Topics for This Week

• Transmission Control Protocol (TCP)
  – Connection Management
  – Error/Flow/Congestion Control

• User Datagram Protocol (UDP)

• Readings
  – Sections 6.4, 5.3.1, 5.3.2, 5.3.6
TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream*:
  - no “message boundaries”
- pipelined:
  - 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
- 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

<table>
<thead>
<tr>
<th></th>
<th>source port #</th>
<th>dest port #</th>
<th>sequence number</th>
<th>acknowledgement number</th>
<th>head len</th>
<th>not used</th>
<th>U</th>
<th>A</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>F</th>
<th>rcvr window size</th>
<th>checksum</th>
<th>ptr urgent data</th>
</tr>
</thead>
</table>

**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)

**application data** (variable length)
TCP Connection Management

TCP sender, receiver establish “connection” before exchanging data segments
- initialize TCP variables:
  - seq. #s
  - buffers, flow control info
- **client**: connection initiator
- **server**: contacted by client

**Three way handshake:**

**Step 1:** client end system sends TCP SYN control segment to server
- specifies initial seq #

**Step 2:** server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server → receiver initial seq. #

**Step 3:** client replies with an ACK segment
TCP Connection Management (cont.)

Closing a connection:

client closes socket:

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

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

Diagram:

- Client closes socket.
- Server receives FIN, replies with ACK.
- Connection closed.
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.
TCP Connection Management (cont)

TCP client lifecycle

- **CLOSED**: client application initiates a TCP connection
- **TIME_WAIT**: receive FIN, send ACK
- **SYN_SENT**: receive SYN & ACK, send ACK
- **FIN_WAIT_2**: receive ACK, send nothing
- **ESTABLISHED**: client application initiates close connection
- **FIN_WAIT_1**: send FIN
TCP Connection Management (cont)

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

User
types
‘C’

Host A

Seq=42, ACK=79, data = ‘C’
host ACKs receipt of ‘C’, echoes back ‘C’

Seq=79, ACK=43, data = ‘C’
host ACKs receipt of echoed ‘C’

Seq=43, ACK=80
time

simple telnet scenario
TCP: Retransmission Scenarios

- **Host A**
  - Seq=92, 8 bytes data
  - ACK=100
  - timeout
  - lost ACK scenario

- **Host B**
  - Seq=100, 20 bytes data
  - ACK=100

- **Host A**
  - Seq=92, 8 bytes data
  - ACK=100
  - timeout
  - premature timeout, cumulative ACKs

- **Host B**
  - Seq=100, 8 bytes data
  - ACK=120
Q: how to set TCP timeout value?
• longer than RTT
  – note: RTT will vary
• 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, cumulatively ACKed segments
• SampleRTT will vary, want estimated RTT “smoother”
  – use several recent measurements, not just current SampleRTT
TCP Round Trip Time and Timeout

\[ \text{EstimatedRTT} = (1-x) \times \text{EstimatedRTT} + x \times \text{SampleRTT} \]

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

**Setting the timeout**

- \textbf{EstimatedRTT} plus “safety margin”
- large variation in \textbf{EstimatedRTT} \implies larger safety margin

\[ \text{Timeout} = \text{EstimatedRTT} + 4 \times \text{Deviation} \]

\[ \text{Deviation} = (1-x) \times \text{Deviation} + x \times |\text{SampleRTT} - \text{EstimatedRTT}| \]
Flow/Congestion Control

- Sometimes sender shouldn’t send a pkt whenever its ready
  - Receiver not ready (e.g., buffers full)
  - React to congestion
    - Many unACK’ed pkts, may mean long end-end delays, congested networks
    - Network itself may provide sender with congestion indication
  - Avoid congestion
    - Sender transmits smoothly to avoid temporary network overloads
TCP Flow Control

**flow control**
sender won’t overrun receiver’s buffers by transmitting too much, too fast

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- **RcvWindow** field in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

**RcvBuffer** = size of TCP Receive Buffer

**RcvWindow** = amount of spare room in Buffer

receiver buffering
What is Congestion?

- Informally: “too many sources sending too much data too fast for network to handle”
- Different from flow control!
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)
Causes/Costs of Congestion: Scenario I

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput
Causes/Costs of Congestion: Scenario II

- one router, *finite* buffers
- sender retransmission of lost packet
Causes/Costs of Congestion: Scenario II

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- “perfect” retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes $\lambda'_{in}$ larger (than perfect case) for same $\lambda_{out}$

“Cost” of congestion: more work for given goodput
Causes/Costs of Congestion: Scenario III

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{in}$ and $\lambda'_{in}$ increase?
Another “cost” of congestion:

- when packet dropped, any “upstream” transmission capacity used for that packet was wasted!
Effects of Retransmission on Congestion

• Ideal case
  – Every packet delivered successfully until capacity
  – Beyond capacity: deliver packets at capacity rate

• Realistically
  – As offered load increases, more packets lost
    • More retransmissions \(\rightarrow\) more traffic \(\rightarrow\) more losses …
  – In face of loss, or long end-end delay
    • Retransmissions can make things worse
  – Decreasing rate of transmission
    • Increases overall throughput
Congestion: Moral of the Story

• When losses occur
  – Back off, don’t aggressively retransmit

• Issue of fairness
  – Social versus individual good
  – What about greedy senders who don’t back off?
Taxonomy of Congestion Control

• Open-Loop
  – Make sure congestion doesn’t occur
  – Design and provision the network to avoid congestion

• Closed-Loop
  – Monitor, detect and react to congestion
  – Based on the concept of feedback loop

• Hybrid
  – Avoidance at a slower time scale
  – Reaction at a faster time scale
Closed-Loop Congestion Control

RTT: D

source → λ → buffer → bottleneck server → μ → sink

feedback loop
Closed-Loop Congestion Control

• **Explicit**
  – network tells source its current rate
  – Better control but more overhead

• **Implicit**
  – End point figures out rate by observing network
  – Less overhead but limited control

• **Ideally**
  – overhead of implicit with effectiveness of explicit
Closed-Loop Congestion Control

• **Window-based vs Rate-based**
  – **Window-based**: No. of pkts sent limited by a window  
  – **Rate-based**: Packets to be sent controlled by a rate  
    • Fine-grained timer needed  
    • No coupling of flow and error control

• **Hop-by-Hop vs End-to-End**
  – **Hop-by-Hop**: done at every link  
    • Simple, better control but more overhead  
  – **End-to-End**: sender matches all the servers on its path
Approaches towards Congestion Control

Two broad approaches towards congestion control:

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 (SNA DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at
TCP Congestion Control

• Idea
  – Each source determines network capacity for itself
  – Uses implicit feedback, adaptive congestion window
  – ACKs pace transmission (self-clocking)

• Challenge
  – Determining the available capacity in the first place
  – Adjusting to changes in the available capacity
Additive Increase/Multiplicative Decrease

- **Objective:** Adjust to changes in available capacity
  - A state variable per connection: CongWin
    - Limit how much data source has is in transit
  - $\text{MaxWin} = \text{MIN}(\text{RcvWindow}, \text{CongWin})$

- **Algorithm**
  - Increase CongWin when congestion goes down (no losses)
    - Increment CongWin by 1 pkt per RTT (linear increase)
  - Decrease CongWin when congestion goes up (timeout)
    - Divide CongWin by 2 (multiplicative decrease)
TCP Congestion Control

- Window-based, implicit, end-end control
- Transmission rate limited by congestion window size, Congwin, over segments:

\[ \text{throughput} = \frac{w \times \text{MSS}}{\text{RTT}} \text{ Bytes/sec} \]

- \( w \) segments, each with MSS bytes sent in one RTT:
TCP Congestion Control

• “probing” for usable bandwidth:
  – ideally: transmit as fast as possible (Congwin as large as possible) without loss
  – increase Congwin until loss (congestion)
  – loss: decrease Congwin, then begin probing (increasing) again

• two “phases”
  – slow start
  – congestion avoidance

• important variables:
  – Congwin
  – threshold: defines threshold between two slow start phase, congestion avoidance phase
Why Slow Start?

• Objective
  – Determine the available capacity in the first place

• Idea
  – Begin with congestion window = 1 pkt
  – Double congestion window each RTT
    • Increment by 1 packet for each ack

• Exponential growth but slower than one blast

• Used when
  – First starting connection
  – Connection goes dead waiting for a timeout
TCP Slowstart

**Slowstart algorithm**

initialize: Congwin = 1
for (each segment ACKed)
  Congwin++
until (loss event OR CongWin > threshold)

- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)
TCP Congestion Avoidance

Congestion avoidance

/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
    Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart
Fast Retransmit/Fast Recovery

- Coarse-grain TCP timeouts lead to idle periods
- Fast Retransmit
  - Use duplicate acks to trigger retransmission
  - Retransmit after three duplicate acks
- Fast Recovery
  - Remove the slow start phase
  - Go directly to half the last successful CongWin
TCP Fairness

**Fairness goal:** if $N$ TCP sessions share same bottleneck link, each should get $1/N$ of link capacity
Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally
Dealing with Greedy Senders

• Scheduling and dropping policies at routers
• First-in-first-out (FIFO) with tail drop
  – Greedy sender can capture large share of capacity
• Solutions?
  – Fair Queuing
    • Separate queue for each flow
    • Schedule them in a round-robin fashion
    • When a flow’s queue fills up, only its packets are dropped
    • Insulates well-behaved from ill-behaved flows
  – Random early detection (RED)
More on TCP

• Deferred acknowledgements
  – Piggybacking for a free ride
• Deferred transmissions
  – Nagle’s algorithm
• Silly window syndrome
  – Clark’s solution
• Lost window advertisements
  – Send a 1 byte to elicit an advertisement by receiver
• TCP over wireless
Connectionless Service and UDP

- User datagram protocol
- Does little besides
  - Encapsulating application data with UDP header
- No connection management
- UDP segment header fields
  - Source/destination port numbers
  - Checksum, length
# TCP ACK generation

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order segment arrival, no gaps, everything else already ACKed</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>in-order segment arrival, no gaps, one delayed ACK pending</td>
<td>immediately send single cumulative ACK</td>
</tr>
<tr>
<td>out-of-order segment arrival higher-than-expect seq. # gap detected</td>
<td>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 ACK if segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
TCP Segment Structure

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td>acknowledgement number</td>
</tr>
<tr>
<td>rcvr window size</td>
<td>checksum</td>
</tr>
<tr>
<td>ptr urgent data</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>application data (variable length)</td>
<td></td>
</tr>
</tbody>
</table>

**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)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept