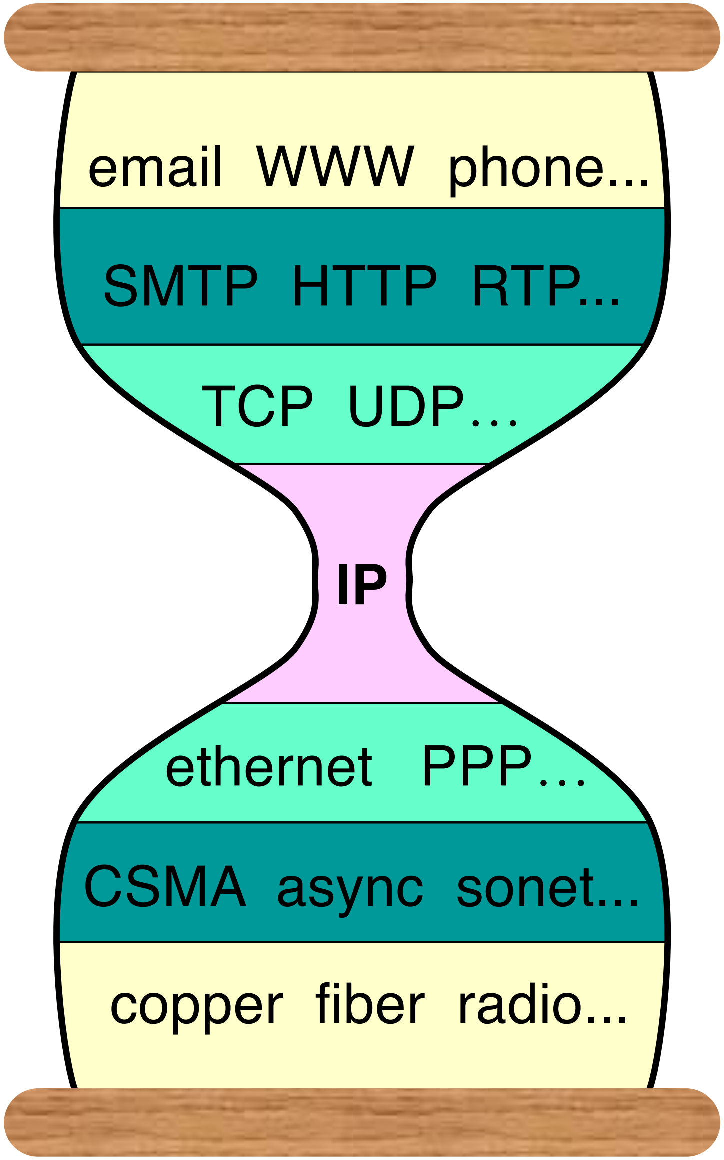
CSC 525: Computer Networks

Where we are


email WWW phone...
SMTP HTTP RTP...
TCP UDP…
IP
ethernet PPP…
CSMA async sonet...
copper fiber radio...

Duties of Transport Layer

- Demultiplex:
  - IP gets packets to a host, i.e., dest address.
    - Its header points to a transport protocol
  - Transport gets packets to the app, i.e., dest port.
    - Identified by a port number on the host.
    - Well-known ports vs. ephemeral ports
- Provide different services for certain apps:
  - Different apps may want different services on top of IP's best-effort delivery
    - UDP: no added feature; just whatever datagrams received.
    - TCP: reliable, in-order, byte-stream, with flow and congestion control.
    - There're many other transport protocols in between.

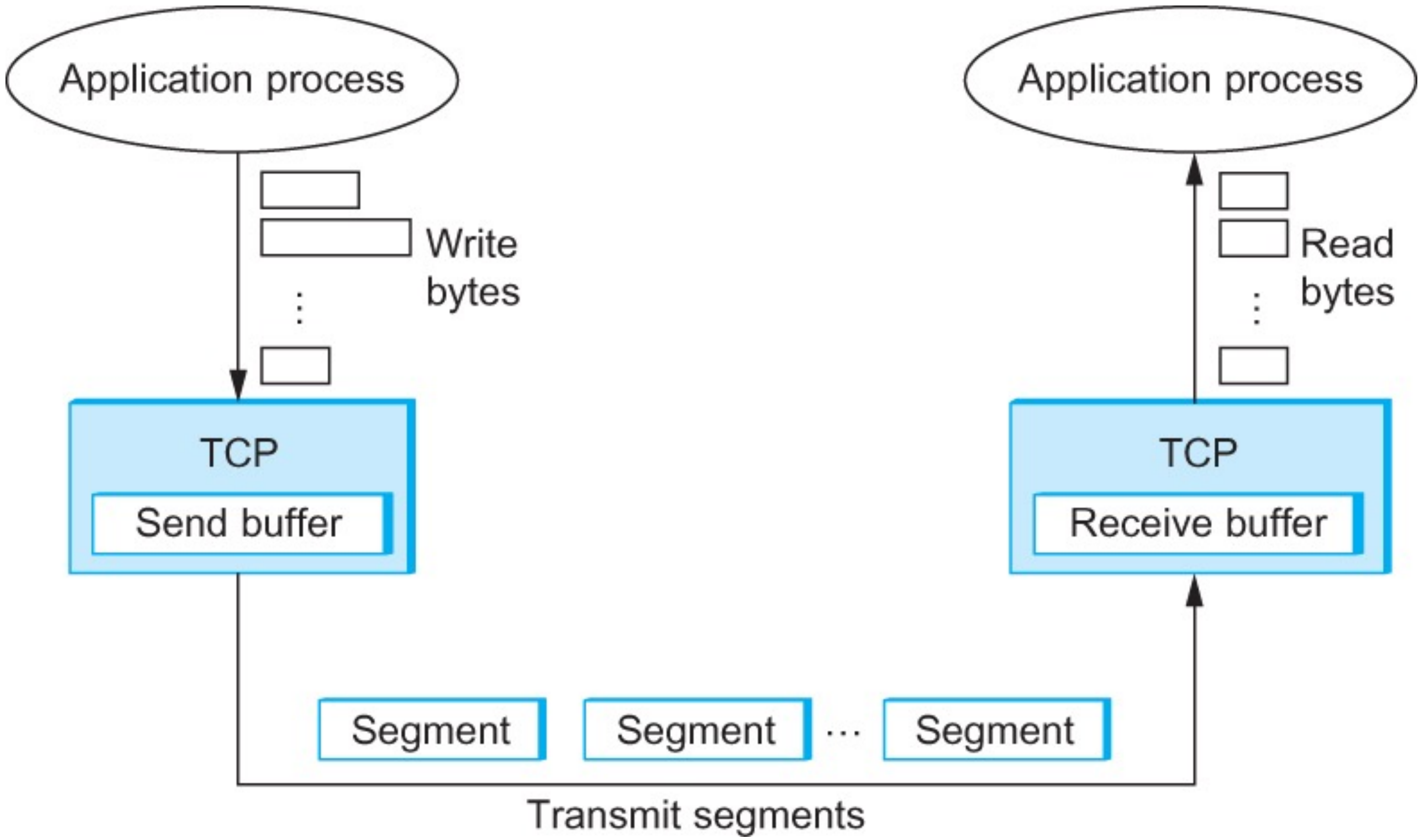
TCP Overview

- End-to-End unicast: one sender, one receiver
- Connection-oriented:
  - exchange control msgs first to initialize sender & receiver states
- Full duplex data delivery:
  - bi-directional data flow over the same connection
- **Reliable, in-order, byte-steam delivery**
  - no “message boundaries”
  - sender & receiver must buffer data before delivering to apps.
- Flow controlled
  - Prevent sender from overrunning receiver buffer.
- Congestion controlled
  - Prevent sender from overrunning router buffers.

TCP Segment

- TCP provides a byte-stream service to applications
  - Applications read from and write to a byte-stream.
- But TCP itself does not transmit individual bytes over the network.
  - On the source host TCP buffers enough bytes from the sending process to fill a reasonably sized packet.
  - On the destination host TCP empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure.
- The packets exchanged between TCP peers are called *segments*.

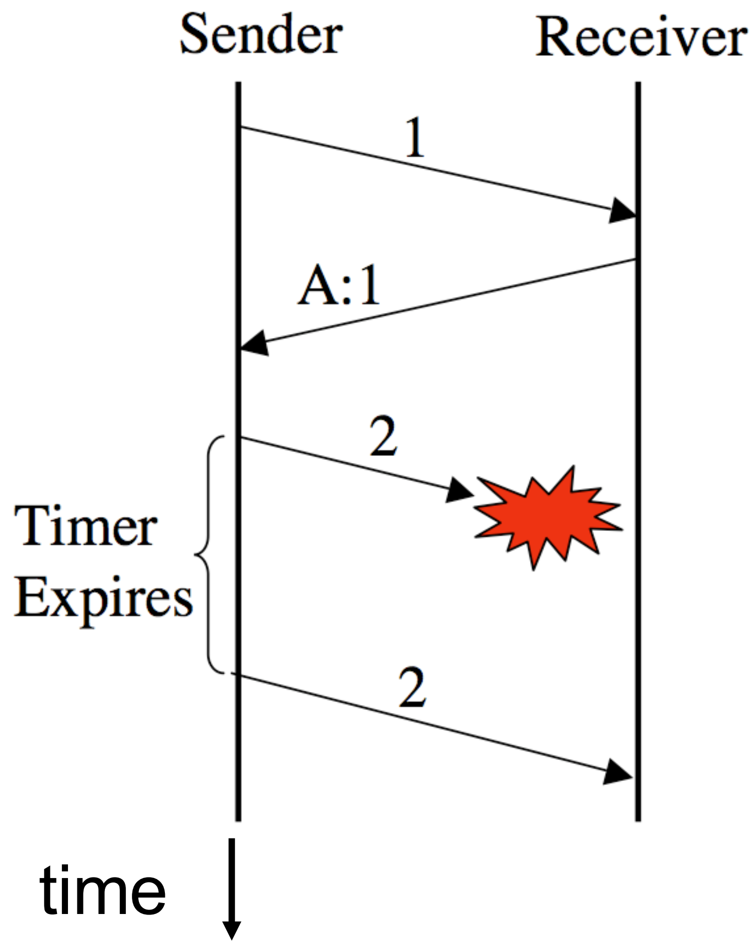
TCP Segment



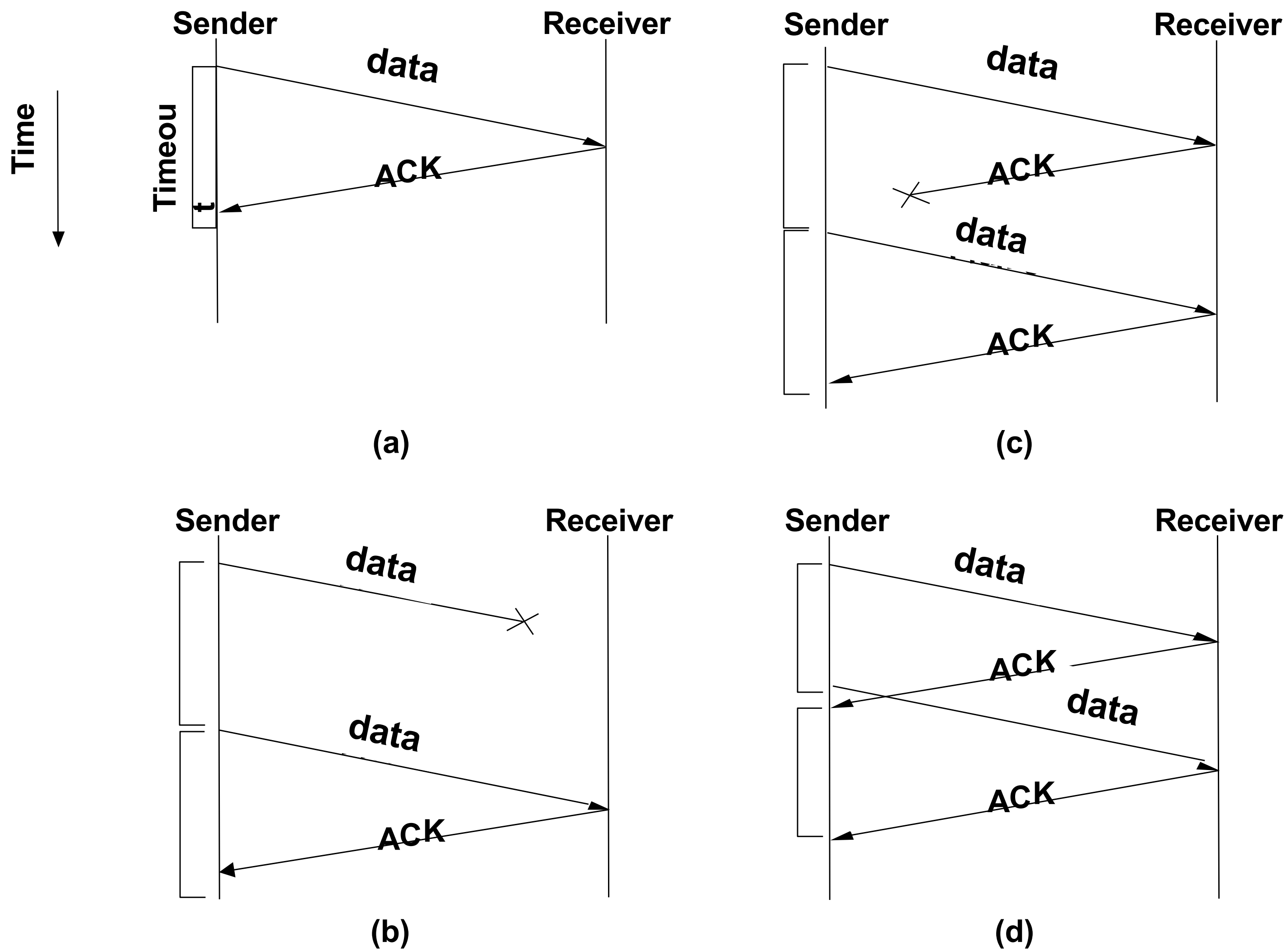
How TCP manages a byte stream.

Reliable Transmission

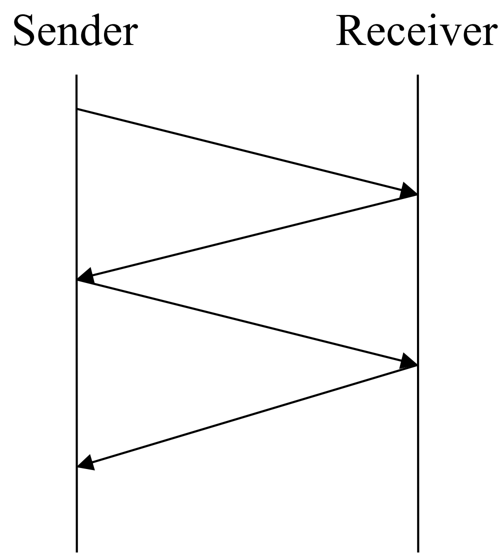
- How to achieve reliable, in-order delivery with controlled rate?
- The basic idea is called Automatic Repeat reQuest (ARQ)
  - Sender sets a timer and waits for ACK.
  - Receiver sends ACK.
  - If timed out, sender retransmits.



Different Scenarios



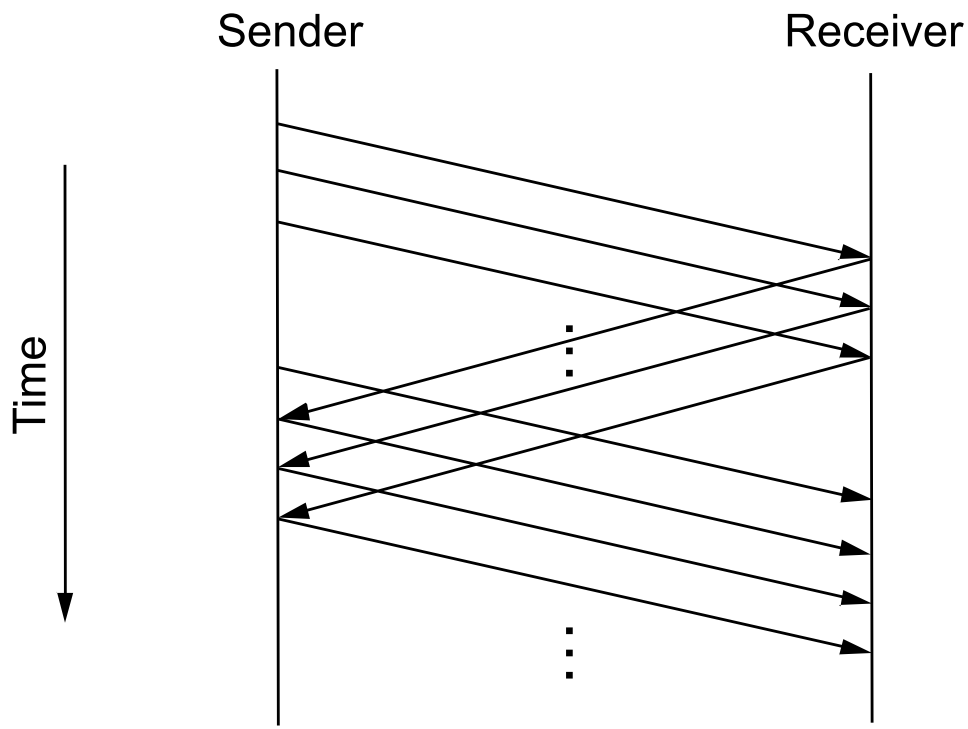
Stop-and-Wait



- At most one outstanding (un-acked) data segment at any time.
- Problem: the pipe is empty most of the time.
  - Throughput = one pkt size / RTT

Sliding Window

- Pipelining: send multiple outstanding segments (up to a limit), advance the window as the ACKs arrive.
- The max number of un-ACKed segments allowed is called ***window size***.



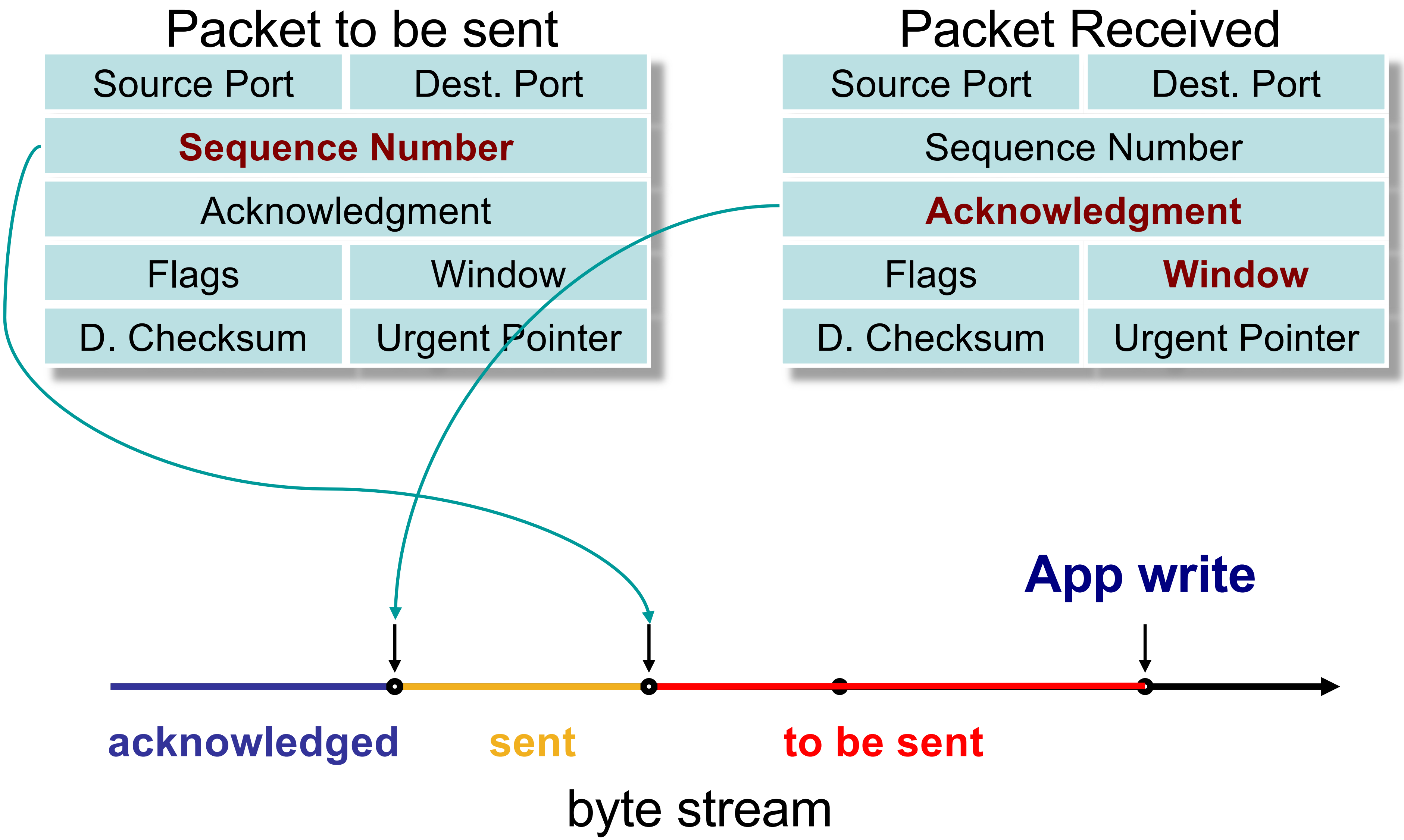
Acknowledgement

- Receiver sends acknowledgement back to sender.
  - Can be a separate packet, but more often piggybacked on data packet (and mark the A flag bit in TCP header).

- Cumulative ACK: ACK number X means
  - The receiver has got all data preceding X.
  - The next data the receiver is expecting is X.
  - E.g., if received 1, 2, 3, 5, 6, then ACK = 4.

- Other schemes exist, such as selective ACK.

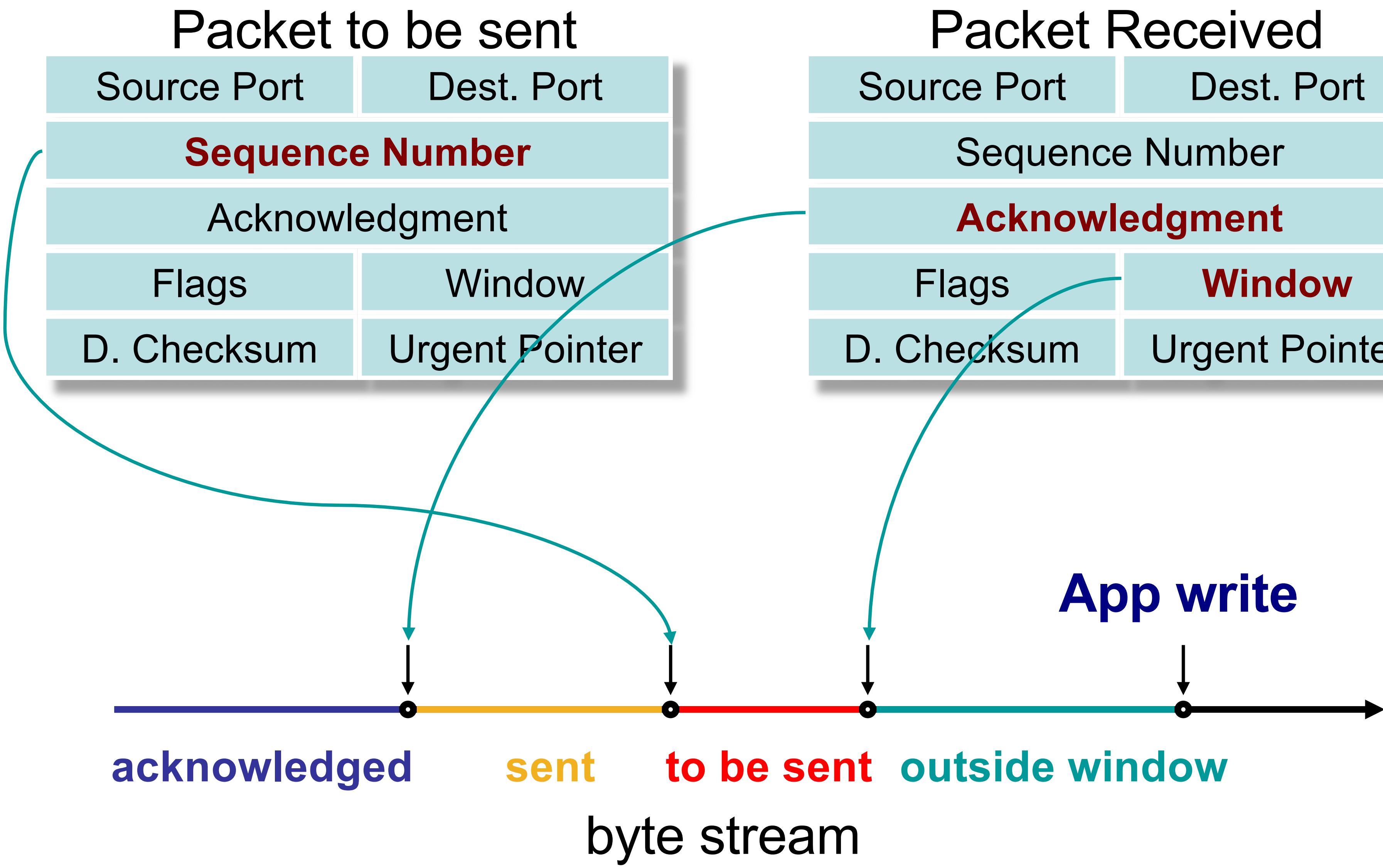
Seq# and Ack#


Packet to be sent
Source Port
Dest. Port
Sequence Number
Acknowledgment
Flags
Window
D. Checksum
Urgent Pointer
Packet Received
Source Port
Dest. Port
Sequence Number
Acknowledgment
Flags
Window
D. Checksum
Urgent Pointer
App write
acknowledged
sent
to be sent
byte stream

TCP Flow Control

- Make sure receiving end can keep up with incoming data.
- The receiver puts its *available* buffer size in the TCP header as *AdvertisedWindow*.
- The sender should not send more data than what AdvertisedWindow allows.

TCP Flow Control: Sender Side

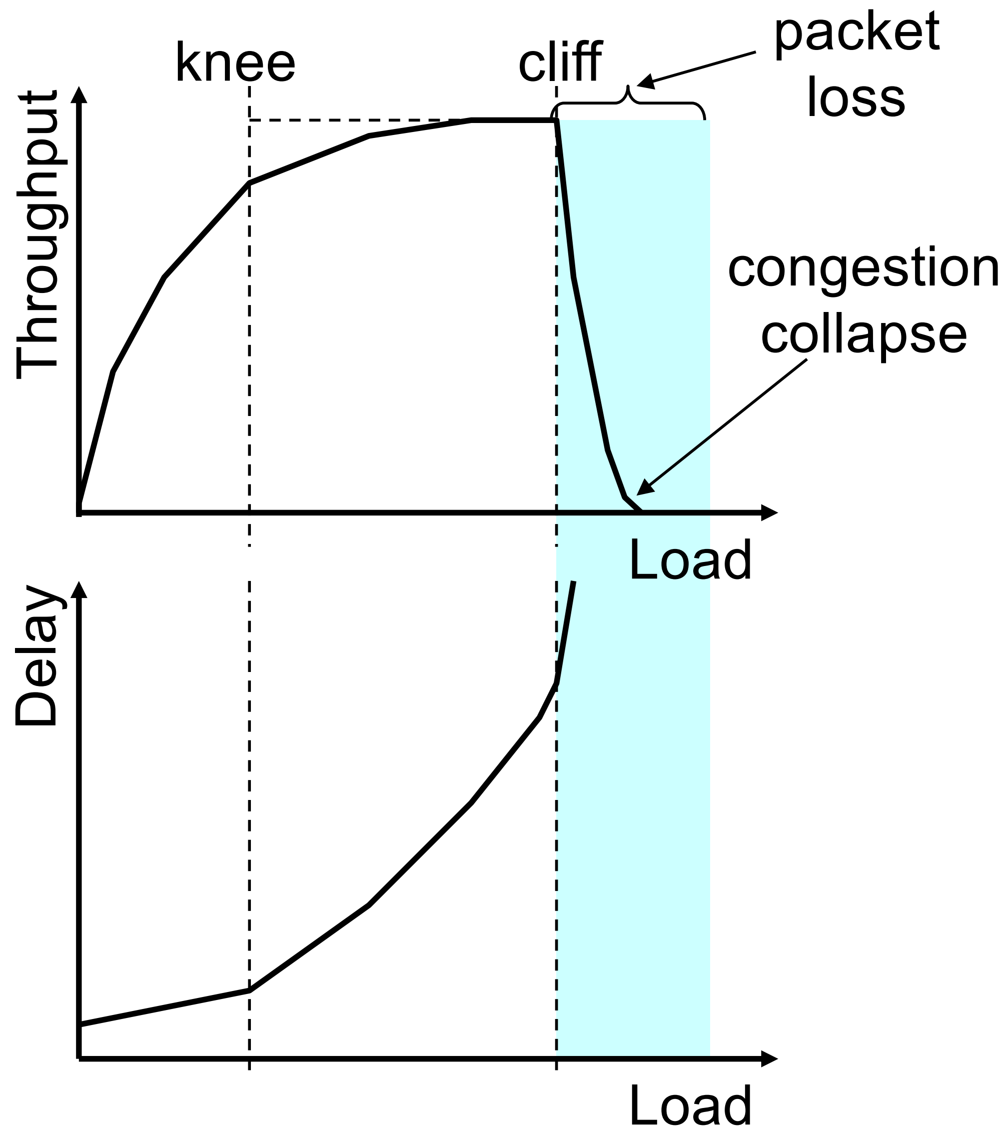


TCP Congestion Control

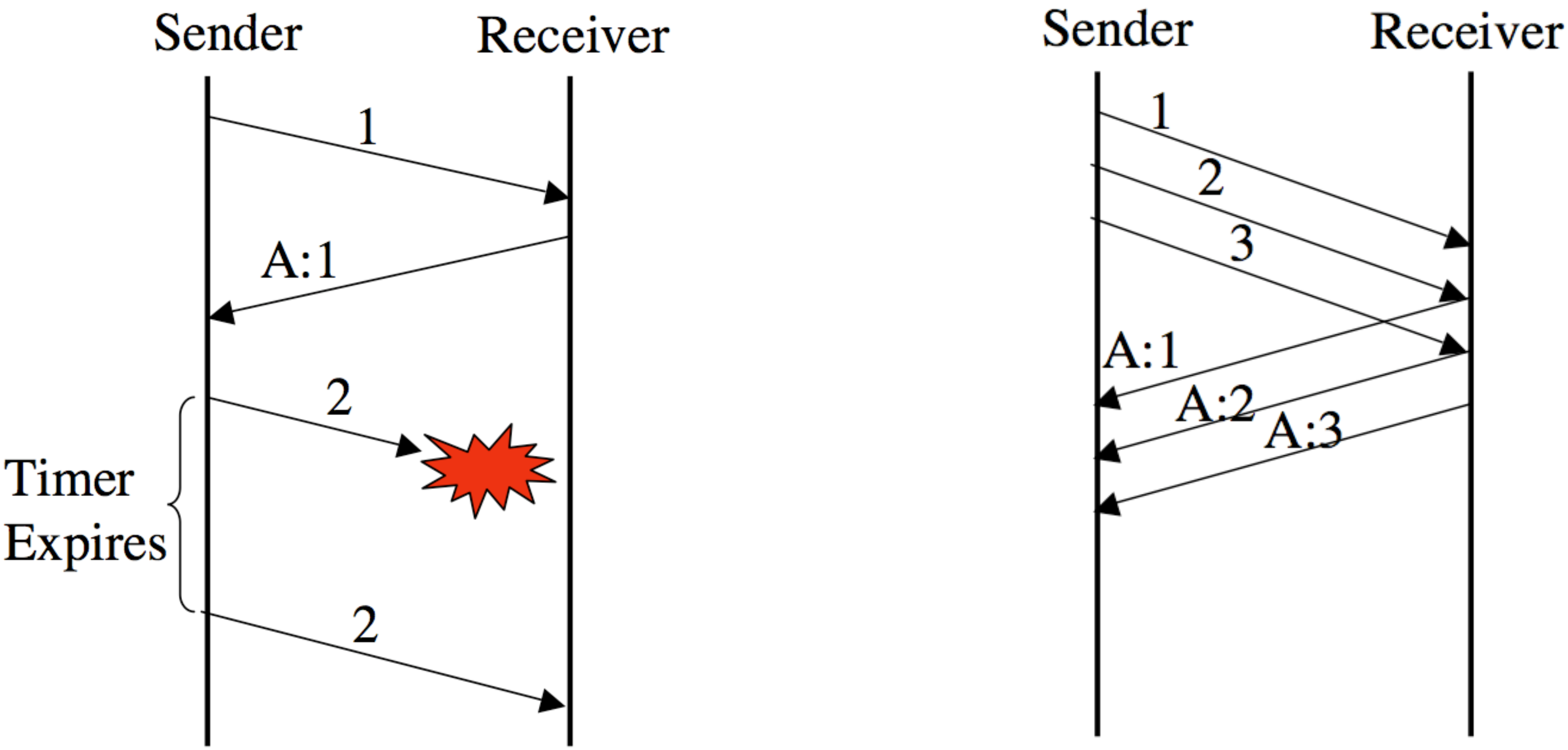
- Can the network keep up with the data rate?
  - Finite link bandwidth
  - Finite buffer size at routers
- End-to-end solution: TCP makes guesses about the available network capacity and adapts. Only end-hosts are involved, not routers.

The danger of congestion

- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- Cliff – point after which
  - Throughput starts to decrease very fast to zero (congestion collapse)
  - Delay approaches infinity



Performance of Sliding Window



- *Throughput = window size * pkt size / RTT*
  - How to set the retransmission timer?
  - How to set window size?

Setting Retransmission Timer

- Observe and adapt: use exponential moving average based on measurements.

$$A(n) = b*A(n-1) + (1 - b)T(n)$$
$$D(n) = b*D(n-1) + (1- b)*(T(n) – A(n))$$
$$Timeout(n) = A(n) +4D(n)$$

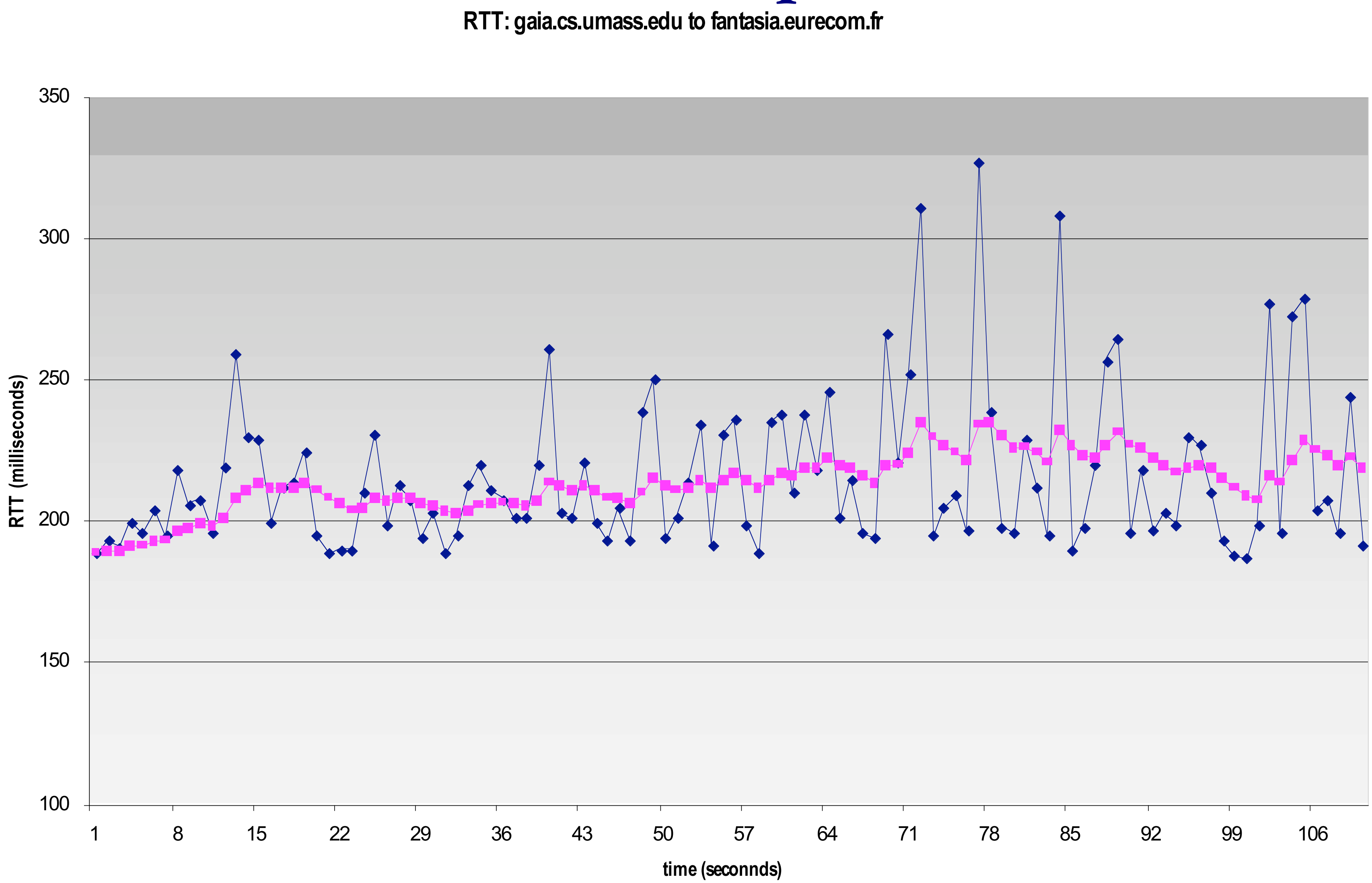
Question: Why not set timeout to average delay?

T(n) is current measured RTT, A(n) is estimated average RTT, D(n) is estimated average deviation.

Karn’s algorithm for updating timer in case of retransmission

1. Don’t take sample of T(n) if retransmitted.
2. Double Timeout after timeout …
3. Reset Timeout for new packet when receiving ACK

RTT from Experiments



Blue: sampleRTT, Pink: SRTT

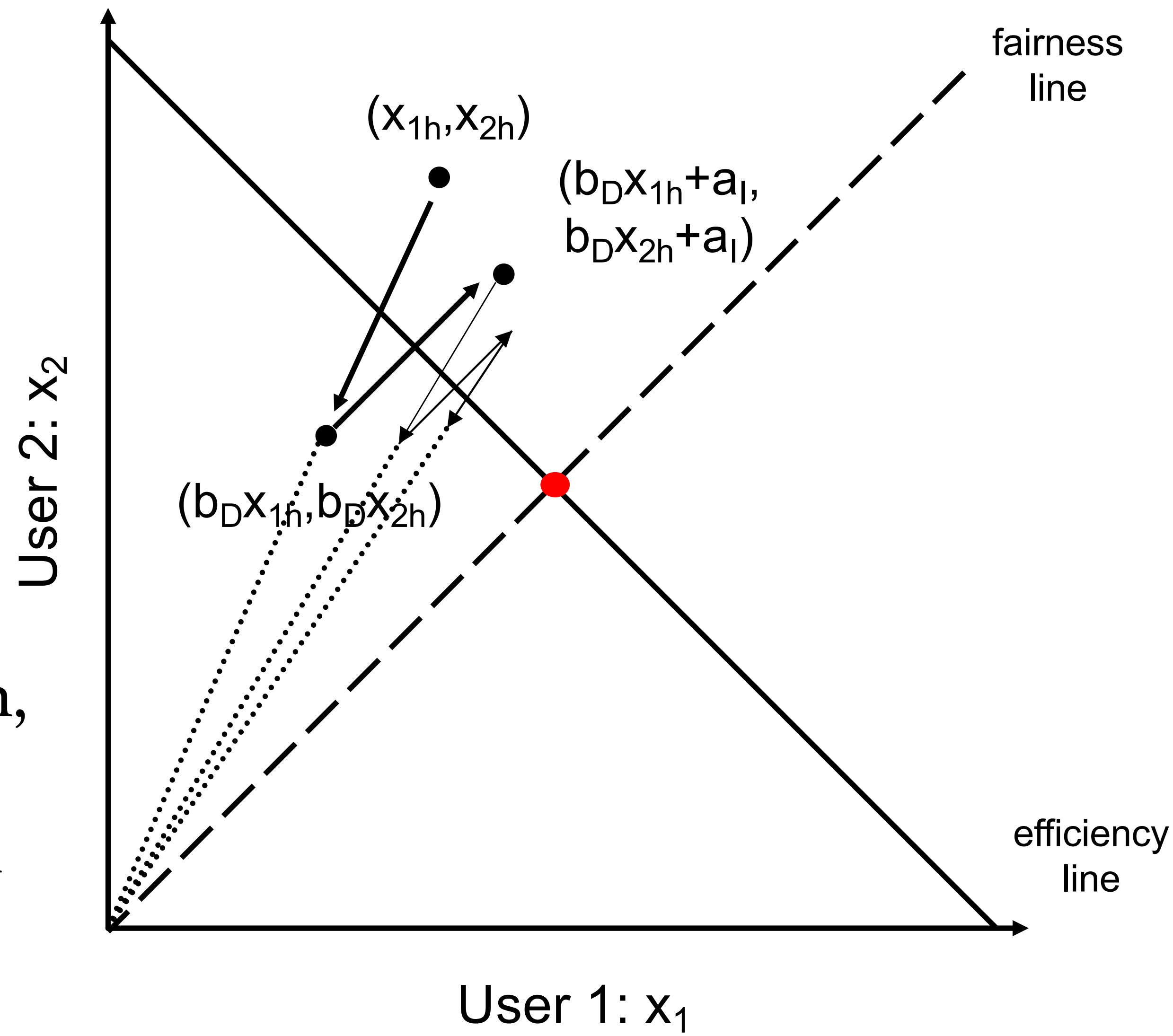
How to adjust window size?

- Maintain three variables:
  - cwnd: congestion window
  - flow_win: flow window, i.e., receiver's advertised window
  - ssthresh: threshold size (used to update cwnd)

  For sending, use: win = **min**(flow_win, cwnd)
- Test the limit of the network
  - when `cwnd < ssthresh`, increase `cwnd` exponentially.
  - when `cwnd ≥ ssthresh`, increase `cwnd` linearly (AI)
- Detect congestion by timeout:
  - If packet lost, have gone too far
  - reduce rate drastically (MD)
- Additive Increase, Multiplicative Decrease (AIMD)

Additive Increase, Multiplicative Decrease

- Two TCP flows share a single bottleneck link.
- In an ideal design, they should get equal share of the bottleneck bandwidth, and total traffic is within the bottleneck capacity.



Slow Start Phase

- Goal: reach the knee point quickly
- Upon (re)starting a TCP connection:
  - Set *cwnd* =1
  - Each time a segment is acknowledged increment *cwnd* by one,

    cwnd += 1

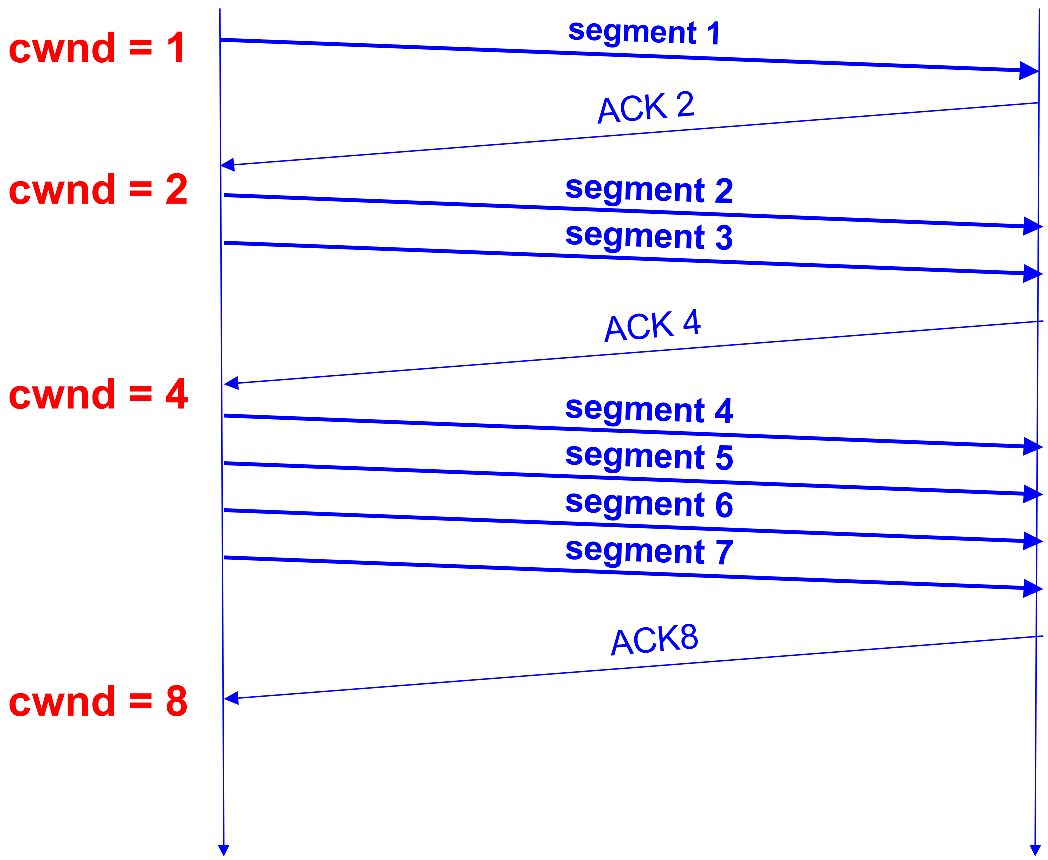
- Slow Start is not actually slow
  - After all segments in the window have been acked, *cwnd* has been doubled over one RTT. That is, *cwnd* increases exponentially over time.
  - But slower than sending all at once.

Slow Start Example

- The congestion window size grows rapidly

- TCP will slow down the increase of *cwnd* when

  cwnd >= ssthresh



Congestion Avoidance Phase

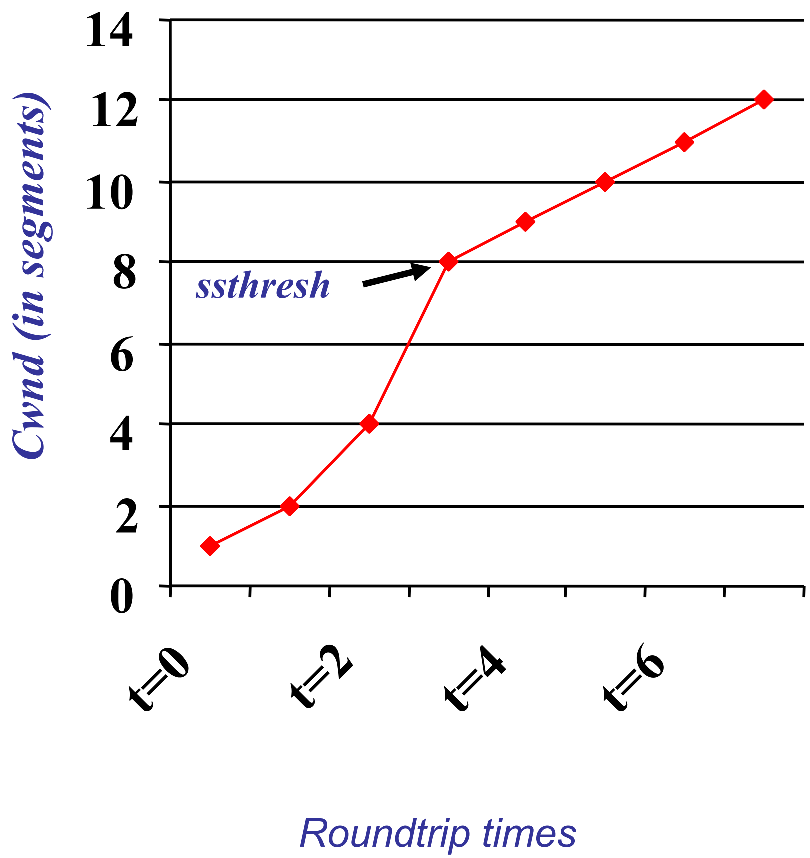
- Slow down "Slow Start"
- *ssthresh* is lower-bound guess about location of knee

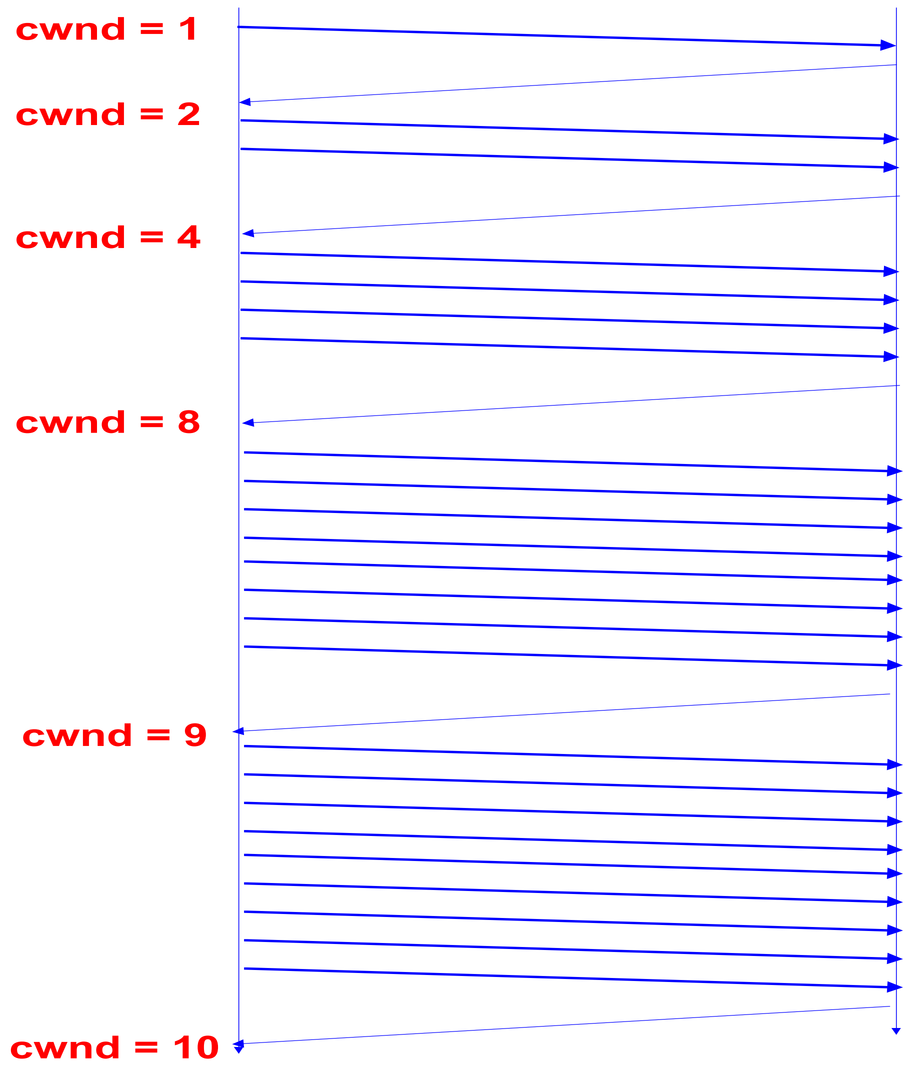
- **If** *cwnd* >= *ssthresh* **then**
  each time a segment is acknowledged
  increment *cwnd* by *1/cwnd*, i.e, cwnd += 1/cwnd

- The effect is that, after *all* segments in the window have been acknowledged (i.e., one RTT), *cwnd* has been incremented by 1.

Example

- Assume that *ssthresh* = *8*





Retransmission Timeout (RTO)

- Upon a retransmission timeout
  - Assume packet loss due to congestion
  - Cut window size drastically

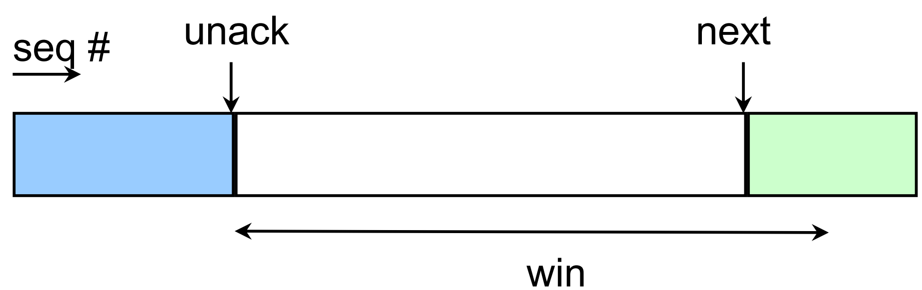
```
ssthresh = cwnd/2;
cwnd = 1;
```

Putting Everything Together

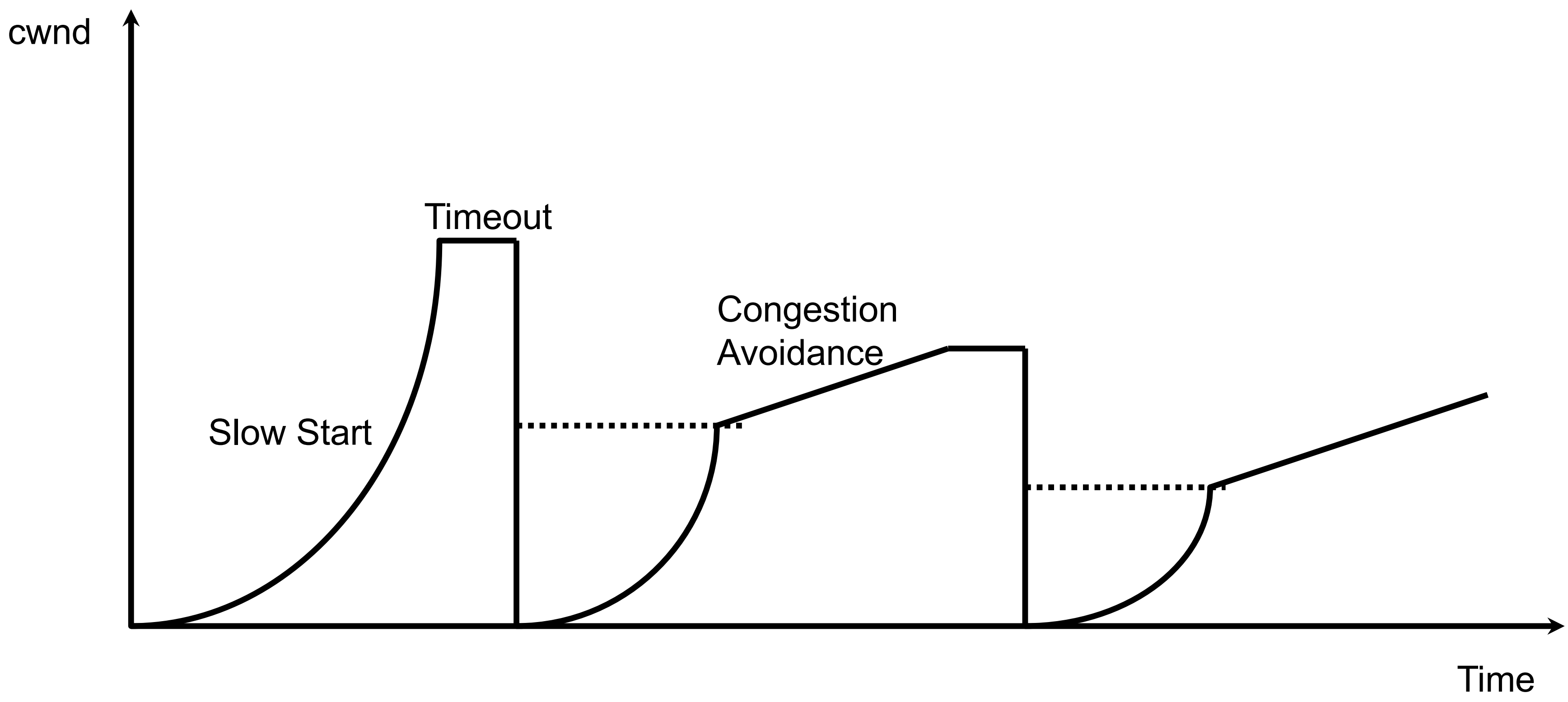
```
Initially:
    cwnd = 1;
    ssthresh = recvWin;
New ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd;
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2;
    cwnd = 1;
```

```
while (next < unack + win)
    transmit next packet;

where   win = min(cwnd,recvWin);
```

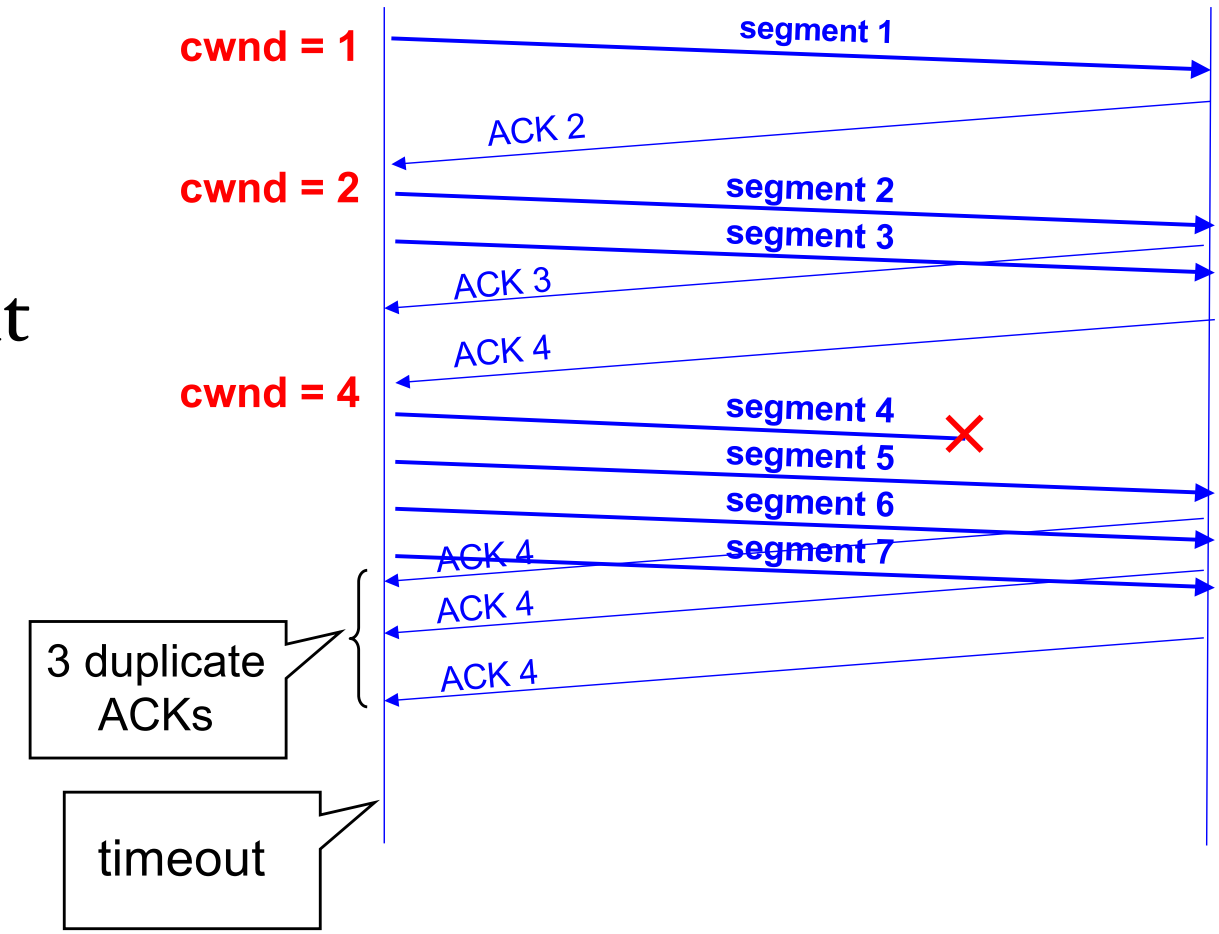


TCP Reno behavior



Fast Retransmission

- Don't wait for window to drain
- Resend a segment after 3 duplicate ACKs



Fast Recovery

- After a fast retransmission, set *cwnd* to *cwnd/2*
  - i.e., don't reset *cwnd* to 1
- But when RTO expires still do *cwnd* = 1
- Fast Retransmit and Fast Recovery
  - Implemented by TCP Reno
  - widely used version of TCP today
- Lesson: avoid RTOs at all costs!

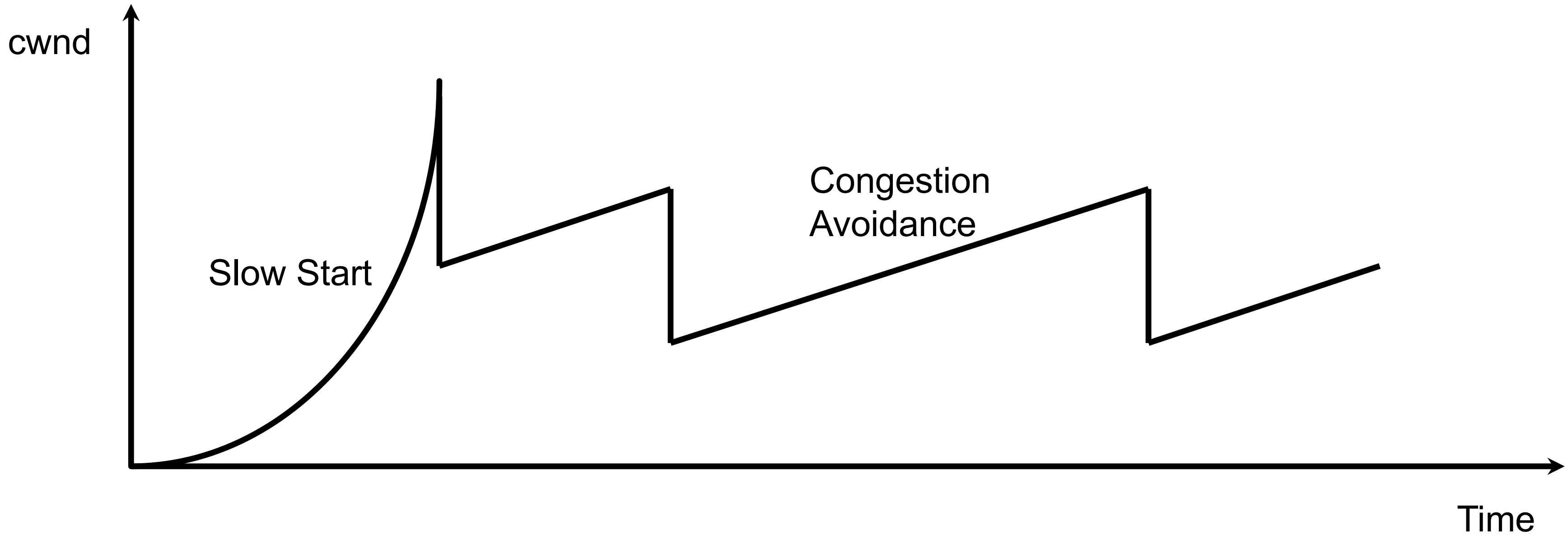
Fast Retx and Fast Recovery

```
Initially:
    cwnd = 1;
    ssthresh = recvWin;

New ack received:
    if (cwnd < ssthresh)          /* slow start*/
        cwnd = cwnd + 1;
    else                          /* congestion avoidance */
        cwnd = cwnd + 1/cwnd;

Loss detected:
    ssthresh = cwnd/2;            /* multiplicative decrease */
    if (3 duplicated ACKs)        /* fast retransmission */
            cwnd = ssthresh;      /* fast recovery*/
    else
            cwnd = 1;
```

TCP NewReno Behavior (Fast Retransmit and Fast Recovery)



- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, *cwnd* oscillates around the optimal window size.

Future Development

- Many more TCP congestion control schemes have been proposed and implemented.
  - Only end-hosts are involved, not routers

- There're also designs that require router support
  - Will be able to detect and react to congestion more accurately
  - But need deployment and support from routers
  - Usually only happens in special networks.