TELE 3018 in one slide

Protocols, layering, client/server, router, connection-oriented/less, delays, "bandwidth"
Frame operators: framing, parity/CRC error detection
Error control: stop&wait/sliding window, selective repeat/go-back-N; cumulative, duplicate and delayed acks; fast retransmit; RTT estimation
MAC: Random access, carrier-sense, collision detection, exponential backoff, wireless issues
Identifiers: MAC/IP addresses, ARP, DHCP, NAT, DNS
Interconnection: bridges, prefix matching, ICMP
Routing: Dijkstra’s algorithm, link-state/distance-vector, Autonomous Systems, multicast, Mobile IP
Congestion: AIMD, slow start
Applications: email (SMTP, POP/IMAP), web ((non-)persistent HTTP, HTML), sockets API, UDP vs TCP

Client: core

int main(int argc, char **argv) {
    struct sockaddr_in channel; /* info about server */
    struct sockaddr_in channel;
    h = gethostbyname(argv[1]); /* look up host’s IP address */
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
    while (1) {
        bytes = read(s, buf, BUF_SIZE); /* read from socket */
        write(1, buf, bytes); /* write to standard output */
    }
}

gethostbyname(): Use DNS to translate name into IP address (stored in hostent, later extract address for bind)
socket(): Create a "descriptor" for future communication: IP protocol, stream (TCP). Under Unix, sockets are essentially file descriptors (distinct under Windows).
connect(): Establish a connection with a specified address, return(): I/O through socket (Unix only)

Server: core

int main(int argc, char *argv[]) {
    struct sockaddr_in channel;
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
    setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));
    b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
    l = listen(s, QUEUE_SIZE); /* specify queue size */
    while (1) {
        sa = accept(s, 0, 0); /* block for connection request */
        read(sa, buf, BUF_SIZE); /* read file name from socket */
        fd = open(buf, O_RDONLY); /* open the file to be sent back */
        while (1) {
            bytes = read(fd, buf, BUF_SIZE); /* read from file */
            write(sa, buf, bytes); /* write bytes to socket */
        }
    }
}

setsockopt(): Set options for socket. "REUSEADDR": Allows address to be reused, despite possibly already being used (e.g. wait for max(RTT))
bind(): Specify which local "address" (IP+port #) socket should connect to.
listen(): Prepare socket to receive connection requests. Up to QUEUE_SIZE will be buffered.
accept(): Accept a connection request & create a distinct socket for communication

UDP: User Datagram Protocol [RFC 768]

- "no frills," “bare bones” Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

Client: core

int main(int argc, char **argv) {
    struct sockaddr_in channel; /* info about server */
    struct sockaddr_in channel;
    h = gethostbyname(argv[1]); /* look up host’s IP address */
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
    while (1) {
        bytes = read(s, buf, BUF_SIZE); /* read from socket */
        write(1, buf, bytes); /* write to standard output */
    }
}

gethostbyname(): Use DNS to translate name into IP address (stored in hostent, later extract address for bind)
socket(): Create a "descriptor" for future communication: IP protocol, stream (TCP). Under Unix, sockets are essentially file descriptors (distinct under Windows).
connect(): Establish a connection with a specified address, return(): I/O through socket (Unix only)
Outline

TCP: Overview
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?
- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions
  - SampleRTT will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current SampleRTT

TCP sender events:

- data rcvd from app:
  - Create segment with seq #
  - seq # is byte-stream number of first data byte in segment
  - start timer if not already running
    (think of timer as for oldest unacked segment)
  - expiration interval: TimeOutInterval

- timeout:
  - retransmit segment that caused timeout
  - restart timer

- Ack rcvd:
  - If acknowledges previously unacked segments
    - update what is known to be asked
    - start timer if there are outstanding segments
TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver | TCP Receiver action
--- | ---
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed | Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #: One other segment has ACK pending | Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq #: Gap detected | Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap | Immediate send ACK, provided that segment starts at lower end of gap

Fast Retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs:
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments
- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)
- client: connection initiator
  - Socket clientSocket = new Socket("hostname","port number");
- server: contacted by client
  - Socket connectionSocket = welcomeSocket.accept();

Three way handshake:

Step 1: client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

Step 2: server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

Outline
**TCP Flow control: how it works**

(Suppose TCP receiver discards out-of-order segments)
- spare room in buffer
  - RcvWindow
  - RcvBuffer - \(\text{LastByteRcvd} - \text{LastByteRead}\)

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
  - guarantees receive buffer doesn't overflow

**TCP AIMD**

**Multiplicative decrease:**
- Cut CongWin in half after a loss event

**Additive increase:**
- Increase CongWin by 1 MSS every RTT in the absence of loss events: probing

**TCP Slow Start (more)**

- When connection begins, increase rate exponentially until first loss event:
  - double CongWin every RTT
  - done by incrementing CongWin for every ACK received
- **Summary:** initial rate is slow but ramps up exponentially fast

**TCP sender congestion control**

<table>
<thead>
<tr>
<th>Event</th>
<th>State</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK received for previously unacked data</td>
<td>Slow Start (SS)</td>
<td>CongWin = CongWin + MSS, if CongWin = Threshold set state to “Congestion Avoidance”</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td>ACK received for previously unacked data</td>
<td>Congestion Avoidance (CA)</td>
<td>CongWin = CongWin/2 (MSS/CongWin)</td>
<td>Additive increase, resulting in an increase of CongWin by 1 MSS every RTT</td>
</tr>
<tr>
<td>Data event detected by Triple Duplicate ACK</td>
<td>SS or CA</td>
<td>CongWin = Threshold, set state to “Congestion Avoidance”</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td>Timeout</td>
<td>SS or CA</td>
<td>CongWin = Threshold, set state to “Slow Start”</td>
<td>Enter slow start</td>
</tr>
<tr>
<td>Duplicate ACK</td>
<td>SS or CA</td>
<td>Increment duplicate ACK count for segment being acked</td>
<td>CongWin and Threshold not changed</td>
</tr>
</tbody>
</table>
Outline

DNS

DNS services
- Hostname to IP address translation
- Host aliasing
  - Canonical and alias names
- Mail server aliasing
- Load distribution
  - Replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance
doesn't scale!

Recursive queries

recursive query:
- puts burden of name resolution on contacted name server
- heavy load?

iterated query:
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"

DNS: caching and updating records

- once (any) name server learns mapping, it caches mapping
  - cache entries timeout (disappear) after some time
  - TLD servers typically cached in local name servers
    - Thus root name servers not often visited
- update/notify mechanisms under design by IETF
  - RFC 2136
HTTP overview

HTTP: hypertext transfer protocol
Used to deliver web information, e.g. HTML pages & other content.
2 major versions: HTTP 1.0 (RFC 1945) & HTTP 1.1 (RFC 2068)

Uses TCP:
- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"
- server maintains no information about past client requests

Aside
- Protocols that maintain "state" are complex!
- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP request message

- two types of HTTP messages: request, response
- HTTP request message:
  - ASCII (human-readable format)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
```

Carriage return, line feed indicates end of message

HTTP connections

Nonpersistent HTTP
- At most one object is sent over a TCP connection.
- HTTP/1.0 uses nonpersistent HTTP

Persistent HTTP
- Multiple objects can be sent over single TCP connection between client and server.
- HTTP/1.1 uses persistent connections in default mode
Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type

```plaintext
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data ....
...........................
......base64 encoded data
```

Mail access protocols

- SMTP: delivery/storage to receiver’s server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
  - IMAP: Internet Mail Access Protocol [RFC 1730]
  - HTTP: Hotmail, Yahoo! Mail, etc.
  - more features (more complex)
  - manipulation of stored msgs on server

```
$ telnet smtp.unsw.edu.au 25
Trying 149.171.96.25...
Connected to smtp.unsw.edu.au.
Escape character is '^]'.
220 smtp.unsw.edu.au ESMTP Sendmail 8.11.2/8.11.2; Wed, 22 Oct 2003 08:05:55 +1000 (EST)
HELO host.ee.unsw.edu.au
250 smtp.unsw.edu.au Hello host.ee.unsw.EDU.AU [#.#.#.#], pleased to meet you
MAIL FROM: user@source.unsw.edu.au
250 2.1.0 user@source.unsw.edu.au... Sender ok
RCPT TO: user2@destination.org
250 2.1.5 user2@destination.org... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
Fun
Fun
CRLF CRLF indicates end of message
250 2.0.0 h9LM6Wc20247 Message accepted for delivery
QUIT  ^ Message ID allocated by server
221 2.0.0 smtp.unsw.edu.au closing connection
Connection closed by foreign host.
```
SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

Comparison with HTTP:
- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg