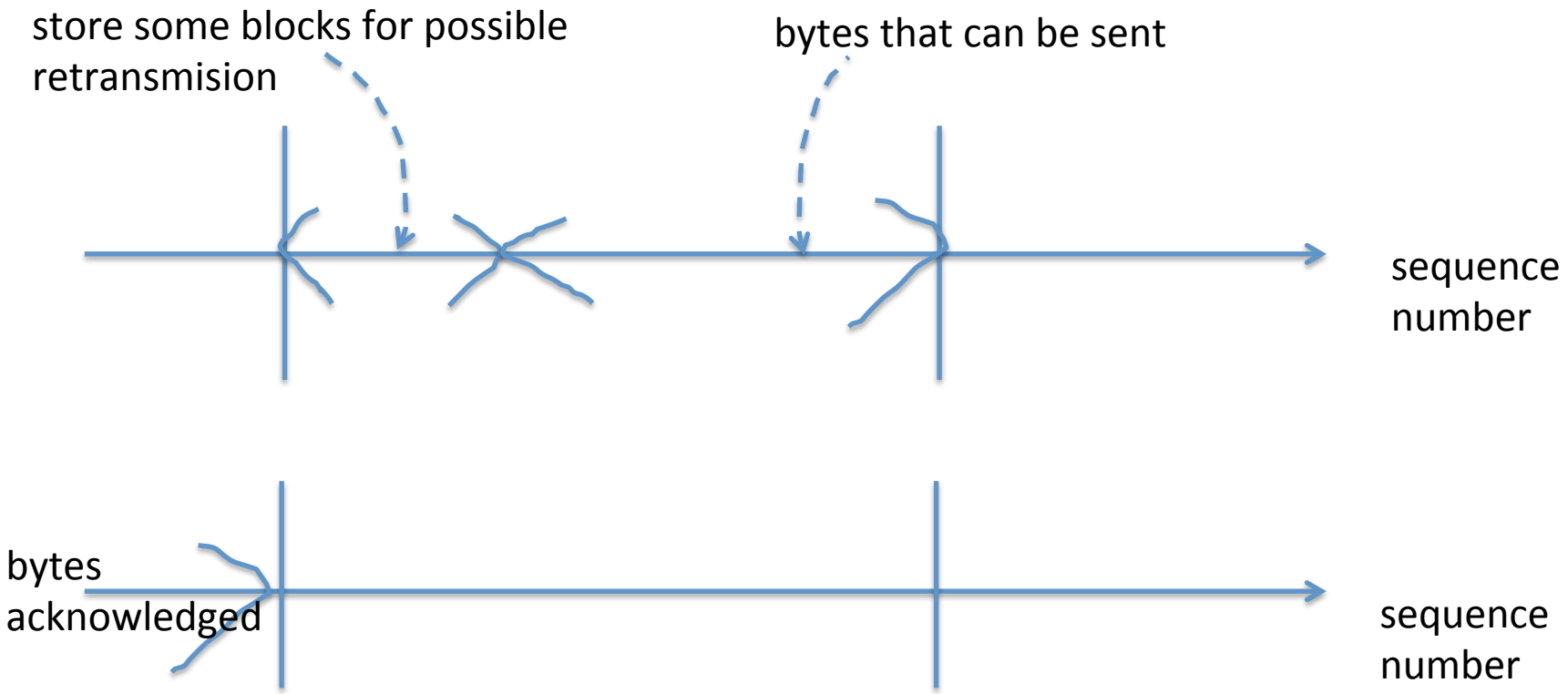
ESX (freebsd 8) TCP intro

Ricardo Koller

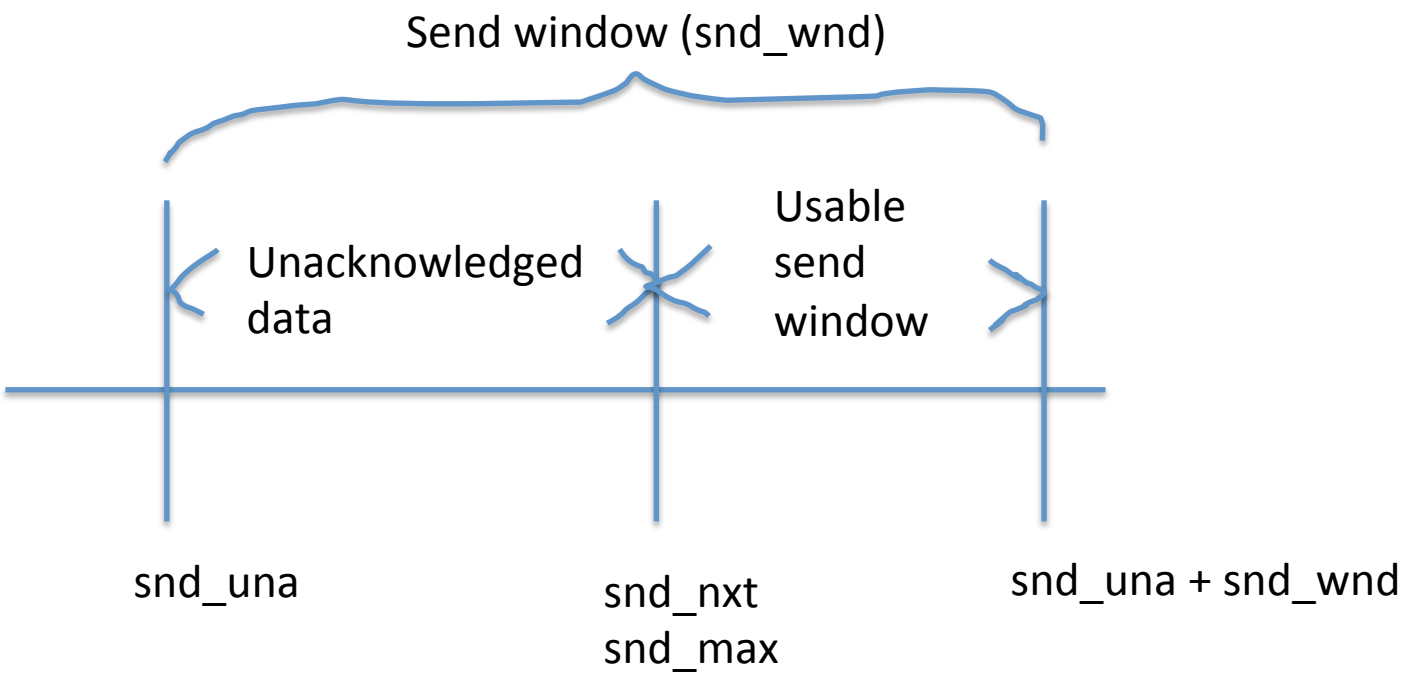
TCP sequencing

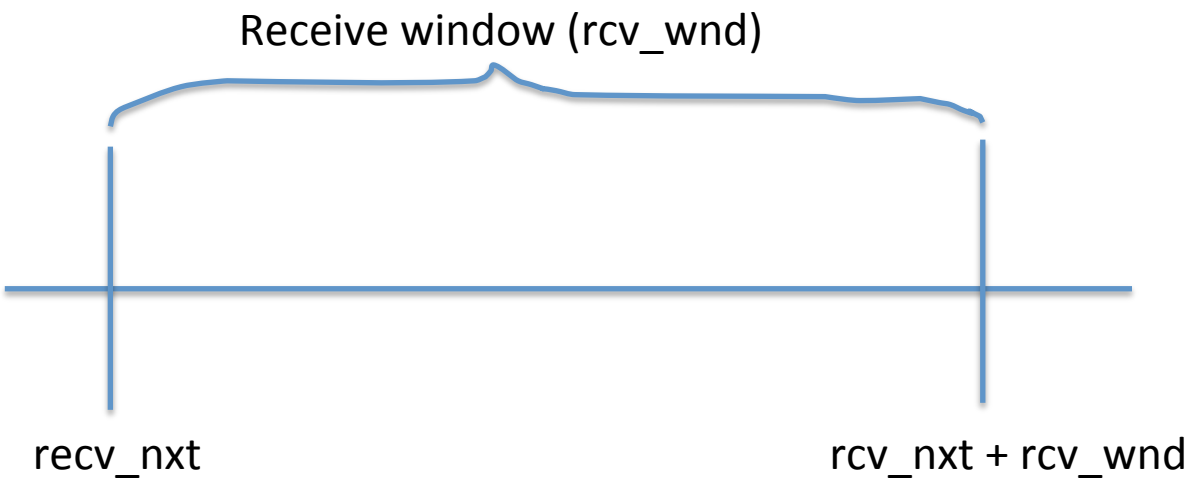


TCP control block

```
struct tcpcb {
    snd_una
    snd_nxt
    snd_wnd
    snd_max
    rcv_nxt
    rcv_wnd
    …
}
```

one control block per direction


Send window (snd_wnd)
Unacknowledged data
Usable send window
snd_una
snd_nxt
snd_max
snd_una + snd_wnd


Receive window (rcv_wnd)
recv_nxt
rcv_nxt + rcv_wnd

TCP control block

```
struct tcpcb {
    snd_una
    snd_nxt
    snd_wnd
    snd_max
    rcv_nxt
    rcv_wnd
    …
}
```

one control block per direction

Code structure (freebsd and esx)

- Event based:
  1. request from the user:
     tcp_usrreq()
  2. receipt of a packet from the connection:
     tcp_input(tcpcb)
  3. timer expiration:
     tcp_timers()

- Each function process the event
  - if the processing of the event requires an output:
    tcp_output(tcpcb)

tcp_output(tcpcb)

- When to output data:
  1. user places data in the send buffer
  2. receipt of a window update from peer
  3. retransmission timer timeout
- When to output without data
  1. receipt of data that must be ACKed
  2. change in connection state
  3. change in the receive window

multiple stack implementations

tcp_output

- Net_SendTo → sf->sendto → vmk_sendto()
  - sosend() // wait for enough space in send buffer
    - tcp_usrreq()
      - tcp_output() // possibly send data immediately
        // i.e. if allowed by window
        1. packet.seq = snd_nxt
        2. packet.ack = rcv_nxt
        3. packet.advertised_win = space in the receive buffer
        4. ip_output(packet)
        5. snd_nxt += whatever we sent

Code structure

- tcp_timers.c
  - tcp_timers()
- tcp_usrreq.c
  - tcp_usrreq.c
- tcp_input.c
  - tcp_input()
- tcp_output.c
  - tcp_output()

modules/vmkernel/tcpip4/freebsd/netinet/

Retransmission

- Time based retransmission
  - Retransmit if there is no ACK
- Fast retransmission
  - Retransmit if 3 duplicated ACKs

Time based retransmission

- One timer per connection: TT_REXMT
- Started whenever there is data sent, unless it is already started
- Stopped when all outstanding data is ACKed

```
tcp_timer_rexmt(tp) // expire
{
    tp->snd_nxt = tp->snd_una
    // tp->snd_max is kept constant (max byte sent)
    tcp_output(tp)
}
```

tcp_output:

```
/*
 * Set retransmit timer if not currently set,
 * and not doing a pure ack or a keep-alive probe.
 * Initial value for retransmit timer is smoothed
 * round-trip time + 2 * round-trip time variance.
 */
if (!tcp_timer_active(tp, TT_REXMT) &&
   ((sack_rxmit && tp->snd_nxt != tp->snd_max) ||
    (tp->snd_nxt != tp->snd_una))) {
            tcp_timer_activate(tp, TT_REXMT, tp->t_rxtcur);
}
```

tcp_input:

if max byte sent was just acked

```
/*
 * If all outstanding data is acked, stop retransmit
 * timer and remember to restart (more output or persist).
 */
if (th->th_ack == tp->snd_max) {
            tcp_timer_activate(tp, TT_REXMT, 0);
            needoutput = 1;
}
```

Estimation of RTT

- Smoothed moving average

  srtt = (α * srtt) + ((1 – α) * rtt)

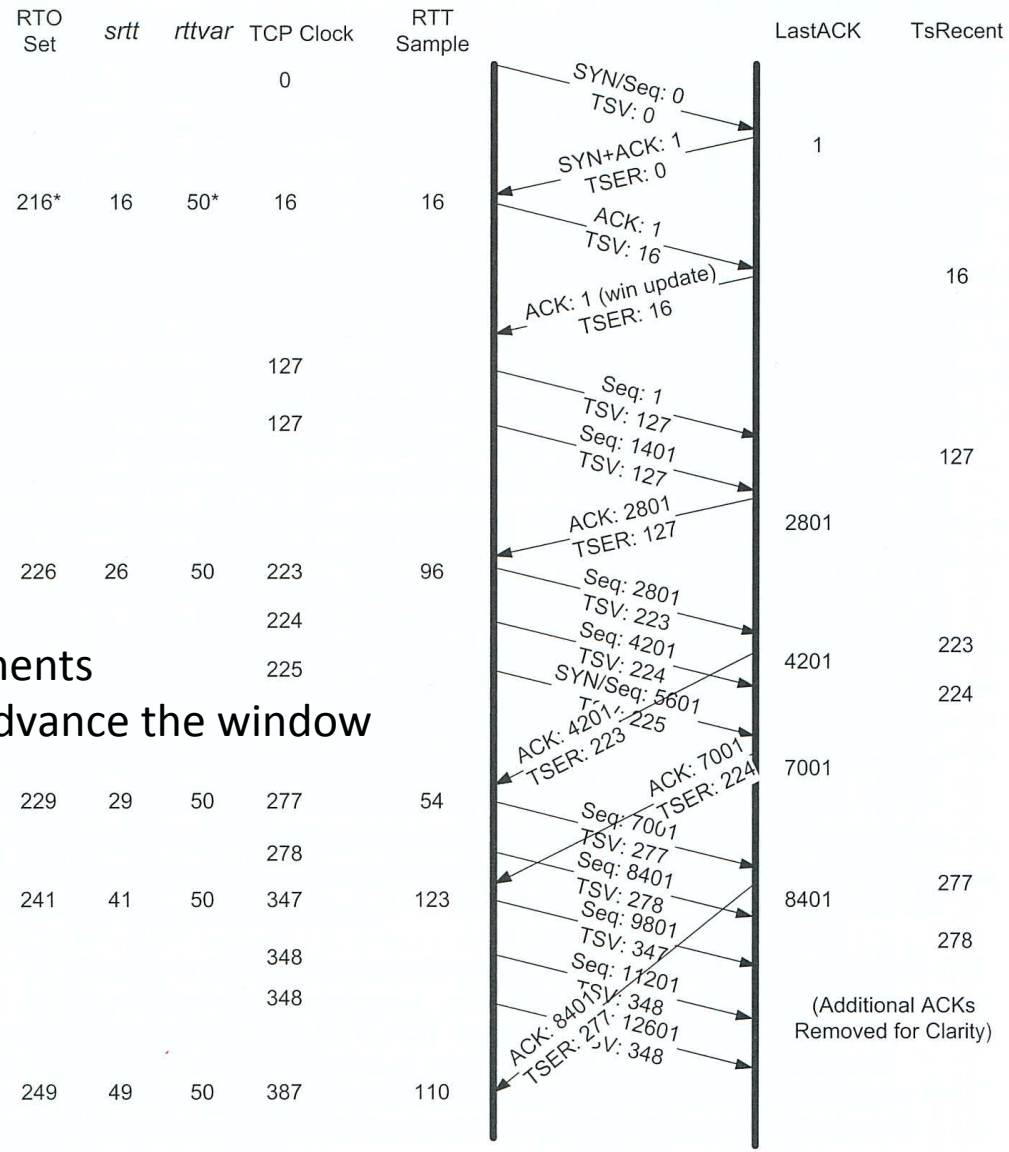
  α = 0.9
- esx:
  - srtt + 2 * smoothed_variance
  - maximums: sysctl
  - minimums: sysctl
- start:

  tp->t_srtt = TCPTV_SRTTBASE;
- moving average (tcp_xmit_timer(tp, rtt)):

  tp->t_rxtcur = srtt + 2 * smoothed_variance;

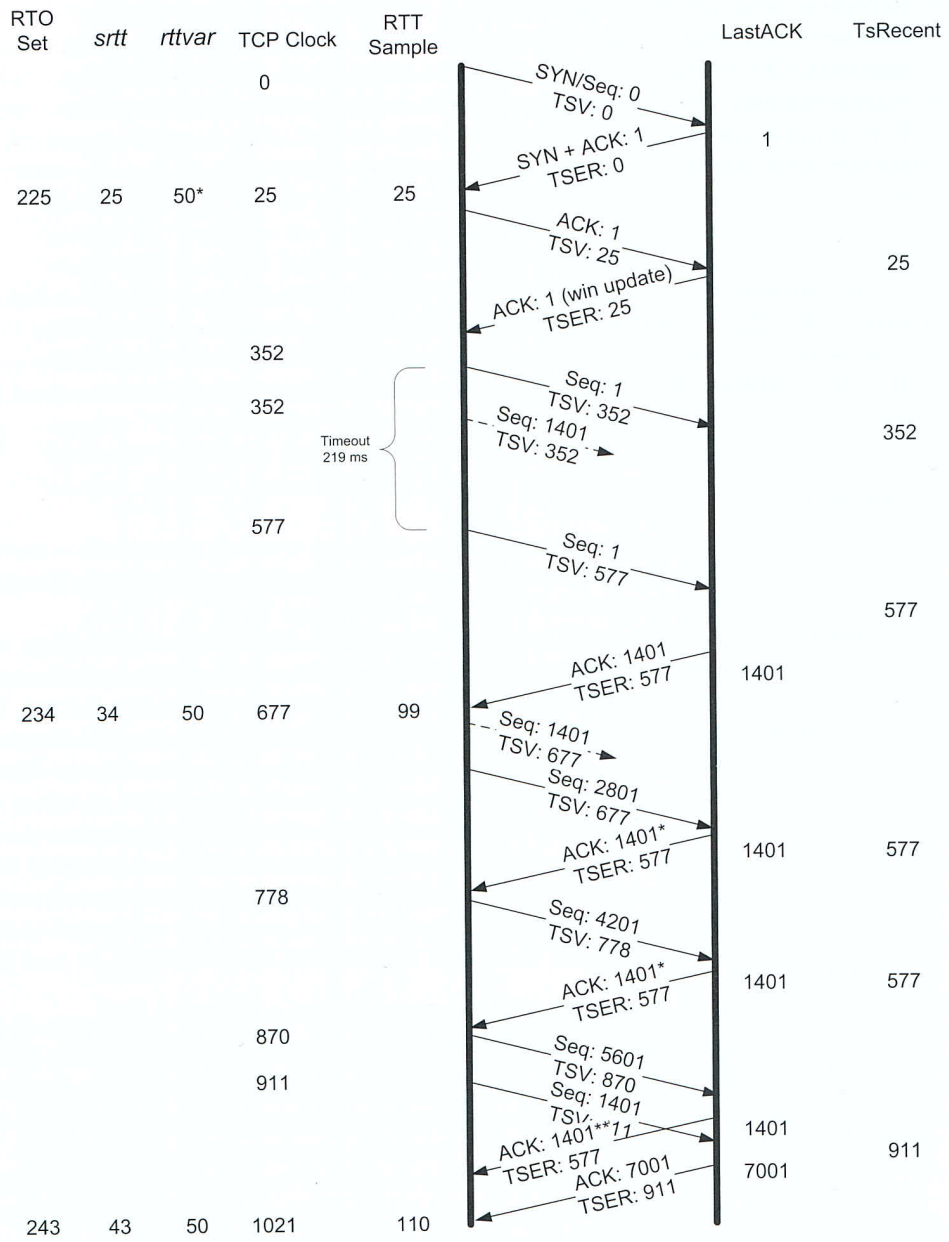
  tcp_timer_activate(tp, TT_REXMT, tp->t_rxtcur);

RTT measurements



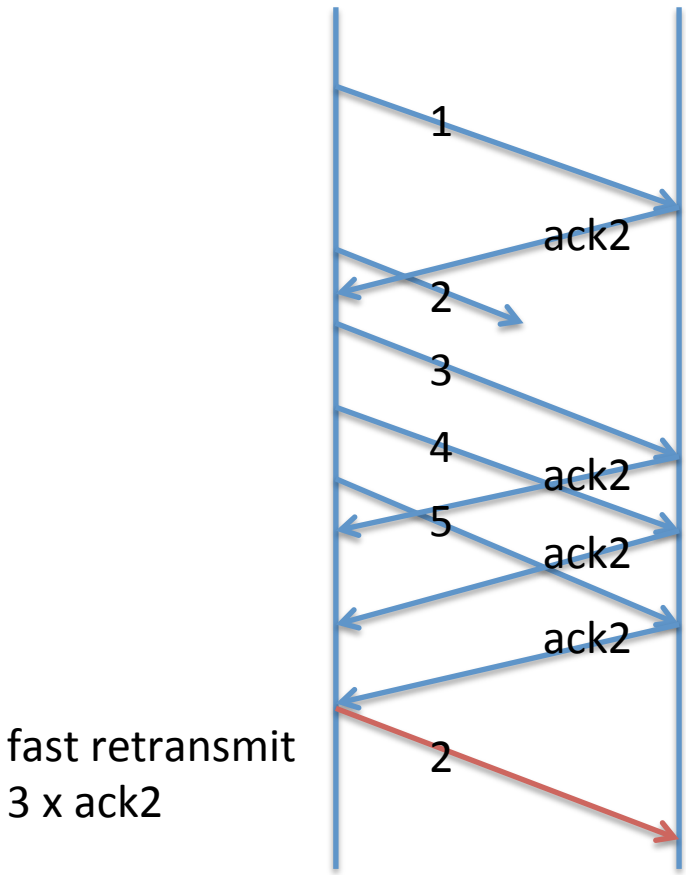
only measurements
on ACKs that advance the window

Timer based retransmission



Fast retransmission

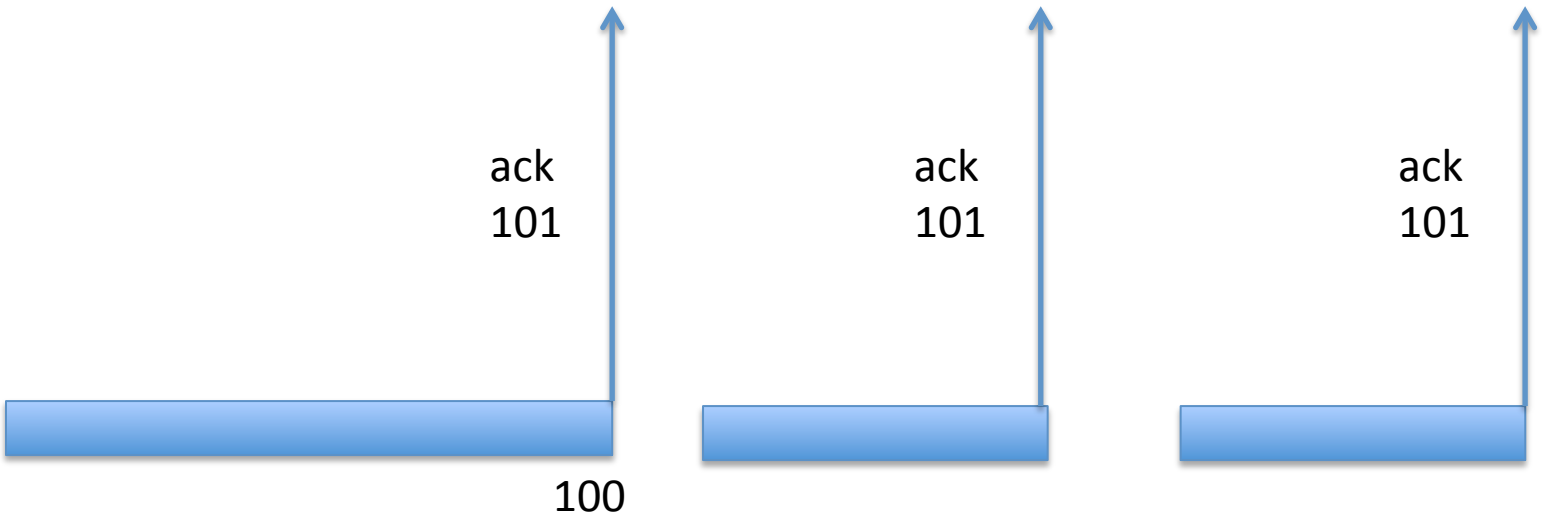
- Retransmit if 3 duplicated ACKs are received



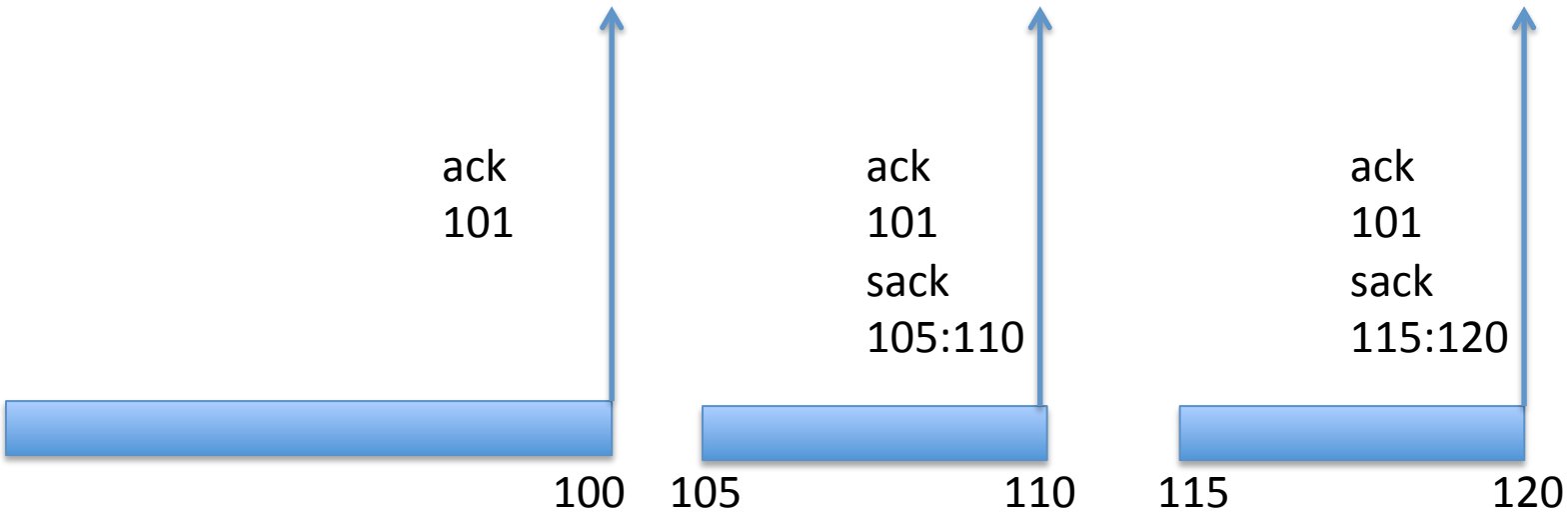
tcp_input.c:

```
const int tcprexmtthresh = 3;
…
if (++tp->t_dupacks >= tcprexmtthresh) {
        // start fast retransmit
        (void) tcp_output(tp);
}
```

Retransmission with SACK



Retransmission with SACK



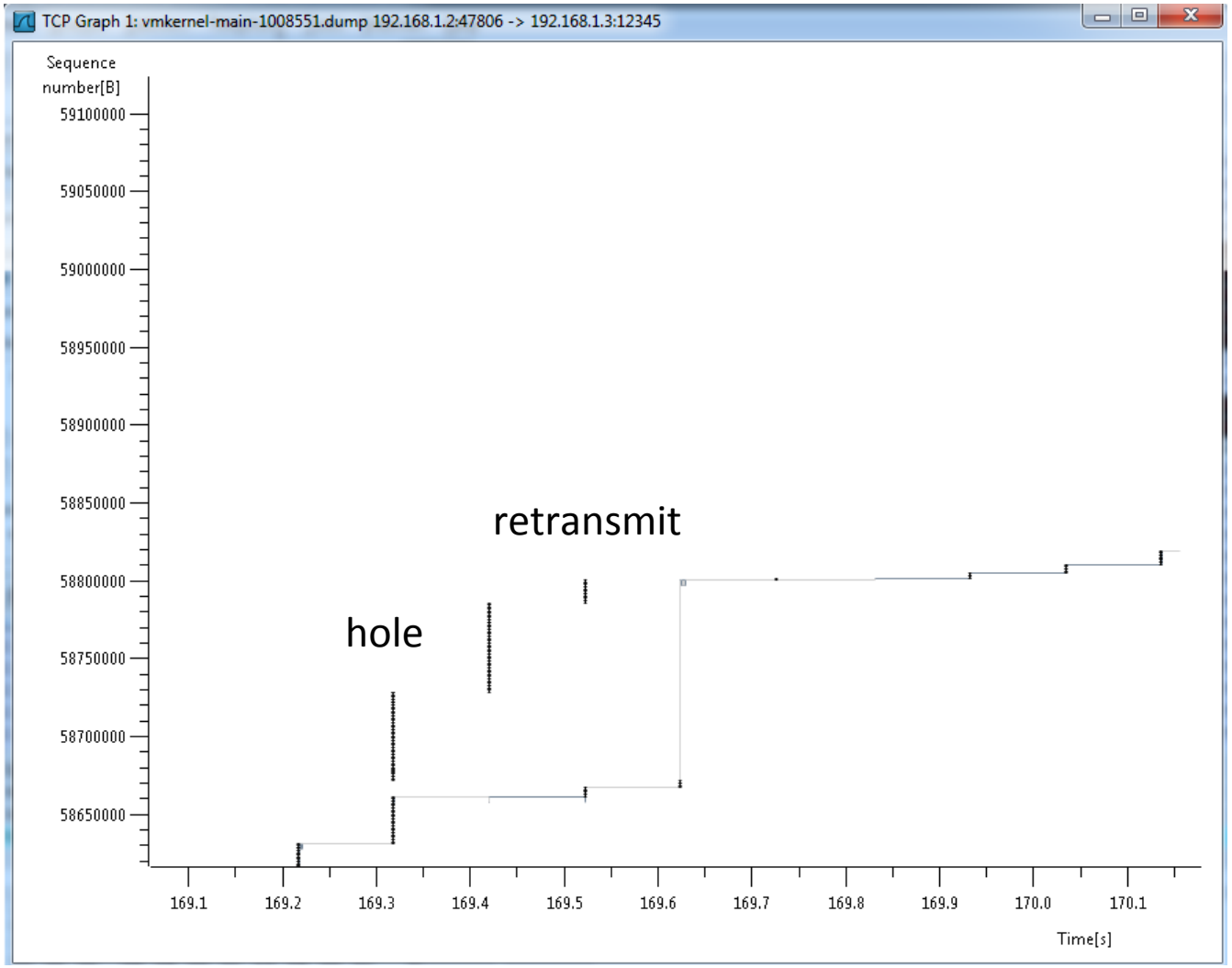
Retransmission with SACK

On the sender side:

```
tcp_input.c: // receive list of holes
    tcp_update_sack_list(tp, save_start, save_start + tlen);

tcp_output: // get next hole to retransmit
    // p is next_hole
    p = tcp_sack_output(tp, &sack_bytes_rxmt);
        p->rexmit
        p->end
```

ESX example


TCP Graph 1: vmkernel-main-1008551.dump 192.168.1.2:47806 -> 192.168.1.3:12345
Sequence number[B]
59100000
59050000
59000000
58950000
58900000
58850000
58800000
58750000
58700000
58650000
retransmit
hole
169.1
169.2
169.3
169.4
169.5
169.6
169.7
169.8
169.9
170.0
170.1
Time[s]

ESX example

vmkernel-main-1008551.dump [Wireshark 1.8.5 (SVN Rev 47350 from /trunk-1.8)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
66348	169.420211	192.168.1.2	192.168.1.3	TCP	1514	ap > italk [ACK] Seq=58781633 Ack=1 Win=13107200 Len=1448
66349	169.420211	192.168.1.2	192.168.1.3	TCP	1514	ap > italk [ACK] Seq=58783081 Ack=1 Win=13107200 Len=1448
66350	169.420353	192.168.1.2	192.168.1.3	TCP	1514	ap > italk [ACK] Seq=58784529 Ack=1 Win=13107200 Len=1448
66351	169.420588	192.168.1.3	192.168.1.2	TCP	78	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=58661449 Wi
66352	169.420597	192.168.1.3	192.168.1.2	TCP	78	[TCP Dup ACK 66351#1] italk > ap [ACK] Seq=1 Ack=58661449
66353	169.420599	192.168.1.3	192.168.1.2	TCP	78	[TCP Dup ACK 66351#2] italk > ap [ACK] Seq=1 Ack=58661449
66354	169.420600	192.168.1.3	192.168.1.2	TCP	78	[TCP Dup ACK 66351#3] italk > ap [ACK] Seq=1 Ack=58661449
66355	169.420602	192.168.1.3	192.168.1.2	TCP	78	[TCP Dup ACK 66351#4] italk > ap [ACK] Seq=1 Ack=58661449
66356	169.420603	192.168.1.3	192.168.1.2	TCP	78	[TCP Dup ACK 66351#5] italk > ap [ACK] Seq=1 Ack=58661449
66357	169.420604	192.168.1.3	192.168.1.2	TCP	78	[TCP Dup ACK 66351#6] italk > ap [ACK] Seq=1 Ack=58661449
66358	169.521850	192.168.1.2	192.168.1.3	TCP	1514	ap > italk [ACK] Seq=58785977 Ack=1 Win=13107200 Len=1448
66359	169.521855	192.168.1.2	192.168.1.3	TCP	1514	ap > italk [ACK] Seq=58787425 Ack=1 Win=13107200 Len=1448

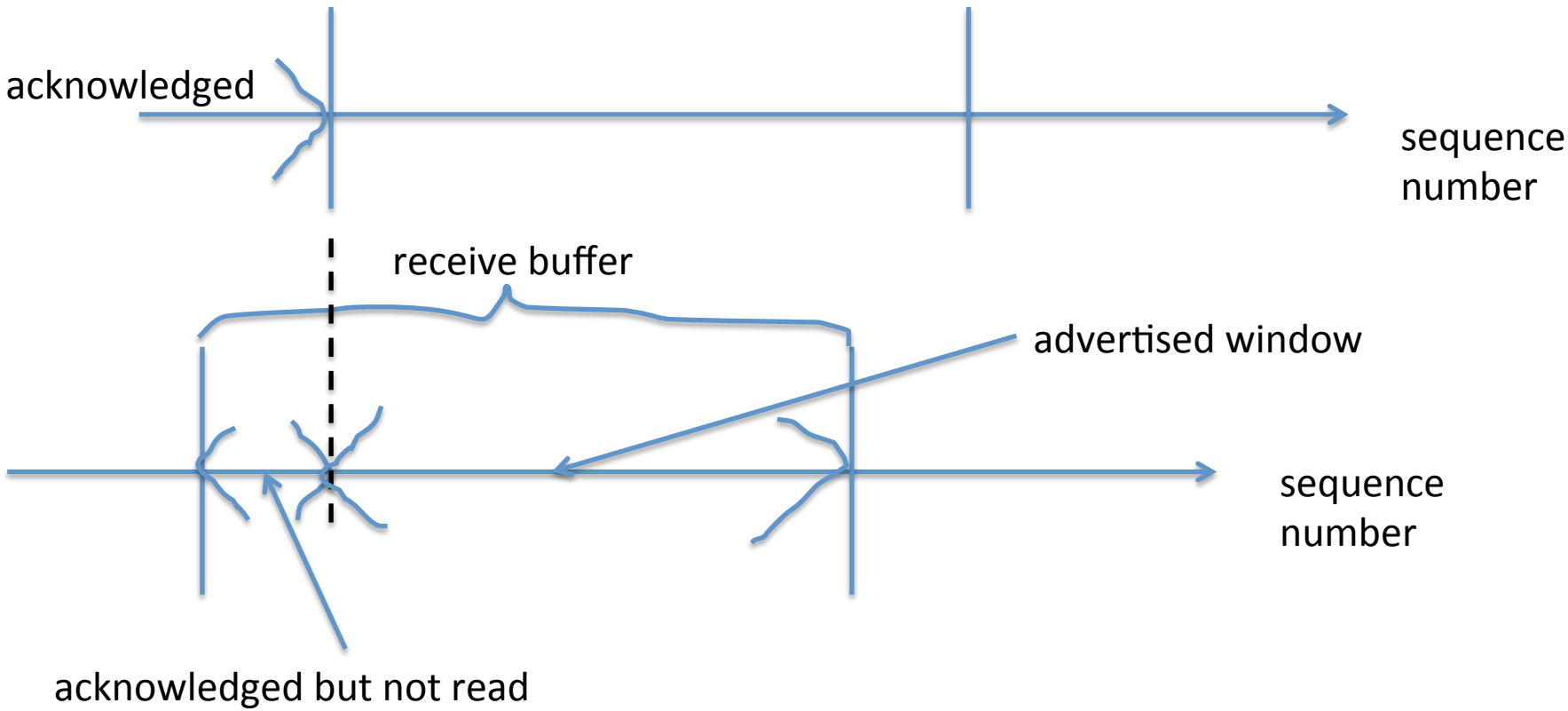
```
⊞ Timestamps: TSval 137385, TSecr 137595
⊞ No-Operation (NOP)
⊞ No-Operation (NOP)
⊟ SACK: 58671585-58749985
    left edge = 58671585 (relative)
    right edge = 58749985 (relative)
⊞ [SEQ/ACK analysis]
```

```
0000  80 c1 6e 65 79 7d 80 c1  6e 65 88 29 08 00 45 00   ..ney}.. ne.)..E.
0010  00 40 81 47 40 00 40 06  36 1b c0 a8 01 03 c0 a8   .@.G@.@. 6.......
0020  01 02 30 39 ba be c4 d5  d3 b9 ce 05 a3 d5 b0 10   ..09.... ........
0030  63 ff cc 9d 00 00 01 01  08 0a 00 02 18 a9 00 02   c....... ........
0040  19 7b 01 01 05 0a ce 05  cb 6d ce 06 fd ad         .{...... .m....
```

File: "\\vmware-host\Shared Folders\tcpdu... | Packets: 167707 Displayed: 16770... | Profile: Default

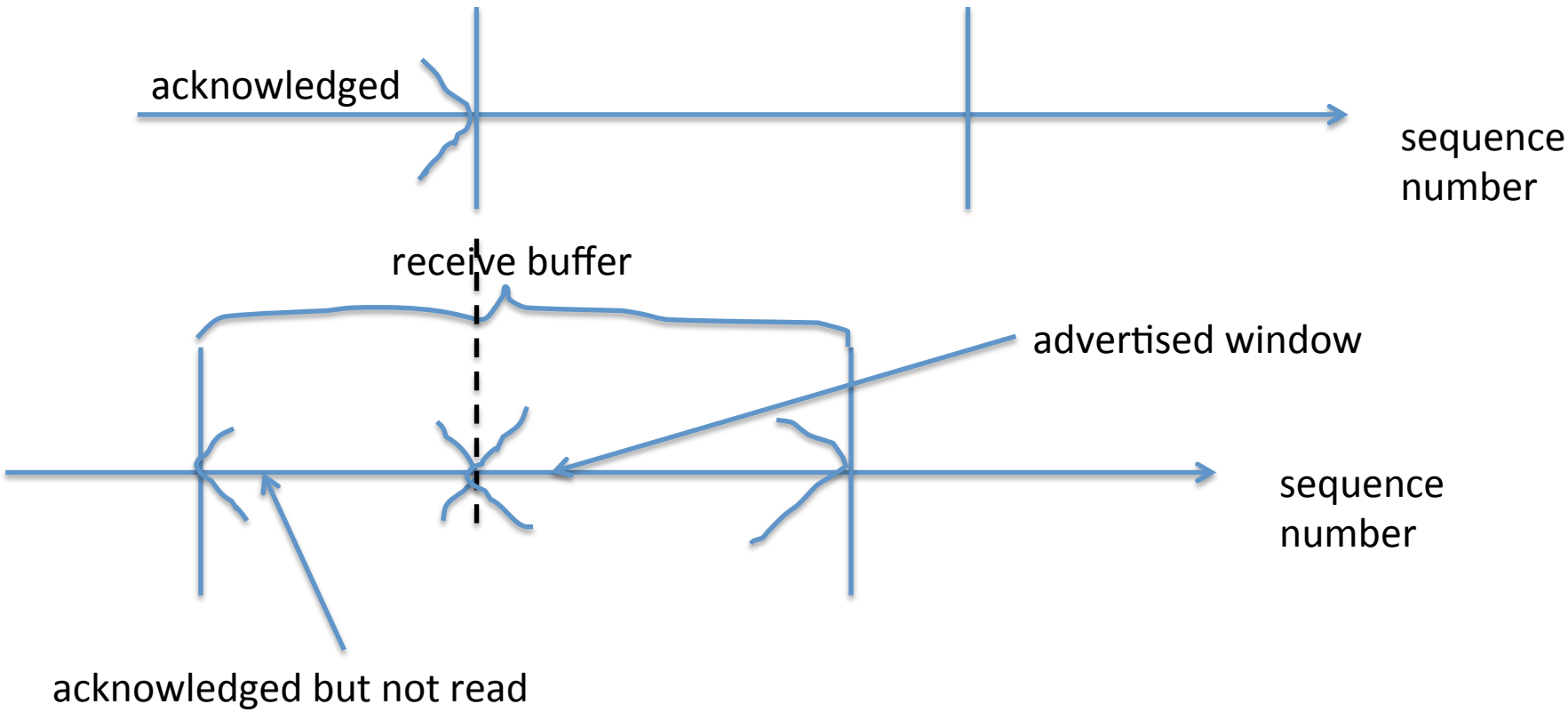
Flow control

- What if receiver reads too slowly



Flow control

- What if receiver reads too slowly



Zero windows and TCP persist timer

- If the receiver continues without reading, window will eventually be zero
  - After this, sender will eventually fill its buffer (window is not sliding)
  - sender will be waiting at:
    - sosend() // wait for enough space in send buffer

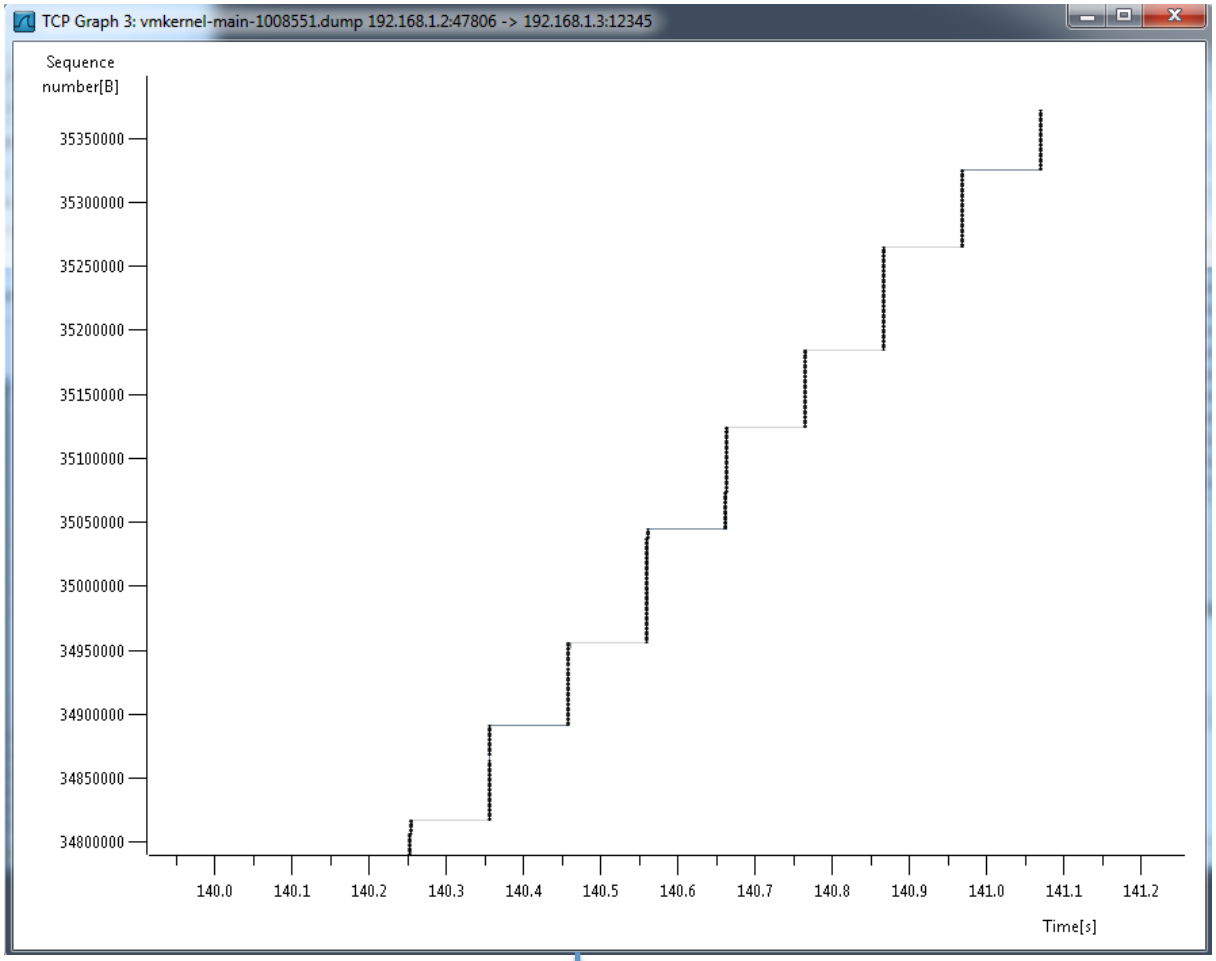
- window update from receiver (no data = no reliable)
  - persist timer (just in case window update is lost)
    - tcp_timer_activate(tp, TT_PERSIST, tt);

Delayed ACKs

- tcp_intput()
  - if received segment starts at rcv_nxt
    - delay tcp_timer_activate(tp, TT_DELACK , 200ms);
  - else
    - tcp_output() immediately // fast retransmit needs to know

- tcp_output()
  - send ACK
  - tcp_timer_activate(tp, TT_DELACK, 0); // deactivate

Delayed ACKs



100ms

200ms → half throughput?

tcp_output(tcpcb)

- When to output data:
  1. user places data in the send buffer
  2. receipt of a window update from peer
  3. retransmission timer timeout
- When to output without data
  1. receipt of data that must be ACKed
  2. change in connection state
  3. **<u>change in the receive window → receiver read something</u>**

Delayed ACKs

window update

same ack sequence

vmkernel-main-1008551.dump [Wireshark 1.8.5 (SVN Rev 47350 from /trunk-1.8)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Info
40537	140.357005	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13083136
40538	140.357247	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13086208
40539	140.357256	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13089280
40540	140.357487	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13092352
40541	140.357496	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13095424
40542	140.357497	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13098496
40543	140.357734	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13101568
40544	140.357739	192.168.1.3	192.168.1.2	TCP	[TCP Window Update] italk > ap [ACK] Seq=1 Ack=34891081 Win=13104640
40545	140.457197	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34891081 Ack=1 Win=13107200 Len=1448 TSval=13470
40546	140.457207	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34892529 Ack=1 Win=13107200 Len=1448 TSval=13470
40547	140.457209	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34893977 Ack=1 Win=13107200 Len=1448 TSval=13470
40548	140.457211	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34895425 Ack=1 Win=13107200 Len=1448 TSval=13470
40549	140.457213	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34896873 Ack=1 Win=13107200 Len=1448 TSval=13470
40550	140.457214	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34898321 Ack=1 Win=13107200 Len=1448 TSval=13470
40551	140.457216	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34899769 Ack=1 Win=13107200 Len=1448 TSval=13470
40552	140.457218	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34901217 Ack=1 Win=13107200 Len=1448 TSval=13470
40553	140.457220	192.168.1.2	192.168.1.3	TCP	ap > italk [ACK] Seq=34902665 Ack=1 Win=13107200 Len=1448 TSval=13470

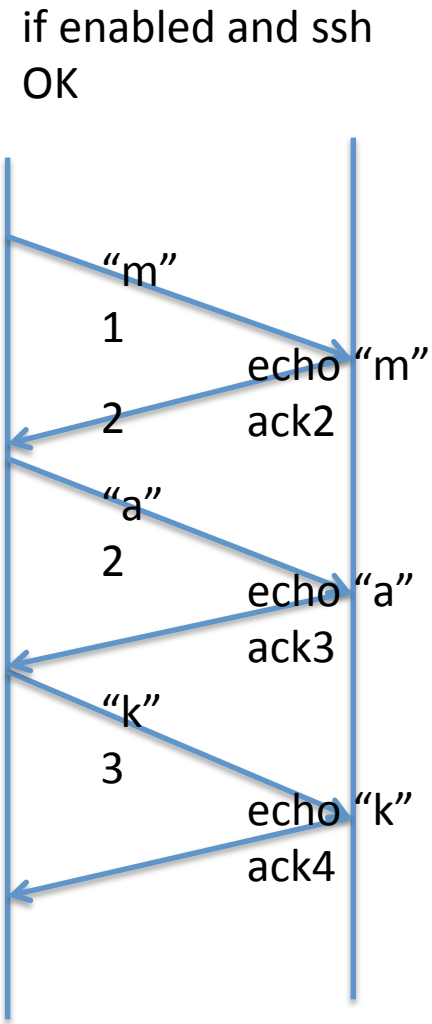
File: "\\vmware-host\Shared Folders\tcpdu... Packets: 167707 Displayed: 167707 Marke... Profile: Default

Nagle algorithm

- In ssh, every keystroke (1 byte) is a packet sent
- TCP should work relatively well for ssh and for bulk data transfer
- Nagle algorithm:
  - if there is a small segment to be sent, we can send it only if there is no data waiting to be acked

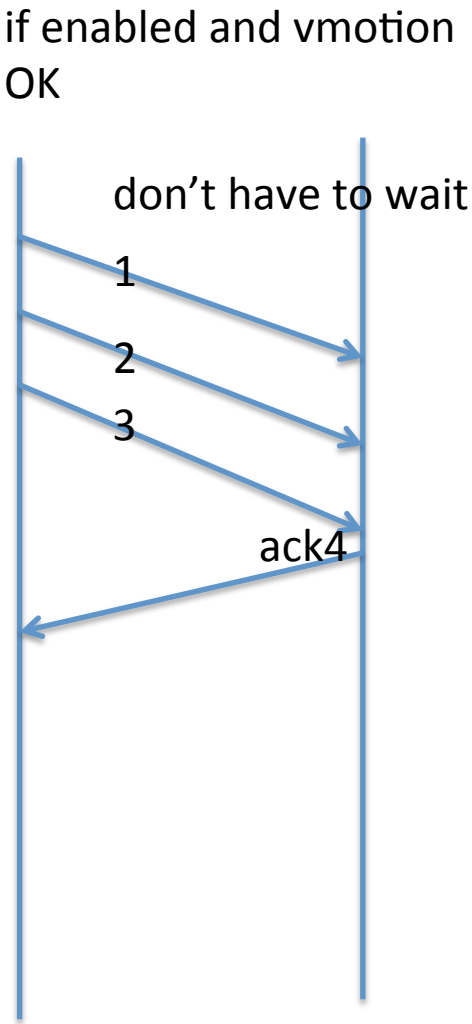
Nagle algorithm

- In ssh, every keystroke (1 byte) is a packet sent
- TCP should work relatively well for ssh and for bulk data transfer
- Nagle algorithm:
  - if there is a small segment to be sent, we can send it only if there is no data waiting to be acked



Nagle algorithm

- In ssh, every keystroke (1 byte) is a packet sent
- TCP should work relatively well for ssh and for bulk data transfer
- Nagle algorithm:
  - if there is a small segment to be sent, we can send it only if there is no data waiting to be acked



to disable it:

```
 if (isControlChannel ...)) {
     optval = 1;
     status = Net_SetSockOpt(socket, IPPROTO_TCP, TCP_NODELAY,
                    &optval, sizeof(optval), DEFAULT_STACK);
```

TCP segmentation offload (TSO)

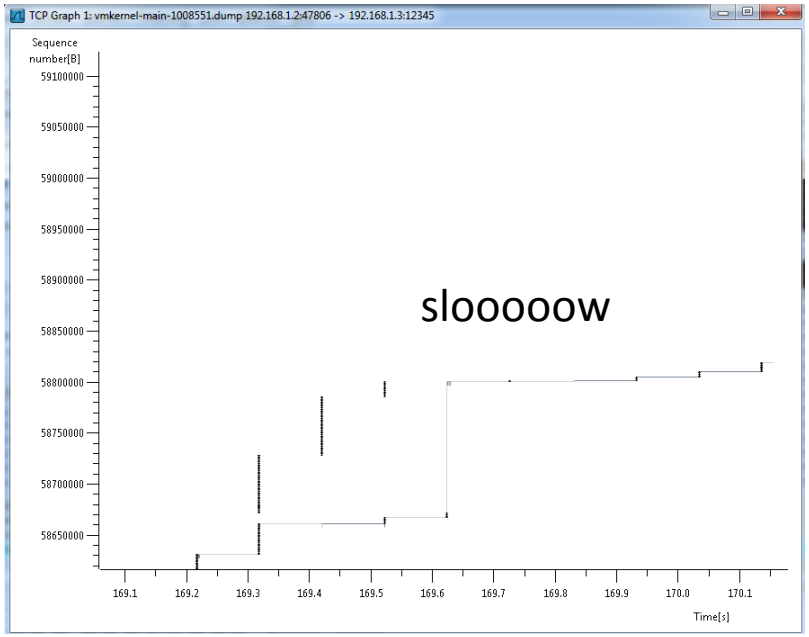
- tcp_output() of 64 Bytes is segmented into 44 segments of 1500 bytes each
- TCP pretends the nic has a huge MTU

```
tcp_output()
{
    if (CSUM_TSO) { // do we have TSO
        ip_output(huge packet) // the drive does the magic and split this
    } else {
        if (segment is huge) {
            tcp_output(short segment); // call recursively
            tcp_output(large segment); // call recursively
        }
    }
}
```

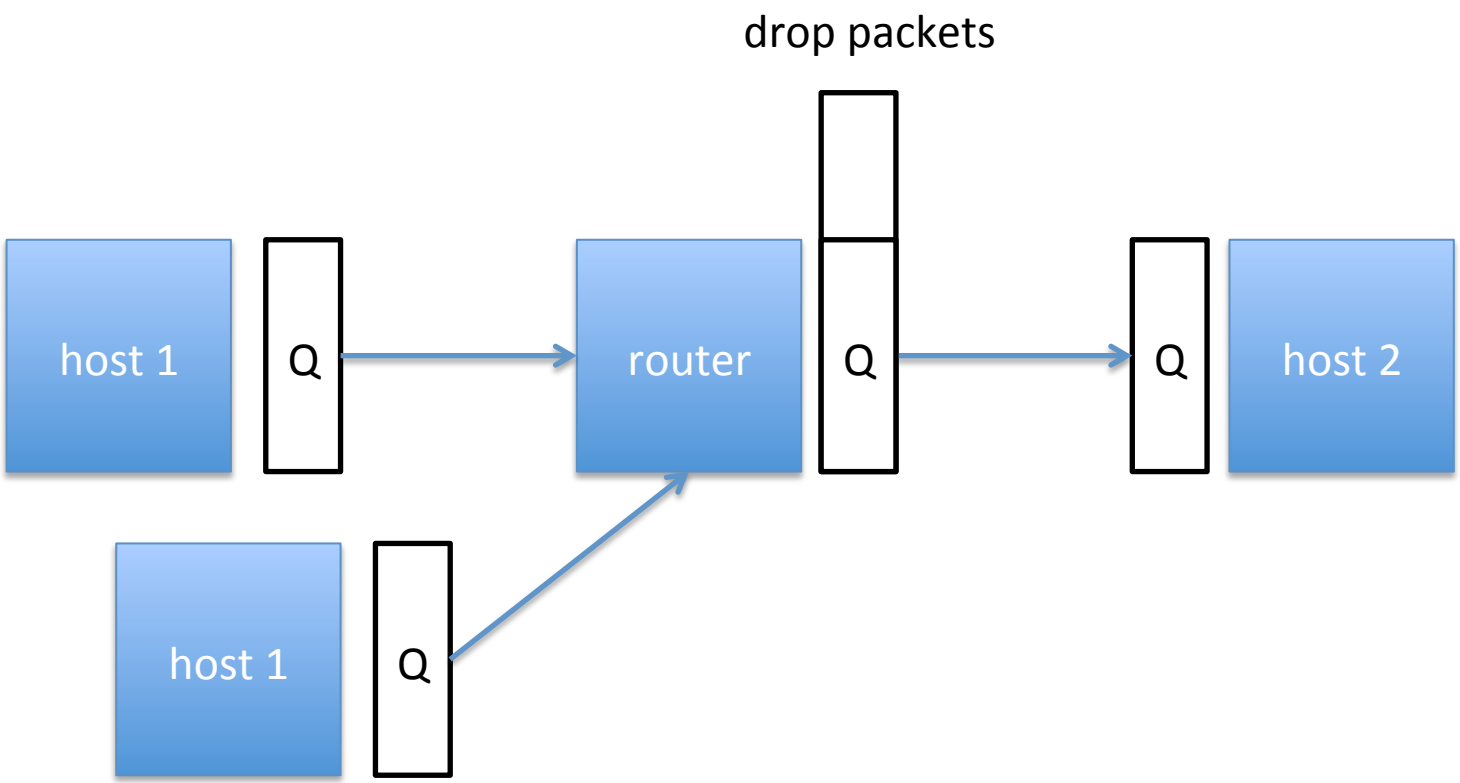
BTW: same thing happens for UDP

Slow start

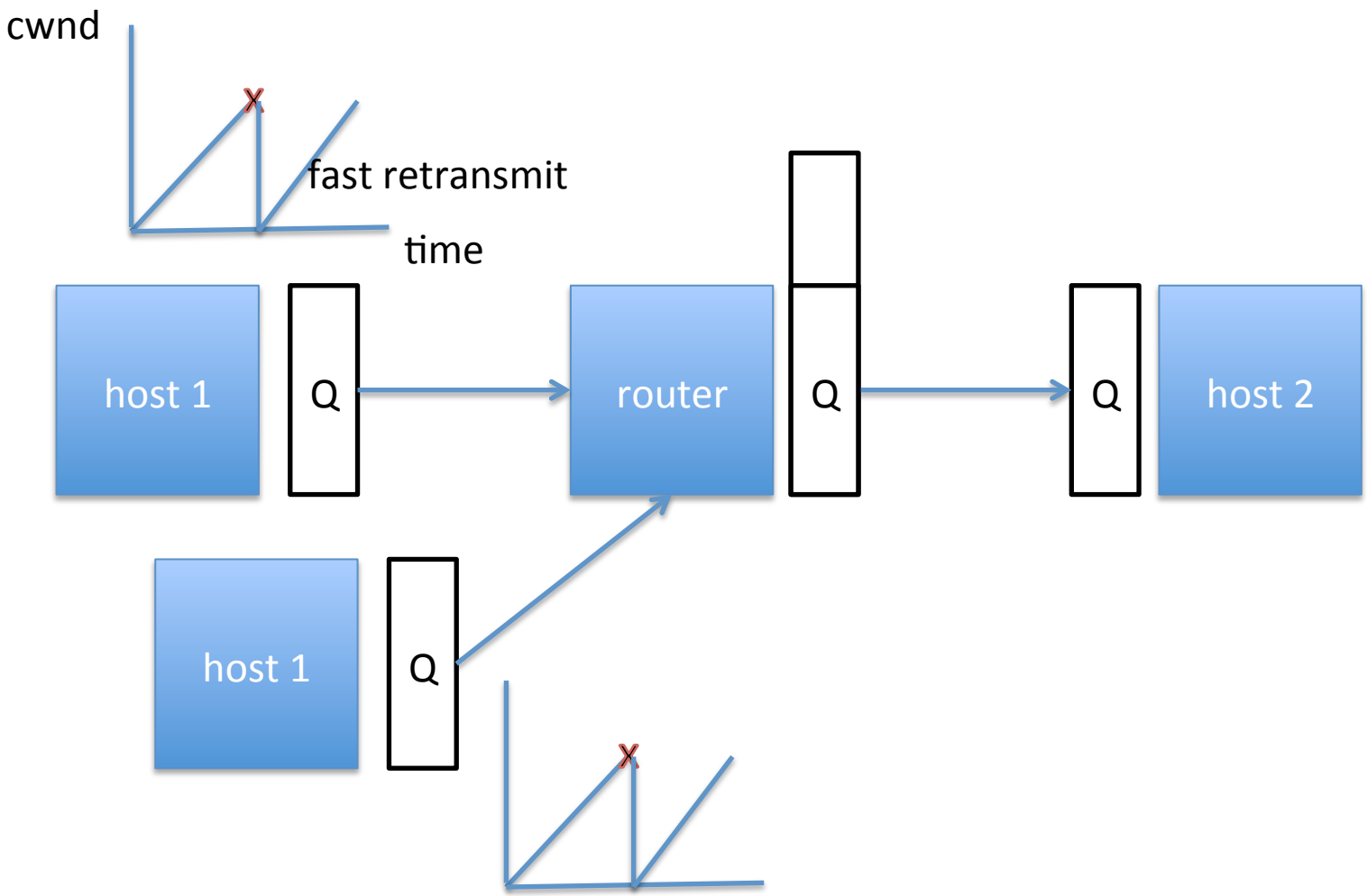
- tp->snd_cwnd // usable window has to be smaller than this
- set to previous cached value (or 0)
- set to 0 after a timer retransmit
- set to 0 (larger in others) after a fast retransmit
- tp->snd_cwnd += MSS; // every ACK, new reno



Congestion avoidance with slow start

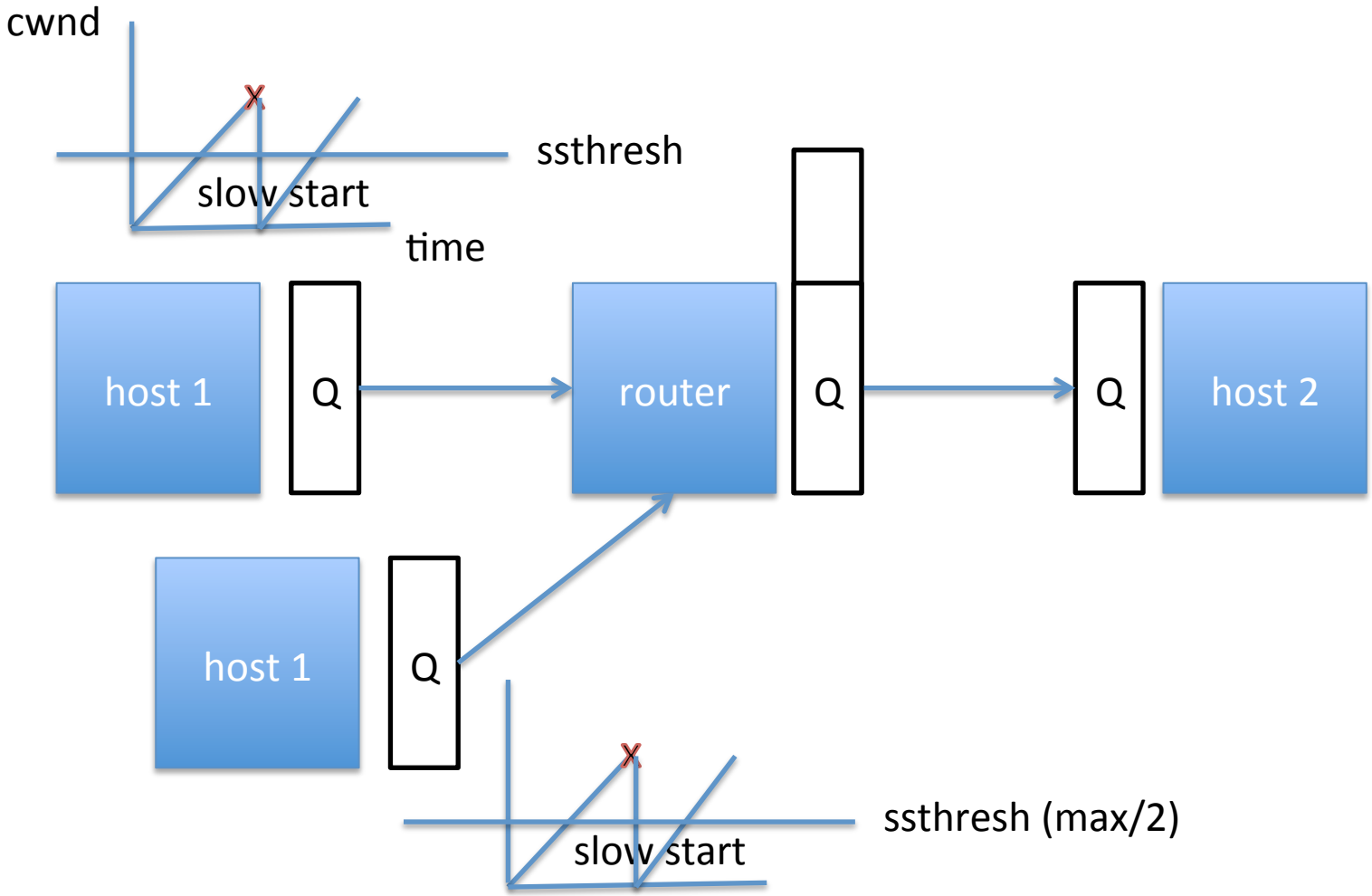


Congestion avoidance with slow start

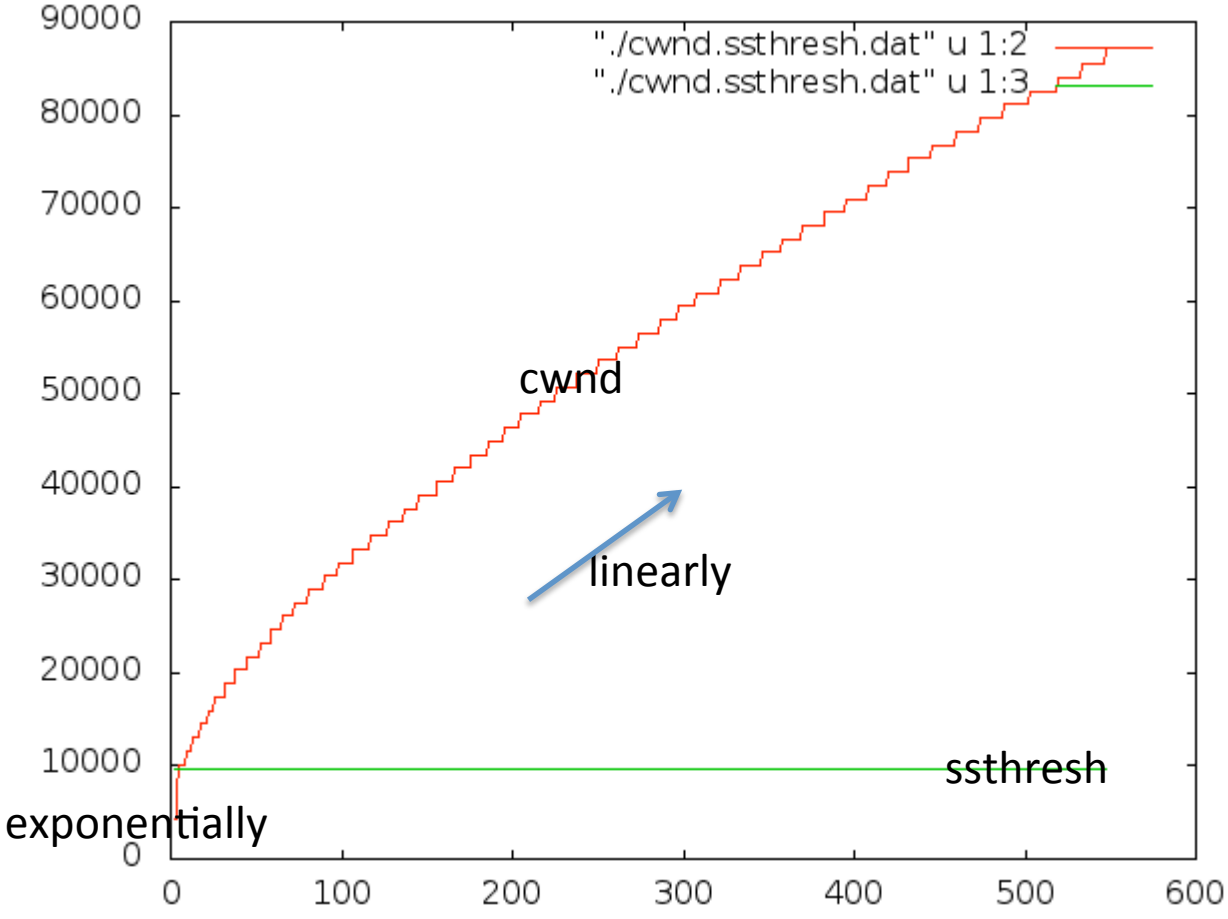


Congestion avoidance with slow start

- tp->t_ssthresh // estimate of the usable window



Congestion avoidance (ssthresh)



Next time

- more on congestion control
  - cubic, newreno, some newer RFCs about SACK
  - manipulate ssthresh, snd_cwnd, and RTT estimates
- memory management