TCP - Part II

Relates to Lab 5. This is an extended module that covers TCP data transport, and flow control, congestion control, and error control in TCP.

Interactive and bulk data transfer

TCP applications can be put into the following categories

bulk data transfer - ftp, mail, http

interactive data transfer - telnet, rlogin

TCP has heuristics to deal these application types.

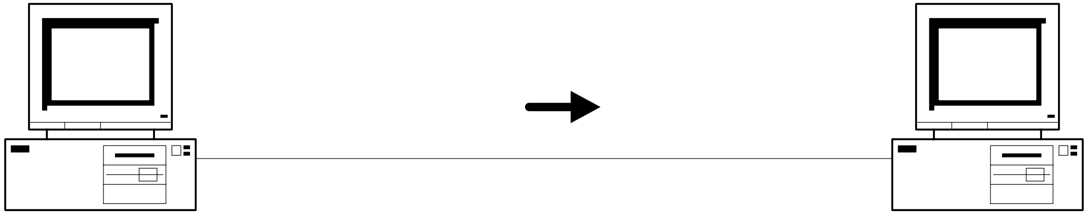
For interactive data transfer:

- Try to reduce the number of packets

For bulk data transfer:

Telnet session on a local network

Telnet session
from Argon
to Neon



Argon.cs.virginia.edu *Neon.cs.virginia.edu*

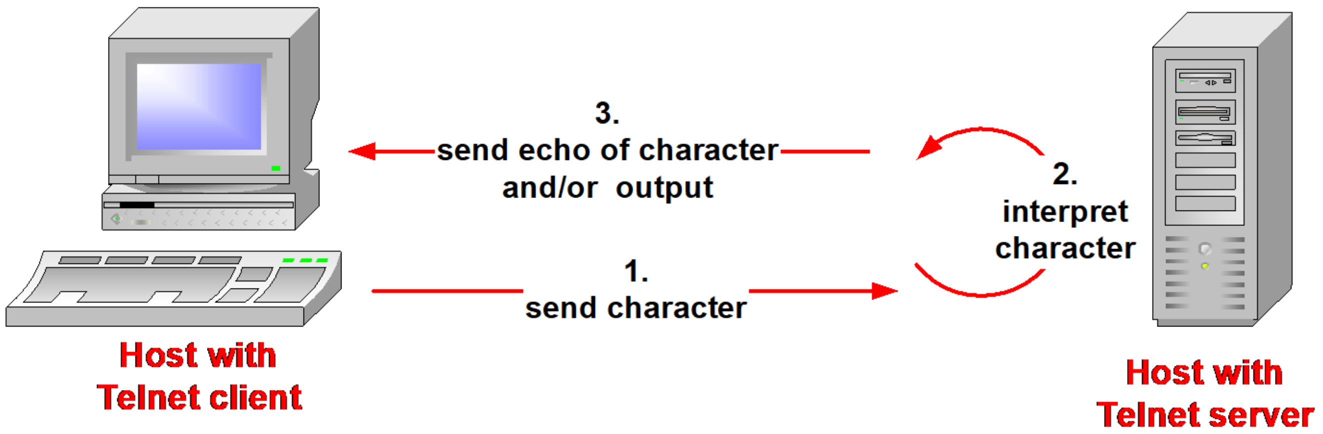
- This is the output of typing 3 (three) characters :

Time 44.062449: Argon → Neon: Push, SeqNo 0:1(1), AckNo 1
Time 44.063317: Neon → Argon: Push, SeqNo 1:2(1), AckNo 1
Time 44.182705: Argon → Neon: No Data, AckNo 2

Time 48.946471: Argon → Neon: Push, SeqNo 1:2(1), AckNo 2
Time 48.947326: Neon → Argon: Push, SeqNo 2:3(1), AckNo 2
Time 48.982786: Argon → Neon: No Data, AckNo 3

Time 55.116581: Argon → Neon: Push, SeqNo 2:3(1) AckNo 3
Time 55.117497: Neon → Argon: Push, SeqNo 3:4(1) AckNo 3
Time 55.183694: Argon → Neon: No Data, AckNo 4

Interactive applications: Telnet

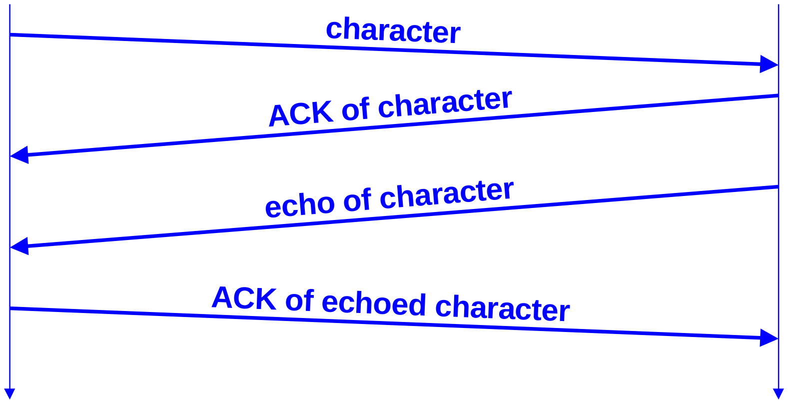


- Remote terminal applications (e.g., Telnet) send characters to a server. The server interprets the character and sends the output at the server to the client.

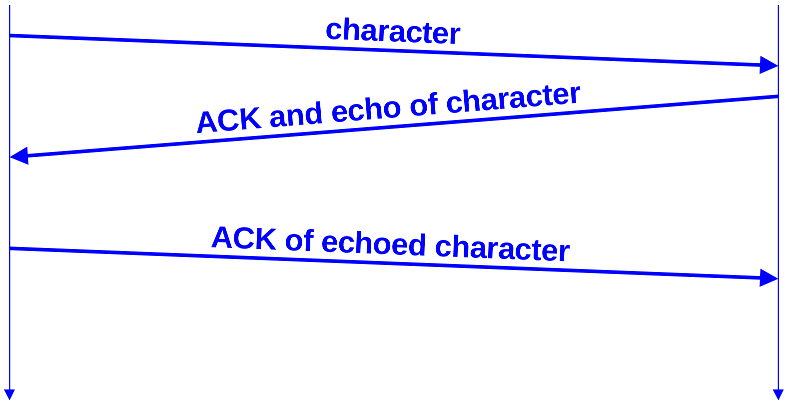
- For each character typed, you see three packets:
  1. **Client → Server:** Send typed character
  2. **Server → Client:** Echo of character (or user output) and acknowledgement for first packet
  3. **Client → Server:** Acknowledgement for second packet

Why 3 packets per character?

- We would expect four packets per character:



- However, tcpdump shows this pattern:



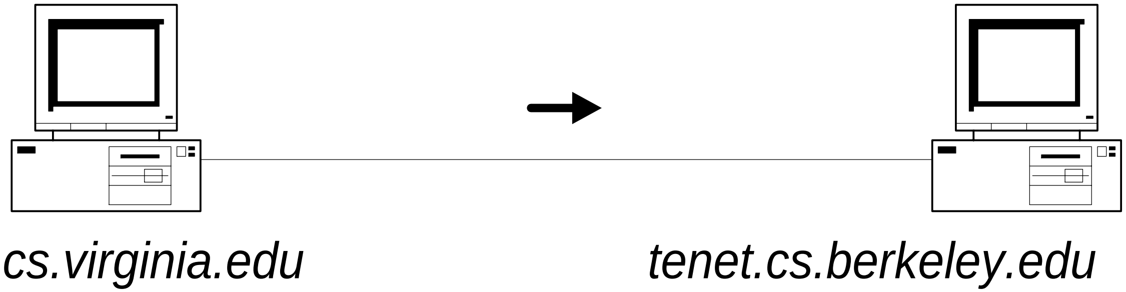
- What has happened?
TCP has delayed the transmission of an ACK

Delayed Acknowledgement

- TCP delays transmission of ACKs for up to 200ms
- The hope is to have data ready in that time frame. Then, the ACK can be piggybacked with a data segment.
- Delayed ACKs explain why the ACK and the “echo of character” are sent in the same segment.

Telnet session to a distant host

Telnet session
between argon.cs.virginia.edu
and
tenet.cs.berkeley.edu

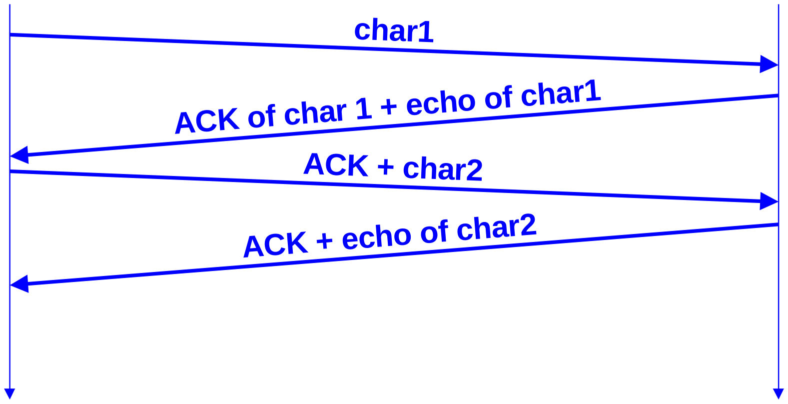


- This is the output of typing nine characters :

Time 16.401963:	**Argon → Tenet:**	**Push, SeqNo 1:2(1), AckNo 2**
Time 16.481929:	**Tenet → Argon:**	**Push, SeqNo 2:3(1) , AckNo 2**
Time 16.482154:	**Argon → Tenet:**	**Push, SeqNo 2:3(1) , AckNo 3**
Time 16.559447:	**Tenet → Argon:**	**Push, SeqNo 3:4(1), AckNo 3**
Time 16.559684:	**Argon → Tenet:**	**Push, SeqNo 3:4(1), AckNo 4**
Time 16.640508:	**Tenet → Argon:**	**Push, SeqNo 4:5(1) AckNo 4**
Time 16.640761:	**Argon → Tenet:**	**Push, SeqNo 4:8(4) AckNo 5**
Time 16.728402:	**Tenet → Argon:**	**Push, SeqNo 5:9(4) AckNo 8**

Observation 1

- **Observation:** Transmission of segments follows a different pattern, i.e., there are only two packets per character typed



- The delayed acknowled-gment does not kick in
- The reason is that there is always data at Argon ready to sent when the ACK arrives.

Observation 2

- **Observation:**
  - Argon never has multiple unacknowledged segments outstanding
  - There are fewer transmissions than there are characters.

- This is due to **Nagle's Algorithm:**

  Each TCP connection can have only one small (1-byte) segment outstanding that has not been acknowledged.

- **Implementation:** Send one byte and buffer all subsequent bytes until acknowledgement is received.Then send all buffered bytes in a single segment**.** (Only enforced if byte is arriving from application one byte at a time)

- Nagle's algorithm reduces the amount of small segments.
- The algorithm can be disabled.

TCP: Flow Control Congestion Control Error Control

What is Flow/Congestion/Error Control ?

- **Flow Control:** Algorithms to prevent that the sender overruns the receiver with information?
- **Congestion Control:** Algorithms to prevent that the sender overloads the network
- **Error Control:** Algorithms to recover or conceal the effects from packet losses

→ The goal of each of the control mechanisms are different.

→ But the implementation is combined

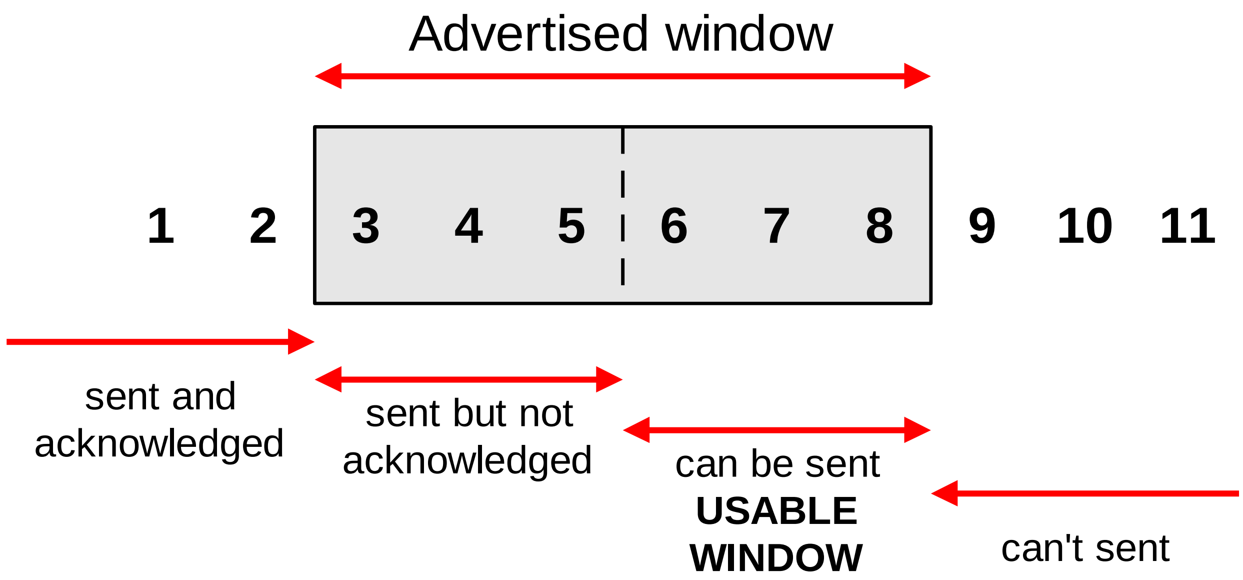
TCP Flow Control

TCP Flow Control

- TCP implements sliding window flow control
    - Sending acknowledgements is separated from setting the window size at sender.
    - Acknowledgements do not automatically increase the window size
    - Acknowledgements are cumulative

Sliding Window Flow Control

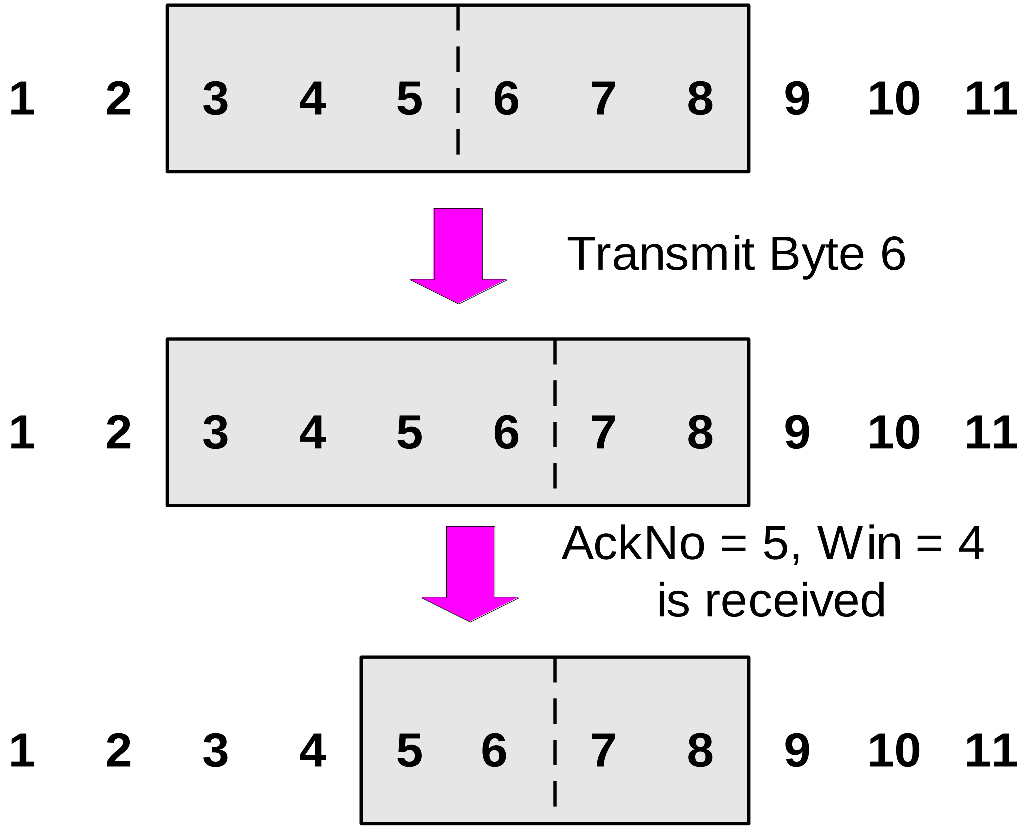
- Sliding Window Protocol is performed at the byte level:



- Here: Sender can transmit sequence numbers 6,7,8.

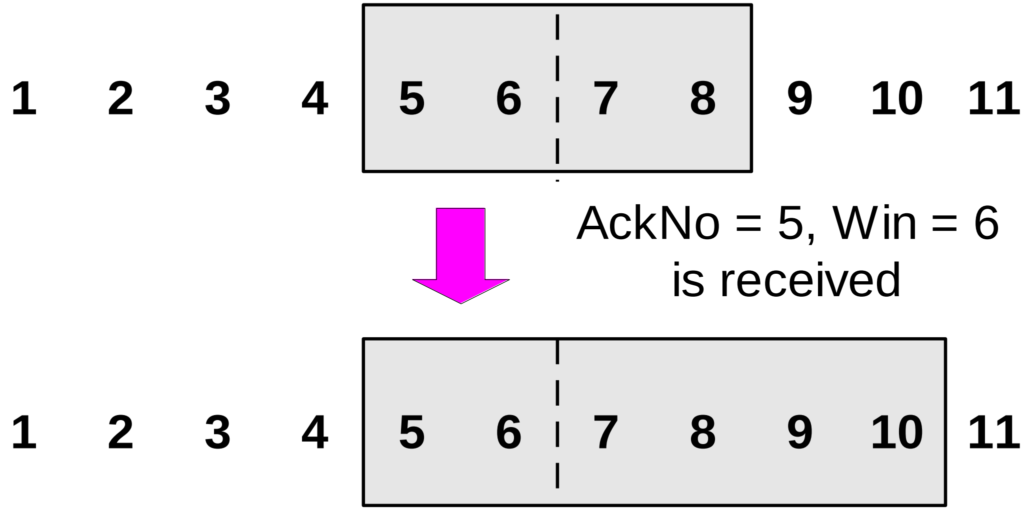
Sliding Window: "Window Closes"

- Transmission of a single byte (with SeqNo = 6) and acknowledgement is received (AckNo = 5, Win=4):



Sliding Window: "Window Opens"

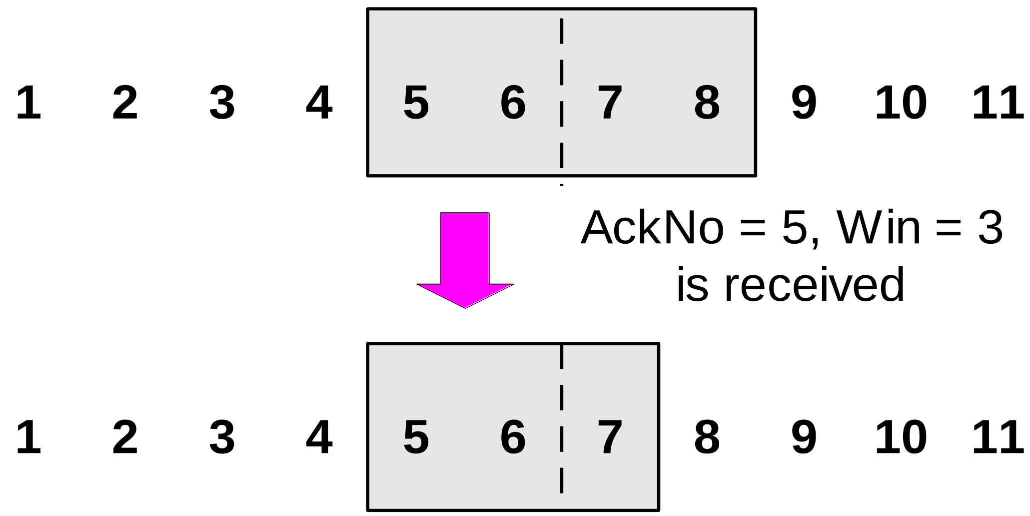
• Acknowledgement is received that enlarges the window to the right (AckNo = 5, Win=6):



• A receiver opens a window when TCP buffer empties (meaning that data is delivered to the application).

Sliding Window: "Window Shrinks"

- Acknowledgement is received that reduces the window from the right (AckNo = 5, Win=3):



- Shrinking a window should not be used

Window Management in TCP

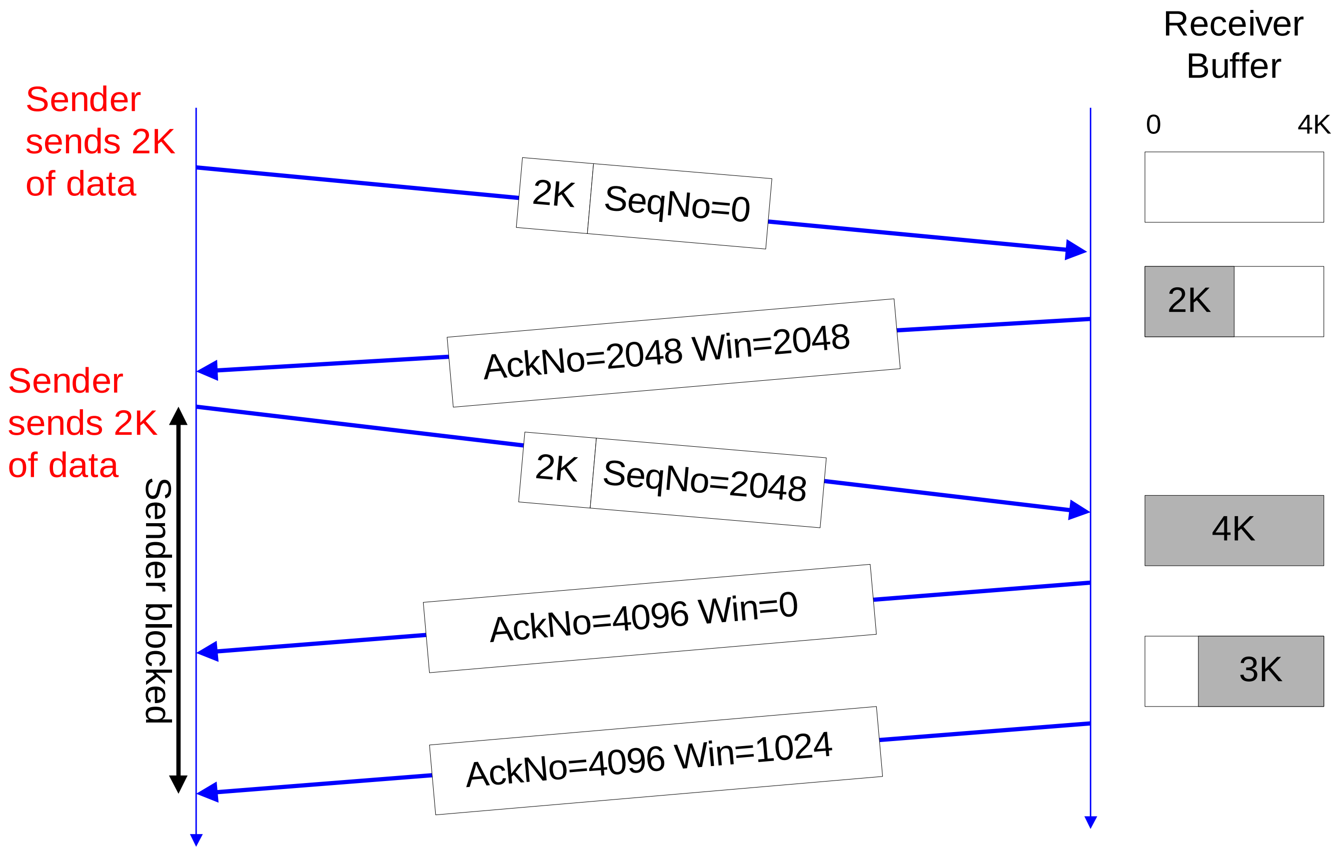
- The receiver is returning two parameters to the sender

AckNo	window size (win)
32 bits	16 bits

- The interpretation is:
  - **I am ready to receive new data with**
    SeqNo= AckNo, AckNo+1, …., AckNo+Win-1

- Receiver can acknowledge data without opening the window
- Receiver can change the window size without acknowledging data

Sliding Window: Example



TCP Congestion Control

TCP Congestion Control

- TCP has a mechanism for congestion control. The mechanism is implemented at the sender

- The sender has two parameters:
  - **Congestion Window** (**cwnd**)

  - **Slow-start threshhold Value** (**ssthresh)**

    Initial value is the advertised window size

- Congestion control works in two modes:
  - **slow start** (cwnd < ssthresh)
  - **congestion avoidance** (cwnd >= ssthresh

Slow Start

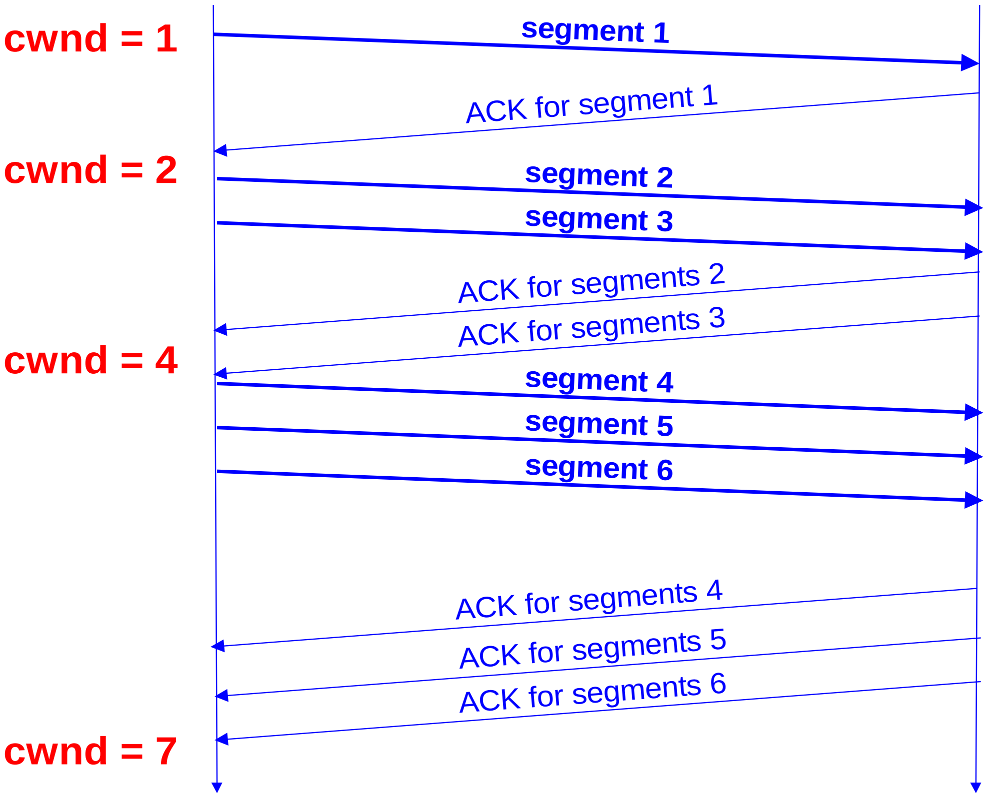
- Initial value: **Set cwnd = 1**
  - » Note: Unit is a segment size. TCP actually is based on bytes and increments by 1 MSS (maximum segment size)
- The receiver sends an acknowledgement (ACK) for each packet
  - » Note: Generally, a TCP receiver sends an ACK for every other segment.
- Each time an ACK is received by the sender, the congestion window is increased by 1 segment:

  cwnd = cwnd + 1

  - » If an ACK acknowledges two segments, cwnd is still increased by only 1 segment.
  - » Even if ACK acknowledges a segment that is smaller than MSS bytes long, cwnd is increased by 1.
- Does Slow Start increment slowly? Not really.
  In fact, the increase of cwnd is exponential

Slow Start Example

- The congestion window size grows very rapidly
  - For every ACK, we increase cwnd by 1 irrespective of the number of segments ACK'ed
- TCP slows down the increase of *cwnd* when ***cwnd > ssthresh***



Congestion Avoidance

- Congestion avoidance phase is started if cwnd has reached the slow-start threshold value

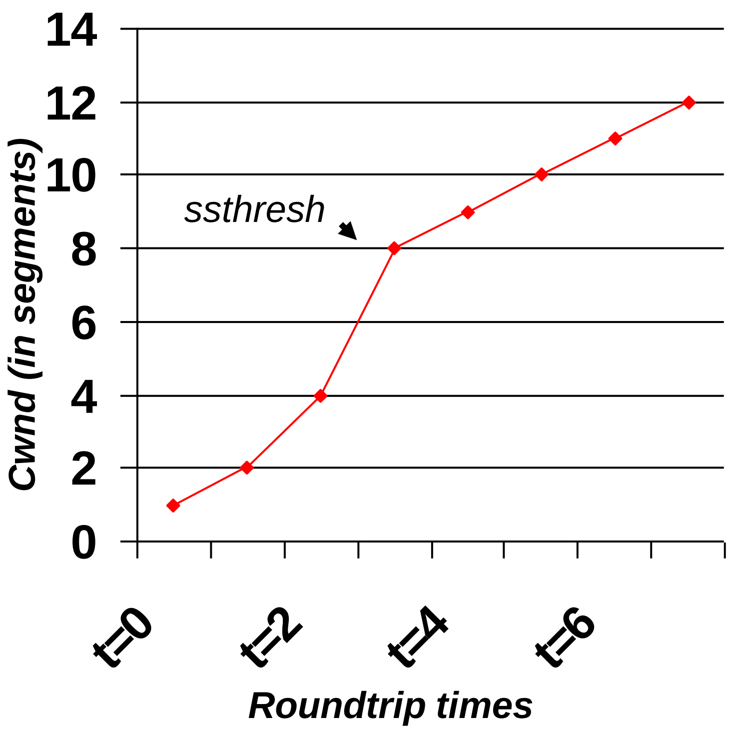
- If cwnd >= ssthresh then each time an ACK is received, increment cwnd as follows:
    - cwnd = cwnd + 1/ [cwnd]

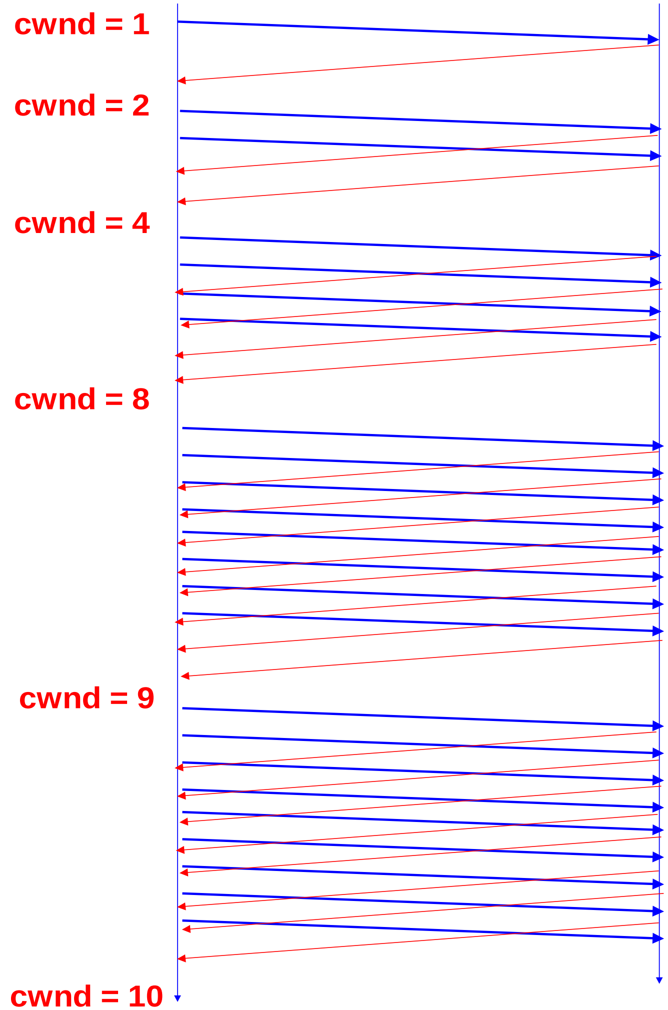
Where [cwnd] is the largest integer smaller than cwnd

- So *cwnd* is increased by one only if all *cwnd* segments have been acknowledged.

Example of Slow Start/Congestion Avoidance

Assume that *ssthresh = 8*





Responses to Congestion

- So, TCP assumes there is congestion if it detects a packet loss
- A TCP sender can detect lost packets via:
  - Timeout of a retransmission timer
  - Receipt of a duplicate ACK

- TCP interprets a Timeout as a binary congestion signal. When a timeout occurs, the sender performs:
  - cwnd is reset to one:

    cwnd = 1
  - ssthresh is set to half the current size of the congestion window:

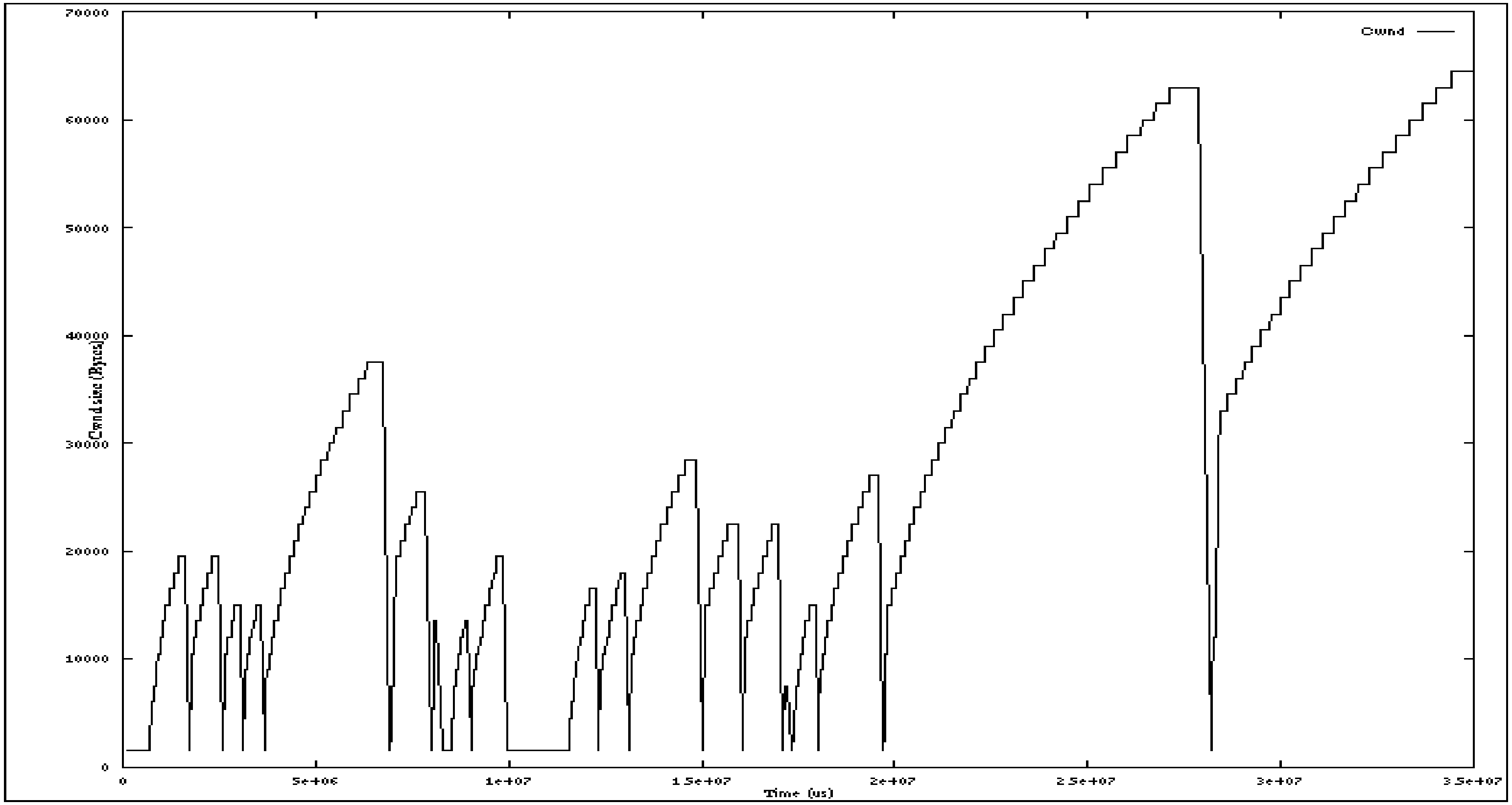
    ssthressh = cwnd / 2
  - and slow-start is entered

Summary of TCP congestion control

```
Initially:
    cwnd = 1;
    ssthresh =
          advertised window size;
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;
```

Slow Start / Congestion Avoidance

- A typical plot of cwnd for a TCP connection (MSS = 1500 bytes) with TCP Tahoe:



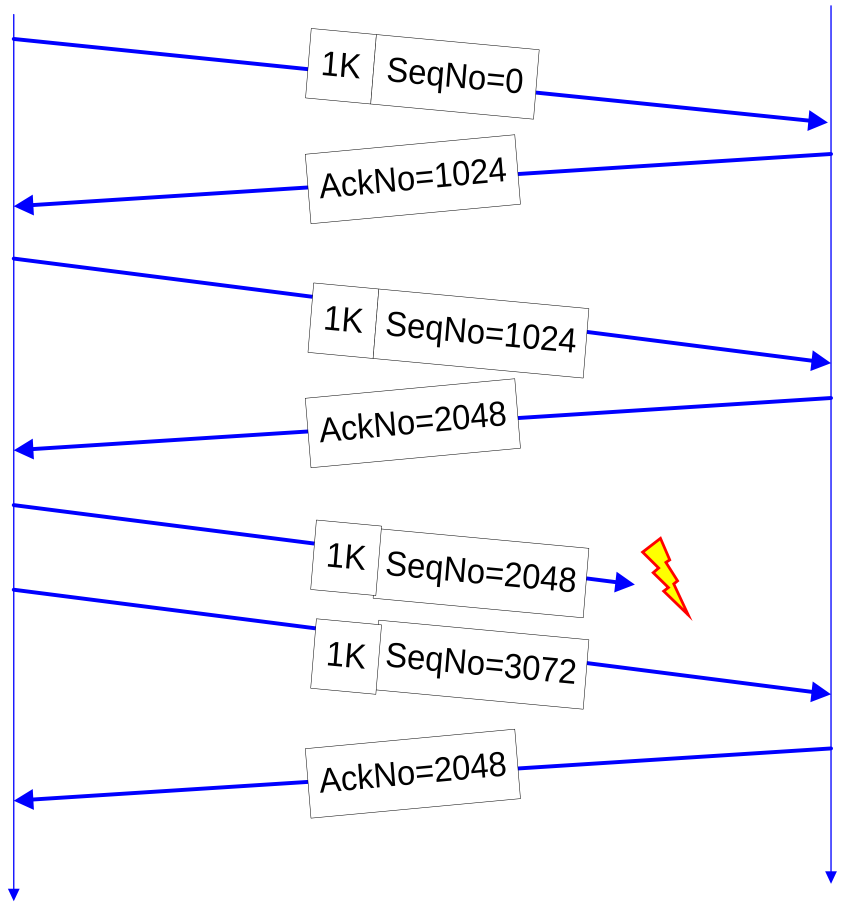
Flavors of TCP Congestion Control

- **TCP Tahoe** (1988, FreeBSD 4.3 Tahoe)
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit
- **TCP Reno** (1990, FreeBSD 4.3 Reno)
  - Fast Recovery
- **New Reno** (1996)
- **SACK** (1996)

- **RED** (Floyd and Jacobson 1993)

Acknowledgments in TCP

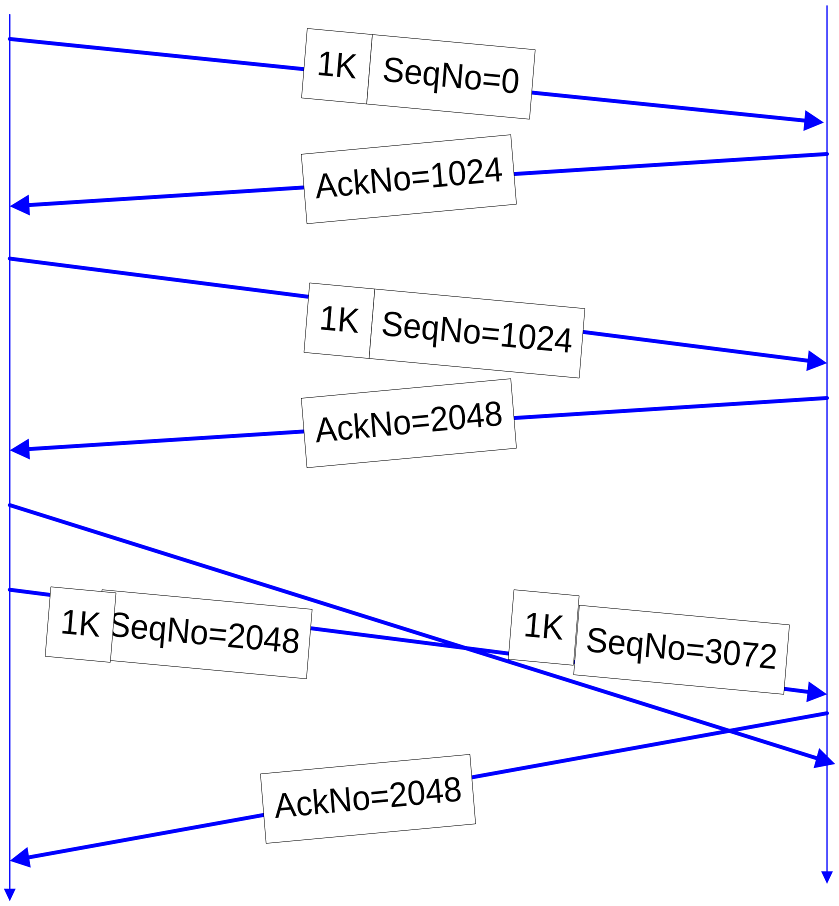
- Receiver sends ACK to sender
  - ACK is used for flow control, error control, and congestion control
- ACK number sent is the next sequence number expected
- Delayed ACK: TCP receiver normally delays transmission of an ACK (for about 200ms)
  - Why?
- ACKs are not delayed when packets are received out of sequence
  - Why?



Lost segment

Acknowledgments in TCP

- Receiver sends ACK to sender
  - ACK is used for flow control, error control, and congestion control
- ACK number sent is the next sequence number expected
- Delayed ACK: TCP receiver normally delays transmission of an ACK (for about 200ms)
  - Why?
- ACKs are not delayed when packets are received out of sequence
  - Why?



Out-of-order arrivals

Fast Retransmit

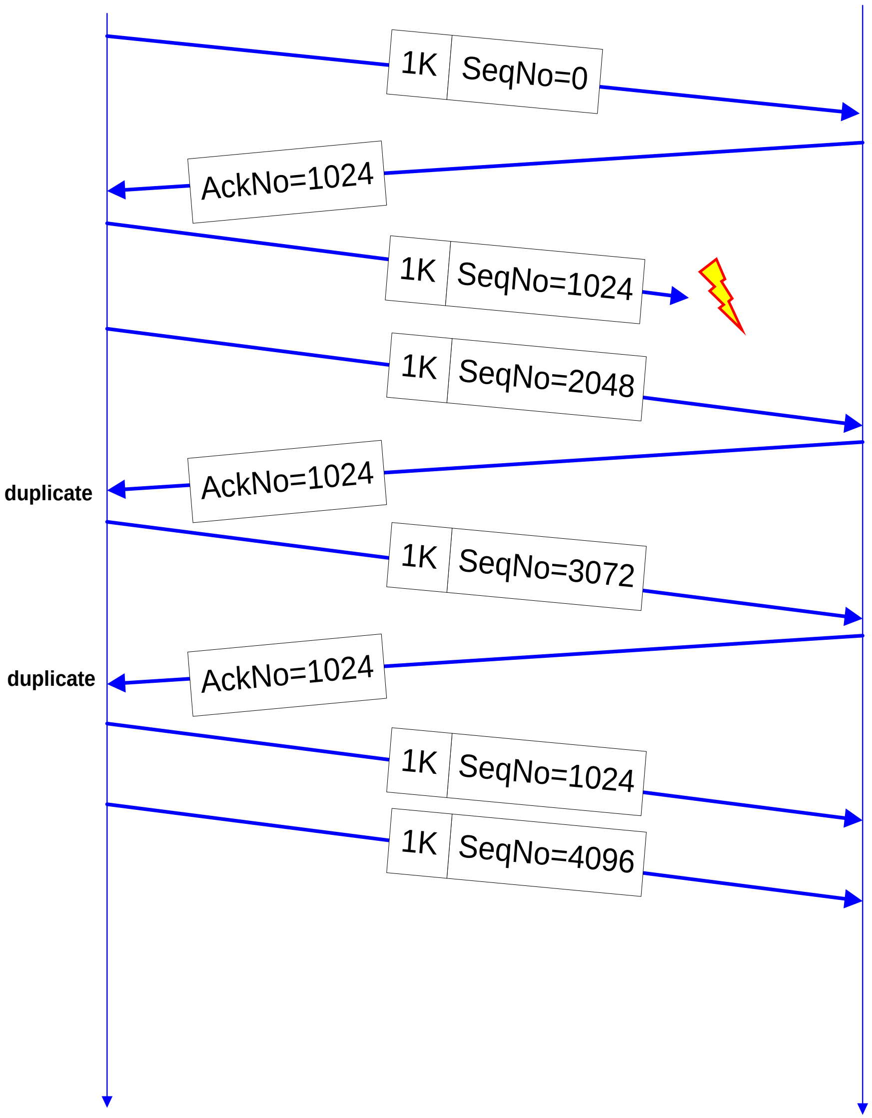
- If three or more duplicate ACKs are received in a row, the TCP sender believes that a segment has been lost.

- Then TCP performs a retransmission of what seems to be the missing segment, without waiting for a timeout to happen.

- Enter slow start:

  ssthresh = cwnd/2

  cwnd = 1

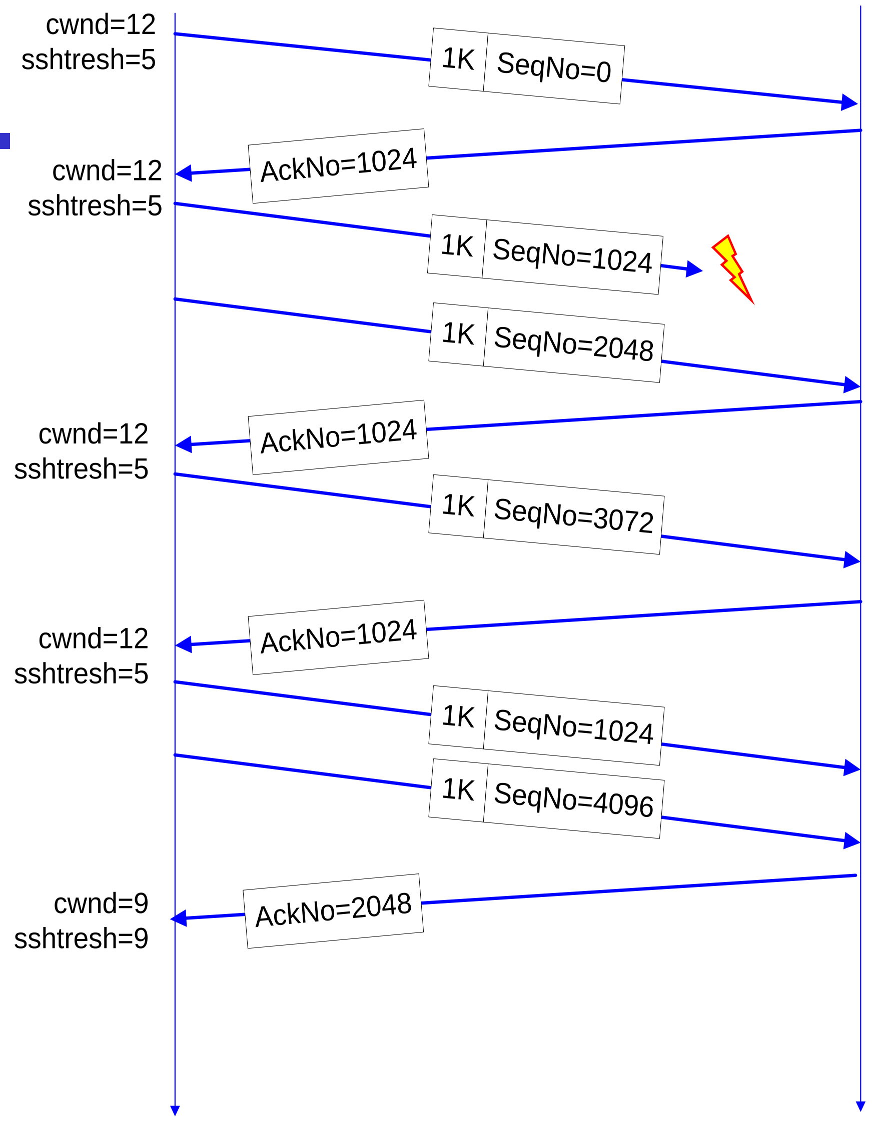


Fast Recovery

- Fast recovery avoids slow start after a fast retransmit
- **Intuition:** Duplicate ACKs indicate that data is getting through
- After three duplicate ACKs set:
  - Retransmit "lost packet"
  - ssthresh = cwnd/2
  - cwnd = cwnd+3
  - Enter congestion avoidance
  - Increment cwnd by one for each additional duplicate ACK
- When ACK arrives that acknowledges "new data" (here: AckNo=2028), set:

  cwnd=ssthresh

  enter congestion avoidance



TCP Reno

- Duplicate ACKs:
  - Fast retransmit
  - Fast recovery

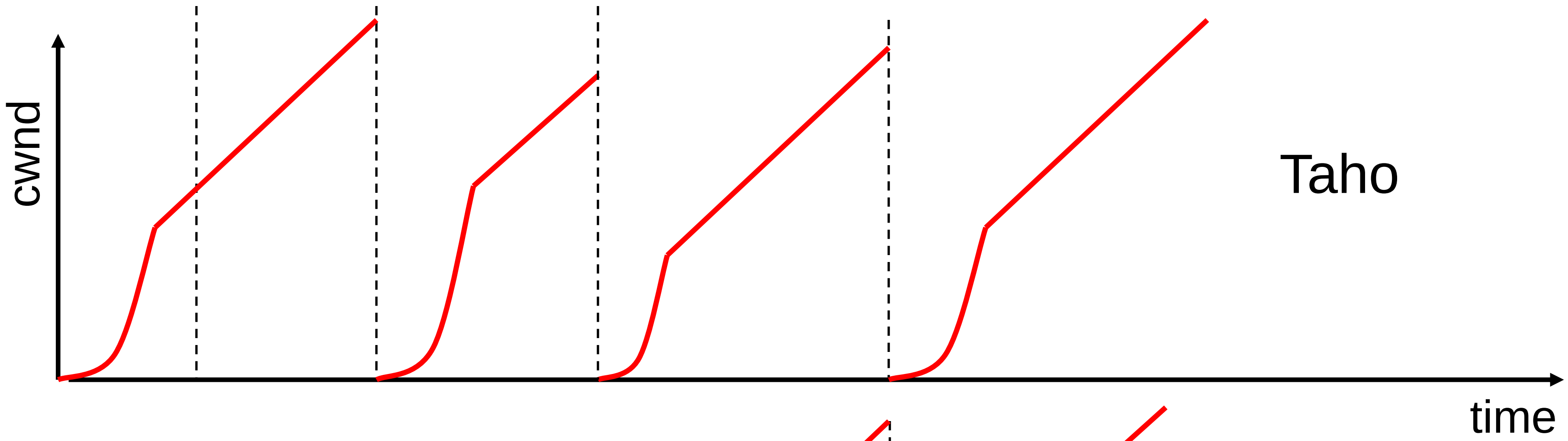
  → Fast Recovery avoids slow start

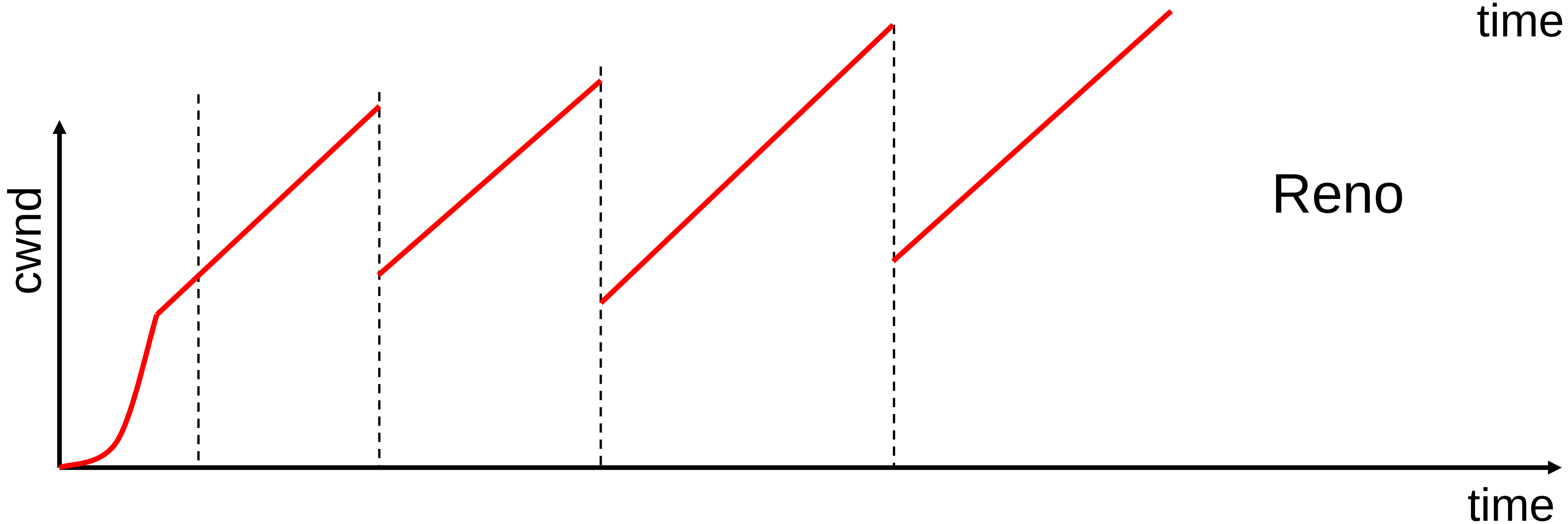
- Timeout:
  - Retransmit
  - Slow Start

- TCP Reno improves upon TCP Tahoe when a single packet is dropped in a round-trip time.

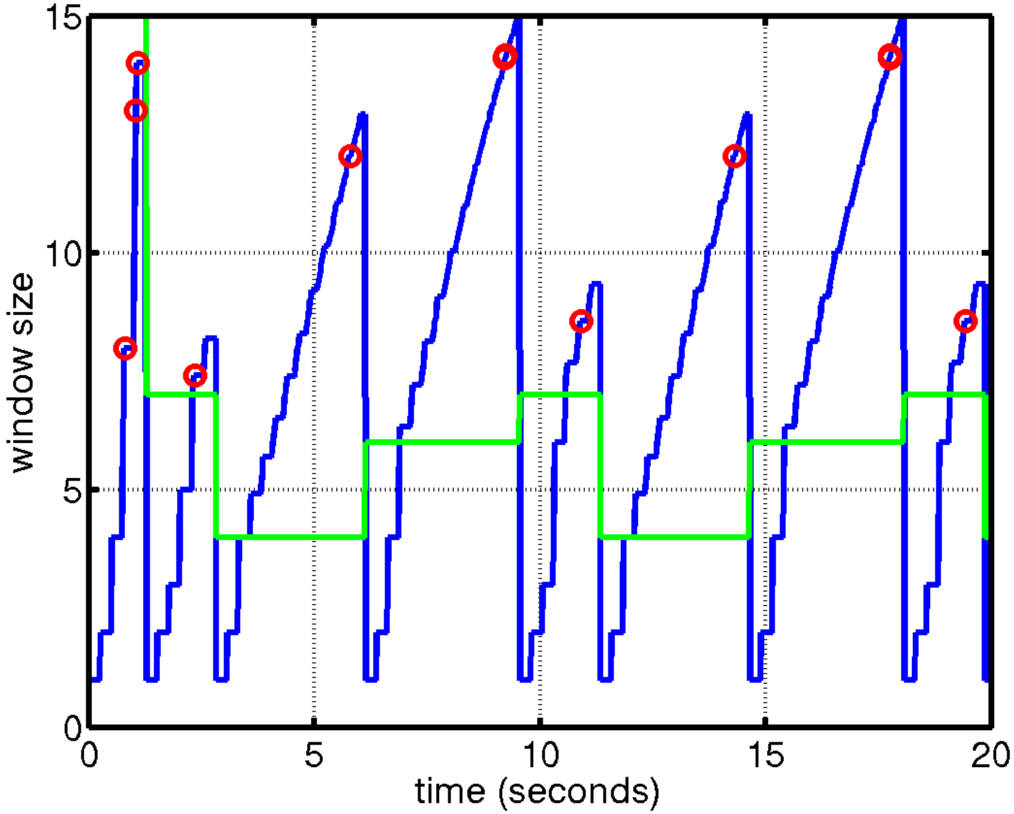
TCP Tahoe and TCP Reno

(for single segment losses)



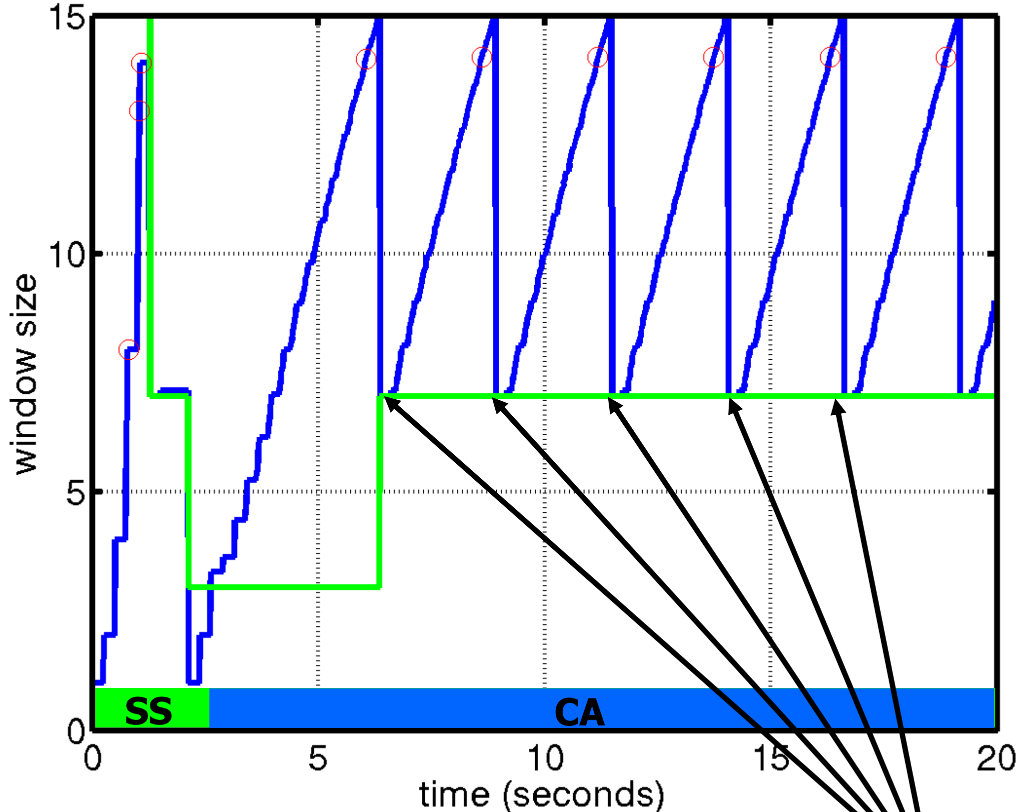


TCP Tahoe



This picture is copied from somewhere

TCP Reno (Jacobson 1990)



Fast retransmission/fast recovery

This picture is copied from somewhere

TCP New Reno

- When multiple packets are dropped, Reno has problems
- Partial ACK:
  - Occurs when multiple packets are lost
  - A partial ACK acknowledges some, but not all packets that are outstanding at the start of a fast recovery, takes sender out of fast recovery

  → Sender has to wait until timeout occurs
- **New Reno:**
  - Partial ACK does not take sender out of fast recovery
  - Partial ACK causes retransmission of the segment following the acknowledged segment
- New Reno can deal with multiple lost segments without going to slow start

SACK

- SACK = Selective acknowledgment
- Issue: Reno and New Reno retransmit at most 1 lost packet per round trip time
- **Selective acknowledgments:** The receiver can acknowledge non-continuous blocks of data (SACK 0-1023, 1024-2047)
- Multiple blocks can be sent in a single segment.
- TCP SACK:
  - Enters fast recovery upon 3 duplicate ACKs
  - Sender keeps track of SACKs and infers if segments are lost. Sender retransmits the next segment from the list of segments that are deemed lost.