CS640: Introduction to Computer Networks

Aditya Akella

Lecture 16
TCP - III
Reliability and Implementation Issues

So Far

- Transport protocols and TCP functionality overview
- Principles of reliable data transfer
- TCP segment structure
- Connection management
- Congestion control

More on Reliability

- TCP provides a "reliable byte stream"
  - "Loss recovery" key to ensuring this abstraction
  - Sender must retransmit lost packets

- 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:
  - Only one that it knows is lost.
  - Sent after timeout.
  - Network is congested → shouldn't overload it.

- Choice of timeout interval → crucial.

Round-trip Time Estimation

- Reception success known only after one RTT.
  - 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.

Original TCP RTT Estimator

- Round trip times exponentially averaged:
  - New RTT = α(old RTT) + (1 - α)(new sample).
  - Recommended value for α: 0.8 - 0.9.
    - 0.875 for most TCPs.

- Retransmit timer set to (2 * RTT):
  - Whenever timer expires, RTO exponentially backed-off.

- Not good at preventing spurious timeouts:
  - Why?
Jacobson's Retransmission Timeout

- Key observation:
  - At high loads round trip variance is high

- Solution:
  - Base RTO on RTT and deviation
    - \( RTO = RTT + 4 \times rttvar \)
    - \( \text{new}_{\text{rttvar}} = \beta \times \text{dev} + (1-\beta) \times \text{old}_{\text{rttvar}} \)
      - \( \text{Dev} \) = linear deviation
      - Inappropriately named – actually smoothed linear deviation

AIMD Implementation

- If loss occurs when \( cwnd = W \)
  - Network can handle \( W \) segments
  - Set \( cwnd \) to \( 0.5W \) (multiplicative decrease)
    - Known as "congestion control"

- Upon receiving ACK
  - Increase \( cwnd \) by \( \frac{1}{cwnd} \) packet
    - What is 1 packet? 1 MSS worth of bytes
    - After \( cwnd \) packets have passed by \( \approx \) approximately increase of 1 MSS
      - Known as "congestion avoidance"

- Implements AIMD

Control/Avoidance Behavior

- Congestion Window
- Packet loss + Timeout
- Cut Congestion Window and Rate
- Grabbing back Bandwidth
Improving Loss Recovery:
Fast Retransmit
- Waiting for timeout to retransmit is inefficient
- Are there quicker recovery schemes?
  - Use duplicate acknowledgments as an indication
    - Fast retransmit
- What are duplicate acks (dupacks)?
  - Repeated acks for the same sequence
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
- 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

Packet Pacing
- In steady state, a packet is sent when an ack is received
  - Data transmission remains smooth, once it is smooth (steady state)
  - "Self-clocking" behavior
How to Change Window

• When a loss occurs have $W$ packets outstanding
  - A bunch of dupacks arrive
  - Revisit on 3rd dupack
  - But dupacks keep arriving
  - Must wait for a new ack

• New cwnd = $0.5 \times \text{cwnd}$
  - Send new cwnd packets in a burst
  - Risk losing ack clocking

Preserving Clocking: 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 $0.5 \times \text{cwnd}$ worth of dupacks
  - Transmits at original rate after wait
  - Ack clocking rate is same as before loss

Fast Recovery (Reno)
Timeouts can still happen!

Reaching Steady State
- Doing AIMD is fine in steady state...
  - But how to get to steady state?
- How does TCP know what is a good initial rate to start with?
- Quick initial phase to help get up to speed
  - Called "slow" start (!)

Slow Start
- Slow start
  - Initialize cwnd = 1
  - Upon receipt of every ack, cwnd = cwnd + 1
- Implications
  - Window actually increased to \( W \) in RTT
  - Can overshoot window and cause packet loss
Slow Start Example

Return to Slow Start

• If too many packets are lost self clocking is lost as well
  - Need to implement slow-start and congestion avoidance together

• When timeout occurs set ssthresh to 0.5w
  - If cwnd < ssthresh, use slow start
  - Else use congestion avoidance

The Whole TCP “Saw Tooth”
TCP Performance

- Can TCP saturate a link?

- Congestion control
  - Increase utilization until link becomes congested
  - React by decreasing window by 50%
  - Window is proportional to rate * RTT

- Doesn’t this mean that the network oscillates between 50 and 100% utilization?
  - Average utilization = 75%??
  - No… this is *not* right!

Unbuffered Link

- The router can’t fully utilize the link
  - If the window is too small, link is not full
  - If the link is full, next window increase causes drop
  - With no buffer TCP achieves 75% utilization

TCP Performance

- In the real world, router queues play important role
  - Window is proportional to rate * RTT
  - But, RTT changes as well as the window
  - Window to fill links = propagation RTT * bottleneck bandwidth
  - Role of Buffers → If window is larger, packets sit in queue on bottleneck link
TCP Performance

- In the real world, router queues play an important role
  - Role of Buffers
  - If window is larger, packets sit in queue on bottleneck link

- If we have a large router queue → can get 100% utilization
  - But, router queues can cause large delays

- How big does the queue need to be?
  - Windows vary from \( W \approx W/2 \)
  - To ensure the link is always full
  \[ W > \frac{RTT \times BW}{2} \]
  \[ W > RTT \times BW + Qsize \]
  - Ensures 100% utilization
  - Delay:
    - Varies between RTT and 2 * RTT
    - Queuing between 0 and RTT

Buffered Link

- With sufficient buffering we achieve full link utilization
  - The window is always above the "critical" threshold
  - Buffer absorbs changes in window size
  - Buffer Size = Height of TCP Sawtooth
  - Minimum buffer size needed is \( 2 \times RTT \)
  - This is the origin of the rule-of-thumb

TCP Summary

- General loss recovery
  - Stop and wait
  - Selective repeat

- TCP sliding window flow control

- TCP state machine

- TCP loss recovery
  - Timeout-based
    - RTT estimation
  - Fast retransmit, recovery
TCP Summary

- Congestion collapse
  - Definition & causes

- Congestion control
  - Why AIMD?
  - Slow start & congestion avoidance modes
  - ACK clocking
  - Packet conservation

- TCP performance modeling
  - How does TCP fully utilize a link?
    - Role of router buffers

Next Class

- Naming and the DNS