CS118: Computer Network Fundamentals

Professor Lu

Spring 2020

Contents

CS118: Computer Network Fundamentals

Overview

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

Access Networks

- access networks physically connects 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* hierarchy of networks
  - within the home, a **wireless access point** is connected to the DSL or cable modem
    - various devices can *wirelessly* connect to the access point
    - speed of access point is slower than a direct *wired* connection
      - speed also dependent on the wifi card of the device connecting to the access point
- **enterprise access network** or **Ethernet**:
  - uses a special hardware device called an **Ethernet switch**
  - 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:
      * 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**, the total amount of time to transmit all bits of a packet
  - packets can then be **lost** or dropped if the memory buffer for the queue fills up
- thus **packet delay** overall has multiple sources:
  - **processing delay** from checking bit errors and determining output link
  - **queuing delay** from awaiting transmission, depends on congestion
    - as (L bits * a average arrival rate) / R rate approaches 1, queuing delay becomes large
    - above 1, the average delay becomes infinite
  - **transmission delay** is how long it takes to push out all bits of the packet, depends on packet size
    - L bits / R rate
  - **propagation delay** is the time for a bit to actually travel to another router
    - d length / s speed
- the `traceroute` program provides delay measurement from source to destination
  - send three probe packet that reaches each router along the path
  - measures time interval between transmission and reply
- handling **packet loss**:
  - when a packet is lost, the source must slow its transmission, and also 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
    * 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
  - 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
    - is 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 eeither:
    - 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)
      2. make the HTTP request and to receive the header of the response to return
      - 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 communincation 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
  - (basic) 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
- **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 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

DNS

- hosts can be identified in multiple ways:
  - through a mnemonic **hostname**, eg. google.com or ucla.edu
  - or through its **IP address**, eg. 121.7.106.83
    - has a rigid hierarchical structure, 4 bytes
  - need a directory service that translates hostnames to IP addresses
    - this is the **domain name service (DNS)**
- problems with a *centralized* design:
  - single point of failure
  - high traffic volume
  - can be distant from clients
  - requires high maintenance
- intsead, DNS is a *distributed* database:
  - made up of a *hierarchy* of **DNS 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
  - **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
  - **authoritative servers** form the loweset level
    - every organization with publicly accessible hosts must provide DNS records that map their names to IP addresses
    - thus each of these organizations has an authoritative server
      - 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 **recursive** queries that request another DNS server to obtain the mapping on its behalf
  - or **iterative** queries where replies are returned directly to the DNS server
- DNS details:
  - uses UDP and port 53

  - 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:
    - 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
    - mapping held for around two days (since mapping is not permanent)
  - 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
  - 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:
    - a *header* section of 12 bytes with:
        * a 16-bit ID
        * 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

P2P

- P2P applications use a P2P architecture
    - no reliance on always-on servers
    - 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

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

- 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
    * videos are streamed at a certain bitrate, depending on the **compression** that is applied to it
  - network must be able to provide an average throughput that is at least as large as the bit rate of the compressed video
- 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 encoded into different versions:
  - each version has a different bitrate and quality level
    * server has a **manifest file** with URLs for the different versions
  - 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
  - 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:
    * can be far from many clients
    * popular videos may be sent many times over the same links
  - instead use **content distribution networks (CDNs)** that:
    * manage servers in distributed locations
    * store cached copies in its 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 critical sites, eg. at IXPs:
  - lower maintenance overhead, but more delay and lower throughput for users
- to be succesful, CDNs must *intercept* and *redirect* requests:
  - 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 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

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

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

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

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

Utilities

- note that **byte ordering** matters when transferring data between host systems
    - little endian vs. big endian
    - hosts may use different orderings, but the **network byte order** is always big endian
    - `ntohl` performs net-to-host long translation
    - `ntohs` performs net-to-host short translation
    - `htonl` performs host-to-net long translation
    - `htons` performs host-to-net short translation
    - thus, the port number and IP address in the API address structures should always be converted
        * eg. `servaddr.sin_port = htons(servport)`
- other utilities are provided for working with network addresses:

- `struct hostent* gethostbyname(const char* hostname)` translates a host name to an IP address
- `struct hostent* gethostbyaddr(const char* addr, size_t len, int family)` translates an IP address to host name
- `char* inet_ntoa(struct in_addr inaddr)` translates IP dddres to a dotted-decimal string
- `int gethostname(char* name, size_t namelen)` reads the local host's name
- `in_addr_t inet_addr(const char* str)` translates dotted-decimal string to IP address *in* network byte order
  - `int inet_aton(const char* str, struct in_addr* inaddr)` same translation, different format

`hostent` structure:

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