Placement of Function in a Best Effort World

Course Logistics Update

- 45 total in class. Still off target.
- Prioritized waitlist now available. See me after class.

Internet Architecture Redux

- The network is stateless (w.r.t. e2e connection state), for survivability
- The network is unreliable
  - No guarantees; 1%ish drop rate OK; may burst to higher or have temp. outages
  - ➢End-hosts must do reliability/retransmission
- The network provides no QoS
  - ➢End-hosts must do congestion control

Things the Transport Faces

- Loss
  - Congestion, corruption, routing probs, failures
    - Congestion? Yup: Stat mux! Finite buffers for store-and-forward. Bursts or excess traffic.
      - Queue size is tricky: Too large == too much delay, too small == bad statmux. More later
- Variable delay
- Reordering (how can this happen?)
  - Bugs, multipath
- Duplication
  - Bugs, lower-layer spurious retransmissions

So: TCP?

- A reliable, congestion-controlled, in-order bytestream
  - As mentioned last time: Not a perfect fit for everybody. Drawbacks?
    - Unreliable apps don't need it
    - May delay app processing, causing CPU/memory/disk to be more bursty than necessary
  - Known problem. Nice paper @ SIGCOMM 2006 on DCCP, datagram congestion control: Unreliable, congestion controlled datagrams
  - Also mentioned ITP, image transport protocol: reliable *out of order*
  - Rel to end-to-end arguments?

Reliable Transfers

- Forward Error Correction: (redundancy in-band)
- Automatic Repeat reQuest (ARQ): retransmissions. How does it know?
  - Acknowledgements!
  - In tcp: cumulative ACKs
- How do you detect a loss with ACKs?
  - Timeout
  - Or – (diagram) notice that you're getting dup ACKs

TCP Gets Older

- V1: “go-back-N” retransmissions based on timeouts with a fixed-size sliding window
- V2: Congestion collapse!
  - Van Jacobsen and Mike Karels congestion avoidance (congestion control), better RTT estimators (why? To set the timeout), and slow start. (Coming up later)
- Karn & Partridge: Retransmission Ambiguity
  - How do you tell if an ACK is for orig or Rx packet? If you can’t tell, your RTT estimator can break
  - Next step: TCP timestamp options

TCP Variants

- TCP Tahoe: Fast retransmission + Jacobson/Karels (slow start/cong avoid)
  - First data-driven, not timeout-driven, Rxmit
- TCP Reno: Fast recovery: The now-classic sawtooth
- TCP NewReno: improves fast recovery to survive multiple data losses in large windows
- TCP SACK: Pretty close to the state of the art. Instead of just ACKing last received seq, tell sender about other recv’d packets too (easier to recover from weird loss patterns)
- Most machines today are NewReno / SACK (Sally Floyd 2005 study)

TCP Principles

- Don’t retransmit early; aoid spurious retransmissions
  - A response to congestion collapse. Part of cong. collapse was Rx of packets that were queued, not lost.
- Conserve packets
  - Don’t blast network with extra traffic during retransmissions.
  - General technique: Count # dup acks to know how many packets have left the network.

Timers

- How do you know when packets were lost? If no other ACKs, must timeout.
- How long to wait? Well, depends on the RTT.
  - Easy to measure: segment -> ACK
  - Problems: Variable delay, variable processing times at receiver
  - Solution: Averaging

EWMA

- Exponential Weighted Moving Averages
  - A low-pass filter by another name
  - Srtt = alpha * r + (1 – alpha)*srtt
    - R is the current sample
    - Srtt is the smoothed average
    - TCP uses alpha = 1/8. Doesn’t matter too much.
  - Great! We’re done.
  - Or not.
- EWMA is great to put in your toolbox

Variance

- If we use srtt, then half of our transmissions are spurious.
- TCP early solution: Beta * srtt (beta =~ 2)
  - But what if the path has high variance?
- Real solution:
  - RTO = srtt + 4 * rttdev
  - Rttdev = mean linear deviation (>= stddev)
    - = rttdev = gamma * dev + (1-gamma) * rttdev
    - Dev = |r – srtt|. TCP uses gamma = ¼
- Final note: What to do on timeout?
  - Exponential backoff (another good toolkit thing)

Using more information: Fast Retransmission

- Data-driven retransmission
- What happens on loss? Duplicate ACKs
  - Imagine you sent:
  - 1:1000, 1001:1700, 2501:3000, 3001:4000, 4001:4576
  - And got ACKs 1001 1701 1701 1701 1701
  - Clear that something's going wrong!

Wither dup acks

- Why DUP acks?
  - Window updates: TCP receivers have limited space in socket buffer, so they can tell the source to "shut up" (flow control)
  - Segment loss
  - Or … segment re-ordering
    - TCP says three dupacks == not re-ordered
    - Works pretty well, but now forces network design to abide by it because it works pretty well!
      - Per-packet load balancing

Congestion Control

- Okay. Great. We know we had a loss because of a timeout or dup ACKs. What do we do?
  - 1: Retransmit
  - 2: Adjust our congestion window
    - TCP isn't just doing reliability…

The basics

- Slow start: Ramp up
- Congestion avoidance: Be conservative when you're near the limit

Fast Recovery

- There’s still more information floating in the net.
- If you did fast recovery, you got dup ACKs, and will probably keep getting more.
- Retransmit lost packet. Cut cong window in half. Wait until half of the window has been ACKed, and then send _new_ data.
- Basic idea in TCP Reno.
- TCP NewReno adds more tricks to deal with multiple losses in a window, which kills Reno.

SACK

- You can get still more information!
- 1:1000 1001:1700 2501:3000 3001:4000 4577:5062 6000:7019
- Send ACKs
- 1001 1701 1701 [2501-3000] 1701 [2501-4000] 1701 [4577-5062; 2501-4000] etc.
- Aimed at Long Fat Networks (LFNs; pronounced “Elephants”). Standardized in RFC2018 after years of debate.
- SACK isn’t perfect! If TCP window is very small, not enough Rx to deal with it.

Other tricks in TCP

- Three way handshake: Establishes the sequence # space with both sender and receiver
- Segment size: How big can the network support?
  - Path MTU discovery
  - Set IP “Don’t Fragment” bit.
    - Routers with smaller MTU send back ICMP error message containing their MTU
- Low-bandwidth links: TCP Header Compression.
  - Most fields in TCP header stay the same. Can be compressed on a link-by-link basis.
    - 40 byte TCP+IP header in ~3-6 bytes.

ALF

- Example: Streaming video protocol
- Consider MPEG:
  - Reference frames
  - Difference frames (for simplicity, vs previous frame)
- Problem: Propagation of errors

Video over TCP?

- Completely reliable delivery. Solves the problem!
- But we talked about the problems with this earlier.
  - Common issue: Real-time or quasi-real-time playback...
  - Even in delayed playback, need larger buffers

A better way

- Some packets need to be more reliable (e.g., reference frames): Selective Recovery
- Eliminate delays: Out of order delivery
- Option 1: custom protocol over UDP.
    - Painful!
- Option 2: Hack TCP to do out-of-order
  - Hard! Remember byte-stream abstraction. No way for TCP to communicate about what part of data it's providing

Ergo, ALF and ADUs

- Every application can define some kind of ADU
  - Some are a bit awkward: telnet
  - Better examples: Movie frames, JPEG 8x8 pixel units, file blocks
- Make the protocol data unit the same
  - TCP's PDU is a byte. Not the same.
- ALF and ADUs are a very powerful way to think about protocols that need out of order / selective reliability

ALF in the real world

- Hard to do in-kernel while getting the interface right. (Application naming, etc., all very tied together)
- In practice: A library
- Best example: RTP, the Real-time Transport Protocol
  - Used for audio, video, whiteboarding, etc.
- Modern example: DCCP
- (These are all post-ALF protocols)

Benefits of ALF

- Application defines namespace
- Can send and receive data in units meaningful to the application
- But: Library provides things like
  - Congestion control / flow control
  - Reliability as needed
  - Multiplexing/demultiplexing
  - Path MTU discovery

Summary

- Internet architecture forces smart endpoints.
- TCP is the grab bag: in-order, reliable, duplex, byte-stream. Timer-driven and data-driven retransmissions. Congestion control.
- ALF is a principle for how you might structure a transport protocol aimed at non-bytestream applications.