**Outline**

- Development of reliable protocol
- Sliding window protocols
  - Go-Back-N, Selective Repeat
- Protocol performance
- Sockets, UDP, TCP, and IP
- UDP operation
- TCP operation
  - connection management
  - flow control
  - congestion control

**TCP: Overview**

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte steam:
  - no “message boundaries”
- sliding window:
  - TCP congestion and flow control set window size
- send & receive buffers

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size (typical size 1460 - why?)

- connection-oriented:
  - handshaking (exchange of control msgs) init’s sender, receiver state before data exchange

- flow controlled:
  - sender will not overwhelm receiver
**TCP segment structure**

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

**Options (variable length)**

- **application data (variable length)**

### TCP Connection Management

(includes establishment, closing, and reliable data delivery using sequence numbers and timers)
**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)
- Socket calls:
  - client: connect()
  - server: 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

**TCP Three way Handshake**

![TCP Three way Handshake Diagram]
TCP Connection Management (cont.)

Closing a connection:

(note: two army problem)

client closes socket:
   clientSocket.close();

**Step 1:** client end system sends TCP FIN control segment to server

**Step 2:** server receives FIN, replies with ACK.
   Closes connection, sends FIN.

TCP Connection Management (cont.)

**Step 3:** client receives FIN, replies with ACK.
   - Enters “timed wait” - will respond with ACK to received FINs

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle

TCP seq. #’s and ACKs

**Seq. #’s:**
- byte stream “number” of first byte in segment’s data

**ACKs:**
- seq # of next byte expected from other side
- cumulative ACK

Host A
- User types ‘C’
- Seq=42, ACK=79, data = ‘C’
- host ACKs receipt of ‘C’
- echoes back ‘C’

Host B
- Seq=79, ACK=43, data = ‘C’
- host ACKs receipt of echoed ‘C’
- Seq=43, ACK=80
- simple telnet scenario
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
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT

How to measure RTT when there are retransmissions?

**EstimatedRTT = (1- \( \alpha \))*EstimatedRTT + \( \alpha \)*SampleRTT**

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)
Example RTT estimation:

TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

\[
\text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
\]

(typically, \(\beta = 0.25\))

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
TCP reliable data transfer

- TCP creates reliable service on top of IP’s unreliable service
- Sliding windows improve utilization
- Cumulative ACKs provide redundancy
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate ACKs
- Initially we consider simplified TCP sender:
  - Ignore duplicate ACKs
  - Ignore flow control, congestion control

TCP sender events:

**Data rcvd from app:**
- Create segment with seq #
- Seq # is number of first data byte in segment
- Start timer if not already running
  (think of timer as for oldest unACKed segment)
- Expiration interval: $\text{TimeOutInterval}$

**Timeout:**
- Retransmit segment that caused timeout
- Restart timer

**ACK rcvd:**
- If acknowledges previously unACKed segments
  - Update what is known to be ACKed
  - Start timer if there are outstanding segments
TCP ACK generation [RFC 1122, RFC 2581]

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

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 for that segment
- **If sender receives 3 ACKs for same data**
  - it assumes that segment after ACKed data was lost
  - **fast retransmit**: resend segment before timer expires
TCP Options...

- Recall the TCP header
- Options enable TCP to evolve to accommodate new technologies, etc.
- Most options are quite short
- Most are used only during SYN/SYNACK phase
Maximum Segment Size (MSS)

- Tells receiver not to send segments larger than specified
- Typically based on lower layer limit (Ethernet)

Window Scaling

- 16-bit in window field means 64K limit
- Window scaling shifts window field by specified value (255 in 8 bits, but limited to 14. why?)
Selective Acknowledgments (SACK)

- Allows receiver to indicate it has received blocks of the stream, missing others
- Whether permitted is negotiated at SYN time:

```
MAC header | IP header | TCP header | TCP option 4 | Data
```

TCP Option 4:

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Kind | Length
```

- **Kind**: 8 bits. Set to 4.
- **Length**: 8 bits. Set to 2.

Actual SACK header...

```
MAC header | IP header | TCP header | TCP option 5 | Data
```

TCP Option 5:

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Kind | Length
```

- **Kind**: 8 bits. Set to 5.
- **Length**: 8 bits. Unsigned. 4 to ?
- **SACK Block []**: Variable length.
  One or more SACK block structures.

- **SACK block**: 64 bits.

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Left Edge of Block
Right Edge of Block
```

- **Left Edge of Block**: 32 bits.
  The first sequence number of this block.
- **Right Edge of Block**: 32 bits.
  The sequence number immediately following the last sequence number of this block.
**SYN Flood DOS Attack**

- A classic denial of service (DOS) attack
- Flood a node with TCP connection requests
  - Node allocates resources for connection
  - Sends SYNACK segment
  - Last part of handshake (from attacker) never happens
- Node eventually releases resources for “half-open” connections, but requests come too fast
- All kernel resources for TCP connections are consumed
- Other, legitimate clients are shut out

**Solution**

- Instead of allocating half-open connection
  - Server picks seq # with hash of IP addresses, ports and a secret number (known only to server)
  - This is called a SYN cookie
  - Sends back SYNACK with the SYN cookie
  - Allocates no resources, and forgets the seq #
  - If ACK segment arrives, server regenerates the seq #. The ACK field of the arriving packet should equal that number, plus 1.
  - If so, go ahead and allocate resources for this legitimate client.
  - If an ACK never arrives (as in an attack) we have lost only the time to handle the SYN packet
**Completed handshake attack**

- SYN Floods (were) most effective if launched from multiple clients
  - Distributed Denial of Service (DDoS) Attack
- Some attacks get around SYN cookies by completing the TCP connection
  - Resources are allocated but not used
- Much harder to defend against - hard to tell legitimate clients from attackers

**TCP Flow Control and Congestion Control**
TCP Flow Control

- receive side of TCP connection has a receive buffer (tied to socket):

  - flow control
    sender won’t overflow receiver’s buffer by transmitting too much, too fast

- speed-matching service: matching send rate to receiving application’s drain rate

- receiving process may be slow at reading from buffer (socket)

TCP Flow control: how it works

- receiver: advertises unused buffer space by including rwnd value in segment header (window advertisement)

- sender: limits # of unACKed bytes to rwnd
  - guarantees receiver’s buffer doesn’t overflow
  - Basically, an adaptive sliding window

Unused buffer space: rwnd = RcvBuffer - (LastByteRcvd - LastByteRead)
Principles of Congestion Control

Congestion:
- informally: “too many sources sending too much data too fast for network to handle”
- not the same as flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a major issue in Internet performance

Controlling Congestion

two broad approaches:

end-end congestion control:
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

network-assisted congestion control:
- routers provide feedback to end systems
  - single bit indicating congestion
  - Used in early networks: SNA, DECbit, ATM)
  - explicit rate sender at which sender should transmit
TCP congestion control:

- **goal**: TCP sender should transmit as fast as possible, but without congesting network
  - **Q**: how to find rate *just* below congestion level
- decentralized: each TCP sender sets its own rate, based on *implicit* feedback:
  - **ACK received**:
    - segment arrived (a good thing!)
    - network not congested, so increase sending rate
  - **ACK not received**:
    - assume loss due to congested network
    - so decrease sending rate

Congestion Window

- In **addition to a receiver window (from advertisements)** the sender maintains a *congestion window, cwnd*
- **Rwnd** indicates state of the **receiver**
- **Cwnd** indicates state of the **network**

- *** The send window is min(rwnd,cwnd)
**TCP Slow Start**

- when connection begins, \( cwnd = 1 \) MSS (max segment size)
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double \( cwnd \) every RTT
  - done by incrementing \( cwnd \) by 1 for every ACK received

**TCP Congestion Control: more details**

- **segment loss event:** reducing \( cwnd \)
  - timeout: no response from receiver
    - cut \( cwnd \) to 1
  - 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
    - cut \( cwnd \) in half, less aggressively than on timeout

- **ACK received:** increase \( cwnd \)
  - slowstart phase:
    - increase exponentially fast (despite name) at connection start, or following timeout
  - congestion avoidance:
    - increase linearly
TCP congestion control: bandwidth probing

- “probing for bandwidth”: increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)

![Graph showing TCP's sawtooth behavior](image)

Transitioning into/out of slowstart

- **ssthresh**: cwnd threshold maintained by TCP
- on loss event: set ssthresh to cwnd/2
  - remember (half of) TCP rate when congestion last occurred
- when cwnd >= ssthresh: transition from slowstart to congestion avoidance phase
TCP: congestion avoidance

- when cwnd > ssthresh, grow cwnd linearly
  - increase cwnd by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: cwnd = cwnd + MSS/cwnd for each ACK received

AIMD

- ACKs: increase cwnd by 1 MSS per RTT: additive increase
- loss: cut cwnd in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

Popular “flavors” of TCP

![Diagram showing the window size (in segments) vs. transmission round for TCP Tahoe and TCP Reno, with ssthresh indicated.]
Summary: TCP Congestion Control

- when $cwnd < ssthresh$, sender in slow-start phase, window grows exponentially.
- when $cwnd >= ssthresh$, sender is in congestion-avoidance phase, window grows linearly.
- when triple duplicate ACK occurs, ssthresh set to $cwnd/2$, cwnd set to $\sim ssthresh$
- when timeout occurs, ssthresh set to $cwnd/2$, cwnd set to 1 MSS.

Effects of Wireless LANs

- Loss rates on wireless LANs are much higher than on wired networks.
- Some loss can be mitigated by retransmissions at the link layer (discussed later).
- IP handoffs for mobile devices also cause loss.
- What is TCP’s response to packet loss?
- Many solutions have been proposed (and are still being proposed!) to deal with this issue.
TCP Fairness

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

Why is TCP fair?

Two competing sessions:
- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally
Fairness (more)

Fairness and UDP
- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss

Fairness and parallel TCP connections
- nothing prevents app from opening parallel connections between 2 hosts.
- web browsers do this
- example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2!

Summary

- Developed basis of reliable protocol
- Learned about sliding window protocols
  - Go-Back-N, Selective Repeat
- Analyzed protocol performance
- Discussed demultiplexing
- UDP operation (ports, checksum)
- TCP operation
  - connection management (sequence numbers)
  - flow control (receive window)
  - congestion control (congestion window)