# TCP - Part II

# What is Flow/Congestion/Error Control ?

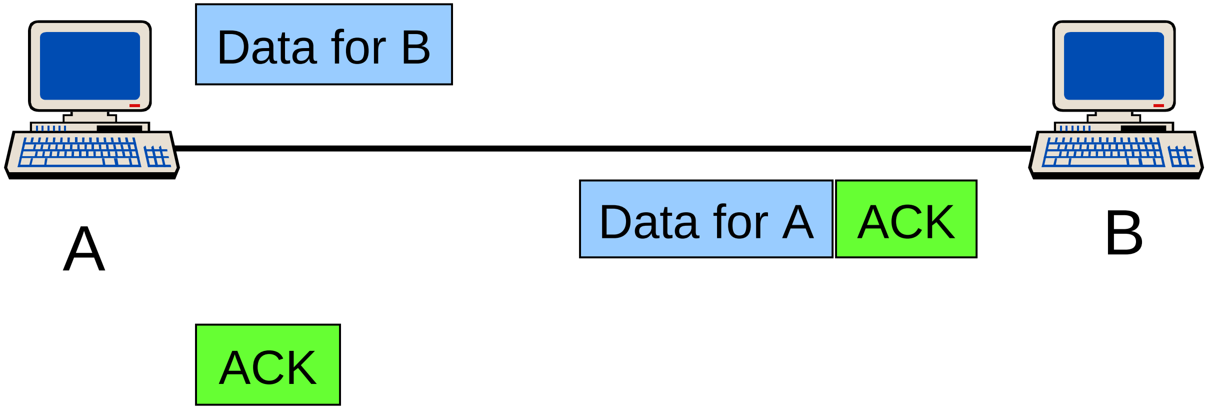
- **Flow Control:** Algorithms to prevent that the sender overruns the receiver with information
- **Error Control:** Algorithms to recover or conceal the effects from packet losses
- **Congestion Control:** Algorithms to prevent that the sender overloads the network

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

→ In TCP, the implementation of these algorithms is combined

# Acknowledgements in TCP

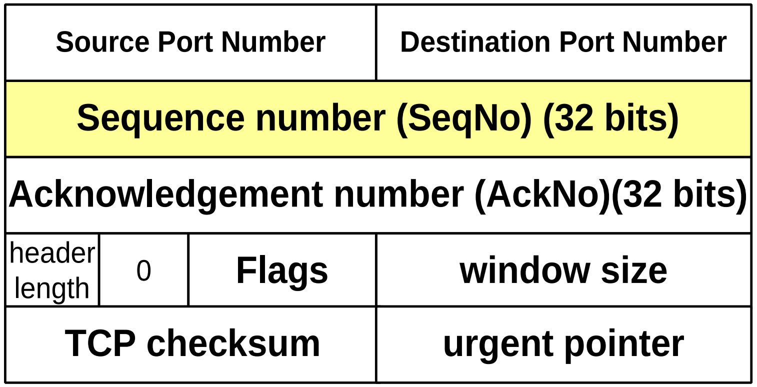
- TCP receivers use acknowledgments (ACKs) to confirm the receipt of data to the sender
- Acknowledgment can be added (“piggybacked”) to a data segment that carries data in the opposite direction
- ACK information is included in the TCP header
- Acknowledgements are used for flow control, error control, and congestion control



# Sequence Numbers and Acknowledgments in TCP

- TCP uses sequence numbers to keep track of transmitted and acknowledged data
- Each transmitted byte of payload data is associated with a sequence number
- **Sequence numbers count bytes and not segments**
- Sequence number of first byte in payload is written in *SeqNo* field
- Sequence numbers wrap when they reach $2^{32}$-1

- The sequence number of the first segment (Initial sequence number) is negotiated during connection setup



<table>
<tr><td colspan="3">Source Port Number</td><td>Destination Port Number</td></tr>
<tr><td colspan="4">Sequence number (SeqNo) (32 bits)</td></tr>
<tr><td colspan="4">Acknowledgement number (AckNo)(32 bits)</td></tr>
<tr><td>header length</td><td>0</td><td>Flags</td><td>window size</td></tr>
<tr><td colspan="3">TCP checksum</td><td>urgent pointer</td></tr>
</table>

# Sequence Numbers and Acknowledgments in TCP

- An acknowledgment is a confirmation of delivery of data
- When a TCP receiver wants to acknowledge data, it
  - writes a sequence number in the AckNo field, and
  - sets the ACK flag

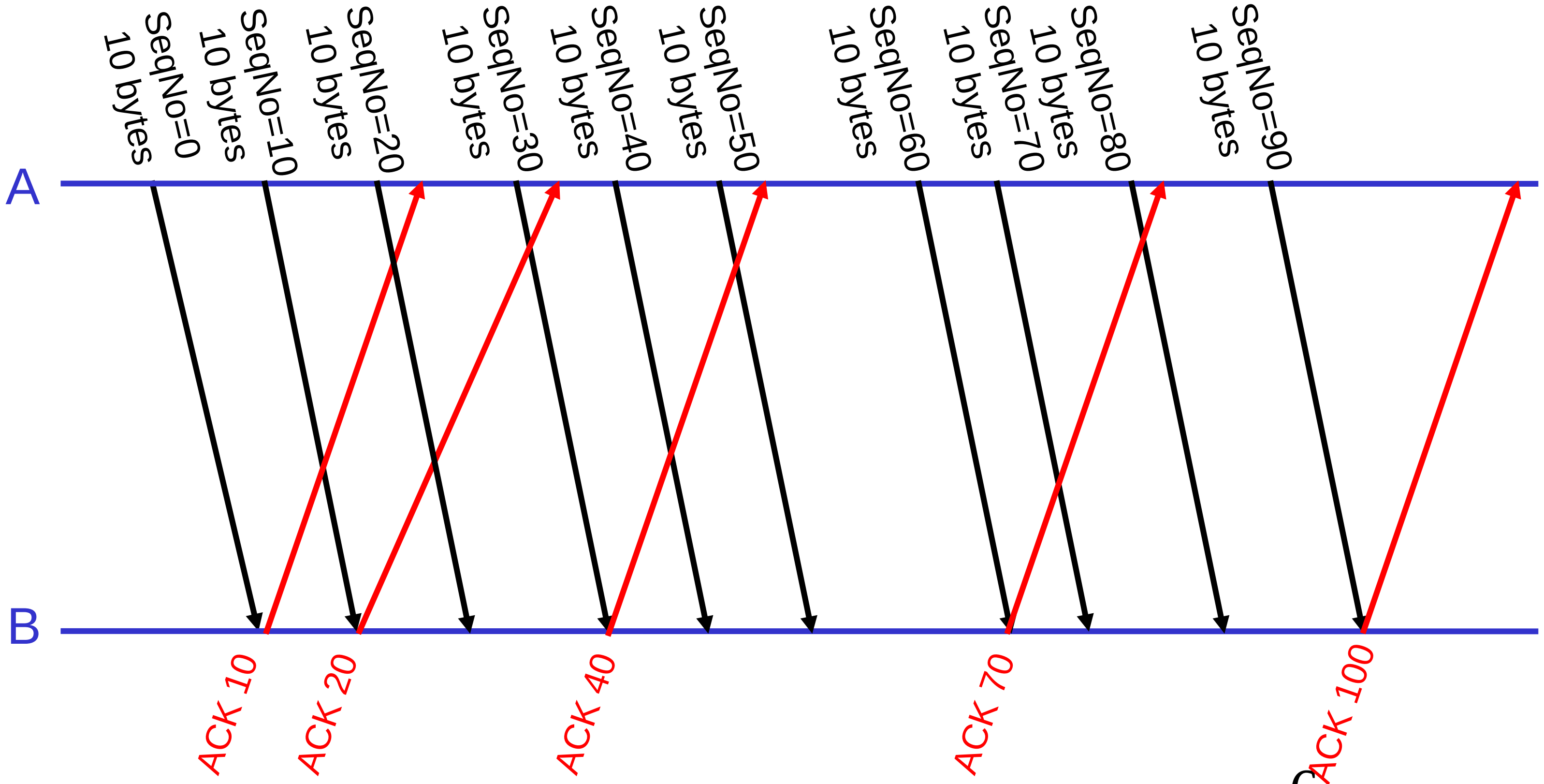
<table>
<tr><td colspan="3">Source Port Number</td><td>Destination Port Number</td></tr>
<tr><td colspan="4">Sequence number (SeqNo) (32 bits)</td></tr>
<tr><td colspan="4">Acknowledgement number (AckNo)(32 bits)</td></tr>
<tr><td>header length</td><td>0</td><td>Flags</td><td>window size</td></tr>
<tr><td colspan="3">TCP checksum</td><td>urgent pointer</td></tr>
</table>

**IMPORTANT: An acknowledgment confirms receipt for all unacknowledged data that has a smaller sequence number than given in the AckNo field**

Example: AckNo=5 confirms delivery for 1,2,3,4 (but not 5).

# Cumulative Acknowledgements

- TCP has **cumulative acknowledgements**:
  An acknowledgment confirms the receipt of all unacknowledged data with a smaller sequence number



# Cumulative Acknowledgements

- With cumulative ACKs, the receiver can only acknowledge a segment if all previous segments have been received

- With cumulative ACKs, receiver cannot selectively acknowledge blocks of segments:
  e.g., ACK for $S_0$-$S_3$ and $S_5$-$S_7$ (but not for $S_4$)

- Note: The use of cumulative ACKs imposes constraints on the retransmission schemes:
  - In case of an error, the sender may need to retransmit all data that has not been acknowledged

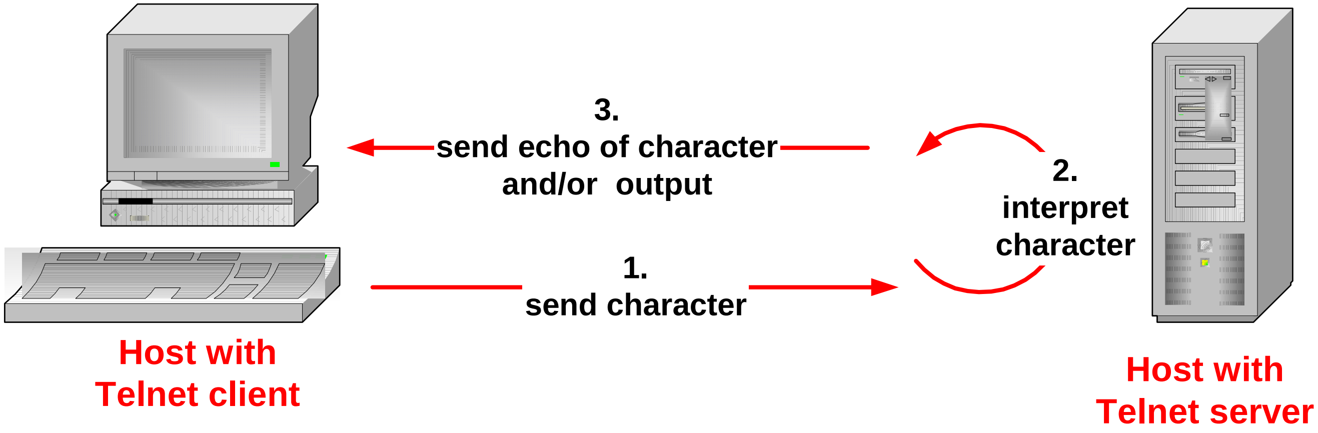
# Rules for sending Acknowledgments

- TCP has rules that influence the transmission of acknowledgments

- Rule 1: Delayed Acknowledgments
  - *Goal:* Avoid sending ACK segments that do not carry data
  - *Implementation:* Delay the transmission of (some) ACKs

- Rule 2: Nagle’s rule
  - *Goal:* Reduce transmission of small segments
    *Implementation:* A sender cannot send multiple segments with a 1-byte payload (i.e., it must wait for an ACK)

# An Example – Telnet Application

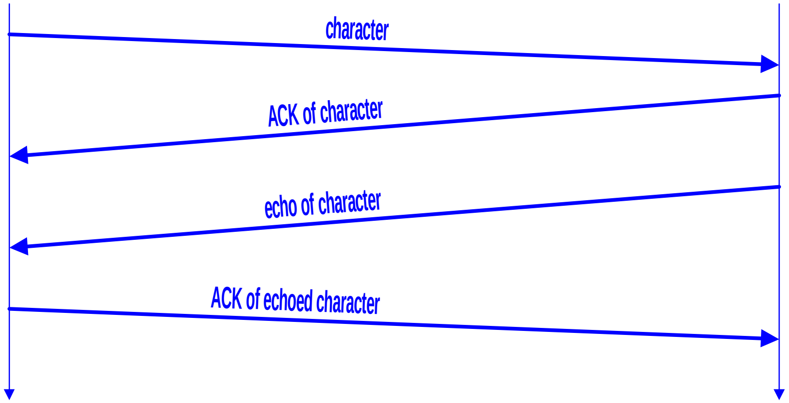


- Remote terminal applications (e.g., Telnet) send characters to a server. The server interprets the character and sends the output at the server to the client.

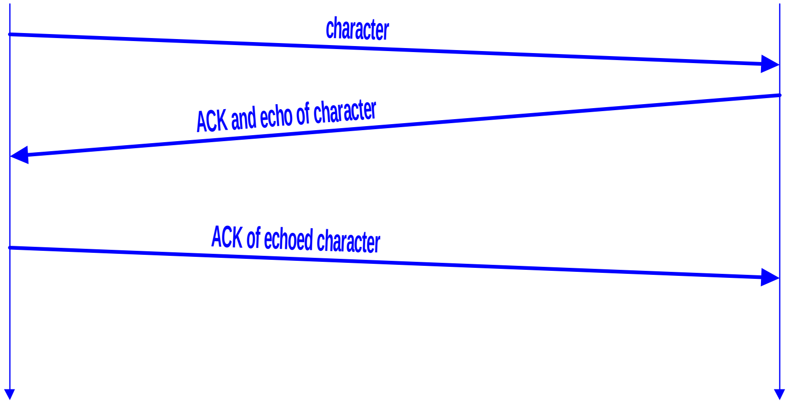
- For each character typed, you see three packets:
  1. **Client → Server:** Send typed character
  2. **Server → Client:** Echo of character (or user output) and acknowledgement for first packet
  3. **Client → Server:** Acknowledgement for second packet

# Why 3 segments per character?

- We would expect four segments per character:



- But we only see three segments per character:



- This is due to delayed acknowledgements

# Delayed Acknowledgement

- TCP delays transmission of ACKs for up to 200ms

- **Goal:** Avoid sending ACK packets that do not carry data.
  - The hope is that, within the delay, the second host will have data ready to be sent to the first host.  Then, the ACK can be piggybacked with a data segment

**In Example:**

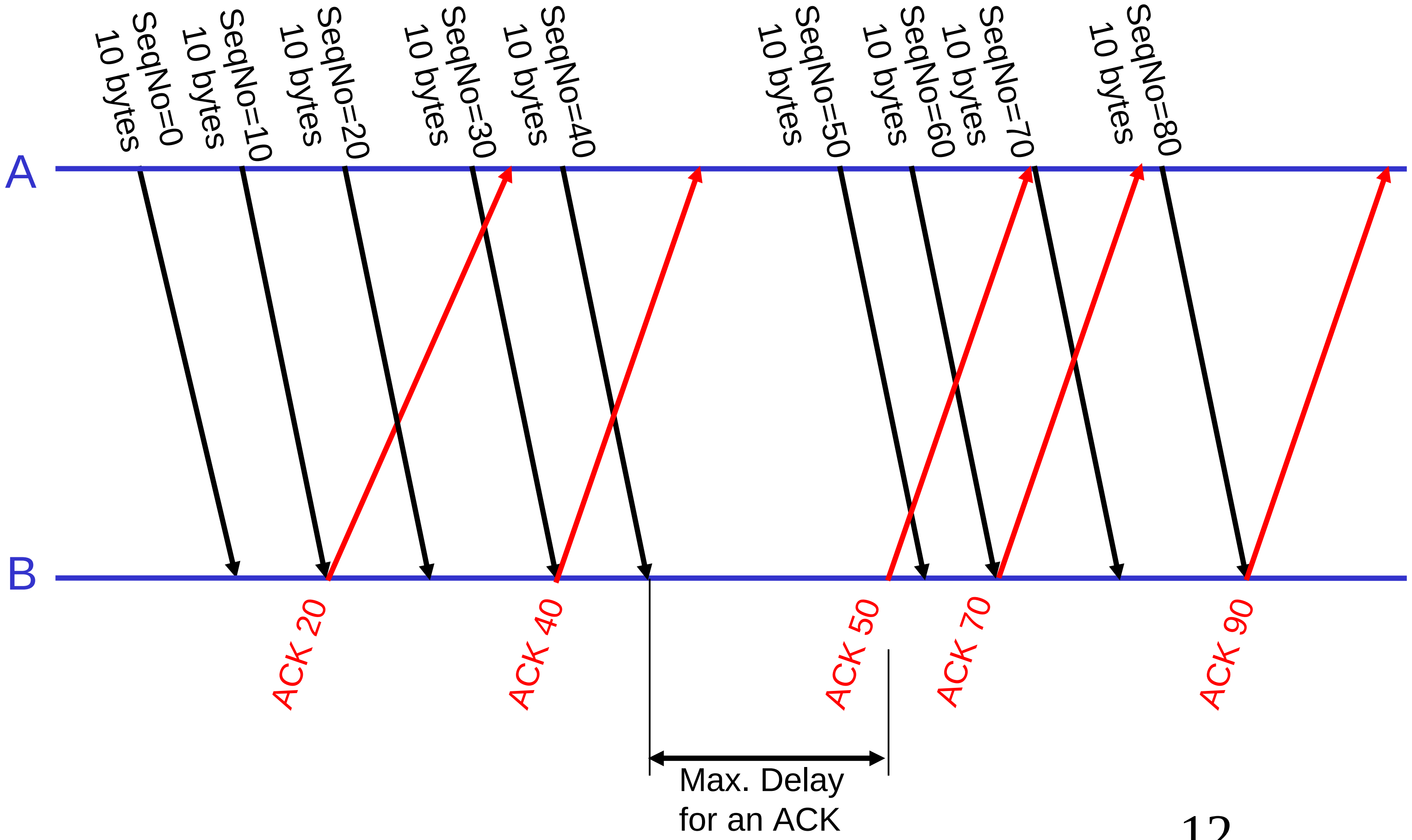
- Delayed ACK explains why the "ACK of character" and the "echo of character" are sent in the same segment
- The duration of delayed ACKs can be observed in the example when client sends ACKs

**Exceptions:**

- ACK should be sent for every second full sized segment
- Delayed ACK is not used when packets arrive out of order

# Delayed Acknowledgement

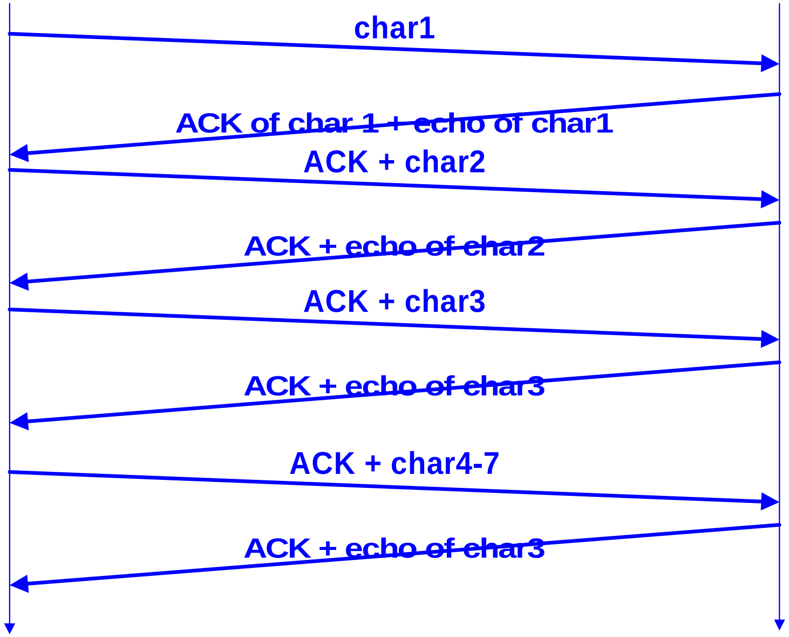
- Because of delayed ACKs, an ACK is often observed for every other segment



# An Example – Telnet Application

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

- Delayed acknowledgment does not kick in at the client
- The reason is that there is always data at client ready to send when the ACK arrives
- To avoid this the client does not send the data (typed character) as soon as it is available?



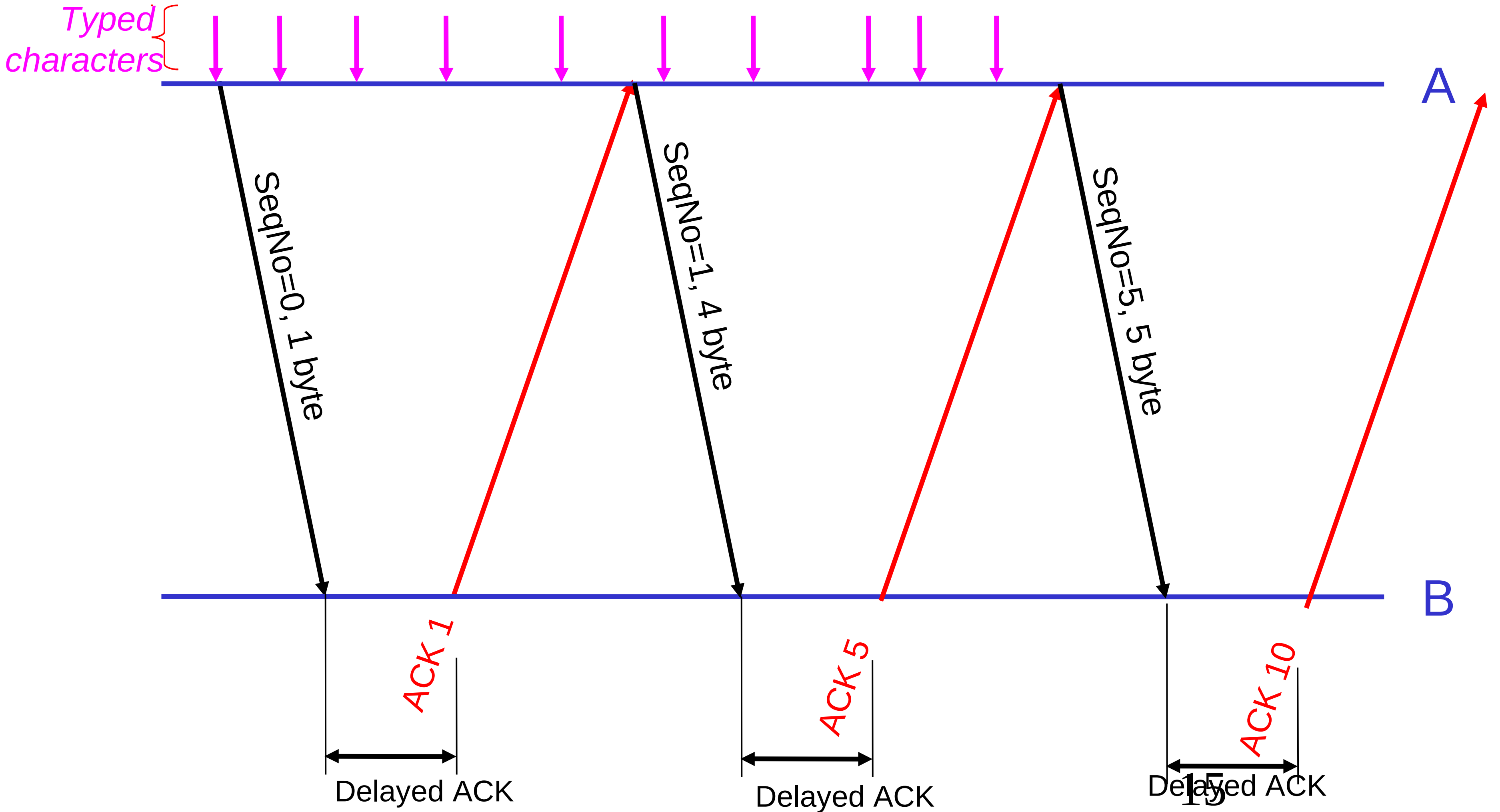
# Observing Nagle’s Rule

- **Observations:**
  - Client never has multiple unacknowledged segments outstanding
  - There are fewer transmissions than there are characters.
- This is due to **Nagle’s Rule:**
  - 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)
- **Goal of Nagle’s Rule:** Reduce the amount of small segments.

***There is an option for disabling Nagle's algorithm***

# Nagle’s Rule

- Only one 1-byte segment can be in transmission (Here: Since no data is sent from B to A, we also see delayed ACKs)



# Nagle's Algorithm

```
if there is new data to send
  if the window size >= MSS and available data is >= MSS
    send complete MSS segment now
  else
    if there is unconfirmed data still in the pipe
      enqueue data in the buffer until an acknowledge is received
    else
      send data immediately
    end if
  end if
end if
```

# Interaction between Nagle's Algorithm and Delayed ACK

Receiver waits for data and sender waits for acknowledgement – results in starvation

An example –

Consider a request or reply flow -

the size of the payload in the flow may not be an exact multiple of the MSS (the last packet of the flow will be smaller)

this last packet will not be transmitted until the previous packet is acknowledged.

# Interaction between Nagle's Algorithm and Delayed ACK

In the best case, the penultimate packet represents an even-numbered packet, triggering an immediate acknowledgement from the receiver which in turn “releases” the final small packet.

- In this case, the Nagle penalty is equal to one network round-trip for the entire flow.

If the penultimate packet be an odd-numbered packet, it will not be acknowledged by the receiver until the Delayed ACK timer expires

- the penalty becomes one network round-trip plus approximately 200 milliseconds.

# TCP Flow Control

# TCP Flow Control

- TCP uses a version of the **sliding window flow control,** where
  - Sending acknowledgements is separated from setting the window size at sender
  - Acknowledgements do not automatically increase the window size at sender side
- During connection establishment, both ends of a TCP connection set the initial size of the sliding window

# Window Management in TCP

- The receiver is returning two parameters to the sender

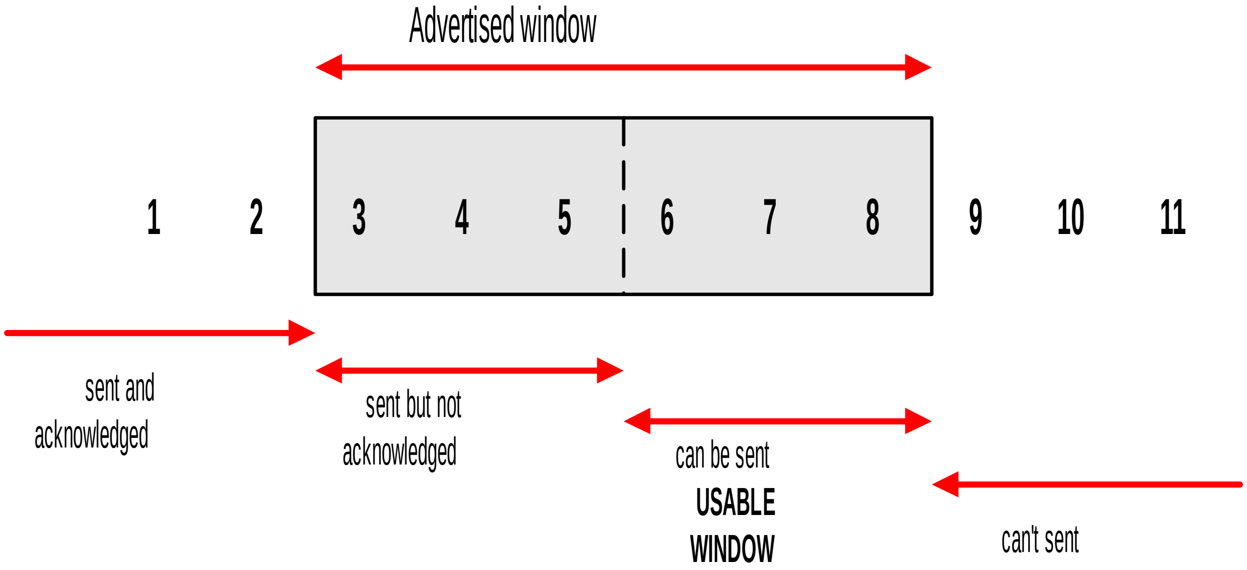
| AckNo | window size (win) |
|---|---|
| 32 bits | 16 bits |

- 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
  - Without changing its size
- Receiver can change the window size without acknowledging data

# Sliding Window Flow Control

- Sliding Window Protocol is performed at the byte level:



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

# Example - 1

*What is the value of the receiver window (rwnd) for host A if the receiver, host B, has a buffer size of 5,000 bytes and 1,000 bytes of received and unprocessed data?*

*The value of rwnd = 5,000 − 1,000 = 4,000. Host B can receive only 4,000 bytes of data before overflowing its buffer. Host B advertises this value in its next segment to A.*

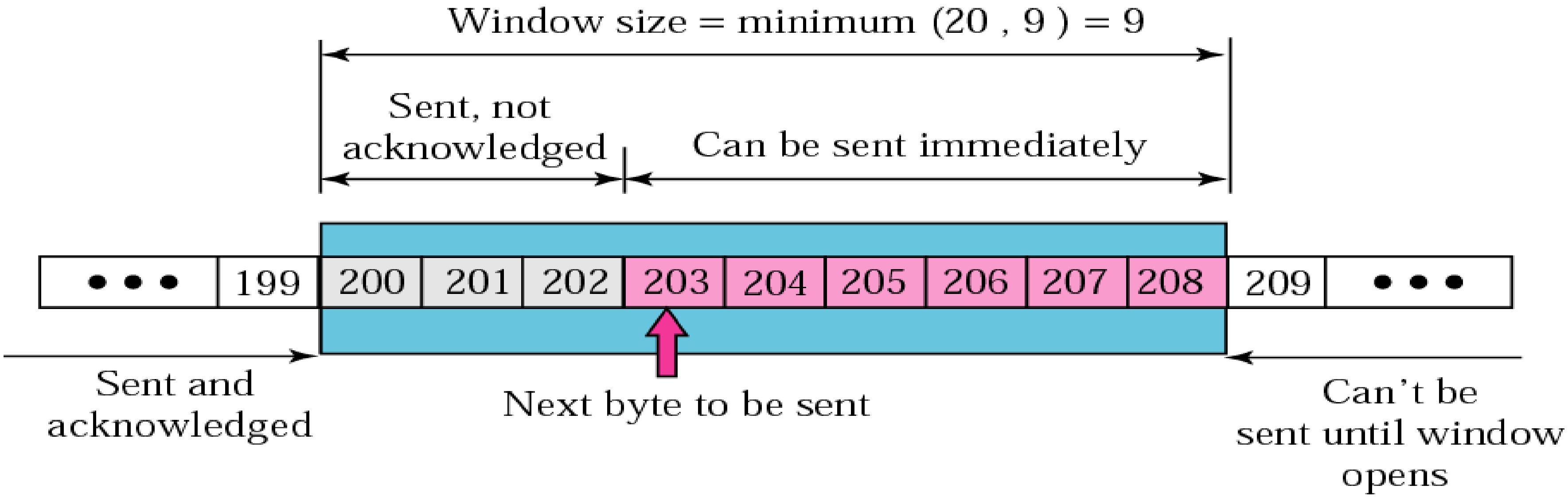
# Example - 2

*What is the size of the window for host A if the value of rwnd is 3,000 bytes and the value of cwnd is 3,500 bytes?*

*The size of the window is the smaller of rwnd and cwnd, which is 3,000 bytes.*

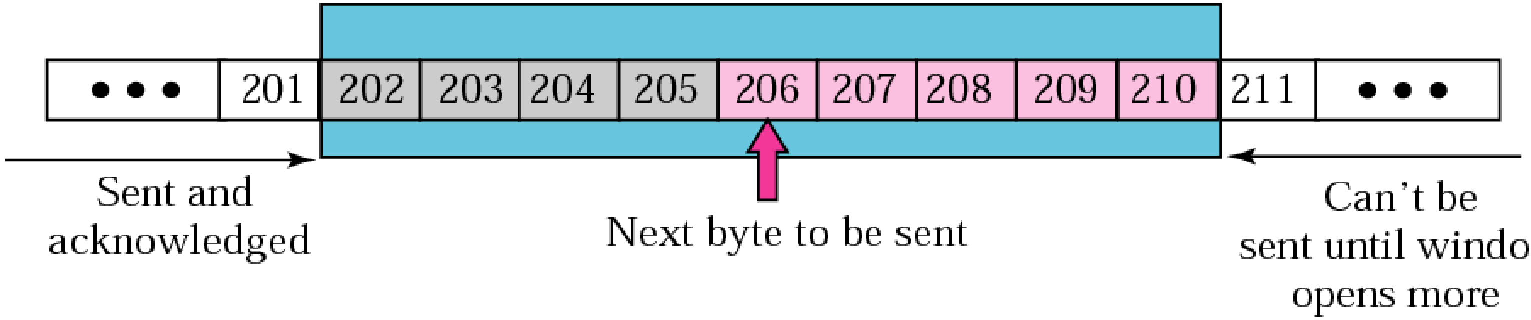
# Example - 3

*Assume that the sender has sent bytes up to 202 and cwnd is 20. The receiver has sent an acknowledgment number of 200 with an rwnd of 9 bytes.*



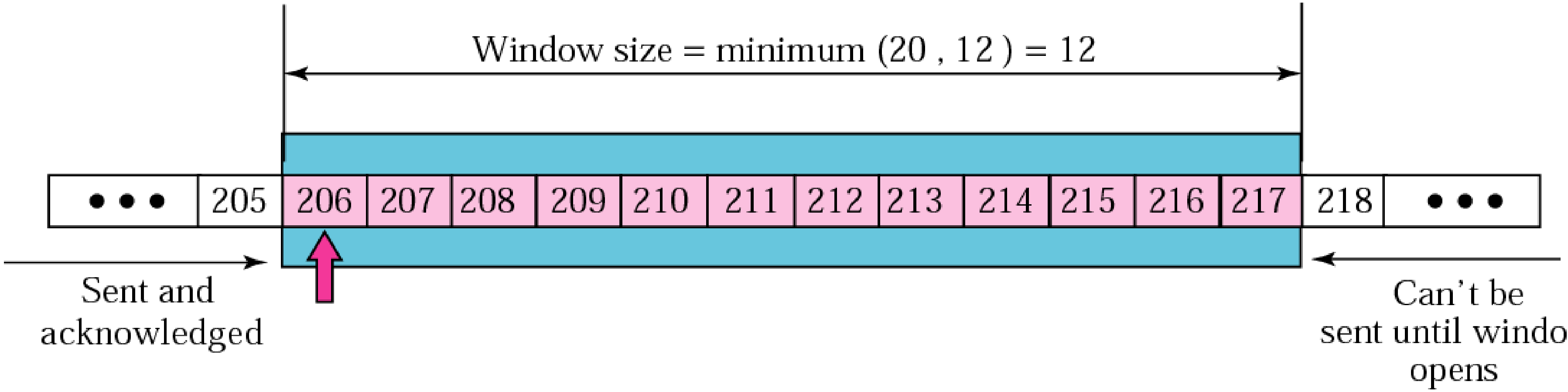
# Example – 3 contd.

*Next, the server receives a packet with an acknowledgment value of 202 and an rwnd of 9. The host has already sent bytes 203, 204, and 205. The value of cwnd is still 20.*



# Example – 3 contd.

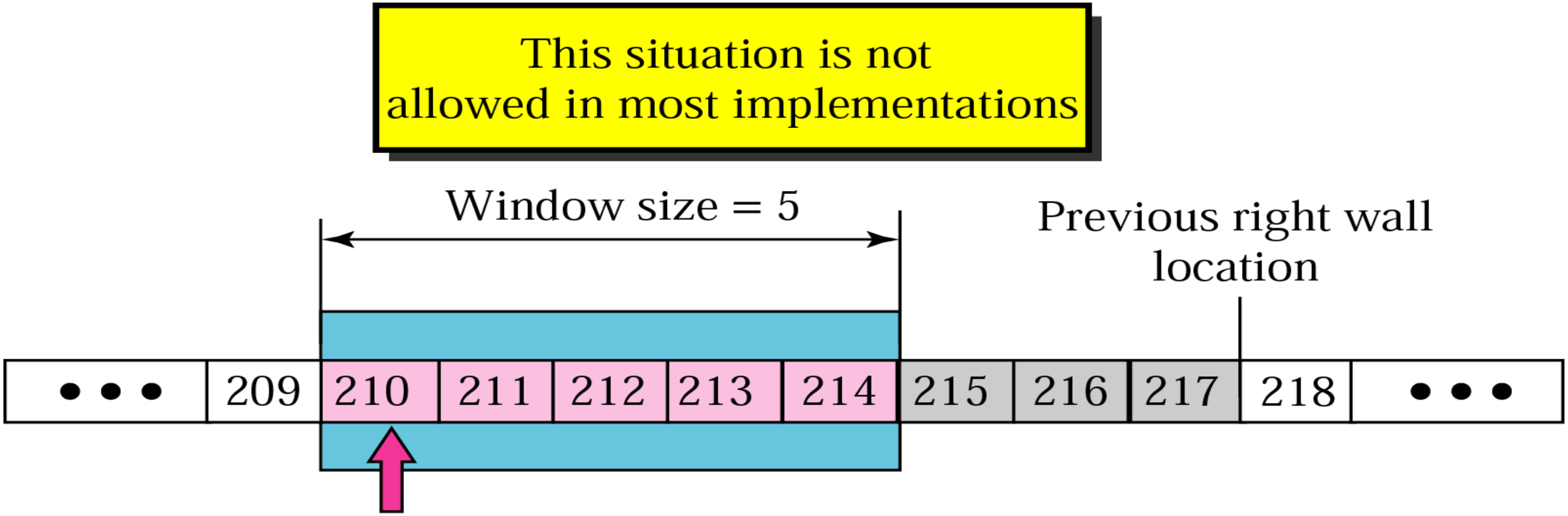
*Next, the sender receives a packet with an acknowledgment value of 206 and an rwnd of 12. The host has not sent any new bytes. The value of cwnd is still 20.*



# Example – 3 contd.

*Assume the sender has sent bytes 206 to 209. The sender's window shrinks accordingly.*

*Now the sender receives a packet with an acknowledgment value of 210 and an rwnd of 5. The value of cwnd is still 20.*



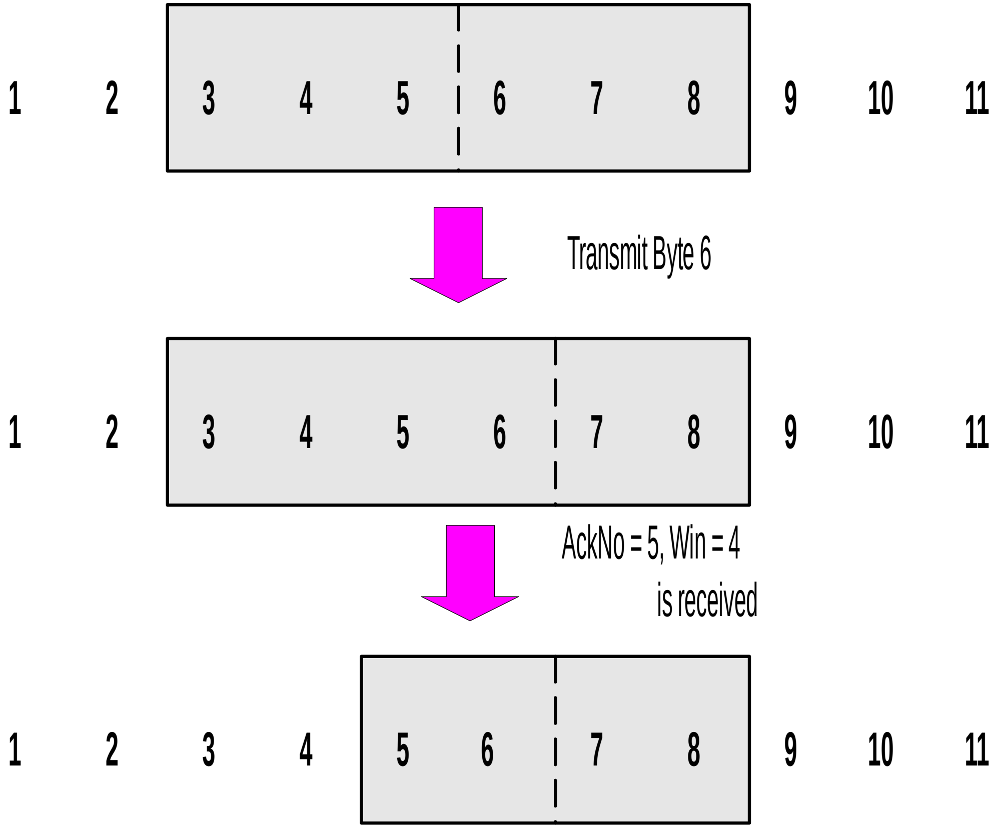
This situation is not allowed in most implementations

# Example – 3 contd.

*Window Shutdown - While one shouldn’t shrink the window, one can send rwnd = 0 to close the window. Sender can still send a “probe” packet.*

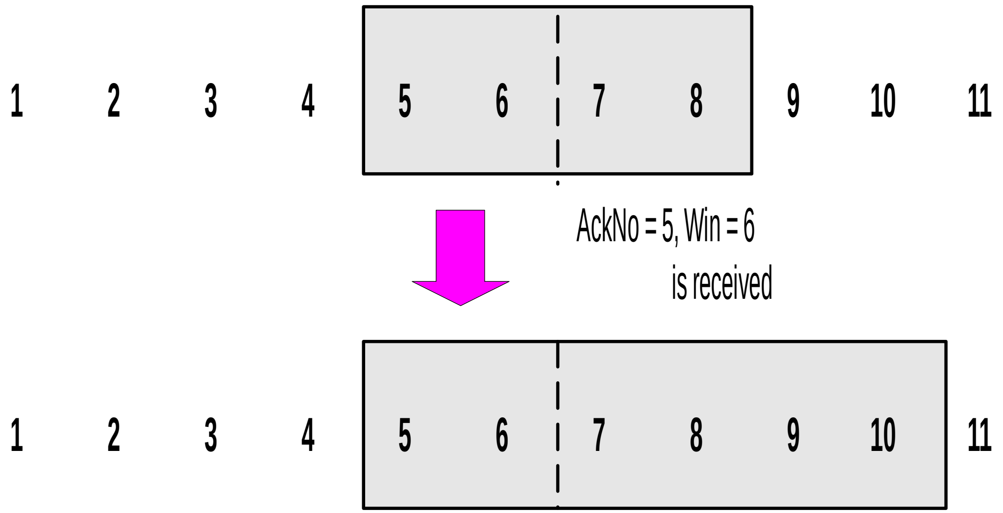
# Sliding Window: “Window Closes”

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



# Sliding Window: “Window Opens”

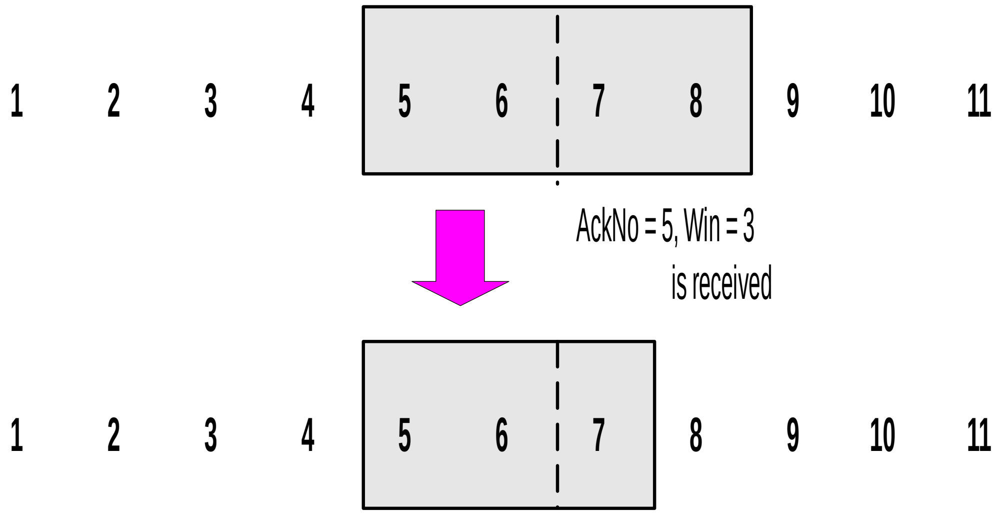
- Acknowledgement is received that enlarges the window to the right (AckNo  = 5, Win=6):



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

# Sliding Window: “Window Shrinks”

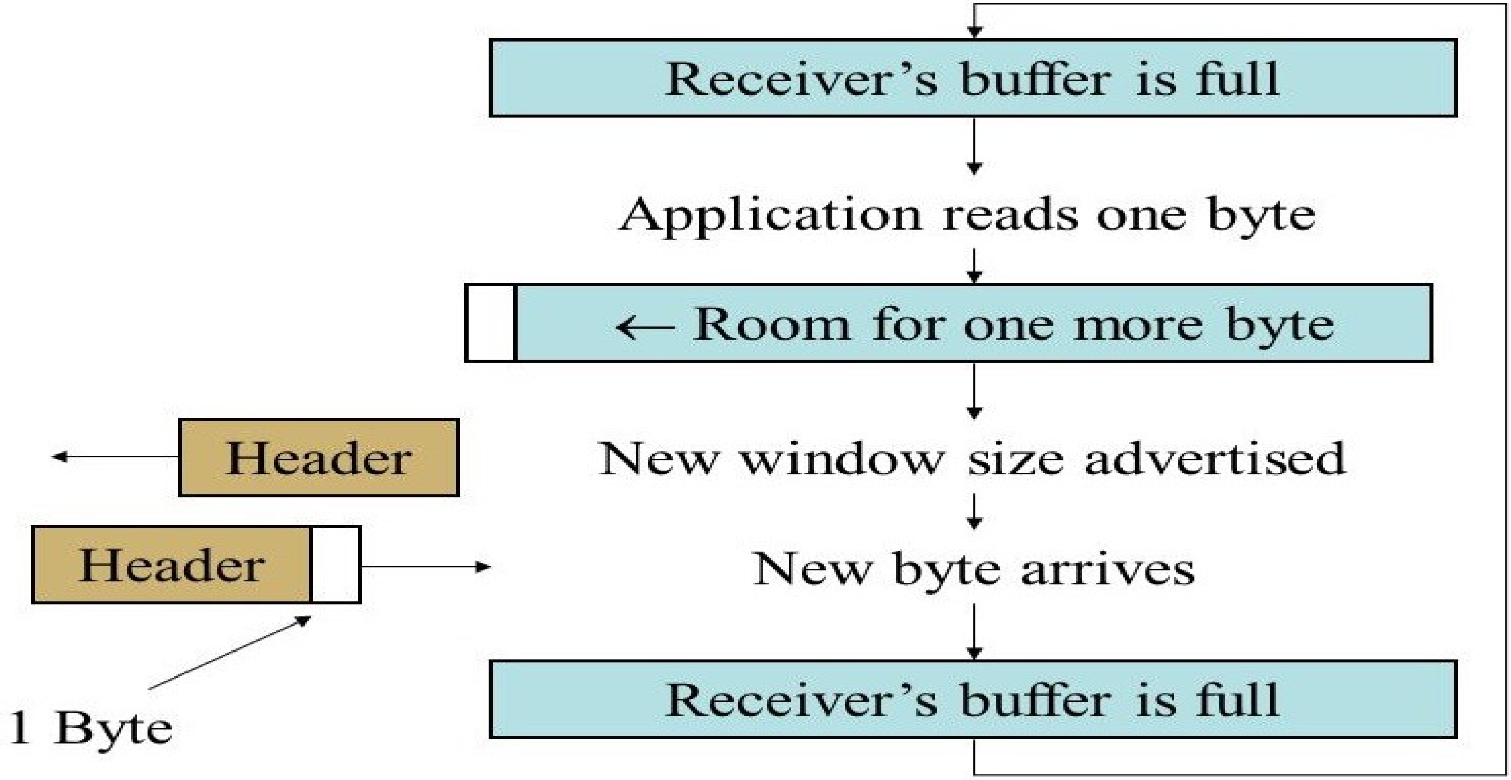
- Acknowledgement is received that reduces the window from the right (AckNo = 5, Win=3):



- Shrinking a window should not be used

# Silly Window Syndrome

What happens if the receiving TCP has a buffer size of 1000 bytes and the sending TCP has just sent 1000 bytes. The receiving buffer is now full so the receiver tells the sender to stop (window size = 0).


Receiver's buffer is full
Application reads one byte
← Room for one more byte
Header
New window size advertised
Header
New byte arrives
1 Byte
Receiver's buffer is full

# Silly Window Syndrome

Clark’s Solution - Acknowledge receipt right away, but don’t change the window size until you have at least half the buffer space available. Or, delay the ack until there is a decent amount of buffer space available.

# TCP Error Control

# Error Control in TCP

- TCP Error Control –
  - End-to-end reliability
    - no corrupted segments
    - no lost segments
    - all segments in order

# Error Control in TCP

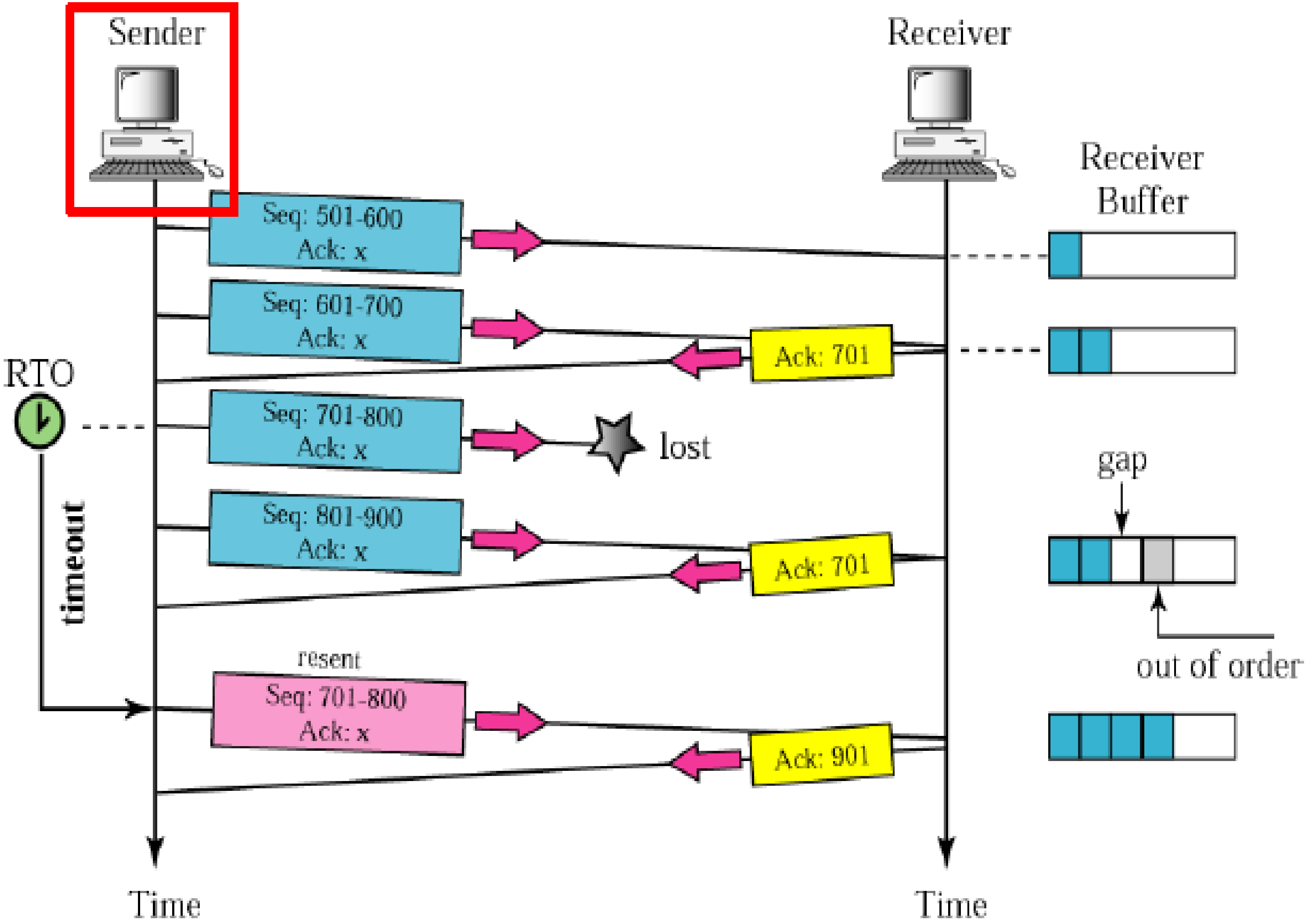
- TCP Error Control –
- Tools for TCP Error Control
  - Receiver side
    1. checksum – enables detection of corrupted segments on receiver’s side
    2. sequence numbers – enable detection of lost, out-of - order or duplicate segments on receiver side
  - Sender side
    1. acknowledgments
    2. Retransmission timers

# Error Control in TCP

- TCP maintains a **Retransmission Timer** for each connection:
  - The timer is started during a transmission. A timeout causes a retransmission

- **TCP couples error control and congestion control** (i.e., it assumes that errors are caused by congestion)
  - Retransmission mechanism is part of congestion control algorithm

- **Here:** How to set the timeout value of the retransmission timer?


Sender
Receiver
Receiver Buffer
Seq: 501-600
Ack: x
Seq: 601-700
Ack: x
Ack: 701
RTO
Seq: 701-800
Ack: x
lost
timeout
gap
Seq: 801-900
Ack: x
Ack: 701
out of order
resent
Seq: 701-800
Ack: x
Ack: 901
Time
Time

# TCP Retransmission Timer

- **Retransmission Timer:**
  - The setting of the retransmission timer is crucial for efficiency
  - A Timeout value is required
  - Timeout value too small → results in unnecessary retransmissions
  - Timeout value too large → long waiting time before a retransmission can be issued

  - Delays in the network are not fixed
  - Therefore, the retransmission timers must be adaptive

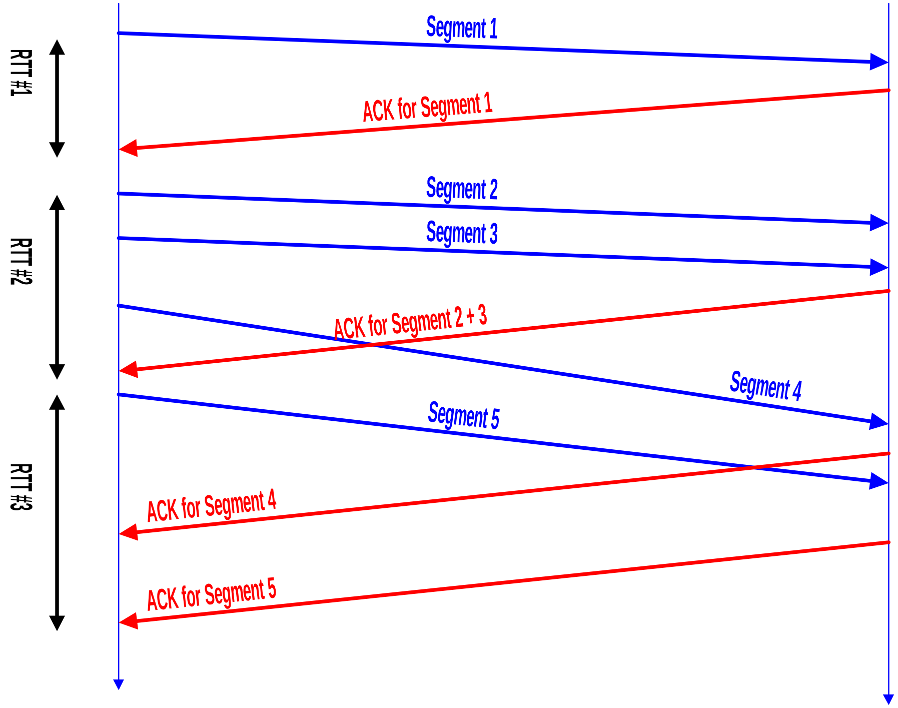
# Round-Trip Time Measurements

- The retransmission mechanism of TCP is adaptive
  - Retransmission timer is set to a **Retransmission Timeout (RTO) value**.
  - RTO is calculated based on the *RTT* measurements.

The RTT is based on time difference between segment transmission and ACK

**But:**

- TCP does not ACK each segment
- Each connection has only one timer



# Round-Trip Time Measurements

- Retransmission timer is set to a **Retransmission Timeout (RTO) value**.
- RTO is calculated based on the *RTT* measurements.
- a TCP sender maintains two state variables:
    - SRTT (smoothed round-trip time) and
    - RTTVAR (round-trip time variation)
- Until a round-trip time (RTT) measurement is made for a segment sent between the sender and receiver, the sender SHOULD set RTO = 1 second

(RFC 6298)

# Round-Trip Time Measurements

- When the first RTT measurement R is made, the host MUST set

  SRTT = R, RTTVAR = R/2,

  RTO = SRTT + max (G, K*RTTVAR)

  where K = 4, clock granularity is G seconds

- *Every time when a new RTT measurement R' is made,* the following estimators *srtt* and *rttvar* are recalculated

$rttvar = \beta ( | srtt - R' | ) + (1- \beta ) rttvar$

$srtt = \alpha R' + (1- \alpha ) srtt$

- The gains are set to $\alpha = 1/8$ and $\beta = 1/4$
- *rttvar and srtt must be computed in the above order*

# Round-Trip Time Measurements

After setting the two values, RTO is updated as follows:

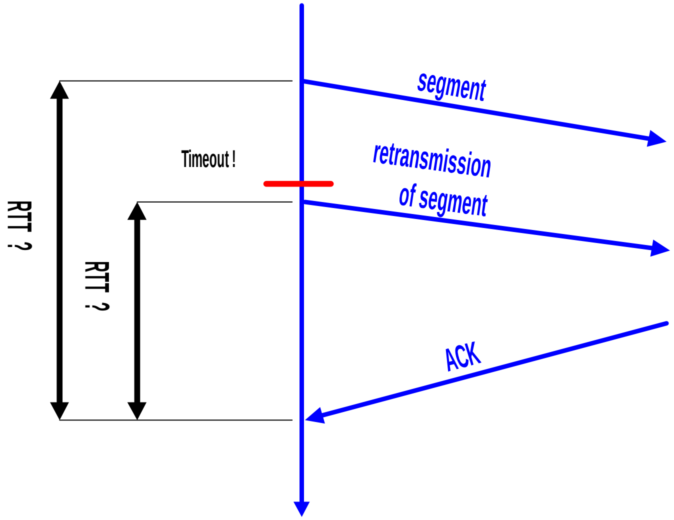
RTO = srtt + max (G, K*rttvar)

G is needed when K*rttvar equals to zero

It is observed that finer clock granularities (<= 100 msec) perform somewhat better than coarser granularities.

# Karn's Algorithm

- If an ACK for a retransmitted segment is received, the sender cannot tell if the ACK belongs to the original or the retransmission.



**Karn's Algorithm**:

Don't update *srtt* on any segments that have been retransmitted.

Each time when TCP retransmits, it sets:

$RTO_{n+1} = max\ (\ 2\ RTO_n\ ,\ 64)$ (exponential backoff)

# Managing the RTO Timer

**Placement On Retransmission Queue, Timer Start:** As soon as a segment containing data is transmitted, a copy of the segment is placed in a data structure called the retransmission queue.

The queue is kept sorted by the time remaining in the retransmission timer, so the TCP software can keep track of which timers have the least time remaining before they expire.

**Acknowledgment Processing:** If an acknowledgment is received for a segment before its timer expires, the segment is removed from the retransmission queue.

**Retransmission Timeout:** If an acknowledgment is not received before the timer for a segment expires, a retransmission timeout occurs, and the segment is automatically retransmitted.

# Managing the RTO Timer

1. Every time a packet containing data is sent (including a retransmission), if the timer is not running, start it running so that it will expire after RTO seconds (for the current value of RTO).

2. When all outstanding data has been acknowledged, turn off the retransmission timer.

3. When an ACK is received that acknowledges new data, restart the retransmission timer so that it will expire after RTO seconds (for the current value of RTO).

# Managing the RTO Timer

4. Retransmit the earliest segment that has not been acknowledged by the TCP receiver.

5. The host MUST set RTO = RTO * 2 ("back off the timer"). A maximum value is set to provide an upper bound to this doubling operation.

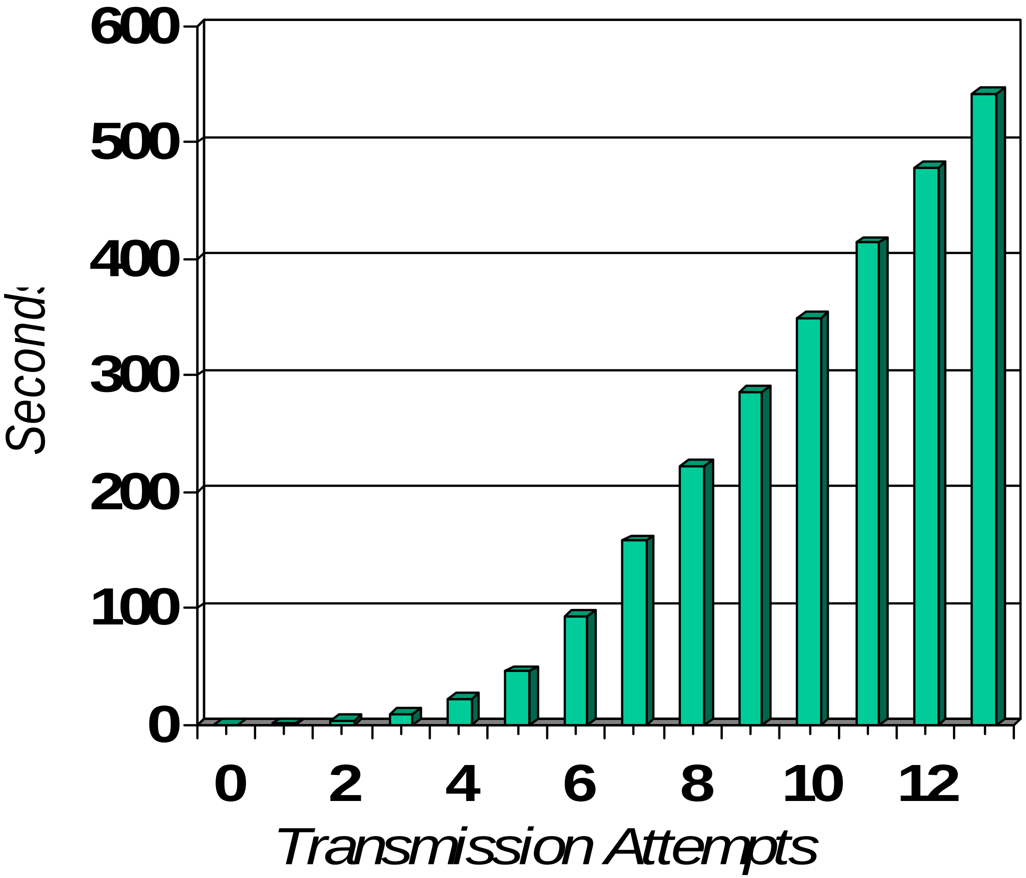
6. Start the retransmission timer, such that it expires after RTO seconds.

7. If the timer expires awaiting the ACK of a SYN segment and the TCP implementation is using an RTO less than *the initial value*, the RTO MUST be re-initialized to the initial value when data transmission begins (i.e., after the three-way handshake completes).

# Exponential Backoff

- Scenario: File transfer between two machines. Disconnect cable.
- The interval between retransmission attempts in seconds is:

  *1.03, 3, 6, 12, 24, 48, 64, 64, 64, 64, 64, 64, 64.*
- Time between retransmissions is doubled each time (Exponential Backoff Algorithm)
- Timer is not increased beyond 64 seconds
- TCP gives up after 13th attempt and 9 minutes.



# TCP Congestion Control

# TCP Congestion Control

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

- The window size at the sender is set as follows:

  **Send Window = MIN (flow control window, congestion window)**

where

- – flow control window is advertised by the receiver
- – congestion window is adjusted based on feedback from the network

# TCP Congestion Control

- TCP congestion control is governed by two parameters:
  - **Congestion Window** (**cwnd**)
  - **Slow-start threshhold Value** (**ssthresh)**

    Initial value is $2^{16}$-1

- 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.
  - » CWND increments by 1 MSS (maximum segment size) at a time

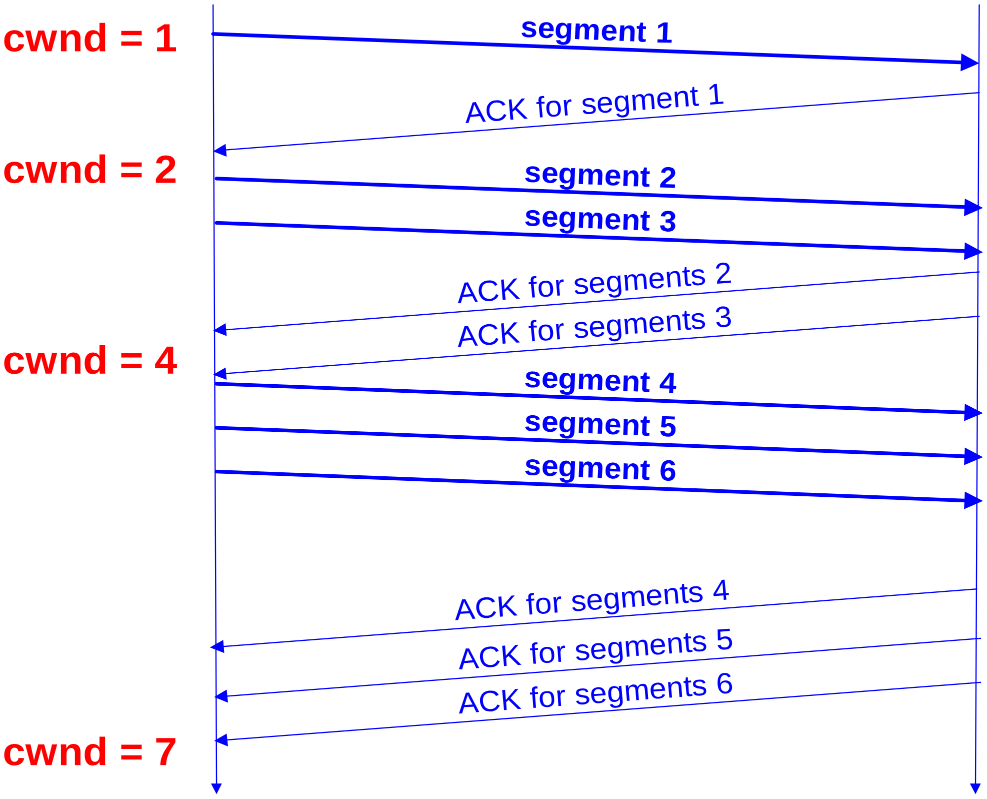
- The receiver sends an acknowledgement (ACK) for each Segment
  - » Note: Generally, a TCP receiver sends an ACK for every other segment.
- Each time an ACK is received by the sender, the congestion window is increased by 1 segment:

  **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.

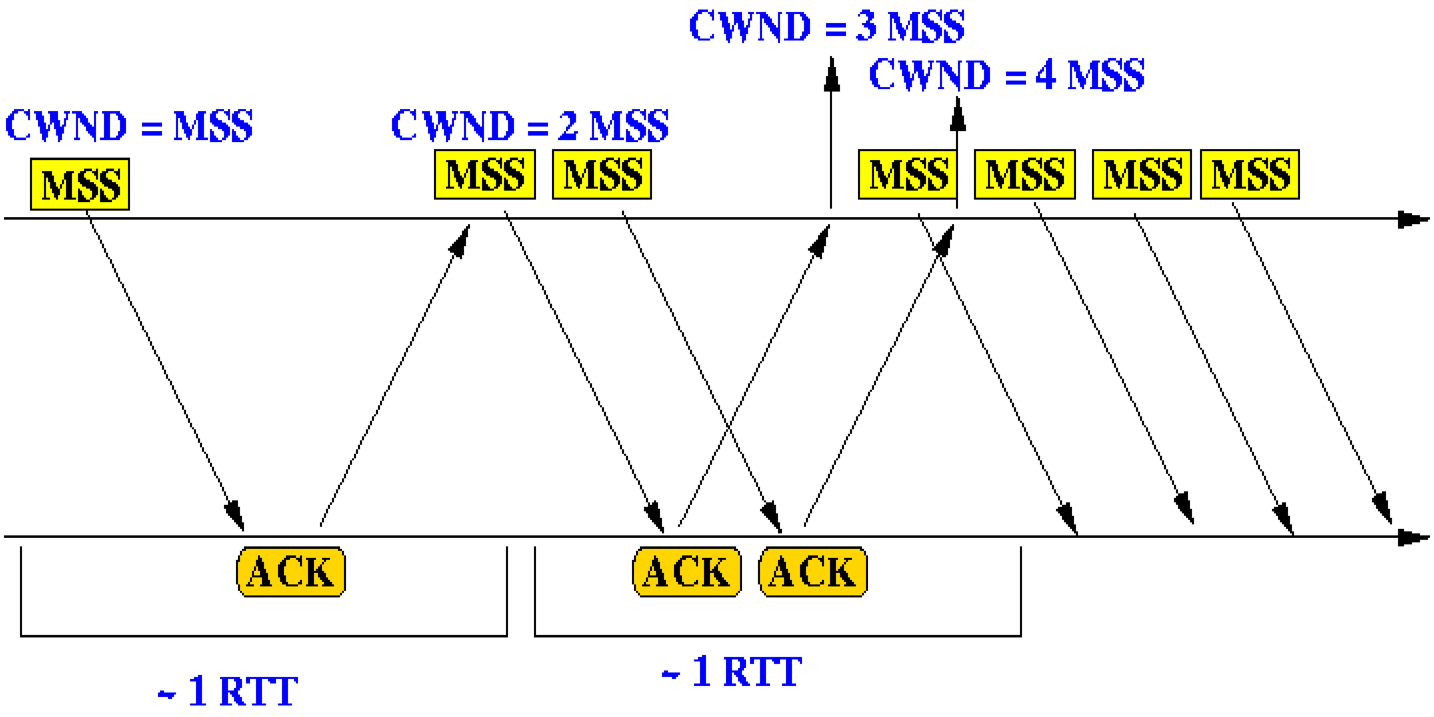
- Does Slow Start increment slowly? Not really.
  In fact, the increase of cwnd is exponential

# 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***



# Slow Start Example



The congestion window size (CWND) DOUBLES after approximately 1 RTT (Exponential Increase)

# Congestion Avoidance

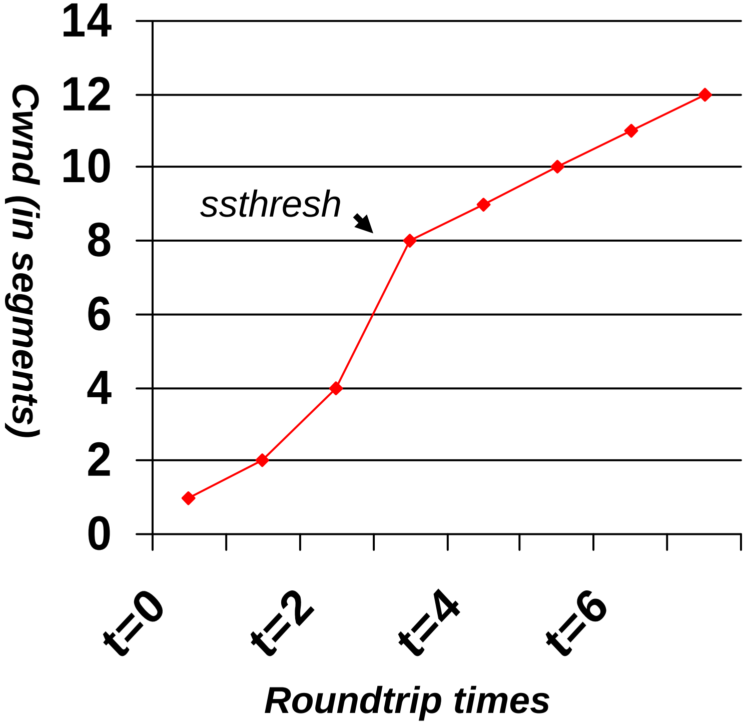
- Congestion avoidance phase is started if cwnd reaches the slow-start threshold value

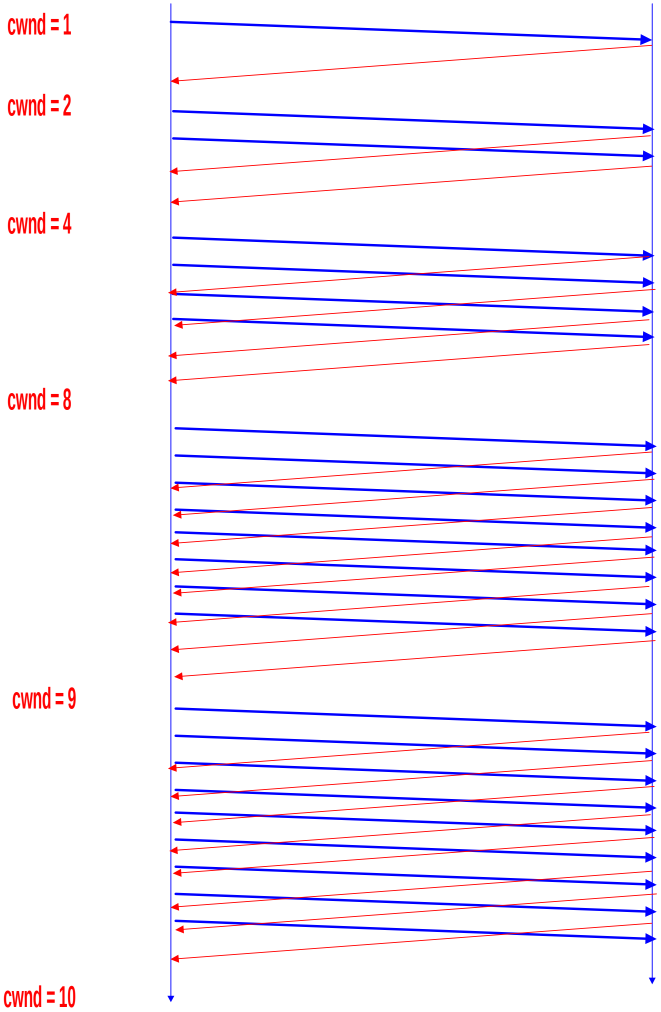
- If $cwnd \geq ssthresh$ then each time an ACK is received, increment cwnd as follows:
  - $cwnd = cwnd + 1/ cwnd$

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

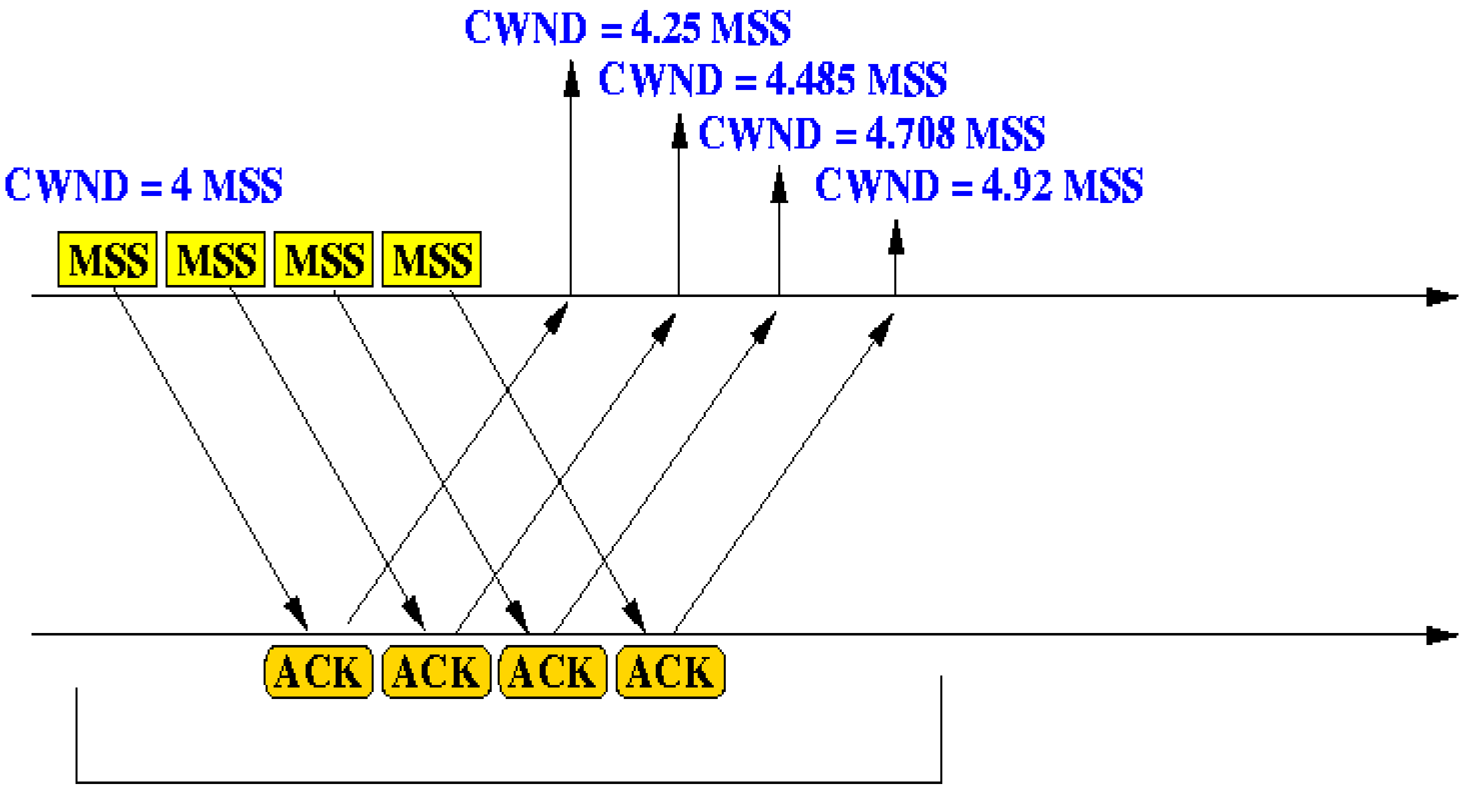
# Example of
# Slow Start/Congestion Avoidance

Assume that *ssthresh = 8*





# Example of Congestion Avoidance



The congestion window size (CWND) increases APPROXIMATELY by 1 after approximately 1 RTT (Linear Increase)

# Responses to Congestion

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

      ssthressh = cwnd / 2
    - cwnd is reset to one:

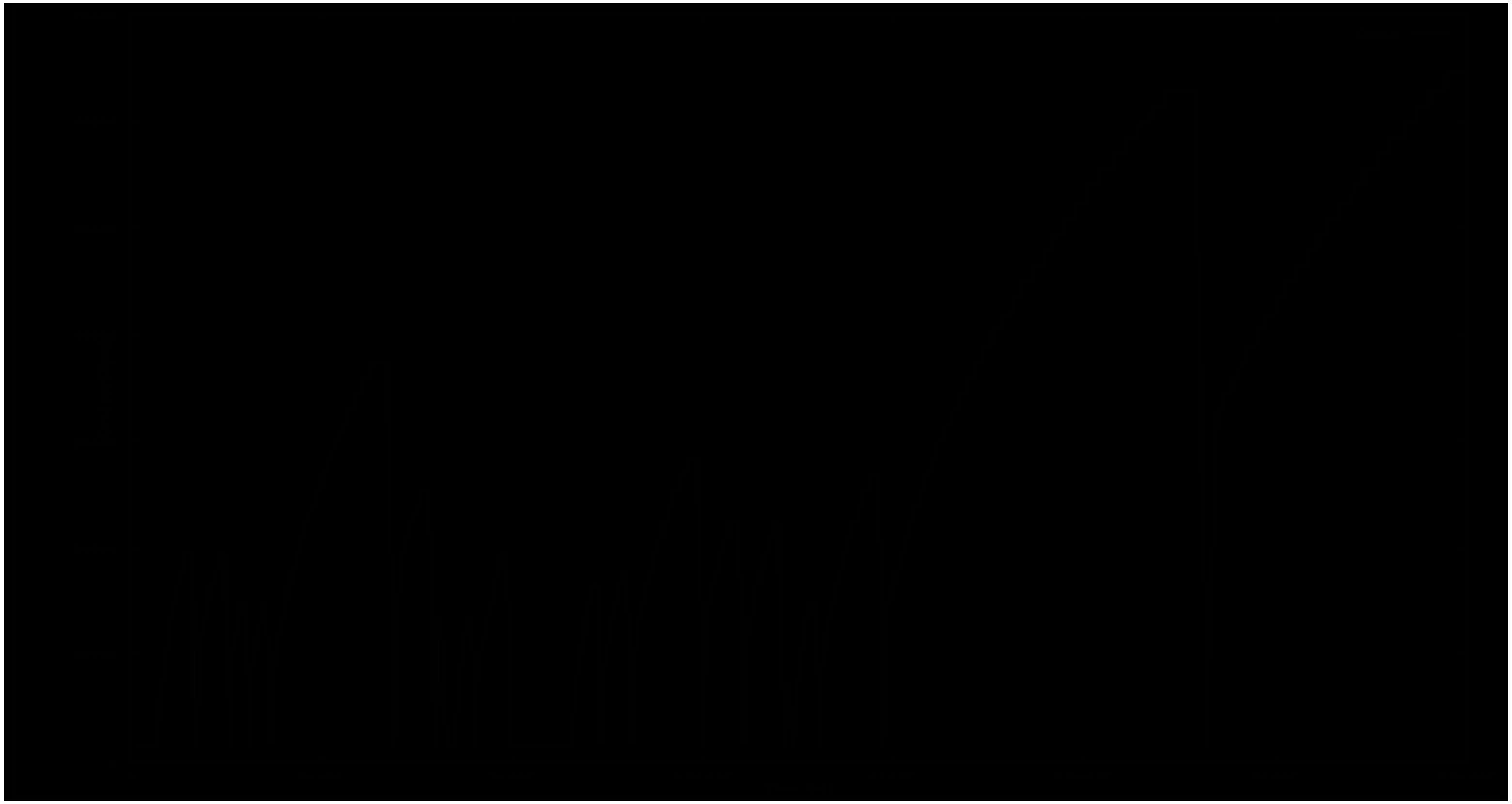
      cwnd = 1
    - and slow-start is entered

# 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)

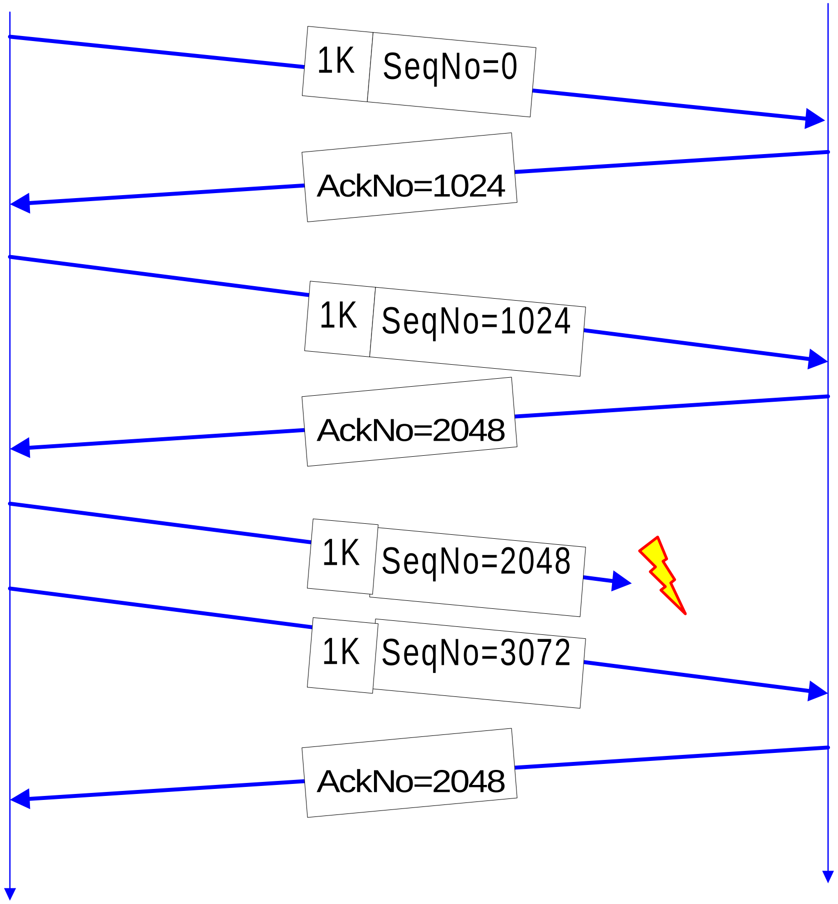
- **RED** (Floyd and Jacobson 1993)

# Acknowledgments in TCP (Review)

- 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)

- ACKs are not delayed when packets are received out of sequence



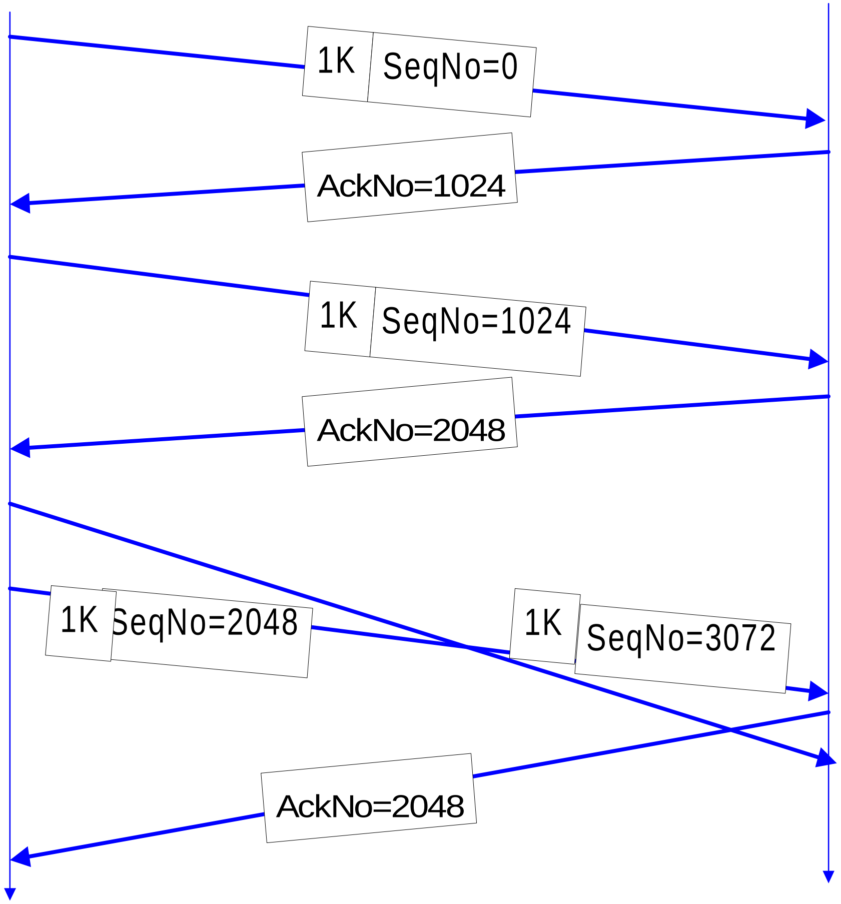
Lost segment

# Acknowledgments in TCP (Review)

- 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)

- ACKs are not delayed when packets are received out of sequence



Out-of-order arrivals

# Fast Retransmit

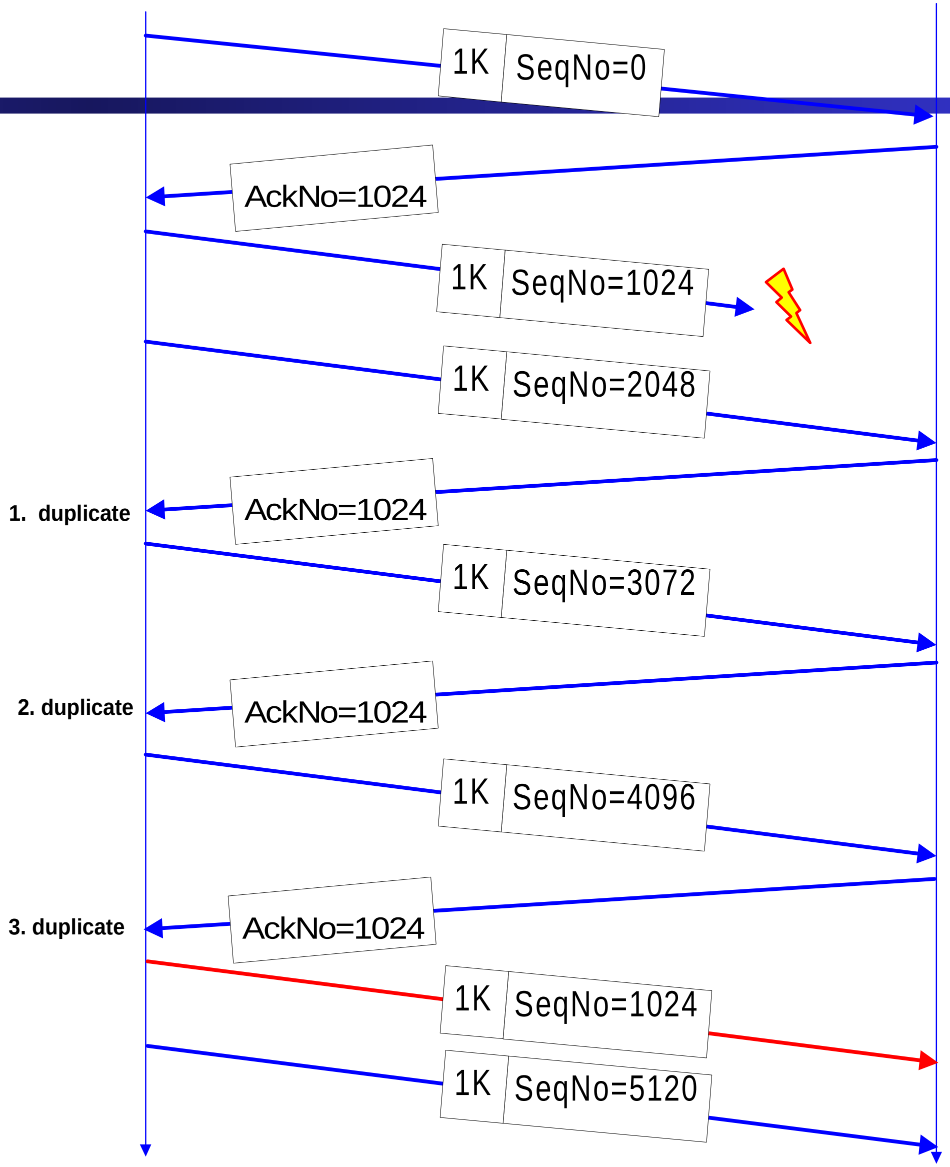
- If three or more duplicate ACKs are received in a row, the TCP sender believes that a segment has been lost.

- 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



1K SeqNo=0

AckNo=1024

1K SeqNo=1024

1K SeqNo=2048

1. duplicate AckNo=1024

1K SeqNo=3072

2. duplicate AckNo=1024

1K SeqNo=4096

3. duplicate AckNo=1024

1K SeqNo=1024

1K SeqNo=5120

# Fast Recovery

- Fast recovery avoids slow start after a fast retransmit

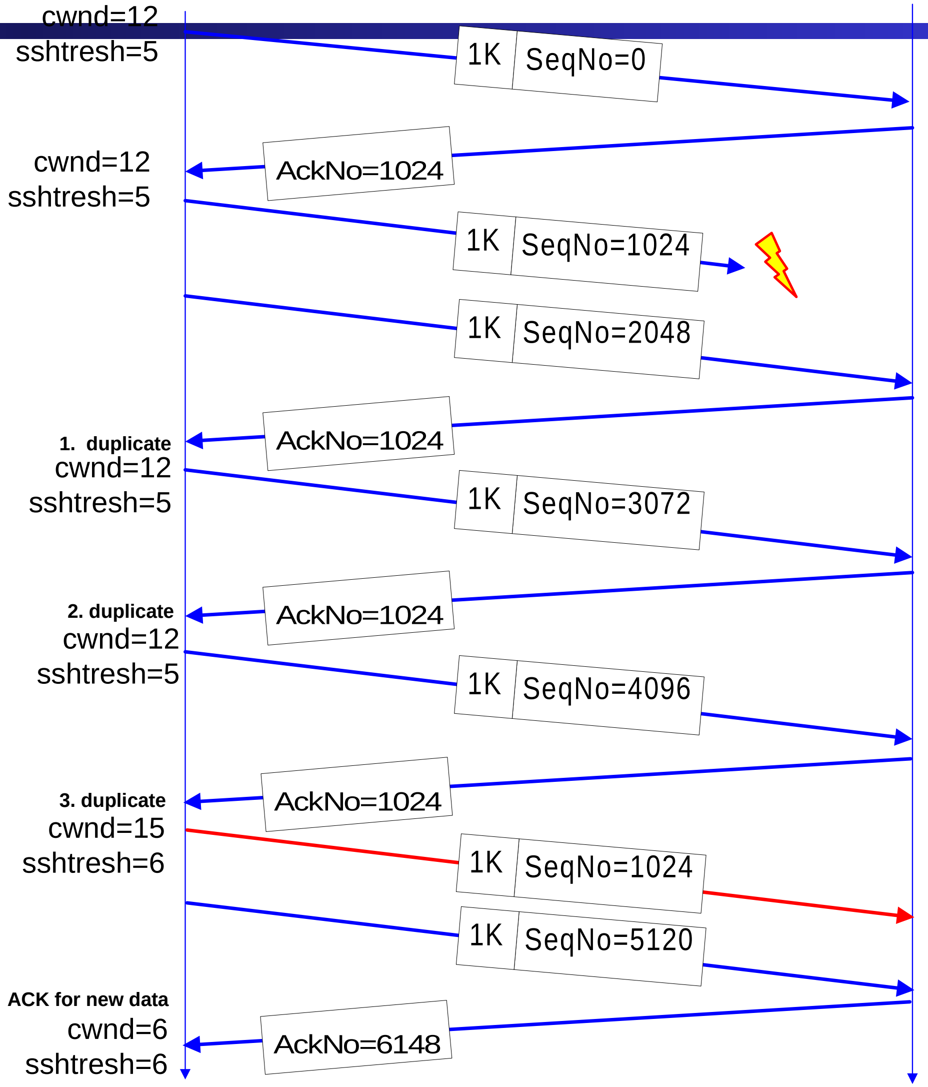
- **Intuition:** Duplicate ACKs indicate that data is getting through

- After three duplicate ACKs set:
  - Retransmit packet that is presumed lost
  - ssthresh = cwnd/2
  - cwnd = cwnd+3
  - (note the order of operations)
  - Increment cwnd by one for each additional duplicate ACK

- When ACK arrives that acknowledges "new data" (here: AckNo=6148),

  cwnd=ssthresh

  enter congestion avoidance



# TCP Reno

- Duplicate ACKs:
    - Fast retransmit
    - Fast recovery

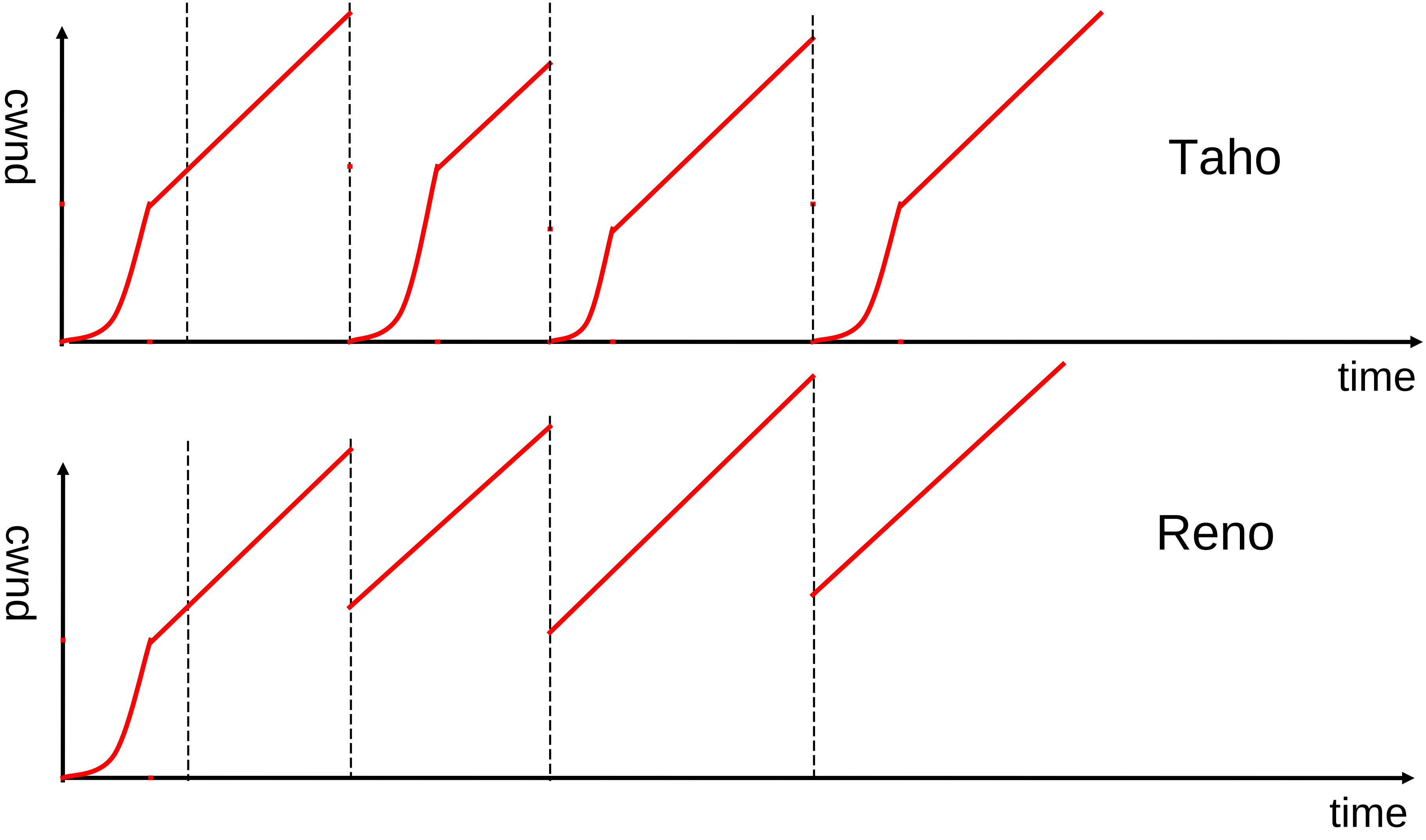
  → Fast Recovery avoids slow start

- 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 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.

# TCP Options

- Maximum Segment Size (MSS)
- Window scaling
- Selective Acknowledgements (SACK)
- Timestamps
- NOP

# Security holes in TCP

**TCP SEQUENCE NUMBER PREDICTION**

- First pointed out by R.T. Morris of AT&T Bell Laboratories
- TCP sequence number prediction can be used to construct a TCP packet sequence without ever receiving any responses from the server
- This allows to spoof a trusted host on a local network

# Security holes in TCP – Blind Spoofing

**Example: Usual Scenario for setting up a TCP connection**

**C -> S:SYN(ISN C)**

**S -> C:SYN(ISN S) , ACK(ISN C)**

**C -> S:ACK(ISN S)**

**C -> S:data**

**and / or**

**S -> C :data**

Example: Unwanted Intrusion by X

(X can predict ISN S)

X -> S:SYN(ISN X) , SRC = T

S -> T:SYN(ISN S) , ACK(ISN X)

X -> S:ACK(ISN S) , SRC = T

X -> S:ACK(ISN S) , SRC = T, nasty – data

*the message S -> T does not go to X,*

*However, X was able to know its contents*

# Security holes in TCP

- In Berkeley systems, the ISN is incremented by a constant amount once per second
- Thus, if one initiates a legitimate connection and observes the ISN S used, the ISN S used on the next connection attempt can be easily predicted
- When T receives the SYN message from S, it attempts to reset the connection
- By impersonating a server port on T, and by flooding that port with apparent connection requests, a queue overflow may be generated and the S -> T message would be lost
- Alternatively the attack may occur when T is down for routine maintenance

# Security holes in TCP - Solutions

- Prediction of ISN may be difficult if the ISN is incremented at a rapid rate
  - TCP specification requires that this variable be incremented approximately 250,000 times per second
  - Berkeley system uses a much slower rate
- Another solution is randomizing the increment
- However, with less number of bits for randomization, the attacker can simply compute the next random number
- With some pseudo random number generators, it is possible to determine the seed and predict the random number
- The other possibility is to use a cryptographic algorithm for *ISN S* generation
  - The Data Encryption Standard (DES) is an attractive choice as the *ISN S* source
- Performance of the initial sequence number generator is not an issue as new sequence numbers are needed only once per connection

28, 10, 20, 26, 11, 32