CSE 3231
Computer Networks

Chapter 6
The Transport Layer
part 3 - Flow Control

William Allen, PhD

Spring 2022

End-to-end Issues

TCP needs mechanisms that control the amount of data that the sender transmits to prevent overwhelming the receiver or the network components along the path

- *Congestion control* focuses on preventing too much data from being injected into the network, thereby causing switches or links to become overloaded
- *Flow control* has the goal of preventing senders from overrunning the capacity of the receivers

End-to-end Issues

- To address these issues, TCP uses a *sliding window* algorithm (similar to the Data Link layer)
- Since TCP runs over the Internet rather than a point-to-point link, the following issues need to be addressed by the sliding window algorithm
  - TCP supports *logical connections* between processes that may be simultaneously associated with multiple computers in different networks
  - TCP connections are likely to have *widely different* Round Trip Times
  - Packets may *arrive out of order* at their destination

Sliding Window Revisited

While TCP's version of the sliding window algorithm is similar to the one at the Data Link layer

- it uses acknowledgements to guarantee the reliable delivery of data
- it ensures that data is *delivered* in order

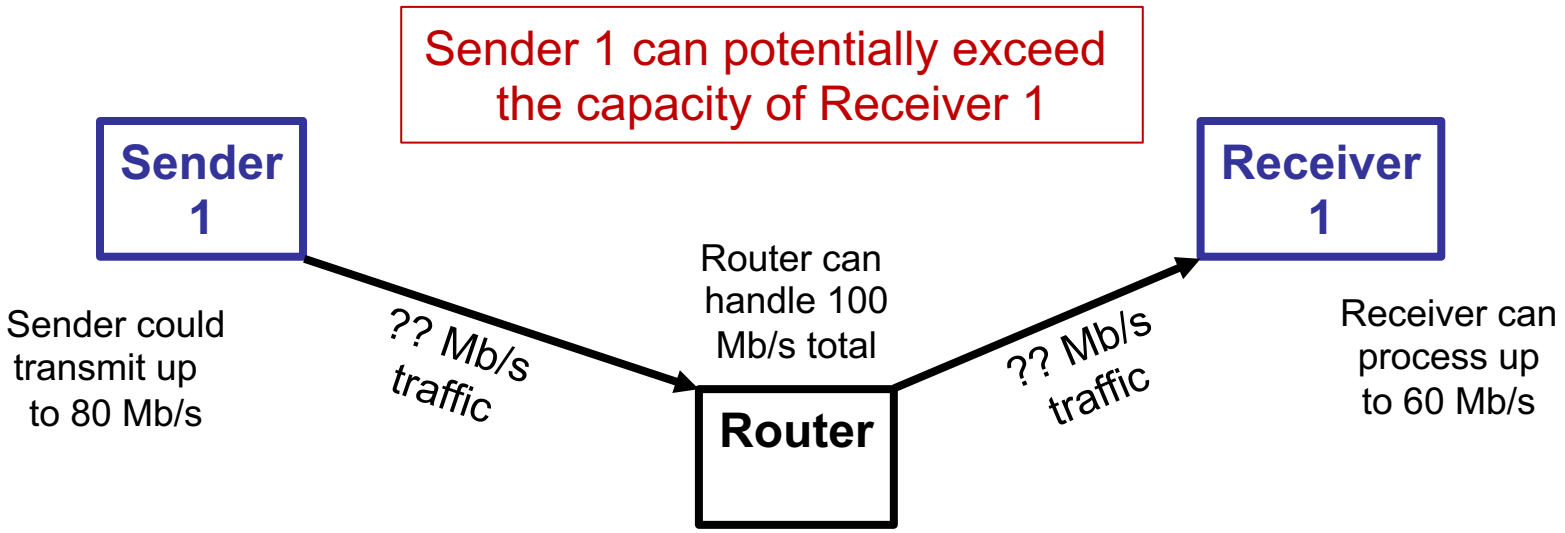
TCP's serves one additional purpose:

- it also enforces flow control between the sender and the receiver

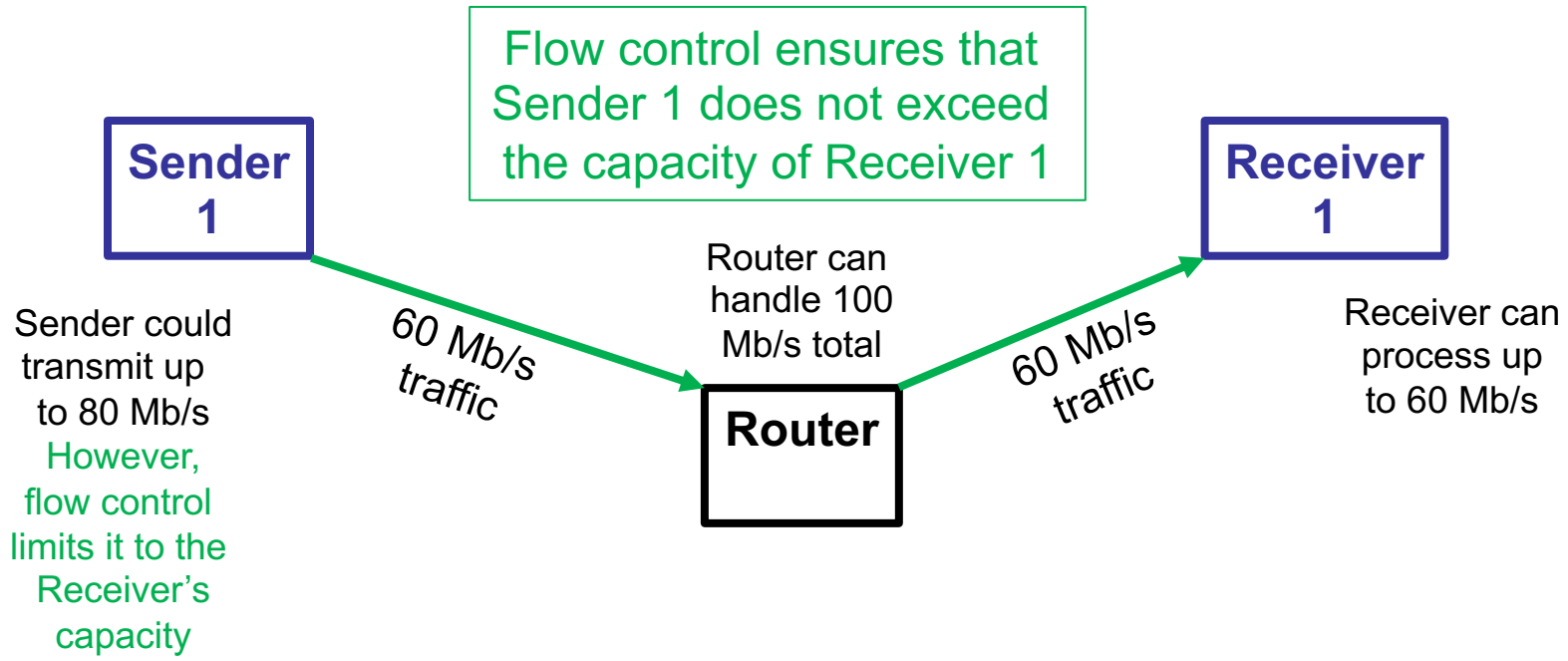
TCP and Flow Control

- TCP's Flow Control manages the flow of packets between the sender and receiver
  - its goal is to prevent senders from transmitting more data than their receivers can accept
  - while it is designed to control the transmission rate of senders in end-to-end flows, it cannot prevent congestion at routers along the path
    - as we saw earlier, routers have to deal with multiple streams of traffic and congestion can occur because the combined rate is too high

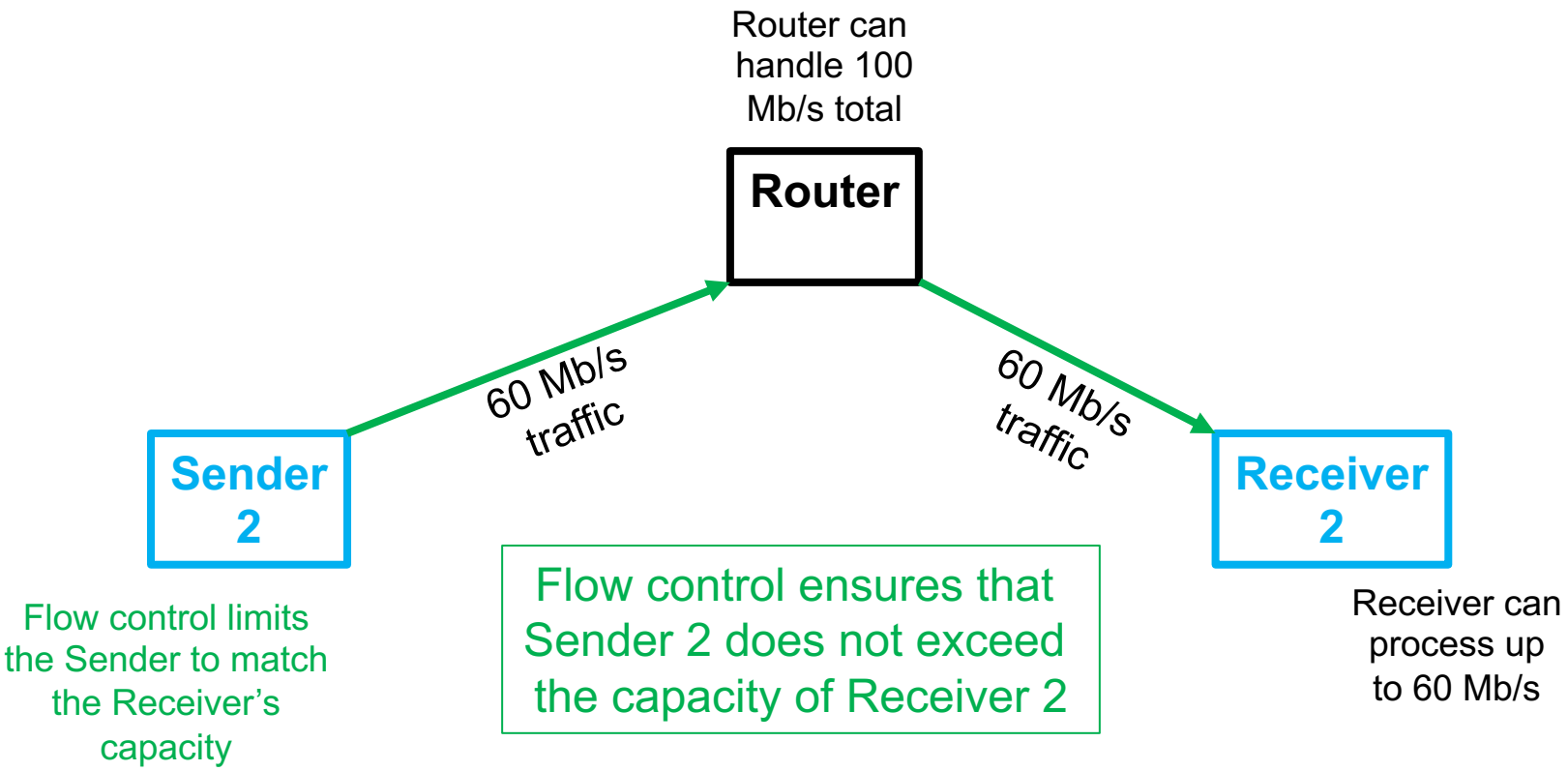
Flow Control vs Congestion



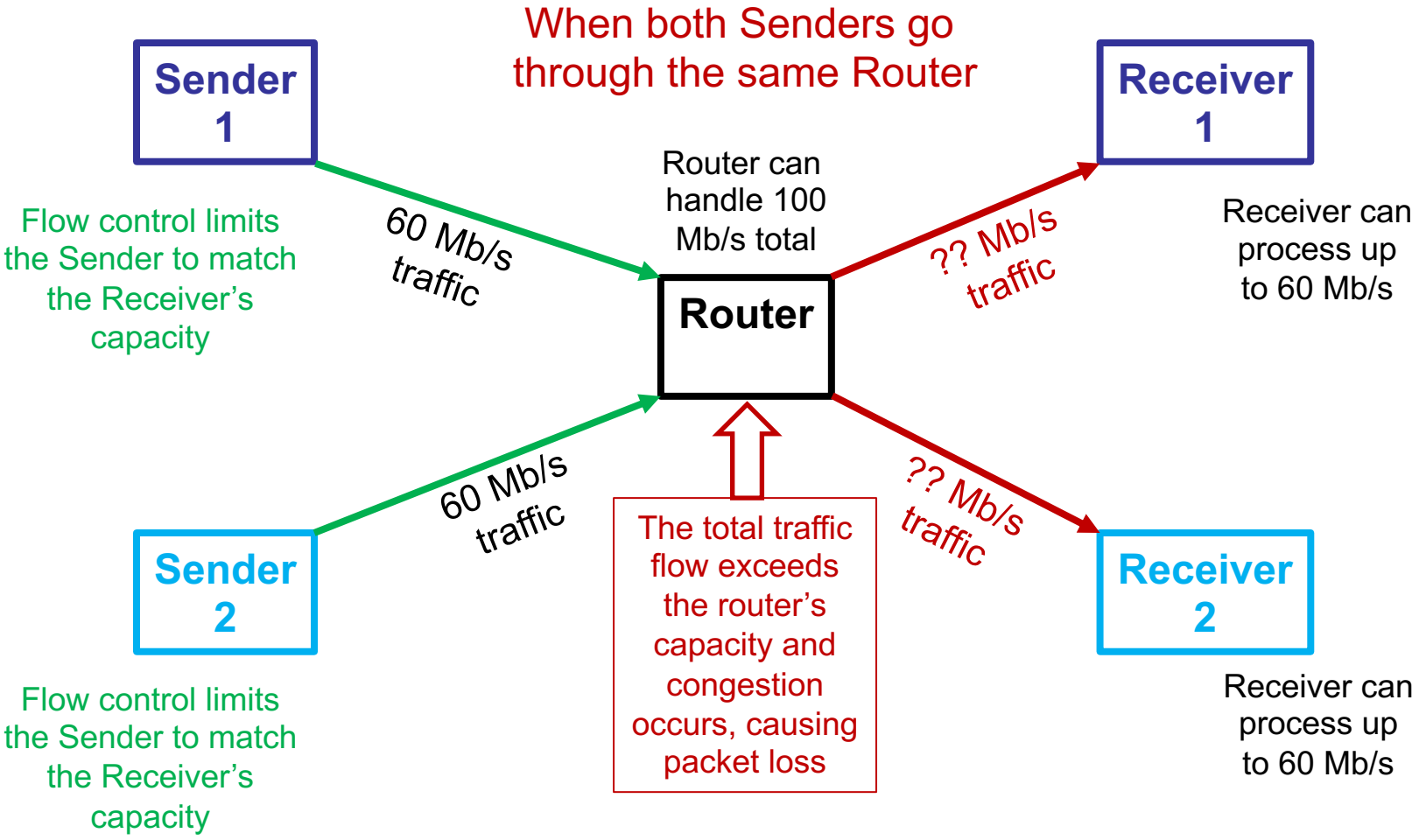
Flow Control vs Congestion



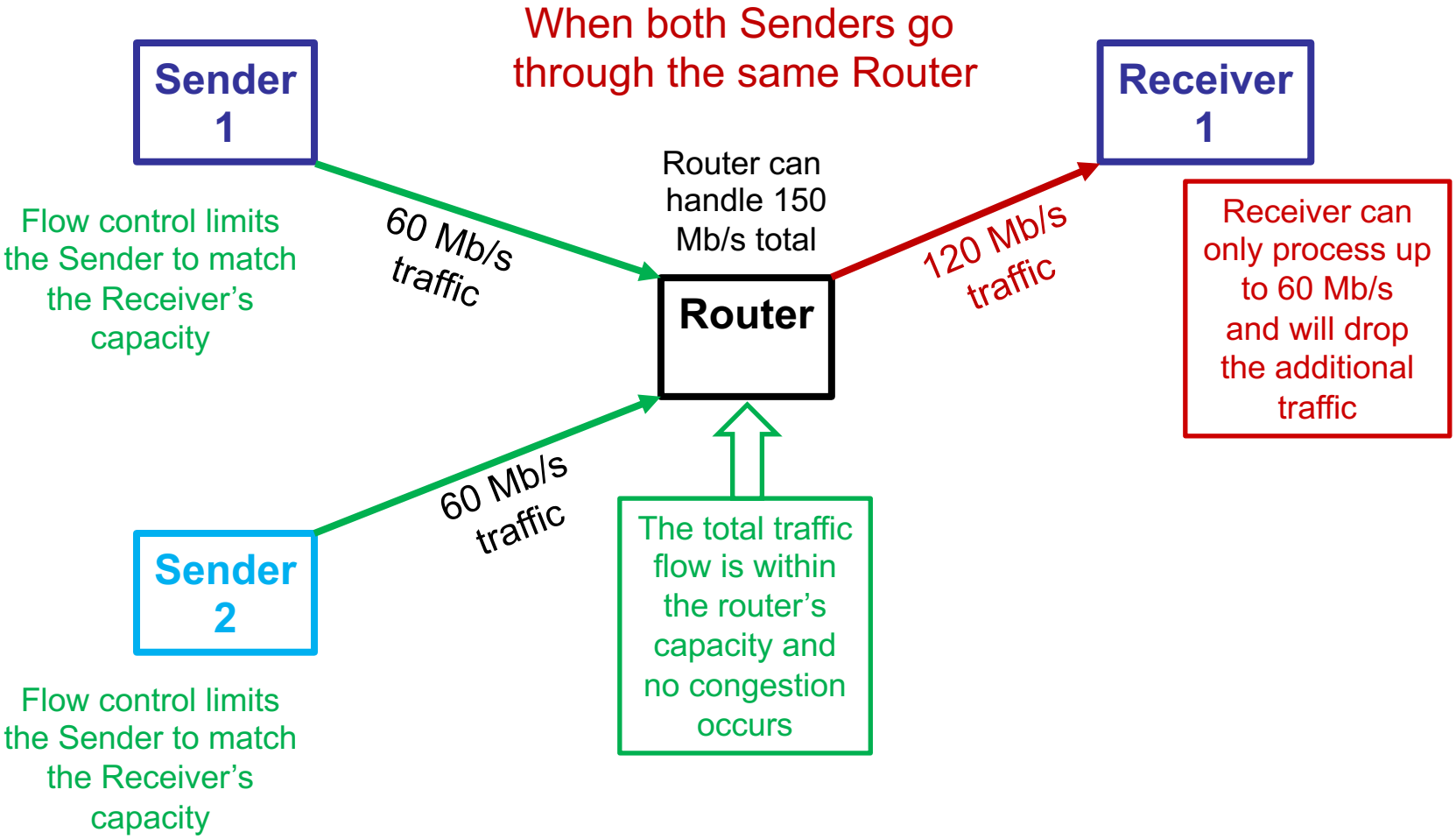
Flow Control vs Congestion



Flow Control vs Congestion



Flow Control vs Congestion



Flow Control vs Congestion

- Congestion can also occur at a receiver because multiple senders can be connected to the same receiver
  - each one may be sending packets at an acceptable rate, but the combination of all of them is too much for the receiver
- Changes in the rate of transmission must be carefully controlled to avoid *oscillations*
  - one slows and another speeds up too quickly and causes congestion, then it slows and...

TCP and Flow Control

- Difference between TCP flow control and Data-Link sliding window:
  - Data-Link layer controls transmission of frames over links between adjacent nodes
    - only one sender can use a link at a time
    - unless there are lost frames, they will all arrive in the order they are sent; acknowledgments and buffering can deal with lost frames
  - TCP deals with the end-to-end flow of packets
    - each receiver can have multiple senders
    - each packet can follow a different path

Bandwidth-Delay Product

- Remember the way we calculated the capacity of a link using the Bandwidth-Delay Product
  - Bandwidth is like the width of the pipe and Delay (latency) is like the length of the pipe
  - With a bandwidth of *100 Mbps* and delay of *50 ms*

    100×10^6 bits/second x 50×10^{-3} seconds = 5×10^6 bits
  - We can transmit 5×10^6 bits before the first bit reaches the other end of the link
  - Thus, we can say this link's *capacity* is 5×10^6 bits
  - Due to the way the capacity is calculated, it is often referred to as the *bandwidth-delay product*

Link Capacity

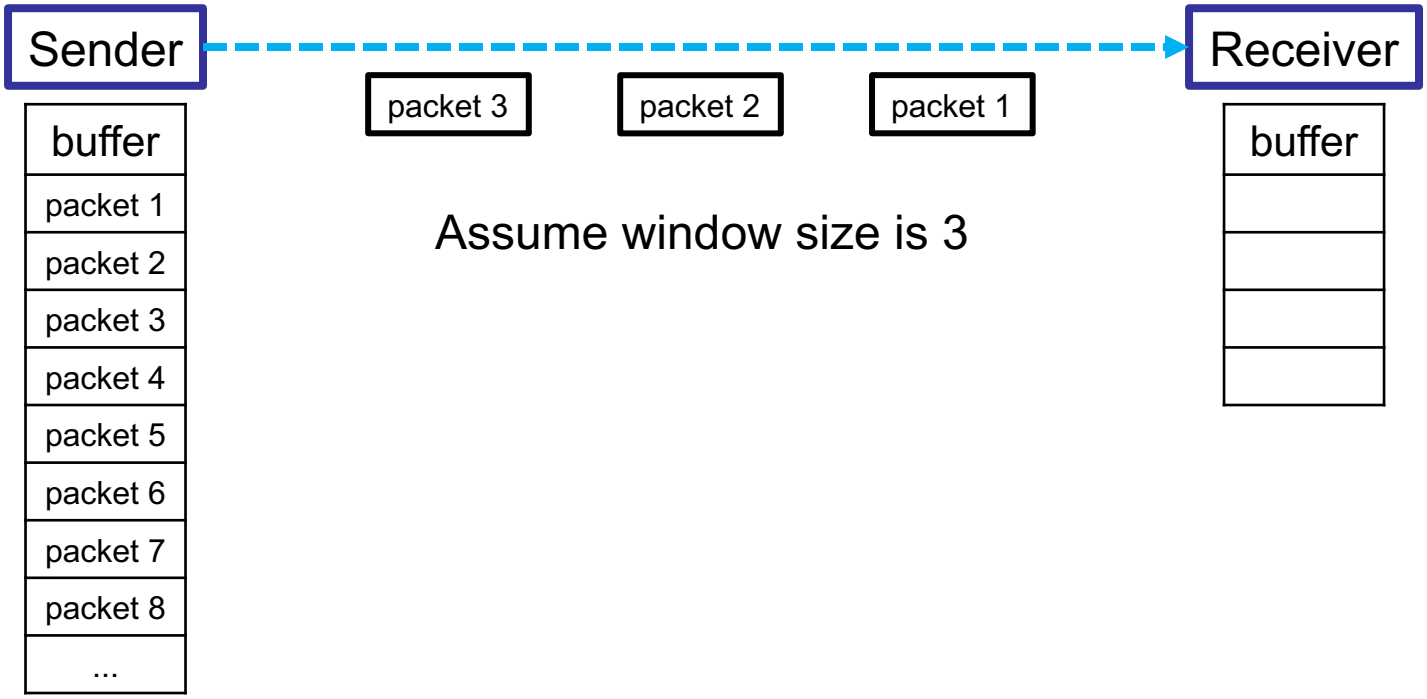
- In that example, the network could hold *5 x 10^6* bits and if packets were ~1500 bits long, the link could hold up to *3,333 packets*!
  - but, it can't really send that many because of the pauses between packets caused by CSMA, etc.
  - also, that won't work across the Internet because lots of other nodes will be transmitting at the same time
- However, it is clear that a node can send a *burst of many packets* before an ACK is received
  - yet, the receiver may not be able to process them as fast as they arrive and its buffer will fill up

Flow Control

- The sender and receiver need to work together to control the rate of transmissions to avoid filling the receiver's buffer and losing packets
  - The receiver has to let the sender know how many packets it can send in a burst
- After the sender transmits that burst of packets, then it has to wait for acknowledgements before it sends any more packets
  - This process is *dynamic*, depending on how busy the receiver is, it may need to slow the rate even more or it may be able to increase the rate

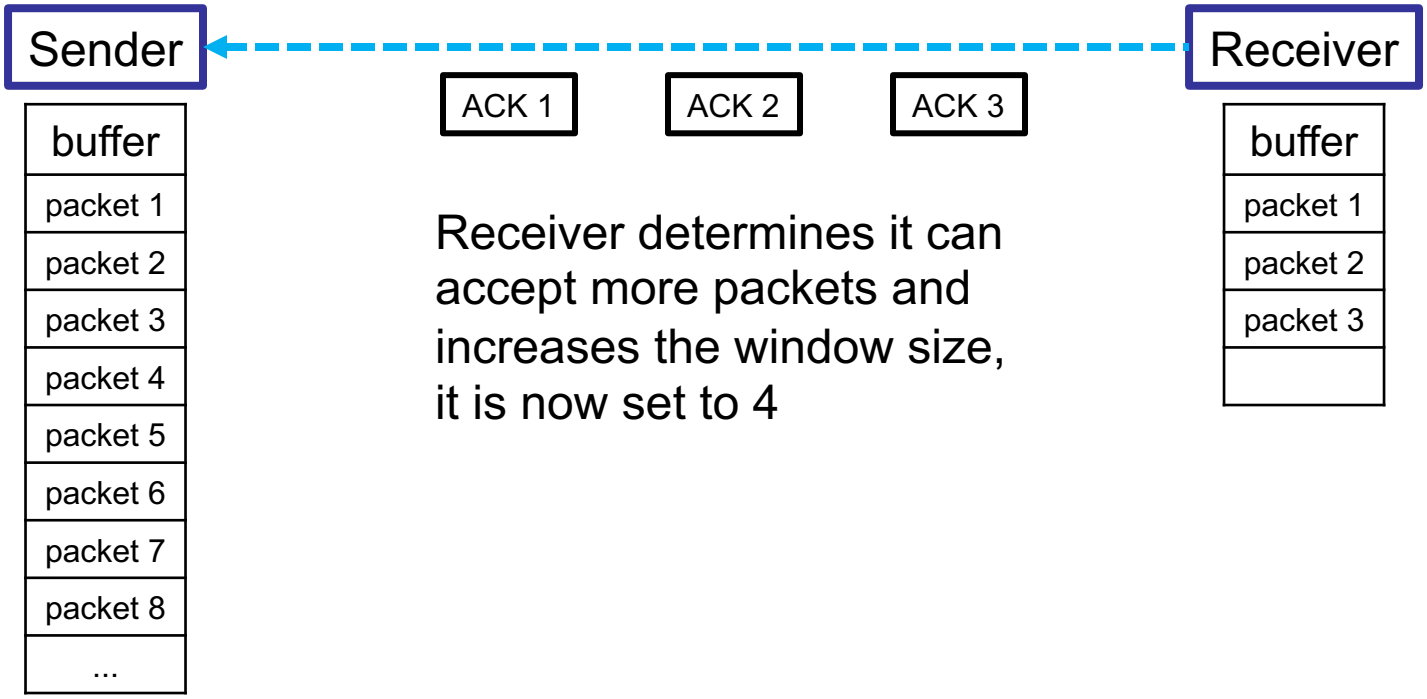
Flow Control

Sender transmits first window



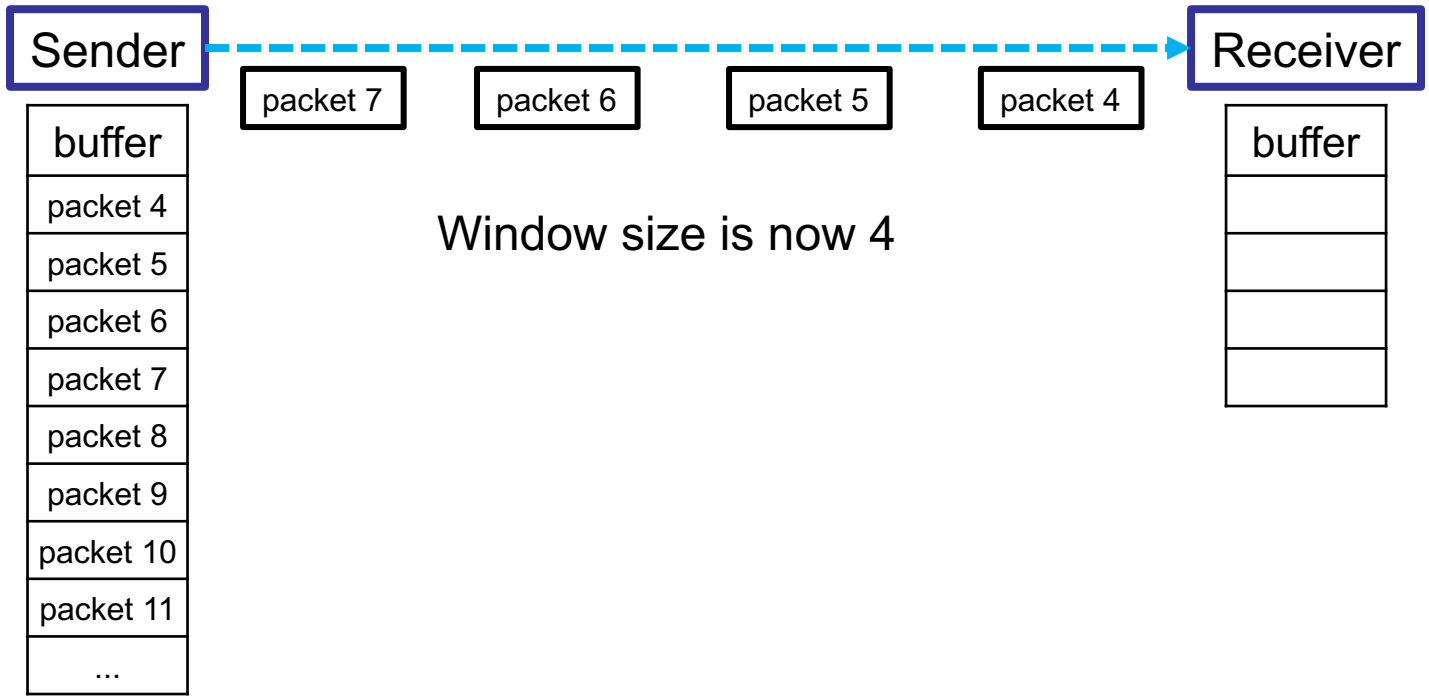
Flow Control

Packets arrive, are checked and ACKs sent

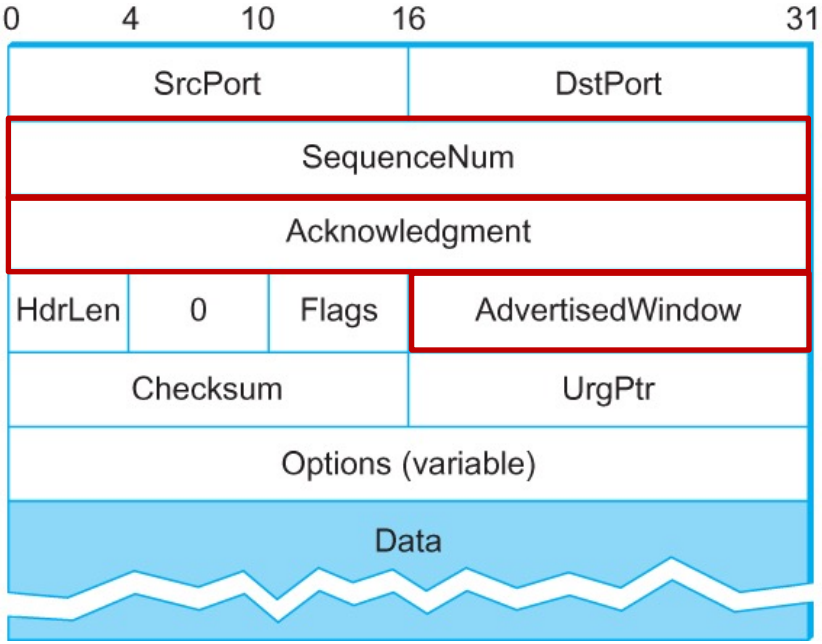


Flow Control

Sender transmits next window



TCP and Flow Control



- Note the following:
  - *SequenceNum*
  - *Acknowledgement*
  - *AdvertisedWindow*
- TCP uses a count of the bytes sent and received to measure the flow of data

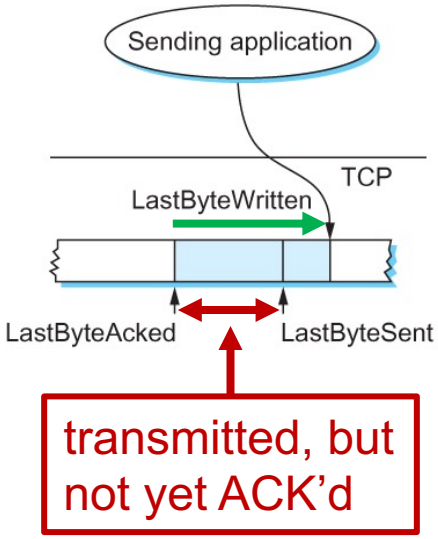
TCP and Flow Control

- *SequenceNum* contains a number that represents the first byte of the segment
  - this information comes from the sender
  - initial value supplied during SYN handshake
- *Acknowledgement* indicates the next sequence number that is *expected*
  - confirms the last byte that has been received
    - *Acknowledgement = LastByteReceived + 1*
  - the ACK flag must also be ON

TCP and Flow Control

Sender uses several variables to monitor the amount of data sent from the buffer (buffer is a queue):

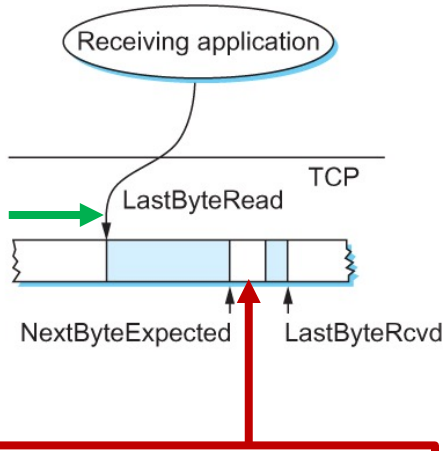
- *LastByteWritten* - most recent byte written into the buffer by the app
- *LastByteSent* - most recent byte transmitted (how much was sent)
- *LastByteAcked* - most recent byte that was acknowledged (don't need to keep bytes < this position)
  - the number of bytes in transit is = *LastByteSent - LastByteAcked*



TCP and Flow Control

Receiver uses variables to monitor the amount of data received and stored in the buffer (buffer is a queue):

- *LastByteRead* - most recent byte read from the buffer by the app
- *NextByteExpected* - all bytes before this byte have been acknowledged
- *LastByteRcvd* - most recent byte received & entered into the buffer
  - bytes available for the application = *NextByteExpected - LastByteRead*



Note the gap after the *NextByteExpected* that shows where a missing packet must be received before the newest bytes can be acknowledged

TCP and Flow Control

AdvertisedWindow is used by the receiver to indicate how much data it can handle

- it measures capacity in bytes, not packets
- this limits how many unacknowledged bytes can be *in transit* at a time
  - i.e., how many packets can be sent before an acknowledgement arrives at the sender?
- the receiver determines this value based on its buffer size and can adjust it if needed
  - packets have to stay in the buffer until they are acknowledged, how many can it hold?

TCP and Flow Control

- How is the *AdvertisedWindow* used?
  - if the receiving node determines that it needs to adjust the window size, it calculates a new value for the *AdvertisedWindow* and sends that updated value to the sender
  - after receiving the new *AdvertisedWindow*, the sender controls how many bytes it can send by conforming to this equation:

$$AdvertisedWindow \geq (LastByteSent - LastByteAcked)$$

(we saw earlier that this is the number of bytes in transit)

TCP Congestion Control

- TCP congestion control was introduced into the Internet in the late 1980s by Van Jacobson, roughly eight years after the TCP/IP protocol stack had become operational.
- Immediately preceding this time, the Internet was suffering from congestion collapse—
  - hosts would send packets into the Internet as fast as the advertised window would allow, congestion would occur at some router (causing dropped packets), and the hosts would time out and retransmit their packets, resulting in even more congestion

TCP Congestion Control

- The idea of TCP congestion control is for each source to determine how much capacity is available *in the network*, so that it knows how many bytes it can safely have in transit.
  - Once a given source has this many bytes in transit, the sender uses the arrival of an ACK as a signal that one of its packets has left the network, and that it is therefore safe to insert a new packet into the network without adding to the level of congestion.
  - By using ACKs to pace the transmission of packets, TCP is said to be *self-clocking.*

TCP Congestion Control

- TCP maintains a new state variable for each connection, called *CongestionWindow*, which is used by the source to limit *how many bytes* it is allowed to have in transit at a given time.
- The congestion window is congestion control's counterpart to flow control's *AdvertisedWindow*.
- To implement congestion control, TCP was modified such that the maximum number of bytes of *unacknowledged data* allowed is now:

  Minimum(*CongestionWindow, AdvertisedWindow*)

TCP Congestion Control

- Unlike the *AdvertisedWindow*, which is sent by the receiving side of the connection, neither node can generate the *CongestionWindow* for the sending side of a TCP connection
  - neither end can predict when congestion will occur
- Then, how does TCP determine the correct value for *CongestionWindow*?
- It has to be learned by trial and error:

  increase the traffic until you discover the point where congestion occurs

TCP Congestion Control

The TCP source sets the *CongestionWindow* based on the level of congestion it *perceives* to exist in the network.

- This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down
- Taken together, the mechanism is called *Additive Increase / Multiplicative Decrease* (abbreviated AIMD)

TCP Congestion Control

How does the source determine that the network is congested enough to slow transmission?

- In modern networks, packets are rarely dropped because of an error, the main reason is that a packet was dropped is *due to congestion*, which will result in a timeout at the sending node
- TCP interprets these timeouts as a sign of congestion and reduces the rate at which it is transmitting
- Specifically, each time a timeout occurs, the source sets *CongestionWindow* to *one half of its previous value* and this represents the "*multiplicative decrease*"

TCP Congestion Control

Although *CongestionWindow* is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets.

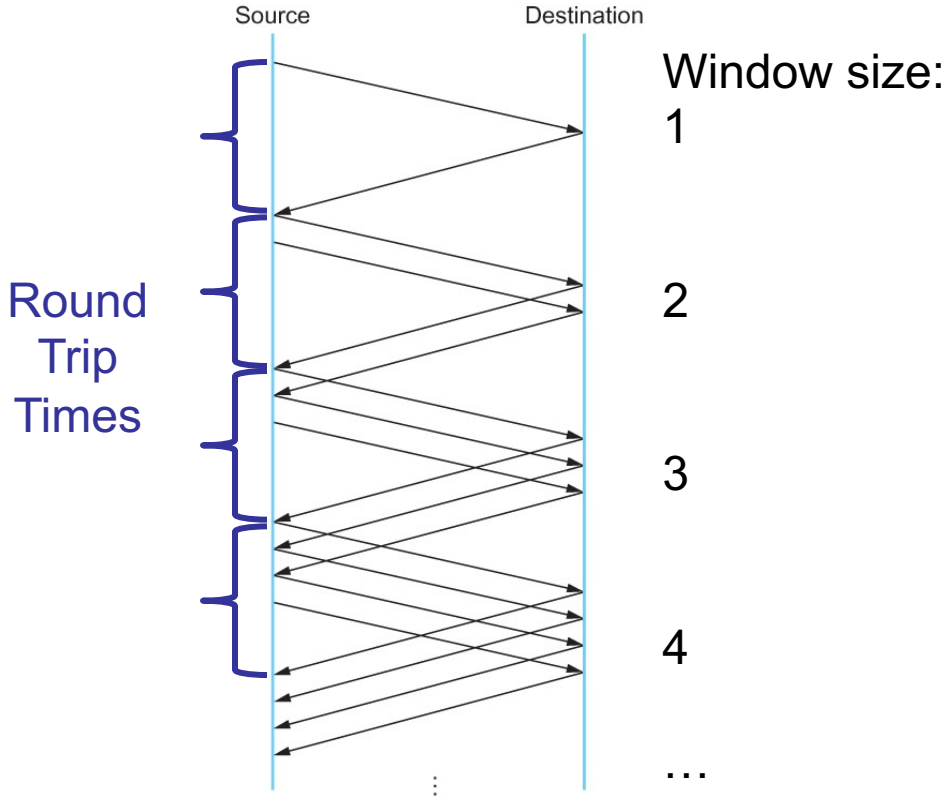
- For example, suppose the *CongestionWindow* is set to 16 packets.
- If a loss is detected, *CongestionWindow* is reduced to 8
- Additional losses will cause *CongestionWindow* to be reduced to 4, then 2, and finally to 1 packet
- *CongestionWindow* is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size (MSS)*

TCP Congestion Control

- We also need to be able to increase the congestion window to take advantage of newly available capacity in the network
- This is the "*additive increase*" part of AIMD, and it works as follows:
  - Every time the source successfully sends a *CongestionWindow*'s worth of packets it adds the equivalent of 1 packet to *CongestionWindow*
  - Success is measured by one ACK per RTT
  - Thus, the increase is slower than the decrease and avoids too rapid an increase in transmission rate

TCP Congestion Control

Additive Increase, Multiplicative Decrease



Transmission rate growing slowly during *additive increase*, with one packet being added each RTT.

TCP Maximum Segment Size

- TCP has a Maximum Segment Size (MSS) that is used to calculate the number of bytes to increase the *CongestionWindow*
  - MSS is negotiated during the TCP connection handshake and remains constant for the connection
  - MSS is calculated to be *smaller* than the MTU, to prevent fragmentation of packets
  - Thus, messages that are larger than the MSS are divided into segments *before transmission*, where each segment is in a separate IP packet

MSS = MTU – (IP header size) – (TCP header size)

TCP Maximum Segment Size

- TCP documentation specifies a default MSS of 536 bytes
- A larger MSS can be used if the endpoints can determine that the minimum MTU is larger than 576 bytes (536=576-20-20)
  - For example, If MTU is 1000 bytes, then MSS = 1000-20-20 = 960
- MSS negotiations use the TCP Options fields

MSS and Additive Increase

- If MSS=960 and *CongestionWindow*=960
  - Send one packet with size = MSS
  - Receive the first ACK:
    - Increment = MSS×(MSS/CongestionWindow)
      - Increment = 960×(960/960) = 960×1
    - CongestionWindow = 960+960 = 1920 (2*MSS)
  - Then, for the next two ACKs:
    - Increment = 960×(960/1920) = 960×1/2 = 480
    - CongestionWindow += 1920+480 = 2400
    - CongestionWindow += 2400+480 = 2880 (3*MSS)

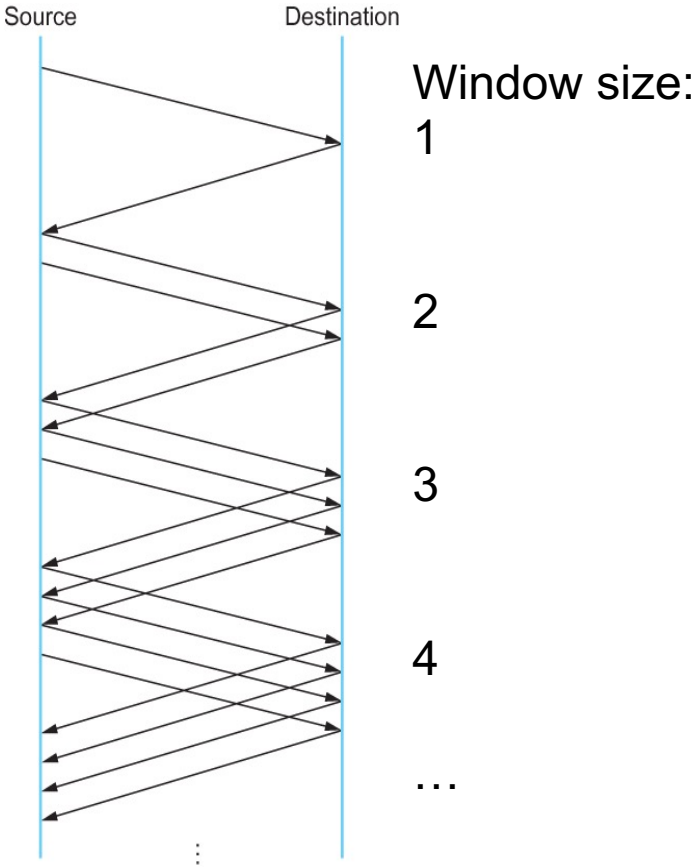
TCP Congestion - Slow Start

- The additive increase mechanism just described is the right approach to use when the source is operating close to the available capacity of the network, but it takes too long to ramp up a connection when it is *starting from scratch*
- TCP provides a second mechanism, ironically called *slow start,* that is used to increase the congestion window rapidly from a cold start
- Slow start effectively increases the congestion window exponentially, rather than linearly

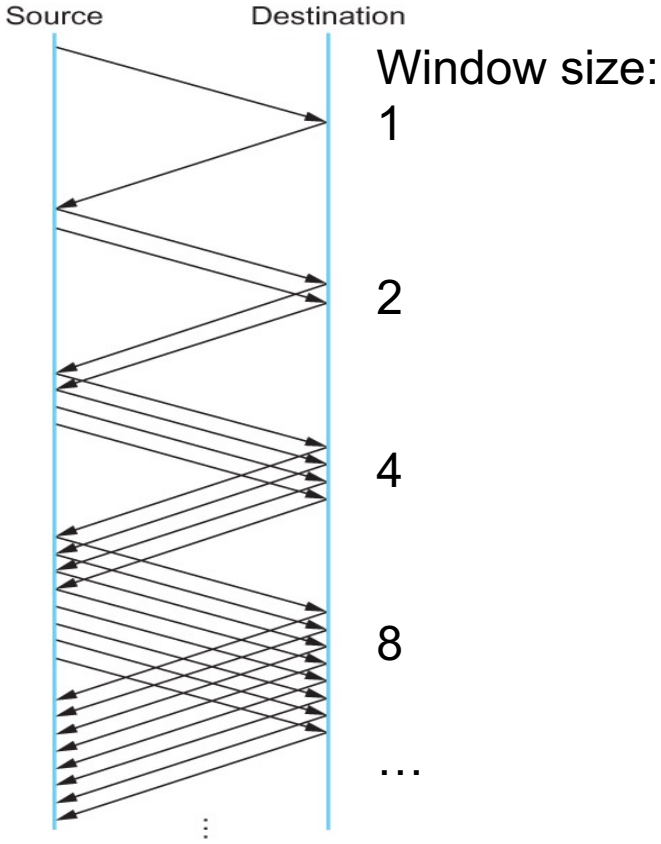
TCP Congestion - Slow Start

- Specifically, the source starts out by setting *CongestionWindow* to one packet
- When this packet's ACK arrives, TCP adds 1 to *CongestionWindow* and then sends two packets
- Upon receiving the corresponding two ACKs, TCP increments *CongestionWindow* by 2—*one for each ACK*—and sends four packets
- The end result is that TCP effectively doubles the number of packets in transit every RTT

TCP Congestion - Slow Start



Packets in transit during **additive increase**



Packets in transit during **slow start**

MSS and Slow Start

- In terms of bytes instead of packets, we use MSS=960 and *CongestionWindow*=960
  - Send one packet with size = MSS
    - Receive the first ACK and increase congestion window by one MSS (960+960) (2*MSS)
  - For each subsequent ACK, increase the congestion window by one MSS, until a *preset threshold* is reached, then switch to additive increase, multiplicative decrease (AIMD)
    - Where AIMD increased for each *group of ACKs*, slow start increases for each individual ACK

MSS and Slow Start

- Start: window = 960, send one segment
- 1st ACK arrives, window increases to 1920, two segments are transmitted
- 2 ACKs arrive, window *increases* by 2*960 to 3,840 and 4 segments are transmitted
- 4 ACKs arrive, window *increases* by 4*960 to 7,680 and 8 segments are transmitted
- If SlowStartThreshold=8, change to AIMD