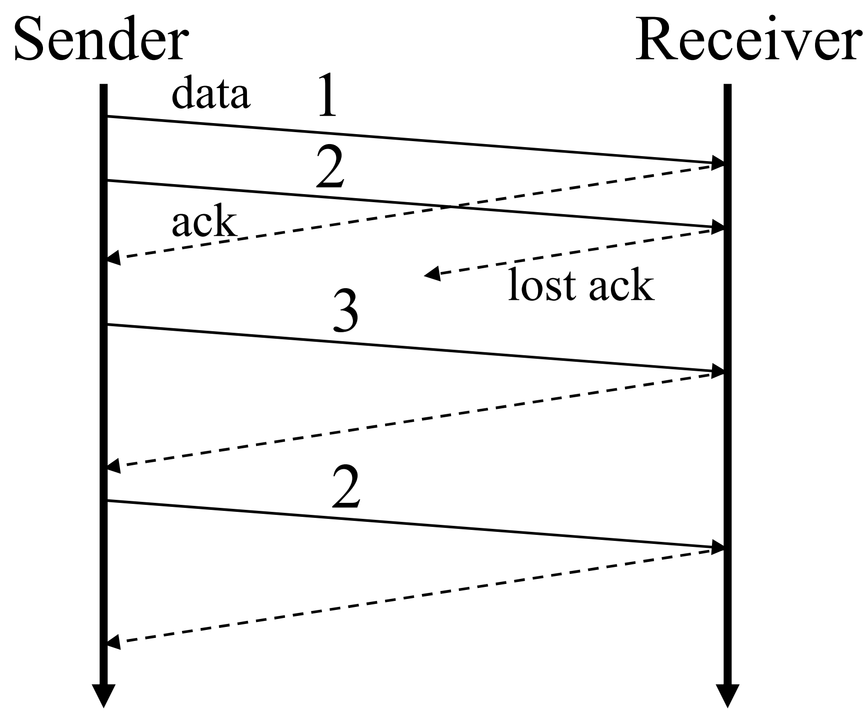
CS557:
Basic TCP Mechanisms

Christos Papadopoulos

Introduction to TCP

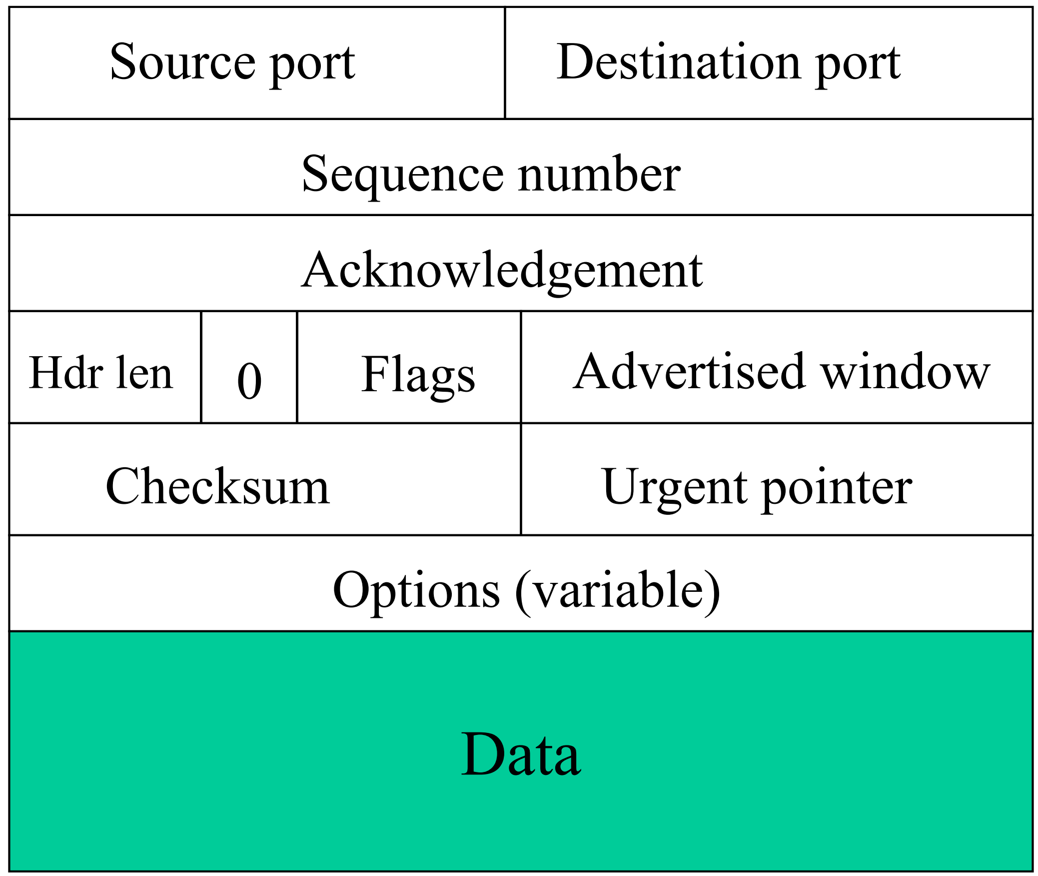
- Communication abstraction:
  - Reliable
  - Ordered
  - Point-to-point
  - Byte-stream
- Protocol implemented entirely at the ends
  - Assumes unreliable, non-sequenced delivery
  - Fate sharing
- Operations
  - OPEN/LISTEN, CONNECT, SEND, RECEIVE, ABORT

TCP Reliability Mechanism



TCP Header

Flags:
- SYN
- FIN
- RESET
- PUSH
- URG
- ACK

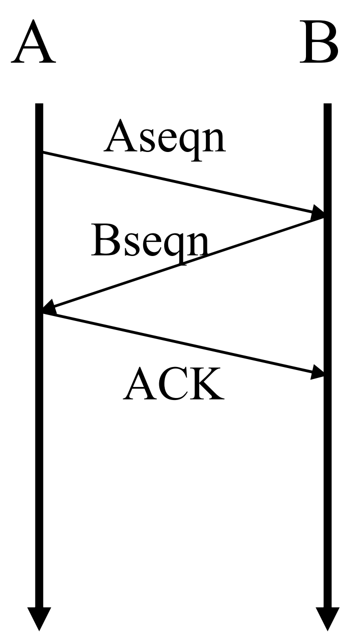


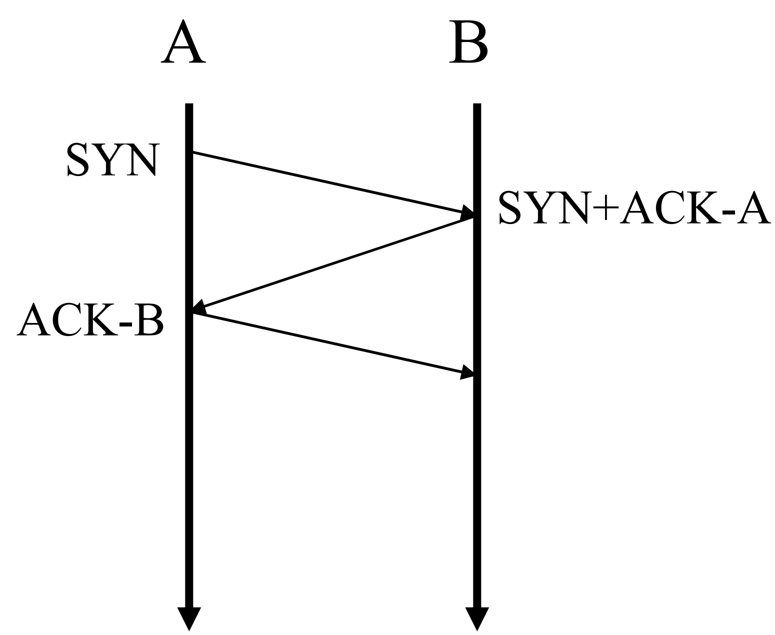
TCP Mechanisms

- Connection establishment
- Sequence number selection
- Connection tear-down
- Round-trip estimation
- Window flow control

Connection Establishment

A and B must agree on initial sequence number selection:
Use 3-way handshake





Connection Setup

```
                              +---------+ ---------\      active OPEN
                              |  CLOSED |            \    -----------
                              +---------+<---------\   \   create TCB
                                |     ^              \   \  snd SYN
                   passive OPEN |     |   CLOSE        \   \
                   ------------ |     | ----------       \   \
                    create TCB  |     | delete TCB         \   \
                                V     |                      \   \
                              +---------+            CLOSE    |    \
                              |  LISTEN |          ---------- |     |
                              +---------+          delete TCB |     |
                   rcv SYN      |     |     SEND              |     |
                  -----------   |     |    -------            |     V
 +---------+      snd SYN,ACK  /       \   snd SYN          +---------+
 |         |<-----------------           ------------------>|         |
 |   SYN   |                    rcv SYN                     |   SYN   |
 |   RCVD  |<-----------------------------------------------|   SENT  |
 |         |                    snd ACK                     |         |
 |         |------------------           -------------------|         |
 +---------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +---------+
   |           --------------   |     |   -----------
   |                  x         |     |     snd ACK
   |                            V     V
   |  CLOSE                   +---------+
   | -------                  |  ESTAB  |
   | snd FIN                  +---------+
```

Sequence Number Selection

- Initial sequence number (ISN) selection
  - Why not simply chose 0?
  - Must avoid overlap with earlier incarnation
- Possible solutions
  - Assume non-volatile memory
  - Clock-based solutions
- Requirements for ISN selection
  - Must operate correctly
    - Without synchronized clocks
    - Despite node failures

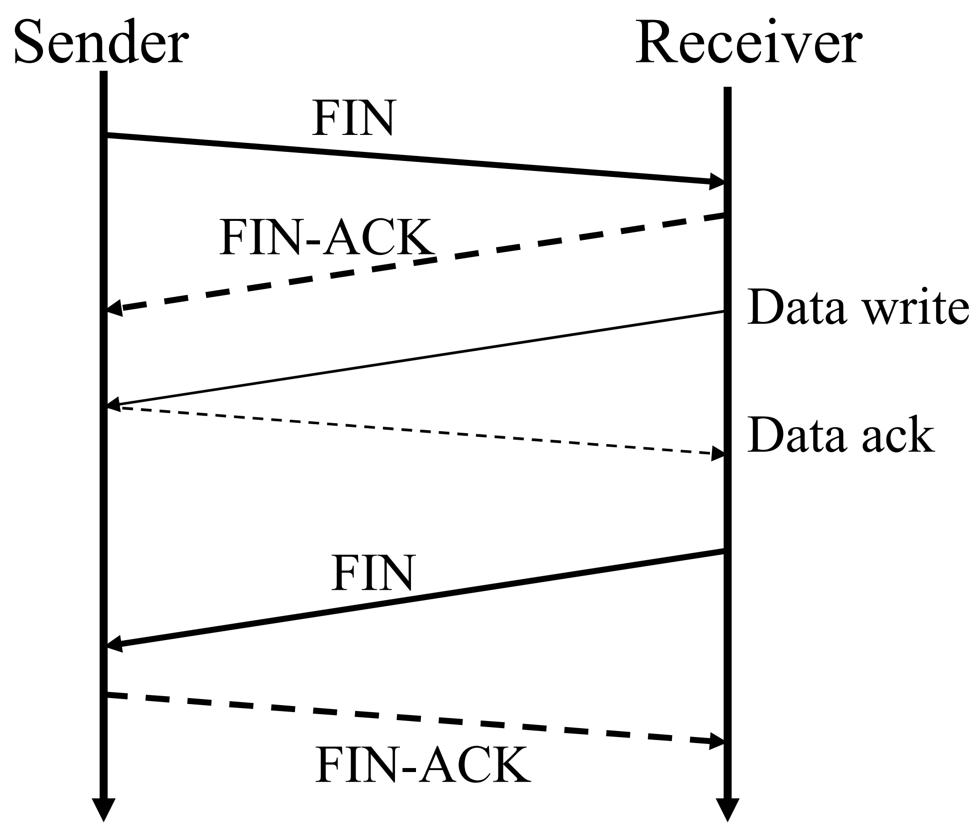
ISN and Quiet Time

- Assume upper bound on segment lifetime (MSL)
  - In TCP, this is 2 minutes
- Upon startup, cannot assign sequence numbers for MSL seconds
- Can still have sequence number overlap
  - If sequence number space not large enough for high-bandwidth connections

Connection Tear-down

- Normal termination
  - Allow unilateral close
  - Avoid sequence number overlap
- TCP must continue to receive data even after closing
  - Cannot close connection immediately: what if a new connection restarts and uses same sequence number?

Tear-down Packet Exchange



Connection Tear-down

```
   |  CLOSE                      +---------+
   | -------                     |  ESTAB  |
   | snd FIN                     +---------+
   |                     CLOSE    |     |    rcv FIN
   V                    -------   |     |    -------
 +---------+          snd FIN  /       \   snd ACK          +---------+
 |  FIN    |<-----------------           ------------------>|  CLOSE  |
 | WAIT-1  |------------------                              |   WAIT  |
 +---------+          rcv FIN  \                            +---------+
   | rcv ACK of FIN   -------   |                            CLOSE  |
   | --------------   snd ACK   |                           ------- |
   V        x                   V                           snd FIN V
 +---------+                  +---------+                   +---------+
 |FINWAIT-2|                  | CLOSING |                   | LAST-ACK|
 +---------+                  +---------+                   +---------+
   |                rcv ACK of FIN |                 rcv ACK of FIN |
   |  rcv FIN       -------------- |    Timeout=2MSL -------------- |
   |  -------              x       V    ------------        x       V
    \ snd ACK                 +---------+delete TCB         +---------+
     ------------------------>|TIME WAIT|------------------>| CLOSED  |
                              +---------+                   +---------+
```

Detecting Half-open Connections

```
    TCP A                                                  TCP B

1.  (CRASH)                                  (send 300,receive 100)

2.  CLOSED                                            ESTABLISHED

3.  SYN-SENT --> <SEQ=400><CTL=SYN>               --> (??)

4.  (!!)     <-- <SEQ=300><ACK=100><CTL=ACK>      <-- ESTABLISHED

5.  SYN-SENT --> <SEQ=100><CTL=RST>               --> (Abort!!)

6.  SYN-SENT                                          CLOSED

7.  SYN-SENT --> <SEQ=400><CTL=SYN>               -->
```

TIME-WAIT Assassination

```
     TCP A                                                    TCP B

 1.  ESTABLISHED                                              ESTABLISHED

     (Close)
 2.  FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

 3.  FIN-WAIT-2  <-- <SEQ=300><ACK=101><CTL=ACK>      <-- CLOSE-WAIT

                                                          (Close)
 4.  TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

 5.  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>      --> CLOSED

- - - - - - - - - - - - - - - - - - - - - - - - - - - -

 5.1. TIME-WAIT   <--  <SEQ=255><ACK=33> ... old duplicate

 5.2  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>    -->  ????

 5.3  CLOSED      <-- <SEQ=301><CTL=RST>             <--  ????
    (prematurely)
```

Round-trip Time Estimation

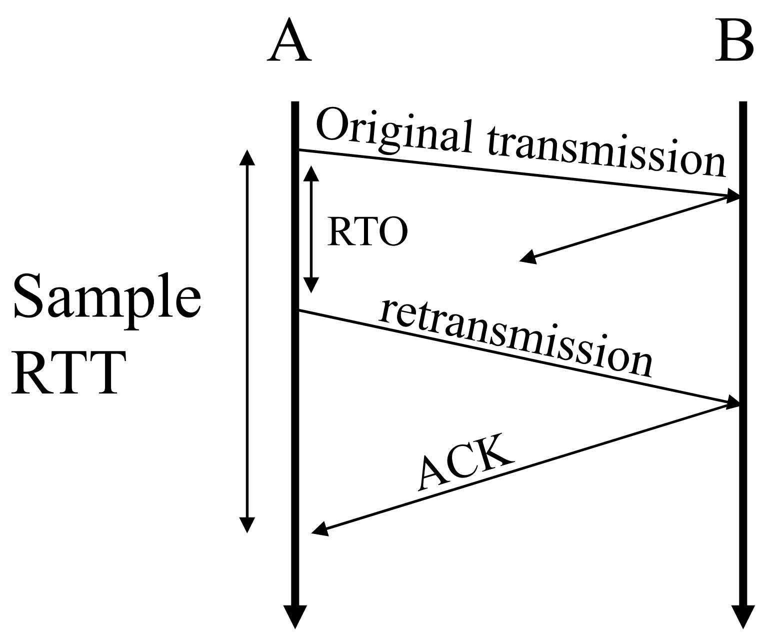
- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT -> unneeded retransmissions
  - High RTT -> poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!

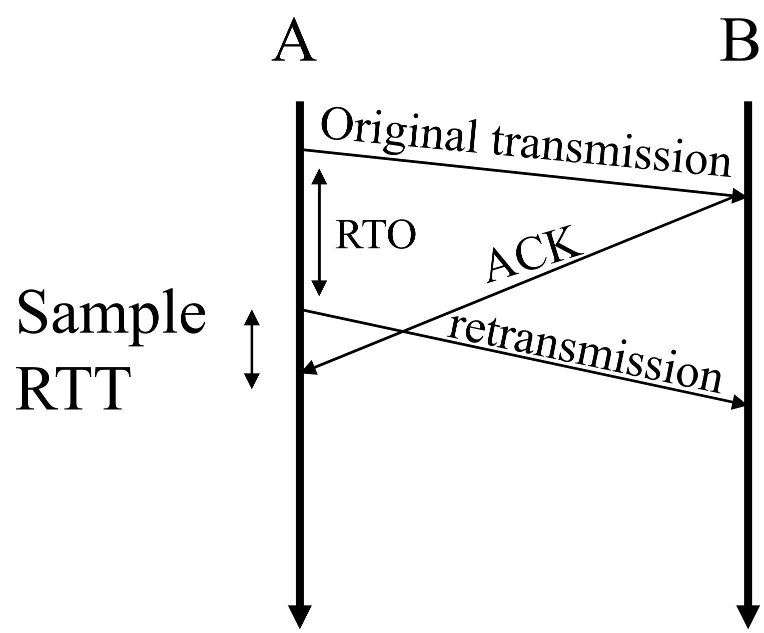
Initial Round-trip Estimator

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 time timer expires, RTO exponentially backed-off

Retransmission Ambiguity





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

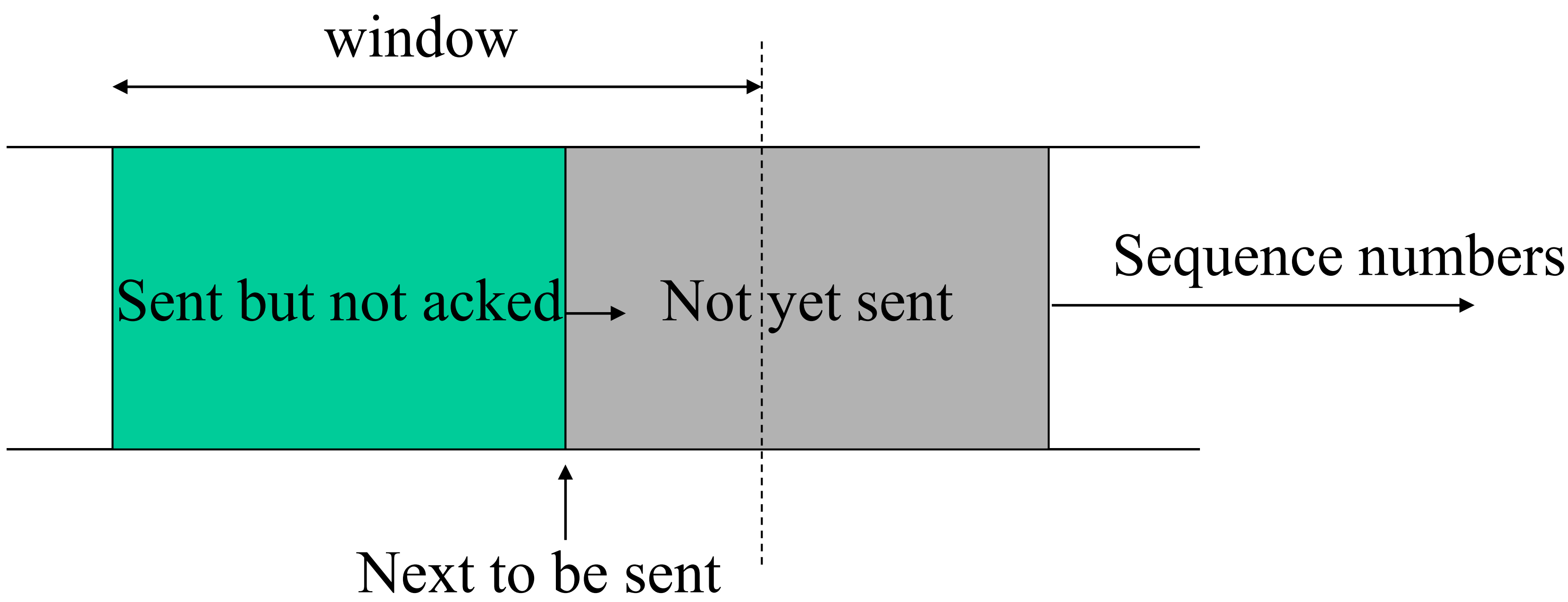
Jacobson's Retransmission Timeout Estimator

- Key observation:
  - Using β∗RTT for timeout doesn't work
  - At high loads round trip variance is high
- Solution:
  - If D denotes mean variation
  - Timeout = RTT + 4D

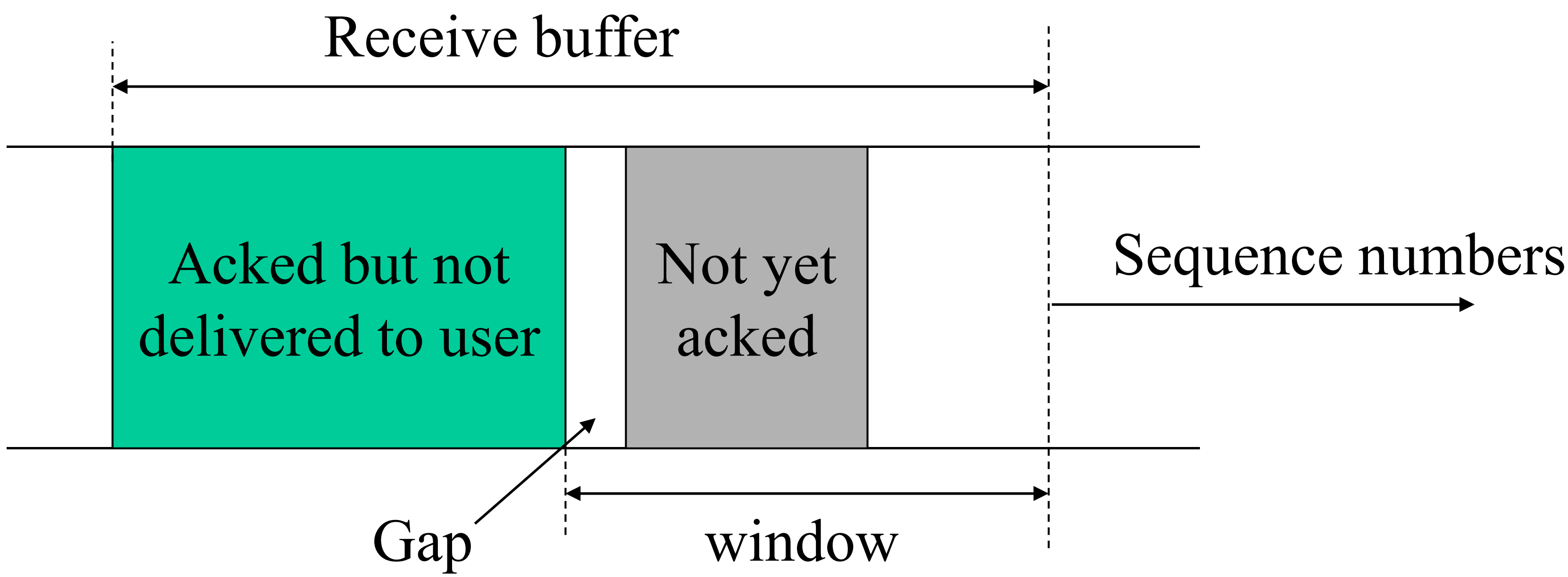
Flow Control

- Problem: Fast sender can overrun receiver
  - Packet loss, unnecessary retransmissions
- Possible solutions:
  - Sender transmits at pre-negotiated rate
  - Sender limited to a window's worth of unacknowledged data
- Flow control different from congestion control

Window Flow Control: Send



Window Flow Control: Receive



Window Advancement Issues

- Advancing a full window
  - When the receive window fills up, how do things get started again?
  - Sender sends periodic probe while receive win is 0
- Silly window syndrome
  - Fast sender, slow receiver
  - Delayed acks at receiver help, but not a full solution
- The small packet problem and Nagle's algorithm
  - If window < 1MSS when do I send?
  - Delay sending if un-acked data in flight
  - Overwrite with TCP-NODELAY option

TCP Extensions

- Needed for high-bandwidth delay connections
  - Accurate round-trip time estimation
  - Window size limitations
  - Impact of loss
- Implemented using TCP options
  - Timestamp
  - Protection from sequence number wraparound
  - Large windows

Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
- Receiver echoes timestamp in ACK

Protection From Wraparound

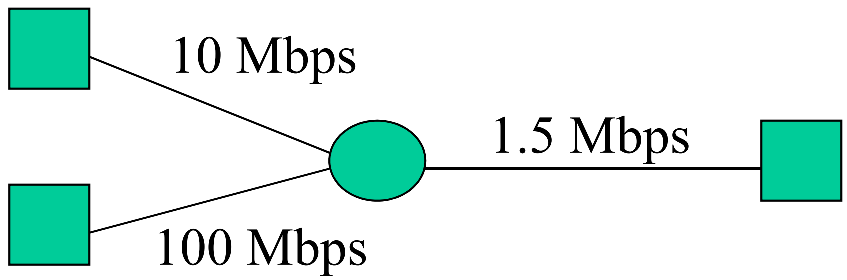
- Wraparound time vs. Link speed:
    - 1.5Mbps: 6.4 hours
    - 10Mbps: 57 minutes
    - 45Mbps: 13 minutes
    - 100Mbps: 6 minutes
    - 622Mbps: 55 seconds
    - 1.2Gbps: 28 seconds
- Use timestamp to distinguish sequence number wraparound

Large Windows

- Apply scaling factor to advertised window
  - Specifies how many bits window must be shifted to the left
- Scaling factor exchanged during connection setup

TCP Congestion Control

Congestion

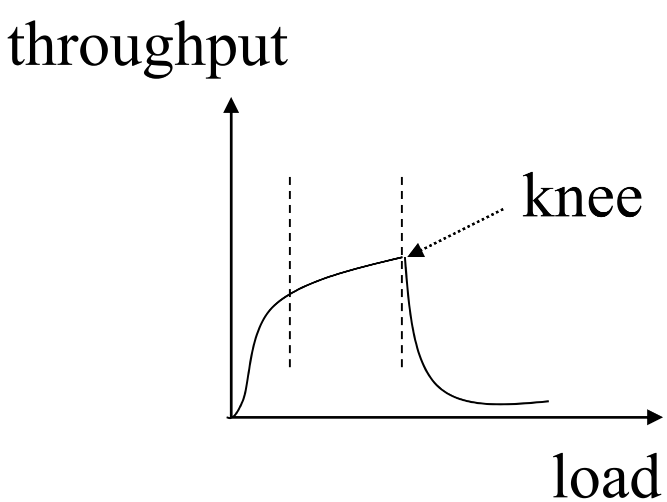


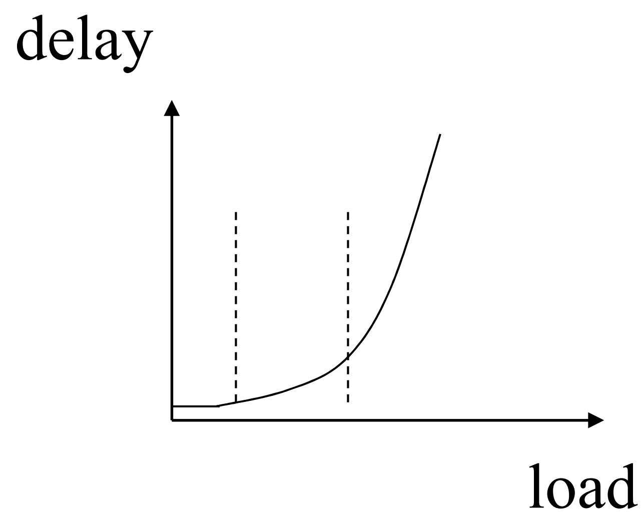
- Caused by fast links feeding into slow link
- Severe congestion may lead to network collapse
  - Flows send full windows, but progress is very slow
  - Most packets in the network are retransmissions
- Other causes of congestion collapse
  - Retransmissions of large packets after loss of a single fragment
  - Non-feedback controlled sources

Congestion Control and Avoidance

- Requirements
  - Uses network resources efficiently
  - Preserves fair network resource allocation
  - Prevents or avoids collapse
- Congestion collapse is not just a theory
  - Has been frequently observed in many networks

Congestion Response





Criteria

- Efficiency:
  - System is most efficient at knee of throughput curve
    - Most throughput without excessive delay
  - One proposed efficiency metric: Power ($throughput^{\alpha}$/delay), where 0<=a<=1
- Fairness:
  - In the absence of knowing requirements, assume a fair allocation means equal allocation
  - Fairness index: $(\Sigma x_i)^2/n(\Sigma x_i^2)$
  - Index ranges between 0..1 with 1 being fair to all flows

Congestion Control Design

- Avoidance or control?
  - Avoidance keeps system at knee of curve
  - But, to do that, need routers to send accurate signals (some feedback)
- Sending host must adjust amount of data it puts in the network based on detected congestion
  - TCP uses its window to do this
  - But what's the right strategy to increase/decrease window

Feedback Control Model

- We study this question using a feedback control model:
  - Reduce window when congestion is perceived
  - Increase window otherwise
- Constraints:
  - Efficiency
  - Fairness
  - Stability or convergence (the system must not oscillate significantly)

Linear Control

$$X_i(t + 1) = a_i + b_i X_i(t)$$

- Formulation allows for the feedback signal:
  - to be increased/decreased additively (by changing a_i)
  - to be increased/decreased multiplicatively (by changing b_i)
- Which of the four combinations is optimal?

TCP Congestion Control

- A collection of interrelated mechanisms:
  - Slow start
  - Congestion avoidance
  - Accurate retransmission timeout estimation
  - Fast retransmit
  - Fast recovery

Congestion Control

- Underlying design principle: Packet Conservation
  - At equilibrium, inject packet into network only when one is removed
  - Basis for stability of physical systems

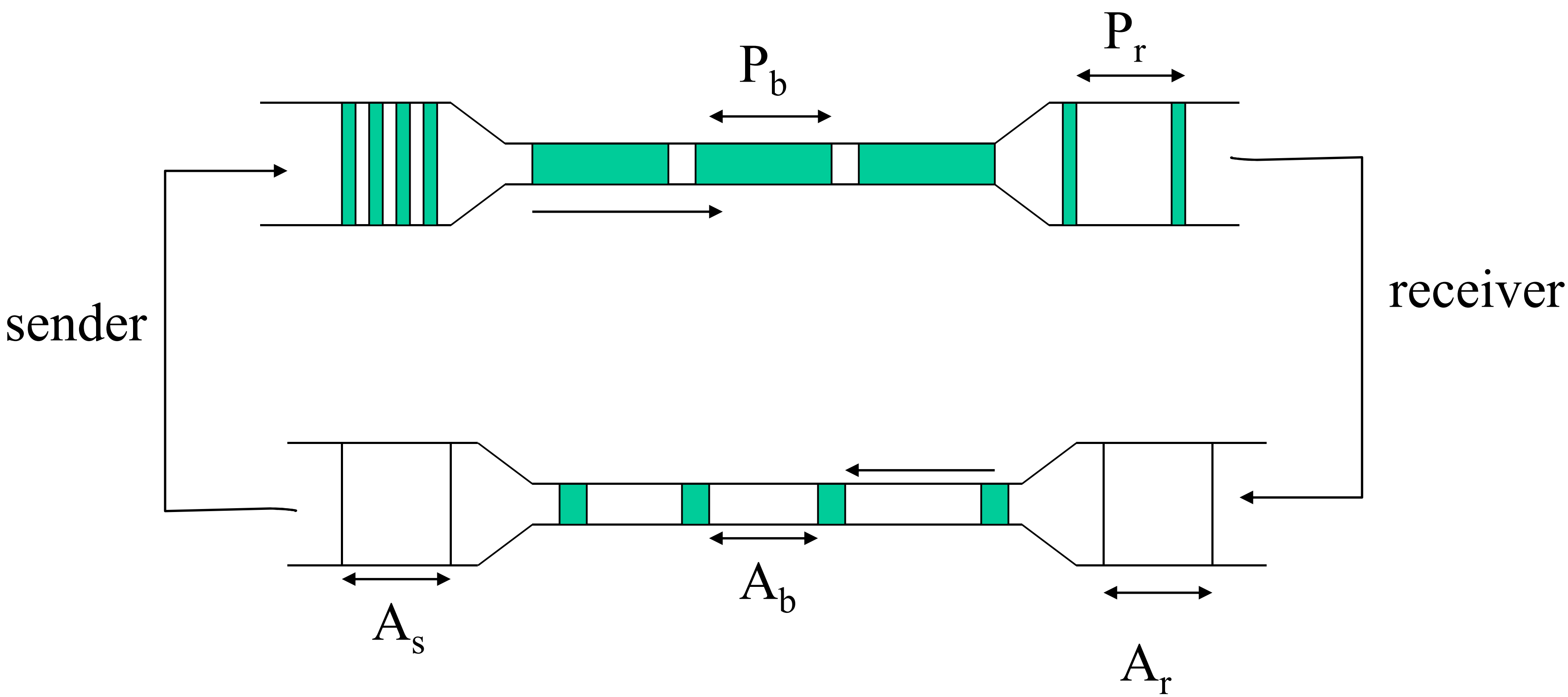
TCP Congestion Control Basics

- Keep a congestion window, cwnd
  - Denotes how much network is able to absorb
- Sender's maximum window:
  - Min (advertised window, cwnd)
- Sender's actual window:
  - Max window - unacknowledged segments

Clocking Packets

- Suppose we have large actual window. How do we send data?
  - In one shot? No, this violates the packet conservation principle
  - Solution: use acks to clock sending new data
  - Ack reception means at least one packet was removed from the network

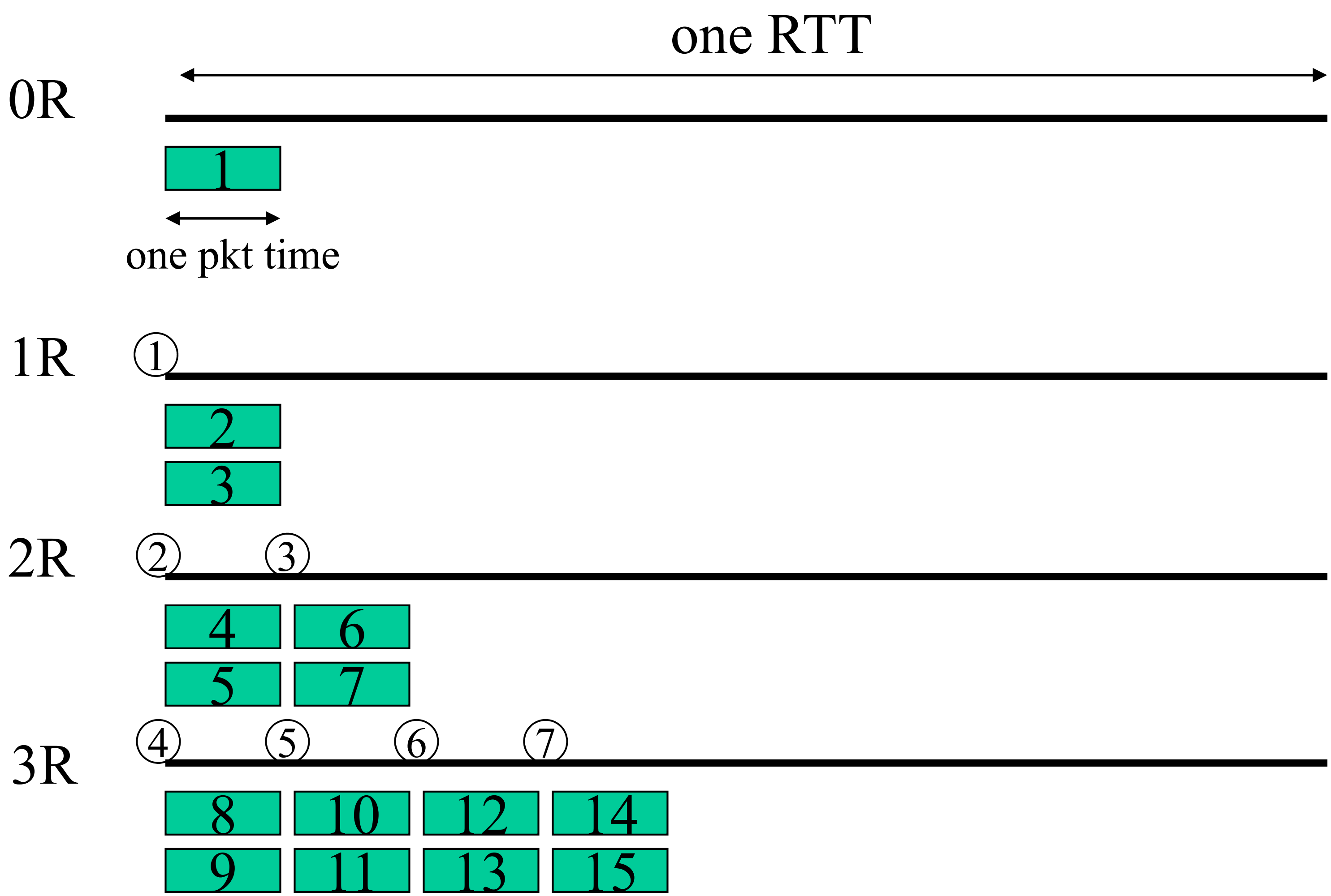
TCP is Self-clocking



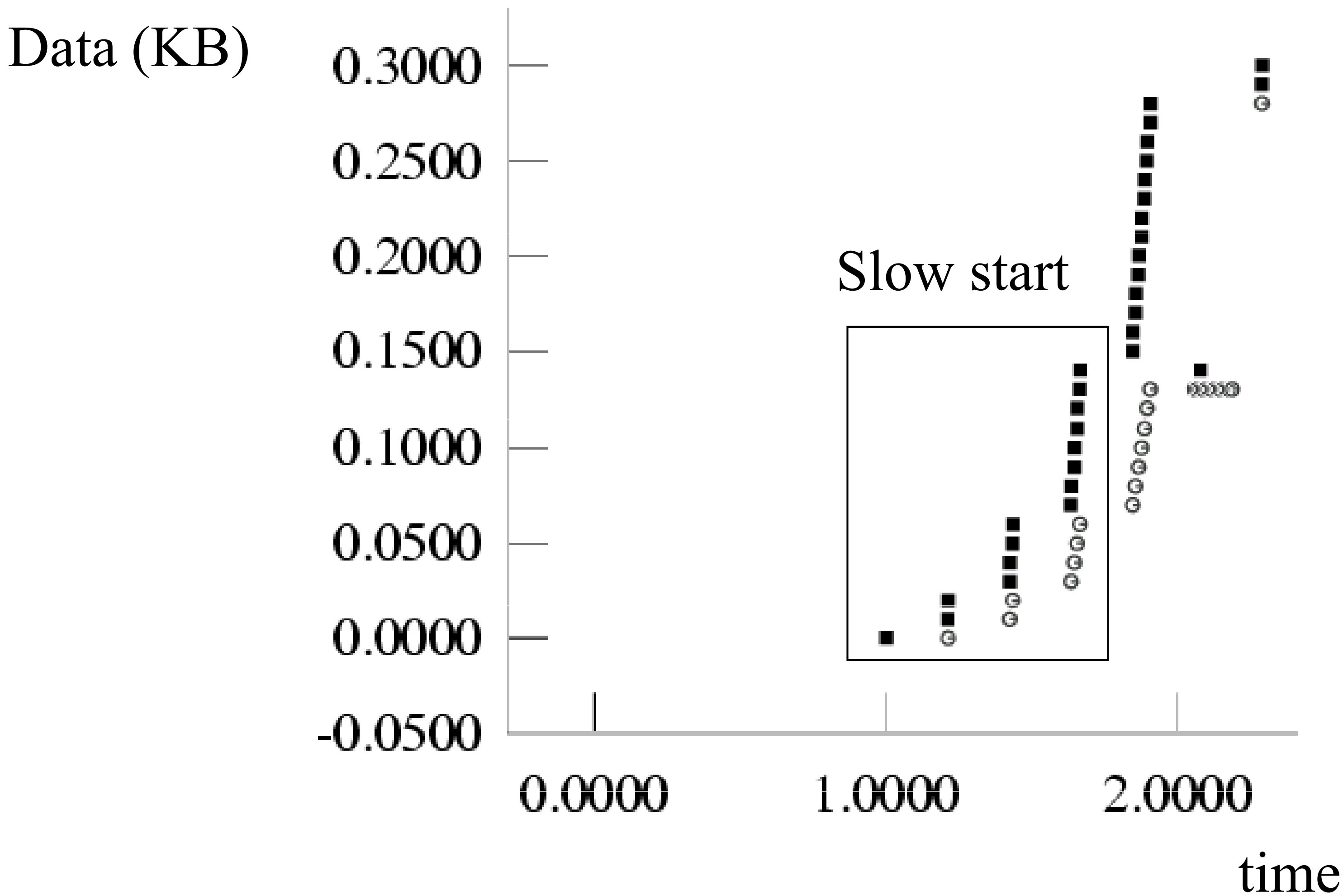
Slow Start

- But how do we get this clocking behavior to start?
  - Initialize cwnd = 1
  - Upon receipt of every ack, cwnd = cwnd + 1
- Implications
  - Window *doubles* on every RTT
  - Can overshoot window and cause packet loss

Slow Start Example

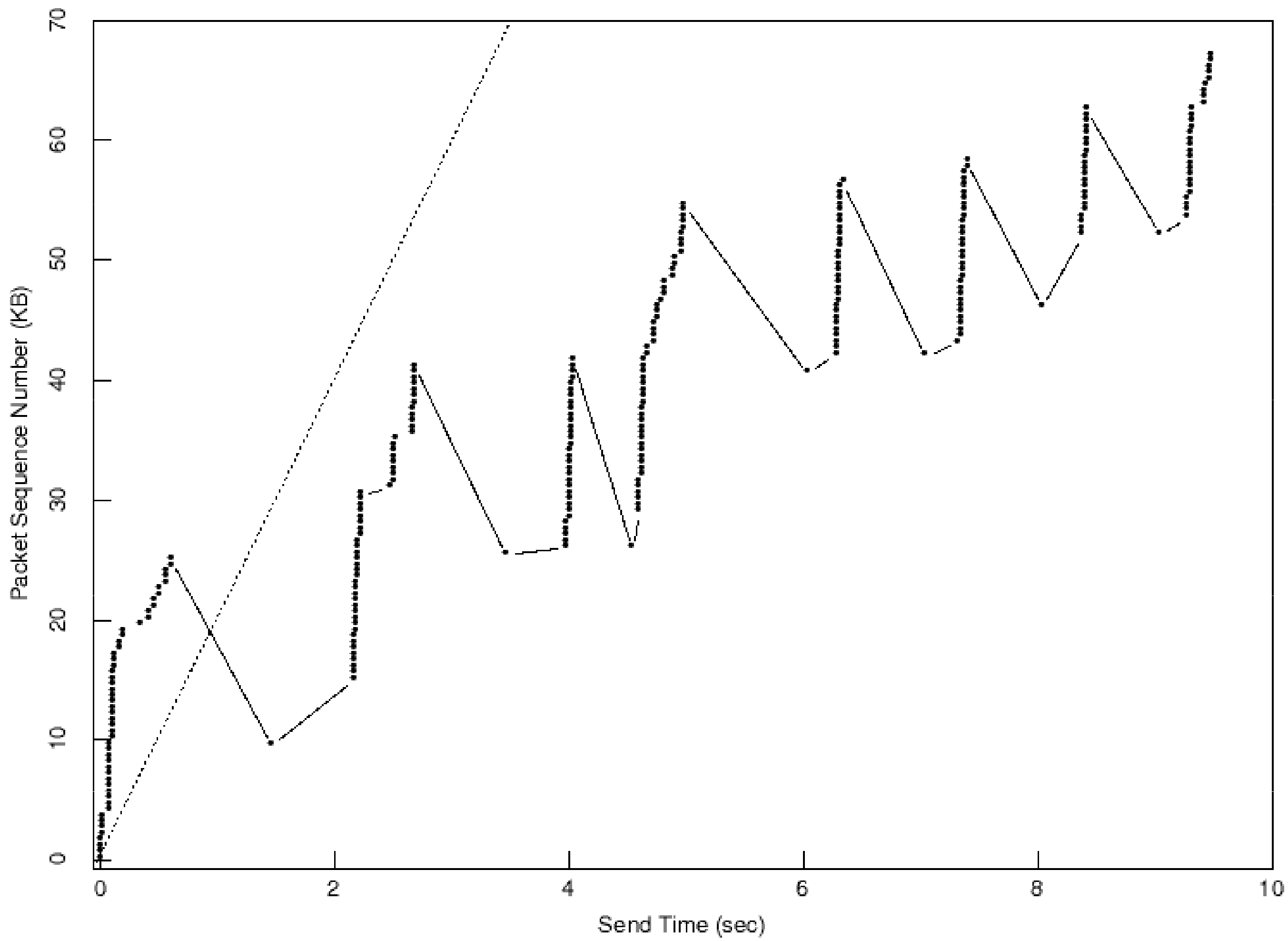


Slow Start Sequence Plot

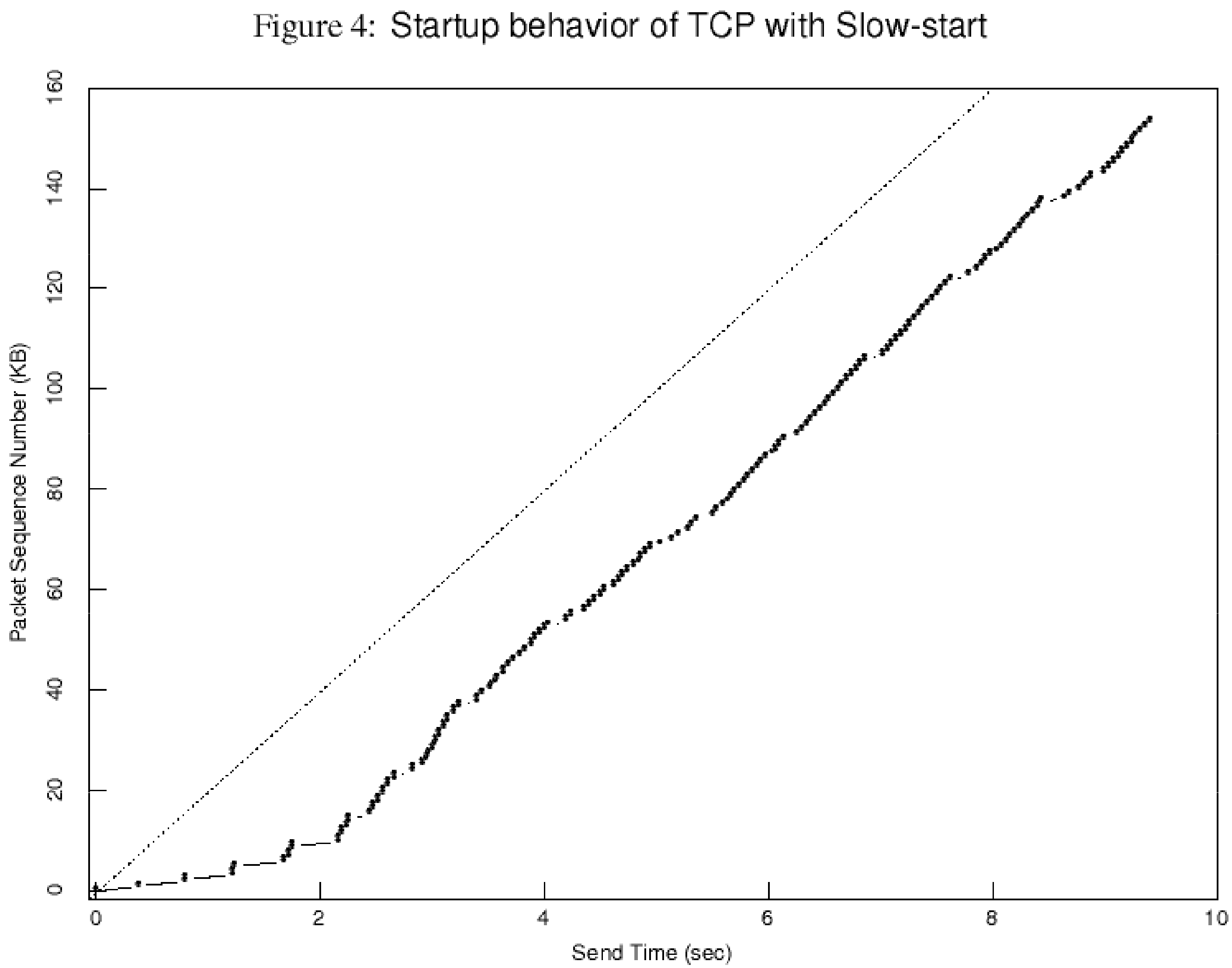


Jacobson, Figure 3: No Slow Start

Figure 3: Startup behavior of TCP without Slow-start



Jacobson, Figure 4: with Slow Start



Congestion Avoidance

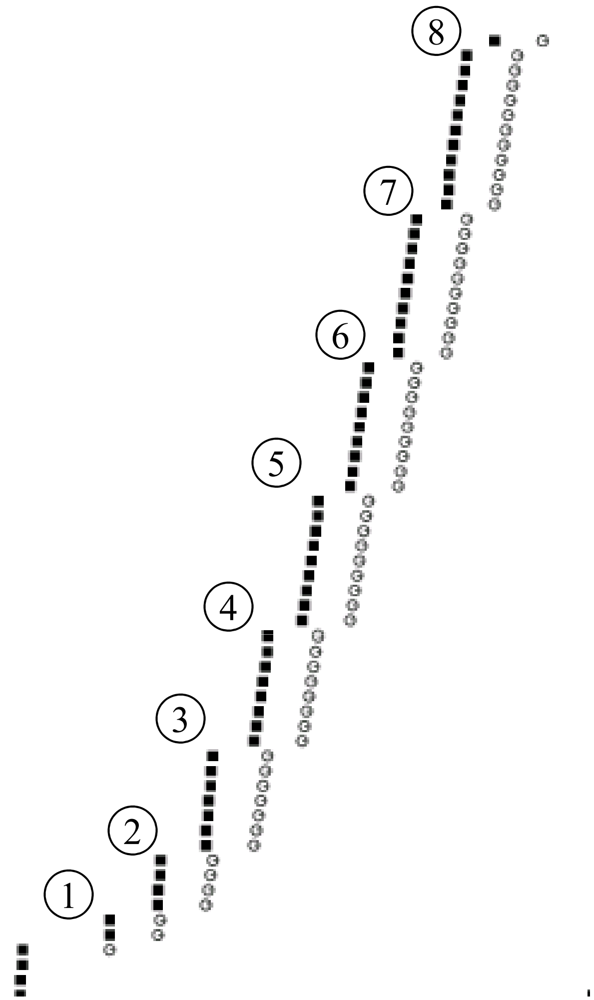
- Coarse grained timeout as loss indicator
- Suppose loss occurs when cwnd = W
  - Network can absorb 0.5W ~ W segments
  - Conservatively set cwnd to 0.5W (multiplicative decrease)
  - Avoid exponential queue buildup
- Upon receiving ACK
  - Increase cwnd by 1/cwnd (additive increase)
  - Multiplicative increase -> non-convergence

Slow Start and Congestion Avoidance

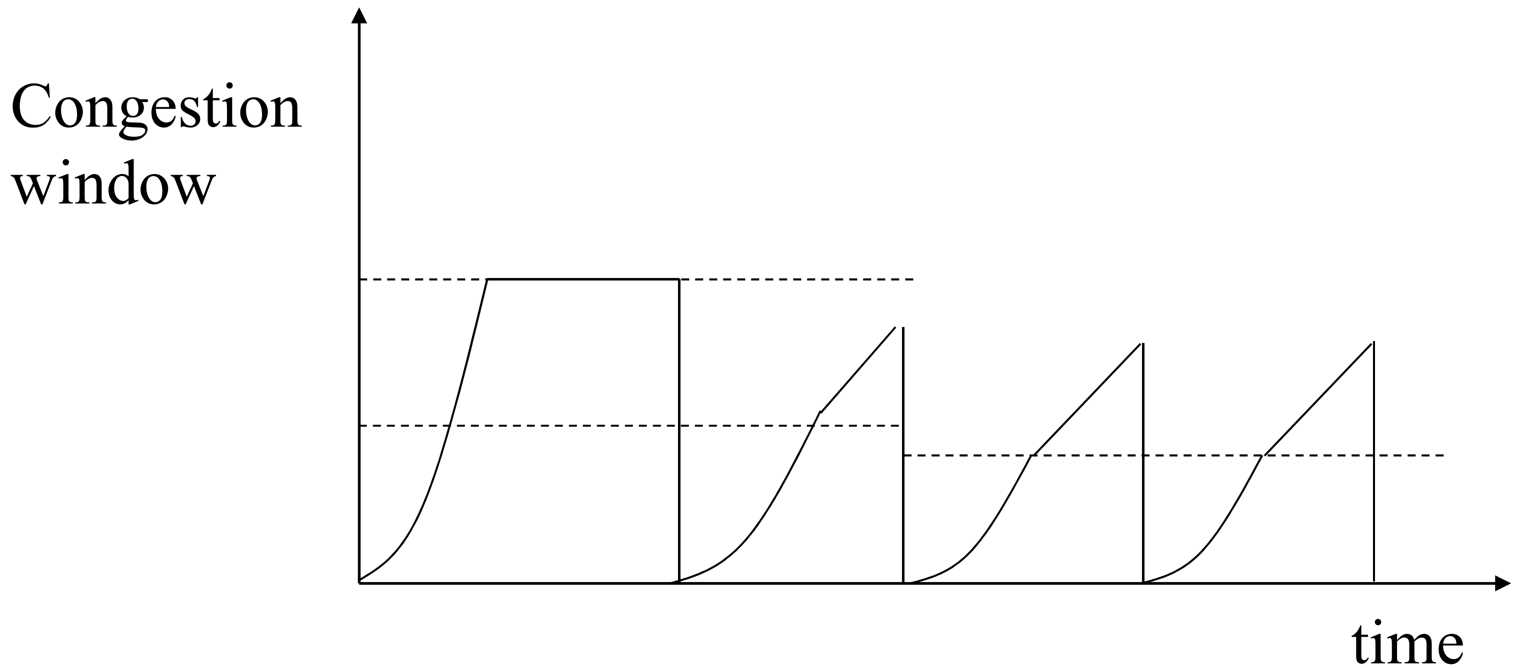
- If packet is lost we lose our self clocking as well – timeout has caused link to go quiet
  - Need to implement slow-start and congestion avoidance together
  - New variable: ssthresh (slow-start threshold)
- When timeout occurs set ssthresh to 0.5W
  - If cwnd < ssthresh, use slow start
  - Else use congestion avoidance

Congestion Avoidance Sequence Plot

Where is the transition
From SS to CA?

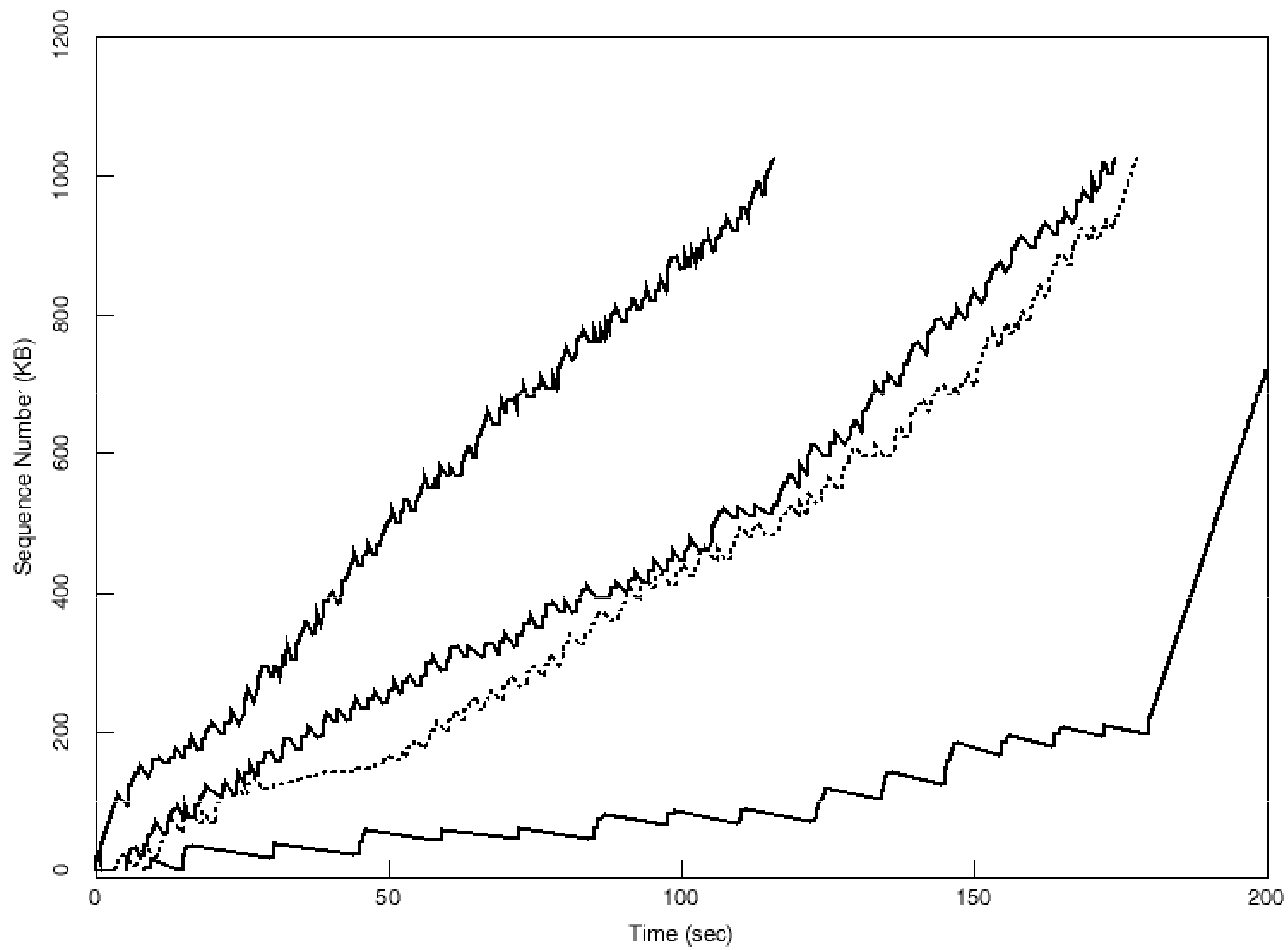


Congestion Window Variation



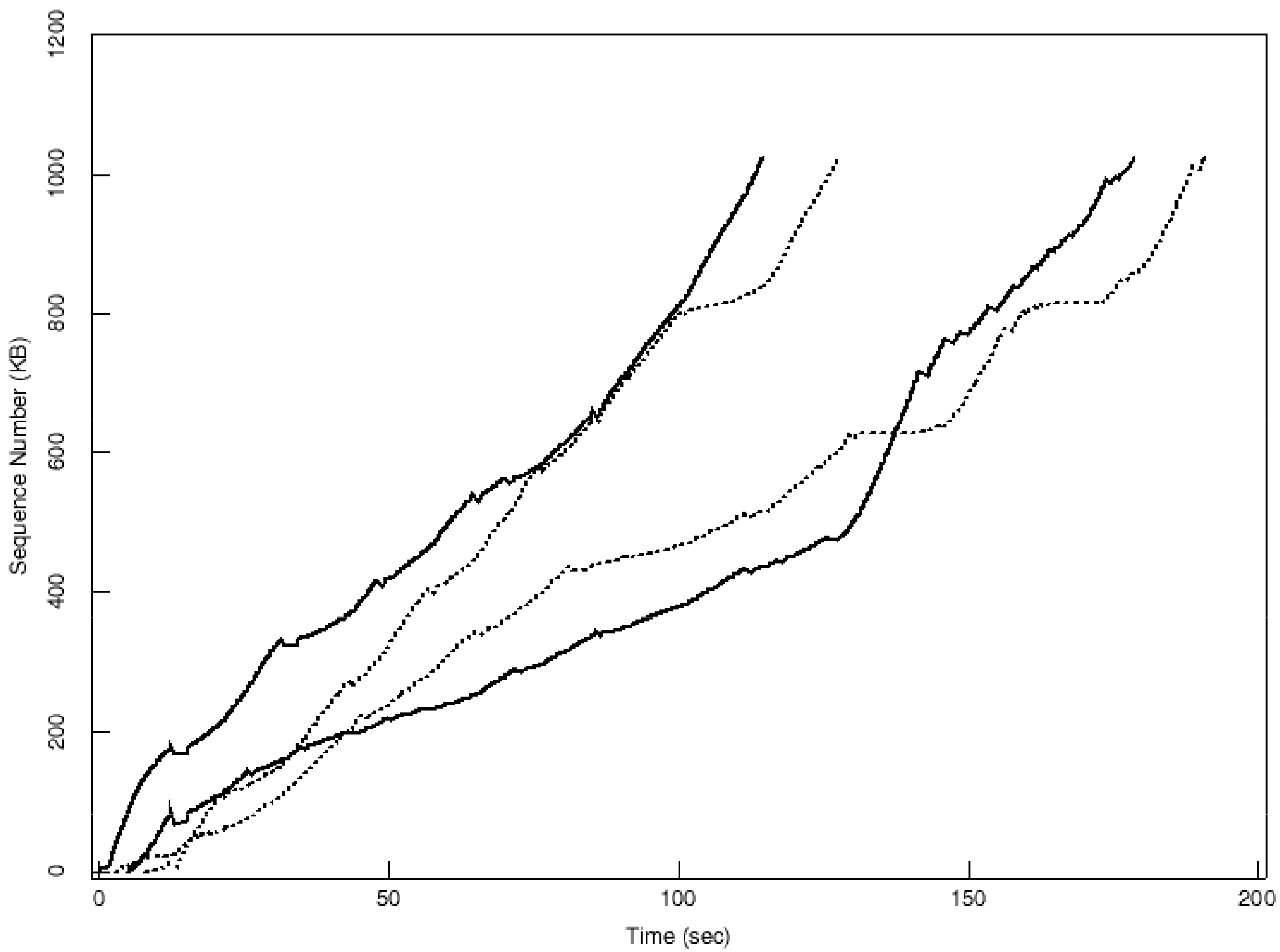
Jacobson, Figure 8: 4x no Congestion avoidance

Figure 8: Multiple, simultaneous TCPs with no congestion avoidance



Jacobson, Figure 9: 4 TCPs with Congestion Avoidance

Figure 9: Multiple, simultaneous TCPs with congestion avoidance



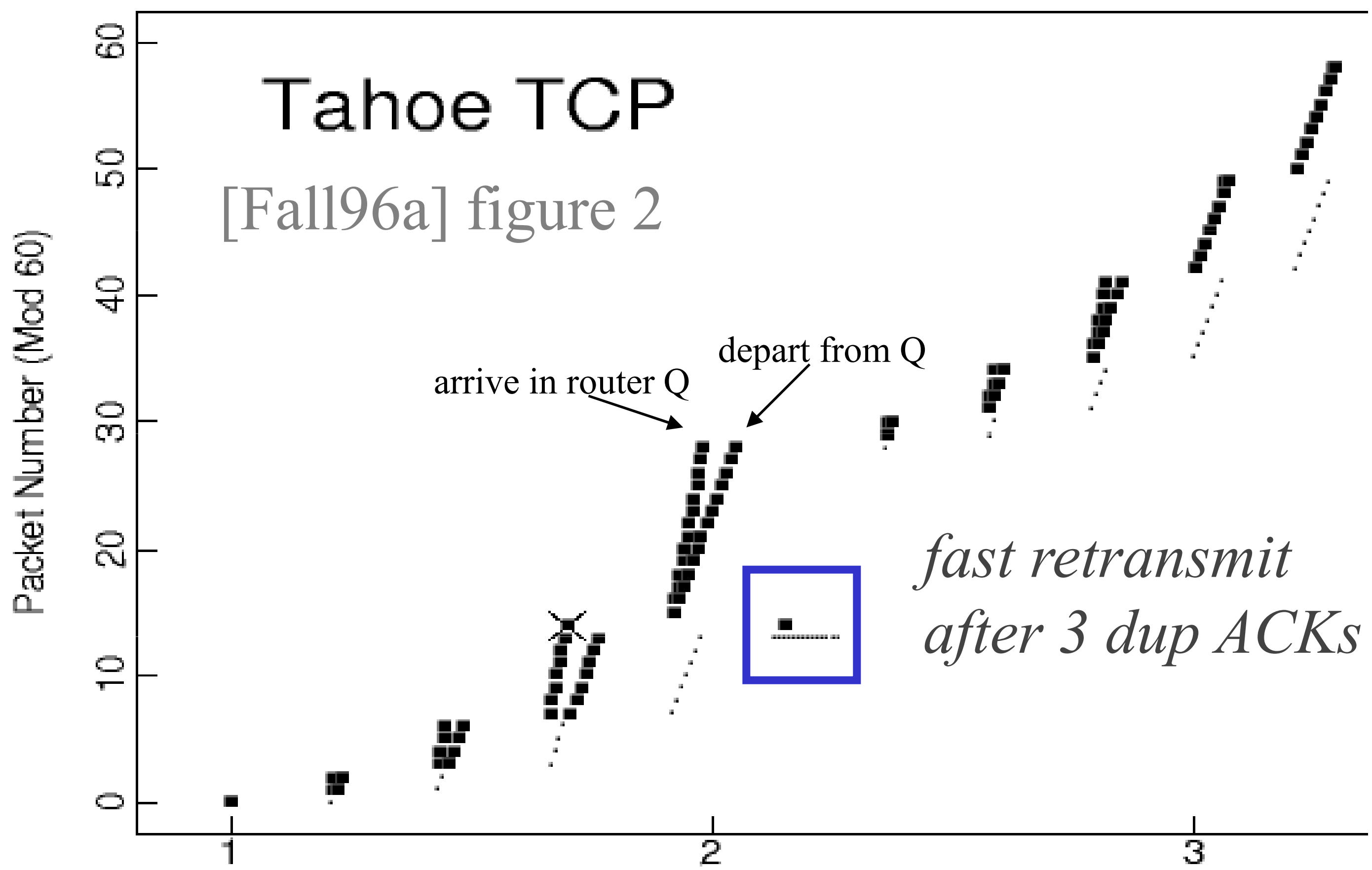
Impact of Timeouts

- Timeouts can cause sender to
  - Slow start
  - Retransmit a possibly large portion of the window
- Bad for lossy high bandwidth-delay paths
- Can leverage duplicate acks to:
  - Retransmit fewer segments (fast retransmit)
  - Advance cwnd more aggressively (fast recovery)

Fast Retransmit

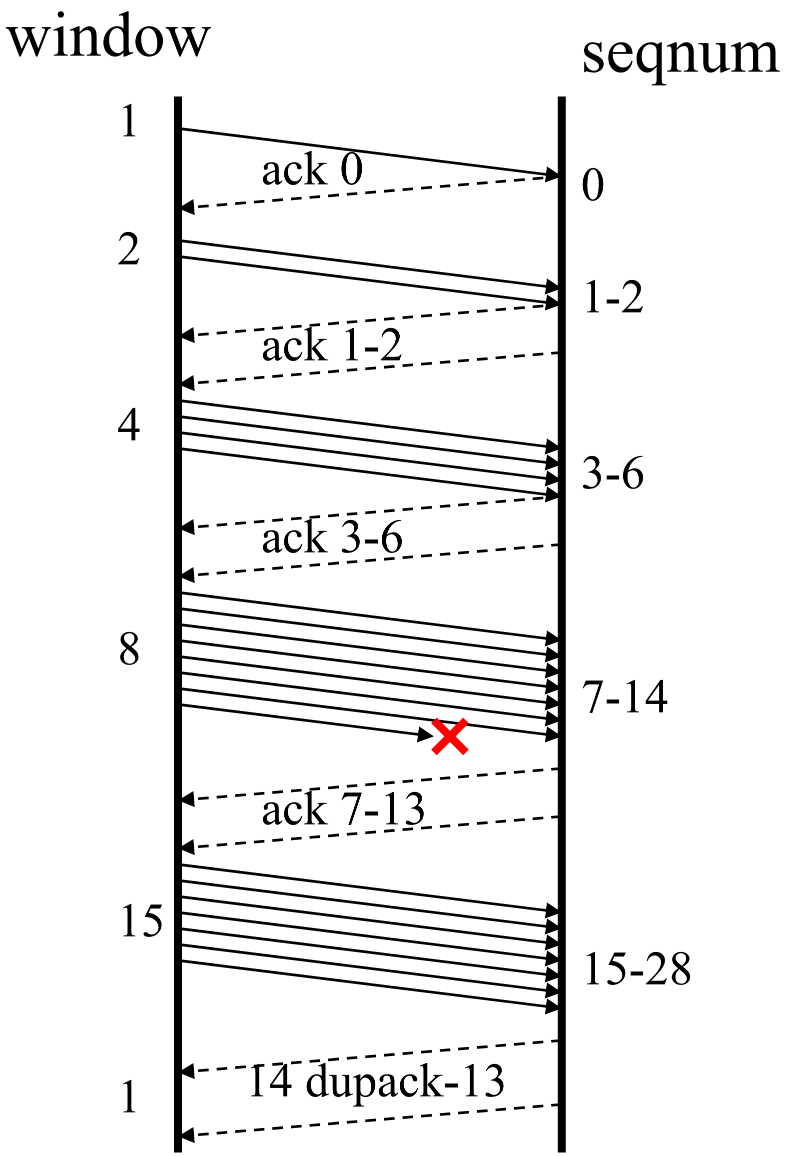
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
- Assume packet re-ordering is infrequent
  - Use receipt of 3 or more duplicate acks as indication of loss
  - Retransmit that segment before timeout
  - Value of 3 was a guess initially, but later validated through experiments by Paxson

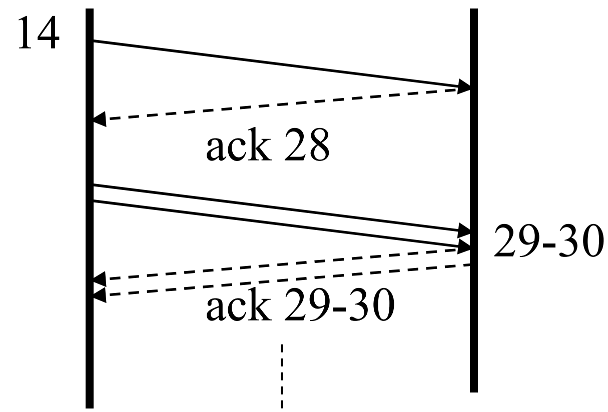
Fast Retransmit Example



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

Fast Retransmit - 1 Drop





<u>Actions after dupacks for pkt 13:</u>

1. On 3rd dupack 13 enter fast rtx
2. Set ssthresh = 15/2 = 7
3. Set cwnd = 1, retransmit 14
4. Receiver cached 15-28, acks 28
5. cwnd++ continue with slow start
6. At pkt 35 enter congestion avoidance

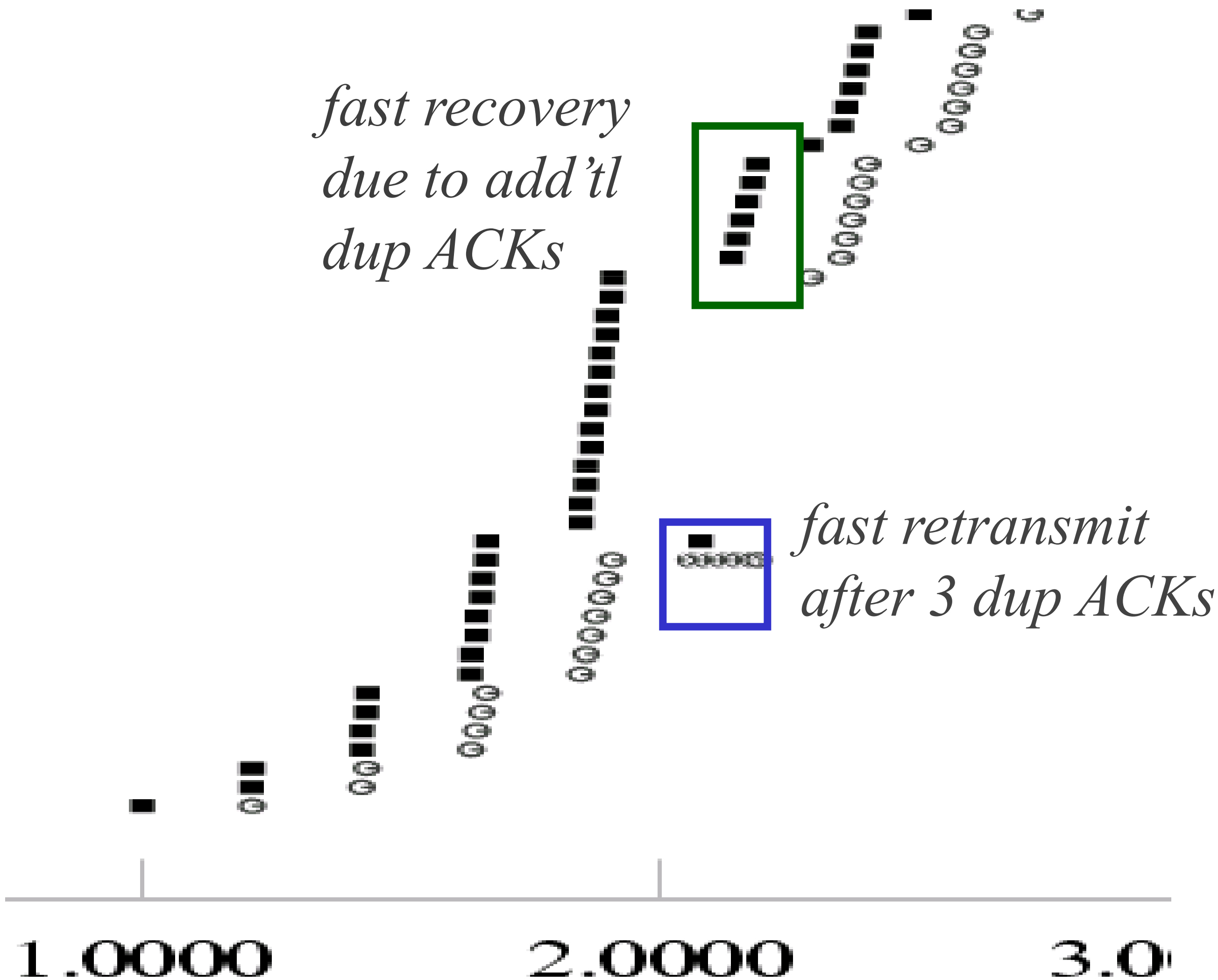
Fast Recovery

- In congestion avoidance mode, if three duplicate acks are received we reduce cwnd to half
- But if n successive duplicate acks are received, we know that receiver got n segments after lost segment
  - Allowed to advance cwnd by that number
  - Does not violate packet conservation

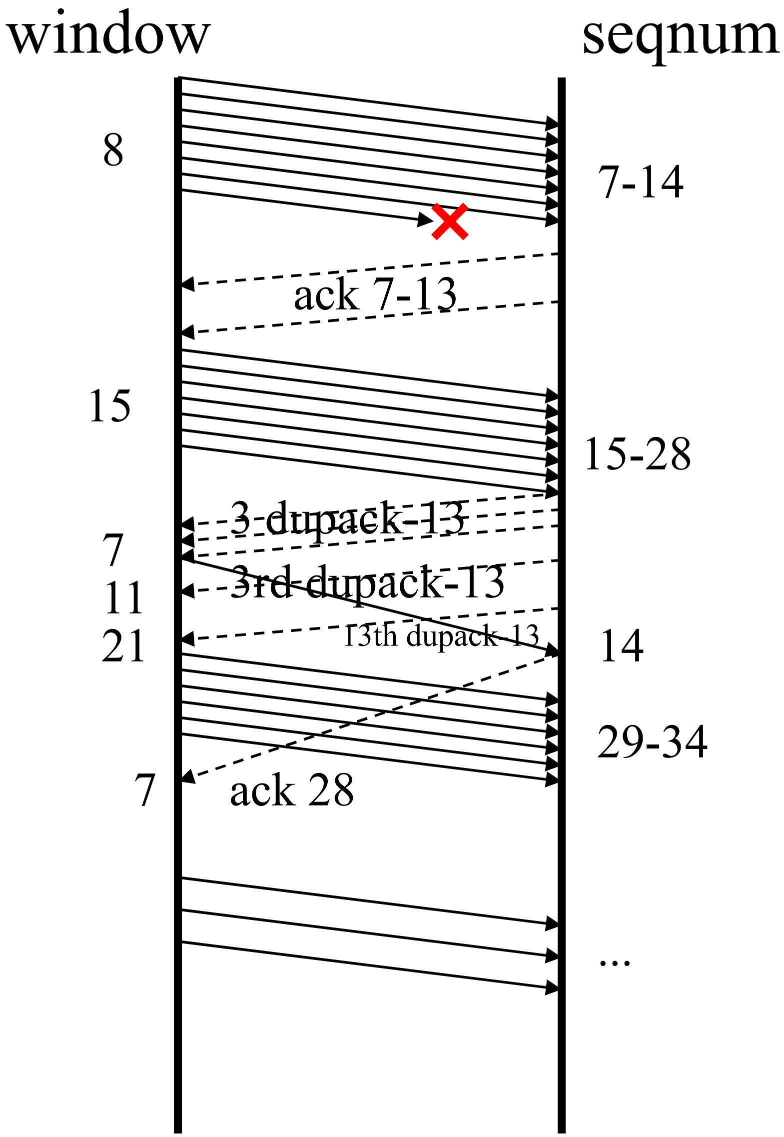
Fast Retransmit and Recovery

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

Fast Recovery Example



Fast Recovery - 1 Drop



Actions after dupacks for pkt 13:

1. On 3rd dupack 13 enter fast recovery
2. Set ssthresh = cwnd = 15/2 = 7
3. retransmit 14
4. Receipt of 3rd dupack sets W=11
5. By 13th dupack, W = 21, send 29-34
6. After ack 28, exit fast recovery
7. Set cwnd = 7
7. Continue with congestion avoidance

TCP Flavors

- Tahoe, Reno, New-Reno, SACK
- TCP Tahoe (distributed with 4.3BSD Unix)
  - Original implementation of van Jacobson's mechanisms (VJ paper)
  - Includes:
    - Slow start (exponential increase of initial window)
    - Congestion avoidance (additive increase of window)
    - Fast retransmit (3 duplicate acks)

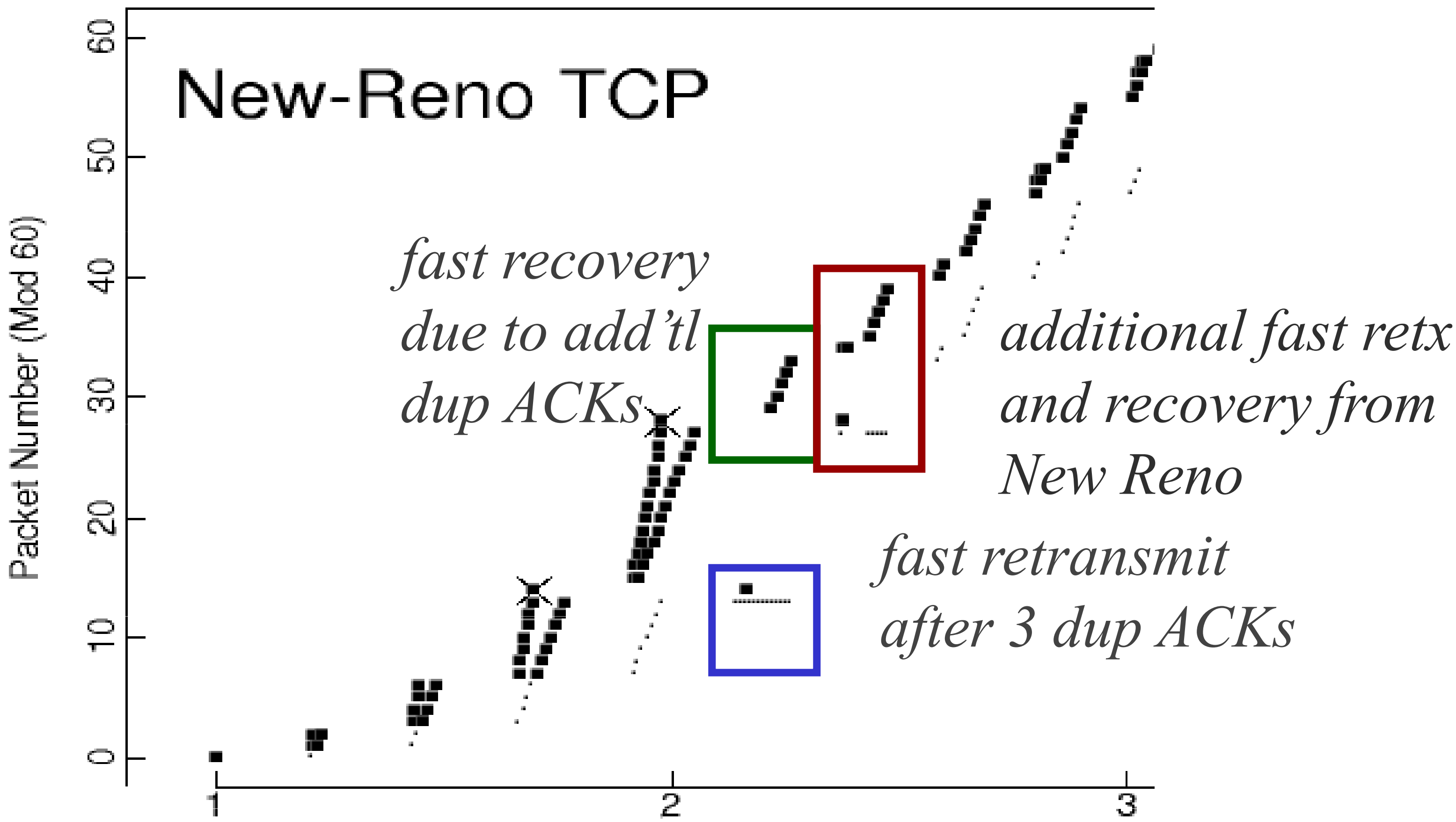
TCP Reno

- 1990: includes:
  - All mechanisms in Tahoe
  - Addition of fast-recovery (opening up window after fast retransmit)
  - Delayed acks (to avoid silly window syndrome)
  - Header prediction (to improve performance)
- Most widely deployed variant

TCP New-Reno

- In Reno's fast recovery, multiple packet drops within window can cause window to deflate prematurely
- In New-Reno
  - Remember outstanding packets at start of fast recovery
  - If new ack is only a partial ACK, assume following segment was lost and resend, don't exit fast recovery

New-Reno Example



TCP Sack

- Reno suffers timeouts with more than 2 losses per window
- New-Reno avoids that, but can only re-send *one dropped packet per RTT*
  - Because it can learn of multiple losses only once per RTT
- TCP SACK
  - Implements the SACK option in TCP
  - Can transmit more than one dropped packet because the sender *now knows which packet was dropped*
  - Sends dropped packets in preference to new data

Other Issues in High BW - Delay Networks

- Slow start too slow
  - Takes several RTTs to open window to proper size
- Restart after long idle time
  - May dump large burst in the network

Connection Hijacking

- Problem:
  - some systems authenticate based on TCP connections
  - if you can *steal* a running TCP connection, you're in
  - it *is* possible, but not easy

Other Performance Issues

Misbehaving TCP implementations

- Misbehaving Sender:
  - Ignore slow start
- Misbehaving Receiver (Savage, 1999)
  - ACK division: open up congestion window faster
  - DupACK spoofing: send multiple dup acks to inflate window
  - Optimistic Acking: send acks for packets you didn’t receive yet – emulates shorter RTT
- Above problems are implementation dependent

SYN Attacks

- Problem:
  - Easy to take over computers (*zombies*) and stage SYN attacks

  ⇒Overflows listen queue, wastes kernel resources (TCB)
- Mitigation: SYN cookies
  - rather than make a new TCB for a new (probably bogus) connection, encode the info in the ISN on the SYN-ACK
  - when you get the ACK, recreate the missing state