Internet Programming & Protocols
Lecture 10

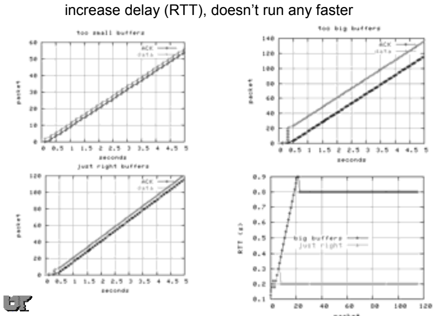
TCP RTT estimation
Tiny packets – delayed ACKs, Nagle, silly windows
TCP timers
TCP slow-start

Plan of attack

- Network overview ✓
- BSD sockets and UDP ✓
- TCP
  - Socket programming ✓
  - Reliable streams ✓
  - Header and states ✓
  - Flow control and bandwidth-delay ✓
  - Measuring performance ✓
  - Historical evolution
  - Congestion control
- Network simulation (ns)
- TCP accelerants
- TCP implementations
- TCP over wireless, satellite, …

Bandwidth delay product

- Buffers too small, and you run slow 86kb, just right is 183 kbs
- Buffers too big, consume host/net resources, may cause congestion, increase delay (RTT), doesn't run any faster



Applet data	
User buffers	
16K	**18%**
64K	**55%**
128K	**7%**
256K	**11%**
Bigger	**8%**

Timeouts and RTT estimation

- TCP handles lost packets with a send timer for data packets.
  - If the data packet is lost, or the returned ACK is lost, the timer will expire and TCP will retransmit the lost packet, restarting the timer.
    - RFC 793 says nothing about backoff (that came later)
    - RFC 793 says nothing about the receiver retaining out of order packets, so if sender is using N-packet window, on a timeout, it may re-transmit any other packets that were sent after the missing packet (go-back-N)
- What to use for a timeout value?
  - Too small, and sender may unnecessarily re-transmit, congest network.
  - Too large, and application performance may suffer
  - Need to wait at least one RTT time
  - But RTT may vary
    - Routing path may change
    - Queuing delays at various routers or even at destination host
    - Need dynamic estimate of RTT
      - Note time when segment sent, then when ACK arrives == RTT

RTT estimation (RFC 793)

Since RTT will be fluctuating, RFC 793 suggests a weighted average
$R \leftarrow \alpha R + (1-\alpha)M$
where R is current RTT estimate and M is latest measurement and α is 0.9
The timeout value (RTO) is βR where $\beta=2$ (in RFC 793) to account for RTT variations
RFC 793: RTO min 1 sec, max 60 s

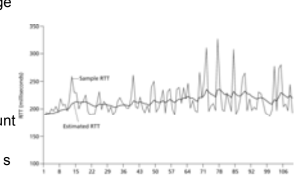


Figure 3.32 • RTT samples and RTT estimates

© Kurose

Jacobson ('88) notes that this β can adapt to loads of at most 30%. Above that point, a connection will respond to load increases by retransmitting packets that have only been delayed. This is useless work for the network and can lead to congestion collapse.

Jacobson RTT estimator

- Jacobson extends the RFC 793 estimator by keeping track of the variance in the RTT. RTO is calculated based on both the mean and variance.
- To keep the arithmetic simple in the kernel, mean deviation (E) is used to approximate standard deviation.

$E = M - R$

$R \leftarrow R + g E$ g is the gain (1/8)

$D \leftarrow D + h(|E| - D)$ D is smoothed mean deviation

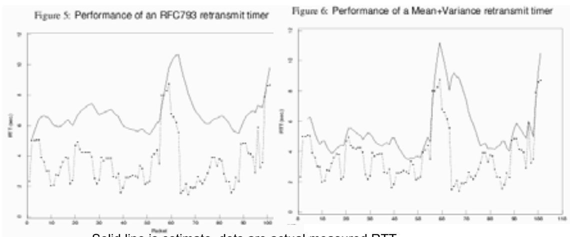
h is ¼

$RTO = R + 4D$

arithmetic can be done with shifts and "implied" binary fixed point

Implemented in 4.3BSD. Initial R is 0, initial D is 3 seconds, initial RTO is 6 seconds. Same bounds on RTO (min 1 sec, max 60 secs)

RTT estimators

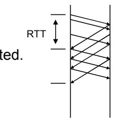


Solid line is estimate, dots are actual measured RTT

This better RTT estimator is one form of TCP congestion control – better timeout values.

Packet burst can cause sudden queue build up, difficult for estimator to track burstiness

4.3 BSD (Tahoe) RTT estimation



- Jacobson's RTT estimator incorporated in 4.3 BSD
- when kernel sends a TCP data packet for a flow, timer is started.
  - Actually a counter updated when TCP 500 ms ticker fires
  - TCP stores sequence number and tick counter value
- Usually only time one packet per RTT
  - Other packets in the available window could be sent but not timed
- When ACK for packet arrives note current tick count, calculate RTT and update estimators and RTO, start new timer if unACK'd data in flight
- Ambiguity when ACK arrives for re-transmitted packet
  - Karns: don't do RTT estimation on re-transmitted packets
- On each tick interrupt (500 ms), check if packet has "timed out"
  - Subtract stored tick value from current tick counter, if greater than RTO, retransmit
- New (optional) timestamp option permits easier RTT calculations (later)

Exponential backoff

- Jacobson '88 (BSD 4.3)
- Exponential backoff when same packet has to be retransmitted
  - Like Ethernet, if congestion is bad, keep backing off so congestion reduced
  - First try after RTO seconds, then 2*RTO, then 4*, then 8* …
    - Finite number of tries then close connection
- Why exponential backoff?
  - Network is good approximation to a linear system
    - Composed of linear operators (integrators, delays, gain stages,…)
    - Linear system theory says, if system stable, the stability is exponential
    - If network is unstable (due to random load shocks), stabilize with exponential damping to excitation sources (e.g., senders)

Lost connection and exponential backoff

- Connection breaks while sender is sending data. Retransmit with exponential backoff, eventually write() fails (connection closed (RST))

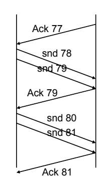
```
14:04:09.729116 thdsun.1566 > victory.7654: P 1:6(5) ack 1 win 16384
14:04:09.729116 victory.7654 > thdsun.1566: P 1:6(5) ack 6 win 32736 (DF)
14:04:09.779118 thdsun.1566 > victory.7654: . ack 6 win 16379
                Tom pulls out Ethernet cable
14:04:26.079726 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:04:26.679749 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:04:28.679824 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:04:32.679973 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:04:40.680271 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:04:56.680869 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:05:28.682063 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:06:32.684451 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:07:36.686838 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:08:40.689225 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379
14:09:44.691611 thdsun.1566 > victory.7654: P 6:11(5) ack 6 win 16379


exponential backoff, max 11 tries  (not in RFC 793, but in 4.3 BSD and later)
```


RFC 793 tweaks

- Jacobson's fixes help TCP perform better under heavy load, reducing unnecessary re-transmissions and backing off in the face of packet loss
- Jacobson also implemented additional congestion control features (next time)
- The other problem noted with TCP flows in the 80's was too many tiny packets
  - Early implementations sent an ACK for each data packet and another ACK packet for the window advertisement, then maybe another packet with the reply data
  - Small changes in the receiver's advertised window (as application consumed data from receive buffer) resulted in ACK packets carrying new window info (Silly Window syndrome)

Delayed ACKs

- RFC 813 '82
- Receiver should delay sending an ACK in the hopes that it can be piggybacked on data (timer typically 200 ms, max 500 ms)
- Receiver should only ACK "immediately" out of order packets or if 2nd packet arrives before timer expires
  - Steady flow, ACKing every other packet
- If receiver has data to send back, you won't see delayed ACKs
- Good news ☺
  - a delayed ACK can substantially reduce protocol processing overhead by reducing the total number of packets to be processed
  - Reduce packet load on network
- Bad news ☹
  - excessive delays on ACK's can disturb the round-trip timing
  - delay can disturb packet "clocking" algorithms
  - Delay can reduce TCP bandwidth (slow-start)
  - Broken implementations ("stretch ACK")



TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 200ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

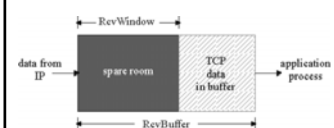
Tinygrams and Nagle algorithm

- Interactive sessions (telnet/rlogin) generate a packet per keystroke
  - 40 bytes of header, one byte of data, then the return ACK, then echo data
  - On congested wide-area nets these tiny-grams contribute to congestion
- data coalescing (Nagle '84, RFC 896)
  - Connection can have only one outsanding "small" segment
  - Sender should collect small amounts of data and send in one segment
    - Delay 200 ms before sending
  - Or if ACK comes in, send the next segment.
    - Delayed ACK can slow things even more
  - Often enabled by default, some applications (X for mouse movements) need to disable Nagle (setsockopt(), SO_NODELAY)

Silly window syndrome

- Another source of small packets in the early 80s resulted from TCP's flow control mechansim
- Some receiver side implementations would send an "available window" update (empty ACK) each time the application read a little more data
- Some sending side implementations would send a wee bit of data any time the "available window" allowed
- Lots of empty ACK packets, lots of tiny data packets … inefficient

TCP Flow control: available window from receiver



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

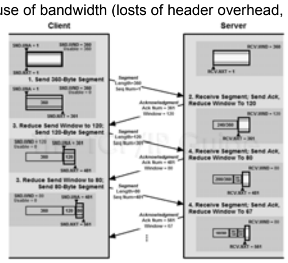
= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- Rcvr advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow

Silly window syndrome

- Receiver slowly consuming receive buffer data. New available window update for each nibble…
- Sender agressively sends a few bytes whenever the receive window allows
- Inefficient use of bandwidth (losts of header overhead, lots of packets)



Silly window syndrome fixes

- Receiver: don't advertise small segments
  - Only advertise new window if it bigger than ½ MSS or ½ receiver's buffer space (whichever is smaller)
- Sender (RFC 896, Nagle): only transmit if
  - A) full-sized segment can be sent
  - B) can send at least one-half the advertised window
  - C) send everything we have when no outstanding data (not expecting an ACK) and when Nagle is disabled
    - If Nagle enabled, wait 200 ms before sending tinygram

Echo delayed

- sending 2K packet to echo server, we get Nagle'd and delay ACK'd

```
menkar sending 2k packets to echo server on whisper
39.975513 MENKAR.2319 > whisper.8989: . 1:1461(1460) ack 1 win 8192
39.975562 whisper.8989 > MENKAR.2319: . ack 1461 win 30660 (DF)
39.976000 MENKAR.2319 > whisper.8989: P 1461:2001(540) ack 1 win 8192
39.978696 whisper.8989 > MENKAR.2319: . 1:1461(1460) ack 2001 win 32120 (D
```

200 ms, whisper doesn’t want to send just 540 bytes, menkar is delaying ACK

```
40.178545 MENKAR.2319 > whisper.8989: . ack 1461 win 8192
40.178591 whisper.8989 > MENKAR.2319: P 1461:2001(540) ack 2001 win 32120
40.183020 MENKAR.2319 > whisper.8989: . 2001:3461(1460) ack 2001 win 8192
40.183134 whisper.8989 > MENKAR.2319: P 2001:3461(1460) ack 3461 win 32120
40.183507 MENKAR.2319 > whisper.8989: P 3461:4001(540) ack 2001 win 8192
40.194201 whisper.8989 > MENKAR.2319: . ack 4001 win 32120 (DF)

40.378562 MENKAR.2319 > whisper.8989: . ack 3461 win 8192
40.378596 whisper.8989 > MENKAR.2319: P 3461:4001(540) ack 4001 win 32120
40.383012 MENKAR.2319 > whisper.8989: . 4001:5461(1460) ack 4001 win 8192
40.383118 whisper.8989 > MENKAR.2319: P 4001:5461(1460) ack 5461 win 32120
40.383501 MENKAR.2319 > whisper.8989: P 5461:6001(540) ack 4001 win 8192
40.394205 whisper.8989 > MENKAR.2319: . ack 6001 win 32120 (DF)
```

- 2KBytes every .2 seconds ☹
- TCP_NODELAY on server would fix it

TCP debugging

- netstat –s

```
Tcp:
    1079 active connections openings
    1433 passive connection openings
    0 failed connection attempts
    13 connection resets received
    2 connections established
    49209819 segments received
    97764784 segments send out
    149 segments retransmited
    0 bad segments received.
    297 resets sent

      …. Plus a bunch more “extended” status info
```

- SO_DEBUG socket option
  - Some kernels will trace TCP events for a socket (Solaris, not linux)
  - Use trpt to examine trace buffer (small, circular)
  - Include TCP states
  - Some TCP clients (telnet, ftp) support “debug”

TCP trace of ttcp over 100 mbs FDDI

FDDI (MTU 4352) ttcp (1K) between two Sun's -- slow???
sender's trace. sender window 2K, receiver window 8k

TCP state info

```
0.36 SYN_SENT:output [677d2a00..677d2a04)@0(win=1000)<SYN> -> SYN_SENT
0.36 SYN_SENT:input 2e93d400@677d2a01(win=1000)<SYN,ACK> -> ESTABLISHED
0.36 E:output 677d2a01@2e93d401(win=1000)<ACK> -> ESTABLISHED
0.37 E:output [677d2a01..677d2e01)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED
0.37 E:output [677d2e01..677d3601)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED
0.37 E:input 2e93d401@677d2e01(win=8000)<ACK> -> ESTABLISHED

0.54 E:input 2e93d401@677d3601(win=8000)<ACK> -> ESTABLISHED
0.54 E:output [677d3601..677d3e01)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED

0.74 E:input 2e93d401@677d3e01(win=8000)<ACK> -> ESTABLISHED
0.74 E:output [677d3e01..677d4601)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED

0.94 E:input 2e93d401@677d4601(win=8000)<ACK> -> ESTABLISHED
0.94 E:output [677d4601..677d4e01)@2e93d401(win=1000)<ACK,PUSH> -> ESTABLISHED
```

- If the sender's window size is less than 2*MSS, the receiver will delay after every data packet (waiting for a 2nd full-sized segment -- which is not going to arrive). 2KB/200ms = 80kbs Delayed ACK effect

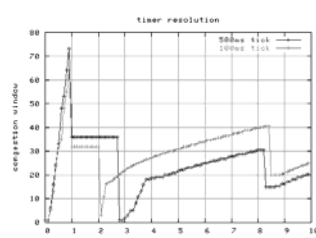
TCP keepalive

- Not part of RFC spec
- Idle TCP connection exchanges no data, could sit forever
  - Intervening routers/link could go up/down, as long as end hosts don’t crash
- Some applications want to know if the other end is still there
  - Might need this to free up resources associated with connection
  - Could do this with application packets
  - Most OS’s TCP provide keepalive option SO_KEEPALIVE
    - TCP will send a packet every 2 hours on idle connection
    - Sequence number one less than next sequence number and no data
    - Receiver should just ACK it
    - If packet is lost (after normal retries), connection is closed and application is informed (error in read/write/select)
- Controversy
  - Can cause perfectly good connections to be dropped during transient failures
  - Consume unnecessary bandwidth

TCP timers & timeouts

- connect timeout: 75s
- delayed-ACK timeout: 200ms
- keepalive: 2 hr+
- retransmit: 3-5+ minutes
- close wait: 30s (2MSL)
- 0-window persist: forever @ 60s
- IP fragment assembly: 30s
- TCP uses a 200ms and 500ms timer to manage the various timeouts.
  - Every 500 ms, check for packet timeouts, bump tick count
  - RTT estimator uses tick count from 500 ms timer
  - Timestamp is current tick count
  - Every 200 ms, see if any delayed ACKs need to be transmitted
  - Faster timer (100 ms) can improve TCP performance <u>when there are timeouts</u>
  - Newer OS’s have replaced 500 ms timer with 100 ms timer

Timer granularity



•Two competing TCP Reno flows with timeouts (ns simulation).

• higher resolution timer allows timeout to be detected “sooner”

•Check every 100 ms rather than every 500 ms

•Throughput: 816 kbs with 100 ms timer, 486 kbs with 500 ms timer

TCP socket options

- SO_RCVBUF SO_SNDBUF socket buffers (performance enhancers)
- SO_LINGER change close behavior
- SO_REUSEADDR avoid "port in use", TIME_WAIT
- TCP_NODELAY disable Nagle
- SO_KEEPALIVE 2hr idle check
- SO_DEBUG enable kernel trace

TCP protocol 1984

- TCP as defined by RFC 793
  - Sliding window flow control
    - Keeps sender from over-running receiver
    - Limits max sending rate
  - Simple RTT estimation used for timeout
  - ACK for in-order packets received plus cumulative ACK
    - Doesn't require receiver buffer out of order packets
    - Sender may have to go-back-N if a packet is lost
  - Timer for detecting lost packet and doing retransmission
- Sender blasts initial window of packets
- ACK clocking adapts flow to available and changing bandwidth

Packet loss

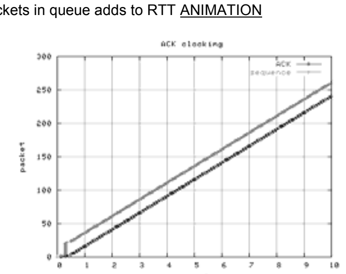
- Packet loss from bit errors on media
  - Random
  - Each media has a defined bit-error rate
  - Often link layer recovers (e.g., CSMA/CD collision detect, retransmit)
  - Media loss can be bursty, but bit-loss usually contained within a packet
  - No need to adjust sending rate
- Packet loss due to congestion
  - Classic queuing theory
    - Arrival rates faster than service rates
    - Queues grow
      - RTT increases (response degrades)
      - Throughput (goodput) decreases
  - Finite queues at routers overflow, packets are dropped
    - Queues typically FIFO (droptail)
  - Sender should reduce sending rate until (?) congestion subsides

congestion

- informally: "too many sources sending too much data too fast for network to handle"
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- Congestion control: providing feedback to sender to control sending rate
  - different from flow control! (too fast for receiver too handle)

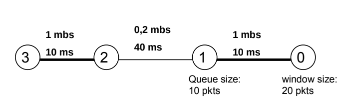
ACK clocking example (no losses)

- '83 TCP (RFC 793) did initial blast of window of packets, and timeout and go-back-N for packet loss
- Initial window blast builds up queue, then runs at link speed (200kbs)
- 20 packets in queue adds to RTT ANIMATION



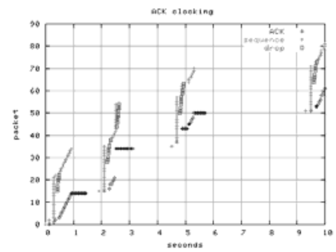
TCP response to congestion in 1984

- self-clocking example but now with a queue size of 10



- nam cc84.nam
- Sender blasts 20 packets, overflows queue, dropped packets, timeout and retransmission, eventually resulting in another blast!
- Data rate (goodput) drops to 48 Kbs (out of 200 Kbs)
- Some retransmitted packets were already at receiver, so sender response adds to congestion (unneeded packets)

Queue size of 10 window of 20 (old TCP)



- 20 packet burst overflows router queue, drops (dup ACKs)
- Some packets made it, and their ACKs release additional packets
- Timeout, retransmit at 1.9, ACK'd and go-back-N blast
- More drops, some packets are making it to receiver, but sent again
- Goodput 48 kbs, but lots of wasted bandwidth

TCP congestion avoidance (1984)

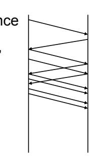
- RFC 896 (1984) noted performance problems with growing Internet
- 1) Excess of small packets (inefficient)
  - Silly window syndrome (Nagle fix)
  - Too many ACKs (delayed ACK fix)
- 2) congestion collapse
  - Interaction of reliable TCP on top of unreliable IP
  - Problems at routers connecting links of widely different bandwidths
  - Queues grow and overflow
  - Senders are retransmitting but not adjusting sending rate, so problem worsens
  - Little new data getting through … congestion collapse
- Congestion fix ('84):
  - Routers send ICMP source quench when queues start to build
    - This is congestion avoidance
  - When TCP sender receives a source quench, set "effective window" to zero for 10 ACKs or so … briefly backoff?
  - Source quench still allows ACKs and retransmissions

TCP congestion 1988

- The 1984 "recommendations" helped some …
- Problems
  - Traffic bursty – sudden build up of queues and RTT
  - Not all routers would send ICMP source quench
  - Not all senders would respond to source quench with rate reduction
  - At time of congestion when things are real "busy", the router is supposed to figure out who the big senders are and send 'em ICMP messages
    - Takes time away from forwarding operation (draining queue)
    - Actually injects MORE packets into the network
- October '86 (Van Jacobson)
  - Data rate between Internet sites LBL and UC Berkeley (400 yards) dropped by a factor of 1000! Congestion collapse was back.
  - Recommendations (and implemented in 4.3 BSD)
    - Better RTT variance estimation and thence better timeout value
    - Exponential backoff for retransmit timer
    - Slow-start
    - Congestion control (cwnd and ssthresh)
      - Based on packet loss (not congestion avoidance)

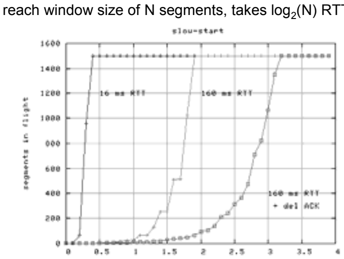
TCP slow-start

- Rather than blasting an initial window of packets, increase sending rate exponentially up to window size
  - Network friendly
  - A form of congestion control
  - **Goal**: get ACK clock running
- Sender sends one packet, when ACK arrives send next two packets
- As each ACK arrives, send two more packets
- sending rate doubles each RTT
- For long running flows, this will have little effect on performance
- For short flows (a few data packets), performance (response, throughput) may be noticably reduced (almost stop and wait)
- Use slow-start when ACK clocking lost
  - Startup
  - Timeout (or 3 dup ACKs – later)
  - Idle (no packets in flight for RTO seconds)



Slow-start examples

- Flows with longer RTT will take longer to ramp up
- Delayed ACK algorithm makes slow-start even slower
  - ACK every other packet (Linux turns off del ACK during slow-start)
  - Studies use "byte counting" rather than ACK counting
- To reach window size of N segments, takes $\log_2(N)$ RTT's



N = 1500

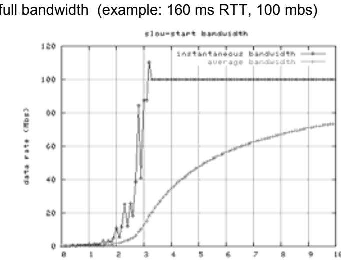
$\log_2$ (N) = 11

11 RTTs

If RTT=160ms then 1.76 seconds

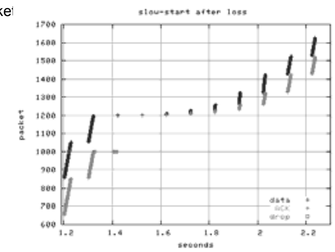
Slow-start effect on throughput

- Not a problem for LANs (tiny RTT)
  - 4.3 BSD Tahoe only did slow-start if destination was not on local subnet
- For long RTT (and delayed ACK) and high speed link, can take a while to reach full bandwidth (example: 160 ms RTT, 100 mbs)



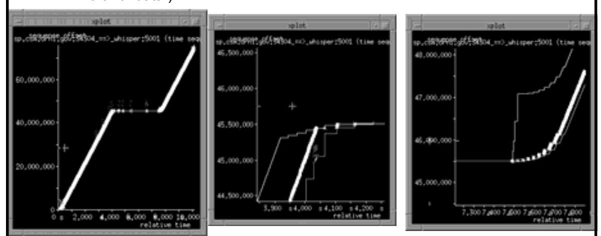
Slow-start after packet loss

- Packet lost
- TCP retransmits lost packet
- Enters slow-start



Slow-start after idle

- Flow can go idle due to application pauses in writing/reading data
- If no unACK'd packets (nothing in flight) for more than RTO seconds, TCP has lost ACK clocking, enter slow-start (don't blast a windows worth of data!)



Application pauses …

Resumes, adv. window opens, slow-start

Next time …

- TCP congestion control
- TCP Tahoe