Lecture #7: TCP

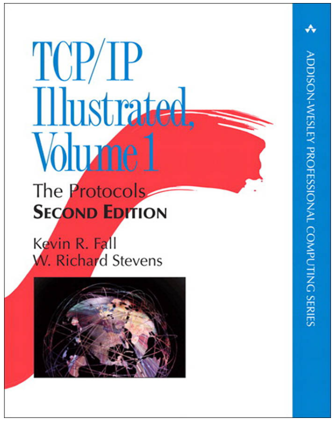
WPI CS4516 Spring 2019 D term

Instructor: Lorenzo De Carli (ldecarli@wpi.edu)

(slides include material from Christos Papadopoulos, CSU)

Sources

- Fall and Stevens, "TCP/IP Illustrated Vol. 1", 2nd edition



- "Congestion Avoidance and Control", Jacobson and Karels, SIGCOMM 1988

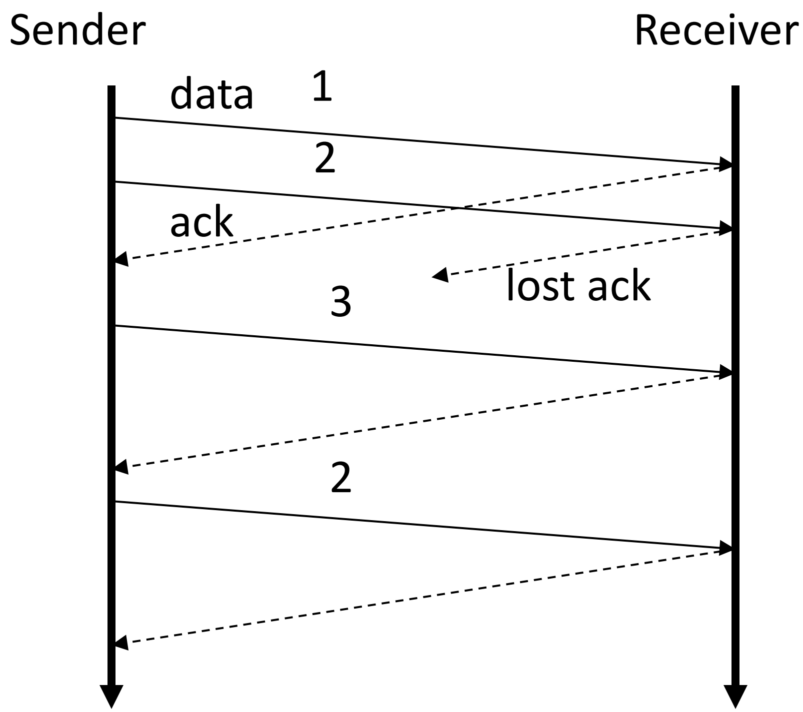
Why is TCP needed?

- Can’t we send everything on top of IP?
  - **No reordering**, **retransmissions**, etc.
- Applications **care about sending data, not packets**
  - In many cases, the **appropriate abstraction** is a **pipe** between sender and receiver
  - TCP provides that pipe
- Also, TCP **ensures that available bandwidth is used** while striving to **prevent congestion**

Introduction to TCP

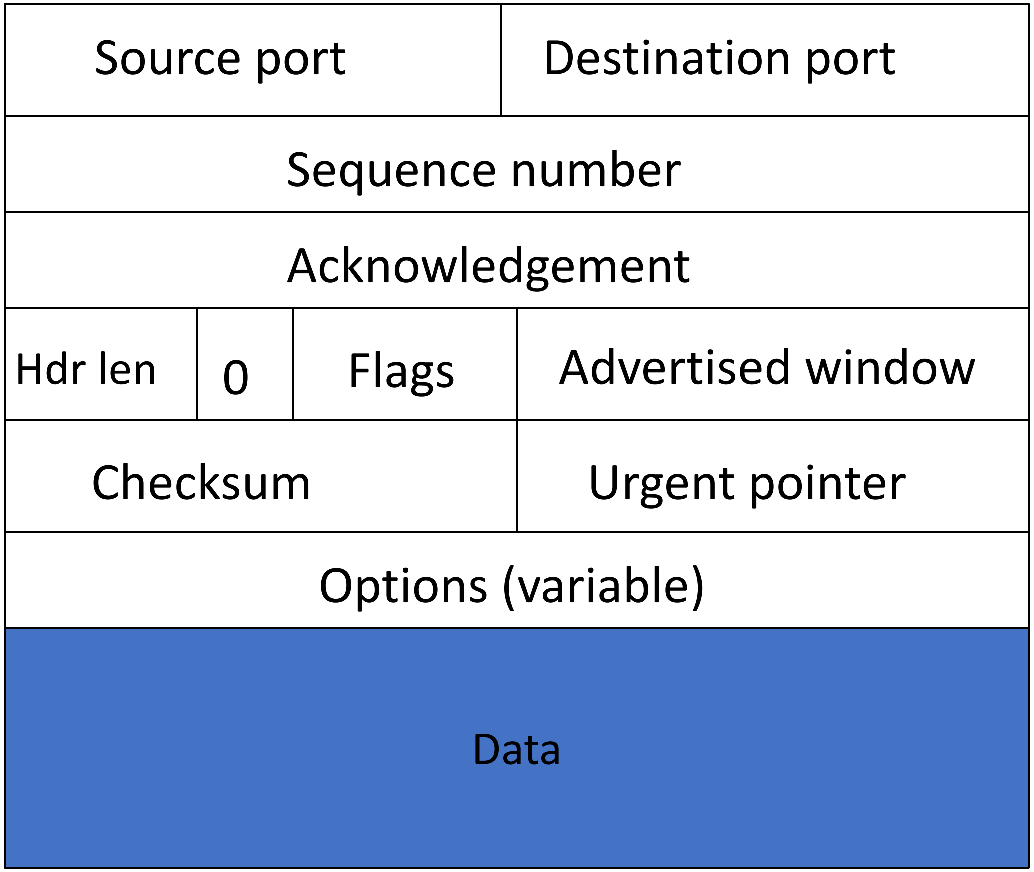
- Communication abstraction:
    - **Reliable**
    - **Ordered**
    - **Point-to-point**
    - **Byte-stream**
- Protocol implemented **entirely at the ends**
    - Assumes unreliable, non-sequenced delivery
    - **Fate sharing**
        - Can someone remind me what this means?

TCP Reliability Mechanism



TCP Header

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

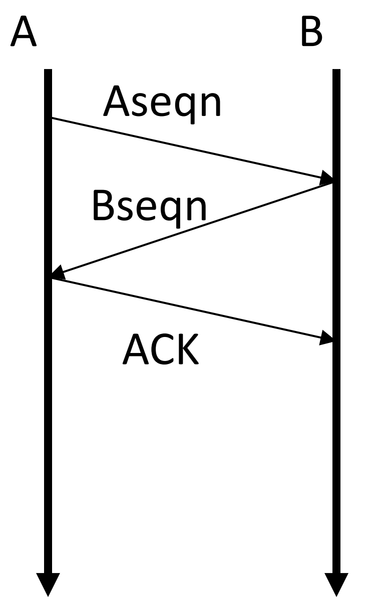


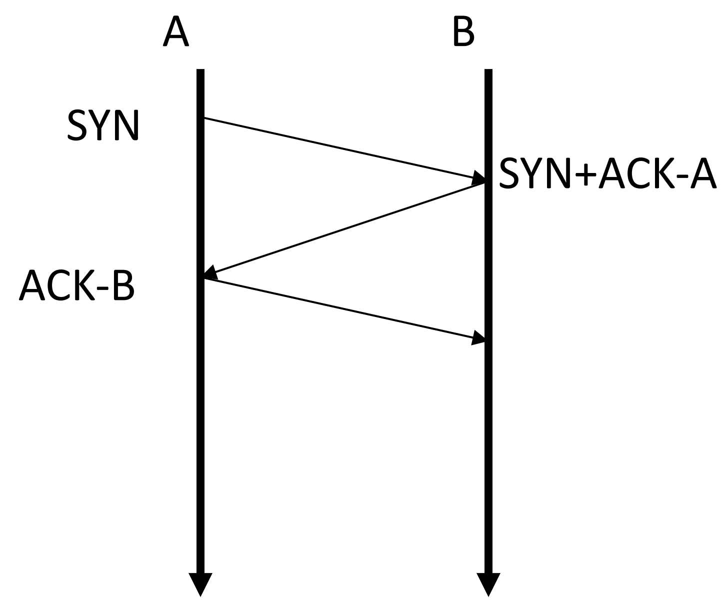
TCP Mechanisms

- 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





Can you explain why we need three packets? Why isn't two enough?

TCP Sequence Numbers

- Each direction of a TCP connection assigns a **progressive sequence number** (in bytes) to each segment

- **Initial sequence number** (ISN) selection is performed separately by each peer

ISN and Quiet Time

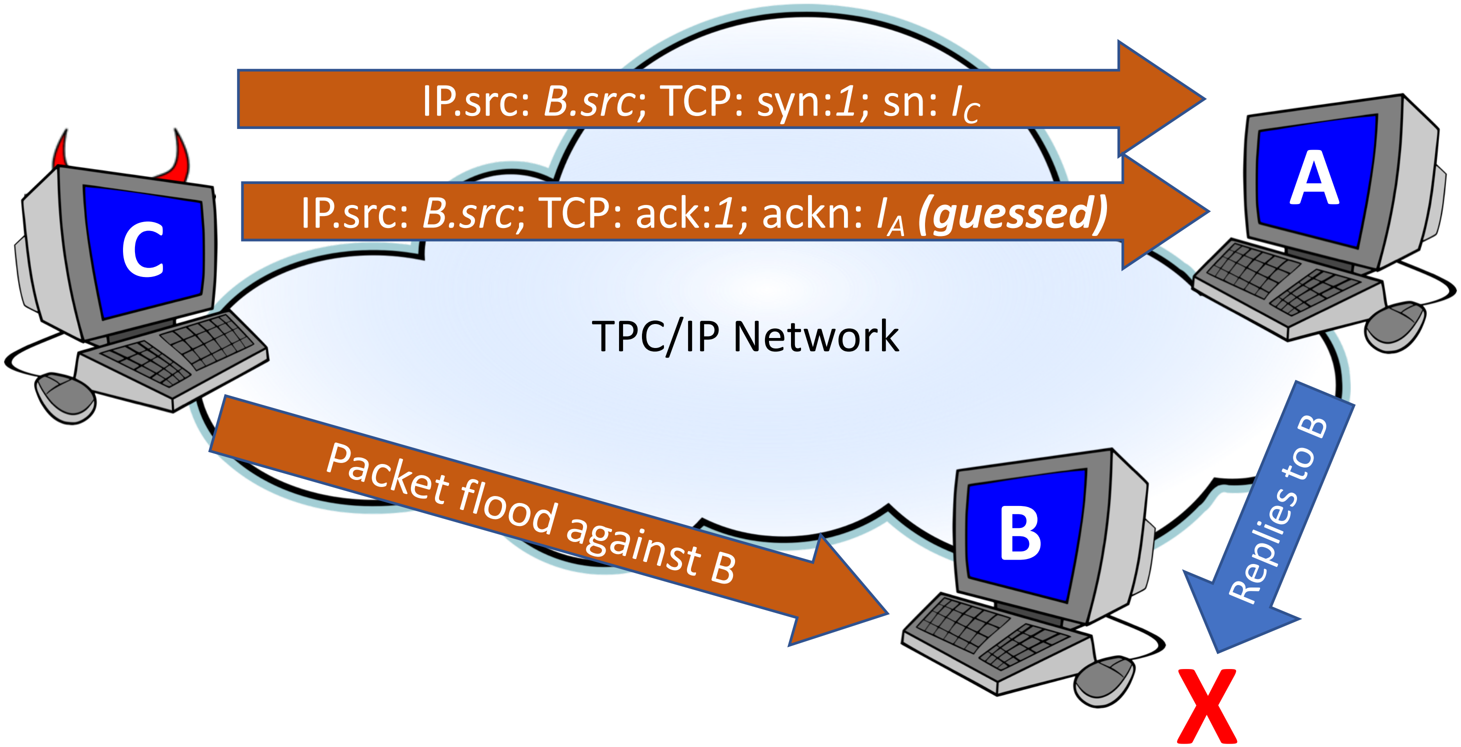
- Assume **upper bound on segment lifetime** (MSL)
  - In TCP, this is 2 minutes
- Upon startup, **cannot assign sequence numbers** for MSL seconds
- Can still have sequence number overlap (**wraparound**)
  - If sequence number space not large enough for high-bandwidth connections
  - Sequence numbers are 32-bit long: must send >= 4GB data in <= 2 minutes for wraparound

A bit of history…

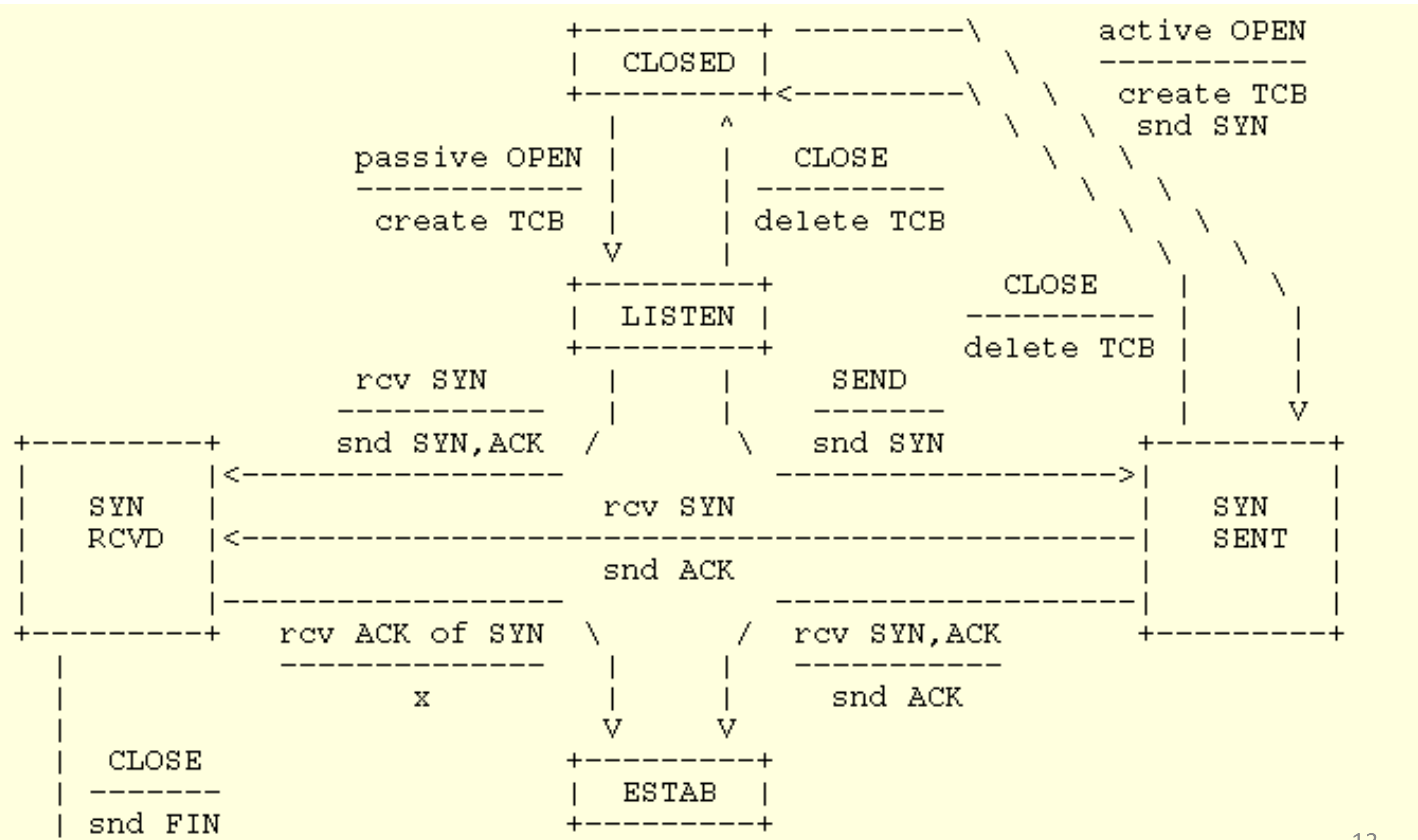
- TCP randomize the ISN: why?
- Reason 1: to **prevent confusion** w/o respecting the quiet time
- Reason 2: to **prevent TCP hijacking**

TCP Hijacking in a nutshell

- **Context:**
  - Host A trusts connections from host B
  - Host C wants to impersonate B to run commands on A



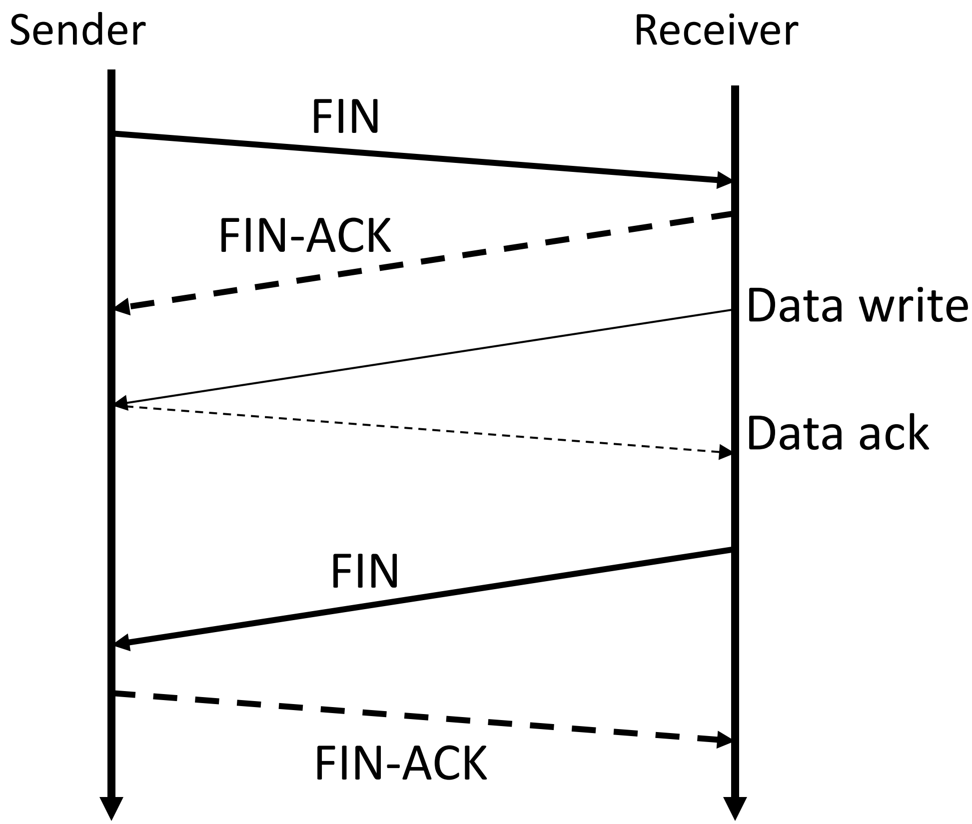
Connection Setup



Connection Tear-down

- **Normal termination**
  - Allow unilateral close
- **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  |
                             +---------+                   +---------+
```

Round-trip Time Estimation

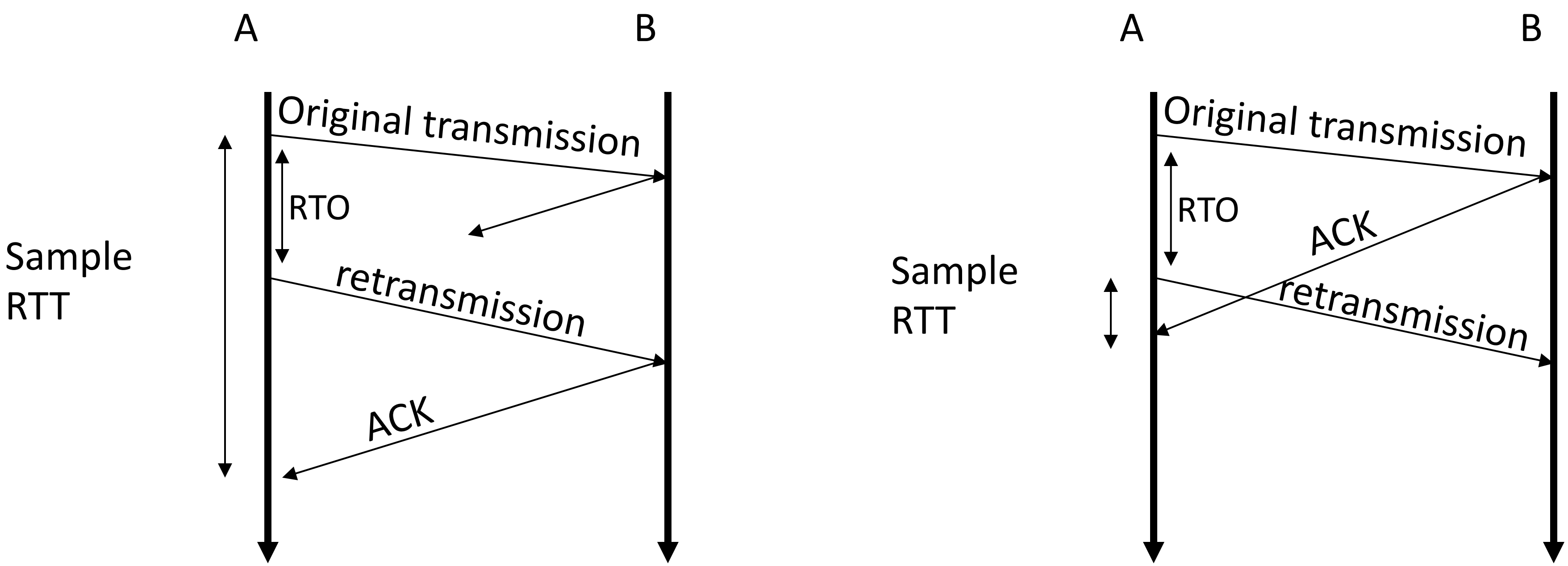
- **I suspect a packet may have been lost – now what?**
- **Wait at least one RTT before retransmitting**
- Importance of accurate RTT estimators?
  - Low RTT -> unneeded retransmissions
  - High RTT -> poor throughput
- RTT estimator **must adapt** to change in RTT
  - But not too fast, or too slow!

Early 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 β = 2
- Every time timer expires, RTO exponentially backed-off
  - Typically RTO is not allowed to exceed a maximum threshold (Linux: TCP_RTO_MAX; default 120s)

Retransmission Ambiguity



- Suppose RTO expires, segment is retransmitted, and then an ACK is received
- Is the ACK referring to the original transmission, or to the retransmission?

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**

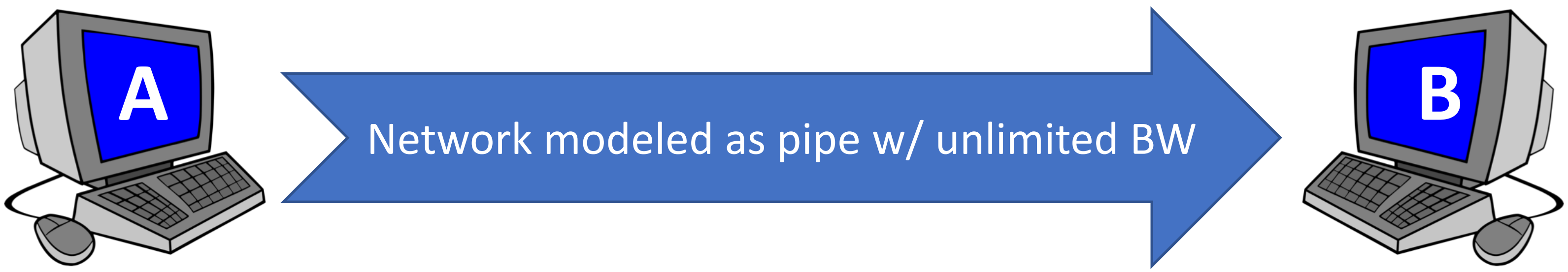
Jacobson's Retransmission Timeout Estimator

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

Up next: flow control and congestion control

- Can you explain the difference between them?

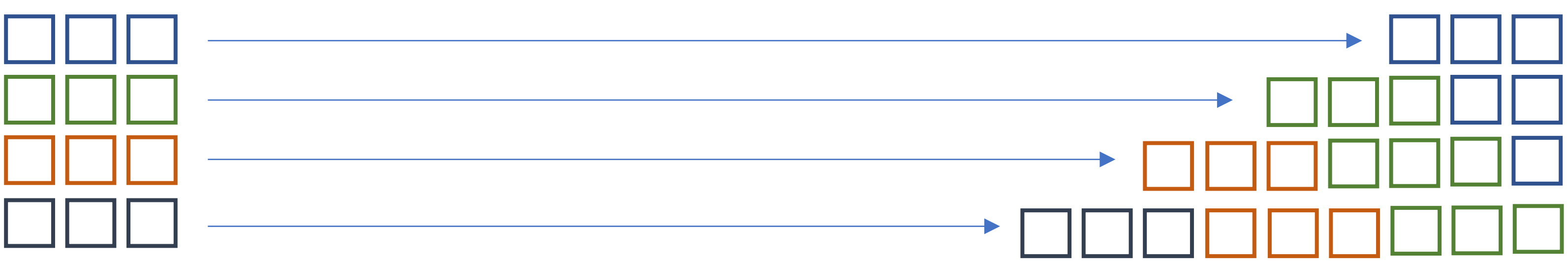
What's the purpose of flow control?



Can generate data at 3 packet/s

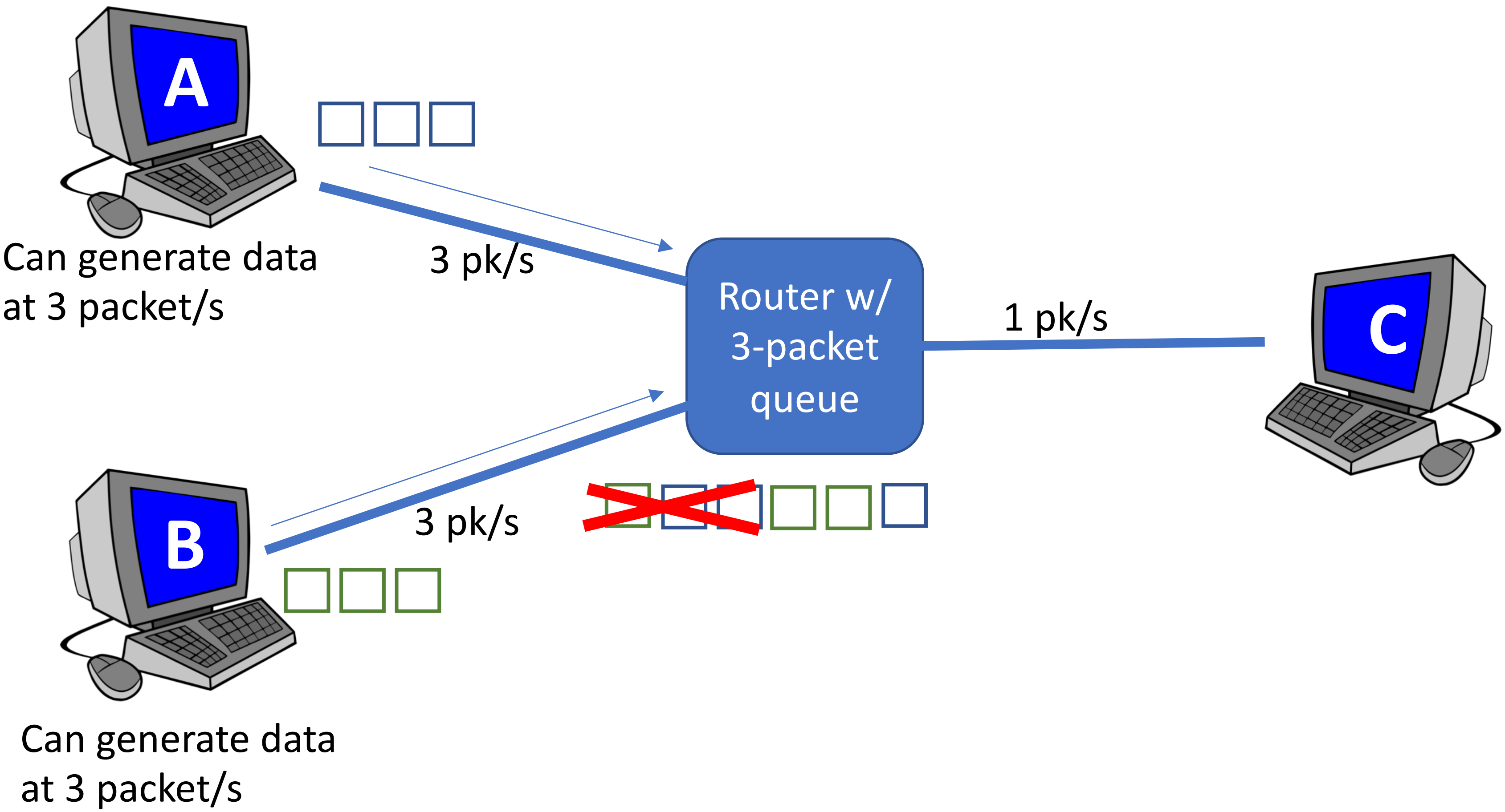
Can consume data at 1 packet/s

What happens w/o flow control?



What's the purpose of congestion control?

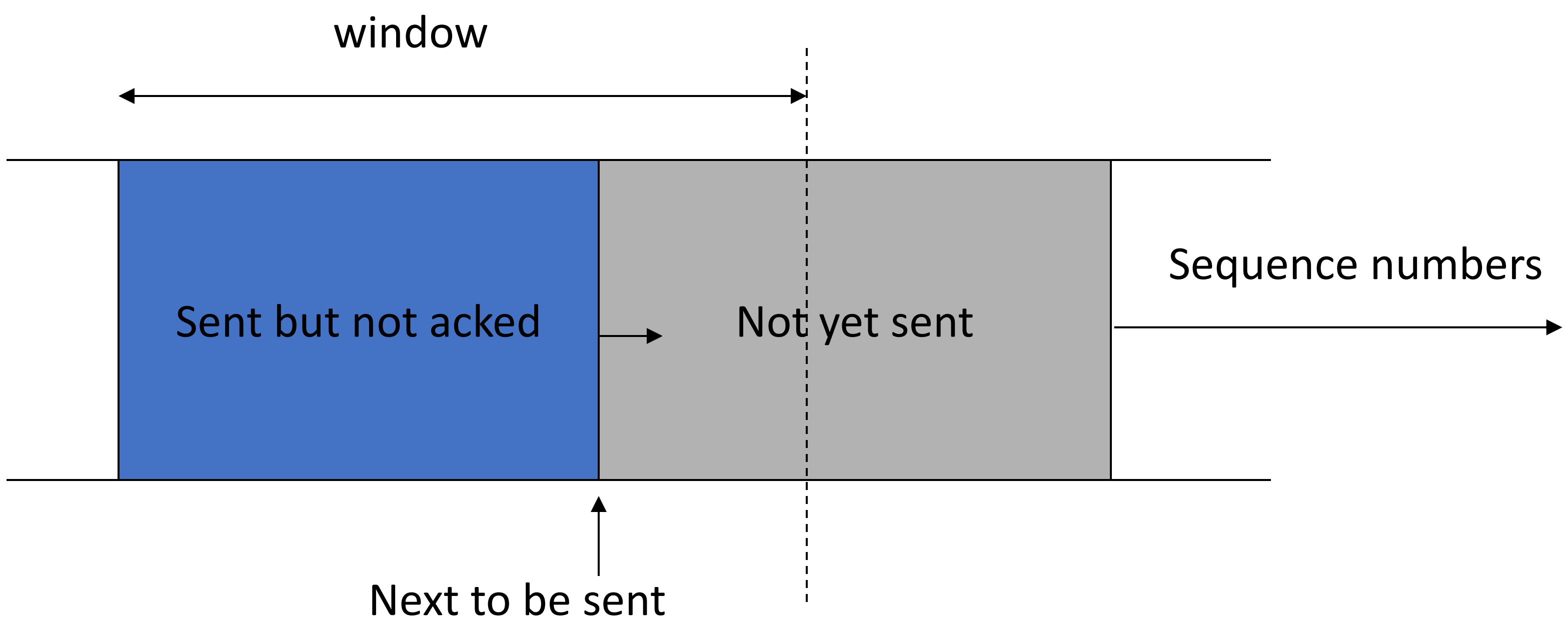
What happens w/o congestion control?



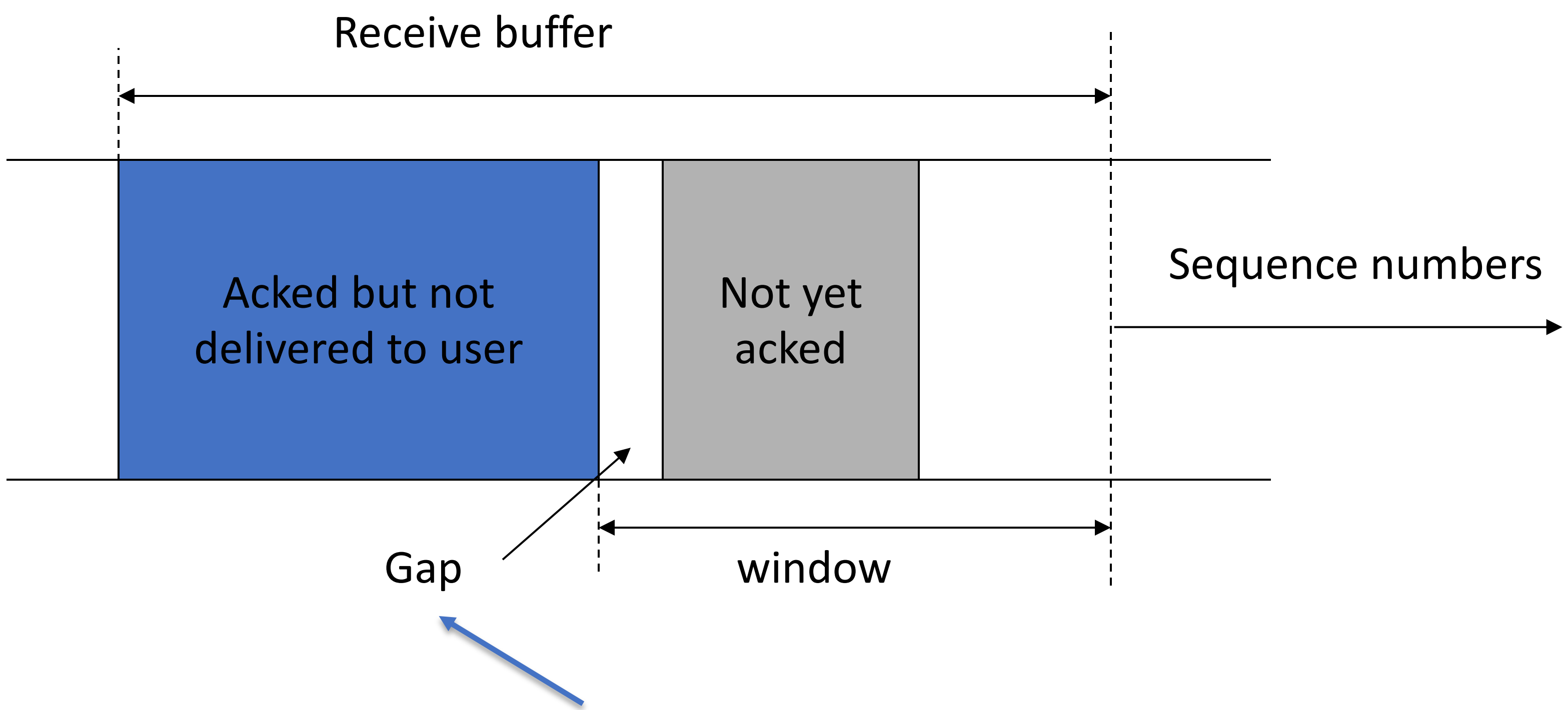
Flow Control

- 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

Window Flow Control: Send



Window Flow Control: Receive



"The receiving TCP may be forced to hold on to data with larger sequence numbers before giving it to an application until a missing lower-sequence-numbered segment ("a hole") is filled in" (Fall & Stevens, TCP/IP illustrated vol. 1)

Window Advancement Issues

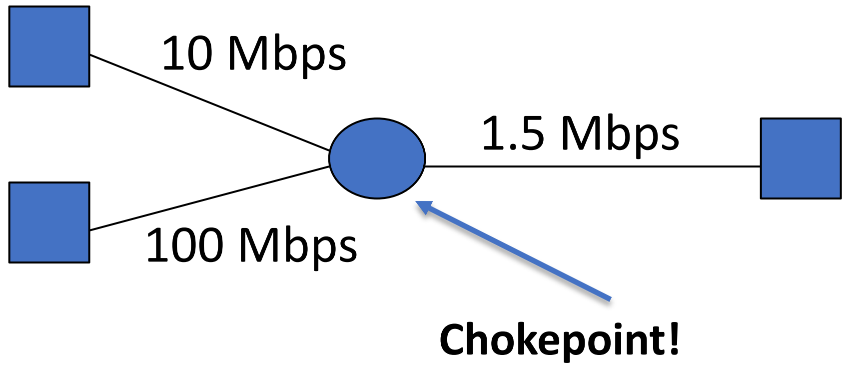
- **Advancing a full window**
  - When the receive window fills up, how do things get started again?
  - Sender sends periodic probe while receive win is 0
- **Silly window syndrome**
  - Fast sender, slow receiver
  - Delayed acks at receiver help, but not a full solution

Nagle's algorithm

- In the Internet's early days, a lot of traffic consisted of **interactive terminal sessions**
- Sending **one packet per keypress** generates a lot of overhead!
  - TCP/IP headers + data: ~88 bytes for 1 byte of data
- In most cases, **multiple keypresses can be aggregated in a single packet** with no perceivable degradation of responsiveness
- Nagle's algorithm:
  - **Delay sending if un-acked data in flight**
  - Overwrite with TCP-NODELAY option

TCP Congestion Control

Congestion

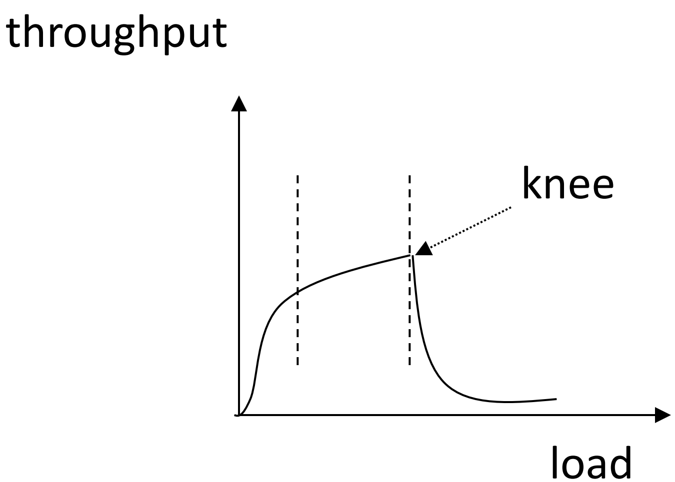


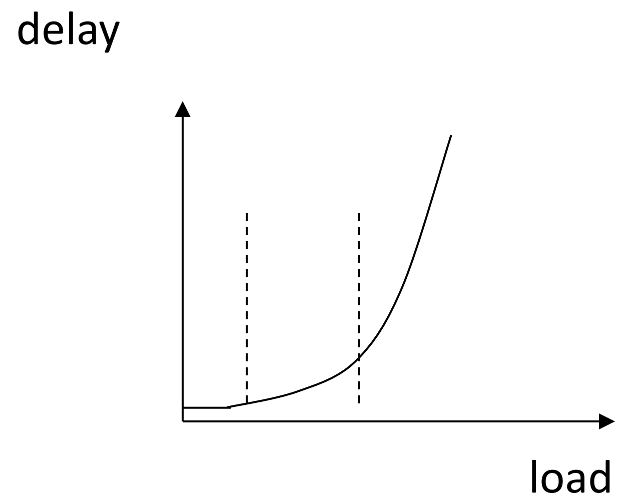
- Caused by **fast links feeding into slow link**
- **Severe congestion may lead to network collapse**
  - Flows send full windows, but progress is very slow
  - Most packets in the network are retransmissions
- Other causes of congestion collapse
  - **Non-feedback controlled sources**

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
    - **Example:** (from Jacobson & Karels): in October ’86, the link between LBL and UC-Berkeley experienced a **~800X** throughput decrease

Congestion Response





Congestion Control Design

- **Avoidance** or **control**? Need both!
  - **Avoidance keeps system at knee of curve**
  - But, to do that, need routers to send accurate signals (some feedback)
  - **Control is necessary when things go bad**
- 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?

TCP Congestion Management

- 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

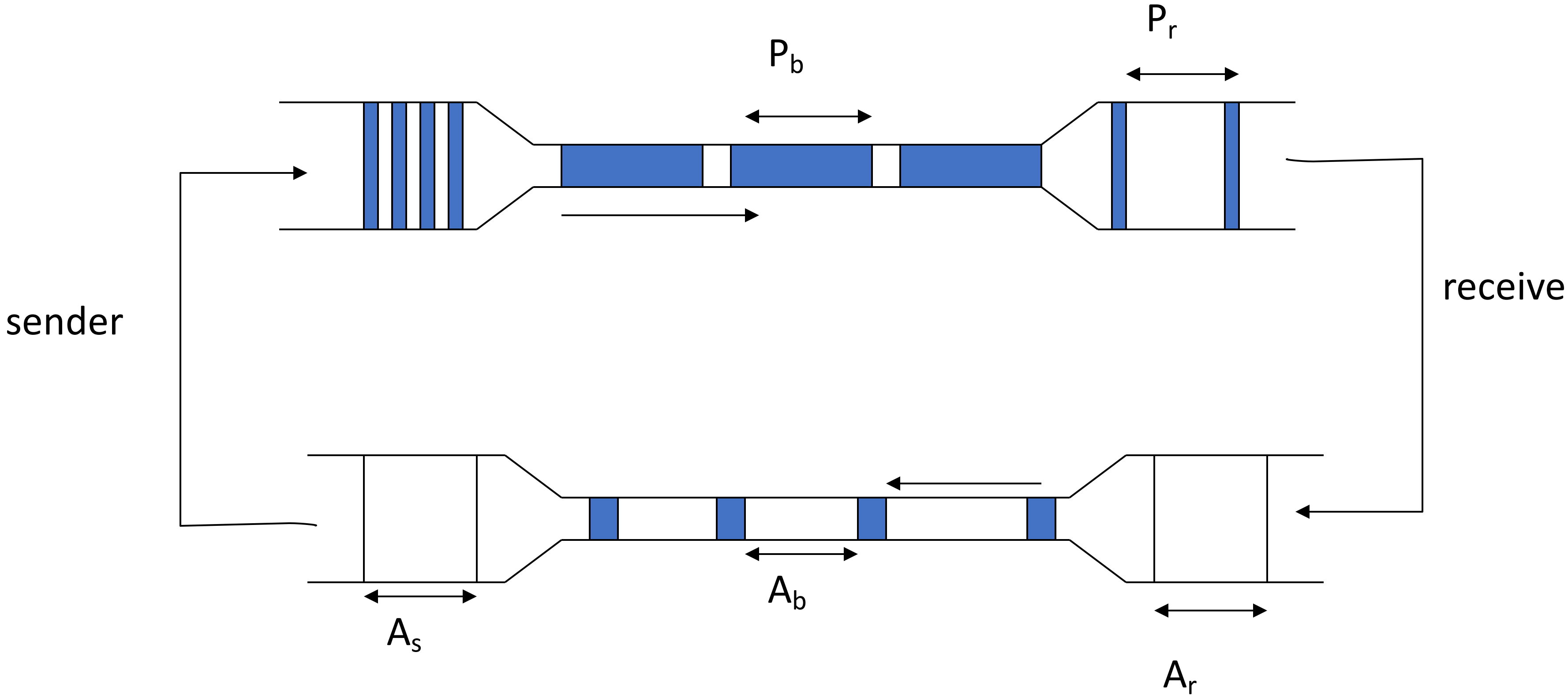
TCP Congestion Control Basics

- Keep a **congestion window**, cwnd
  - Denotes **how much data** 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**

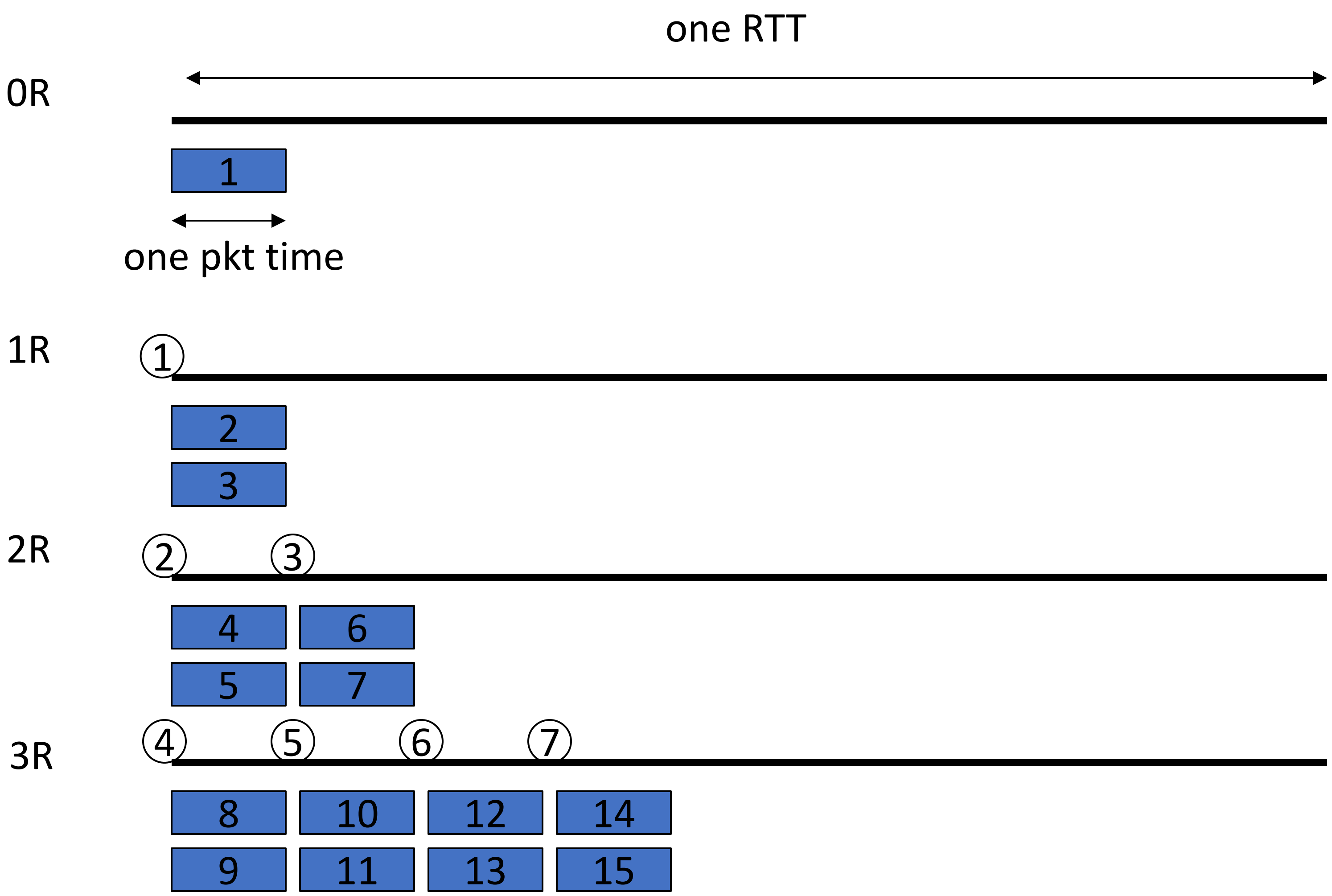
TCP is Self-clocking



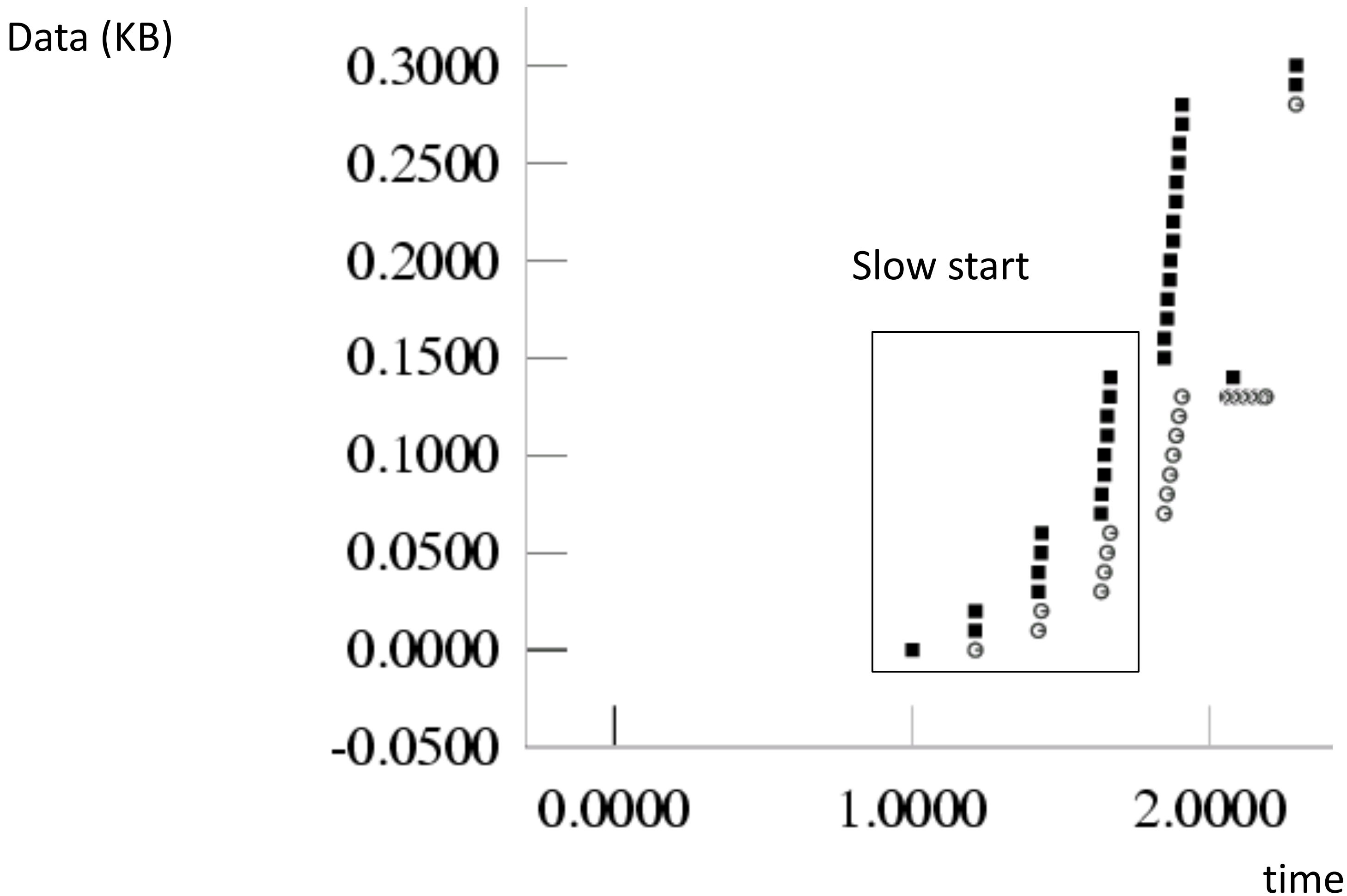
Slow Start

- 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
    - **Multiplicative increase!**
  - Can **overshoot window and cause packet loss**

Slow Start Example

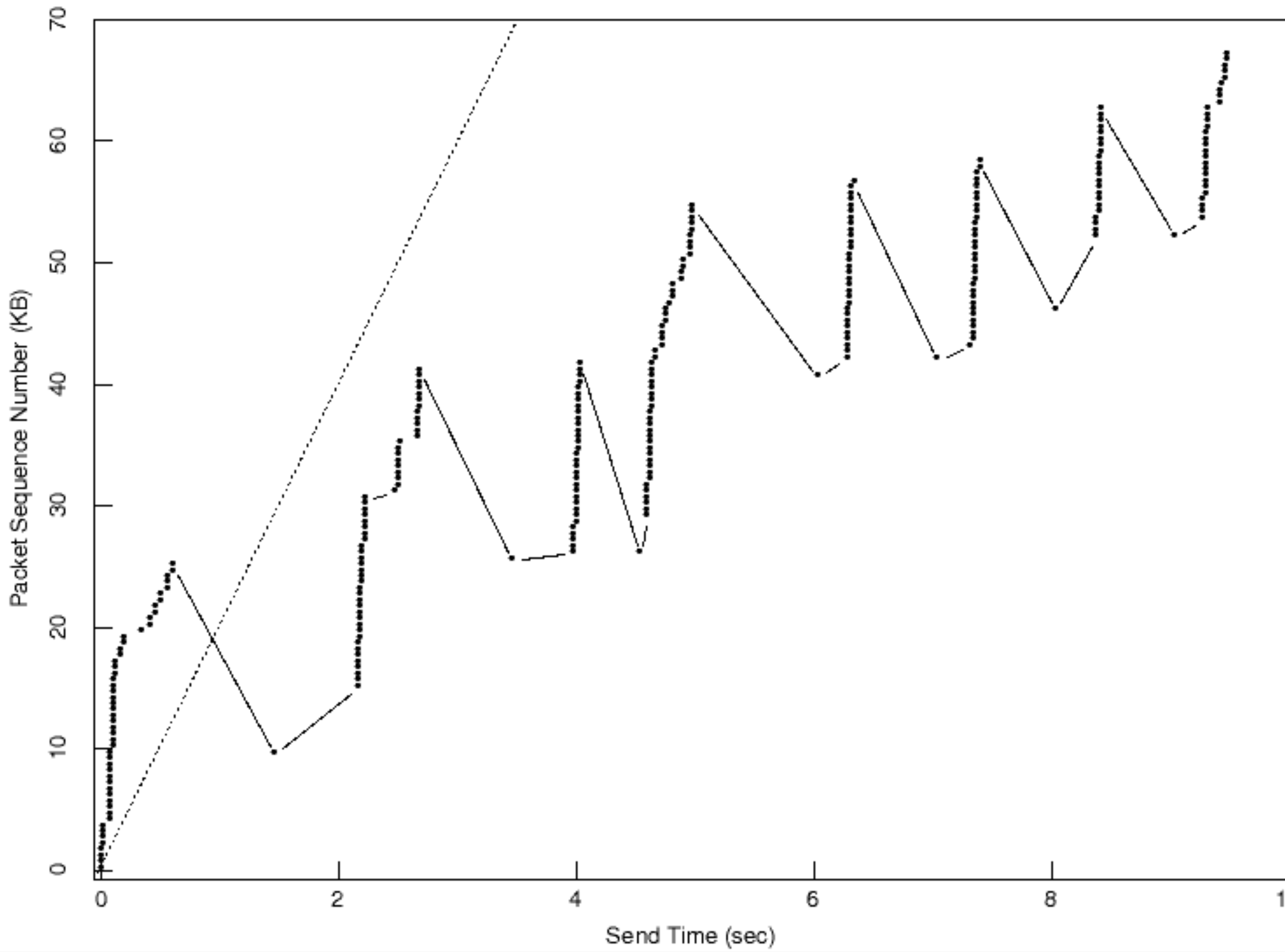


Slow Start Sequence Plot

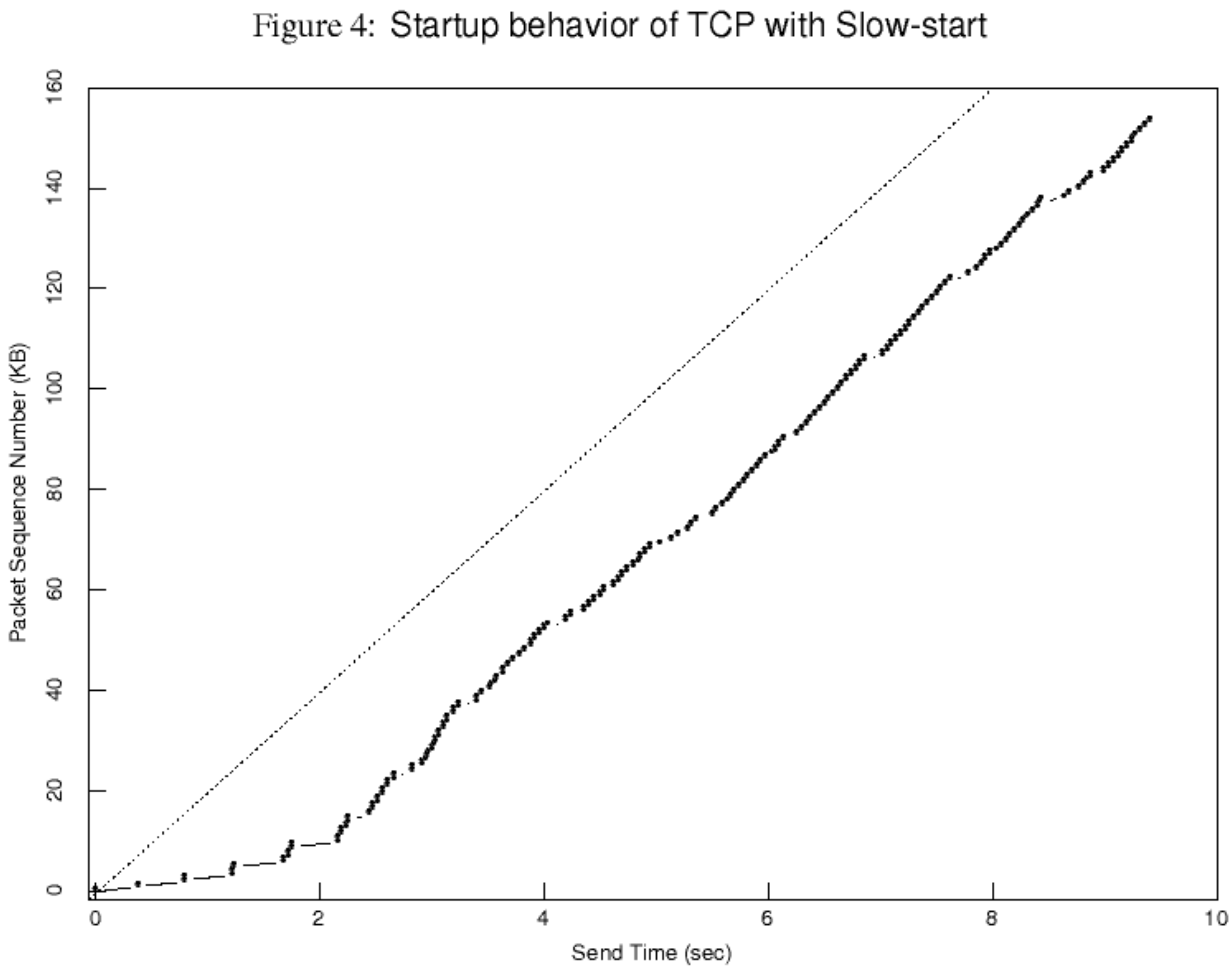


Jacobson, Figure 3: No Slow Start

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

Rationale for halving the congestion window

- If loss happened during **slow start**:
  - CWND doubled at every RTT
  - If we got to a window size W, it is because W/2 worked
- If loss happened during **congestion avoidance** (steady state):
  - Probably this means that we are competing against a new flow
  - Worst-case scenario: previously I was the only flow using the path, now there is another one (i.e. the capacity was cut in half)

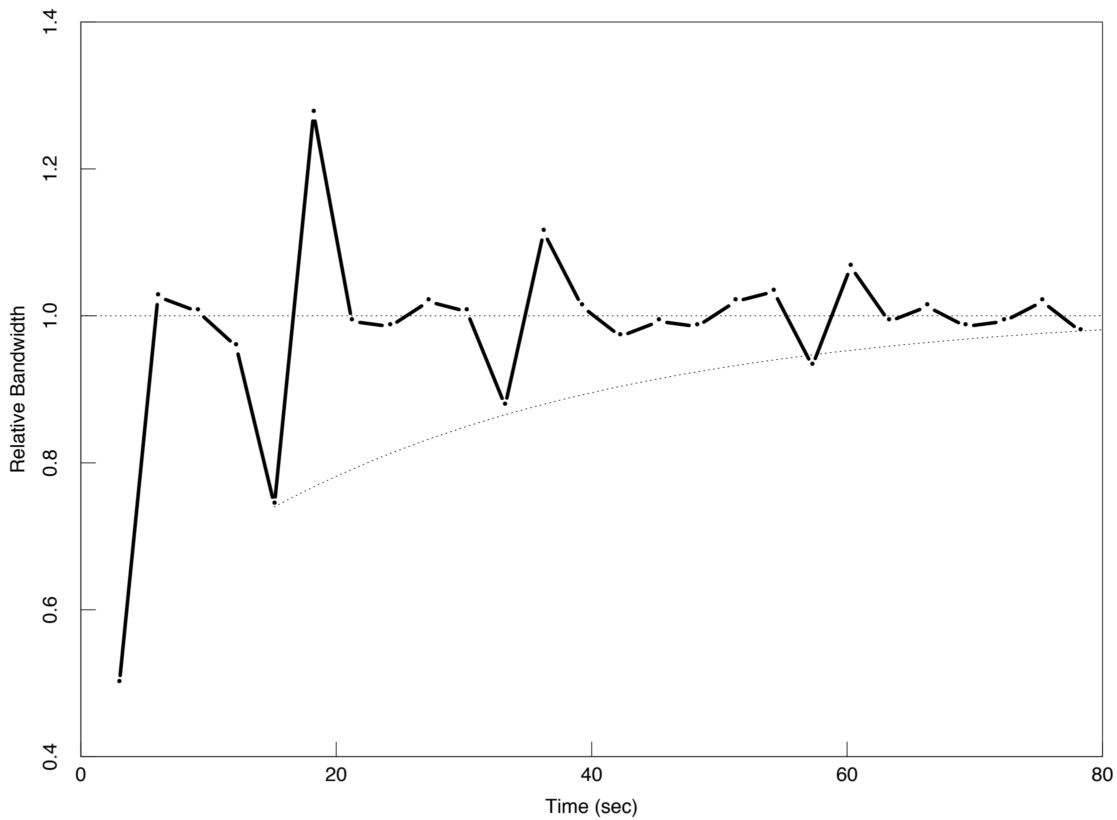
How does this all work together?

- On a timeout, half the current window size is recorded in *ssthresh* (this is the multiplicative decrease part of the congestion avoidance algorithm), then *cwnd* is set to 1 packet (this initiates slow-start). When new data is acked, the sender does :

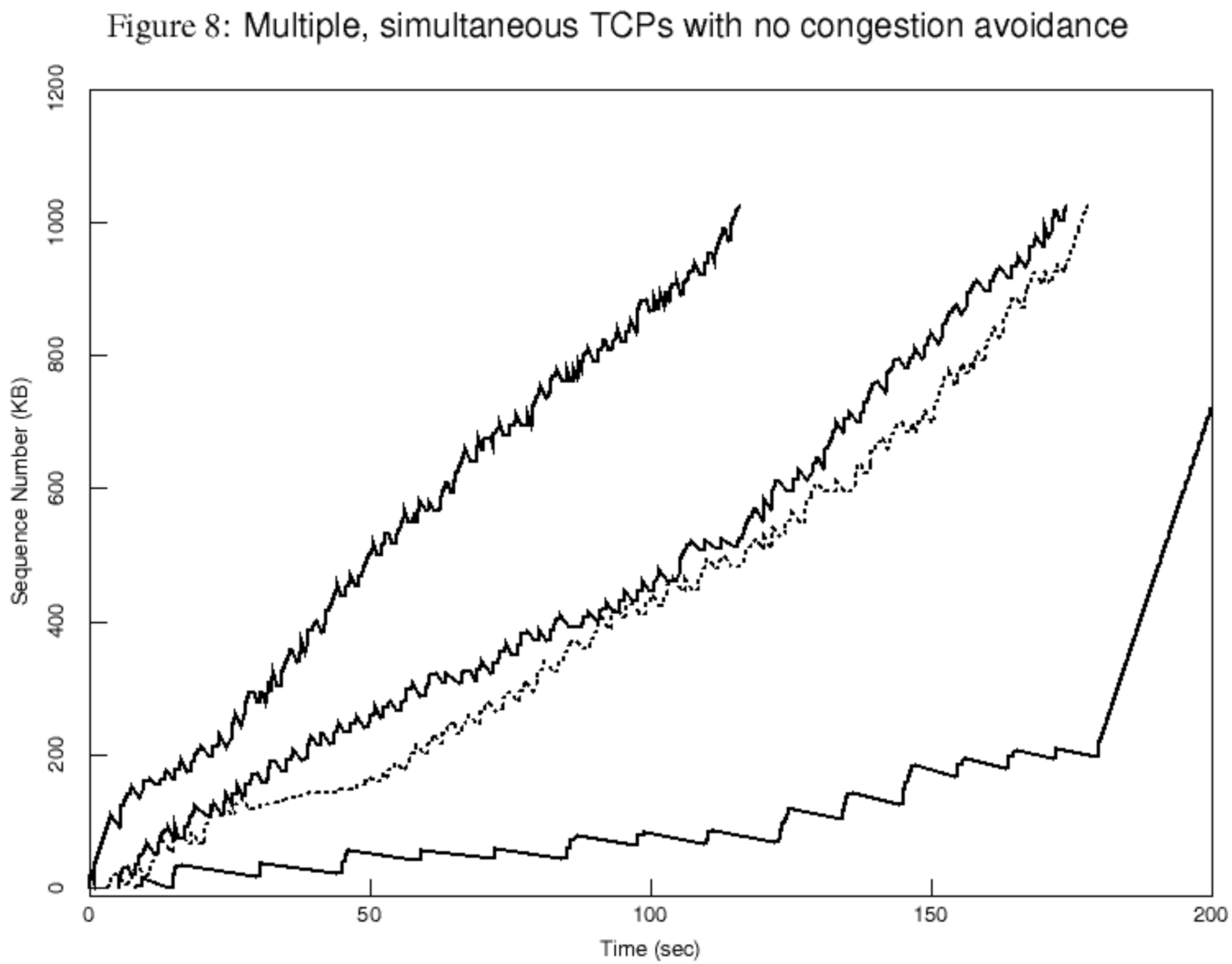
```
if (cwnd < ssthresh)
        /* if we're still doing slow-start
         * open window exponentially */
        cwnd += 1;  Why?
else
        /* otherwise do Congestion
         * Avoidance increment-by-1 */
        cwnd += 1/cwnd;
```

Jacobson, Figure 12: window adjustment details

Figure 12: Window adjustment detail

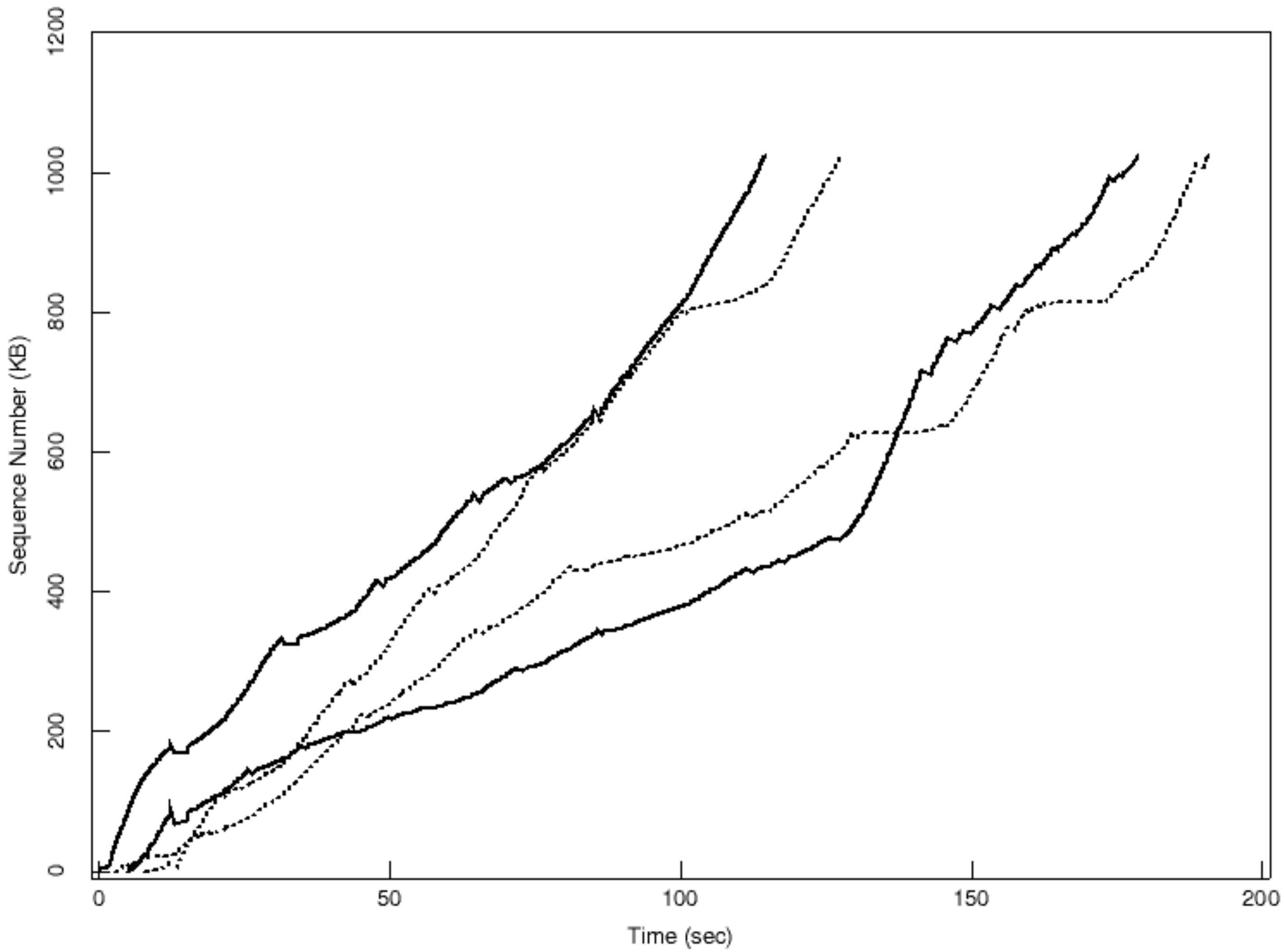


Jacobson, Figure 8: 4x no Congestion avoidance



Jacobson, Figure 9: 4 TCPs with Congestion Avoidance

Figure 9: Multiple, simultaneous TCPs with congestion avoidance



Some TCP optimizations and problems

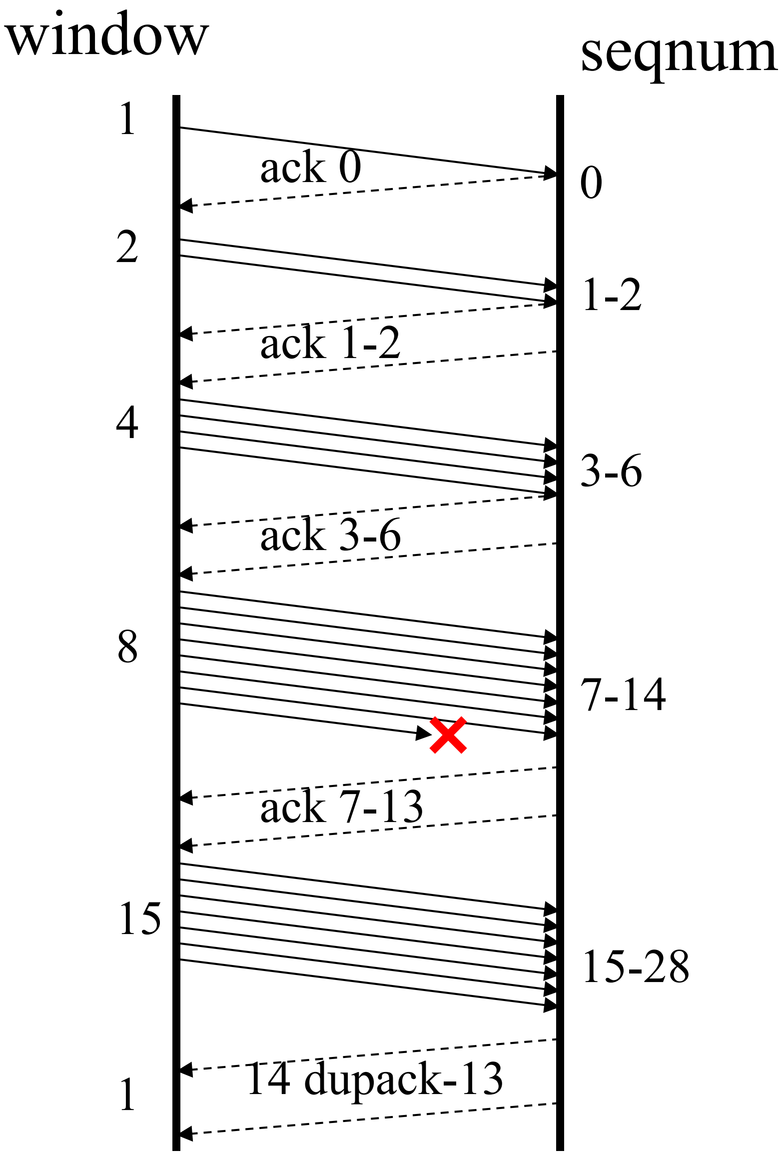
Impact of Timeouts

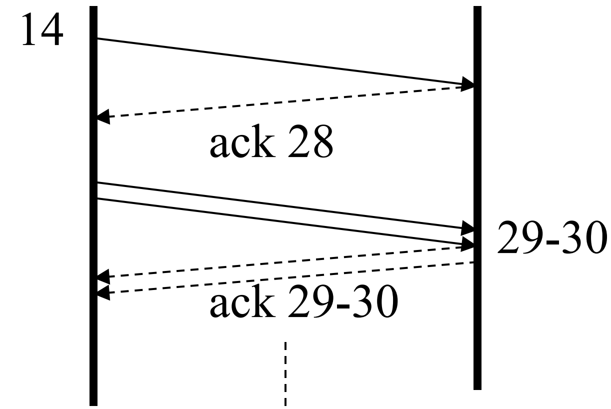
- **Timeouts** can cause sender to
  - Slow start
  - Retransmit a possibly large portion of the window
- Bad for lossy high bandwidth-delay paths
- Can leverage duplicate acks to:
  - Avoid waiting for a timeout on loss (**fast retransmit**)
  - Advance cwnd more aggressively (**fast recovery**)

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 - 1 Drop





Actions after dupacks for pkt 13:

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 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
- Optimistically skips slow-start

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

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

- (Historical) 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
    - Receiver acknowledging sub-segment sequences when in congestion avoidance
  - DupACK spoofing: send multiple dup acks to inflate window
    - Works against fast recovery
  - Optimistic Acking: send acks for packets you didn't receive yet – emulates shorter RTT
    - Wrong RTT causes retransmissions
- Above problems are implementation dependent

SYN Attacks

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

  $\Rightarrow$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