Good Ideas So Far…

- Flow control
  - Stop & wait
  - Parallel stop & wait
  - Sliding window (e.g., advertised windows)
- Loss recovery
  - Timeouts
  - Acknowledgement-driven recovery (selective repeat or cumulative acknowledgement)
- Congestion control
  - AIMD \(\rightarrow\) fairness and efficiency
- How does TCP actually implement these?

Sequence Number Space

- Each byte in byte stream is numbered.
  - 32 bit value
  - Wraps around
  - Initial values selected at start up time
- TCP breaks up the byte stream into packets.
  - Packet size is limited to the Maximum Segment Size
- Each packet has a sequence number.
  - Indicates where it fits in the byte stream
Establishing Connection: Three-Way handshake

- Each side notifies other of starting sequence number it will use for sending
  - Why not simply chose 0?
    - Must avoid overlap with earlier incarnation
    - Security issues
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- Can combine second SYN with first ACK

TCP Connection Setup Example

| Client SYN | SeqC: Seq. #4019802004, window 65535, max. seg. 1260 |
| Server SYN-ACK+SYN | Receive: #4019802005 (= SeqC+1) |
| Client SYN-ACK | Receive: #3428951570 (= SeqS+1) |

TCP State Diagram: Connection Setup

Tearing Down Connection

- Either side can initiate tear down
  - Send FIN signal
    - "I’m not going to send any more data"
- Other side can continue sending data
  - Half open connection
  - Must continue to acknowledge
- Acknowledging FIN
  - Acknowledge last sequence number + 1
TCP Connection Teardown Example

09:54:17.585396 IP 128.2.222.198.4474 > 128.2.210.194.6616: F 1489294581:1489294581(0) ack 1909787689 win 65434 (DF)
09:54:17.585732 IP 128.2.210.194.6616 > 128.2.222.198.4474: F 1489294589:1489294589(0) ack 1909787682 win 5840 (DF)

- Session
  - Echo client on 128.2.222.198, server on 128.2.210.194
- Client FIN
  - SeqC: 1489294581
- Server ACK + FIN
  - Ack: 1489294582 (= SeqC+1)
  - SeqS: 1909787689
- Client ACK
  - Ack: 1909787690 (= SeqS+1)

Outline

- TCP connection setup/data transfer
- TCP reliability
- TCP congestion avoidance

Reliability Challenges

- Congestion related losses
- Variable packet delays
  - What should the timeout be?
- Reordering of packets
  - How to tell the difference between a delayed packet and a lost one?
TCP = Go-Back-N Variant

- Sliding window with cumulative acks
  - Receiver can only return a single “ack” sequence number to the sender.
  - Acknowledges all bytes with a lower sequence number
  - Starting point for retransmission
  - Duplicate acks sent when out-of-order packet received
- But: sender only retransmits a single packet.
  - Reason???
    - Only one that it knows is lost
    - Network is congested → shouldn’t overload it
- Error control is based on byte sequences, not packets.
  - Retransmitted packet can be different from the original lost packet
    - Why?

Round-trip Time Estimation

- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT estimate
    - unneeded retransmissions
  - High RTT estimate
    - poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- Spurious timeouts
  - “Conservation of packets” principle – never more than a window worth of packets in flight

Original TCP Round-trip Estimator

- Round trip times exponentially averaged:
  - New RTT = \( \alpha \) (old RTT) + (1 - \( \alpha \)) (new sample)
  - Recommended value for \( \alpha \): 0.8 - 0.9
    - 0.875 for most TCP's
  - Retransmit timer set to \((b \times \text{RTT})\), where \(b = 2\)
    - Every time timer expires, RTO exponentially backed-off
    - Not good at preventing spurious timeouts
    - Why?

RTT Sample Ambiguity

- Karn’s RTT Estimator
  - If a segment has been retransmitted:
    - Don’t count RTT sample on ACKs for this segment
    - Keep backed off time-out for next packet
    - Reuse RTT estimate only after one successful transmission
Jacobson’s Retransmission Timeout

- Key observation:
  - At high loads, round trip variance is high
- Solution:
  - Base RTO on RTT and standard deviation
    - $RTO = \text{RTT} + 4 \times \text{rttvar}$
  - $\text{new}_{\text{rttvar}} = \beta \times \text{dev} + (1-\beta) \times \text{old}_{\text{rttvar}}$
    - Dev = linear deviation
    - Inappropriately named – actually smoothed linear deviation

Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current time into option
  - 4 bytes for time, 4 bytes for echo a received timestamp
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet

Timer Granularity

- Many TCP implementations set RTO in multiples of 200,500,1000ms
- Why?
  - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
  - Make timers interrupts efficient
- What happens for the first couple of packets?
  - Pick a very conservative value (seconds)

Fast Retransmit

- What are duplicate acks (dupacks)?
  - Repeated acks for the same sequence
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
  - Window update – advertisement of new flow control window
- Assume re-ordering is infrequent and not of large magnitude
  - Use receipt of 3 or more duplicate acks as indication of loss
  - Don’t wait for timeout to retransmit packet
**Fast Retransmit**

- Sequence No
- Time
- Packets
- Acks
- Retransmission
- Duplicate Acks

**TCP (Reno variant)**

- Sequence No
- Time
- Packets
- Acks
- Now what? - timeout

**SACK**

- Basic problem is that cumulative acks provide little information
- Selective acknowledgement (SACK) essentially adds a bitmask of packets received
  - Implemented as a TCP option
  - Encoded as a set of received byte ranges (max of 4 ranges/often max of 3)
- When to retransmit?
  - Still need to deal with reordering → wait for out of order by 3pkts

**SACK**

- Sequence No
- Time
- Packets
- Acks
- Now what? – send retransmissions as soon as detected
Performance Issues

• Timeout >> fast rexmit
• Need 3 dupacks/sacks
• Not great for small transfers
  • Don’t have 3 packets outstanding
• What are real loss patterns like?

Outline

• TCP connection setup/data transfer
• TCP reliability
• TCP congestion avoidance

Additive Increase/Decrease

• Both $X_1$ and $X_2$ increase/decrease by the same amount over time
  • Additive increase improves fairness and additive decrease reduces fairness

Muliplicative Increase/Decrease

• Both $X_1$ and $X_2$ increase by the same factor over time
  • Extension from origin – constant fairness
What is the Right Choice?

- Constraints limit us to AIMD
  - Improves or keeps fairness constant at each step
  - AIMD moves towards optimal point

TCP Congestion Control

- Changes to TCP motivated by ARPANET congestion collapse
- Basic principles
  - AIMD
  - Packet conservation
  - Reaching steady state quickly
  - ACK clocking

AIMD

- Distributed, fair and efficient
- Packet loss is seen as sign of congestion and results in a multiplicative rate decrease
  - Factor of 2
- TCP periodically probes for available bandwidth by increasing its rate

Implementation Issue

- Operating system timers are very coarse – how to pace packets out smoothly?
- Implemented using a congestion window that limits how much data can be in the network.
  - TCP also keeps track of how much data is in transit
- Data can only be sent when the amount of outstanding data is less than the congestion window.
  - The amount of outstanding data is increased on a “send” and decreased on “ack”
  - (last sent – last acked) < congestion window
- Window limited by both congestion and buffering
  - Sender’s maximum window = Min (advertised window, cwnd)
Packet Conservation

- At equilibrium, inject packet into network only when one is removed
  - Sliding window and not rate controlled
  - But still need to avoid sending burst of packets \( \rightarrow \) would overflow links
    - Need to carefully pace out packets
    - Helps provide stability
- Need to eliminate spurious retransmissions
  - Accurate RTO estimation
  - Better loss recovery techniques (e.g. fast retransmit)

TCP Packet Pacing

- Congestion window helps to “pace” the transmission of data packets
- In steady state, a packet is sent when an ack is received
  - Data transmission remains smooth, once it is smooth
  - Self-clocking behavior

Congestion Avoidance

- If loss occurs when \( cwnd = W \)
  - Network can handle \( 0.5W \sim W \) segments
  - Set \( cwnd \) to \( 0.5W \) (multiplicative decrease)
- Upon receiving ACK
  - Increase \( cwnd \) by \( (1 \text{ packet})/cwnd \)
    - What is 1 packet? \( \rightarrow \) 1 MSS worth of bytes
    - After \( cwnd \) packets have passed by \( \rightarrow \) approximately increase of 1 MSS
- Implements AIMD

Congestion Avoidance Sequence Plot
Congestion Avoidance Behavior

- When a loss occurs have $W$ packets outstanding
- New $cwnd = 0.5 \times cwnd$
  - How to get to new state without losing ack clocking?

How to Change Window

Fast Recovery

- Each duplicate ack notifies sender that single packet has cleared network
- When $< cwnd$ packets are outstanding
  - Allow new packets out with each new duplicate acknowledgement
- Behavior
  - Sender is idle for some time – waiting for $\frac{1}{2} cwnd$ worth of dupacks
  - Transmits at original rate after wait
    - Ack clocking rate is same as before loss

Fast Recovery

- Sent for each dupack after $W/2$ dupacks arrive

Sequence No

Packets

Acks

Time
Important Lessons

• TCP state diagram → setup/teardown

• TCP timeout calculation → how is RTT estimated

• Modern TCP loss recovery
  • Why are timeouts bad?
  • How to avoid them? → e.g. fast retransmit