TCP Congestion Control [Jacobson88a]

(started Feb. 9; finished Feb. 16)

CSci551: Computer Networks
SP2006 Thursday Section
John Heidemann

Agenda

- connection setup and teardown
- flow control
- congestion control theory
- **congestion control practice (in TCP)**
  - **slow start**
  - **congestion avoidance**
- **loss recovery**
- **putting it together**
- security
- performance

TCP Congestion Principals

- underlying principle: *packet conservation*
  - at equilibrium, inject packet into network only when one is removed
  - basis for stability of physical systems
- components:
  - how to get there: slow start
  - how to stay there: congestion avoidance
  - if we overshoot: congestion recovery

Key ideas

- [Jacboson88a]:
  - new algo to calculate RTO, RTT variance
  - congestion control for TCP
  - fast retransmission
- problem: network was not stable
  - why?
    - congestion
    - oscillations in congetion

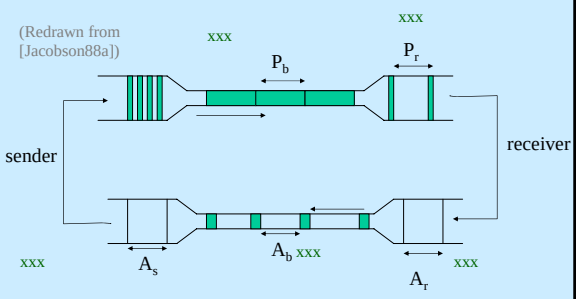
TCP Congestion Control

- three mechanisms:
  - slow start: goal: come up to speed quickly and determine what bitrate the network can support
  - congestion avoidance: goal: TCP at "equilibria", goal stability (don't be aggressive), but keep adding a little bit of tfc to see if there's more room in the network, also to eventually converge on a "fair" allocation
  - congestion recovery: goal: backoff after congestion (indicated by loss or ECN bit) and let network recover; go to slow start (cwnd=1) and halve ssthresh
- interacts very closely with loss repair:
  - good retransmit timeout (RTO) estimation
  - fast retransmit and recovery
  - why?
    - main signal from the net about congestion or overutilization is loss
    - we must *exepct* loss as part of regular TCP operation

TCP Congestion Control Mechanisms

- new congestion window *cwnd*
  - what the *network* can handle
  - vs. flow control window (*wnd*): what the other end can handle
- sender limits tx
  - min (*wnd*, *cwnd*)
  - (and also considers outstanding data)

TCP Self-clocking



depends on ACK stream to keep packets flowing

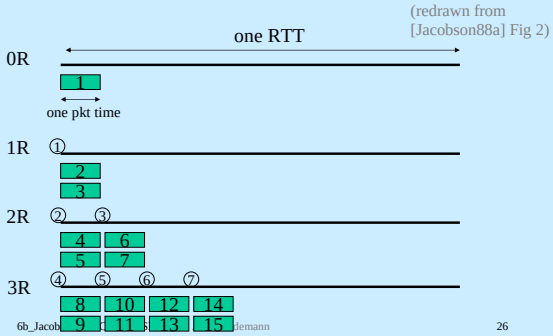
Three Phases

- slow start
  - enter: when start OR after loss (usually)
  - exit: when cwnd > ssthresh
- congestion avoidance
  - enter: after slow start (see above)
  - exit: after loss (go to congestion control)
- congestion control
  - what:
    - ssthresh := cwnd/2; cwnd=1 pkt
    - could do fast retx, or maybe just slow start
  - enter: after loss in either cong avoid or slow start
  - exit: immediately go to slow start

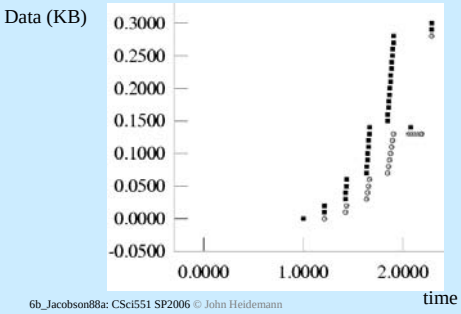
Slow Start

- How do we get the ACK clock started?
  - Initialize cwnd = 1
  - Upon receipt of every ACK, cwnd = cwnd + 1
- Implications
  - how much in each RTT? exponential increase
  - Will overshoot window and cause packet loss (but remember, packet loss is part of the plan)

Slow Start Example



Slow Start Time-Sequence Plot



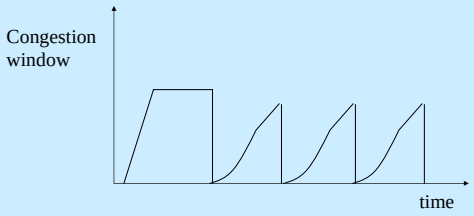
When to End Slow-Start?

- *Want* to end when the pipe is full
  - *do* end when *cwnd* > *ssthresh*
  - start with large *ssthresh*, but then refine it
- On packet loss
  - *cwnd*=1 and go back to slow start
  - *ssthresh* = *cwnd* / 2
    - assume that pipe size was somewhere between last good window (*cwnd*/2) and current window (*cwnd*)
- Eventually, *ssthresh* is right and transition to *congestion avoidance* without packet loss

Congestion Avoidance

- upon receiving ACK
  - Increase cwnd by 1/cwnd
  - This is *additive* increase (over 1 RTT it adds up to increasing by 1 segment)
- why not multiplicative increase?
  - don't want to overly congest the network
  - assume ssthresh is good estimate of target rate, so increase slowly after that

Congestion Window



Problems So Far

- have way to fill pipe (slow start)
- have way to run at equilibrium (congestion avoidance)
- but tough transition
  - no good initial *ssthresh*
  - large *ssthresh* causes packet loss, *every time*
  - ⇒ need approaches to quickly recover from packet loss (or explicit signal of congestion)

Agenda

- connection setup and teardown
- flow control
- congestion control
- **loss recovery**
- security
- performance

TCP Loss Recovery

- **timeout and retransmit**
- fast retransmit
- fast recovery
- New-Reno partial ACKs
- SACK

Fallback Mechanism: Timeout

- *retransmission timer* (RTO)
  - if no ACK after RTO fires, reset *cwnd* and resend lowest unACK'ed segment
- but they're very crude
  - completely stop the ACK clock
  - force slow-start again
  - are often slow—a long time with no traffic
- …but is there more information?

Digression: RTO Calculation

- Must *estimate* RTO
  - don't know it at start
  - may change due to congestion or path change
- But need a good estimate
  - too low => timeout too quickly, introduce duplicate pkts into the net
  - too high => when loss happens, you figure it out only later than you should => low performance

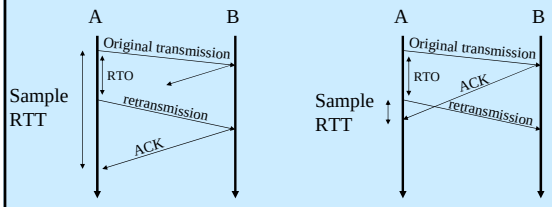
Initial Round-trip Estimator

Round trip times exponentially averaged:

- New RTT = α (old RTT) + (1 - α) (new sample)
- Recommended value for α: 0.8 - 0.9
- Retransmit timer set to β RTT, where $\beta = 2$
- Every RTO expiration, increase it multiplicatively

Retransmission Ambiguity

what does the ACK indicate?



Karn's Retransmission Timeout Estimator

- Accounts for retransmission ambiguity
- 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 Estimator

- Key observation:
  - Using β RTT for timeout doesn't work

  (not adaptive enough with fixed β: at high loads, *variance* is high)
- Solution:
  - If D denotes mean variation (measured)
  - Timeout = RTT + 4D
  - β is now *adaptive*

(and can do it with integer math in few l.o.c.)

TCP Loss Recovery

- timeout and retransmit
- **fast retransmit**
- **fast recovery**
- New-Reno partial ACKs
- SACK

Why is recovery important?

- goal is high utilization
- want recover quickly because slow start is a big penalty
  - to recover quickly, need to try and keep the ACK clock going
  - so need to keep some packets in flight
- and have to recover the packet
  - want to quickly determine what was lost and resend it

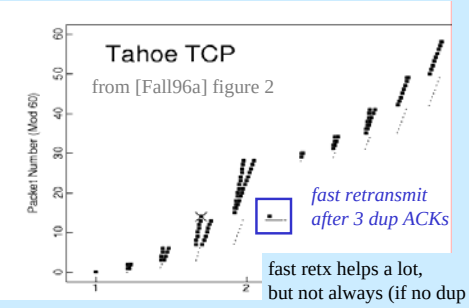
Fallback Recovery: Timeout and Retransmit

- how
  - timeout after RTO
- pros
  - simple (no help from routers)
  - doesn't require extra net traffic (like NACKs)
  - can depend on this
- cons
  - need good RTO estimation
  - need timers (can be expensive)
  - can be slow (at most one pkt retx per RTT)

Fast Retransmit

- Interpret *n* duplicate ACKs as loss indication
  - in fact, send a dup ACK for every packet you get after a missing one
  - *but beware:* now packet re-ordering causes problems
- Goal: avoid RTO by fixing the one missing segment

Fast Retransmit Example



fast retx helps a lot,
but not always (if no dup ACKs)

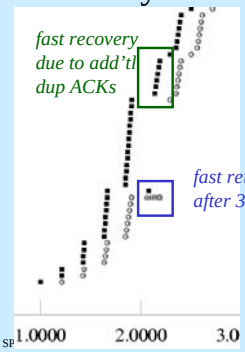
Fast Recovery [see Fall96a]

- Problem: fast retx still forces slow-start, breaking the ACK clock
- Fast Recovery Solution: artificially *inflate* the *cwnd* as more dup ACKs come in
  - cut *cwnd*, but instead of slow start, do additive increase for each ACK
  - justification: each dup ACK represents a packet leaving the network, so we can increase *cwnd*
  - exit when out of dup ACKs

Fast Retransmit and Recovery

- If we get 3 duplicate ACKs for segment *N*
  - Retransmit segment *N*
  - Set *ssthresh* to 0.5**cwnd*
  - Set *cwnd* to *ssthresh* + 3 [why?]
- For every subsequent duplicate ACK
  - Increase *cwnd* by 1 segment
- When new ACK received
  - Reset *cwnd* to *ssthresh* (resume congestion avoidance)

Fast Recovery Example

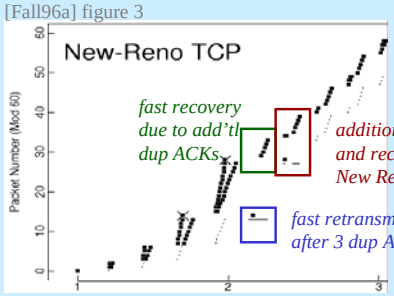


New-Reno Partial ACKs

- But fast retx and recovery only repair one lost segment per RTT
- New-Reno idea: use *partial ACKs* to stay in fast recovery and fix more lost segments
- (But there are diminishing returns here…)

New-Reno Example

from [Fall96a] figure 3



SACK

- Forget these hacks, have receiver just tell sender what's missing

⇒ SACK: selective acknowledgement

- use TCP options to encode some info about multiple losses and avoid all of this guess work
- but why is SACK deployment so much slower than Reno, New-Reno?

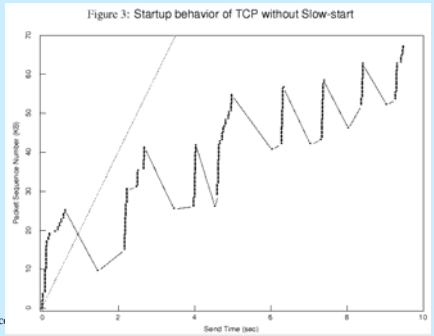
Agenda

- connection setup and teardown
- flow control
- congestion control theory
- congestion control practice (in TCP)
- loss recovery
- **putting it together**
- security
- performance

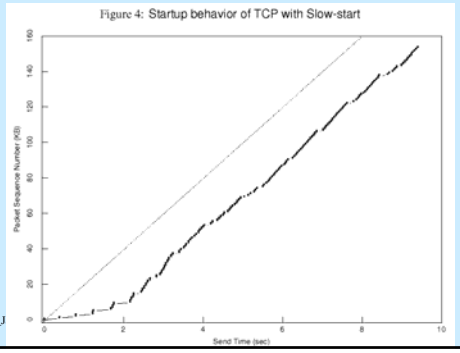
Jacobson Observations

- Compare Figure 3 vs. Figure 4
- Compare Figure 8 vs. Figure 9

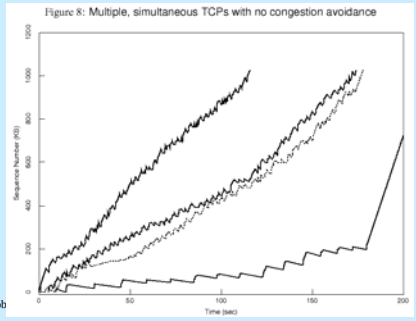
Jacobson88a Figure 3: No CC



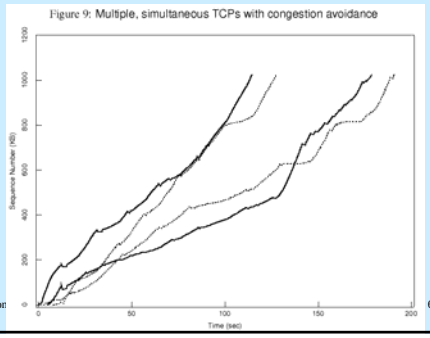
Jacobson88a Figure 4: with CC



Jacobson88a Figure 8: 4x no CC



Jacobson88a Figure 9: 4x w/CC



Animations

- see classroom animations of
  - stop and wait
  - slow-start
  - fast retransmit

Other questions/observations?

- xxx