CS118: Computer Network Fundamentals

Professor Lu

Thilan Tran

Spring 2020

Contents

CS118: Computer Network Fundamentals

Overview

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

Access Networks

- access networks physically connect end systems to the first router or **edge**

router

- **digital subscriber line (DSL)**:
  - uses the *existing* dedicated *telephone* line to connect to a central **DSL access multiplexer (DLSAM)**
    - **splitter** sends data on the DSL line through Internet and voice on the DSL line to telephone net
    - DLSAM is handled by an ISP, located in their central office
  - requires a dedicated hardware device called a **DSL modem**
    - the modem takes digital data and translates it to frequencies for transmission to the DLSAM
    - the DLSAM then translates the analog signals from different houses back into digital form
  - downstream transmission rate is usually *much faster* than the upstream transmission rate
    - based on user patterns, users typically download much more than they upload
  - the telephone line can carry both data and telephone signals simultaneously, at different frequencies
    - an example of frequency-division multiplexing

- **cable network**:
  - alternatively, use the *television* line
  - *similarities* with DSL:
    - data and TV is *split* and transmitted at different frequencies over a shared cable distribution network
    - requires hardware device called **cable modem**
      - modem also converts digital data into analog frequencies
    - connected to a central **cable modem termination system (CMTS)** or **cable headend**
      - CMTS translates analog signals from modems back into digital form
    - CMTS is handled by an ISP
    - asymmetric transmission rate
  - unlike DSL, multiple homes are connected via the cable network to the ISP's cable headend
    - access network is shared, instead of having dedicated access to the central office as with DSL
    - ie. a *shared broadcast* medium

- **home network**:
  - a *lower* network hierarchy, within the home
  - eg. different types, usually a **wireless access point** connected to the

DSL or cable modem
    * various devices can *wirelessly* connect to the access point
    * speed of access point is slower than a direct *wired* connection
        · speed also dependent on the wifi card of the device connecting to the access point

- **enterprise access network** or **Ethernet**:
    - uses a special hardware device called an **Ethernet switch**
    - an example of a **local area network (LAN)**
        * used often in university and corporate settings
    - connected with ISP through some institutional link and router
    - allows for *much higher* possible transmission rates
    - end systems typically connect into Ethernet switch, eg. WiFi router and PC
- **wireless access networks**:
    - shared access networks that connect end systems to routers *wirelessly*
    - wireless LANs can reach within a building (100 ft)
        * supports up to 450 Mbps rate
        * eg. 802.11 b/g/n
    - **wide-area wireless access** coverage is almost universal (10's km)
        * provided by a cellular operator
        * much slower, between 1 and 10 Mbps
        * eg. 4G, 5G, LTE

Physical Media

- data is *physically* transferred using **bits** that propagate between transmitter/receiver pairs
- a **physical link** lies between the transmitter and receiver
    - eg. common **twisted pair** with two insulated copper wires
- **guided media**:
    - signals propagate through *solid* media eg. copper, fiber, coax
    - coax cable is made of concentric rather than parallel conductors, allows for bidirectionality
        * supports multiple channels, **hybrid fiber coax (HFC)**
    - fiber optic cable is a glass fiber carrying light pulses to represent bits
        * allows for extremely high-speed operation
        * *immune* to electromagnetic noise
- **unguided media**:
    - signals carried freely through electromagnetic spectrum

    * no physical wire
  - has issues of reflection, obstruction, inteference
  - different ranges eg. LAN, wide-area, satellite

Network Core

- the **network core** is a mesh of interconnected routers
  - its role is to send **packets** or chunks of data between hosts
- two key *functions*:
  - **forwarding** relays packets from a router's input to the appropriate router output
    * every router has a **forwarding table** that maps destination addresses to outbound links
  - **routing** determines the source-destination route taken by packets
    * these routes are computed locally and proactively using **routing protocols**, and are stored within the router
- key *technologies*:
  - **packet switching**:
    * hosts *break* application-layer messages into packets
    * packets are forwarded between routers, across links, from source to destination
      · packets *hop* through a certain number of intermediate nodes
    * each packet is transmitted *back-to-back*, not simultaneously, allowing for **full link capacity** transferrence
      · sending packets takes time (L bits) / (transmission rate R bits/sec)
    * entire packet must arrive before it can be transmitted (**store and forward**)
      · thus, the *end-to-end* delay is therefore *scaled* to the number of hops the packet must make
    * *cons*:
      · unpredictable, a link may become quickly congested
  - alternatively, **circuit switching**:
    * used in traditional telephone networks
    * no packets, switching granularity is in terms of **circuits**
      · a fixed number of circuits are available within a link
    * resources/circuits are *dedicated* for a particular call
      · essentially reserving a constant transmission rate equal to the circuits dedicated to the call
    * reservation-based, no sharing of an in-use circuit
    * circuits are *released* on call completion

    * *cons*:
      · not as efficient during idle or silent periods (eg. user on call pauses talking)
  – *sharing between users* with circuit switching:
    * ie. how to split a link into different circuits
    * with **frequency division multiplexing (FDM)**, split up the frequency domain of a link
    * alternatively, with **time division multiplexing (TDM)**, use time slices and time sharing to share the link
- why is packet switching used by the Internet over circuit switching?
  – circuit switching is less **robust**, in that if a part of a circuit fails, it may break the entire network
    * on the other hand, with packet switching, the network infrastructure is maintained even if some routers go down
  – packet switching also allows for *more users* to use the network
    * many users will be *idle* for a percentage of their time on the network
    * eg. with a 1 Mbs link, and each user using 100 Kbs and active 10% of the time:
      · this user pattern is an example of *bursty data*
      · for circuit switching, can only support up to 1 Mbs / 100 Kbs = 10 users at a time (*dedicated* circuits)
      · for packet switching, can support 35 users with a probability that > 10 are active that is less than 0.0004
  – the probability that x users are active is: $P(N,x) = \binom{N}{x}p^x(1-p)^{N-x}$
    * in order to afford a certain number of users, the probability that more than the threshold number of users are active at the same time should be extremely small
  – however, excessive **congestion** is still possible with packet switching:
    * packet delay and loss may occur when the network becomes overloaded with active users
      · packets may have to jump more links in order to alleviate network congestion
    * thus, certain protocols are needed for reliable data transfer and congestion control
  – ie. circuit switching uses *reserved* resources and allows for consistent service, while packet switching uses *on-demand* allocation and less guaranteed service

Packet Delay, Loss, and Throughput

- if the arrival rate to a link *exceeds* the transmission rate for a time:
  - packets will **queue**, and await transmission
    - the **queuing delay** is the time waited in the buffer before transmitted
    - *different* from **transmission delay**, which is the total amount of time to transmit all bits of a packet
  - packets can then be **lost** or dropped if the memory buffer for the queue fills up
- thus overall **packet delay** has multiple sources:
  - **processing delay** from checking bit errors and determining output link
  - **queuing delay** from awaiting transmission, depends on congestion
    - as (L bits * a average arrival rate) / R rate approaches 1, queuing delay becomes large
    - above 1, the average delay becomes infinite
  - **transmission delay** is how long it takes to push out all bits of the packet, depends on packet size
    - L bits / R rate
  - **propagation delay** is the time for a bit to actually travel to another router
    - d length / s speed
- the `traceroute` program provides delay measurement from source to destination
  - sends three probe packet that reaches each router along the path
  - measures time interval between transmission and reply
- handling **packet loss**:
  - when a packet is lost, the source must slow its transmission, and also retransmit the lost packet
    - different *response* for different applications:
      - eg. for video streaming, the media will buffer and prioritize lower delay and allow dropping of some packets
      - eg. for emails and communications, delay is not as important as data integrity
  - the exact response is dictated by different transmission protocols eg. TCP
- the **throughput** is the rate at which bits are transferred between sender and receiver
  - can be *instantaneous* or *average*
  - often constrained by the slowest **bottleneck link** in the network
    - note that although the time it takes for packets to initially *reach* the slowest link affects the overall throughput, this is negligible compared to the botteneck of the slowest link
    - thus the constraining factor for today's Internet is usually the access network

The Internet

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

Protocol Layers

- **protocols** control sending and receiving messages
  - specifies a certain format, order of messages, and actions taken on messages transmission or receipt
  - eg. HTTP, TCP, IP
  - all communication activity in the Internet governed by protocols
- these protocols are standardized by **protocol specifications**

  - the **Internet Engineering Task Force (IETF)** handles these Internet standards
  - uses **request for comments (RFC)** for standard definitions
- however, networks have a lot of *complexity*, with many different parts:
  - eg. hosts, routers, links, applications, hardware
- protocols are therefore organized into a *stack* of **layers**:
  1. **application** layer supports network applications
     - eg. HTTP (web), FTP (files), SMTP (mail), DNS (address lookup)
     - packet of information is a **message**
  2. **transport** layer allows for process-processs data transport
     - mainly TCP (reliable) and UDP (no-frills)
     - packet of information is a **segment**
  3. **network** layer allows for global packet delivery
     - routes datagrams from source to destination
     - mainly just IP, as well as routing protocols
     - at this layer and below, implemented partly in hardware
  4. **link** layer allows for local packet delivery
     - data transfer between neighboring network elements
     - eg. Ethernet, PPP, 802.111 (WiFi)
     - packet of information is a **frame**
  5. **physical** layer allows for physical transport of bits
     - eg. copper, radio
  - note that routers and switches only implement some of the lower layers, while hosts implement all five layers
    * eg. switches implement only the physical and link layers, while routers also implement the network layer
  - every layer during the transport process attaches a header to the message
- the Internet stack originally also included:
  - **presentation** layer allowed applications to interpret data, eg. encryption and compression
  - **session** layer dealt with synchronization, checkpointing, and recovering data
  - these layers were *nonessential* for data transfer
    * applications need to implement these functionalities if desired
  - each header is more data that has to be transferred
    * thus, preferable to avoid unnecessary header overheads
- layering allows *decomposing* complex delivery into its *fundamental* components
  - the *explicit* structure identifies the relationship of the different pieces
    * each layer relies on the services provided by the layer below, ie. the **service model** of the lower layer
  - **modularization** also eases maintenance

- however, some possible issues with layering include:
  * unnecessary duplicated functionalities
  * cannot share all information between layers

Application Layer

- the highest layer in the Internet stack
  - used directly by developers to create network applications that run on end hosts
    - there are different types of network applications, **clients** and **servers**
  - *no need* to write software for network core devices, due to the layered Internet structure
- application level protocols define:
  - the types of messages exchanged
  - the message syntax and message semantics
  - rules for when and how processes send and respond to messages
- open protocols include HTTP and SMTP
  - defined in RFCs
- proprietary application protocols include Skype, SPDY (Google)
- popular Internet applications:
  - Web
  - voice-over-IP (VoIP) and video conferencing
  - media distribution
  - multiplayer online games

Application Architectures

- in the **client-server** model, the sending and receiving applications are *asymmetric*
  - **servers** *provide* services
    - are always on, with a permanent IP address to be easily found by clients
    - use data centers for scaling
  - **clients** communicate with the server, and *request* services
    - are intermittently connected, and have dynamic IP addresses
    - do not communicate directly with each other
  - this is the model used by the web and HTTP, FTP, SMTP
    - the web browser acts as a client that requests content from web servers

- in the **peer-to-peer (P2P)**, the sender and receiver have *symmetric* roles
  - there is no server that is always on
  - relies on direct communication between intermittently connected, *arbitrary* end systems called **peers**
  - peers request services from other peers and provide services to other peers in return
    * peers act as both servers and clients
  - P2P allows for **self-scalability**, since new peers bring new demands as well as new service capacity
    * scales by popularity of the network
  - peers are intermittently connected and change IP addresses
    * requires more complex management
  - place much more stress on ISPs with the demand for uploading
- normally, local proccesses communicate using inter-process communication defined by OS
- network processes will instead run on different end hosts, and use underlying network layers to communicate
  - **sockets** provide an API that encapsulates the network transport infrastructure
    * API between application and transport layer
    * proocesses will send and receive messsages to and from their sockets
- to communicate with specific hosts, a unique **identifier** needs to be associated with every host
  - every host has a unique **IP address**
  - but on the same host, multiple processes will be running
    * each process will has a unique **port number** to differentiate them
    * eg. web servers (HTTP) default to port 80, and mail servers default to port 25
  - together, the identifier includes both the IP address and port number of the specific process on the host

Transport Service Considerations

- applications have different service considerations for data transport:
  - **data integrity**:
    * file transfer, email, web transactions require 100% reliable data transfer
    * while media streaming is more **loss-tolerant**
  - **timing**:
    * realtime applications such as interactive games or telephony require low delay to be effective
  - **throughput**:

    - multimedia apps require a minimum amount of throughput to be effective, ie. are **bandwidth-sensitive**
    - more **elastic** apps eg. file transfer make use of whatever throughput they get
  - **security**:
    - encryption and data integrity concerns
- transport layer protocols:
  - **TCP**:
    - provides reliable transport, flow control, and congestion control
      - with **flow control**, sender won't overwhelm the receiver
      - with **congestion control**, the sender is throttled when network becomes overloaded
    - but *does not* provide timing and throughput guarantees, or security
    - requires connection setup between client and server
  - **UDP**:
    - does not provide reliability, flow or congestion control, timing or throughput, or security
    - but faster than TCP, and allows for maximum flexibility for developers
    - typically used by multimedia and telephony applications
  - neither TCP nor UDP provide for any encryption:
    - an ehancement for TCP called the **secure sockets layer (SSL)** allows for encryption, data integrity, and authentication
  - neither provide for timing guarantees either:
    - applications must be designed to cope with this lack of guarantee

HTTP

- content delivered on the Web is mostly just **web pages**
  - transferred using HTTP connections and using HTTP messages
- every web page includes different **objects** eg. HTML file, images, audio
  - starts with a **base HTML-file**, which includes different *referenced* ie. embedded objects that are downloaded in turn
  - each object is addressable by a **uniform resource locator (URL)** with the host name and appended path name
- the **HyperText Tranfer Protocol (HTTP)** is the Web's main *application* level protocol
  - follows the **client-server** model
    - **client** eg. a Web browser, requests, receives and displays Web objects
    - **server** sends objects in response to requests

    - HTTP clearly defines the structure of these messages and protocol for exchanging them
    - HTTP does not handle packet loss or data reordering, those details are handled by the *underlying* transport level protocol, usually TCP for reliability
    - because an HTTP server would not maintain any information about clients, HTTP is a **stateless protocol**
- connection protocol:
    1. client will *initiate* a TCP connection (by creating a socket) to the server, port 80
    2. server *accepts* the client's TCP connection
    - only uses port 80 for incoming connections
    3. client sends HTTP request message containing desired URL into connection socket
    4. server receives request, forms response message containing requested object, and sends message into connection socket
    5. TCP connection *closed* depending on the type of HTTP connection
- HTTP connections have two types depending on how many Web objects the connection can carry:
    - **non-persistent** HTTP can only carry one object (HTTP/1.0)
        * connection is then closed
        * downloading multiple objects required multiple connections
        * requires at least two **round-trip-times (RTT)**
            1. initiate TCP connection (a **three-way handshake** with acknowledgement from both ends)
                · note that the third ACK from client back to server can be *piggybacked* with request data, since ACK is just a flag
            2. make the HTTP request and to receive the header of the returned response
        * then, file transmission time is separate from the RTT
        * *cons*:
            · requires 2 RTT per object
            · OS overhead for each TCP connection
    - **persistent** HTTP can carry multiple objects over a single TCP connection
        * server leaves connection *open* after sending response
            · server closes the connection after a certain time interval
        * subsequent HTTP messages occur over the same open connection
        * client sends requests as soon as it encounters a referenced object
        * allows for **pipelining** requests back-to-back, without waiting for responses
            · responses will be sent back-to-back as well
        * *pros*:

      - only 1 RTT per additional object
  - another alternative with **parallel** TCP connections:
    - still using non-persistent HTTP
    - parallel TCP connections fetch multiple referenced objects
    - *cons*:
      - consumes more server resources
      - number of parallel TCP connections is thus limited by some servers
- HTTP/2 or HTTP/2.0:
  - derived from Google's SPDY
  - designed to *improve throughput* of client-server connections
  - *features*:
    - multiplexing multiple streams over one stream
      - pipelining multiple requests/responses together
    - header compression
    - server push ie. preemptive transfer to client

HTTP Message Format

Sample **request** message format:

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept: text/html
Accept-Language: en-us,en
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8
Keep-Alive: 115
Connection: keep-alive

... entity body ...
```

- the request message is written in ordinary ASCII text
  - each line is terminated by a CR (carriage return) and LF (line feed)
  - last line before the body has additional CR and LF
  - the first line is the **request line**, which includes the **method**, the **url**, and the **HTTP version** fields
    - the method can include:
      - `GET` - request an object
      - `POST` - to send data to server, eg. with a form (modify existing

        object)
        - `HEAD` - similar to `GET` but object is omitted, used for debugging
        - `PUT` - upload an object to a specific path on a server (HTTP/1.1)
        - `DELETE` - delete an object on a server (HTTP/1.1)
  - the rest of the lines are the **header lines**
    - some of these lines are optional
    - the `Host` header may seem redundant, but is used for caching
    - the `Accept-Language` header is an example of a **content negotiation** header
      - allows the server to select a preferred version of the requested object
  - sometimes, the **entity body** of the request can be empty, eg. with a `GET` request

Sample **response** message format:

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
ETag: "4ec9-51ee2554999234"
Accept-Ranges: bytes
Content-Length: 6821
Content-Type: text/html
Vary: Accept-Encoding

... entity body ...
```

- the response message is also written in ordinary ASCII text
  - the first line is the **status line**, which includes the **protocol version** field, **status code**, and **status message**
    - common status codes include:
      - `200 OK` - request succeeded
      - `301 Moved Permanently` - requested object has been moved
      - `400 Bad Request` - generic error code for misunderstood request
      - `404 Not Found` - requested object does not exist
      - `505 HTTP Version Not Supported` - requested protocol version is not supported
  - followed again by the header lines and entity body
    - `Last-Modified` header is used for caching
    - which headers are returned depends on browser/client type and version, user configurations, caching considerations, and more

Advanced HTTP Features

- HTTP is inherently stateless
  - reduces overhead of basic HTTP usage
  - however, it is *convenient* and *user-friendly* to record a website's user history
    - eg. option to stay signed in, or Amazon cart and purchase history
- additionally, basic HTTP only allows many connections over many commuincation links
  - throughput bottleneck

Cookies

- **cookies** allow websites to store state ie. specific information of the client
  - usually associated with a specific account or web browser
  - essentially a unique ID
  - on an initial HTTP request to a server, server creates a unique cookie ID and corresponding entry in its backend database
    - can keep user history or user information in the database entry
- has four components:
  1. `Set-Cookie` header line in HTTP response
  2. `Cookie` header line in any subsequent HTTP request
  - allows specific client to be associated with the server's database
  3. cookie ID kept and managed by user's browser or host
  4. back-end database of cookies at server
- *however*, cookies can impede on user privacy
  - browser maintains users' data or history
  - users can disable or delete all their cookies
  - tradeoff of user convenience and privacy

Caching

- **proxy servers** seek to satisfy client requests *without* involving the origin server
  - may be much closer physically to the client than origin server
  - proxy server will **cache** popular request files, keeping copies of requested objects in storage
    - incoming requests will be intercepted by the proxy server and send the copy in cache to the client
    - if the cache does not have the object, will send a request to the origin server for the object, cache it, and send it back
    - thus acts as both a client and server

    - allows for much smaller propagation delay
        * but also, just as importantly, can greatly reduce ISP *traffic*
    - mostly handled by third party **content distribution networks (CDNs)**, or large companies such as Google
- at a lower level, the *local* web browser will also create a **web cache** of popular HTTP requests
    - if the object is already in cache, the local cache will return the object
    - the dual hierarchy of caches increases chance that requested file is cached somewhere
- however, have to check retrieve the original object if the cache copy is *out of date*
    - HTTP provides a mechanism that allows a cache to verify its objects are updated
- with a **conditional GET**, the server will not send object if the cache has up-to-date cached version:
    - allows for no object transmission delay and lower link utilization
    - client sends `If-Modified-Since: <date>` header based on the cache copy
    - server sends a response with no object if cached copy is up to date
        * `304 Not Modified` status
    - proxy servers will periodically update their cache copy with the origin server's copy

FTP

- in a **file transfer protocol** session, a user wants to transfer files to or from a remote host
    - after authorizing themselves and logging in with the FTP server, files can be transferred between the local and remote file systems
    - also utilizes TCP under the surface, with dedicated port 21
    - however, unlike HTTP, FTP uses two *parallel* TCP connections to transfer:
        * the **control connection** is used to send control information such as ID, passwords, commands between hosts
            · ie. sending control information **out-of-band**, vs. HTTP that sends control info ie. header lines **in-band**
        * the **data connection** is used to actually send files
    - the control connection remains open during the session, while a new data connection is created for each file transferred
    - in addition, unlike HTTP, FTP must retain state of the user:
        * the control connection is associated with a specific user
        * the user's current directory in the remote host needs to be tracked

- common FTP commands:
  - `USER <username>` sends user ID to the server
  - `PASS <password>` sends user password to the server
  - `LIST` requests a list of files in the current directory (sent over new data connection)
  - `RETR <filename>` retrieves a file
  - `STOR <filename>` stores a file
- example FTP responses:
  - `331 Username OK, password required`
  - `125 Data connection already open; transfer starting`
  - `425 Can't open data connection`
  - `452 Error writing file`

Email

- also uses the client-server model between **user agents** and **mail servers**
  - user agent is a **mail reader** that composes, edits, and reads mail messages
  - mail server has a **mailbox** containing incoming messages for users, as well as a **message queue** of outgoing mail messages
    - servers act as both SMTP clients and SMTP servers
- when a user A sends an email to user B using their user agent (UA):
  1. the UA sends message to A's message server
  - placed in message queue
  2. A's mail server opens TCP connection with B's mail server
  - if B's mail server is down, A's mail server will try again periodically
  3. A's mail server sends message over TCP connection
  4. B's mail server places message in B's mailbox
  5. B reads the message with their UA
- mail servers use **simple mail transfer protocol (SMTP)** to deliver messages:
  - client is the sending mail server
  - server is the receiving mail server, at the reserved port 25
  - SMTP uses underlying TCP protocol
    - no packet loss is acceptable
  - note that there is a direct transfer between mail servers, with no router jumps
- SMTP phases:
  1. handshaking or greeting
  2. transfer of messages using TCP
  - note that multiple messages can be sent sequentially
  3. closure

- similar request and response interaction to HTTP:
  - entirely in 7-bit ASCII text
    * to send binary data or unicode characters, the message must be *encoded* into ASCII
  - client sends **commands**
    * eg. `HELO` , `MAIL FROM` , `RCPT TO` , `DATA` , `QUIT`
  - server sends **responses** with status code and phrase
    * eg. `250 XXX ok` , `221 XXX closing connection`
- *comparison* with HTTP:
  - SMTP uses a *push* operation
    * pushing or sending emails rather than pulling objects as with HTTP
  - SMTP uses persistent connections
  - SMTP server uses `CRLF.CRLF` to indicate end of message
    * can **dot stuff** multiple periods to escape the full stop period
  - SMTP requires message *and* data to be in ASCII
- SMTP dictates communication between mail servers
- in addition, there is a standard **text mail message format** for user agents and SMTP:
  - outlined by RFC 822
  - header lines
    * eg. `to` , `from` , `subject`
    * note that these are *different* from the SMTP `FROM` and `RCPT TO`
      · this is how phishing and email scams can occur
  - `CRLF` , ie. empty line
  - body
    * entirely in ASCII for basic mail protocols

Mail Access Protocols

- mail *access* also uses a client-server architecture
  - users read email with a client executing on their end systems
  - note that this client accesses the mailbox stored on an always-on *shared* mail server, usually handled by the ISP
  - however the issue lies in that SMTP is a push operation, so how can a user agent obtain email messages from its mail server?
    * special **mail access protocols** transfer messages from mail servers to user agents
    * users can only access mail from their own mail servers
    * allows for simple operations such as deleting mail and folder organization
  - note that the *initial* delivery of mail from a mail sender's user agent to their mail server is still done using SMTP

- **post office protocol (POP3)**:
  1. user agent opens a TCP connection to the mail server on port 110
  2. user agent sends a username and password (in clear) to **authorize** the user
     - `user <username>` and `pass <password>`
     - server response with `+OK` or `-ERR`
  3. user agent retrieves and downloads messages in the **transaction** phase
     - as well as mark messages for deletion and obtain mail statistics
     - `list` , `retr` , `dele`
  4. user agent issues the `quit` command, ending the session
     - the server deletes marked messages
     - no state information is retained between sessions
- **internet mail access protocol (IMAP)**:
  - more complex than POP3
  - an IMAP server associates each message with a folder
    - initially the `INBOX` folder
  - IMAP provides commands to:
    - create folders
    - move messages bewteen folders
    - read, delete messages
    - obtain parts of messages
  - must maintain the state of folders and messages between sessions
- **HTTP** based:
  - uses ordinary HTTP messages to transfer mail data between the user agent and server
  - eg. Gmail, Yahoo, etc.

DNS

- hosts can be identified in multiple ways:
  - through a mnemonic **hostname**, eg. google.com or ucla.edu
  - or through an **IP address**, eg. 121.7.106.83
    - has a rigid hierarchical structure, 4 bytes
    - the IP address is required to actually open communications
  - need a directory service that translates hostnames to IP addresses
    - this is the **domain name service (DNS)**
    - DNS needs a database of translations and its own application-layer protocol
- problems with a *centralized* design:
  - single point of failure
  - high traffic volume

  - can be distant from clients
  - requires high maintenance
  - doesn't *scale*
- DNS design concepts:
  - all hostnames are *hierarchical*, not *flat*
    - eg. `university_uclae_cs_kiwi` vs. `kiwi.cs.ucla.edu`
    - each dot represents one hierarchy in the overall **name space hierarchy**
  - top level domains such as `edu` , `com` , `org` are at the highest level
  - specific companies, organizations, or universities such as `amazon` or `ucla` are at the next highest level
  - each of these then have their own specific hostname hierarchies, eg. departments within ucla
  - thus, DNS servers that resolve names must *also* be hierarchical
    - each name server only handles a small portion of the name space hierarchy
- intsead, DNS is a *distributed* database:
  - made up of a *hierarchy* of **DNS name servers**
    - no single DNS server has all of the mappings for all hosts
  - **root servers** are at the highest level
    - only 13 unique ones, but replicated for reliability and security
    - provide the *bootstrapping* of the entire DNS system, ie. initial contact point
    - mappings are rarely updated, since the addresses of TLD servers do not change often
  - **top-level domain (TLD) servers** are responsibile for different top-level domains
    - eg. com, org, net, edu, gov, as well as uk, fr, ca, jp
    - different organizations maintain different servers
  - **authoritative servers** form the lowest level
    - every organization with publicly accessible named hosts must provide DNS records that map their names to IP addresses
    - thus each of these organizations has an authoritative server with *authoritative* mappings
      - or pays to store their records in some provider's server
  - a **local DNS server** is not *strictly* part of this hierarchy
    - each ISP has a local DNS server
    - when a host makes a DNS query, query is *proxied* and forwarded through this local DNS server to the DNS server hierarchy
  - note that a single DNS request can involve multiple query and reply messages
    - in addition, TLD servers may only know of *intermediate* DNS servers, which in turn know the authoritative DNS server

  - DNS queries can be configured as **recursive** queries that request another DNS server to obtain the mapping on its behalf
    * ie. higher-level servers directly provide query answers
    * burden of name resolution at upper levels of hierarchy
    * local DNS server may request for recursive queries if it is being overloaded
  - or as **iterative** queries where replies are returned to the original querying DNS server
    * ie. contacted server replies with the name of server to contact, instead of directly providing query answers
    * heavy resolution load at the local DNS server
- DNS details:
  - uses UDP and port 53
    * to reduce overhead as much as possible
    * in addition, most likely, query messages are not lost or corrupted since the message bodies are relatively short
  - utilized by other application level protocols such as HTTP to translate hostnames
  - DNS clients send queries containing hostnames to a DNS server
    * eventually gives a response which includes the IP address for the hostname
  - other services provided by DNS:
    * handling **host aliasing**: many hosts have multiple hostnames
      · DNS can retrieve the **canonical hostname**
    * similarly, handling **mail server aliasing**
    * performing **load distribution**: some sites have *replicated* web servers with *multiple* IP addresses
      · DNS will rotate the order of addresses in its reply to balance the traffic to the replicated servers
- DNS **caching**:
  - DNS is extremely expensive, so it extensively exploits caching to improve the performance and reduce messages
  - in a query chain, when a DNS server receives a DNS reply, it can cache the contained mapping into its local memory
    * cache entries timeout after some **time to live**, since mappings are not permanent
    * eg. mapping held for around two days
  - eg. local DNS servers caching TLD server addresses, or an intermediate DNS server caching authoritative server addresses

Records and Messages

- DNS servers store **resource records (RRs)** that provide the mappings

  - DNS records are stored at DNS resolvers as an entry in the database
  - an RR is a 4-tuple with the fields `(Name, Value, Type, TTL)`
    - `TTL` is time to live, ie. determines when a resource should be removed from cache
  - if `Type=A`, then `Name` is a hostname and `Value` is the IP address
    - if a DNS server is authoritative for a particular hostname, it will contain this type of record
  - if `Type=NS`, then `Name` is a domain and `Value` is the hostname of an authoritative DNS server for hosts in the domain
    - if a DNS server is not authoritative, it will contain an NS record for the domain as well as an A record for the corresponding authoritative server
  - if `Type=CNAME`, then `Value` is the canonical hostname for the alias hostname `Name`
  - if `Type=MX`, then `Value` is the canonical name of a mail server with an alias hostname `Name`
    - a company can thus have the same aliased name its mail server and another of its servers
- DNS messages have the following format:
  - DNS messages are exchanged between various DNS resolvers
    - query and reply messages both have the same message format
  - a *header* section of 12 bytes with:
    - a 16-bit ID to identify the specific message
    - query/reply flag, authoritative flag, recursion flag
    - *number-of* fields for the number of each type of resource record
  - *question* section with information about the query being made
    - eg. name and type
  - *answer* section with the resource records
  - *authority* section with records of authoritative servers
  - *additional* section with other related records
  - each individual field is **fixed-length** in DNS, unlike HTTP or SMTP

P2P

- P2P applications use a P2P architecture
  - no reliance on always-on servers
  - peers are intermittently connected and change addresses
  - self-scalable

File Distribution

- common P2P applications perform **file distribution**

- eg. BitTorrent
- clients download equal-sized **chunks** of a file from their peers
  - while uploading their own chunks
- clients can ask other peers for a list of their downloaded chunks
  - use **rarest first** technique to determine which missing chunks to download next
- clients will also choose to trade with the peers supplying at the highest rate
  - a kind of *trading* algorithm

- consider the **distribution time** D to distribute a copy of a file of F bits to N peers
  - let the server's upload rate be u_s
- in client-server file distribution, server would send a copy of the file to *each* of the peers
  - thus the distribution time is limited by the time for the server to upload the file N times, and the time for the peer with the lowest download rate d_{min} to download the file
  - for large enough N, this distribution time grows linearly with the number of peers

$$D_{client\text{-}server} = max\{\frac{NF}{u_s}, \frac{F}{d_{min}}\}$$

- on the other hand, in P2P file distribution, each peer can *redistribute* any portion of the file it receives to other peers
  - initially, only the server has the file
    - to get the file out, each bit of the file must be sent at least once
  - the distribution time is still limited by teh peer with the lowest download rate d_{min}
  - however, the total upload capacity of the system is now equal to the server's upload rate *as well as* the sum of the upload rates of each individual peer, u_{total}
  - this distribution time is *always* less than in a client-server architecture, for any number of peers

$$D_{P2P} = max\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s + u_{total}}\}$$

Video Streaming

- multimedia streaming has become increasingly popular, especially video streaming
  - video streaming can be extremely expensive traffic-wise due to immense number of users

  - network must be able to provide an average throughput that is at least as large as the bit rate of the compressed video
  - how do video streaming providers *scale* to provider for all their users?
    * while taking into acount the different capabilities of different users (eg. wired vs. mobile, low vs. high bandwidth)
- video as a multimedia:
  - videos are a sequence of images displayed at a certain rate
  - images are an array of pixels
  - can exploit redundance *within* (spatial redundance) and *between* (temporal redundance) images to decrease number of bits for encoding messages
    * eg. sending less bits for repeated colors, or only the differences between two similar frames
  - videos are streamed at a certain bitrate, depending on the **compression** that is applied to it
    * for a **constant bit rate (CBR)**, the encoding rate is fixed
    * for a **variable bit rate (VBR)**, the encoding rate changes as the redundance enconding changes
- with HTTP streaming, the video is served as an ordinary file:
  - a TCP connection is established and the server sends the video file within an HTTP response
    * as fast as the network and traffic conditions allow
  - on the client side, the bytes are collected in an application **buffer**
    * once the bytes reach a certain predetermined **threshold**, the client begins playback
    * thus the video streaming application displays video as it is receiving and buffering frames over HTTP
  - *cons*:
    * all clients receive the same encoding of the video, regardless of their own available bandwidth or speed
- with **dynamic adaptive streaming over HTTP (DASH)**, the video is divided into many chunks encoded into different versions:
  - each version has a different bitrate and quality level
    * server has a **manifest file** with URLs for the different chunks and their encoded rates
  - the client *dynamically* requests chunks of video segments a few seconds in length:
    * with high available bandwidth, client selects chunks from the high-rate version
    * with little available bandwidth, client selects the low-rate chunks
    * also selects *when* to request chunks so that buffer starvation does not occur
  - client performs a **rate determination algorithm** to select chunks

- allows videos to be streamed at different rates, *adaptively* over the session

CDNs

- for video streaming companies, optimizing the servers is an important goal:
  - building a centralized data center for all server needs is not ideal:
    - single point of failure
    - can be far from many clients
    - popular videos may be sent many times over the same links
  - instead use multiple **content distribution networks (CDNs)** that:
    - manage servers in distributed locations
    - store cached copies of videos in their server clusters
      - *replicated* across CDN clusters
      - cached based on user requests
    - direct users to the CDN locations that allow for the best user experience
- CDNs may be placed according to an *enter-deep* philosophy that deploys server clusters in access ISPs all over the world:
  - high maintenance cost, but best user throughput
- or to a *bring-home* philosophy that places large clusters in a smaller number of **points of presence (POPs)**, eg. at IXPs:
  - lower maintenance overhead, but more delay and lower throughput for users
- to be succesful, CDNs must *intercept* and *redirect* requests:
  - essentially a service acting *over the top* of the Internet to deal with a congested Internet
    - different copies may be more accessible depending on the client location or current congestion of network paths
    - additionally must decide what content to place where
  - accomplished using DNS
    - some companies with a large enough network may bypass DNS altogether
  - eg. authoritative DNS server may *hand over* the DNS query to a closer CDN
    - the CDN then uses the client's local DNS server's IP address to select an appropriate cluster
  - different cluster selection strategies:
    - *geographically* closest:
      - not always the least number of hops
      - local DNS server may be far from client
    - use *real-time* measurements of delay and loss:
      - eg. send probes to local DNS servers

Transport Layer

- the **transport layer** is sandwiched between the network layer and application layer
  - provides *logical communication* between *processes* running on different hosts and moves messages to and from the network edge
    - logical communication allows applications to treat hosts running procceses as directly connected, even though they may be on opposite sides of the world
    - ie. abstracts away the physical infrastructure required to carry messages
    - on the other hand, the network layer provides logical communication between *hosts* and moves messages within the network core
  - transport layer messages are called **segments**
    - application layer messages are broken down into chunks, each with a transport layer header
- transport protocols run in end systems
  - sending side breaks app messages into segments, and passes to network layer
  - receiving side reassembles segments into messages, passes to app layer
  - two main transport protocols, TCP and UDP
- **transmission control protocol (TCP)** provides *reliable*, in-order delivery with:
  - congestion and flow control
  - connection setup
- **user datagram protocol (UDP)** provides *unreliable*, unordered delivery:
  - no-frills extension of IP, which is a best effort, unreliable protocol that makes *no guarantees*
  - does provide some error checking
- neither protocol provides:
  - guarantee of a certain bandwidth or a minimal delay

Multiplexing / Demultiplexing

- since *multiple* processes can run at once on a host, how can different hosts interact with different processes?
  - ie. each process can have one or more sockets, so transport layer protocols are responsible for directing incoming segments to the appropriate socket
- **multiplexing** at sender:

  - handling data from multiple sockets and adding transport header
  - ie. gather outgoing data sent to the same IP from different sockets together, and send them out together
  - simple concatenation of data
- **demultiplexing** at receiver:
  - uses header info to deliver received segments to correct socket
  - each recieved IP datagram has source IP address and destination IP address
    * each datagram carries one transport layer segment
  - while each transport header indicates source port and destination port
    * port numbers are 16 bit
    * ports 0 through 1023 are **well-known port numbers** and restricted to well-known application protocols
  - uses IP addresses and port number to direct segment to appropriate socket
- *connectionless* demux (UDP) checks destination port number only:
  - when host receives UDP segment, checks destination port number and directs UDP segment to the corresponding port
  - ie. datagrams with the same destination port, but different source ports or addresses, will be directed to the same socket
  - the source port number acts as a *return address* to send segments back to
- *connection-oriented* demux (TCP) checks destination port number as well as source IP and port:
  - TCP uses a 4-tuple to identify sockets:
    * `(source IP, source port, dest IP, dest port)`
  - uses all four values to direct segment to appropriate socket
    * thus host supports and differentiates many simultaneous TCP sockets with unique 4-tuples
    * ie. datagrams with the same destination port, but different source ports or addresses, will be directed to different sockets
  - eg. web servers can support different sockets for each connected client
    * since destination address is different, can open up a different socket
- note that in both demux examples, there is no check of the destination IP
  - correctness of IP address is ensured in the networking layer

UDP

- **user datagram protocol (UDP)** is a bare bones transport protocol offering a *minimal* function set:
  - a *best effort* service where UDP segments may be lost or delivered out-

of-order
    * only provides multiplexing / demultiplexing and light error checking
- a **connectionless** service *without* handshaking between UDP sender and receiver
    * each UDP segment handled independently of others
- used for:
    * streaming multimedia that is loss tolerant, but rate sensitive
    * DNS for faster responses
    * RIP routing protocol
    * SNMP network management
    * lightweight applications that want to avoid connection establishment, maintaining connection state, and header overhead
- if some elements of reliable transfer are required, reliability must be added at the application layer
    * application-specific error recovery

- UDP segment format:
  - fixed length header like DNS
    * limits on port number and body length, but faster processing
  - 8-byte segment header includes 2 bytes each for source port, destination port, length, and checksum
    * length specifies total bytes in segment, allowing for variable length body
    * checksum for error checking
  - followed by a payload of application data
- UDP **checksum**:
  - detects errors such as flipped bits in transmitted segments
  - computed as one's complement sum of 16-bit integers through the segment contents
    * on sum overflow, carryout bit is added back to the sum
    * then one's complement is applied to the sum, flipping 1's and 0's
  - checksum also includes an **IP pseudoheader** that is not actually part of the UDP header
    * but added to the calculation to confirm that parts of the IP header (IP addresses, total length) are included in the error detection
  - receiver can recalculate the checksum on the entire transmitted segment, including the checksum, and compare
    * if no errors are introduced, due to the one's complement of the sent checksum, the new checksum will be all 1's
  - of course, if certain bits are flipped, the checksum will not catch *all* errors
    * the link layer error detection can be used instead (CRC), which is much more powerful

    * although many link layer protocols also provide error checking, there is no guarantee that all links do
        · *redundant* error checking
    * an example of providing error detection on an end-to-end basis, as per the **end-to-end principle**
  – the action upon detecting an error is implementation dependent
    * may discard damaged segment, or provide a warning

Reliable Data Transfer Mechanisms

- dealing with delivering segments *reliably* from sender to receiver
  – the complexity of a **reliable data transfer protocol (RDT)** depends on characteristics of the underlying unreliable channel
    * ie. implementing reliable data transfer functions on top of builtin, unreliable data transfer functions
    * must deal with retransmitting lost messages or reordering messages
- consider only unidirectional data transfer
  – but messages flow in both directions in reality
  – **finite state machines (FSM)** can model RDT
    * certain unique **states** for sender and receiver
    * **events** cause state transition
    * **actions** taken on state transition
- RDT 1.0: reliable transfer over a reliable channel:
  – underlying channel is perfectly reliable
    * no bit errors, no loss of packets
  – *sender*:
    * waits for call to `rdt_send`
    * then, creates a packet from the data, and sends it over `udt_send`
  – *receiver*:
    * waits for call to `rdt_rcv`
    * then, reads packet and extracts the data
- RDT 2.0: channel has bit errors:
  – underlying channel may flip bits in a packet
    * checksum allows for bit error detection
  – how should the protocol recover from errors?
    * needs two way feedback (ie. control messages) to know when to retransmit
    * **acknowledgements (ACKs)**: receiver explictly tells sender that the packet was received
    * **negative acknowledgements (NAKs)**: receiver explitly tells sender

that packet had errors
    · sender retransmits packet on receipt of NAK
- *sender*:
  * will only send the next packet after receiving ACK for the current packet
  * if NAK is received, sender will retransmit the current packet
- *receiver*:
  * will double check the checksum and the packet
  * always responds with an ACK or NAK so sender transmission can continue
- *however*, a fatal flaw in that ACK and NAK *themselves* can become corrupted:
  * ie. *garbled* ACK and NAK
  * much less likely due to very small message size
  * choosing to always either retransmit the packet or continue to the next packet can cause issues:
    · retransmitting would transmit a duplicate packet if corrupted message was an ACK
    · continuing would cause the receiver to never receive retransmitted corrupted packet if corrupted message was a NAK
  * a naive solution would be to have another acknowledgement of the ACK and NAK:
    · that acknowledgement can also become corrupted, leading to a cycle of garbled acknowledgement
  * there needs to be an *additional* mechanism to deal with this flaw

• RDT 2.1 handles the fatal flaw from RDT 2.0:
  - the conservative solution would be to always retransmit the packet:
    * in the worst case, receiver has the packet already and can drop the new duplicated copy
    * need a mechanism to detect duplicate copies
  - called a unique **sequence number** for each sent packets:
    * sequence number can be applied at the packet or byte granularity
    * sender adds sequence number to each packet
    * receiver discards packets with duplicate sequence numbers
  - leads to extra concerns about the space of sequence numbers
    * ie. minimum number of bits to encode all sequence numbers
  - can a 1-bit sequence number suffice?
    * ie. just differentiating between previous and current packet
      · eg. send packet 0, packet 1, reuse the sequence number and send packet 0, etc.
    * this is enough for the receiver to differentiate and detect duplicates in the current stop-and-wait protocol
    * to handle more complex scenarios such as out-of-order delivery, a

single bit is not enough
- RDT 2.2 same as RDT 2.1 using *only* ACKs, no NAKs:
  - want to simplify protocol as much as possible and halve number of message types
    * reducing message type is beneficial for pipelining operations, results in less server processing overhead
  - instead of a NAK, receiver sends ACK for the last packet received without corruption
    * receiver must explictly include sequence number of acknowledged packet
  - succesfully encapsulates both ACK and NAK functionality
    * at the cost of an extra space for sequence number
- RDT 3.0 same as previous RDT, but with the aim of handling packet errors *and* packet loss:
  - once again, need to introduce additional mechanisms to deal with packet loss
    * have mechanisms to retransmit, but need to detect packet loss
  - use a countdown **timer**:
    * sender waits a reasonable amount of time for ACK, and retransmits if no ACK is received
    * retransmission may be duplicate, but this situation is handled by sequence numbers
      · essential for receiver to transmit an ACK after receiving the duplicate to escape the countdown timer loop
  - *premature* timeout:
    * if the timeout value is too low, the server may retransmit *before* the ACK arrives back at the sender
      · leads to unnecessary retransmissions as duplicate ACKs and data is continuously retransmitted
      · note that protocol still functions correctly, just much more inefficient
    * on the other hand, if timeout value is too high, throughput is much lower
      · sender is waiting more time than neccesary to retransmit lost packets
    * ideally, the timeout value should be *slightly* larger than RTT
  - handles both packet loss of data or ACKs
  - still ignoring out-of-order delivery or extreme packet delay
  - the current and previous iterations of RDT all use a **stop-and-wait** operation:
    * only one packet is transmitted over a link at a time
      · leads to low throughput
    * server *must* wait transmission time + RTT to receive the ACK

- link is *unused* for one entire RTT: sending transmission and awaiting ACK
- sender utilization: $\frac{L/R}{RTT+L/R}$

* network protocol greatly limits use of physical resources

Pipelining

- using **pipelining**, the sender allows multiple, in-flight, yet-to-be-acknowledged packets
  - ie. multiple data packets as well as ACK packets flowing in both directions
  - new *considerations*:
    * issue of out-of-order arrival is introduced as packets follow different routes over the network
    * range of sequence numbers must be increased
    * buffering at both sender and receiver
    * separate timers may be required for each pipelined packet
  - sender utilization is much higher: $\frac{nL/R}{RTT+L/R}$ where n is the number of pipelined packets
    * improved by a factor of n
- two generic forms of pipelining:
  - for both forms, sender can have up to n unACKed packets in the pipeline
    * AKA the **window size**
  - **go-back-N**:
    * on packet error or loss, n packets will be retransmitted
  - **selective repeat**
    * on packet error or loss, a selected number of specified packets will be retransmitted
- go-back-N details:
  - AKA *sliding window* protocol
  - the receiver sends an ACK for every packet, but this ACK is *cumulative*
    * each ACK is the highest correctly received, *in-order*, sequence number
      - as new ACKs are received, the window of covered packets *slides* forward
    * ie. an ACK for packet n acknowledges all packets up to and including n
  - the receiver may receive packets out-of-order:
    * simply discards these packets, and reACKs with the highest in-order sequence number
    * generates duplicate ACKs to server

- the sender has a timer for the oldest in-flight unACKed packet
  - only one timer actually running at a time
    - when lowest ACK is received, timer is stopped and *reset* to the sent time minus elapsed time for the next packet
  - when timer expires, retransmit *all* packets in the window
- if a corrupted or duplicate ACK is received by the sender, ignore it
  - since more ACKs should be sent back correctly for the other packets in the window, the sender can infer which packets have been correctly received depending on the sequence number
  - otherwise, on normal ACK, the window slides forward by one and another packet is sent
- protocol seems *inefficient*, but is necessary when receiver buffer has a size of only one packet
  - receiver thus cannot handle packets out-of-order
  - receiver *only* has to keep track of the current expected sequence number

- go-back-N considerations:
  - receiver discards out-of-order packets, so sender *wastes* transmission on these packets
    - but required for low-end receivers with little buffer size
    - communication link usage vs. memory tradeoff
  - nowadays, memory is much cheaper, so the tradeoff is no longer as beneficial
  - requires $n + 1$ sequence numbers in packet header:
    - n numbers represent the packets currently in the window
    - $n + 1$th number indicates the next packet once the window has shifted
- selective repeat details:
  - the receiver sends an *individual* ACK for each packet
    - if lowest packet, deliver all lower, consecutive packets to application and move the receiver window base forward
    - otherwise, buffer out-of-order packets, instead of dropping them
    - on duplicate received packets, still send back a duplicate ACK to stop the retransmission cycle
  - the sender maintains a timer for *each* unACKed packet (a single hardware timer can be used to mimic multiple logical timers)
    - when an ACK is received, stop the corresponding timer
      - if lowest packet, move the sender window base forward
    - when timer expires, retransmit only that unACKed packet
    - ie. sender only has to resend packets that were not ACKed
  - here, both the sender and receiver have a window of size n
  - all packets covered by the sender window are ready to be sent
    - sender window slides forward *only* when ACK is received for the

lowest packet
    - packets arriving at the receiver window will be buffered if received out-of-order
        * consecutive packets at the front of the window are *delivered*
- selective repeat considerations:
    - receiver needs a buffer to hold n packets
    - trading memory for lower communication link usage
    - requires $2n$ sequence numbers in packet header:
        * with only $n+1$ numbers, say all n packets are succesfully received, but all n ACKs are lost
        * when sender retransmits packet 0, this duplicate packet will *not be dropped* by the receiver
        * not enough sequence numbers to differentiate between two extreme cases:
            · perfect case where all packets received and ACKed (need to transmit n new packets)
            · worst case where all packets received but all ACKs lost (need to retransmit n old packets)
- sequence number considerations:
    - sequence numbers are finite and will eventually be reused
    - care must be taken against duplicate packets that are long-delayed and delivered out of order
    - thus a packet cannot *live* in the network for longer than a *fixed* maximum amount of time
        * eg. 3 minutes in TCP
        * afterwards its sequence number can be reused

Version	Channel	Mechanism
RDT 1.0	no error/loss	none
RDT 2.0	bit errors, no loss	checksum error detection, ACK/NAK, retransmission
RDT 2.1	same as 2.0	1-bit sequence number
RDT 2.2	same as 2.0	no NAK
RDT 3.0	errors and loss	timer, ACK-only
Go-back-N	same as 3.0	pipeline, N sliding window, discard out-of-order
Selective Repeat	same as 3.0	pipeline, N sliding window, selective recovery

TCP

- **TCP** is a basic transport layer protocol with many more features than UDP
  - point-to-point operation with one sender and one receiver
    - *multicasting* is not possible in a single send operation
  - reliable, in-order *byte stream* without message boundaries
  - pipelined, with dynamic window size depending on:
    - TCP congestion (prevent overwhelming network) and flow control (prevent overwhelming receiver)
  - full duplex data:
    - bi-directional data flow in the same connection
    - size of each data in a segment is bounded by **maximum segment size (MSS)**
      - maximum TCP segment plus TCP/IP header that will fit into a single link layer frame
      - maximum frame length is the **maximum transmission unit (MTU)**
  - connection-oriented:
    - requires intializing of sender and receiver state, ie. handshaking
- TCP segment format:
  - *basic* or mandatory header:
    - 4 byte wide rows
    - source port, destination port
    - sequence number
      - counted in bytes not segments
      - note that for TCP messages *without* data such as ACKs or SYNs, the sequence number is still included, and the segment carries *logically* one byte
    - acknowledgement number
      - sequence and acknowledgement numbers are separate due to duplex nature of TCP
      - counted in bytes not segments
    - header length, flags for TCP connection establishment, receiver window bytes
      - reciever window indicates how many bytes receiver is willing to accept, used for flow control
      - flags include the `URG ACK PSH RST SYN FIN` bits, in order
    - checksum, urgent data pointer
      - again, like UDP, checksum includes an IP pseudoheader in its calculation
    - note that the data length is not stored here:
      - IP header contains overall packet length
      - UDP contains a length field because UDP packets are self-contained and UDP may *not* run over IP
  - *options* header field of variable length, optional

        * normally empty, so length of TCP header defaults to 20 bytes
    - application data of variable length
- TCP sequence and ACK numbers:
    - both numbers are counted in terms of *bytes* instead segments
    - sequence number indicates the byte stream number of the *first byte* in the data
    - ACK number indicates the sequence number of the overall *next expected byte*
        * using a *cumulative* ACK like GBN
    - due to TCP's duplex feature, ACK and sequence numbers are sent in both directions between two hosts
        * each indicates a different direction of communication
    - TCP specification doesn't specify how receiver handles out-of-order segments
        * but most implementations follow the buffering model from selective repeat

TCP Timeout

- choosing the TCP timeout value:
    - should be longer than RTT, but RTT varies
        * *too short* of a timer leads to premature timeout and unnecessary retransmissions
        * *too long* of a timer leads to slow reaction to segment loss
- *estimating* RTT:
    - each *measured* RTT is the `SampleRTT`
        * approximately one sample taken every RTT, instead of for every receieved segment
        * measured time from segment transmission until ACK receipt
        * note that there is an ambiguity for the ACK of retransmitted segments
            · is the ACK long-delayed, belonging to the original transmission, or in response to the retransmitted segment?
            · thus TCP *ignores* the RTTs of retransmitted segments (Karn's algorithm)
    - `SampleRTT` will vary too much, want a *smoother* estimated RTT by averaging recent measurements
    - use an exponential, weighted moving average:
        * $\mathit{EstimatedRTT} = (1-\alpha) \times \mathit{EstimatedRTT} + \alpha \times \mathit{SampleRTT}$
        * ie. trusting recent sampled RTTs more than aggregated past samples, allowing influence of past sample to decrease exponentially fast
        * typically, $\alpha = 0.125$

- calculating deviation:
  - similar biased moving average formula as before
  - $DevRTT = (1 - \beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT|$
  - typically, $\beta = 0.125$
- setting the actual timeout interval:
  - $TimeoutInterval = EstimatedRTT + 4 \times DevRTT$
  - mean plus four standard deviations guarantees 99.96% reliability
  - a conservative calculation, leaning towards keeping the timeout on the long side
  - an initial value of 1 second is recommended

Reliable Data Transfer

- TCP creates RDT service on top of IP's unreliable service using:
  - pipelined segments
  - cumulative ACKs
  - single retransmission timer for earliest segment in window
- retransmissions are triggered by:
  - timeoute events
  - as well as duplicate ACKs, known as fast retransmission
- TCP *sender* events:
  - create segment with sequence number of the first data byte
  - start timer for the oldest unACKed segment if not already running
    - ie. only one timer in the window, much cheaper implementation
  - on *timeout*:
    - retransmit segment that caused timeout
      - like selective repeat, only the segment believed to be lost is retransmitted
    - restart timer for the same segment, with double the previous time-out interval
      - if multiple timeouts continue to occur, the interval thus grows exponentially
      - interval is reset to the one calculated from sampling after an ACK is received normally or new segments are sent
      - this doubling is a *limited* form of congestion control
    - note that early versions of TCP do not deal with the case where multiple segments are lost at a time
      - would have to wait multiple timeouts to retransmit more than one lost segment
      - some versions of TCP implement **selective acknowledgement** to better deal with multiple segment loss
  - on *ACK*:
    - if ACK acknowledges previously unACKed segments

      - move the window forward
      - disable and start a new timer if there are still unACKed segments
    - note that due to cumulative ACK, some initial ACKs can be lost
      - sender can infer from later, cumulative ACKs which segments were properly received
- TCP *ACK generation* on *receiver* end:
  - upon arrival of in-order segment with expected sequence number and all previous data has been ACKed:
    - optionally, send a **delayed ACK** that waits up to 500 ms for next segment to send a cumulative ACK
      - attempts to save bandwidth by reducing the number of sent ACKs
  - upon arrival of in-order segment with expected sequence number and a segment has a pending ACK:
    - immediately send single cumulative ACK
  - upon arrival of out-of-order segment with a higher-than-expected sequence number:
    - a gap is detected
    - immediately send duplicate ACK, indiciating sequence number of next expected byte
  - upon arrival of segment starting at lower end of gap that partially or completely fills gap:
    - immediately send ACK

Fast Retransmit

- in reality, the timeout period is often relatively long
  - **fast retransmission** uses duplicate ACKs to detect lost segments
  - TCP simply uses the method that triggers first to detect loss
- TCP fast retransmit:
  - if the sender receives 3 duplicate ACKs for the same data, ie. **triple duplicate ACKs**:
    - ie. *third* duplicate copy
    - resend unACKed segment with the smallest sequence number
  - likely that the unACKed segment was lost, so avoid waiting further for timeout
    - usually triggers much faster than the full timeout period
  - note that the sender should not retransmit on the very first or even second duplicate ACK due to possible out-of-order delivery
    - segment sent to receiver may simply be delayed and arrive after later segments

Flow Control

- receiver controls sender so sender will not overflow the receiver's buffer by transmitting too much, too fast
  - receiver *advertises* free buffer space by include the `rwnd` value in TCP header of receiver-to-sender segments
  - `RcvBuffer` is the overall buffer space, typically set by OS as 4096 bytes, and `rwnd` is the remaining, free buffer space
  - sender will limit amount of unACKed, in-flight data to receiver's `rwnd` value
  - this guarantees buffer will not overflow
- one technical problem:
  - when `rwnd = 0` for a host A as its buffer becomes full from processing segments, if A has no more data to send, then another host B will never know when A's buffer has open space again
  - TCP requires that B send segments with one data byte when A's receive window is zero, so that when the buffer of A begins to empty, A will ACK with a nonzero receive window, and B will be able to send its data again normally

Connection Management

- with TCP, before exchanging data, the sender and receiver must *handshake*
  - agree to establish connection, and agree on connection parameters
  - ie. the initial 1 RTT *boostrapping* time
- note that a 2-way handshake is not sufficient, and fails under the following scenario:
  - if the client *retransmits* a connection request before the corresponding connection acceptance response
  - server may timeout the client after no data transmission, which terminates the client
  - the retransmitted connection request reaches the server, but client is closed
  - half open connection without client
  - need to establish two one-way communications
- connection establishment:
  - requires **3-way handshake**
  - first, client chooses initial sequence number and sets the `SYN` flag (synchronize)
    - from here, the initial sequence number will increase
    - starting with random sequence numbers minimizes the possibility that a segment from a previous connection is mistaken
  - server responds with an initial sequence number and also sets the `SYN`

flag
    * while *also* setting the `ACK` flag and number to acknowledge `SYN` that was just received
    * AKA the SYNACK message
    * confirmation while serving as another one-way connection request
  - client then receives SYNACK indicating server is live, and sends an ACK for the SYNACK
    * this segment may contain *piggybacked* client to server data
  - server then receives ACK indicating client is live
  - the usual retransmission rules apply to TCP connection setup messages
- teardown:
  - here, requires **4-way handshake** to close connection
    * broken down into two pairs of two-way handshakes
    * but in certain scenarios, when both sides have no more data to communicate, can combine two messages together to create another 3-way handshake
  - client and server each close their one-way side of communication
    * send a TCP segment with `FIN` bit = 1 (finished)
    * indicates side will no longer send data, but can still receive
  - respond to FIN with ACK
    * on receiving FIN, ACK can be combined with own FIN
  - note that one side may not *immediately* close after ACKing a received FIN
    * will wait to ensure that the other host has received the corresponding ACK, and resend the final ACK in case it is lost
    * ie. a **time-wait** state
  - simultaneous FIN exchanges can be handled
- note that SYN and FIN messages without any data payload take logically one byte
  - reflected in the sequence numbers
- TCP reset:
  - the `RST` bit is set in a TCP segment response if the destination port of a previous segment does not have an associated socket
  - indiciates source not to resend the original segment because the destination is invalid

Congestion Control Principles

- **congestion** stems from too many sources sending too much data too fast for network to handle
  - when a majority of Internet resources are occupied

- *manifestations*:
  - lost packets from buffer overflow at routers
    - when dropped packets are detected, can assume caused by congestion
  - long delays from queueing at router buffers
- different from congestion *avoidance*:
  - congestion *control* deals with congestion, not avoiding it
  - in reality, some degree of congestion is unavoidable
- different from flow control:
  - limiting based on network's capabilities, not receiver's capabilities

- two broad approaches:
  - **end-to-end** congestion control:
    - no explicit feedback from network
    - congestion inferred from observed loss or delay in the end systems
    - approach taken by TCP
  - **network-assisted** congestion control:
    - routers provide feedbacks to end systems
    - eg. a single bit indicating congestion
    - eg. an explicit rate sender should send at
      - usually too expensive of an operation for routers to provide
- TCP congestion control:
  - *idea*:
    - assumes best-effort network
    - each source determines network capacity for itself
      - independent of other routers or hosts
    - implicit feedback via ACKs or timeout events
    - ACKs pace transmission, self-clocking
  - *challenges*:
    - determining *initial* available capacity
    - adjusting to changes in capacity in a *timely* matter

Congestion Window

- using a congestion window:
  - congestion control revolves around reducing the sender window size
    - equivalent to reducing the sender rate
  - add notion of the **congestion** window
    - dynamic size, function of perceived network congestion
  - *effective* window is the minimum of:
    - the advertised window `rwnd` for flow control
    - the congestion window `cwnd` for congestion control
  - changes in congestion window size:
    - slowly increases (in absence of congestion) to absorb and maximize

new bandwidth
    * quickly decreases (in presence of congestion) to quickly eliminate congestion
- perceiving congestion at the sender:
  - note that packet loss *does not* always imply congestion
    * eg. routers can drop corrupt packets
    * but TCP makes the assumption that packet loss implies congestion
  - timeout or 3 duplicate ACKs indicate a loss event
    * after a loss event, sender should reduce the congestion window size

Mechanisms

- different mechanisms are required for congestion control:
  - AIMD, to grow and shrink `cwnd`
    * shrinking triggered by duplicate ACKs
  - slow-start, to initialize and grow `cwnd` at startup
  - conservatively reducing sending window after timeout events
    * on *heavy* congestion, which is indicated by timeout rather than duplicate ACKs, reset everything
- **additive increase, multiplicative decrease (AIMD)**:
  - increase tranmission rate ie. window size, probing for usable bandwidth, until loss occurs
    * fundamental golden rule for congestion control
  - additive increase:
    * increase `cwnd` by 1 **maximum segment size (MSS)** every RTT until loss detected
  - multiplicative decrease:
    * cut `cwnd` by 50% after loss
  - leads to a characteristic sawtooth behavior as sender *probes* for bandwidth
  - AIMD rule is *motivated* by fairness:
    * other rules, eg. multiplicative increase, exponential decrease, do not guarantee TCP fairness
    * with multiple connections sharing a link, AIMD allows for fair sharing
      · because of the multiplicative decrease, every connection is periodically reduced *proportionally* in half
      · over time, approaches an equal bandwidth share
    * eventually, each connection should have $\frac{1}{N}$ of the bottleneck link bandwidth
  - implementation wise, additive increase is implemented by the **congestion avoidance** algorithm
    * while multiplicative decrease is implemented by the **fast retrans-**

mission and **fast recovery** algorithms
- **slow-start** algorithm allows for TCP to jump-start the `cwnd` at startup:
  - want to quickly ramp up connection speed to operable rate
  - when connection starts, `cwnd` set to 1 or 2 MSS
    - in modern TCP, as high as 4 MSS
    - bandwidth is equal to $cwnd \times MSS/RTT$
  - then, aggressively increase rate *exponentially* fast until `cwnd` reaches a threshold value ie. the slow-start threshold `ssthresh` :
    - occurs while `cwnd ≤ ssthresh`
    - *double* the `cwnd` over every RTT:
      - to implement, increment `cwnd += 1*MSS` for every ACK received
    - slow-start is a misnomer
  - using `ssthresh` to deliniate when to switch over to linear increase:
    - optimally `ssthresh` should be half of the maximum window size
    - in TCP:
      - initialized by educated guess, usually set by OS
      - on a loss event, `ssthresh` will be set to half of `cwnd` just before loss event
- **congestion avoidance** algorithm applies the additive increase from AIMD:
  - increases `cwnd` by 1 MSS per RTT until loss is detected:
    - occurs while `cwnd > ssthresh` , after slow-start initialization
    - to implement, `cwnd += 1/cwnd * MSS` where `MSS` is in bytes for every *non-duplicate* ACK received
- **fast retransmit** algorithm detects and repairs loss:
  - based on incoming duplicate ACKs:
    - uses triple duplicate ACKs to infer losses and differentiate from transient out-of-order delivery
    - do nothing on 1 or 2 duplicate ACKs
  - occurs after triple duplicate ACKs
  - to implement:
    - `ssthresh = max(cwnd/2, 2*MSS)`
    - `cwnd = ssthresh + 3*MSS`
    - retransmit lost packet
  - implements multiplicative decrease, but with an *additional* 3 MSS
    - extra 3 MSS is due to having detected the loss *late* because of waiting for 3 duplicate ACKs
    - ie. a delay factor, since receiver has received 3 extra segments that triggered the duplicate ACKs
    - thus there are still *only* `ssthresh` packets circulating in the internet
- **fast recovery** algorithm governs transmission of new data until a non-duplicate ACK arrives:

  - occurs while there are greater than 3 duplicate ACKs, until a non-duplicate ACK arrives
  - to implement:
    * increase `cwnd` by 1 MSS upon every duplicate ACK
    * similar motivation as the delay factor in fast retransmit
      · still does not increase the overall number of packets in the internet
  - after fast restransmit and fast recovery completes, `cwnd` should be decreased to match `ssthresh`
  - once non-duplicate ACK arrives:
    * set `cwnd = ssthresh`
  - note that as fast recovery ends, either slow-start or congestion avoidance will kick in on the *same* ACK
- **retransmission timeout** occurs when the retransmission timer triggers:
  - the entire system should be reset because *heavy* loss has been detected due to the timer
    * as opposed to receiving three duplicate ACKs
  - to implement:
    * `ssthresh = max(cwnd/2, 2*MSS)`
    * `cwnd = 1*MSS`
    * retransmit lost packet

Network Layer

- the **network layer** provides a best-effort *global* packet delivery
  - transports segments from sending to receiving host
    - encapsulate segments into datagrams on sending side
    - deliver extracted segments to transport layer on receiving side
  - running on every end host as well as every router
    - effectively in every Internet device
  - simple and *best-effort*, does not provide any loss, order, timing, or delivery guarantees
    - alternative, previously considered network architectures such as ATM did provide some guarantees
  - two basic functions:
    - **forwarding** moves packets from router's input to appropriate router output
      - hardware implemented, takes place on very short timescales of a few nanoseconds
    - **routing** determines the route taken by packets from source to destination
      - software implemented, takes place on much longer timescales of a few seconds
  - two fundamental components:
    - the **data plane** deals with the local, per-router forwarding functions
    - the **control plane** deals with network-wide routing of datagrams
      - coordinates individual per-router functions together
- in a decentralized, *per-router* control plane:
  - individual routing algorithm components in each and every router interact in order to perform network-wide routing
  - a *monolithic* implementation of the data plane and control plane in each router
- alternatively, in **software-defined networking (SDN)**:
  - control plane functions are implemented as a separate *service*, in remote servers
    - the control interacts with local router control agents
  - creates a network-wide, *logically* centralized control logic
  - a *separated* implementation between data and control plane
    - each router performs forwarding only
    - remote controller computes and distributes forwarding tables

Router Architecture

- a **router** is made up of:
  - data plane components that handle data forwarding:
    - hardware components, operating in nanosecond timeframe
      - hardware implementation is necessary for such strict speed requirements
    - **input** and **output ports**
      - number of inputs may be different than outputs
    - high-speed **switching fabric**, connecting input and output ports
  - control plane components that handle routing:
    - software components, operating in millisecond or slower timeframe
    - **routing processor**
      - maintains and computes a forwarding table
- **input port** functions:
  - line termination ie. physical layer bit-level reception
  - link layer protocol interoperation
  - lookup, forwarding, queuing:
    - using header fields, extract destination IP, and use a forwarding table to map input to output ports
      - note that some packets like control packets will be forwarded to the routing processor, rather than an output port
    - goal of decentralized switching, for each input port
    - queuing occurs if datagrams arrive *faster* than the forwarding rate into switching fabric
  - other operations such as:
    - checking packet's version number, checksum, TTL field
    - updating network management metadata
- the **forwarding table**:
  - maps a destination address range to a link interface
    - as well as provides a default mapping
    - router matches a *prefix* of the destination address with the table entries
  - the *longest* address prefix that matches the destination address should be used
  - typically uses **ternary content addressable memories (TCAMs)** that provide quick address retrieval, *regardless* of table size
  - generalized in the **match-plus-action** paradigm:
    - *match* header fields
    - perform some specific *action*, such as:
      - forward to output ports

        · load balance across multiple interfaces
        · rewrite header values (NAT)
        · block packets (firewalls)

- **switching fabrics** transfer packets from input buffer to appropriate output buffer:
  - the **switching rate** is the rate at which packets can be transferred
    * for N inputs, a switching rate of $N \times$ *linerate* is desirable
- three types of switching fabrics:
  1. **memory** switching:
     - used with traditional, first-generation routers
     - packet is copied to system's memory, under direct CPU control
       * ports act as traditional I/O devices
     - speed limited by memory bandwidth, 2 bus crossings per datagram
     - only one packet can forward at a time, since only one memory access can occur at a time
  2. **bus** switching:
     - packet moved from input port memory to output port memory via a shared bus
       * without CPU intervention
       * all ports on the shared bus see the packet, but only one port will accept it
     - speed limited by bus bandwidth
     - only one packet can cross the bus at a time
  3. **interconnection network** switching:
     - goal of overcoming bus bandwidth limitations
     - connect processors in a multiprocessor together using different networks, eg. crossbar, banyan networks
       * eg. in a crossbar switch, use $2N$ buses to connect N input and output ports
     - can also fragment datagram into fixed length cells for even more parallelization
     - multiple packets can be forwarded in parallel
- **output port** functions:
  - analagous to input port, composed of:
    * buffer, link layer, line termination
  - buffer handles queuing caused from a faster fabric rate
  - also handles scheduling ie. choosing next outgoing packet, eg. using priority scheduling
- **queuing**:
  - delay and loss may occur due to queuing
  - input port queuing occurs when the fabric is slower than the input ports combined
    * AKA **head of the line (HOL)** blocking where queued datagram at

front of queue prevents others in queue from moving forward
  - output port queuing occurs when arrival rate via fabric exceeds output line speed
  - how much **buffer space** should be allocated to avoid overflow?
    * average buffering is $RTT \times C$, where C is the link capacity
    * in a more recent recommendation, with N flows, average buffering is $\frac{RTT \times C}{\sqrt{N}}$
- **scheduling mechanisms**:
  - ie. choosing next packet to send on link
  - most popular is **first in**, **first out (FIFO)** scheduling
    * send in order of arrival to queue
  - also priority, round robin, and weighted fair scheduling
  - different **discard policies** if a new packet arrives to full queue:
    * *tail drop*: drop arriving packet
    * *priority drop*: drop on a priority basis
    * *random drop*: randomly drop packet

IP

- the **Internet protocol (IP)** is the key network layer protocol, and encapsulates:
  - addressing conventions
  - datagram format
  - packet handling conventions
- IP datagram format:
  - basic header:
    * 5 rows, 4 bytes wide
    * protocol version, header length, type of service, length
      · dominant protocol version is IPv4, type of service field not commonly used
      · length specifies the *total* datagram length in bytes, so largest IP packet is 2^{16} bytes
    * 16-bit ID, 3-bit flags, fragment offset
      · flags and fragment offset are used for fragmentation and reassembly
    * time to live, upper layer protocol, header checksum
      · TTL is essentially the max number of remaining hops, decremented at each router
      · TTL was introduced to bypass infinite **looping delivery** over the same routers
      · upper layer protocol indicates which protocol to deliver pay-

load to
          · checksum for bit error detection, computed only over the header
    * 32-bit source IP address
    * 32-bit destination IP address
    * note that with TCP and IP headers together, there is already 40 bytes of header overhead
        · payload data should be quite large to amortize the overhead, often 1000 bytes
  - options header ie. optional, extended IP header section
    * may include timestamp, record route taken, or specify list of routers to visit
  - payload data of variable length

Fragmentation

- IP **fragmentation** and **reassembly**:
  - the global Internet is built over many different physical networks
    * IP must *glue* together these physical networks and their different technologies
    * different networks can accommodate different IP datagram sizes, eg. optical link vs. wifi networks
  - each network link has a **max transfer size (MTU)**, indicating the capped, largest possible link level frame
    * different MTUs for different link types
  - thus a large IP datagram will be divided and *fragmented* within the network if it is larger than the next hop's MTU:
    * one datagram becomes *several* datagrams
        · note that this introduced additional overhead with extra headers being used
    * *reassembled* only at the final destination, by the end systems rather than network routers
        · should not reassemble at the next router after being fragmented, in case another fragmentation may be required
    * note that fragmenting multiple times over is rare
    * IP header bits used to identify and order related fragments
- using IP header for fragmentation:
  - `ID` is the same for fragments of the same original datagram
  - 3-bit flags for `0 donotfrag morefrags`, and a 13-bit fragment `offset` field
  - `donotfrag` (DF) indicates not to fragment
  - `morefrags` (MF) indicates there are more fragments following this datagram

    * set to 0 for the last fragment
  - `offset` indicates relative positioning of fragment in terms of 8 bytes
    * if data cannot be evenly divided in 8, just transfer in the largest granularity divisible by 8, with the remainder in the final datagram
  - eg. a 4000 byte datagram going through a link with MTU of 1500 bytes is fragmented in three:
    * length 1500 with `ID = x`, `fragflag = 1`, `offset = 0`
      · 1480 bytes of data, or 185 8-byte units of offset
    * length 1500 with `ID = x`, `fragflag = 1`, `offset = 185`
    * length 1040 with `ID = x`, `fragflag = 0`, `offset = 370`

IP Addresses

- IP addressing:
  - an **IP address** is a 32-bit identifier for host and router interface
  - an **interface** is a connection between a host or router and a physical link
    * routers typically have multiple interfaces
    * host typically has one or two interfaces, eg. wired Ethernet and wireless 802.11
    * IP addresses are associated with *each interface*, rather than each host or router
- IP address is composed of parts:
  - the **subnet part** or higher order bits:
    * a device interfaces directly with the same **subnet** part of IP address
      · ie. can physically reach each other *without* intervening router
      · thus to determine the subnets, detach each interface from its host or router, creating isolated networks or subnets
    * the **subnet mask** `/24` indicates the higher 24 bits are used for the subnet ID
    * over time, **classless InterDomain routing (CIDR)** now allows subnet portions of arbitrary length
      · more flexible than the previous fixing of the ID to 8, 16, 24 bits in *classful* addressing
  - the **host part** or lower order bits
  - dividing into parts allows routing protocols to focus only on parts of IP address such as the subnet part alone
    * greatly reduces the size of the forwarding table
- how does a *host* get an IP address?
  1. *hard-coded* by system admin in a file or setting:
     - Windows control panel, UNIX `/etc/rc.config`
  2. **dynamic host configuration protocol (DHCP)** dynamically gets address from a server:
     - automated assignment, less reliant on system admin

        * ie. a *plug-and-play* or zero configuration protocol
    - allow host to *dynamically* obtain IP address from network server when it joins network
        * also allows reuse or holding of address while host is connected
    - DHCP is an application layer protocol, running over UDP
- DHCP overview:
    - a client-server protocol
    - host broadcasts `DHCP discover` to find a DHCP server, DHCP server responds with another broadcasted `DCHP offer`
        * `DCHP offer` contains a `yieldaddr` field for the offered IP address, and `lifetime` field for the lease time of the IP address
    - host requests a specific IP address `DHCP request`, DHCP server sends address `DHCP ack`
        * because a host may receive IP offers from multiple DHCP servers, the `DHCP request` indicates that a host has chosen to accept one of the offers and use one specific IP address
        * `DHCP request` is also broadcast to notify all the DHCP servers of the host's choice
        * the host cannot use their offered IP address until receiving the `DHCP ack` confirming their new IP
    - by default, host address and DHCP server address is unkown, so they are set to placeholders:
        * `0.0.0.0.x` represents a *local* source address
        * `255.255.255.255.x` represents a *broadcast* address that reaches all hosts in the same subnet
    - DHCP can also return other information:
        * address of first-hop router for client
        * name and IP address of DNS server
        * network ie. subnet mask
    - note that a new IP address is obtained from DHCP each time a node connects to a new subnet
        * so a TCP connection to a remote application cannot be maintained as a mobile node moves between subnets
- how does a *network* get subnet part of an IP address?
    - it gets allocated a portion of its provider ISP's address space
    - another example of hierarchical addressing:
        * Internet routers can just use the ISP address space portion to route to all underlying address spaces, rather than routing to the underlying address spaces separately
            · this **route aggregation** allows routing to scale and improves performance
            · ie. no need for a forwarding table entry for every underlying organization, but instead they can share an aggregated route

specified by the overall ISP subnet mask
    * after arriving in the ISP network, it can then be relayed within the ISP network to specific organizations and subnets
  - eg. an ISP may have subnet mask `20`, and provide for 8 organizations with masks `23`
- how does an *ISP* get a block of addresses?
  - the **Internet Corporation for Assigned Names and Numbers (ICANN)** has the following functions:
    * allocates addresses
    * manages DNS
    * assigns domain names, resolves disputs
  - however, the address blocks available in IPv4 are running out:
    * longterm solution is to switch to IPv6, which supports a greater range of addresses
    * shortterm solution has been to use NAT

NAT

- in **network address translation (NAT)**, all leaving datagrams have the same single source IP, but with different port numbers:
  - ie. the local NAT router maps the local IP address to the global public IP address
  - the local unique IPs are *private* and not directly visible to the outside world
  - *motivation*:
    * local network uses just one public IP
      · difficult for different private addresses to distinguish themselves publicly
    * a range of addresses from ISP is not needed
  - there are already dedicated address spaces for NAT:
    * `100.64.0.0/10` for carrier grade NAT:
      · 4 million addresses
      · used for carrier networks
    * `10.0.0.0/8` and `192.168.0.0/16` are commonly used for private IP address spaces
  - *pros*:
    * helps solve issue of dwindling IP addresses
    * can change addresses of devices in LAN without notifying outside world
    * can change ISP without changing device addresses in LAN
    * devices inside LAN are not explicitly addressable or visible by outside world
      · privacy and security advantages

  - *cons*:
    * routers should only process up to the network layer, but NAT routers also function in the transport layer to modify the port number
      · not a purely network layer solution
    * port number should be used for addressing processes
    * has greatly delayed IPv6 adoption, which is a truer, longterm solution
    * violates end-to-end agreement
      · possibility of NAT must be taken into account in certain applications, such as P2P applications
    * NAT traversal problem, ie. client wants to connect to a server behind NAT
- NAT *implementation*:
  - for outgoing datagrams, replace `(src addr, src port)` with `(NAT addr, new port)`
  - in NAT router **NAT translation table**, remember every `(src addr, src port)` to `(NAT addr, new port)` tuple translation
  - for incoming datagrams, replace with the opposite translation
  - with a 16-bit port number, there can be 60000 simultaneous connections with a single LAN-side address
- problem of **NAT traversal**:
  - if a client wants to connect to a server behind NAT, the server IP address is not publicly accessible, and there is only one single visible external NAT address
  - various solutions:
    1. statically configure NAT to forward incoming connection requests at given port to server:
       * statically allocting a port number to the given server
    2. use a **Universal Plug and Play (UPnP) Internet Gateway Device (IGD)** protocol:
       * learn public IP address, and add and remove port mappings with lease times
       * ie. automate static NAT port map configuration
    3. use relaying:
       * NATed host establishes connection to relay
       * external client connects to relay
       * relay bridges packets between connections
  - breaks the universal internet guarantee that given an IP address, you can communicate with another device
    * with NAT, you can no longer directly communicate with devices behind a NAT

IPv6

- **IPv6** is the newest version of the Internet protocol
  - not yet very widely adopted
  - *motivation*:
    * 32-bit address space soon to be completely allocated
    * header format changes to improve speed of processing and forwarding
    * header changes to facilitiate quality of service (QoS)
  - important changes:
    * expanded addressing capabilities
      · new **anycast** type of address that allows a packet to be delivered to any one of a group of hosts
    * streamlined 40 byte header
    * flow labeling to categorize packets into different flows
- IPv6 datagram format:
  - fixed-length 40 byte header, each row 4 bytes wide
  - version, priority, flow label
    * priority ie. traffic class field identifies the priority among datagrams in the flow
    * flow label identifies datagrams in the same flow ie. stream
      · new field for future Internet functionality
  - payload length, next header, hop limit
    * next header identifies the upper layer protocol, but can also be a pointer to an options field
    * hop limit or hop count is the same as the IPv4 time to live field
  - 128-bit, 4-row source address
  - 128-bit, 4-row destination address
- major header *changes* from IPv4:
  - header size twice as large without options
    * but addresses are four times as long
  - no fragmentation allowed
    * related fields removed
  - checksum field removed:
    * removed entirely to reduce processing time at each hop
    * data corruption is more rare than before
    * rely entirely on link layer error detection
  - variable length options field removed
    * options are still allowed using next header field as a pointer to header options
- *transitioning* from IPv4 to IPv6:
  - issue of deployment:
    * according to Google, 8% of clients access services via IPv6

        * according to NIST, 1/3 of all US government domains are IPv6 capable
    - not all routers can be upgraded simultaneously:
        * how will network operate with *mixed* IPv4 and IPv6 routers?
        * eg. traversing through an IPv4 *tunnel* that connects IPv6 routers
    - solution of using **tunneling**:
        * carrying an IPv6 datagram as a *payload* within an IPv4 datagram
            · the source and destination of the IPv4 datagram are the IPv6 routers at the start and end of the IPv4 tunnel
        * IPv6 routers on either side of the IPv4 tunnel thus *must also* run IPv4
            · note that these IPv6 routers are *neighbors* in the logical view of IPv6
            · thus routers need to know which of their neighboring nodes are IPv6 as IPv6 is deployed further
            · even though they are not *physical* neighbors via IPv6
    - technique of tunneling can be used in other scenarios to deploy services on the Internet
    - enormously difficult to change network layer protocols

Routing

- **routing** or determining the route taken by packets from source to destination is the central **control plane** functionality
    - different approaches:
        * traditional, per-router control
        * software defined, logically centralized control (SDN)
- *goal* of the routing protocol:
    - determine good paths from sending hosts to receiving hosts, through a network of routers
    - a **path** is a sequence of routers packets will traverse in
        * can regularly involve 50 hops
    - a *good* path is characterized by the one with least cost, fastest speed, or least congestion, depending on the criteria
        * path cost is the sum of the traversed links' costs
- can abstract the network as a graph:
    - routers as nodes, links as edges, costs as weights
        * the *cost* of each edge may be associated with its phsyical, length congestion, delay, etc.
            · or just a standard fixed cost of 1 for all edges
            · different criterias for link costs

  - want to *minimize* the cost of the path taken

Algorithms

- routing algorithm classifications:
  - **global** and centralized:
    - all routers have to know the *complete* topology and link costs
    - *link state* algorithms eg. Dijkstra's algorithm
  - **local** and decentralized:
    - router knows physically connected neighbors and link costs to neighbors only
    - iterative process of computation, exchanging info with neighbors
    - *distance vector* algorithms eg. Bellman Ford algorithm
  - **static**:
    - routes change slowly over time
  - **dynamic**:
    - routes change more quickly, in response to traffic load or topology changes
      - periodic updates, in response to link cost changes
    - more responsive, but more susceptible to problems such as loops and oscillations
- **Dijkstra's algorithm**:
  - global routing algorithm
  - net topology and link costs must be known to all nodes
    - accomplished via a **link state broadcast**
  - computes least cost paths from one source node to *all other* nodes
    - generates the **forwarding table** for that source
  - an *iterative* algorithm:
    - after k iterations, we know least cost path to k destinations
    - each iteration *finalizes* the distance to one new node
  - runs in $O(n^2)$ trivially for n nodes, or $O(nlogn)$ using heap optimizations
  - potential issue of *oscillation*:
    - eg. if link costs equals amount of carried traffic
    - best path betwen two nodes oscillates between similar options, as the slected path cost will increase by the current carried traffic
    - need to choose a cost measure that isn't dependent on dynamic variables like carried traffic or latency
      - eg. for Internet, every routers' cost is the hop count, so the link cost is always 1
      - but dynamic link cost is useful for load balancing
    - another solution would be to ensure that not all routers run their routing algorithms at the same time, which is difficult to guarantee

Dijsktra's algorithm pseudocode:

```
% c(x,y) := link cost from node x to y, infty if not neighbors
% D(v)   := current value of cost of path from source to destination v
% N'     := set of nodes with finalized least cost path

N' = {u}
for all nodes v:
  if v adjacent to u
    then D(v) = c(u,v)
  else D(V) = infty

loop until all nodes in N':
  find w not in N' such that D(w) is a minimum
    add w to N'
    update D(V) for all v adjacent to w and not in N':
      D(v) = min(D(v), D(w) + c(w,v))
```

- **Bellman-Ford algorithm**:
  - dynamic programming solution
  - a **distance vector (DV)** for node v is the list of *estimates* of least cost from v to every other node in the network
    - only an estimate since the DVs are built locally, in response to updates from neighbors
  - each node knows:
    - its own DV
    - the cost to each of its neighbors
    - each of its neighbors' DV
  - node functions:
    - periodically, each node will send its own DV to neighbors
    - when it receives a new DV from a neighbor, it updates its own DV using the Bellman-Ford equation
  - node behavior:
    - *waits* for asynchronous changes in the local link cost or message from neighbor
    - *recompute* estimates
    - if distance vector to any destination has changed, *notify* neighbor
  - thus networks are using a *distributed* implementation of Bellman-Ford algorithm:
    - where each node notifies neighbors only when its DV changes
    - but each node only has to perform local computations among its neighbors
  - this distributed implementation has the following attributes:

- *asynchronous* since nodes do not have to operate in lockstep with each other
- *iterative* since the algorithm runs until no more information is exchanged between neighbors
- self-terminating

Bellman-Ford pseudocode:

```
% d_x(y) := the cost of least-cost path from x to y
% d_v(y) := the cost of least-cost path from v to y
% min_v  := the minimum taken over all neighbors v of x
% c(x,v) := the cost to neighbor v

for all nodes y:
  if y adjacent to x
    then d_x(y) = c(x,y)
  else d_x(y) = infty

for all node y:
  d_x(y) = min_v {c(x,v) + d_v(y)}
```

Distance-Vector algorithm pseudocode:

```
% D_x    := the distance vector for node x
%        := {D_x(y) : y in N}
% D_x(y) := the cost estimate from x to y in x's distance vector
% min_v  := the minimum taken over all neighbors v of x
% F_x(y) := forwarding table entry for destination y from x

for all nodes y:
  if y adjacent to x
    then D_x(y) = c(x,y)
  else D_x(y) = infty
for each neighbor w of x:
  for all nodes y:
    initialize D_w(y) to ?

loop:
  when there is a link cost change to some neighbor
    or a distance vector is received from some neighbor:
      for all nodes y:
        D_x(y) = min_v {c(x,v) + d_v(y)}
        F_x(y) = neighbor v from above that achieved minimum cost
    if D_x(y) changed for any y:
```

```
    send D_x to all neighbors
```

- reacting to link cost changes:
  - *good news travels fast*:
    - when link cost decreases, few iterations are needed for the distance vector to *converge* among the nodes
    - because DV records the minimum path, any DV changes will be at *least* as good as the previous one
      - no looping computations, since old routes ie. previous computations will still hold
  - *bad news travels slow*:
    - on the other hand, when link cost increases, *many* iterations may be needed before algorithm stabilizes
    - a long cost path will only be updated incrementally:
      - other nodes may be *unaware* of the direct change in cost, since the distance vector only records the minimum cost path
      - so nodes will select to go through a *previous* ie. stale minimum cost path, whose cost will incrementally grow until it becomes reflected by the change in link cost
      - neighboring nodes affected by the increased link cost will then repeatedly update each other's DVs in a loop
      - this is a **routing loop**
    - the delay in link cost reacting is due to the algorithm updating in a decentralized manner
      - each node always performs calcuations *with respect to* the DVs of its neighbors
      - and neighboring DVs may be stale
    - ie. the **count-to-infinity** problem
      - major problem with distance vector algorithm
      - many networks have begun to phase out distance vector algorithm and instead use link state algorithm
  - potential heuristic to solve problem:
    - in *poisoned reverse*, if a node x routes through another node y to get to node z:
      - z will advertise its own distance to x as infinity
    - but does not completely solve count-to-infinity problem for large routing loops
- comparison of link state (LS) and distance vector (DV) algorithms:
  - *message complexity*:
    - LS needs to know global topology, so with n nodes and e links, $O(ne)$ messages sent
      - need to *flood* messages so every node is aware of every other node and link

    * DV exchanges messages containing link cost and DVs between neighbors only
        · but DV messages may be very large
  - *speed of convergence*:
    * LS is an $O(n^2)$ or $(Onlogn)$ algorithm
        · may have oscillations with dynamic link cost
    * DV has *varying* convergence time
        · issue of count-to-infinity problem
        · may also have oscillations with dynamic link cost
  - *robustness* if a router malfunctions:
    * with LS, node can advertise incorrect *link* cost
        · but each node computes only its *own* table
    * with DV, node can advertise an incorrect *path* cost
        · triggering a global negative impact since each node's table is used by others
        · ie. error propagation through network
  - routing protocols like OSPF and NLSP use a link state algorithm
  - routing protocols like RIP, BGP, and IGRP use a distance vector algorithm

Hierarchical Routing

- routing has been previously *idealized*, where all routers are identical
  - ie. a *flat* network, which is not true in practice
  - with a scale of 600 million destinations, cannot store all destinations in routing tables
    * routing table exchange would swamp links
  - instead, consider an *administrative* autonomy approach:
    * Internet is a network of networks
    * each network admin (eg. ISP) may want to control routing within its own network
- using **hierarchical routing**:
  - aggregate routers into regions called **autonomous systems (AS)**
    * usually, each IP can have one or more AS
    * routers in the same AS run the same routing protocol ie. *intra*-AS routing
    * routers in different AS can run a different intra-AS routing protocol
    * a **gateway router** or **border router** is at edge of its own AS and has a link to router in another AS
  - AS are denoted by their unique **AS number (ASN)**
    * 16-bit numbers, denoting units of routing policy
  - AS are well connected to one another
  - forwarding table will be configured by intra-AS as well as *inter*-AS rout-

ing algorithms
    * intra-AS routing will set entries for internal destinations, lower level of routing
        · ie. describing hop by hop, how to reach an IP prefix destination in the same AS
    * inter-AS routing will set entries for reachable external destinations
        · ie. describing the intermediate AS to travel through to reach an IP prefix destination
  - inter-AS routing tasks:
    * upon receiving a datagram destined outside of the AS, router should forward packet to gateway router, but which one?
    * *learn* which destinations are reachable through different neighboring AS
    * *propagate* this reachability info to *all* routers within the AS
- setting forwarding tables within AS:
  - suppose AS1 learns from its inter-AS protocol that subnet X is reachable via AS3 but not AS2
  - the inter-AS protocol will propagate reachability info to all internal routers
  - an individual router in AS1 will determine from its intra-AS routing info to install a new forwarding table entry to reach X:
    * suppose its interface `L` is the one in the least cost path to the gateway router in AS1 connected to AS3
    * installs forwarding table entry `(X, L)`
- choosing among multiple AS:
  - suppose AS1 learns from its inter-AS protocol that subnet X is reachable via AS3 and AS2
  - the inter-AS protocol will propagate reachability info to all internal routers
  - an individual router in AS1 must determine which gateway it should forward packets for destination X again using intra-AS routing:
    * eg. **hot potato routing**: send packet towards the minimal least-cost path of the two routers
    * suppose its interface `L` is the one along the minimal least cost path
    * installs forwarding table entry `(X, L)`

Intra-AS Routing Protocols

- intra-AS routing protocols:
  - AKA **interior gateway protocols (IGP)**
  1. **routing information protocol (RIP)**
     - phasing out usage due to count-to-infinity problem

2. **open shortest path first (OSPF)**
   - AKA **IS-IS protocol**
   - most commonly used
3. **interior gateway routing protocol (IGRP)**
   - Cisco proprietary

- OSPF:
  - open ie. *publicly* available
  - uses link-state algorithm:
    * link state packet dissemination
    * topology map at each node
    * route computation using Dijkstra's algorithm
    * all link costs may be set to 1, or inversely proportional to link capacity
  - router floods OSPF link-state advertisements to all other routers in *entire* AS
    * OSPF messages delivered directly over IP, rather than transport layer protocol
    * messages broadcasted periodically, and whenever there is a change in a link's status
  - concerns about *speed* and scaling:
    * since OSPF floods all routers, OSPF should not be used for very large networks
  - note that for intra-AS protocols:
    * there is only a single administrator, so no policy decisions are needed
    * can focus on performance over policy
  - *pros*:
    * security using authentication between OSPF routers
    * multiple same-cost paths
    * integrated support for unicast and multicast routing
    * hierarchical support
- another option is to use *hierarchical* OSPF instead:
  - hierarchical routing saves table size, and reduces update traffic
  - divide the routers in a large network into **areas**
    * routers in an area are **internal routers**
  - multiple areas are connected together through a **backbone network**
    * backbone network runs a *different* level of OSPF ie. link-state advertisements propagated within only one hierarchy
  - areas are connected to the backbone through **area border routers**
    * these routers will *summarize* distances to networks in their own area, and advertise to other area border routers
  - **backbone routers** run OSPF routing limited to the backbone
  - **boundary routers** then connect to other AS

Inter-AS Routing Protocols

- the **border gateway protocol (BGP)** is the *de facto* inter-domain routing protocol
  - "glue that holds Internet together"
    - all communicating AS must run the same inter-AS routing protocol
    - all Internet AS use BGP
  - provides each AS with mechanisms to:
    - **eBGP**: obtain subnet reachability information to neighboring AS
    - **iBGP**: propogate reachability information to all internal routers
      - gateway routers will have to run both eBGP and iBGP
    - determine *good* routes to other networks
    - allows subnets to advertise their existence to the Internet
  - uses distance vector algorithm
    - BGP destinations are CIDRized prefixes
  - note that for inter-AS protocols:
    - each administartor wants control over how its own traffic is routed, so policy-based routing is used
    - in fact, policy may dominate over performance
- BGP routers ie. gateway routers exchange messages between other AS and internal routers:
  1. using TCP on port 179
  2. exchange all active routes
     - ie. advertising paths to different destination network prefixes
     - BGP is a distance vector protocol, specified in granularity of AS using ASNs
     - eg. path `AS3, X` AS3 promises to its neighbor AS2 that it forwards packets towards prefix destination X
       - AS2 will then advertise the path `AS2, AS3, X` to its neighbor AS1
       - after advertising the path `AS3, X` to all internal routers in AS2 (iBGP)
  3. repeatedly, continue exchanging incremental updates
     - and updating internal routers
  - the advertised prefix path also includes certain BGP attributes:
    - `AS-PATH` indicates all the intermediate AS to reach the prefix destination
      - ie. AS-level path, used by border routers
    - `NEXT-HOP` indicates just the specific internal router towards the next-hop AS to reach the prefix destination
      - ie. internal path, used by internal routers
  - BGP message types:
    - `OPEN` opens TCP connection to peer while authenticating sender

    * `UPDATE` advertises new path
    * `KEEPALIVE` keeps connection alive in absence of updates, acts as ACK
    * `NOTIFICATION` reports errors in previous message, closes connection
- BGP uses *policy-based* routing:
  - BGP routers will accept and reject paths based on policies, and then advertise their own accepted paths
    * eg. selecting between multiple routes to the same prefix (could actually reject them all)
  - some policies include:
    1. never routing through certain AS
    2. preferring routes through a collaborating AS
    * competitors will not advertise paths through other competitors
    * ISPs only want to route to and from their customers, as much as possible
  - route selection criteria:
    * route selection is performed in the following order:
    1. local preference decision on policy
    2. shortest `AS-PATH`
    3. closest `NEXT-HOP` router ie. **hot potato routing**
       * choose local gateway with least intra-domain cost, and ignore inter-domain cost
       * a *selfish* algorithm that simply seeks to reduce the cost in its own AS
    4. additional criteria
- BGP **anycast** service for IP:
  - BGP is used to implement IP-anycast
  - IP-anycast allows routers to naturally select the best route to multiple *replicated* server copies
    * eg. CDN copies in different geographical locations
    * CDN can simply assign the *same* IP address to each different CDN server
    * BGN will then *naturally* select between the different routes using its builtin route selection

ICMP and Internet Tools

- the **Internet control message protocol (ICMP)** is used by hosts and routers to communicate network level information
  - usually used for error reporting, and different Internet tools
  - eg. unreachable host, network, port, protocol

  - network layer *above* IP, carried in IP datagrams
  - ICMP message format:
    * the type and code
      · eg. `type 3, code 0` indicates destination host unreachable, `type 8, code 0` indicates route advertisement, etc.
    * the first 8 bytes of IP datagram that caused the error
- **traceroute**:
  - source sends series of UDP segments to an unlikely port number at the destination, with increased TTL for each set
  - when the TTL drops to zero in each set of segments:
    * router will discard datagram and sends source an ICMP message with `type 11, code 0` indicating TTL expired
      · this ICMP message also includes the router name and IP address
  - when ICMP message arrives, source records RTTs
  - eventually, UDP segment will arrive at destination host
    * destination returns ICMP message with `type 3, code 3` indicating port unreachable
    * port stops sending segments
- **packet Internet groper (PING)**:
  - measures RTT and delay between two hosts
  - implemented on top of IP
  - sender sends an ICMP echo request `type 8, code 0`
  - receiver responds with ICMP echo reply message `type 0, code 0`
  - measure RTT between two messages, and output statistical summary
- **iPerf**:
  - tool for measuring maximum achievable bandwidth for IP networks
- **wireshark**:
  - Internet protocol and packet analyzer:
    * a data *capturing* and inspecting program that understands the structure of different protocols
    * can parse and display different fields, along with their meanings
    * uses a tool called **pcap** to capture packets
  - uses **network interface controllers (NiCs)** in a *promiscuous* or monitor mode
    * sees all network traffic visibile on the same interface
- **tcpdump** and **tcptrace**:
  - tcpdump runs on CLI to display TCP/IP packets being transmitted and received
    * also uses pcap to capture packets
  - tcptrace analyzes the traces from tcpdump
    * eg. elapsed time, bytes and segments, retransmissions, RTTs, throughput, etc.

- **network simulator ns-3**:
  - a discrete event network simulator:
    - models evolution of network systems through discrete events in time
    - simulation time advances in jumps from event to event
  - implements a full protocol stack
  - options to specify various different supported modules

Link Layer

- the **link layer** provides a best-effort *local* packet delivery
  - AKA over the **local area network (LAN)**, ie. to a physically adjacent neighbor over a *single* hop
    - while the network layer ties together multiple hops and LANs
  - implemented in each and every host
- link layer services:
  - **framing** or encapsulating datagram into a frame, with additional header and trailer
  - sharing a broadcast channel, multiple access
  - link layer addressing
    - using MAC or Ethernet addresses, separate from IP addresses
  - guaranteed error detection and optional correction:
    - end-to-end error detection also provided by some transport protocols
    - usually much more *powerful* than checksum error detection
    - but link layer provides guaranteed bit error detection and new bit error *correction*
  - reliable delivery:
    - between adjacent nodes instead of end-to-end
    - seldom used on fibers, but used with wireless links which have high error rates
  - flow control
    - between adjacent nodes instead of end-to-end
  - full-duplex and half-duplex:
    - with half-duplex nodes at both ends can trasmit, but not at the same time
- link layer terminology:
  - **nodes** are hosts and routers
  - **links** are communication channels that connect adjacent nodes along communication path
    - eg. wired links, wireless links, LANs
  - **frames** are link layer packets
    - encapsulate IP datagrams while adding frame header
- link layer protocols:
  - different links may transfer datagrams using *different* link protocols
    - each protocol provides different services
    - but all routers will speak the same IP *language*
  - eg. Ethernet, frame relay, 802.11
- link layer functionality usually implemented in three pieces:

   1. software, link technology specific device drivers
   2. firmware, low level chipsets
   3. hardware, IC
   - combined into an *adaptor* or **network interface card (NIC)** or chip
     * the NIC implements the link and physical layer
     * attaches into host's system buses
- communicating adaptors:
  - sending side:
    * encapsulates datagram in frame
    * adds error checking bits, flow control, rdt, etc.
  - receiving side:
    * looks for errors, rdt, flow control, etc.
    * extracts datagram, passes to upper layer

Error Detection

- additional bits are used for **redundancy** in error detection and correction, ie. the EDC field:
  - sender adds EDC field from the data
  - receiver checks data against the EDC field
    * datagram only passed to upper layer if frame passes the check
- error detection may not be 100% reliable
  - protocol may rarely miss some errors
  - larger EDC field yields better detection and correction
- different error detection algorithms:
  - *single* bit parity:
    * indicates odd or even number of 1's
    * detects single bit errors
  - *2D* bit parity:
    * data as a matrix, parity for rows and columns
    * detect *and* correct single bit errors
  - Internet *checksum*:
    * compute 16-bit sum
  - **cyclic redundancy check (CRC)**:
    * more powerful error-detection coding
      · widely used in practice
    * sender and receiver agree upon a divisor G
      · perform certain modulo 2-arithmetic to get r bit check value
      · modulo-2 subtraction is equivalent to $X XOR Y$
    * detects arbitrary bit errors less than $r + 1$ bits
    * eg. given D data, G divisor, r CRC bits, find R CRC value:

        · want $D \times 2^r \oplus R$ to be exclusively divisible by G
        · $R = \textit{remainder}\,\{\frac{D \times 2^r}{G}\}$

Multiple Access Links

- two types of links:
  - **point-to-point**
    * eg. dial-up access, Ethernet switch and host
  - **broadcast** ie. shared wire or medium
    * eg. old-fashioned shared Ethernet, upstream HFC, wireless LAN
    * must deal with possibility of *interference* or simultaneous transmissions by nodes
      · a **collision** will occur if a node receives two or more signals at the same time
      · in this case, receiver cannot retrieve every received signal
- a multiple access protocol must:
  - use a distributed algorithm to determine how nodes can share channel, ie. determine when a node can transmit
  - communication about channel must use the channel *itself*
    * no out-of-band channel for coordination
- characteristics of ideal multiple access protocols (MAC protocols):
  1. when one node wants to transmit, it can send at rate R
  2. when M nodes want to transmit, each can send at average rate $\frac{R}{M}$
  3. fully decentralized
     - no special node for coordination, no clock synchronization
  4. simple
- three broad classes of MAC protocols:
  1. **channel partitioning**:
     - divide channel into smaller *pieces* eg. time slots, frequency, code
     - allocate piece to node for exclusive use
     - share channel *efficiently* at high load
     - inefficient at low load
  2. **random access**:
     - channel is not divided, allow and subsequently *recover* from collisions
     - efficient at low load
     - at high load, overhead of detecting and recovering from collisions
  3. **taking turns**:

    - nodes take turns, but nodes with more to send can take longer turns
    - aims to combine advantages previous protocols
    - but requires some central coordination with an efficiency overhead

- channel partitioning MAC protocols:
- **time division multiple access (TDMA)**:
  - access to channel in *rounds*
  - each node gets fixed length slot ie. time slice in each round
  - unused slots go *idle*
  - similar to round robin OS scheduling
- **frequency division multiple access (FDMA)**:
  - channel spectrum is divided into frequency *bands*
  - each node assigned a fixed frequency band
    * but transmitter can remain on permanently
  - unused transmission in frequency bands again go idle
  - more expensive to implement in hardware than TDMA
- **code division multiple access (CDMA)**:
  - invented by Qualcomm
- eg. in the cable access network:
  - multiple homes are all connected to a single cable headend
  - specified by the **data over cable service info spec (DOCSIS)**
  - FDMA is used over upstream and downstream frequency channels
  - TDMA is used upstream, with some assigned slots and some slots with contention
- random access protocols:
  - when node has packet to send, transmit at full channel data rate
    * without any prior coordination
  - two or more transmitting nodes at a time is a *collision*
    * random access MAC protocols must specify how to *detect* and *recover* from collisions
- **slotted ALOHA**:
  - named from being developed in Hawaii
    * has slots, but different from TDMA
  - assumptions:
    * all frames same size, time divided into equal size slots
    * nodes start to transmit only at slot beginning
    * if 2 or more nodes transmit in slot, all nodes detect collision

  - when node obtains fresh frame, transmit in *next* slot:
    * if no collision, node can send new frame in next slot
    * if collision, node retransmits frame in each *subseqeunt* slot with some *probability* until success
  - *pros*:
    * single active node can continuously transmit at full rate of the channel
    * highly decentralized, only have to sync the slots between nodes
    * simple
  - *cons*:
    * collisions *waste* slots
    * *idle* slots
    * nodes may be able to detect collision in less than the time to transmit packet
    * require clock synchronization
  - *efficiency*:
    * probability given node has success in a slot is $p(1-p)^{N-1}$
    * probability any node has success is $Np(1-p)^{N-1}$
    * maximum efficiency after taking the limit turns out to be $\frac{1}{e} \approx 0.37$
    * channel used for useful transmissions only 37% of the time

- **pure** or **unslotted ALOHA**:

  - simpler, no synchronization
  - when frame first arrives, transmit immediately
  - collision probability increases
    * a pure ALOHA frame will overlap with *two* slots in the corresponding slotted ALOHA slots
  - efficiency is worse than slotted ALOHA, at 18%

- **carrier sense multiple access (CSMA)**:

  - before transmitting, *listen* to the channel:
    * if channel sensed idle, transmit entire frame
    * if channel sensed busy, defer transmission
  - ie. don't interrupt other transmissions
  - collisions can *still* occur, since propagation delay means two nodes may not *hear* each other's transmission
  - in a collision, entire packet transmission time is wasted

- **CSMA/CD (collision detection)**:

  - additional collision detection in addition to carrier sensing:
    * collisions detected within a short time
    * colliding transmissions are aborted, reducing channel wastage
  - detecting collisions:

    * easy in wired LANs: measure signal strengths, compare transmitted vs. received signals
    * difficult in wireless LANs: received signal strength overwhelmed by local transmission strength
  - used with Ethernet
  - *efficiency*:
    * depends on t_p, the max propagation delay between 2 nodes in the LAN, and t_t, the time to transmit max-size frame
    * efficiency approaches 1 as t_p goes to 0 and t_t goes to ∞
    * better performance than ALOHA, but still simple, cheap, and decentralized

- CSMA/CD Ethernet algorithm:

  1. network card receives datagram from network layer, creates frame
  2. if NIC sense channel idle, start frame transmission, otherwise wait until channel idle
  3. if NIC transmits entire frame without detecting another transmission, NIC is done with frame
  4. if NIC detects another transmission while transmitting, aborts and sends *jam signal*
     - jam signal ensures collided bits are long enough for every node to detect a collision
  5. after aborting, NIC enters **binary (exponential) backoff**:
     - after m collisions, NIC chooses K at random from $\{0, 1, 2, ..., 2^m - 1\}$
     - NIC will wait $k \times 512$ bit times, and returns to step 2
     - longer backoff interval with more collisions

- taking turns protocols:

  - seek to have the best of both worlds of channel partitioning and random access

- **polling**:

  - a master node *invites* slade nodes to transmit in turn
  - typically used with *dumb* slave devices
  - *cons*:
    * polling overhead
    * latency
    * single point of failure (master)

- **token passing**:

  - a control token is passed from one node to the next sequentially
  - token message

- *cons*:
    - oken overhead
    - latency
    - single point of failure (token)

LAN Addressing

- addresses:
  - the 32-bit IP address is a network layer address:
    - a *hierarchical* address that is *not* portable
      - address depends on IP subnet to which node is attached
  - the MAC (or LAN or Ethernet) address:
    - 48-bit MAC address
    - is used *locally* to get frame from one interface to another physically-connected interface
    - burned into NIC ROM, also sometimes software settable
    - a *flat* address that is portable
      - can move LAN card between LANs
- the **address resolution protocol (ARP)** is used to determine MAC from IP address and vice versa:
  - each IP node has an **ARP table**, where each table entry is a mapping `<IP address, MAC address, TTL>`
  - time to live is the time after which mapping will be forgotten, typically 20 minutes, ie. soft state mapping
    - some portable devices may *leave* the LAN
  - ARP is *plug-and-play* since nodes create their ARP tables without administrator intervention
- ARP over the same LAN:
  - A wants to send datagram to B, and B's MAC address is not in A's ARP table
  1. A *broadcasts* ARP query packet containing B's IP address
  2. B receives ARP packet, *unicast* replies to A with its MAC address
  3. A caches mapping in ARP table
- ARP to another LAN:
  - A wants to send datagram to B via router R
  - A knows B's IP and R's IP and MAC (directly connected)
  1. A creates IP datagram with IP source A, destination B
     - A creates link frame with R's MAC address as destination address, which encapsulates the IP datagram
  2. Frame sent from A to R
  3. R receives frame, datagram removed and passed up to IP

4. R forwards datagram with IP source A, destination B
   - R creates link frame with B's MAC address as destination address, which encapsulates the IP datagram

Ethernet

- **ethernet** is the dominant wired LAN technology
  - different standards for different speeds and different physical layer media
  - but common MAC protocol and frame format
- Ethernet properties:
  - *connectionless*, without handshaking
  - unreliable
  - uses CSMA/CD with binary backoff
- physical topology:
  - in a **bus** topology, all nodes connected to the same cable, ie. in the same collision domain
  - in a **star** topology, there is an active **switch** in the center:
    * each *spoke* runs a separate Ethernet protocol, so nodes do not collide with each other
    * prevailing topology today
- Ethernet frame format:
  - IP datagram is encapsulated in an Ethernet frame
  - 7 byte preamble used to synchronize receiver and sender clock rates
    * a fixed byte pattern
  - 6 byte source and destination MAC addresses
    * if MAC address doesnot match frame or not broadcast address, frame is discarded
  - type indicates the higher level protocol, usually IP
  - CRC check
    * on error, frame is dropped

Ethernet Switch

- the **Ethernet switch** is a link layer device that takes an active role:
  - store and forwards Ethernet frames
  - examine incoming frame's MAC address, and *selectively* forward frame to outgoing links
  - *transparent*, ie. hosts are unaware of switch presence
  - *plug-and-play* and *self-learning*, switches do not need to be configured
- self-learning:
  - switch *learns* which hosts can be reached through which interfaces:

    * when a frame is received, the location of sender is learned
    * switch records mapping `<MAC address, interface, time stamp>` in switch table
  - when a destination mapping is *unknown*, the switch will *flood* all interfaces with the initial frame
  - as the hosts communicate, the switch table will fill up and flooding will no longer occur
- Ethernet switches can be *interconnected*:
  - through self-learning, switches will be able to incrementally *learn* which interfaces to forward frames over
- switches vs. routers:
  - both are interconnection devices:
    * both use store-and-forward
    * switch operates at link layer, while router operates at network layer
  - both have forwarding tables:
    * routers compute tables using routing algorithms, out of IP addresses
    * switches learn forward table using flooding, out of MAC addresses

VLANs

- support multiple *virtual* LANs over a single physical LAN infrastructure
  - ie. group switch ports under a *single* physical switch, but certain ports are completely *partitioned* into groups as if there were multiple *physical* switches
    * eg. EE department vs. CS department under the same university Ethernet switch
- *traffic isloation*:
  - frames to and from a certain group of ports can *only* reach that group of ports
    * can also define VLAN bsed on MAC address endpoints
  - additional field for a VLAN ID
  - forwarding *between* VLANs is done via routing, just as with *separate* switches
- a **trunk port** carries frames between VLANs defined over multiple physical switches
  - uses a 8-2.1q protocol with additional header fields
  - additional protocol identifer, 12-bit VLAN ID, 3-bit priority field, recomputed CRC

Data Center Networks

- **data centers** are made up of many thousands of hosts, closely coupled, in close proximity
  - eg. Google search engines, YouTube content-servers
  - *challenges*:
    * multiple applications, each serving massive numbers of clients
    * managing and *balancing* load, avoiding different bottlenecks
- built up from a hierarchy of Ethernet switches, ie. **tree** topology:
  - server racks of many hosts
  - **top of rack (TOR)** switches
  - tier-2 switches and tier-1 switches (very expensive)
  - access routers nd border routers
  - **load balancers** perform application layer routing to direct workload within data center
    * receives external client requests
    * returns results to external client, *hiding* data center internals
- alternative **fat tree** topology:
  - rich interconnection between switches and racks
  - increased throughput between racks since multiple routing paths are possible
  - increased reliability via redundancy
  - but usually requires more expensive, high-tier switches

Protocols as a Stack

- *scenario*:
  - student laptop wants to connect to the Internet at university and download a webpage

1. connecting laptop needs get its own IP address:
   - use **DHCP**, running over UDP, IP, and Ethernet (or WiFi)
     - DHCP *broadcasts* request to DHCP server
     - DHCP request eventually returns a DHCP ACK with new IP address
   - also provides address of first-hop router and DNS server
2. client needs MAC address of first-hop router:
   - use **ARP**, running over IP and Ethernet
     - ARP query broadcast, which replies with ARP reply giving MAC address
3. client needs IP address of requested website:
   - use **DNS**, running over UDP, IP, and Ethernet
     - DNS request goes through hierarchy of DNS servers
     - starting at MAC address first-hop router
   - link layer routing forwards the DNS IP datagram to DNS servers
     - using different link layer routing protocols
4. client requests to the web server of the requested website:
   - use **HTTP**, running over TCP, IP, and Ethernet
     - client opens TCP socket to web server
     - after TCP three-way handshake, TCP connection has been established
   - client sends HTTP request into socket
     - web server responds with HTTP reply containing website

Wireless Networks

- wireless phone subscribers now exceeds wired phone subscribers (5 to 1)
  - wireless Internet-connected devices equals wired Internet-connected devies
  - two *different* challenges:
    * *wireless* communication
    * handling *mobility*, ie. users who change point of attachment to network
- elements of a wireless network:
  - **wireless hosts**, that may be stationary or mobile
  - **base stations** are typically connected to wired network
    * acts as a relay fresponsible for sending packets between the wired network and wireless hosts in its area
    * eg. cell towers, 802.11 access points
    * AKA **access points (AP)**
  - **wireless links** are used to connect mobiles to base stations
    * multiple access protocol coordinates link access
    * varying data rate and transmission distance
- wireless network modes:
  - in **infrastructure mode**, the base station connects mobiles into wired network:
    * to handoff connections, mobile changes between base stations
  - in **ad hoc mode**, there are no base stations:
    * nodes only transmit to other nodes within link coverage
    * nodes organize *themselves* into a network, and route among themselves
- wireless link characteristics unique from wired links:
  - *decreased* signal strength
    * radio signal attenuates as it propagates through matter
  - *interference* from other sources
    * standardized wireless network frequencies are shared by many devices
  - *multipath* propagation
    * signal reflects off objects, arriving at slightly different times
  - **signal-to-noise (SNR)** ratio:
    * larger SNR makes it easier to extract signal from noise
    * increasing power increases SNR and decrease BER (bit error rate)
    * SNR may change with mobility
  - these factors make communication across wireless link much more difficult

- hidden terminal problem:
    - two wireless hosts may be hidden from each other (due to physical circumstance)
    - but both may transmit under CSMA, since they are unaware of the other transmitting
        - if there are *other* wireless hosts in the area that can see hear transmissions from both wireless hosts
        - this causes a collision that CSMA cannot avoid

IEEE 802.11

- **IEEE 802.11** wireless LAN:
    - 802.11b, 802.11a, 802.11g, 802.11n (increasing in speed)
        - use either 2.4-5 or 5-6 GHz spectrum
    - all use CSMA/CA for multiple access
    - all have base-station and ad-hoc network versions
- channels:
    - the frequency spectrum is divided into 11 channels at different frequencies
    - AP admin chooses frequency for AP
        - interference is possible, since channel can be the same as that chosen by neighboring AP
    - a host must *associate* with an AP:
        - host scans channel, *passively* listening for **beacon frames** containing AP name and MAC address
            - alternatively, can *actively* broadcast probe requests to APs
        - host chooses AP to associate with
- multiple access:
    - 802.11 uses CSMA for sensing before transmitting
    - but 802.11 *does not* use collision detection:
        - cannot sense all collisions due to hidden terminal, and fading of weak received signals
        - goal instead is to *avoid collisions* using CSMA/CA (collision avoidance)
- 802.11 CSMA (uses ACKs at the link layer):
    - *sender*:
        - if sense channel idle for DIFS (DCF interframe space), transmit entire frame
        - if sense channel busy, start drandom backoff time, timer counts down while channel idle, transmit when timer expires
        - if no ACK, increase random backoff interval and repeat

- *receiver*:
  - if frame received OK, return ACK after SIFS (short interframe space)

- 802.11 CA:
  - hosts send RTS (request to send) frames to receivers:
    - short 20-byte frames
    - if RTS collide at receiver, these RTS frames are wasted
    - but RTS frames are so small, this loss is amortized with respect to the large amount of data sent (often 1000s of bytes)
  - if receiver *accepts* RTS, sends CTS (clear to send) frame:
    - just 14-bytes
    - notifies real sender to start transmission, as well as sent to other wireless hosts in the area
    - addresses issue of hidden terminals
  - after receiver has received all data from sender, sends ACK frames to all hosts in the area
    - another host can now send an RTS to the receiver
- 802.11 frame header:
  - frame control, duration fields
  - 3 address fields:
    1. MAC address of wireless host or AP to receive frame
    2. MAC address of wireless host or AP transmitting frame
    3. MAC address of router interface to which AP is attached
  - sequence control, another address field for ad-hoc mode
  - payload, CRC

Cellular Networks

- cellular network architecture:
  - a **cell** covers a geographical region
  - base stations
  - **mobile switching centers** connect cells to a wired telephone network
  - 3G uses a dual network architecture for voice *and* data:
    - cellular data network operates in *parallel* with existing cellular voice network
    - except at network edge which performs multiplexing
  - 4G-LTE gets rid of the dual architecture:
    - using public Internet alone, instead of public telephone network
    - no separation between voice and data
    - other layers are implemented between IP and link layer
- multiple access uses combined FDMA/TDMA:
  - divide spectrum in frequency channels, divide each channel into time

slots

Mobility

- from the network perspective, there is a *spectrum* of mobility
  - mobile user may use the same access point
  - mobile user may connect and disconnect from the same network using DHCP
  - mobile user passes through multiple APs while maintaining ongoing connections
- *vocabulary*:
  - **home network** is the permanent *home* of the mobile
  - **permanent address** is the address in home network that can *always* be used to reach mobile
  - **home agent** is an entity that performs mobile functions on behalf of the mobile when mobile is remote
  - **visited network** is the network in which the mobile currently resides
  - **care-of-address** is the address in visited network of the mobile
  - **foreign agent** is an entity in visited network that performs mobile functiosn on behalf of the mobile
  - **correspondent** wants to communicate with mobile
- *registration*:
  - mobile contacts foreign agent on entering visited network
  - foreign agent contacts home agent, stating the mobiel is a resident in my network
  - thus foreign agent knows about mobile, and home agent knows location of mobile
- approaches to mobility:
  - allow routing to handle
    * not scalable to millions of hosts
  - allow end-systems to handle:
    * with *indirect* routing, communication goes through home agent, and then forwarded to remote agent
      · as mobile moves around, communication is maintained as mobile registers with different visited networks
      · packets are forwarded through home network to mobile, and mobile replies directly to correspondent
      · note that correspondent *does not* need to be aware that destination is mobile
      · has triangle routing problem where routing is inefficient when correspondent and mobile are in the same network

        * with *direct* routing, correspondent gets foreign address, sends directly to mobile
            · requests foreign address from home network
            · correspondent forwards to foreign agent, and mobile replies directly to correspondent
            · overcomes triangle routing problem, but more responsibility for correspondent
- **mobile IP** is an standardized implementation for mobility:
    – standardizes indirect routing, agent discovery, and mobile registration
    – indirect routing:
        * packet sent by home agent to foreign agent is a packet within a packet
        * foreign agent recovers original tunneled packet, and sends to mobile
    – agent advertisement:
        * accomplished by broadcasting ICMP messages with a special field of bits and *care-of* addresses
    – *impact* on higher layer protocols:
        * logically, impact should be minimal
        * but performance wise:
            · packet loss/delay due to bit errors and handoff
            · loss triggers conegstion control
            · delay impairments for real-time traffic
            · limited bandwidth of wireless links

Network Security

- **network security** attempts to provide the following guarantees:
  - **confidentiality**: only the sender and intended receiver should *understand* message contests, eg. using encryption
  - **authentication**: sender and receiver want to confirm identity of each other
  - **message integrity**: sender and receiver want to ensure message has not been altered without detection
  - **access and availability**: services must be accessible and available to users

Cryptography

- **cryptography** is the usage of different encryption and decryption algorithm
  - cryptography guarantees that encrypted messages *cannot* be bruteforce decrypted in a feasible manner
- in *symmetric* key cryptography:
  - sender and receiver share the same key K_s:
    * to encrypt message m, $K_s(m)$
    * to decrypt message, $m = K_s(K_s(m))$
  - this shared key must be kept safe from attackers
  - requires the two ends to agree on the key in the first place, eg. using public key cryptography
- in *public* key cryptography:
  - sender and receiver *do not* share secret key
  - each have a *public* encryption key known to *all*, even attackers
  - each have a *private* decryption key known *only* to the receiver
  - receiver has public key K_R^+ and private key K_R^-:
    * to encrypt message m meant for the receiver, $K_R^+(m)$
    * for receiver to decrypt message, $m = K_R^-(K_R^+(m))$

Authentication

- **authentication** deals with allowing the sender and receiver to prove their identities to each other
  - try different authentication protocols (ap)
- ap1.0:

  - one end simply tells the other their identity, eg. "I am Bob"
  - not secure, anyone would be able to declare themselves to be someone else
- ap2.0:
  - one end tells the other their identity in an IP packet containing the source IP address
  - not secure, an attacker can create a packet *spoofing* another's address
- ap3.0:
  - one end tells the other their identity and sends their secret password to *prove* it
  - not secure, due to possible **playback attack**:
    * attacker records the sent packet and later *plays it back*
    * may not know the plaintext password, but can simply reuse the message
- ap4.0:
  - use a **nonce** or unique number used only once-in-a-lifetime
  - to prove one end is *alive*, the other end will send a nonce R
    * the end must return R, encrypted with a shared secret key
  - prevents possible playback attack since nonce will expire
- ap5.0:
  - use nonces together with public key cryptography
    * nonce is encrypted with a private key such that only the corresponding public key decrypts it
  - still not secure, due to possible **man in the middle attack**:
    * attacker sits between two communicating ends
    * to either end, attacker pretends to be the other end of communication
    * *difficult* to detect, transmission requires many hops
  - to solve this attack:
    * requires *common trust* from a third party to boostrap
    * a third party needs to verify the identities of each end
- digital **signatures**:
  - sender signs document, establishing he is the document owner
  - recipient can then prove to someone that the sender and no one else must have signed the document
  - *idea*:
    * use private key to create the signature, and public key to verify the signature
    * to reduce the length of the signature, use a **message digest**:
      · message digest is a fixed-length, easily computed digital *fingerprint*
      · apply a hash function that produces a fixed size digest from the larger message

        * to verify signature, use the same hash function to create the digest to compare
- **certification authorities (CA)** bind a public key to a particular entity:
    - acts as a trusted third party
    - an entity registers its public key with CA:
        * entity provides *proof of identity*
        * CA creates certificate binding the entity to its public key
    - certificate is encrypted by the CA private key:
        * to verify the certificate and acquire A's public key, decrypt signature using the CA public key to get A's public key
    - this solves the man in the middle attack security hole

SSL

- the **secure sockets layer (SSL)** is a widely deployed security protocol:
    - supported by almost all browsers and web servers, via HTTPS
        * lies *between* the TCP and application layers
        * provides an API to applications
    - **transport layer security (TLS)** is a similar variation
    - *provides*:
        * confidentiality, integrity, authentication
    - SSL handshake purposes:
        * server authentication (client authentication is optional)
        * negotiation to agree on crypto algorithms
        * establish keys
    - after the initial SSL handshake, all other communications are encrypted

VPN

- institutions often want private networks for security
    - **virtual private networks (VPN)** allow for an institution's inter-office traffic to be sent over public Internet
        * encrypted before entering public Internet
        * logically separated from other traffic
    - VPN is built over IPsec
- IPsec services:
    - data integrity
    - origin authentication
    - replay attack prevention
    - confidentiality

- IPsec can be *tunneled* over edge routers:
  - source and destination is only seen by public Internet as IPsec-aware edge routers
  - the underlying source, destination, and payload is completely encrypted
- two IPsec protocols:
  1. authentication header (AH) protocol provides authentication and integrity but *not* confidentiality
  2. encapsulation security protocol (ESP) provides all three
     - more widely used than AH
- IPsec implementation:
  - before sending data a **security association (SA)** is established from sending to receiving entity:
    - *simplex*, for only one direction
    - sender maintains state information about SA, eg. identifier (SPI), origin interface, type of encryption and integrity check, encryption and authentication keys
  - endpoint holds SA state in a **security association database (SAD)**
    - when sending IPsec datagram, endpoint accesses SAD to determine how to process datagram
    - when IPsec datagram arrives, endpoint indexes into SAD with SPI and processes datagram accordingly

Firewalls

- **firewalls** isolates an organization's internal network from larger Internet
  - allows some packets to pass, blocking others
  - different types of firewalls:
    - stateless packet filters
    - stateful packet filters
    - application gateways
  - *pros*:
    - prevent **denial of service (DDOS)** attacks ie. SYN flooding
    - prevent illegal modification or access of internal data
    - allow only authorized access to inside network
  - *cons*:
    - due to **IP spoofing**, rotuer cannot really know if data comes from a claimed source
    - filters often use all or nothing policy for UDP
- stateless packet filtering:
  - internal network connected to Internet via **router firewall**

- routers *filter* packet-by-packet based on:
  - source and destination addresses or ports
  - ICMP message type
  - TCP SYN and ACK bits
- some filtering *policy* examples:
  - block all incoming and outgoing UDP flows and telnet connections
    - drop all incoming and outgoing datagrams with IP protocol field 17 and source or destination port 23
  - prevent external clients from making TCP connections with internal clients, but allow internal clients to connect outside
    - drop inbound TCP segments with ACK 0:
  - prevent outside web access
    - drop all outgoing packets to any IP address, port 80
- stateful packet filtering:
  - track status of every TCP connection
  - track connection setup and teardown status to determine whether incoming, outgoing packets *make sense*
  - timeout inactive connections at firewall

Appendix

Network Programming

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

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

Socket Programming API

- part of the `sys/socket.h` and `netinet/in.h` headers

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

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

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

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

`sockaddr` and `sockaddr_in`:

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

struct sockaddr_in {
```

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

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

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

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

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

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

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

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

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

- returns number of bytes processed or -1
  - 0 if socket is closed
- `sockfd` is the socket file desriptor

- `buf` is a data buffer
- `nbytes` is the number of bytes to process
  - max to read, or desired number to send

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

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

Utilities

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

`hostent` structure: ‘ `c struct hostent {   char*  h_name;      // canonical host name   char**`