# CS419: Computer Networks

Lecture 10, Part 3: Apr 13, 2005

*Transport:  TCP performance*

# TCP performance

CS419

- We’ve seen how TCP “the protocol” works
- But there are a lot of tricks required to make it work well
  - Indeed, the Internet nearly died an early death because of bad TCP performance problems

# TCP performance

CS419

- Making interactive TCP efficient for low-bandwidth links
- Filling the pipe for bulk-data applications
- Estimating round trip time (RTT)
- Keeping the pipe full
- Avoiding congestion

# Interactive TCP

CS419

- Interactive applications like telnet or RPC send only occasional data
- Data sent in both directions
- Data often very small
- Packet overhead is huge for small packets
  - <3% efficiency for a 1-byte data packet
  - This is bad for low-bandwidth links

# Who cares about low-BW links?

CS419

- Historically low-BW links were a serious problem
  - As access links got faster, people worried less about this
- Ubiquitous computing over TCP/IP wireless links makes this interesting again
  - Low-power devices

# Transmit versus wait



- One basic engineering tradeoff is to wait before transmitting
- Wait for more data to send a bigger packet
- Hold off on the ACK so that data can be piggybacked with the ACK
- This is not an easy tradeoff to make---you can only go so far with this approach

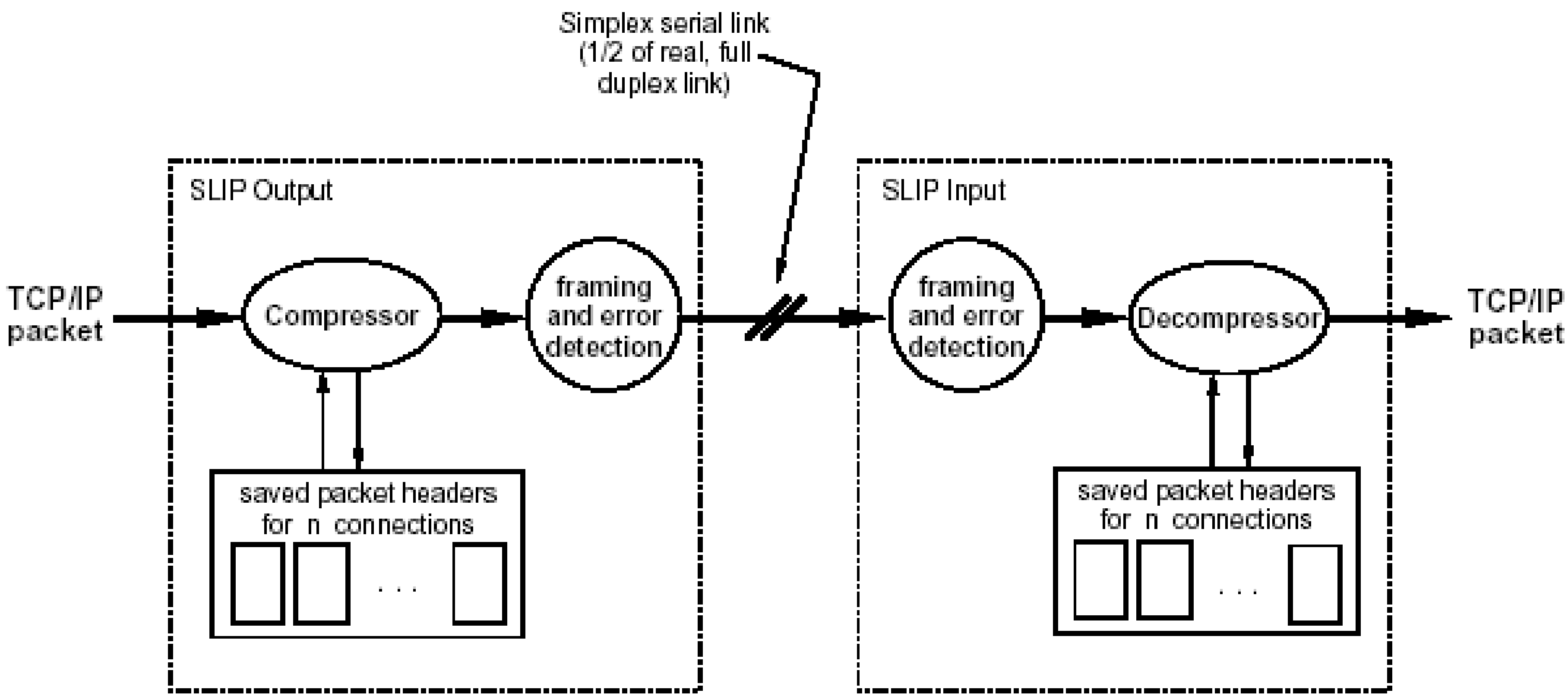
# TCP/IP header compression

CS419

- A better approach is to “compress” the TCP and IP headers (RFC 1144, 2507 - 2509)
- Basic idea is to:
  - not transmit fields that don’t change from packet to packet,
  - and to transmit only the deltas of those fields that do change

# TCP/IP compression components

CS419



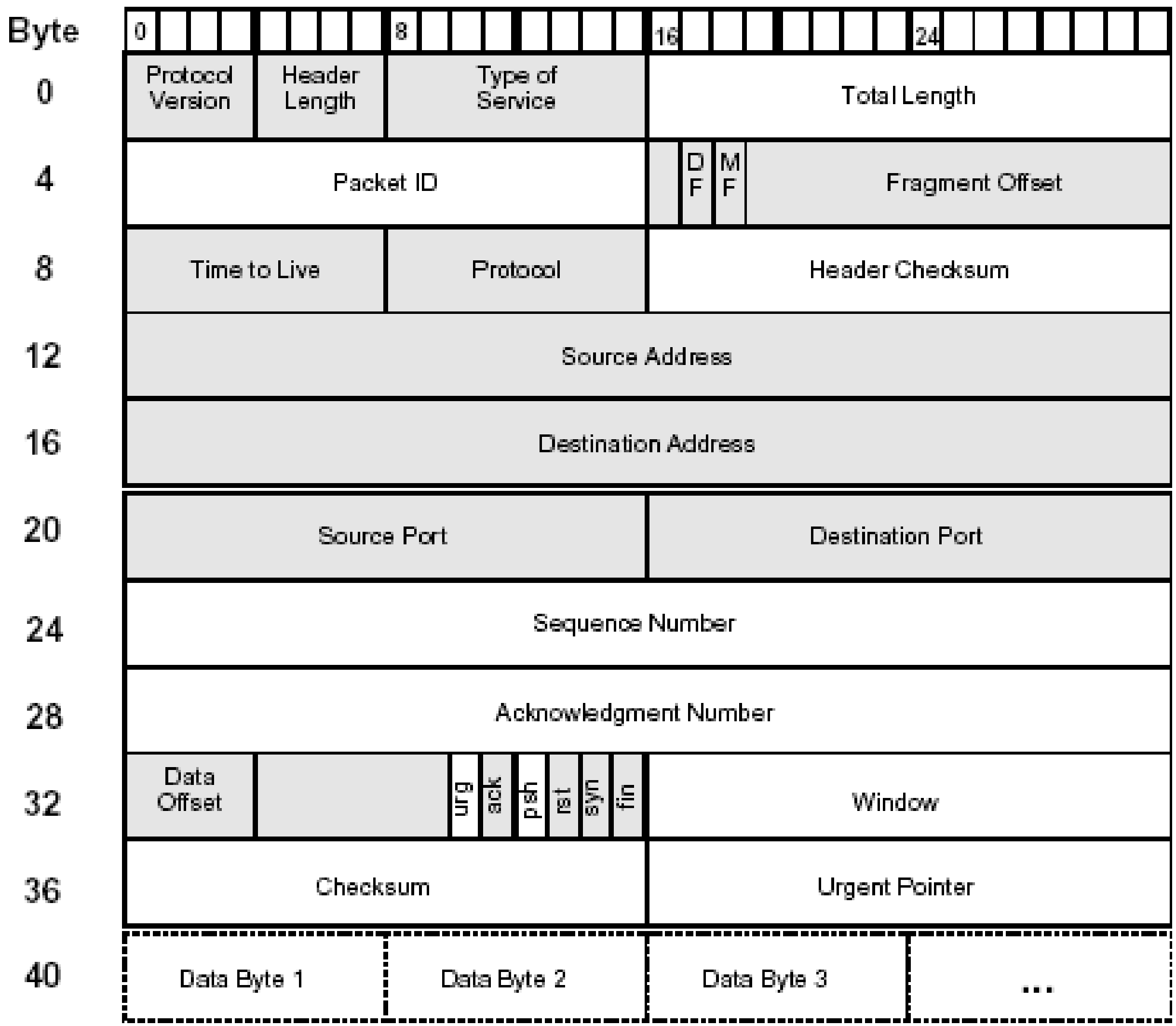
# TCP header compression

CS419

- How much compression can we get out of TCP/IP
- From 40 bytes to:
  - 20 bytes?
  - 10 bytes?
  - 5 bytes?
  - 2 bytes?

# TCP/IP fields that don't change

CS419




This cuts the header in half!

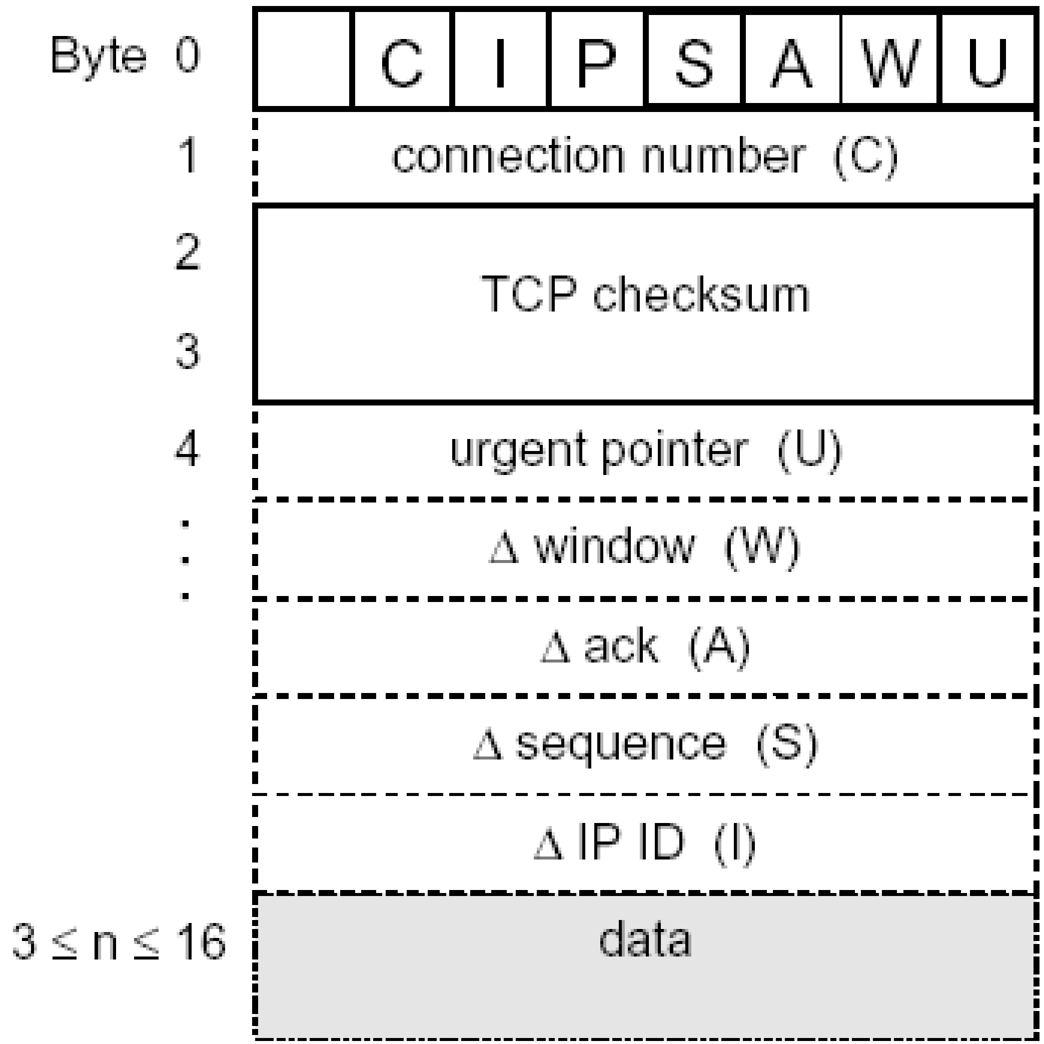
# More compression



- Total length not needed because link layer transmits that (2 bytes)
- IP checksum not needed because there isn't much left to checksum (2 more bytes)

# Compression header

CS419



# Compression issues



- The main issue is how to deal with errors
- Once an error occurs, the decompressor can’t recover unless a new complete packet is sent
- RFC1144 has a clever solution to this . . .

# When to schedule transmission

CS419

- As we saw, TCP segment transmit doesn't have to correspond to app `send()`
- When should TCP send a fragment?
  - As soon as it gets data to send?
  - As soon as it has a packet's worth to send (MSS Max Segment Size)?
  - Not until some timer goes off?

# When to schedule transmission

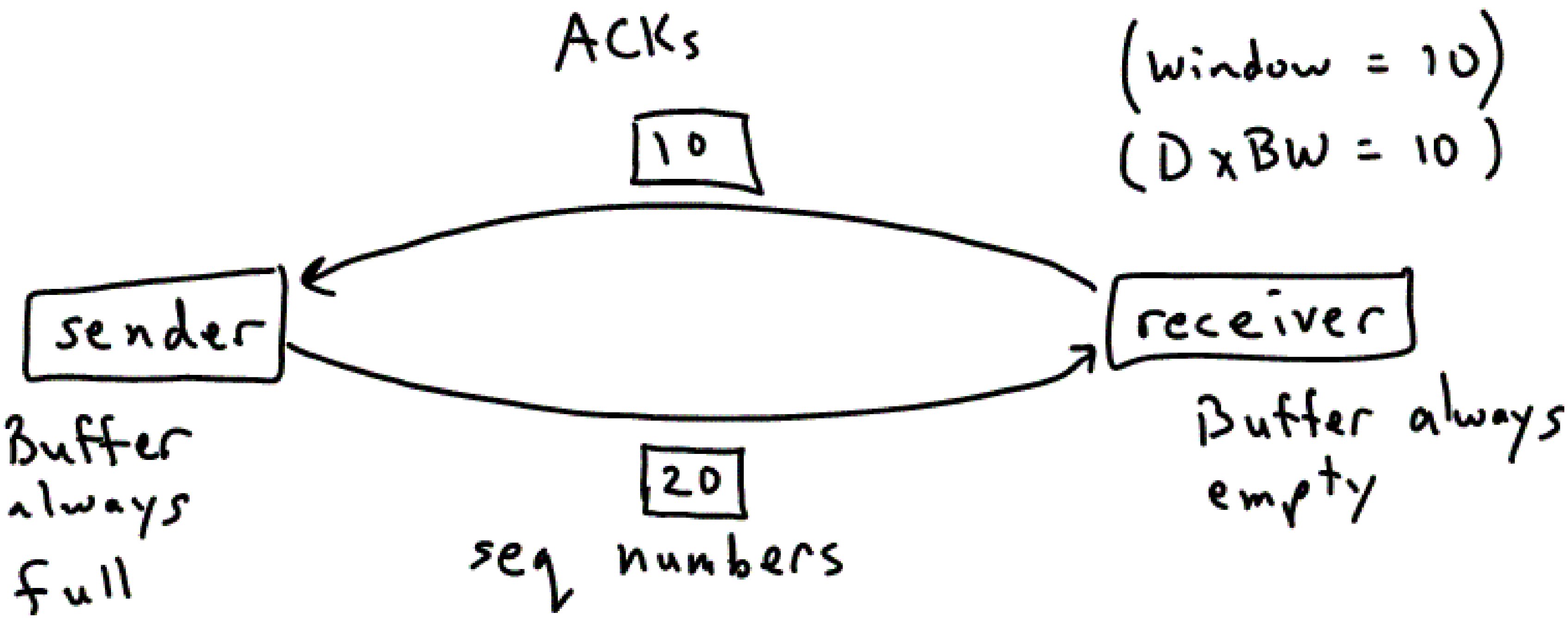


- If TCP sends right away, it may send many small packets
- If TCP waits for a full MSS, it may delay important data
- If TCP waits for a timer, then bad behavior can result
  - Lots of small packets get sent anyway
  - Silly Window Syndrome

# Silly Window Syndrome

CS419

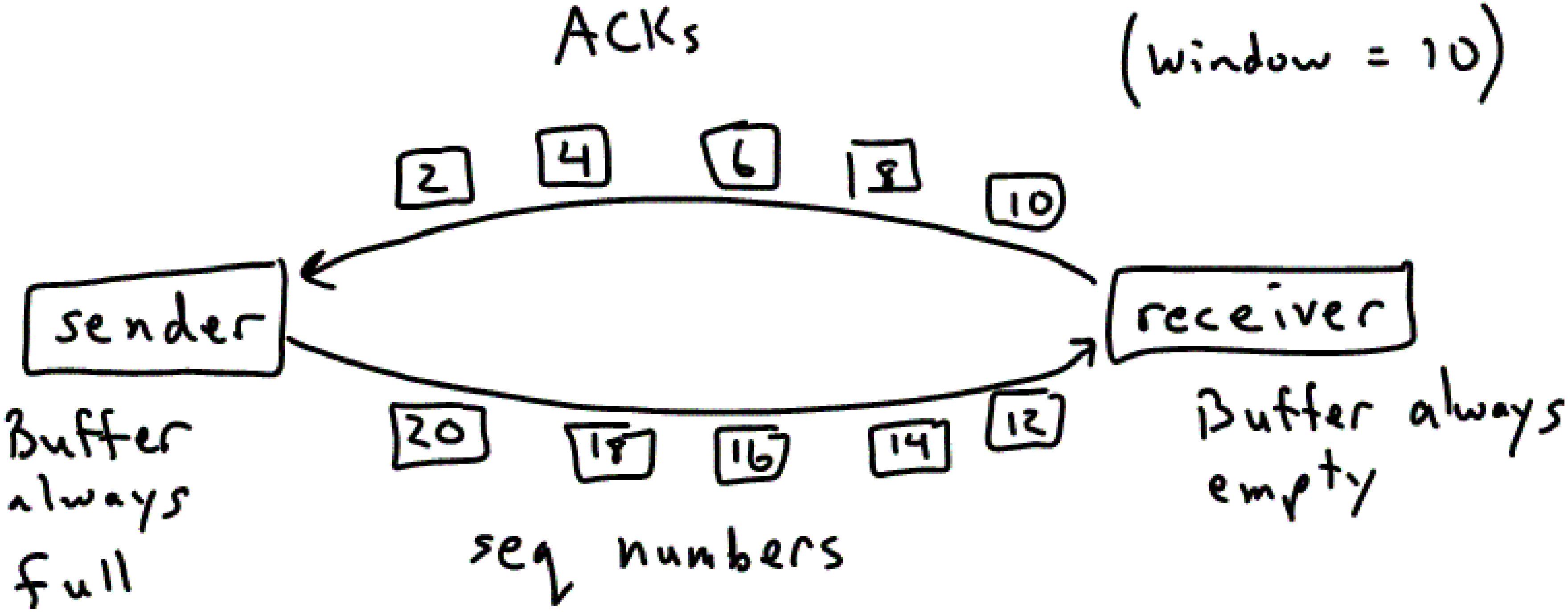
- This is a nice situation:
  - (nice big packets, full pipe)



# Silly Window Syndrome

CS419

- Imagine this situation:
  - How could we get out of it???



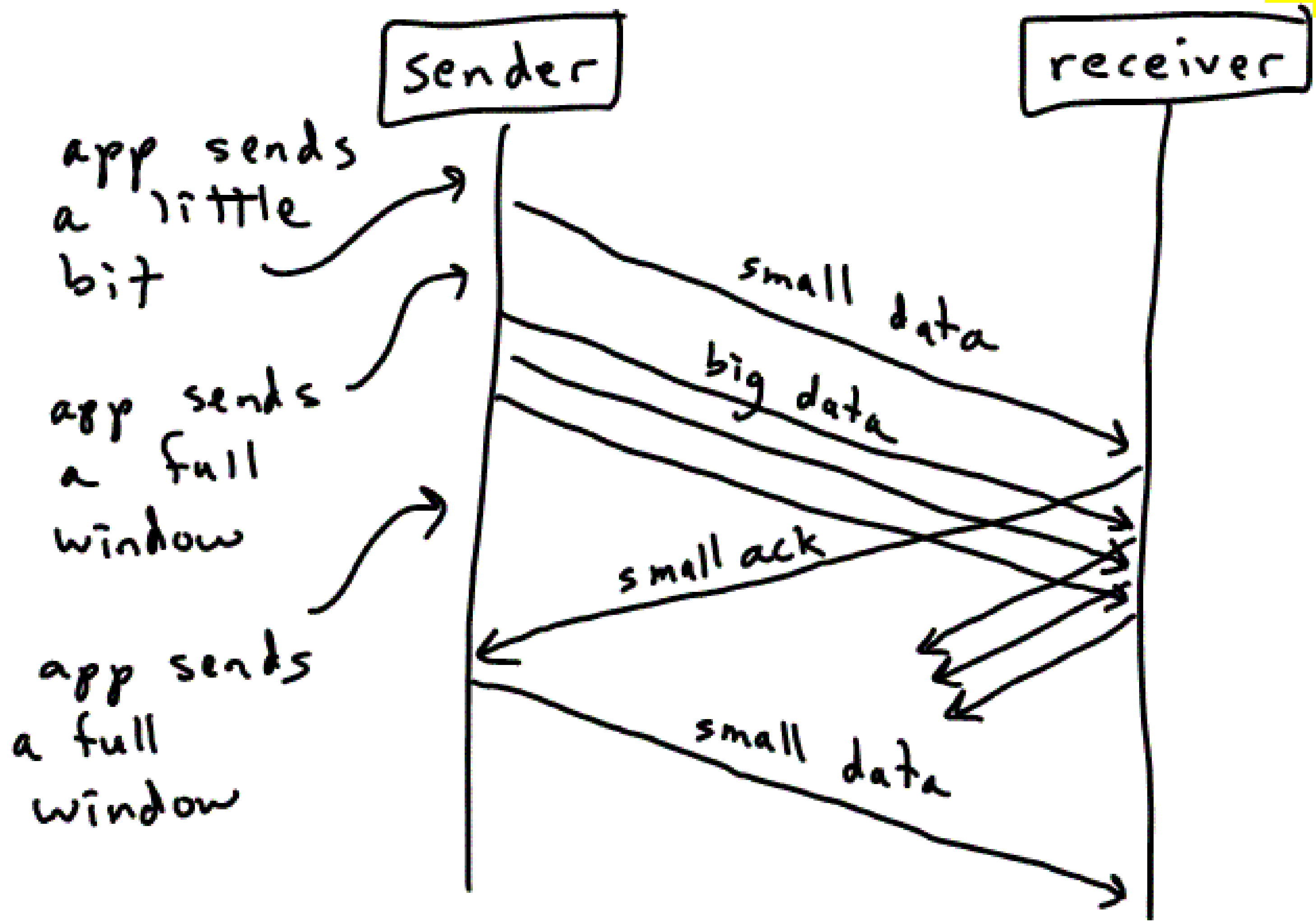
# Silly Window Syndrome



- Small packets introduced into the loop tend to stay in the loop
- How do small packets get introduced into the loop?

# Silly Window Syndrome: Small packet introduced

CS419



# Silly Window Syndrome prevention



- Receiver and sender both wait until they have larger segments to ACK or send
- Receiver:
  - Receiver will not advertise a larger window until the window can be increased by one full-sized segment <u>or</u>
  - by half of the receiver’s buffer space whichever is smaller

# Silly Window Syndrome prevention

CS419

- Sender:
  - Waits to transmit until either a full sized segment (MSS) can be sent <u>or</u>
  - at least half of the largest window ever advertised by the receiver can be sent <u>or</u>
  - it can send everything in the buffer

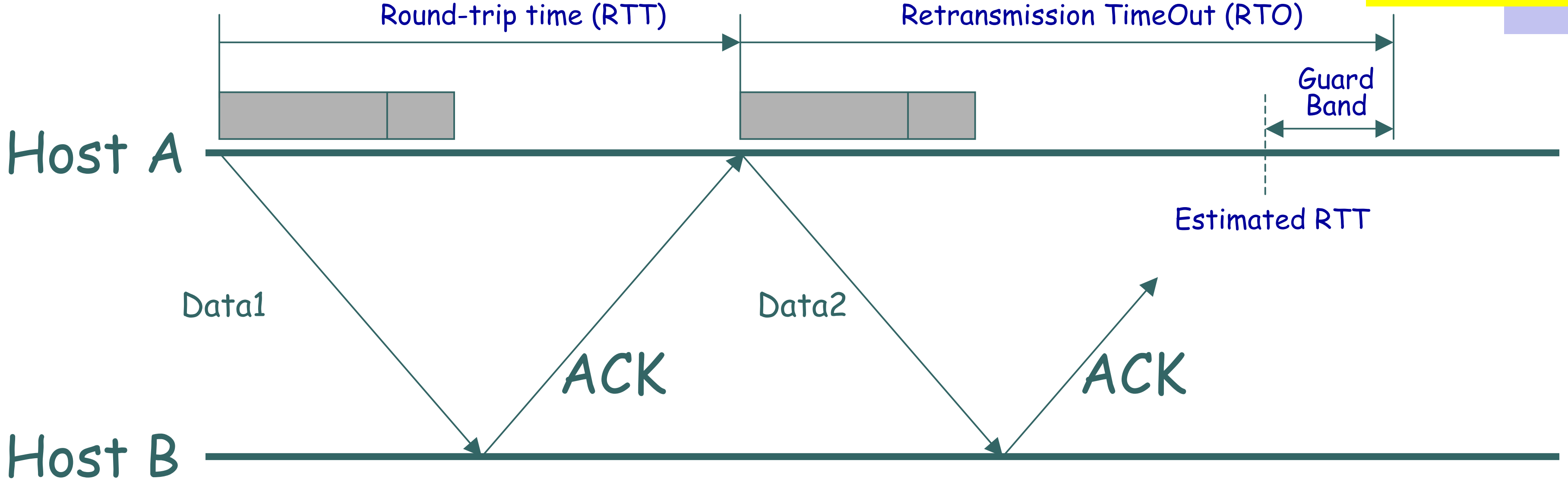
# When to schedule transmission (again)



- App can force sender to send immediately when data is available
  - Sockopt TCP_NODELAY
- Otherwise, sender sends when a full MSS is available
- Or when a timer goes off
  - But with silly window constraints…

# TCP: Retransmission and Timeouts

CS419




TCP uses an adaptive retransmission timeout value:

Congestion
Changes in Routing

} RTT changes frequently

Next few slides from Nick McKeown, Stanford

# TCP: Retransmission and Timeouts

CS419

Picking the RTO is important:

- Pick a values that's too big and it will wait too long to retransmit a packet,
- Pick a value too small, and it will unnecessarily retransmit packets.

<u>The original algorithm for picking RTO:</u>

1. $EstimatedRTT_k = \alpha \, EstimatedRTT_{k-1} + (1 - \alpha) \, SampleRTT$
2. $RTO = 2 * EstimatedRTT$

Determined empirically

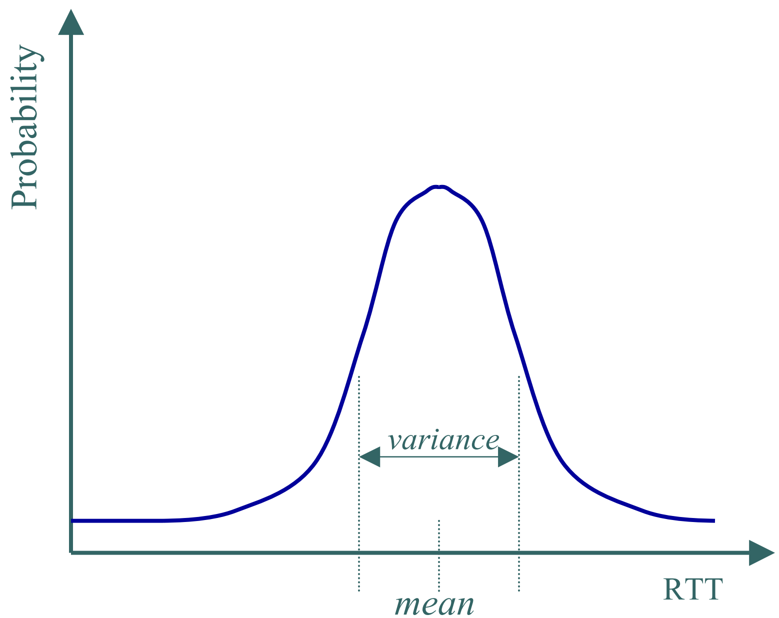
Characteristics of the original algorithm:

- Variance is assumed to be fixed.
- But in practice, variance increases as congestion increases.

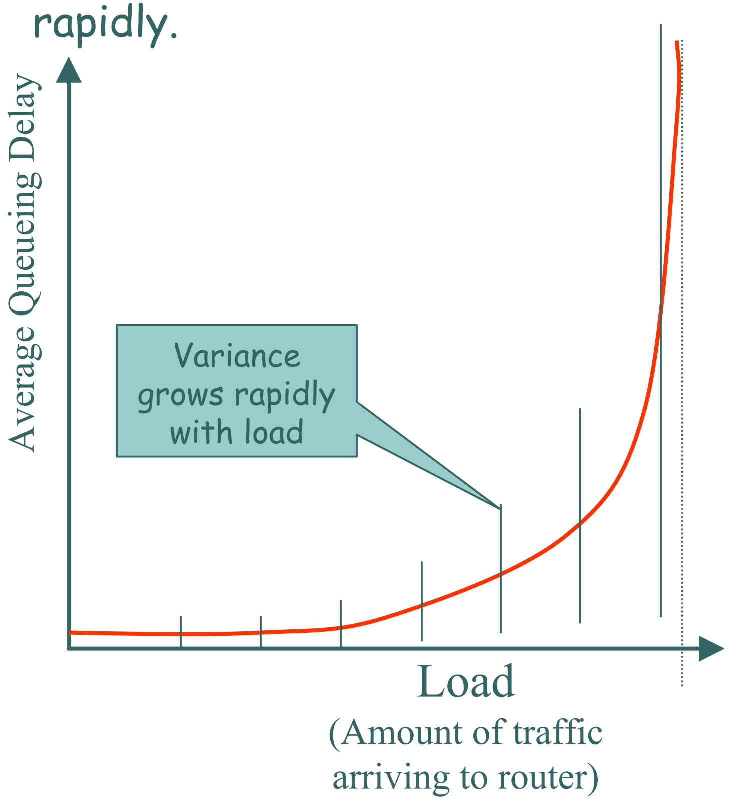
# TCP: Retransmission and Timeouts

CS419

- There will be some (unknown) distribution of RTTs.
- We are trying to estimate an RTO to minimize the probability of a false timeout.



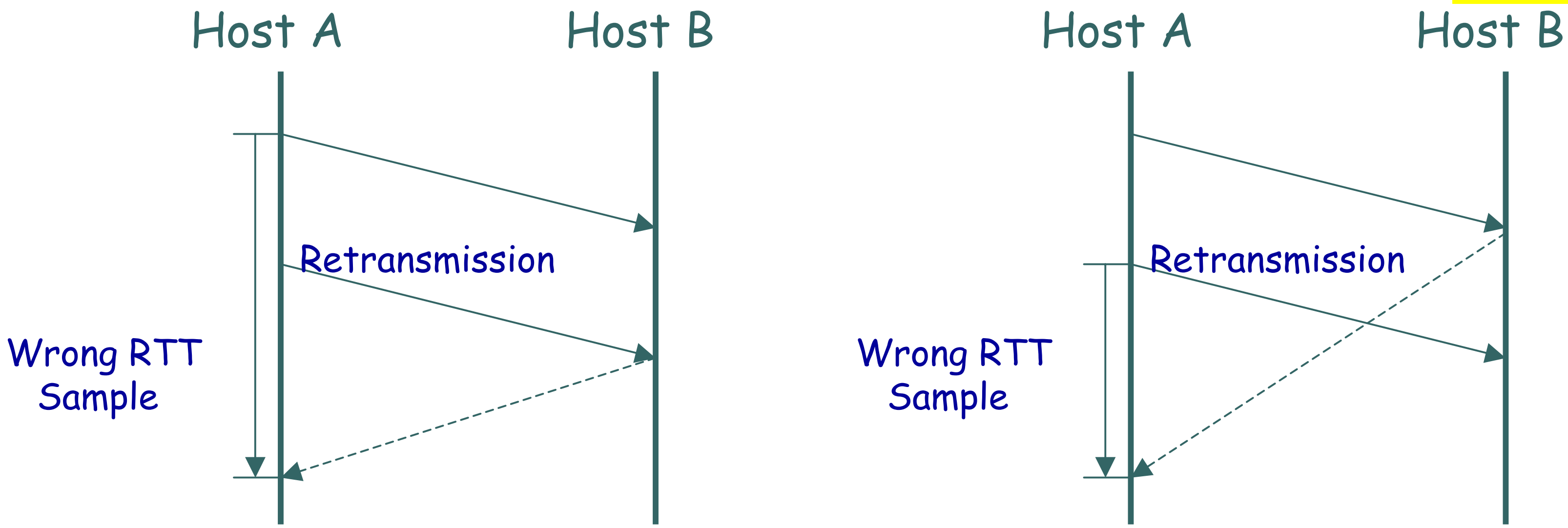

- Router queues grow when there is more traffic, until they become unstable.
- As load grows, variance of delay grows rapidly.



# TCP: Retransmission and Timeouts

*Karn's Algorithm*

CS419




**Problem:**

How can we estimate RTT when packets are retransmitted?

**Solution:**

On retransmission, don't update estimated RTT (and double RTO).

# TCP: Retransmission and Timeouts

CS419

Newer Algorithm includes estimate of variance in RTT:

Same as before

- Difference = SampleRTT - EstimatedRTT
- $\text{EstimatedRTT}_k = \text{EstimatedRTT}_{k-1} + (\delta * \text{Difference})$
- Deviation = Deviation + $\delta$*( |Difference| - Deviation )

- RTO = $\mu$ * EstimatedRTT + $\phi$ * Deviation

  $\mu \approx 1$

  $\phi \approx 4$

# Fast implementation of this

CS419

```
SampleRTT -= (EstimatedRTT >> 3);
EstimatedRTT += SampleRTT;
if (SampleRTT < 0)
      SampleRTT = -SampleRTT;
SampleRTT -= (Deviation >>3);
Deviation += SampleRTT;
TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);
```

# Fast implementation of this

CS419

- Note no floating point arithmetic, just adds, subtract, and shift!

```
SampleRTT -= (EstimatedRTT >> 3);
EstimatedRTT += SampleRTT;
if (SampleRTT < 0)
      SampleRTT = -SampleRTT;
SampleRTT -= (Deviation >>3);
Deviation += SampleRTT;
TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);
```

- Also, TCP implementations use “header prediction” to gain execution speed

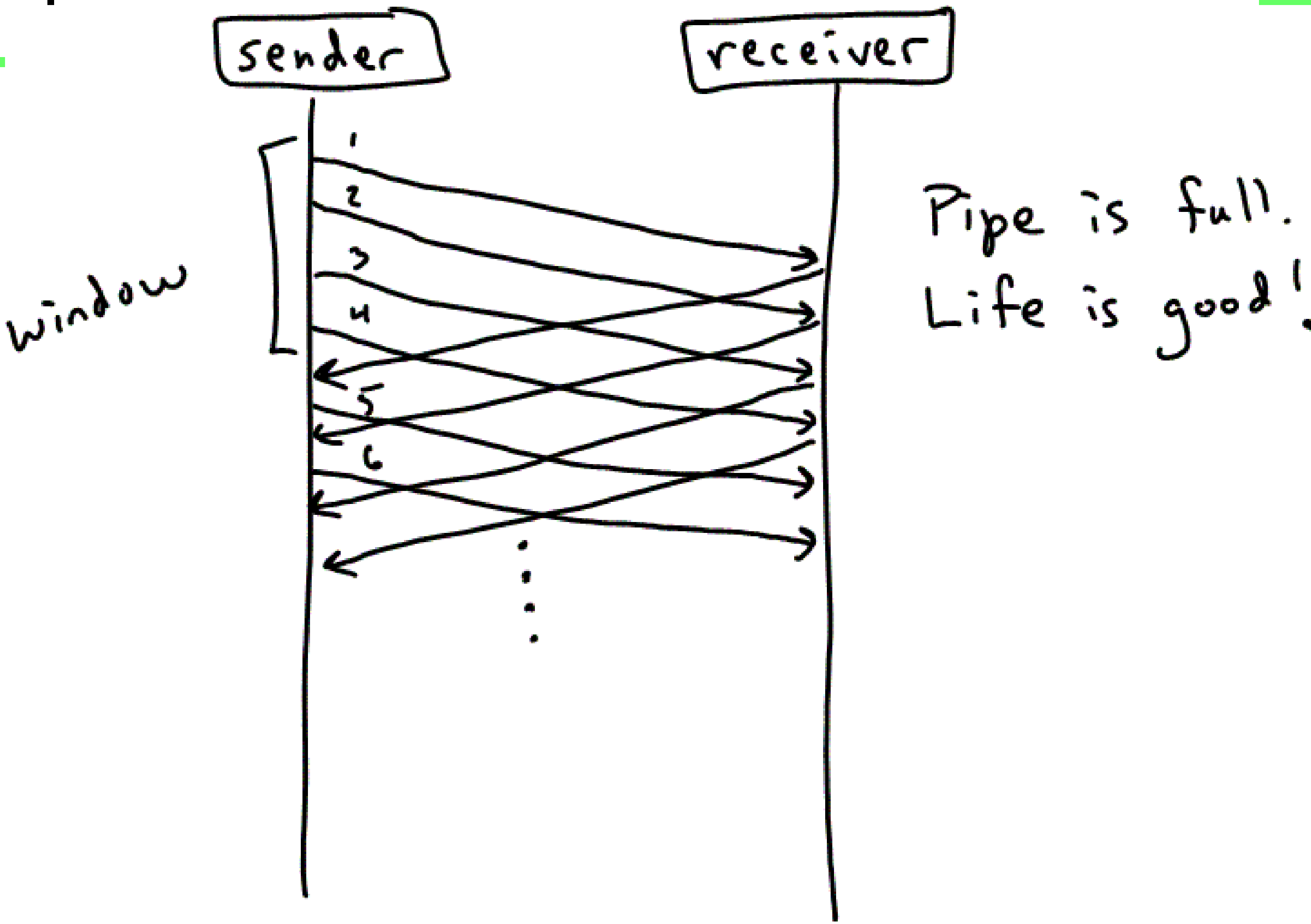
# Fast Retransmit



- Even with all this fancy RTT estimation, retransmits still tend to over-estimate, and TCP can stall while waiting for a time-out
  - Stall because pipe often bigger than window!
- This leads to the notion of “fast retransmit”

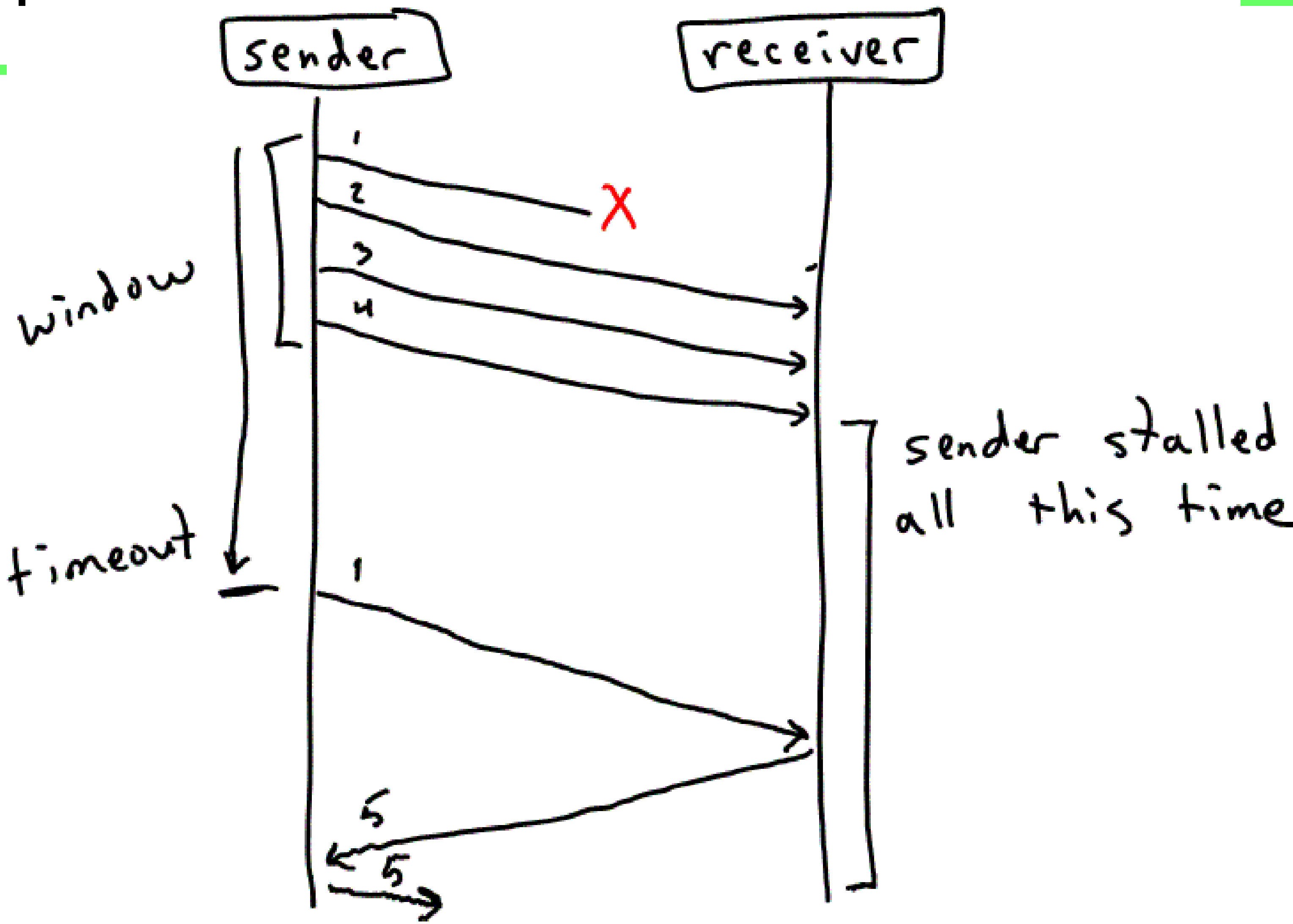
# Delayed connection





# Delayed connection





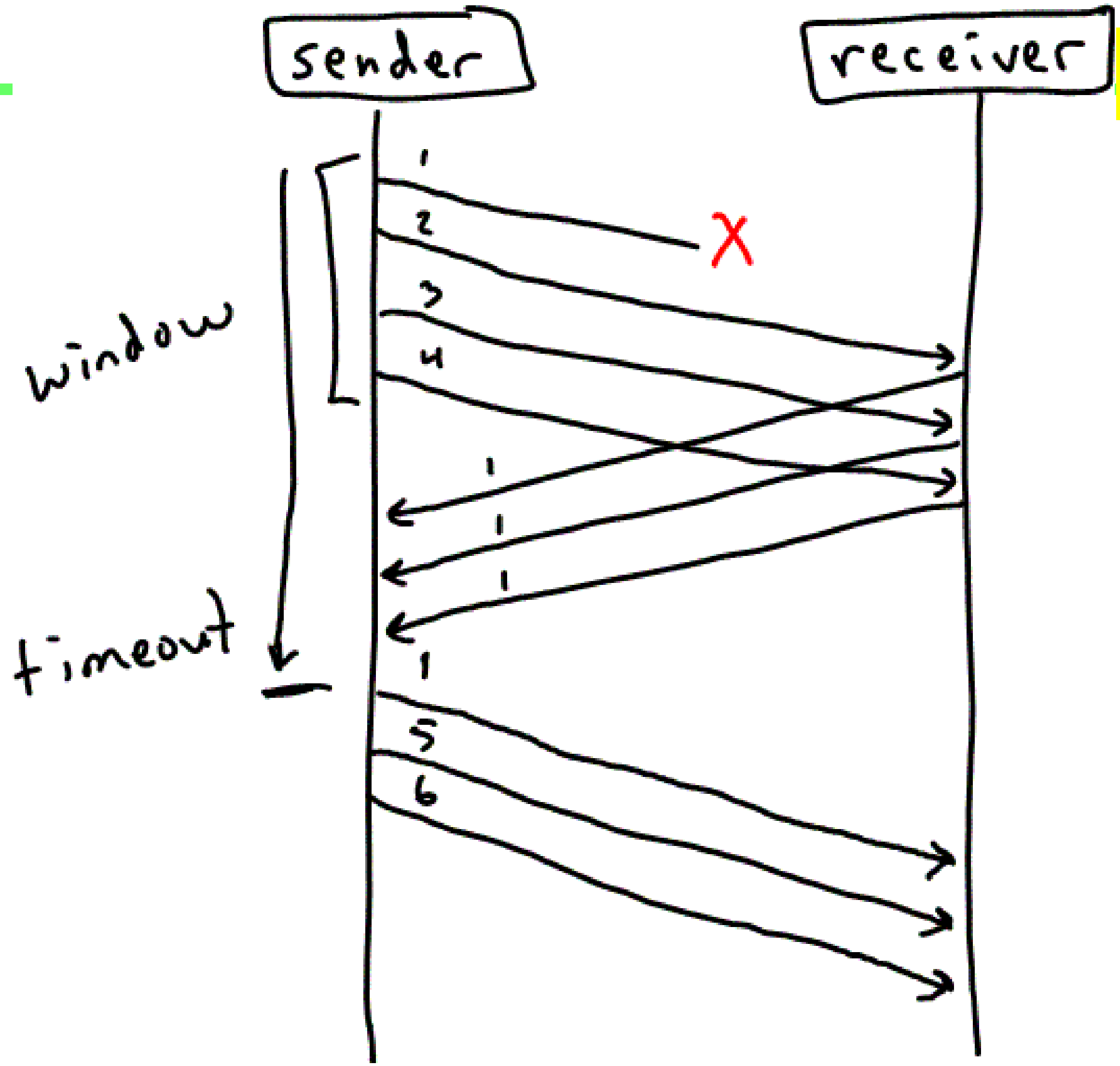
# Fast Retransmit



- Receiver should send an ACK every time it receives a packet, not only when it gets something new to ACK
  - If same bytes are ACK'd, this is called "duplicate ACK"
- Sender interprets 3 duplicate ACKs as a loss signal, retransmits right away
  - Don't wait for timeout

# Fast Retransmit

CS419



# Next Lecture



- TCP congestion control