





CIS 553: Networked Systems

Flow Control

March 25, 2020



Agenda

- Transmission Control Protocol 
 - Reliable in-order delivery 
 - Connection-oriented
 - Flow control
 - Congestion control



Many more optimizations

- Checksums (to detect bit errors)
- Acknowledgements (plus retransmissions)
 - Can we avoid using a timeout per packet?
- Sequence numbers (to deal with duplicates)
 - Can we defend against stale packets?
- Timers (to detect loss)
 - How do we set the timer?



Retransmission timeout

- Problem: How to set timeout for when to retransmit the first packet in the window?
 - Too long: connection has low throughput
 - Too short: retransmit packet that was just delayed
- Solution: make timeout proportional to RTT
 - $RTO = EstimatedRTT + 4 \times StdDevRTT$

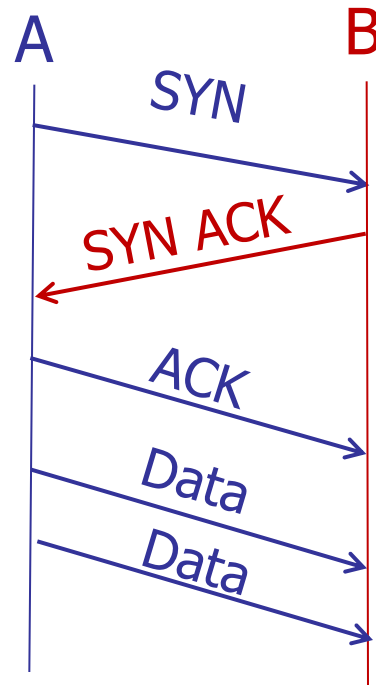


Agenda

- Transmission Control Protocol ✓
 - Reliable in-order delivery ✓
 - Connection-oriented ← NEXT
 - Flow control
 - Congestion control



Establishing a TCP Connection



Each host tells its
ISN to the other
host

- Three-way handshake to establish connection
 - Host A sends a **SYN** (open) to the host B
 - Host B returns a SYN acknowledgment (**SYN ACK**)
 - Host A sends an **ACK** to acknowledge the SYN ACK



TCP Header

Flags: SYN
FIN
RST
PSH
URG
ACK

Source port		Destination port	
Sequence number			
Acknowledgment			
HdrLen	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			
Data			



Step 1: A's Initial SYN Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

A's port		B's port	
A's Initial Sequence Number			
Acknowledgment			
20	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			

A tells B it wants to open a connection...



Step 2: B's SYN-ACK Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

B's port		A's port	
B's Initial Sequence Number			
A's ISN plus 1			
20	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			

B tells A it accepts, and is ready to hear the next byte...
... upon receiving this packet, A can start sending data



Step 3: A's ACK of the SYN-ACK

Flags: SYN
FIN
RST
PSH
URG
ACK

A's port		B's port	
Sequence number			
B's ISN plus 1			
20	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			

A tells B it is okay to start sending
... upon receiving this packet, B can start sending data

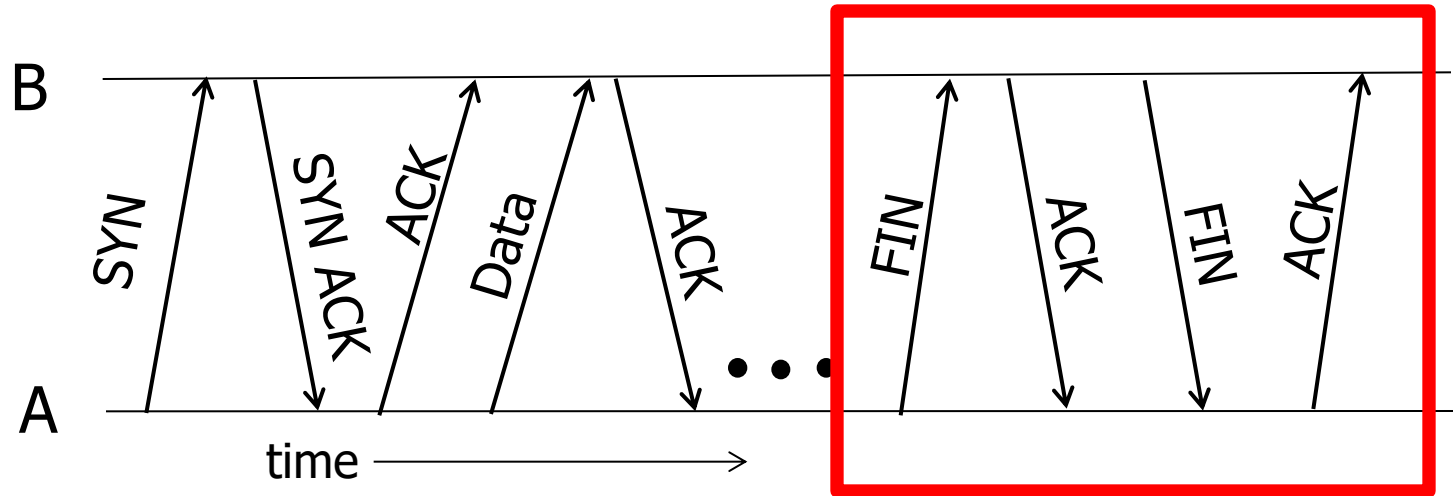


SYN Loss

- Upon sending SYN, sender sets a timer
 - If SYN lost, timer expires before SYN-ACK received
 - Sender retransmits SYN
- How should the TCP sender set the timer?
 - No idea how far away the receiver is
 - Some TCPs use default of 3 or 6 seconds
- Usually handled out-of-band...
 - User gets impatient and hits reload
 - ... Users aborts connection, initiates new socket
 - Essentially, forces a fast send of a new SYN!



Tearing Down the Connection



- Closing (each end of) the connection
 - Finish (FIN) to close and receive remaining bytes
 - And other host sends a FIN ACK to acknowledge
 - Reset (RST) to close and not receive remaining bytes

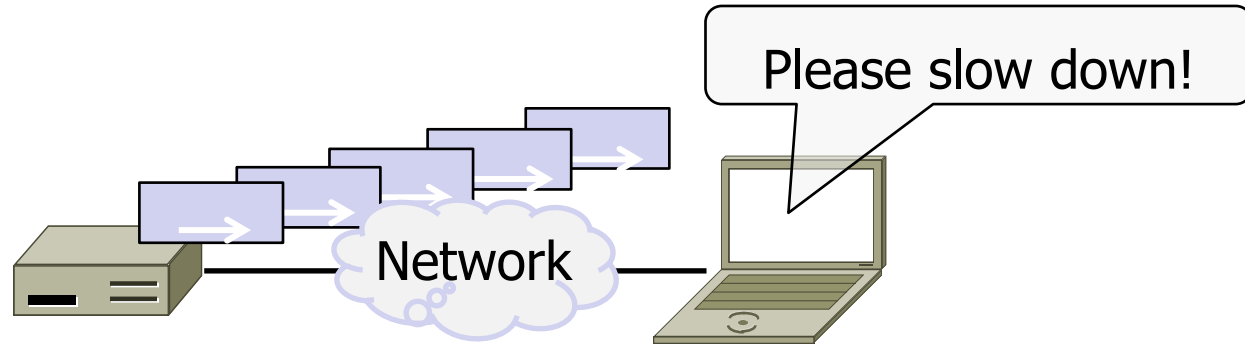


Agenda

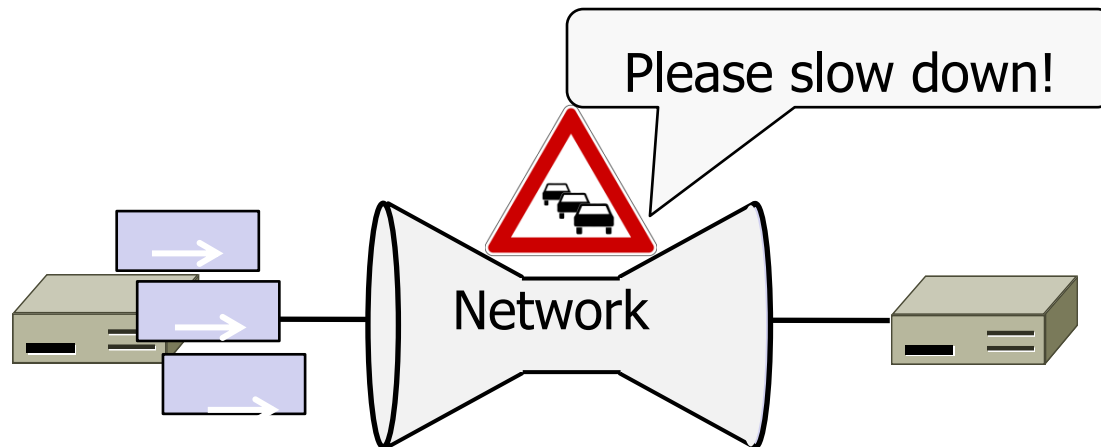
- Transmission Control Protocol ✓
 - Reliable in-order delivery ✓
 - Connection-oriented ✓
 - Flow control ← NEXT
 - Congestion control



Flow control

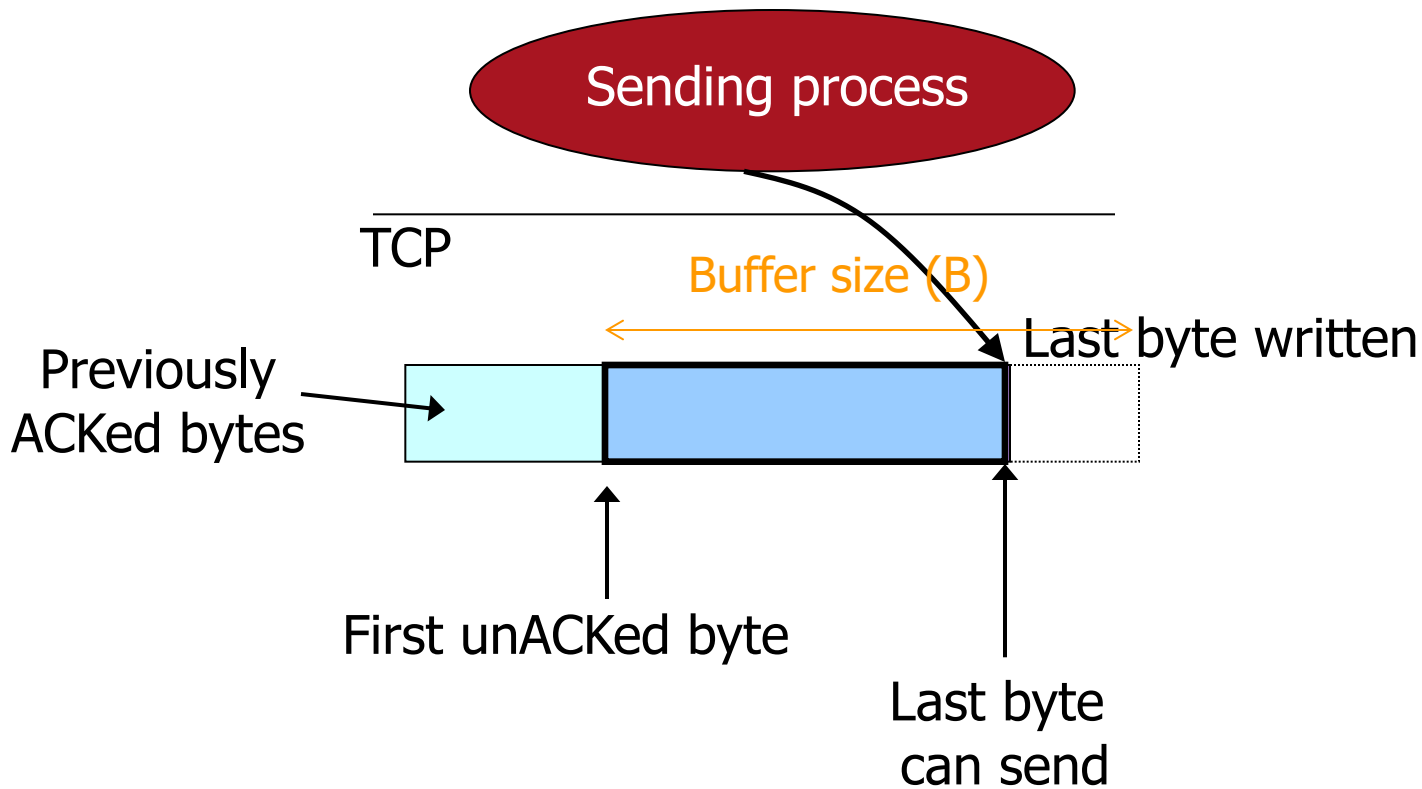


Congestion control





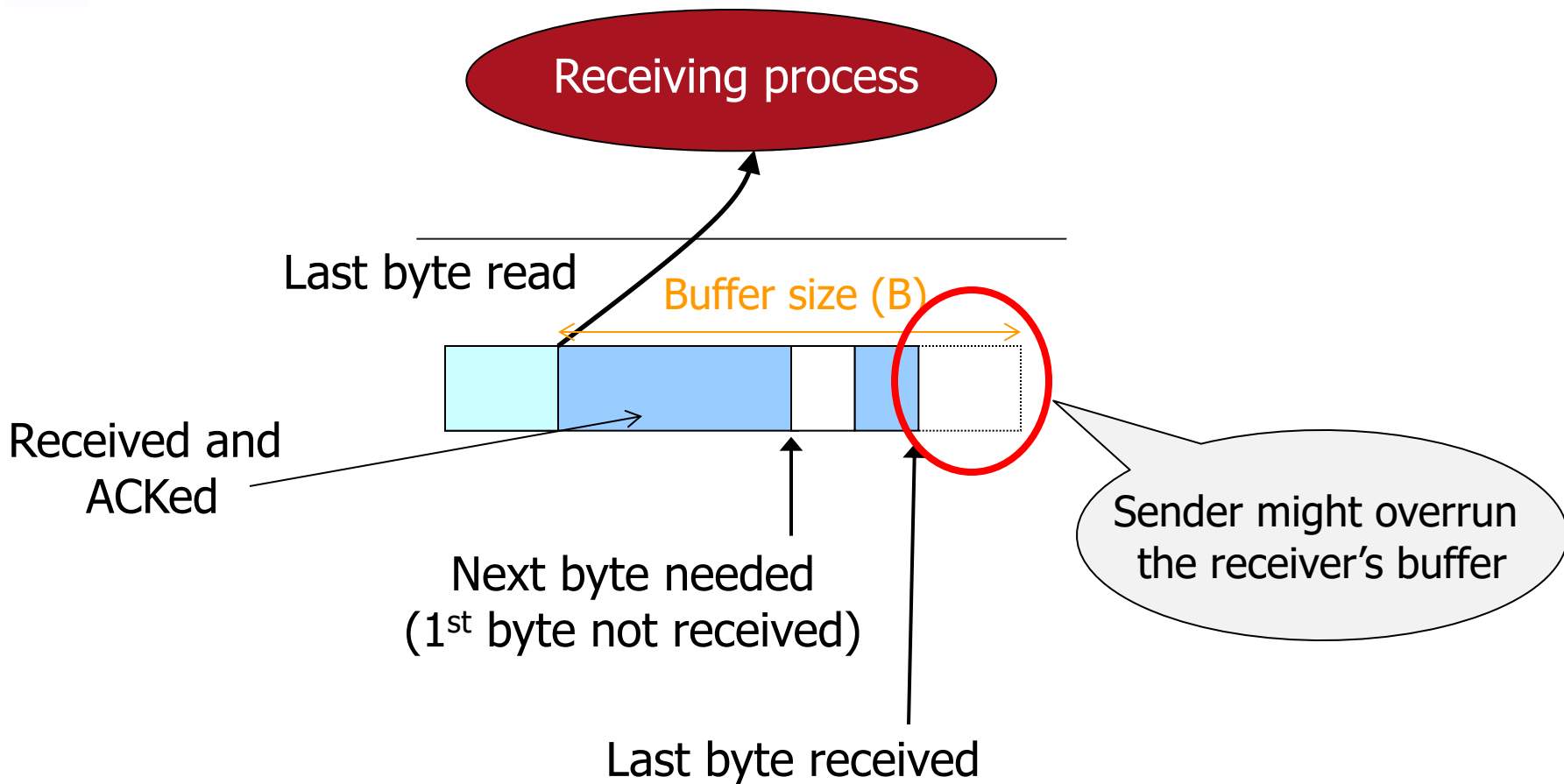
Sliding window at sender



How do we prevent the sender from overflowing the send buffer?



Sliding window at receiver



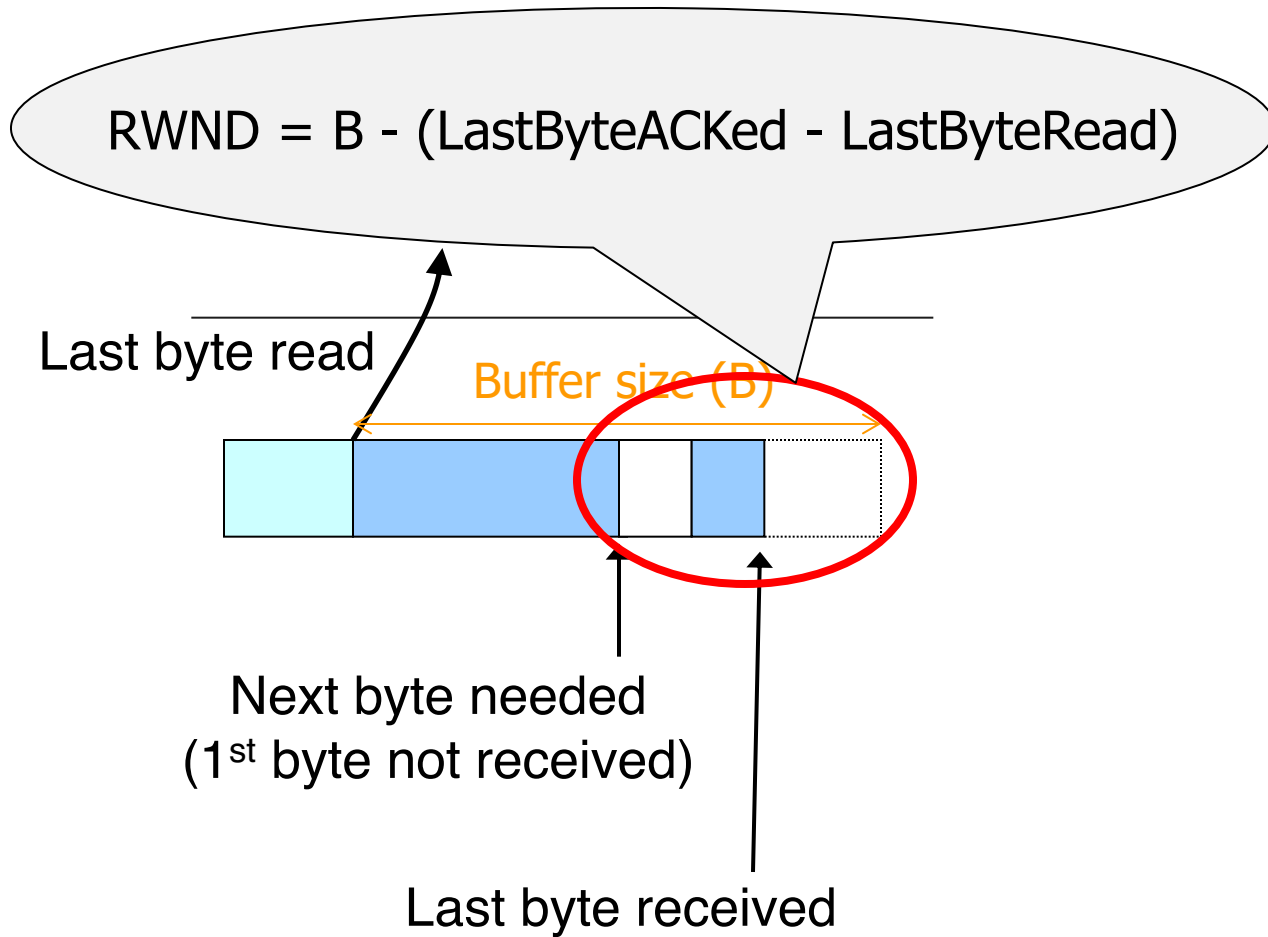


Solution: Flow Control

- Receiver uses an “Advertised Window” (RWND) to prevent sender from overflowing its window
 - Receiver indicates value of RWND in ACKs
 - Sender ensures that the total **number of bytes in flight** \leq RWND

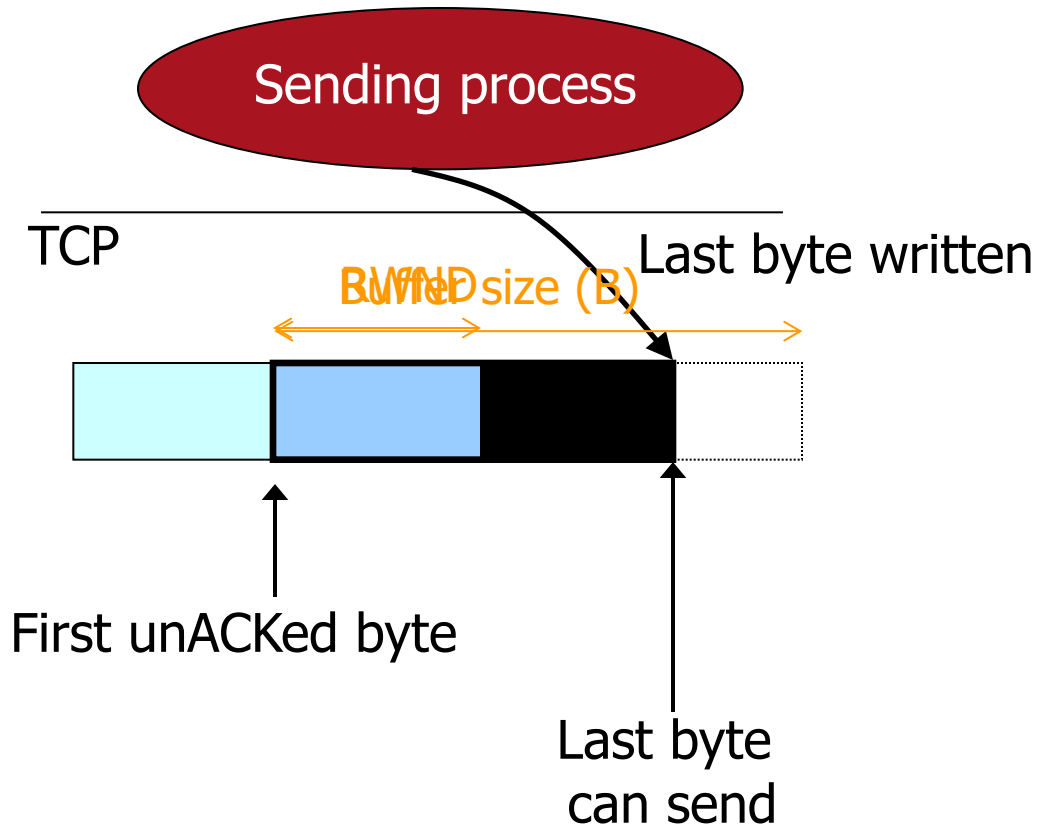


Sliding window at receiver





Sliding window at sender





Sliding window with flow control

- **Sender:** window advances when new data ACK'd
- **Receiver:** window advances as receiving process consumes data
- Receiver advertises to the sender where the receiver window currently ends ("righthand edge")
 - Sender agrees not to exceed this amount
- **UDP does not have flow control**
 - Data can be lost due to buffer overflow



Advertised window limits rate

- Sender can send no faster than $RWND/RTT$ bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- What happens when $RWND=0$?
 - Sender keeps probing with one data bytes
- In original TCP design, that was the sole protocol mechanism controlling sender's rate...



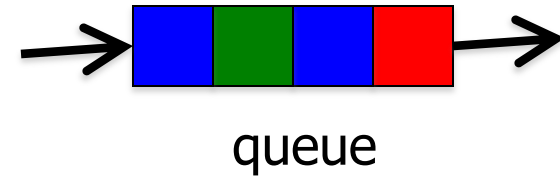
Agenda

- Transmission Control Protocol ✓
 - Reliable in-order delivery ✓
 - Connection-oriented ✓
 - Flow control ✓
 - Congestion control ← NEXT



Congestion

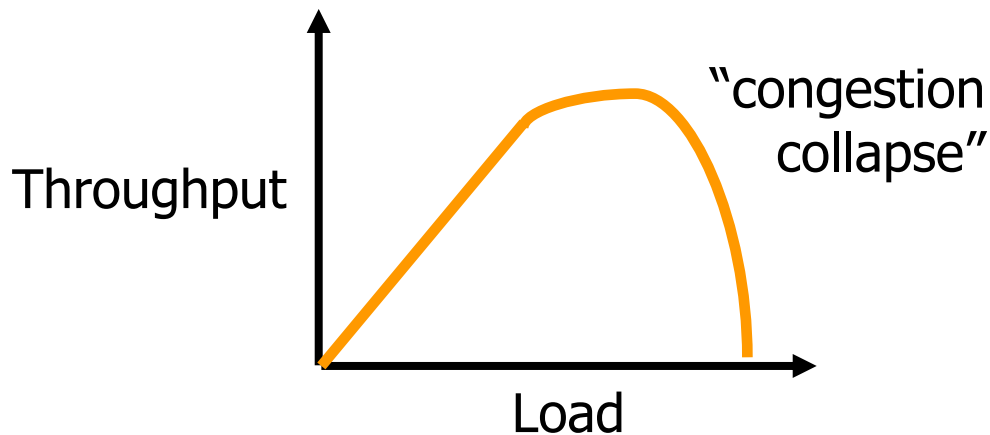
- Best-effort network does not “block” calls
 - So, they can easily become overloaded
 - Congestion == “Load higher than capacity”
- Examples of congestion
 - Link layer: Ethernet frame collisions
 - Network layer: full IP packet buffers
- Excess packets are simply dropped
 - And the sender can simply retransmit





Congestion Collapse

- Easily leads to **congestion collapse**
 - Senders retransmit the lost packets
 - Leading to even greater load
 - ... and even more packet loss



Increase in load that results in a *decrease* in useful work done.



Van Jacobson's solution

- Reuse the sliding window
 - Controls number of packets in flight
- Sending rate \sim Window/RTT
- Vary window size to control sending rate



Windows to keep in mind

- Congestion Window: **CWND**
 - Bytes that can be sent without overflowing routers
 - Computed by sender using congestion control algo.
- Flow control window: **RWND**
 - Bytes that can be sent without overflowing receiver
 - Determined by the receiver and reported to the sender
- Sender-side window = **$\min \{CWND, RWND\}$**
 - Assume for remainder of lecture that $RWND \gg CWND$

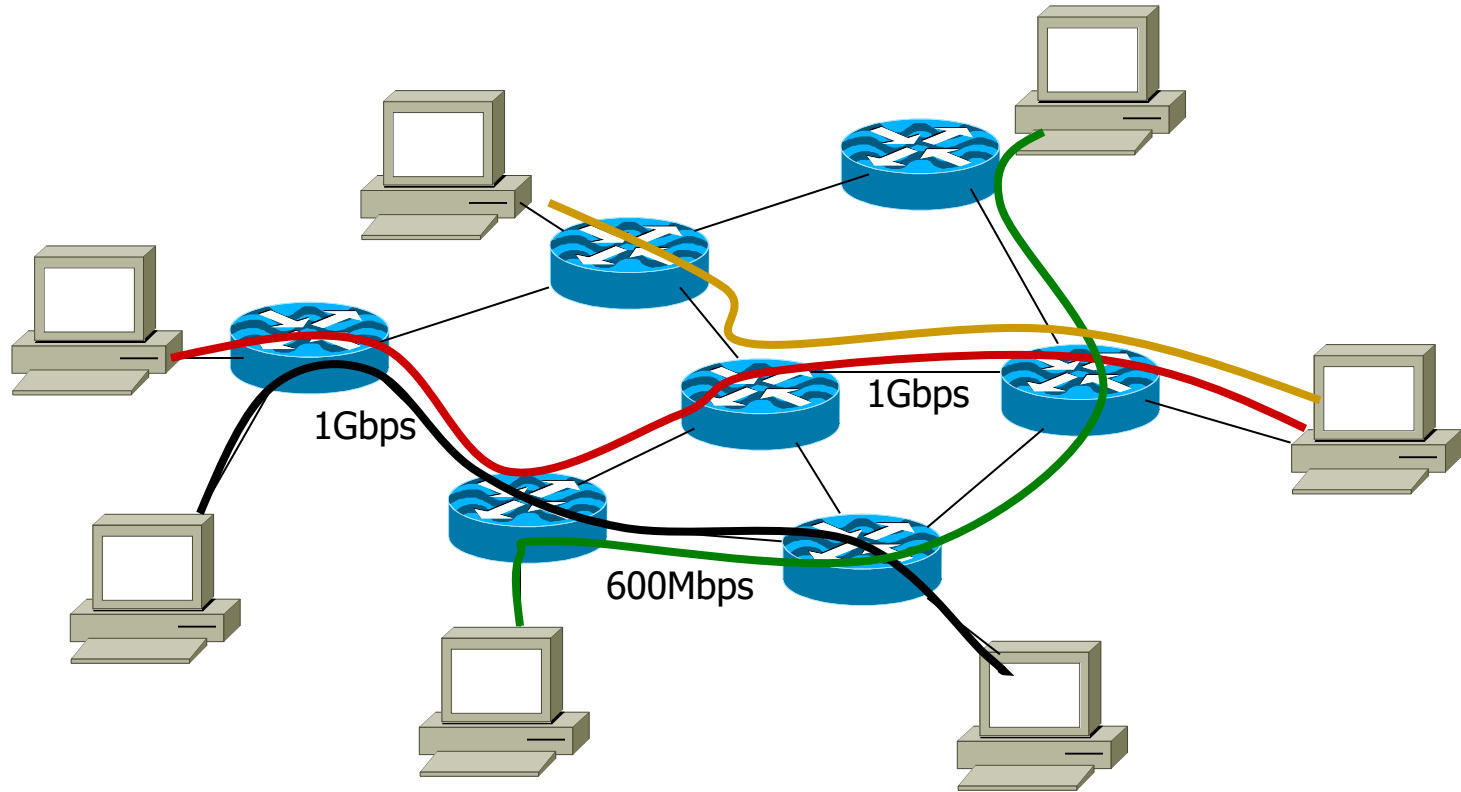


Note

- This lecture talks about CWND in units of MSS
 - MSS (Maximum Segment Size): the amount of payload data in a TCP packet
 - This is only for the simplicity of presentation
- Real implementations maintain CWND in bytes



A complex problem!



Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics



Two Basic Questions



- How does the end host detect congestion?



Detecting congestion

- Packet delays
 - Tricky: noisy signal (delay often varies considerably)
- Routers tell end hosts when they're congested
- Packet loss
 - Fail-safe signal that TCP already has to detect
 - Complications:
 - Non-congestive loss (e.g., checksum errors)
 - Different types of loss



Different types of packet loss

- Duplicate ACKs: isolated loss
 - Still getting ACKs
 - How to tell difference between loss and reordering?
 - Triple duplicate acks
- Timeout: much more serious
 - Must have suffered several losses
- Will adjust rate differently for each case
 - Assume duplicate ACKs for now



Two Basic Questions



- How does the end host detect congestion?
- How fast should the end host send?
 - How can it discover available bandwidth?
 - How should it adjust to changes in available bandwidth?
 - How should it share with other flows?



Responding to Congestion

- Upon detecting congestion
 - Decrease the sending rate
- But, what if conditions change?
 - If more bandwidth becomes available,
 - ... unfortunate to keep sending at a low rate
- Upon *not* detecting congestion
 - Increase sending rate, a little at a time
 - See if packets get through



TCP Rate adjustment

- Basic structure
 - Upon receipt of ACK (of new data): **increase rate**
 - Upon detection of loss: **decrease rate**
- How we increase/decrease the rate depends on the phase of congestion control we're in:
 - Discovering available bottleneck bandwidth vs.
 - Adjusting to bandwidth variations



Bandwidth discovery with “Slow Start”

- Goal: estimate available bandwidth
 - Start slow (for **safety**)
 - Ramp up quickly (for **efficiency**)
- Consider
 - $RTT = 100\text{ms}$, $MSS = 1000\text{bytes}$
 - Window size to fill 1Mbps of BW = 12.5 packets
 - Window size to fill 1Gbps = 12,500 packets
 - Either is possible!



Slow Start phase

- Sender starts at a slow rate, but **increases exponentially** until first loss
- Start with a small congestion window
 - Initially, $CWND = 1$
 - So, initial sending rate is MSS/RTT
- Double the $CWND$ for each RTT with no loss



Slow Start in action

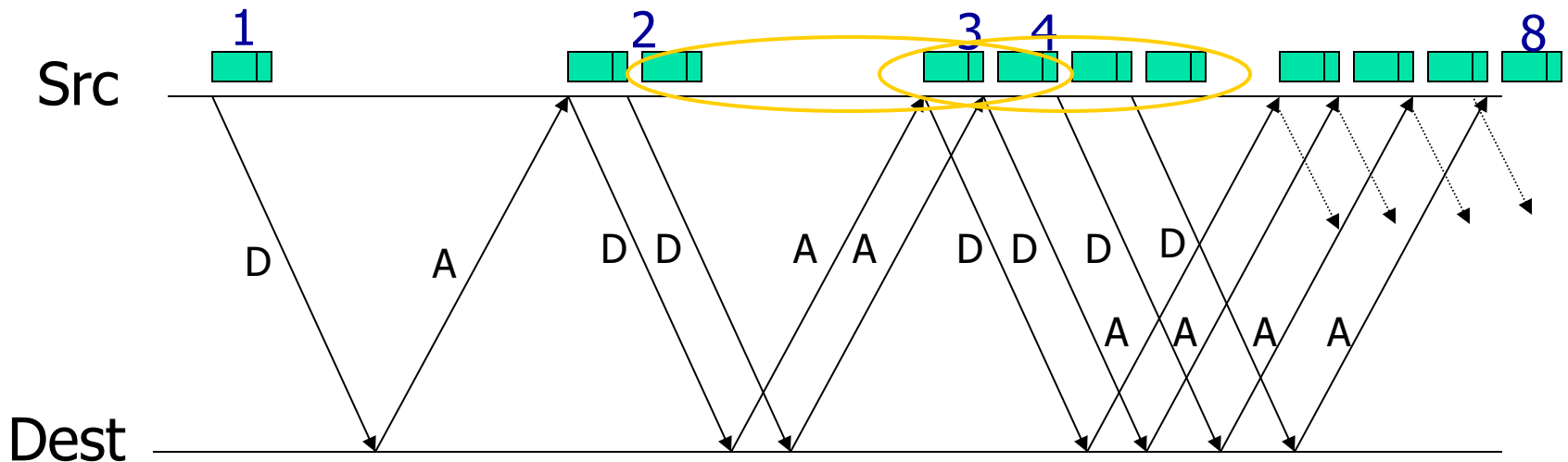
- For each RTT: double CWND
 - i.e., for each ACK, $CWND += 1$

Linear increase per ACK ($CWND+1$) →
exponential increase per RTT ($2 * CWND$)



Slow Start in action

- For each RTT: double CWND
 - i.e., for each ACK, CWND += 1





When does Slow Start stop?

- Slow Start gives an estimate of available bandwidth
 - At some point, there will be loss
- Introduce a “slow start threshold” (**ssthresh**)
 - Initialized to a large value
- If $CWND > ssthresh$, stop Slow Start



Adjusting to varying bandwidth

- $CWND > ssthresh$
 - Stop rapid growth and focus on maintenance
- Now, want to track variations in this available bandwidth, oscillating around its current value
 - Repeated probing (rate increase) and backoff (decrease)



AIMD

- Additive increase
 - For each ACK, $CWND = CWND + 1/CWND$
 - CWND is increased by one only if all segments in a CWND have been acknowledged
- Multiplicative decrease
 - On packet loss, $CWND = CWND/2$



Leads to the TCP "Sawtooth"

Window

