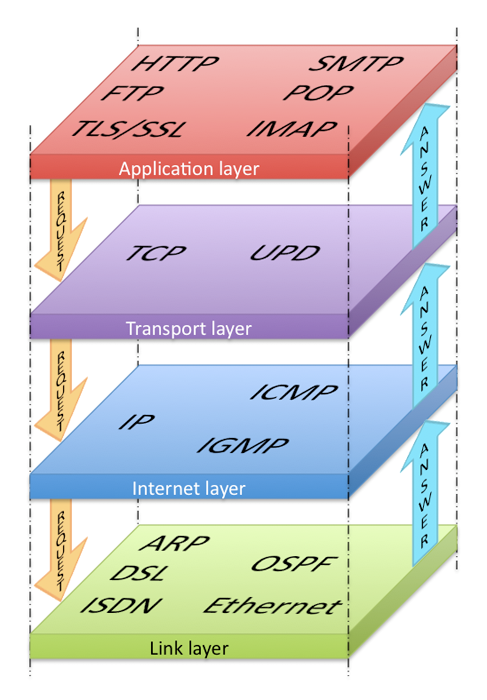
* Dictate message formats, order of messages sent/received, and actions taken on packet transmission & receipt
* Internet protocol standards: from RFCs, IEEE standards, W3C, etc.

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

1. **Application layer**: Support data exchange between application processes
   1. Ex: SMTP, HTTP, DNS
2. **Transport layer**: Handle delivery of packets (esp. ensuring reliability)
   1. Also handles ***multiplexing*** within a host (keeping up multiple “conversations” within a single Internet access point)
   2. Ex: TCP, UDP
3. **Network layer**: Define how formatted packets are forwarded from source to destination
   1. Outline which jumps are taken at every step
   2. Ex: IP
4. **Link layer**: Define how data is transferred between directly connected network elements (e.g. host → router)
   1. 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
   1. 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
   1. 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
   1. Generally ignored, negligible
2. **Transmission delay**: Packet length / rate (link bandwidth)
   1. Represents amount of time needed to “push” the entire packet into a link
   2. *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
   1. Determined by physical propagation speed in medium (e.g. cable)
4. **Queuing**: # packets in queue \* transmission time of each packet
   1. 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**

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

### 

### 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

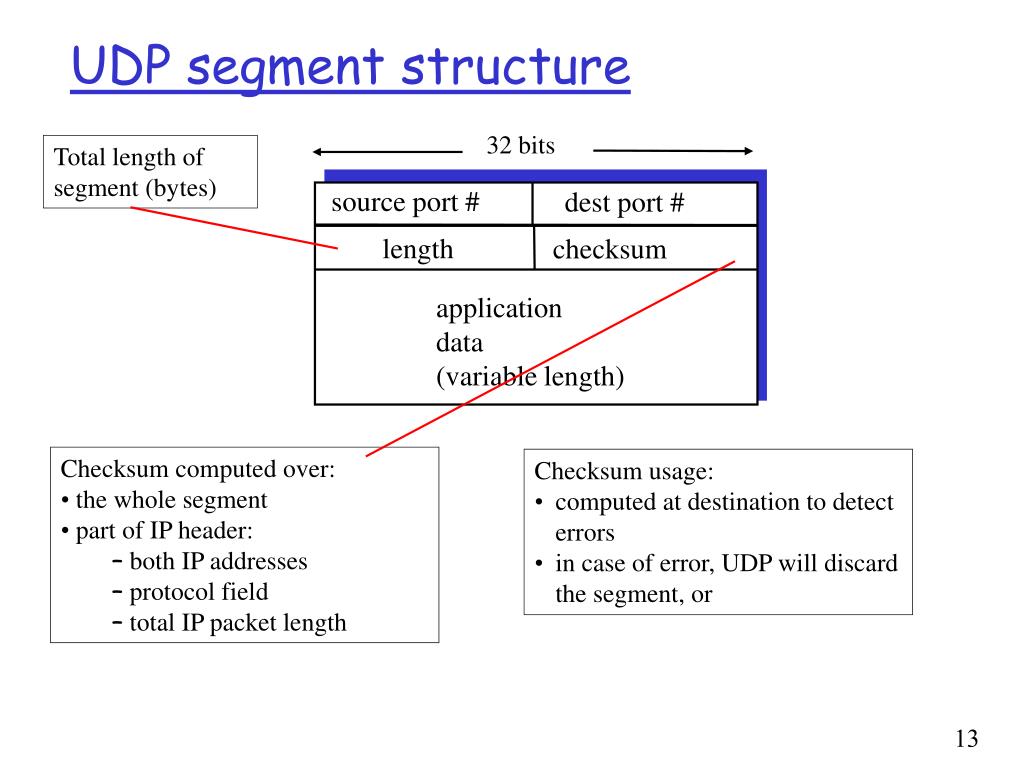
* Multiplexing: sender gathers data from multiple application processes, sends them as a whole under a single message (with associated header)
* Demultiplexing: receiver delivers received segments to correct app layer process

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



### 

### 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

### 

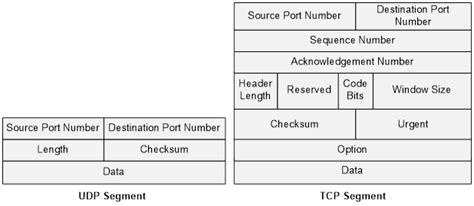
### TCP

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
   1. 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

### 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

## The Network Layer

Network layer: under transport layer, above link layer

Network layer: transports segments from sending to receiving host

* Sender: *encapsulates* segments into IP datagrams, passes to link layer
* Receiver: delivers segments to transport layer protocol (for unpacking/*decapsulating*)
  + Removes IP header to decapsulate

Network layer protocols are in every Internet device: hosts, routers

Routers: examines header fields of all IP datagrams passing through it, moves from input to output ports to transfer along path

* Routers are middle parts of end-to-end path
* Two functions:
  + **Routing**: fill in router’s forwarding table (FIB) with best path to each destination
    - Routing protocols: special application-layer protocols
  + **Forwarding**: use destination address in each packet as input to FIB lookup, FIB determines the best path to take (best next hop)

Routing & forwarding: data and control plane

* **Data plane** (forwarding): local, per-router function
  + Determines how datagram arriving on router input port is forwarded to router output port
* **Control plane** (routing): network-wide logic
  + Determines how datagrams are routed among routers, tries to find optimal path A->B

### 

### IP

Internet: a datagram network

* Hosts are connected to subnets; subnets are interconnected by routers
* All hosts, routers speak IP

Internet Protocol (IP) provides two functions:

* Datagram delivery from source to destination
  + Source, destination determined by IP addresses
* IP fragments packets along the way if needed, reassembling at destination host before passing to transport

IP is stateless

IP datagram format: header, options (if any), data (variable length, typically a TCP/UDP segment)

* Header format:
  + Version #, header length (4 bytes), type of service, total length
    - Both header, payload can have variable lengths
  + *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, causing packets to stay inside Internet forever; starts at 16 (arbitrary)
  + Protocol: indicates upper-layer protocol to deliver data to
  + IP header checksum
  + Source, destination IP address

IP *in-network* fragmentation and reassembly

* Different subnets have different transmission methods (link layer protocols)
  + Ex: Wifi, Ethernet, Bluetooth
* Different links -> different maximum transmission unit (MTU), maximum payload that can be put in the link layer
  + Sending host uses its local MTU size
  + If next link has smaller MTU, routers fragment IP packets
    - Chop packets (i.e. packet payloads) to MTU size of next link
    - May be further fragmented later down the path
    - Each fragment - a separate IP datagram
* Packet fragments are reassembled at destination host
  + Receiving host sets a timer when receiving first segment of an IP packet
  + If all fragments received within window, reassemble & pass to transport; otherwise, drop the packet (delete received segments)

Fragmentation details (IPv4)

* Identifier: generated by sending host, identifies all segments in same IP packet; stays unchanged when re-fragmented
* Flags:
  + Bit 0 (leftmost): reserved; bit 1: don’t fragment; bit 2: more fragments (MF)
* Fragment offset: offset from first byte in original payload (treated as byte stream) <- similar to TCP
  + Counted in units of 8-byte blocks
* IP header: total length, header checksum, offset adjusted when packet is fragmented

Fragmentation: very important at time of IPv4 design

* In-network fragmentation; allows packets to get through networks with smaller MTUs than original packet size
  + Downsides: losing a single fragment -> lose whole IP packet
    - Resources used to forward non-lost fragments is wasted
* Today’s practice: avoid in-network IP packet fragmentation
  + Instead: use ICNP (network layer), Path MTU discovery
    - Path MTU discovery: TCP tries to send a packet source->host, with bit set; if packet with indicator bit is too large, router sends a message back indicating too-large packet
      * Done recursively until message goes through

Source, destination IP address - meaning, format have changed over time

***IP addresses***: a 32-bit identifier associated with each network (host/router) ***interface***

* One device can have multiple interfaces (e.g. one for Ethernet, one for Wifi) -> will have multiple IP addresses
* Each interface has a unique address

***Network interface***: connection between host/router and physical link

* Can have virtual interfaces on a host (e.g. Docker)
* Routers typically have multiple interfaces
* Hosts typically have 1 or 2 interfaces
* Dotted-decimal IP address notation: 223.1.1.1, e.g.

IP ranges in forwarding tables

* IP connects subnets to routers
* Routers list range of addresses (address block)
  + Output link/interface towards destinations
  + Use local forwarding table to map destination address ranges to different output links
* Each router interface: represents a subnet

How does host get IP address?

* May be hard-coded by sysadmin in config file (e.g. /etc/rc.confg, Unix)
* Next lecture: DHCP (dynamic host configuration protocol): dynamically requested & obtained from server

How do networks get IP address ranges?

* Analogous to DNS, hierarchical approach
* Internet Service Providers (ISPs) and some large user sites, get blocks of IP addresses from Regional Internet Registries (RIRs)
  + Internet customers get a sub-block from ISP’s address block (ISP allocates sub-blocks to customers)
  + Network portion can take any arbitrary number of bits
    - Ex: first 20 bytes is ISP’s block, remaining bits are “free space” to suballocate to individual organizations
    - Split is logical, always from perspective of a specific router
  + Top-level organization: ICANN

IP addressing: CIDR and subnetting

* Classless Interdomain Routing (CIDR): network portion of address is of arbitrary length; remaining length is free space for address interfaces
  + Address format: a.b.c.d/x; x is # bits in network portion of address
* Subnetting: allocate additional # of bits to a subnet organization -> new address: a.b.c.d/y (y is original x + # bits in subnet’s portion)
* Routers don’t care about how address is arranged, only how to best construct forwarding table

IP packet forwarding: longest-match lookup

* Destination-based forwarding: only look at destination address
* Routing protocol builds forwarding table/FIB in all routers
  + FIB: matches each IP prefix to an outgoing interface
* To forward a packet: router finds longest-matching prefix entry for destination address, uses that output interface
  + Longest-matching prefix: based on binary encoding

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

* Typically: connect hosts within a switch (e.g. Ethernet switch)
* Local network operator: responsible for connect their subnets with a router
  + To make subnet global-reachable: must assign an address range to each subnet
    - Multiple subnets -> divide address range into smaller address ranges
    - Within each subnet, assign (to hosts) addresses within that subnet’s address range

Special addresses (reserved)

* 255.255.255.255/32: broadcast address of “this network”/subnet (determine dby subnet mask)
  + Set as destination address -> goes to all hosts within subnet (doesn’t go to router)
* Last address of the network (e.g. 223.1.1.255 for 223.1.1.0/24): broadcast address for local network
  + Used to contact network operator in DHCP, e.g.
* First address of network (223.1.1.0 for 223.1.1.0/24): network address
  + Captures all traffic to a particular interface, for routing purposes
  + By convention; is not assigned to end hosts
* 0.0.0.0: indicates “default route”
  + Used in packet forwarding, when no specific route can be determined for a given IPi destination

Addressing is done hierarchically

* ISP routers receive all Internet traffic with destination addresses beginning with their address range prefix; forwards to downstream organizations
* Route aggregation:
  + Switching ISP -> can obtain a more specific route for Internet traffic (less jumps)

CIDR vs subnetting

* Two solutions to same goal (making more efficient use out of limited IP address space) by deciding # bits in an IP address for network ID
* CIDR: routing protocols tell routers length of each address block (network ID) in the router’s FIB
* Subnetting: local network operators configure a subnet mask to the routers within the destination network
  + More specific approach
  + Length of each address block (network ID) in router’s FIB

Local network operator: manage a router

* Configure router interfaces with address, subnets
* Next: configure individual hosts
* Afterward: router runs routing algorithms to build FIB

Subnets: determined by physical connectivity (not routers), identified by an address block

* Not all address blocks correspond to a subnet

### DHCP

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

* Data: IP address of an interface, subnet mask, default router’s IP address
  + HTTP: requires DNS caching resolver IP address(es)
* Can be hard-coded by sysadmin in configuration file, or (today) obtained via DHCP

DHCP overview

* New host sends DHCP discovery
  + Each server must have one DHCP server (or proxy) -> forwards DHCP discovery message to the server
* DHCP server respond with DHCP offer; host responds with DHCP request (accepts offered configuration parameters)
  + Subnet may have multiple DHCP servers -> can receive multiple offers, host can choose one & respond to that specific server
* DHCP server sends address (DHCP ack) to confirm offer
  + DHCP server adds local host to local database

DHCP client-server scenario

* DHCP runs over UDP (since DHCP server address not known):
  + All messages (discover, offer, request, ack) sent to 255.255.255.255
    - 255.255.255.255 -> indicates broadcast message
  + Client’s source address always 0.0.0.0 (in discover, request)
  + Server’s offer, ack source address is a valid IP (server’s)
  + Network configuration is “leased” for a given time period; can be “renewed”
    - Renewal: don’t need to repeat discovery, can just send the request

Today: IP address, netmask, gateway, (optionally/commonly) DNS all configured via DHCP

* DHCP also called “auto config”

### Network Address Translation

Network address translation (NAT): IPv4 addresses (32 bits) aren’t enough to handle all possible IP addresses; need a solution

* ICANN: last IPv4 address handed out in February 2011
* Solution: can simply add another layer of indirection
  + NAT: originally designed as short-term solution, IPv6 intended as long-term

IP address space management

* ICANN -> Regional Internet Registries -> ISPs, institutions, organizations -> end users

Idea: can treat port number as part of namespace (in addition to IP address)

* Can associate a larger private address space to a single port number
  + Can use address blocks (internally) to hide multiple hosts/IP addresses behind a single public IP address, without communicating with any outside party

Private network:

* Every host within a private network has its own private IP address
* All packets leaving private network (via router) is assigned the same source IP address

NAT: network address & port translation

* NAT keeps NAT translation table: maintains mapping between LAN/private address + port # pair, WAN/public address + port # pair
* Local host sends packet to some destination -> NAT replaces packet source address from local address to public address
* Reply arrives with the source address + port number as destination address -> NAT replaces packet dest address with LAN address + port #

NAT: no standardization, different designs between different vendors

* Available port space range varies between organizations
  + RFC6056: 1024-65535, IANA/RFC6335: 49152-65535

Preventing NAT translation table overflow: keep a timer on each table entry

* If a translation entry times out, delete it
* No standard for how to set a timer
  + If table overflows, can’t accommodate new transport sessions

NAT implementation

* NAT translation table: map (source IP addr, port #) to (NAT IP addr, port #)

NAT problems

* Increased complexity: router has to keep a NAT table
* Single point of failure (router)
* Limited scalability due to port #, NAT address block limitations
* Cannot run services inside a NAT box
  + All application designs have to worry about NAT traversal problem

No services: NAT only creates new entry based on outgoing traffic

* No public address for outside servers to contact until entry is made
* One solution: statically configure NAT to forward incoming connection requests at a given static port to server (static port forwarding)
  + Ex: always forward port 2500 to an address 10.0.0.1 : 2500
* Another solution: 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
  + Essentially: tries to automate static NAT port map configuration
* Solution 3: application-layer relaying (used in Skype, e.g.)
  + NATed client app establishes connection to public relay, external client connects to relay -> relay bridges packets between two endpoints
    - Used in cases where there are multiple layers of NAT, e.g. (-> would prefer to operate at application layer)
  + Host, client initiate connection to external relay (through NAT); afterwards, can connect to each other directly via relay

IP-in-IP: IP tunnelling

* Another solution to IP exhaustion problem; originally designed for security
* Idea: encapsulate an IP datagram within a larger IP packet
  + Outside IP packet: deals with public addresses; internal IP datagram: private addresses
* IP node receives IP packet, sees destination address is its own address -> removes outermost header, looks at next header
  + If next header’s source, destination addresses are within the network, can send the datagram within the network

### IPv6

IPv6: original planned long-term solution to IP exhaustion (+ other changes)

IPv6 packet format:

* Fixed-length 40-byte header
  + Length field excludes header
  + Address length: 32 -> 128 bits (much larger address space)
* Moved fragmentation, IP options out of base header
* Eliminated header checksum
* Type of service -> traffic class
* TTL -> hop limit, protocol -> next header
* Added flow label field

IPv6 header format:

* 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)

Notes

* No checksum: checking moved (over time) to link layer
* Fragmentation: performed end-to-end
* IPv6 has various extension header types:
  + Routing, fragmentation, authentication, hop-by-hop options, etc.
    - Most extension headers examined only at destination
  + Extension headers placed between base header, data; can have multiple extension headers
    - Each extension header specifies next header (indicates next header’s type, or TCP if it is last header)

Issue: IPv6 has different header format, but IPv4 is already widely used + difficult to upgrade routers

* Proposed solution: dual stack (operate Internet with mixed IPv4, IPv6 nodes)
  + Dual stacked nodes have both IPv4, IPv6 addresses
    - Issue: doesn’t solve IPv4 address shortage problem
  + Link layer tells dual stacked nodes which protocol to use

Nowadays: tunneling IPv6 packet through IPv4 network

* Routers keep both IPv4, IPv6 FIB
  + Router receives IPv6 packet, but IPv6 FIB points to IPv4 address -> encapsulates IPv6 datagram within IPv4 packet
* Backbone: IPv4

### (∗) Network Security

Issue: when network security was first conceived, existing protocol stack (e.g. IP) already implemented -> needed to somehow implement network security on top of it

* Lowest layer where security could be implemented: transport layer

Defining network security: via the ***CIA principles*** (+1)

1. **Confidentiality**: Only the intended receiver can see the message contents
2. **Authentication**: The receiver can confirm the identity of the message sender
3. **Data integrity**: Any changes to the message (in transit or afterwards) can be detected

* **Availability** (New): Services/data are available to users
  + Big modern threat: DDoS (distributed denial of service)
    - IP spoofing: pretend to be the target, and send numerous requests to DNS cache resolvers
  + CIA: via cryptography; availability: more sophisticated

Basic network security: friends and enemies (intruders)

* Assume:
  + 2 entities (friends) want to communicate securely
  + Intruder may intercept, delete, add, or modify messages

In general: how to protect secrets?

* Easy to make, hard to guess: password, security questions
  + Issue: does not provide full verifiability, mathematically
* Mathematical idea: Factoring is slow, multiplication is easy
  + Factoring a very large number is NP-hard

Cryptography: Using number theory properties to

* Given a plaintext message m, encrypt into ciphertext Ka(m) with key Ka; use decryption key Kb to recover m = Kb(Ka(m))

Two types of cryptography: ***symmetric*** and ***asymmetric key cryptography***

* **Symmetric**: Both sides share the same key Ks
  + Issue: Both sides have to somehow securely share the same key value with each other
    - Requires an “out-of-band” channel
* Asymmetric:
  + **Public key cryptography**: Sender and receiver do not share a secret key; each of them produce a pair of keys - public key (known to all) and private key (known to themself)
    - Public key is somehow derivable from private key, but not vice versa
      * Kpriv(Kpub(m))
      * Modern RSA: must take at least a thousand years to derive private key from public key
    - Important property (RSA): Kpriv(Kpub(m)) = m = Kpub(Kpriv(m))
      * Ensures confidentiality (use public key to encrypt private message), nonrepudiation (if sender uses private key to encrypt, everyone can use public key to prove they sent it)

RSA: Creating public/private key pair

* Choose two large prime numbers p, q (e.g. 1024 bits each)
* Compute n = pq, z=(p-1)(q-1)
* Choose e< n that is relatively prime with z (shares no common factors)
* Choose d such that ed-1 is exactly divisible by z
* Public key is (n, e); private key is (n, d, p, q)

Issue: public-key is computationally expensive to encrypt long messages

* Lots of multiplication involved
* One alternative: use public key cryptography to obtain (symmetric) session keys
  + Use public-key cryptography to share a symmetric key between both sides; once sent, switch to symmetric key cryptography
  + One issue: how does each side know the other side’s message was actually sent by the other side?
    - Via digital signature (hashing)

Cryptographic hash function/message digest: want to map a (potentially long) variable length message to a fixed-length, easy-to-compute fingerprint

* By public key crypto: sender has two keys
  + Cryptosign(privvate key, data) -> signature of data
  + Cryptoverify(public key, data, signature) -> validation
* Sender: uses hashing algorithm to generate one-way hash from original data; private key encrypts hash to form digital signature; sends signature, data
  + Receiver: public key decryption to obtain one-way hash; uses hash algorithm on data; compares new hash with decrypted to verify signature

Message authentication code (MAC): three algorithms

* Key generation algorithm - selects a key from key space uniformly at random
* Given key and message: symmetric signing algorithm efficiently returns a tag (MAC code)
* Given key, tag: verifying algorithm efficiently verifies the authenticity of the message
* MAC
  + Strong protection than hash; compared to signature, provides authentication without identity (same secret key used for MAC generation and message verification)

Tradeoffs: asymmetric keys are mathematically more difficult to break than symmetric keys, but are significantly more expensive to use

* For extremely secure applications: asymmetric
* For one-time authentication: symmetric

Transport-layer security (TLS): Want to achieve CIA principles within transport layer

* Goals:
  + Data confidentiality via symmetric key cryptography
  + Data integrity via keyed message authentication checksum (MAC)
* Issue: TCP sends byte streams, where to put MAC?
  + Don’t want to put at beginning (no data sent yet), end (would need to wait for all data sent)
  + Solution: chop byte stream up into a series of ***TLS data records***
    - Each record carries a MAC; receiver checks each record as it arrives
    - Sender chops messages into records, encrypts, calculate MAC, send; receiver reads data, verifies MAC, decrypts data, reassembles chunk back to message, and delivers the message to upper protocol layers
* Two components of TLS:
  + Handshake protocol: authenticates communicating parties, negotiates cryptographic parameters, and establishes shared keying material
  + Record protocol: uses handshake parameters to protect subsequent traffic
* TLS: data delivery, reliability provided by TCP

TLS handshake protocol

* Client connects to a TLS-enabled server, chooses a cipher suite (cipher + hash function) from a server-provided list of supported
* Server notifies client of its pick of cipher, hash function
* Server provides a digital certificate: contains server name, public key, proof of certificate authenticity by a trusted ***certificate authority*** (CA)
  + In practice: certificate also contains a TTL, eventually expires
* Client confirms validity of server certificate before proceeding
* Generting session key:
  + Old method (SSL): encrypts random number with server’s public key, sends result to server; both parties use random number to generate unique session key for subsequent encryption & decryption of data during session
    - Issue: if private key is ever stolen, attacker can decrypt all previously-sent old messages (since same # used for encryption, decryption)
  + New method (key exchange, Diffie-Herman): securely generate a random and unique session key for encryption and dceryption (after handshake)
    - Derive a secret key from public key -> gives forward secrecy: if server’s private key is disclosed in future, cannot be used to decrypt current session, even if session is intercepted and recorded by a third party

TLS over TCP

* 3-way handshake to set up TCP connection
  + Last ACK piggybacks TLS handshake
* TLS starts its own handshake
  + Last message of TLS handshake piggybacks message data
* Send application data using TLS record protolc

Public key certification authorities (CA): binds public key to particular entity E

* Entity E registers public key with CA; provides proof of identity (via out-of-band channel, e.g.); CA creates certificate binding E to public key, certificate containing E’s public key digitally signed by CA (verifies authentication)
  + Apply CA’s public key to certificate to authenticate
* Proof of identity: 2 methods
  + E changes its DNS record
* Notes
  + List of CAs is technically “personal” decision; in practical usage, list of CAs come preset in browser/OS
    - CA certificates are self-signed certificates
    - LetsEncrypt: free CA; only verifies DNS ownership
  + How do end hosts get CAs’ public keys a priori?

## The Application Layer

### Routing Protocols

Routing: determining path taken by packets to reach destinations

* Routing plane/control plane

Two ways to structure network routinng/control:

1. Per-router control (traditional): routers run routing protocol to set up forwarding table
2. (Logically) centralized control - more recent
   1. Software-defined networking (SDN)

Per-router control plane

* Each router gathers info from other routers (via routing protocols), then uses a routing algorithm to compute local forwarding table

How to find the best path from a router (node) to a destination?

* Route computation algorithm: given a graph (***network graph abstraction***: set of nodes/routers + edges/links), find the least-cost path from a given node to all other nodes in the graph
  + Each link/edge associated with a cost (may represent network load, e.g.)
  + Assumptions: connected graph, all costs positive
* Route computation algorithms
  + Link-state (Djikstra): given a complete graph (i.e. all nodes and link costs), each nod eperforms its own computation of shortest paths to all destinations
    - Issue: requires global knowledge
  + Distance-vector (Bellman-Ford): each node knows its link cost to neighbors, and computes its shortest paths to all destinations based on the shortest paths of its neighbors
    - Each node only keeps local knowledge regarding its neighbors
* 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

**The Link-State Algorithm**

Link-state algorithm (Djikstra): Given a complete network topology graph, each node computes the least-cost paths from itself to all other nodes

* From paths: 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
* (Djikstra’s algorithm explanation)
  + 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 the start node

Each router gains knowledge from network, but computes algorithm locally

Issues:

* Algorithm complexity: O(n^2)
  + More efficient implementation with priority queue -> O(nlogn)
  + In practice: may divide larger networks into smaller units (autonomous systems), use link-state between these
* If link cost is dynamic (e.g. link cost = amount of carried traffic), then need to keep recomputing routing
  + In particular: may see oscillations (the best path cost source -> dest may initially take one path, but on recomputing routing: the best path from an intermediate node -> dest actually goes the other way, back through the source node; can keep happening, causes propagation time to increase)
    - Changes in propagation time can affect upstream layers (e.g. cause RTT to increase)

**The Distance Vector Algorithm**

Distance-vector algorithm (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 (dynamic programming):
  + Notation: 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)
    - Node/neighbor v resulting in shortest path is next hop to be saved in the forwarding table

Distance-vector algorithm

* Initially: 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]
* X sends its distance vector D\_x to all of its neighbors
* X receives D\_v from each neighbor v, calculates D\_x’(y) [new distance vector]
  + If D’*x(y) < D*x(y), update next hop to y
  + If D\_x -> D\_x’ results a change, send an update to each of its neighbors
    - Alt: if link cost changes result in distance vector change
  + On receiving an update, node checks if its table needs to be updated

Distance-vector protocol

* Iterative, asynchronous: each local iteration is caused by local link cost change, or DV update message from neighbor
  + Continues until no more updates occur
* Distributed: each node notifies neighbors only when its DV changes
  + Neighbors then notify their neighbors if necessary
* Asynchronous: nodes need not exchange info/iterate at the same time, can propagate changes through network over time

Initialization: each node sends its distance vector to each of its neighbors

Count-to-Infinity problem

* When a link between two nodes (C and D) is broken, both sides have to update link costs
* Issue: a node C may update its shortest-path cost for D to be cost of an edge B->C + cost of shortest path B->D, not knowing the shortest path B->D takes the broken C->D edge
* C updates its shortest-path cost -> B recomputes its shortest path B->D; since C increased its cost, B increased its cost, causing C to increase cost
* Continues indefinitely, costs go to infinity
  + Meanwhile: packets B->D will keep ping-ponging between B, C (routing loop)

Count-to-Infinity problem

* Due to each node having only local information
* Solutions
  + Split horizon: If a node B reaches d via C, B tells C nothing about node D
    - A node A (relying on B, C to reach D) tells B nothing about nodes C, D
    - Principle: a router should not advertise a route back to the same interface from which it learned it
    - Issue: can only solve routing loops between neighbors; is not effective in non-linear topologies
      * + Ex: two nodes A, B directly connected to C & each other -> ping pong between A, B
  + Split horizon with poison reverse: If a node A goes through C to reach D, tells C (in its distance vector) that its distance to D is infinite [poisoned vector]
    - Issue: prevents routing loops involving only two routers, but 3+ can still have loops (mutual deception)

In practice:

* Link-state within autonomous systems
* Distance vector between autonomous systems

Mitigating routing loops without split-horizon

* Path-vector routing: each node announce its entire route/path to every destination
  + Link failure -> nodes can use info to realize there is no path
  + Used in Border Gateway Protocol/BGP

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

* Distance vector:
  + Each node sends its distance to all destinations to each of its neighbors
    - Update messages can be large (linear with # destinations), but travels over one link only
  + Each node only knows distances to other destinations
* Link-state
  + Each node broadcasts its distance to each neighbor to entire network
    - Update messages are small, but travels through all links in network
  + Each node learns entire topology map

What happens if a router malfunctions?

* Link state:
  + A node can advertise incorrect link cost; each node computes its own table
* Distance vector:
  + A node can advertise incorrect path cost, one node’s distance list used by its neighbors for their own routing selection

Routing protocols - other functions

* Monitor link, neighbor nodes’ statuses
  + If a failure is detected, send routing update to inform rest of network of changes
* Mitigate potential packet losses in routing update delivery
  + Link-state: explicitly tell every node that the link is down
  + Distance-vector: explicitly tell every neighbor that D(v) has changed

### OSPF

Routing in the global Internet

* Within an administrative domain: all routers run the same routing protocol
  + All routers share a common goal (e.g. delay, loss, etc.) when finding paths
* Global Internet: an interconnection of a large number of autonomous systems (ASes)
  + Each AS is 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. corporations, campuses)
    - May connect to multiple service providers (multihoming)

Internet routing: 2-level hierarchy

* Intra-AS: within a campus, within an ISP
  + Intra-domain routing protocols [interior gateway protocols]: RIP, OSPF, e.g.
* Inter-AS: between ISPs, between stub & transit ASes
  + Inter-domain routing protocols: only BGP

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

* Should forward packet to a gateway router in current AS [connected to a router in a different AS] -> which one?
  + Inter-domain routing must learn which destinations reachable through its bordering ASes, propagate reachability information to all routers in AS1
* 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

Routing protocols in use today:

* Open Shortest Path First (OSPF): intra-AS, link-state routing
  + Relies on complete topology map to all nodes in network
* Border Gateway Protocol (BGP): inter-AS, path-vector routing
  + Transmits router’s reachability to (not necessarily all) destinations to direct neighbors

Besides ISPs: Internet eXchange points (IXPs)

* Plug fiber in for Internet; alternative to ISPs

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

* Policy:
  + Inter-AS: admin wants control over how its traffic is routed (+ who routes through its network)
  + Intra-AS: single admin, no policy decisions
* Scale: hierarchical routing saves table size -> reduced update traffic
* Performance:
  + Intra-AS: can focus on performance
  + Inter-AS: policy may dominate over performance

BGP (preview) - allows ASes to:

* Advertise its own IP address prefixes to the rest of the Internet
* Obtain IP address prefix reachability info from neighboring ASes
* Propagate reachability info to all internal routers within the AS
* Determine “good” routes to user for learned reachability to destination prefix
  + Instead of propagating all information to all neighbors: may only propagate partial prefix reachability info to a subset of neighbors, e.g. (based on policy)

***Open Shorted Past First*** (**OSPF**): Link-state

* Each node sends a Hello message to each neighbor periodically
  + Monitors link + neighbor nodes’ statuses
* Each node broadcasts ***link-state advertisement***/***routing updates*** to entire network
  + Broadcasts periodically, or if the status of any neighbor/link changes
* OSPF messages sent over raw IP packet (not transport protocol) with protocol ID 89
  + May run over either UDP or TCP, e.g.

Building a network graph using link-state

* Every node broadcasts its local piece of the topology graph
  + Using all pieces from all nodes, can piece together a complete graph

Exactly what to reach? (routers vs prefixes)

* Each subnet is allocated a specific IP address block (= address prefix)
* Each subnet is connected to 1+ routers
* Ultimate goal of routing: reachability to all prefixes
* Link-state routing: figures out how to reach the router which can reach the destination/prefix

Link-state protocol: need reliable flooding

* Three basic elements: sequence #, timer ack
  + 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 ID or different; depends on LS type)
* LS sequence number
* LS checksum
* Length
* *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
* Link failure detection
  + Neighbor nodes periodically send Hello messages (ever 10 sec., e.g.)
  + No hello message after certain period (40 sec, e.g.) -> failure, send updated LSA
* In absence of failure: send LSA at certain intervals (every 30 min, e.g.

Link-state routing daemon: routing daemon running at each router

* Sends periodic Hello messages to neighbors
* Generates LSA either periodically or event-driven, eeach carries an increasing sequence #
* Upon receiving a new LSA:
  + Replay it (intact) to all neighbors besides the sender/incoming interface
    - Replay LSA contents in a new OSPF datagram with new source, broadcast addresses
  + Process LSP to update R’s topology graph
  + Compute shortest paths
* Each router stores the most recent LSA from all others
  + Decrement TTL of stored LSAs, discard when TTL=0

Reliable flooding of LSA updates

* Each node replays received new LSA to all neighbors, except the sender
* Receive ACK from neighbor, otherwise retransmit the LSA
  + Deliver all LSAs reliably across each hop
  + Use the link state ID + sequence # in an LSA to detect duplicates

Note: from OSPF, obtain a directed graph (link cost can be asymmetric)

* Router stores LSAs in LSDB database, uses to derive directed graph (as a table)
* Use table to compute shortest-path with Djikstra’s

What if a router crashes, then comes back online?

* Q: what is sequence number?
  + LSDB typically on RAM for performance -> on reboot, lose all information
    - Lower seq # -> stale LSA; same seq # -> duplicate; don’t want to just guess
  + Instead: rebuild LSDB from neighbors
    - Send a special request to ask neighbors to summarize database -> afterward: send requests to get LSAs from neighbors
* Vs TCP: TCP connection-oriented & doesn’t care about sender -> no recovery needed, just initiate a new connection

OSPF for a single, large AS [hierarchical] domain: may want to keep LSDB small, not expand subnetworks -> divide & conquer (***hierarchical OSPF***)

* Two-level hierarchy: Divide network into local areas + a larger top-level backbone area (connecting local areas)
  + Internal areas run internal OSPF, backbone runs backbone OSPF
    - Border gateway routers [area border routers]: summarize distances to destinations in own area, advertise to backbone
      * Included in both backbone, area
      * Area border routers: LSDB stores 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 LS in local area only, compute routing within area; forward packets to outside via area border router
    - (Boundary router: connects backbone to other ASes)
  + LSAs flood only in area or backbone; each node has detailed area topology, only knows direction to reach other destinations
  + Area summary is also an LSA, contains tuples of [netaddr, netmask, cost]
* Advantage: can add new areas easily
  + New area runs OSPF internally, only area border router communicates with backbone; other areas don’t care until area border routers are told
    - Internal routers in other areas don’t make any changes to LSDB, just forward any messages (to new area) to their area border router
  + Similarly: can remove areas easily

Big picture (network layer view):

* Network protocol: forwards packets to their destination hosts
  + Very difficult: Internet is large, run by a large # of different parties
    - May need to traverse wifi, backbone, multiple ISP tiers, cloud providers, etc.

Terminology clarifications:

* An AS is not equivalent to a single institution: some institutions have multiple ASes; many do not have their own AS number
  + AS only has to do with network topology, not physical meaning
* An AS is not equivalent to a block of IP addresses (a prefix)
  + Many institutions: use multiple (non-contiguous) prefixes
  + Many institutions: use a small portion of a larger address block belong to ISP
* AS is not equivalent to a DNS domain
  + AS is a unit of topology; DNS domain is independent from network topological connectivity
    - Company can have multiple domain names, DNS domain may not correspond to any AS

### BGP

BGP: the only inter-domain routing protocol

Previously: saw path-vector routing

* Variant of distance vector; announces whole network path to avoid routing loops
  + Inter-domain routing: announce the autonomous systems (not individual networks) along the path
* BGP routers inject prefixes to local OSPF routers
  + IP datagram follows opposite path
* Complications
  + An AS may choose not to allow other ASes to take a particular route through it
  + Different metrics for a best path: cost vs distance, e.g.

Border Gateway Protocol (BGP): Provides each AS a means to:

1. Advertise its own IP address prefixes to the rest of the Internet
2. Obtain IP address prefix reachability info from neighboring OSes
3. Propagate the reachability info to all routers internal to the AS
4. Determine “good” routes to use for learned reachability to destination prefix and policy
   1. Also: propagate a proper set of of the externally learned routers to selected neighbors

Distributing path information in BGP

* 2 neighboring BGP routers establish a BGP session over TCP (on port 179) to exchange routing updates
  + Advertising routes to destination network prefixes [route = prefix + attributes]
* AS1 router 1a advertises a prefix to AS2 -> AS1 promises it will forward packets towards that prefix; AS2 runs local policies to further process

Issue: If a BGP speaker learned reachability to some destination, how to inform other routers inside the same AS? -> ***iBGP*** vs ***eBGP***

* **iBGP**: BGP session between routers in the same AS
  + A gateway router uses iBGP to distribute new prefix info to all routers in its AS, typically
  + Router learns of new prefix -> creates entry for prefix in its forwarding table
* **eBGP**: BGP session between two different ASes (i.e. adjacent gateway routers in 2 different ASes)
  + Each side may advertise reachability to some prefix to the other

BGP messages: routing messages exchanged over TCP

* 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

Path attributes: 3 most important attributes:

1. AS-PATH: a list of ASes, through which prefix advertisement has passed
   1. 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
   1. Names outgoing interface of eBGP router
      1. Can be multiple links from one AS to a neighboring AS
   2. Next-hop changes between eBGP peers, but not iBGP; local preference injected while iBGP
3. Local-Preference: policy preference in path selection
   1. Border routers inject local-preference into received BGP updates
   2. Internal routers use it in path selection (e.g. deciding which AS to pass through to reach a certain destination)

Path attributes and BGP routing policies

* Import policy: which paths to keep vs drop?
  + Filter out unwanted routes from neighbor
  + Depending on policy: gateway router may only advertise certain routes (but not others) within its autonomous system
* Route selection: among multiple routes to a given destination, pick one to use
* Export policy: tell which neighbors about which destinations?
  + Filter out the routes you don’t want to tell your neighbor

BGP route selection: select best next-hop in the following order:

1. Local preference attribute: manually configured value according to AS policies
2. Route with shortest AS path
3. Lowest IGP [internal group] cost (hot potato routing)
   1. Namely: route with shortest path within sender’s AS, to gateway
4. \*Other criteria\*

Note: paths may be asymmetric between X->Y vs. Y->X

Internet AS interconnects: hierarchical structure

* Tier-1 ISPs: top level; full-mesh connected with each other
  + No has provider; peer-relation with each other
* Regional ISPs: customers of tier-1 ISPs
  + May peer with other regional ISPs
* Customer stub networks: lowest level, multihomed in general
  + A special type of customer network: super-giants

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
  + Customer is “multi-homed” [attached to multiple provider networks] -> does not advertise, to any provider, any route that it learned from a different provider
    - Reasoning: customer does not want to forward traffic from one provider to another
* A provider only propagates customers’ routes to peers
  + A provider passes all prefixes to its customer ASes; a customer must not pass prefixes between providers
  + A provider does not pass prefixes that are not its clients’ to other providers

Recall: why different intra- and inter-AS routing? -> Policy, scale, performance

Big picture: computation results from routing protocol are installed to the IP layer

* OSPF, BGP are both application-layer processes; OSPF is built on IP [network layer], BGP built on TCP [transport layer] built on IP [network layer]

Datagram delivery:

* Unicast: a given IP address block is announced from a single location
* Broadcast: if a packet’s destination IP is broadcast, send it everywhere
  + Different broadcast addresses for different scopes (e.g. broadcast to this link-local subnet vs. broadcast for entire address block)
* Multicast: an IP multicast address represents a group of recipients
* Annycast: a given IP address block A is announced from multiple locations
  + A route receives reachability to A from multiple neighbors, picks the shortest path to forward packets

BGP operates on trust

* BGP hijacking: an AS can broadcast a certain prefix to re-route traffic

## The Link Layer

(***Data***) ***Link layer*** (Ethernet, WiFi, etc.) - live below network layer

* Transfers packets between physically connected nodes (i.e. routers, hosts)
* Encapsulate IP packet in another frame

Basic concepts:

* Link layer address: **MAC addresses** (Medium Access Control)
* Link types: simplex, half-duplex, full-duplex
  + Half-duplex: multi-access links (e.g. Ethernet, WiFi)
* Link layer functions
  + Data framing: marking beginning & end of a data chunk
  + Error detection
  + Channel access protocols

Link layer implemented in adaptor (network interface card/NIC) or on a chip

* Ethernet card: implements link & physical layer, e.g.
* Attached to host’s system buses
* Combination of hardware/software/firmware
* Communication between adaptors:
  + Sending side encapsulates IP packet in frame, adds error checking bits; follows access control protocol to send frame out
  + Receiving side checks for errors; if ok, extracts datagram and passes to upper layer

***Data frames*** - link layer term for a block of data

* Each frame contains a header field and data field
  + Optionally: trailer field (at end, between header field)
* ***Byte-oriented framing protocol***: delineate frame with a byte of special bit sequence 0111111110
  + Q: What if bit sequence occurs in data stream?
    - Byte stuffing: sender adds/“stuffs” extra 01111110 byte after each appearance of 01111110 in data stream
    - Receiver: if single 01111110, take as flag byte; otherwise, if multiple two-in-a-row, discard first byte and take second 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: Ethernet, 802.11 WiFi

### 

### 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

Multiple-access control

* Ideal solution: Given a broadcast channel with a rate R bits/sec, if M nodes want to ransmit, each can send at rate R/M
  + Simple: no central controller, synchronization, etc.

3 classes of solutions:

* ***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
  + One approach (***polling***): master node asks slave nodes to transmit in turn
    - Concerns: polling overhead, latency, single point of failure (master)
  + Alt. approach (***token passing***): one token message passed from one node ot 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, monitors token circulation -> if lost, can generate a new one
* ***Random access protocols***: Lets a node transmit at full channel data rate R
  + No a-priori coordination among nodes
  + In case of collision, random-access protocol needs to specify how to detect collision + recover from collision

**Random-access protocols**: ALOHA/slotted ALOHA, CSMA/CD + CSMA/CA

* **ALOHA**: if a node has data to send, send the whole frame immediately
  + In case of collision, retransmit frame with probability p
  + If 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 multiple access/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:
    - 1-persistent: Retry immediately
    - P-persistent: Retry immediately with probability p
    - Non-persistent: Retry after a random interval
  + 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

### Ethernet

(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 tarnsmits 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

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

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 destniatino 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: type field changed to “length”, defined separate type field in 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
  + 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
  + IP address is hierarchical, not portable; is tied to network a node is attached to
  + Analogy: MAC address is SSN; IP address is postal address

Wireles channel characteristics/challenges

* Decreased signal strength: radio signal attenuates as it propagates through matter
* Interference signals from other sources
  + Standardized wireless network frequencies (e.g. 2.4 GHz) may be shared by other devices (e.g. microwave oven, cordless phone)
* Multipath propagation: radio singal reflects off physical objects (e.g. walls), may affect arrival time
* Other problems
  + Hidden terminal: two nodes A, C may each see a node B, but not each other; both try to send simultaneously -> collision
  + Signal attenuation: A, C see each other, but cannot hear each other due to interference at an intervening B

Ethernet switching: better alternative to shared-cable Ethernet

* Ethernet switch: 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 siimultaneously
      * Buffers frames temporarily if next LAN is busy
* Switch keeps forwarding table: 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 buld 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 (with MAC addresses unchanged) except arriving interface
* 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
  + 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: advantages & limitations

* 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

### (∗) 802.11 (WiFi)

IEEE 802.11 (WiFi) LAN architecture

* AP: access point/base station
  + BSS: basic service set (“cell”); contains wireless hosts and access point (AP)
  + SSID: service set identifier
* 802.11: spectrum divided into channels at different frequencies
  + Adminstrator chooses frequency for an AP; neighbor APs use same channel -> interference
* AP sends beacon frame periodically; contains SSID, its own MAC address
* Arriving host: must associate with an AP before transmitting
  + Scans channels, listens for beacon frames
  + Selects an AP to associate with by initiating association protocol
  + Runs DHCP to get IP address in AP’s subnet
* 802.11: passive vs active scanning
  + Passive scanning: beacon frames sent from APs
    - Association Request frame sent from host to selected AP; Association Response frame sent from AP to host
  + Active scanning: host broadcasts Probe Request frame
    - APs send Probe Response frames
    - Host chooses an AP, sends Association Request (like passive scanning)
* 802.11 multiple access
  + Similar to Ethernet, CSMA: sense channel before transmitting, avoid colliding with ongoing transmissions
  + Unlike Ethernet:
    - No collision detection -> once started, transmit a frameeto completion
    - Receiver sends acknowledgment: enables the sender to find out whether the transmission collided or succeeded
      * Why no collision detection? Weak received signals (fading) -> difficult to receive, sense collisions when transmitting; can’t sense all collision, e.g. due to hidden terminal
    - Goal: avoid collisions -> use CSMA / CA (Collision Avoidance)
* CSMA/CA
  + Sender:
    - If sense channel idle for DIFS (Distributed Inter-Frame Spacing) period, transmit entire frame
    - Else if sense channel busy
      * Start random backoff timer; timer counts down while channel busy
      * Once timer expires:
        + Channel still busy -> repeat process
        + Channel idle -> start transmitting frame, then set timer to wait for ACK

ACK received -> success; else, retry

* + Receiver: if frame received OK, then return ACK after SIFS (spacing between transmission, ACK)

Active collision avoidance: want to allow sender to “reserve” channel to avoid collisions of long data frames

* Sender first transmits small request-to-send/RTS packet to AP using CSMA
  + RTSs may collide, but they’re short
  + Set a retransmission timer; if no CTS arrival, retry
* AP broadcasts clear-to-send/CTS in response to RTs
* CTS heard by all nodes within AP’s wireless range
  + Sender transmits its data frame
  + Other stations defer transmissions
  + CTS carries at frame size info -> B knows how long to defer RTS retry

802.11 addressing

* MAC address of transmitting & destination wireless host/AP (2x)
* MAC address of next IP ndoe interface connected to AP
* Additional MAC address, used only in ad hoc mode (optional)

### 

### LANs

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 (multiple NICs)
  + Routers -> one MAC address 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
* Q: given B’s IP address, how to find B’s MAC address
  + Keeps 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

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

* Table entry stores IP address, MAC address, entry TTL
* Soft-state design: information deletes itself after a certain time unless it is refresh
  + Every time an entry is looked up, reset TTL value
* Nodes create ARP tables without intervention from network administrator
  + Requests broadcast; reply is unicast
* 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
  + B replies to A with ARP response + adds A’s info to its ARP table
    - Response is unicast (direct B->A), MAC address B->A
  + A receives B’s reply, adds B entry into its ARP table
    - Other hosts on LAN can add A’s entry to their ARP tables

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
* 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

Virtual LANs - as LAN sizes scale, users change location -> may want to change physical connection [connected switch], without changing logical conncetion

* Port-based VLAN: VLAN-enabled switches can be configured to define multiple virtual LANs over a single physical LAN