Lecture 25: Congestion Avoidance and Adversaries

Mothy Roscoe and Joe Hellerstein

November 22, 2005

Congestion Avoidance

Key idea: “conserve packets”. Appeals to a fluid flow model of the network – when operating close to capacity, flow in == flow out.

1. Slow start (how to get close to capacity):
   - add a second, “congestion” window (emphcwnd) and use the minimum of this and the receiver’s window.
   - *cwnd* starts out at 1 packet (segment), adds one for each (successful) ack received.
   - $\Rightarrow$ each received ACK causes 2 packets to go out, since (1) a packet has been received at the far end, and (2) our window is now bigger.
   - After losing a packet (not ACKed), reduce *cwnd* to 1 and start again.
2. Round-trip time estimator (how to stay roughly at capacity when you get there)
   - Need to incorporate RTT variance into the estimator. This is generally a good thing (see paper for maths), but the variance goes up a lot during congestion
   - Underestimating variance is bad: causes retransmissions when packets (or ACKs) are still in flight $\Rightarrow$ violates conservation.
   - Estimating the variance continuously (versus using a fixed value) helps in low-variance but high-latency links, such as satellites.
   - Need exponential backoff if retransmitting packets (c.f. Ethernet)
   - Assumes a reasonably symmetric link. Systems like DirectPC (satellite downlink at 25Mb/s, dialup uplink at 33.6kb/s) have “interesting” interactions with TCP RTT estimation.

3. Packet loss $\Rightarrow$ congestion (detecting congestion)
    - Only makes sense if you've got RTT estimation right (see point 2)
    - Multiplicative decrease to rapidly return to stability
    - Additive increase probes for extra bandwidth: produces characteristic sawtooth graph under congestion-free conditions.
    - Need additive increase whenever $cwnd < rxwnd$.
    - Now a central tenet of TCP, but doesn't work well in environments where a given loss is unlikely to be due to congestion (such as wireless)
    - Alternative is to explicitly signal congestion in the ACK - see ELN, ECN, etc. proposals. Unfortunately, seem to suffer "feature interaction" with other parts of the Internet.

Strengths:

- Good practical solution to a pressing problem (prevented imminent congestion collapse of the Internet)
- Nice mix of control theory intuition and Real Code

Flaws:

- Not totally clear about its assumptions (for example, doesn't consider any adversaries at all)
- While the control theory makes sense, a lot of the protocol design has "issues" (see below)

Misbehaving receivers

- It's pretty clear that misbehaving senders can abuse congestion control by sending too much data
- Perhaps surprising that receivers can abuse it by inducing an innocent sender to send too much data
- This is an attack on the TCP spec, not any particular implementation!
- $\Rightarrow$ all valid implementations vulnerable
- Receivers have opportunity: source code readily available (and why not?)
- Receivers have motive: web surfing goes faster!

- Question: I wonder how all those advertised “Internet accelerators” for Windows work?

Abadí and Needham principles:

1. Every message should say what it means: the interpretation of the message should depend only on its content
2. The conditions for a message to be acted upon should be clearly set out so that someone reviewing a design may see whether they are acceptable or not
3. If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal’s name explicitly in the message

TCP Daytona

- Note: name is a pun on TCP Tahoe, Reno, Vegas, etc. It’s fast!
- Almost no code for each of the three attacks
- Evaluation: looks like they tried this against `cnn.com`.
- Linux has solution to first attack (even though the attack is on the spec, not on particular implementations)
- Windows NT not vulnerable to attack 2 because it never enters fast retransmit anyway ...
- Otherwise: Daytona seems widely applicable and show dramatic performance improvements!

Attack 1: ACK division

- Each ACK increases cwnd by 1 segment, even if you ACK part of a segment (such as 1 byte)
- Congestion window completely open in two RTTs (for typical web browser)
- Analysis: violates Principle 2 – spec assumes that ACKs occur at segment boundaries, but message format allows arbitrary bits of sequence space.
- Also: seems to be a confusion between congestion based on byte counts or segment counts. Either seems reasonable, but both leads to ambiguity.
- Solution 1: Operate entirely on a byte granularity (forces message format to fit the spec better.

- Solution 2: Wait till an entire segment has been ACKed before updating *cwnd* (what Linux does) – doesn't require protocol to be changed.

Attack 2: Duplicate ACK spoofing

- Send multiple acks for the same sequence number, causing the sender to enter "fast retransmit" and increase *cwnd*.
- Fast recovery: not covered in the Van Jacobson Paper, but aim is to avoid single losses slamming the *cwnd* completely shut.
- Under many conditions, turns out that without fast retransmit TCP would make next to no progress due to over-zealous congestion avoidance.
- Analysis: during fast retransmit / fast recovery, Principle 1 is violated – duplicate ACKs can mean different things: segment was lost, or some later segment was received (thus, opening *cwnd* further). ACKs are being "overloaded", and there's no way for the sender to associate a given duplicate ACK with a particular segment at the far end.
- Solution: *singular nonce* to prove that packet was received. Only increment window for each proven received packet. Note: Can't do this without changing protocol (to include nonce and nonce reply).
- General lesson: nonce is "fresh" (i.e., unpredictable) information. You'll see this general mechanism used a lot, not simply in "real" cryptographic protocols.

Attack 3: Optimistic ACKs

- ACK the data before you actually receive it.
- Increases congestion window, and reduces the RTT estimate – both of which overclock the sender!
- You might ACK data the sender hasn't sent yet. Which is a bit embarassing, but probably has no effect
- You might ACK data that has been sent, but lost en route. TCP will not allow you to recover this data
  - One "solution" is an additional, application-level channel to fetch the missing data (such as byte-ranges from web servers).

- Analysis: violates 3rd principle: the principal here is the segment to which the ACK corresponds. It would be prudent if the ACK refered to it explicitly, hence:
- Solution: *cumulative nonce* ensures all data has been received before it can be acked.

Lessons

- The nature of the contract (and in particular, the trust) between sender and receiver in TCP had not been clearly stated, or investigated.
- The “self-clocking” nature of TCP leads to problems: all that a node has to go on is what the other side tells it.
- Nonces constrain what the other side can say, and when it can say it, so that the best it can do is limit its performance.