

High speed computer networks

TCP traffic control

Sosina M.
Addis Ababa institute of technology (AAiT)
2012 E.C.

Transport layer

- End-to-end data transfer service
- Protocols
  - Connection oriented (TCP)
    - Connection establishment and release
    - Reliable
    - Flow and error control
  - Connectionless (UDP)
    - No need for connection establishment and connection release
    - No Flow and error control
    - For applications that do not need reliability
    - Fast service

Connection-oriented transport protocol

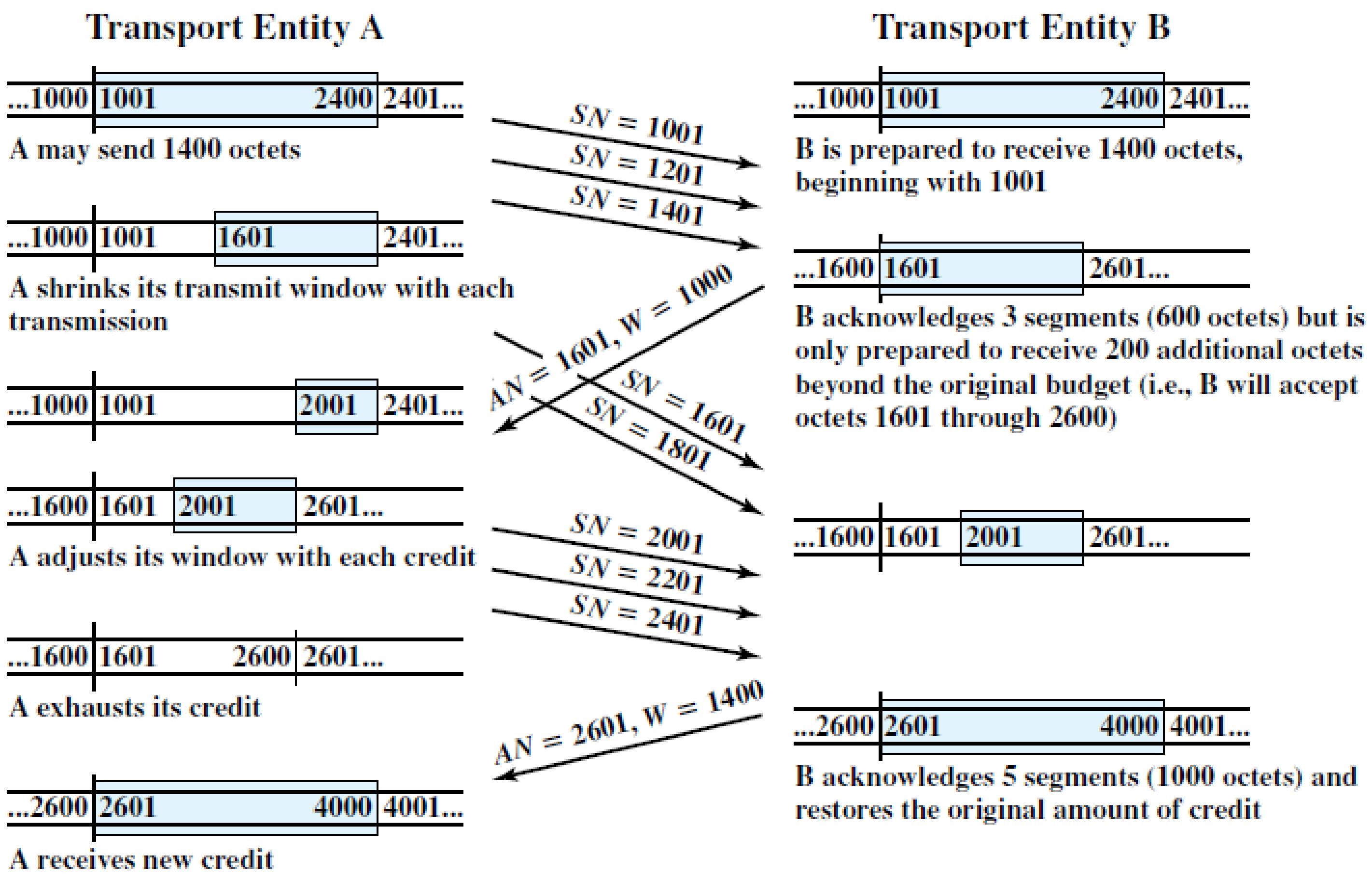
- Logical connection establishment, maintenance and termination
- Functions
  - Addressing
    - (Host, port)
  - Multiplexing
    - Multiple processes employ the same transport protocol
    - Distinguished by port numbers
  - Flow control

TCP flow control

- Sliding window mechanism
- Decouples acknowledgement from flow control
- Applies a credit scheme
  - A segment may be acknowledged without granting new credit
  - Individual octet (byte) of data have a unique sequence number
  - Header of each transmitted segment includes
    - Sequence number (SN) – the sequence number of the first octet
    - Acknowledgment number (AN)
    - Window (W)

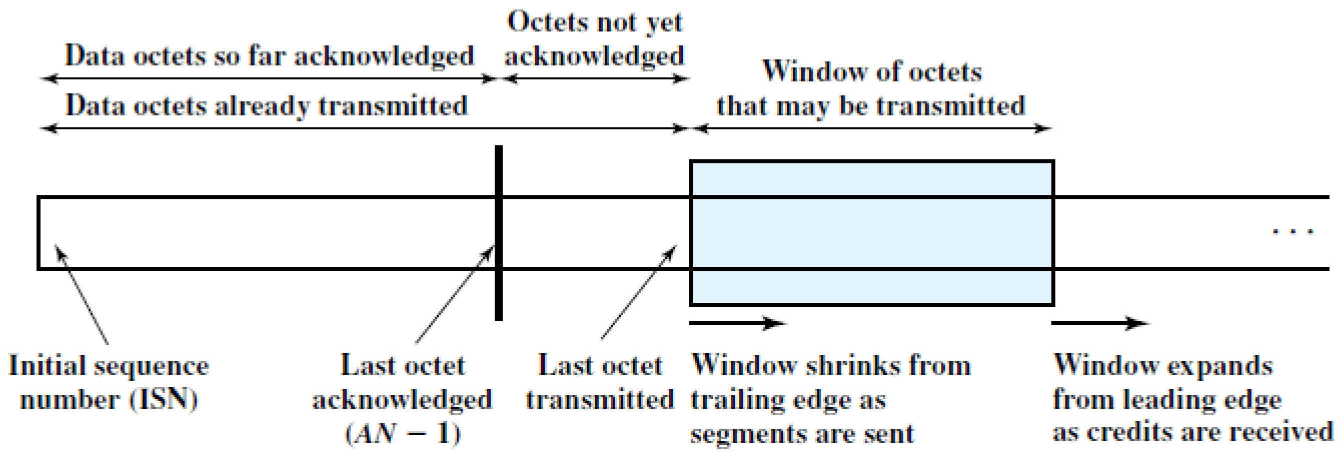
TCP credit allocation mechanism

- E.g., 200 byte of data are sent in each segment, initial credit=1400 byte

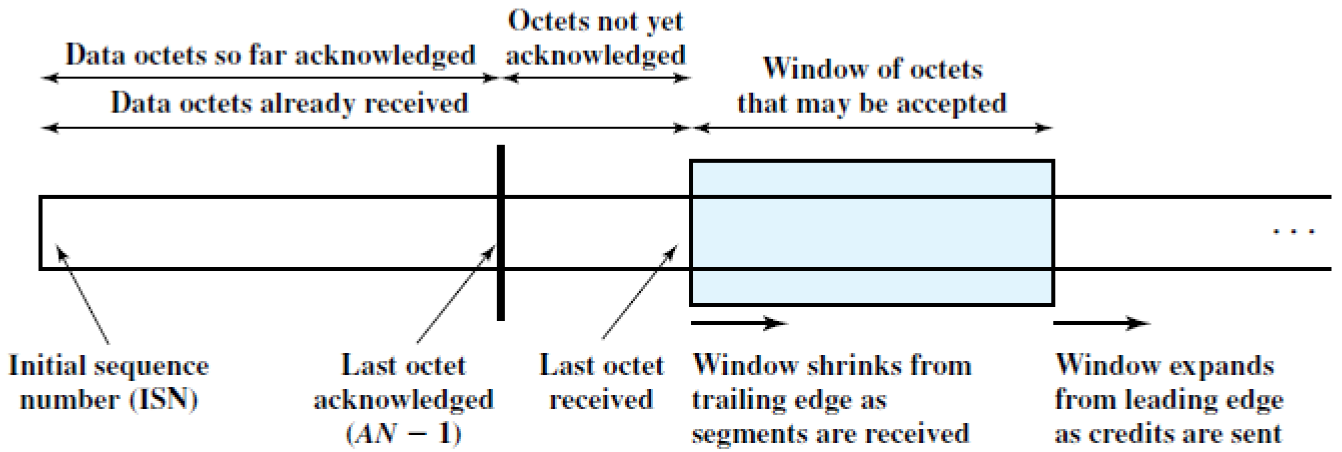


- Initial credit=j
  - ✓ To increase credit to k (K>J) when no new data
    - ▶ B issues AN=i, W=k
  - ✓ To acknowledge segment containing m octets (m<j)
    - ▶ B issues AN=i+m, W=j-m

Send and receive windows



(a) Send sequence space



(b) Receive sequence space

Credit policy

- How much credit a receiver shall give to a sender?
- **Conservative approach**
  - Up to the limit of available buffer space
  - Limit the throughput of the transport connection in long delay situations
- **Optimistic approach**
  - Grant credit for space it doesn't have
    - ✓ Based on the anticipated space release within a round-trip propagation
  - If the receiver can keep up with the sender – this scheme may increase throughput

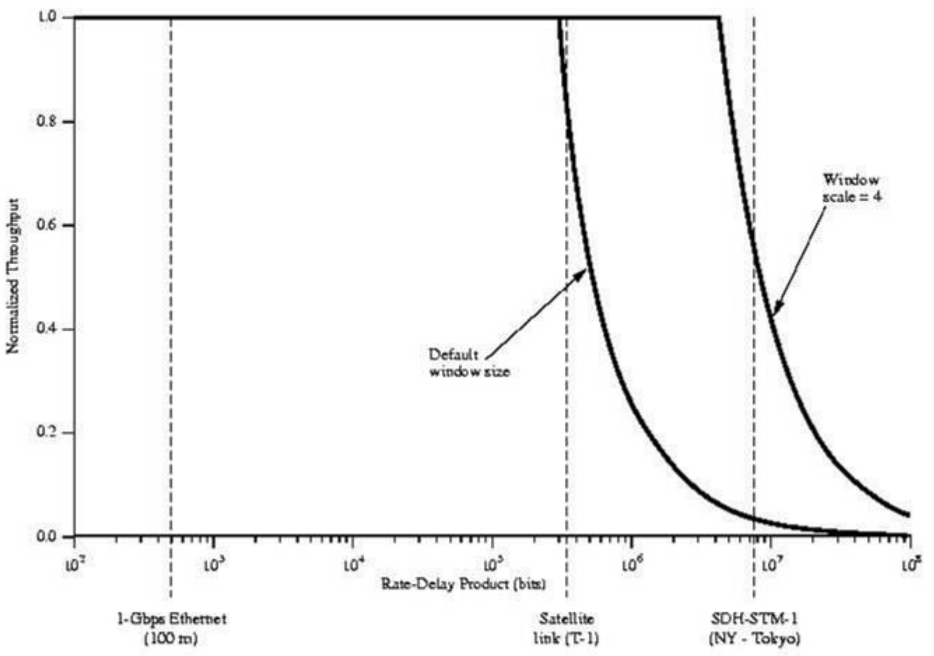
Effect of window size

- Throughput depends on
  - Window size (W)
  - Propagation delay (D)
  - Data rate (R)
- Suppose that a source TCP entity begins to transmit
  - D second for the first octet to arrive at destination + D for acknowledgement to return
  - Within 2D time the source could transmit 2DR bits (DR/4 byte)
  - The source is limited to window size of W octets until ACK is received

Effect of window size

- Normalized throughput (s)

$$S = \begin{cases} 1 & W > RD/4 \\ \dfrac{4W}{RD} & W < RD/4 \end{cases}$$



Normalized throughput VS. rate delay production (The maximum window size=$2^{16} - 1$)

Remarks

- A number of TCP connections are multiplexed over the same network interface
  - Each connection gets the fraction of the available capacity
  - Reduces R => reduces inefficiency
- D= delay across each network + delay at each router
- If R > the data rate encountered on one the hops from Source to Dest.
  - Creates a bottleneck en route => increasing D
- Lost segments are retransmitted
  - Throughput is reduced

Retransmission strategy

- TCP relies on positive acknowledgement
  - Retransmission when an ACK does not arrive within a given time out period
- Retransmission
  - A damaged segment received by a destination
  - Segment fails to arrive

- *At what value should the retransmission time be set?*

Retransmission timer

- Retransmission timer- a key design issue in TCP
  - Too small value →unnecessary retransmission
  - Too large value →delay to respond to lost segments

- Retransmission timer ≈ round time delay
- Approaches
  - **A fixed timer value**
    - ✓ Unable to respond to changing network conditions
  - An adaptive scheme

Adaptive transmission timer

- Based on the pattern of delay for recent segments
  - Set the timer to a value somewhat greater than the estimated round trip delay
- A simple averaging method

$$ARTT(K+1) = \frac{1}{k+1}\sum_{i=1}^{k+1} RTT(i)$$

$$ARTT(K+1) = \frac{k}{k+1} ARTT(K) + \frac{1}{k+1} RTT(k+1)$$

Weighted averaging

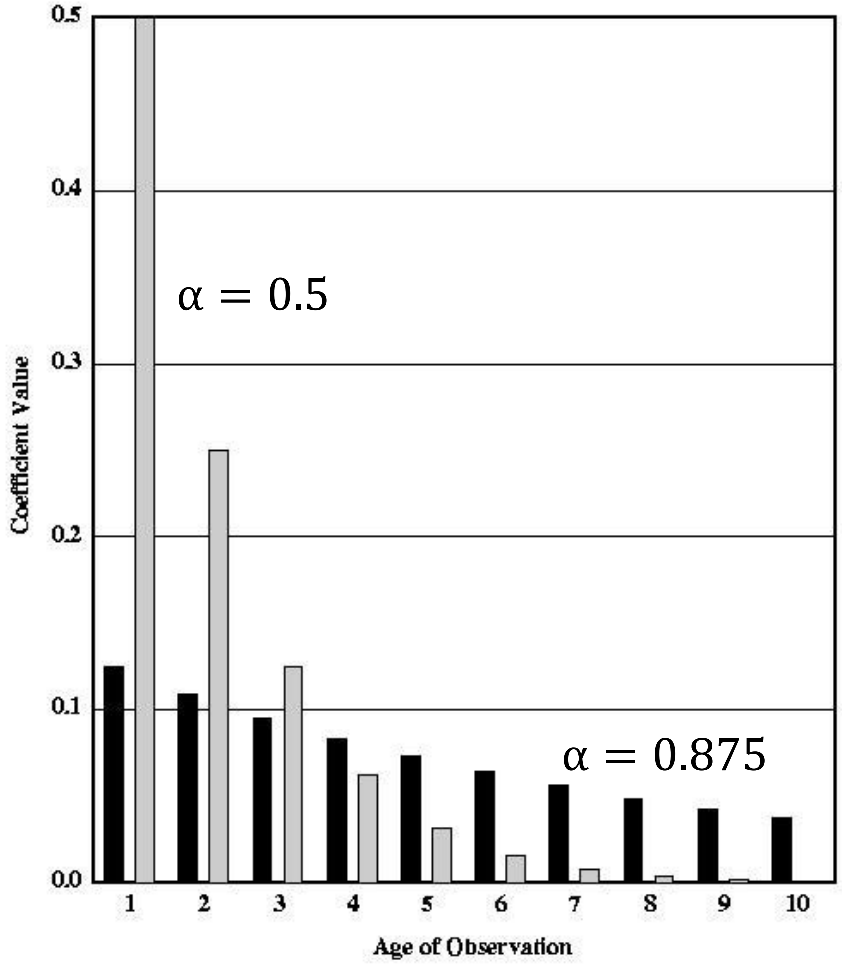
- Gives greater weight to more recent instances
  - More likely to reflect future behavior

$$SRTT(k+1)=\alpha*SRTT(k)+(1-\alpha)*RTT(k+1)$$

- ✓ SRTT(k+1)=smoothed round-trip time estimate
  - ***Exponential smoothing coefficient***

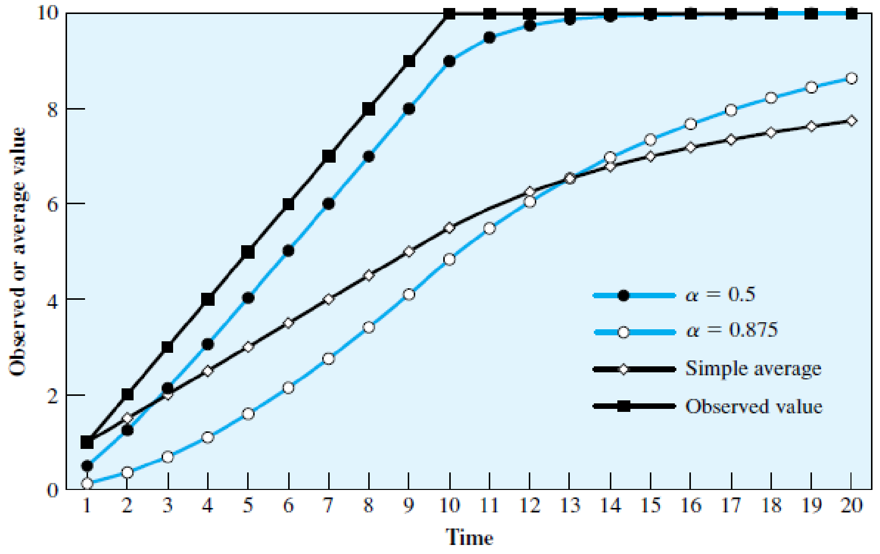
$$SRTT(k+1)=(1-\alpha)*RTT(k+1)+\alpha(1-\alpha)*RTT(k)+\alpha^2(1-\alpha)*RTT(k-1)+\ldots+\alpha^k(1-\alpha)*RTT(1)$$

Exponential smoothing

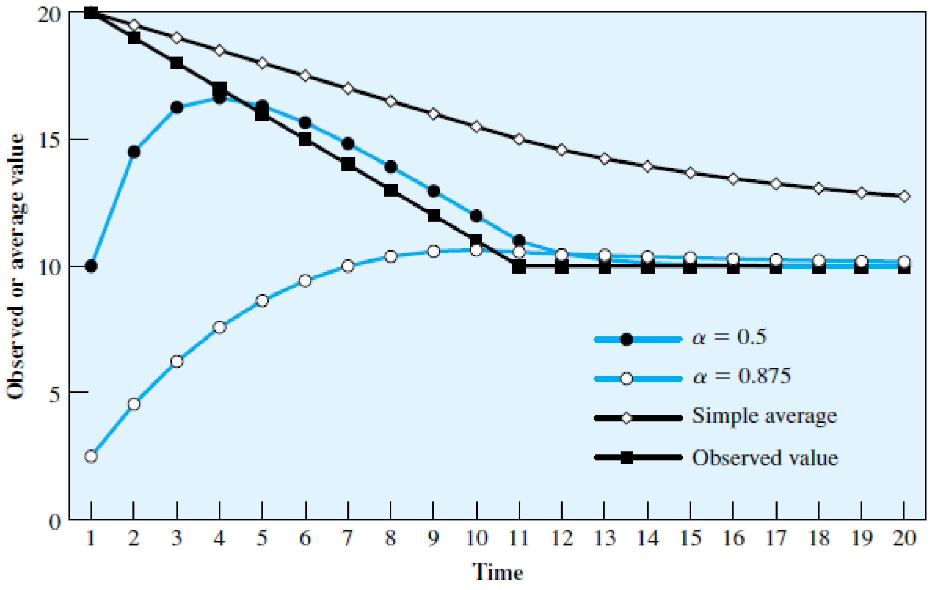


- A small value of α – can reflect a rapid change
  - Results in jerky changes

Simple averaging VS. exponential averaging



(a) Increasing function



(b) Decreasing function

Retransmission timeout

- RFC 793 specifies the use of a timer whose value is proportional to SRTT
  - $RTO(k+1)=MIN(UBOUND, MAX(LBOUND, \beta * SRTT(k+1)))$

TCP implementation policy options

- The design areas for which possible implementation options are specified
  - Send policy
    - TCP may construct a segment for each batch of data or may wait until a certain amount of data accumulates
  - Deliver policy
    - Deliver data as each in-order segment is received or may buffer data from number of segments before delivery
  - Accept policy
    - Accept only segments that arrive in order or accept all segments that are within the receive window

TCP implementation policy options

- Retransmission policy
  - First only – retransmit the segment at the front of the queue
  - Batch – retransmit all segments in the queue
  - Individual
    - ✓ One timer for each segment
    - ✓ Retransmit the corresponding segment individually
- Acknowledge policy
  - Immediate
  - Cumulative

TCP congestion control

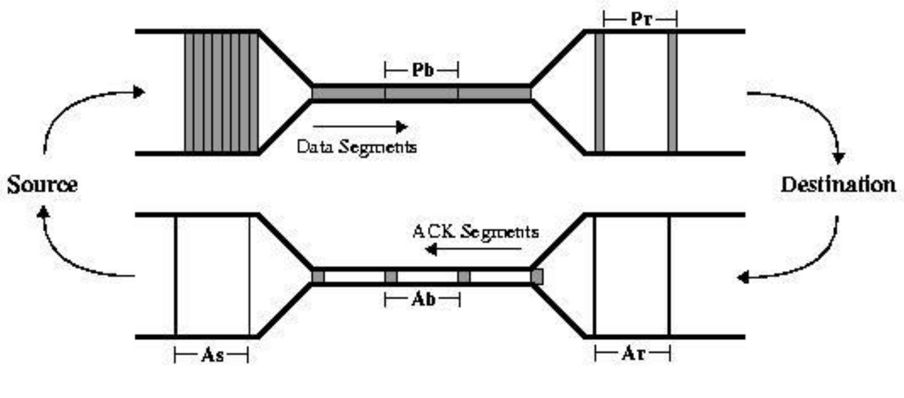
- Congestion => delay and packet drops
- Solutions for unbalanced load
  - Packet switched networks – dynamic routing
  - Routing algorithm – spread load among routers and networks
- Congestion control – limiting the total amount of data entering the network to the amount that the network can carry
- The tool in TCP that control congestion – **sliding window flow and error control**
  - Designed for management of end-to-end traffic
  - Employing this mechanism for congestion control

TCP self clocking behavior

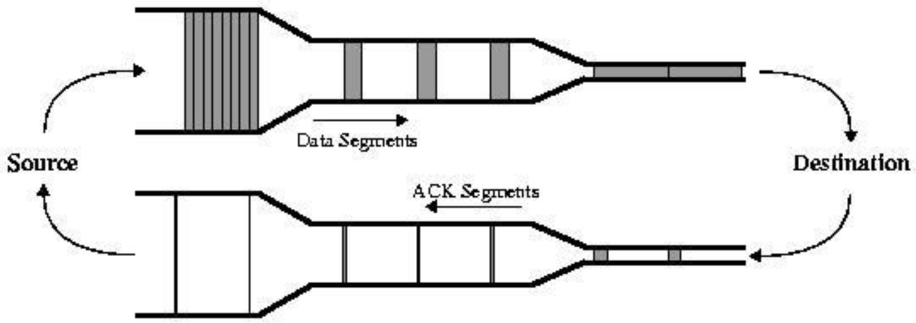
- The rate at which a TCP entity can transmit is determined by the rate of incoming ACKs
- Rate of ACK arrival is determined by round trip path between source and destination
  - The sender's segment rate will match the arrival rate of the ACK
  - The sender rate=the slowest link on the path

- Thus, TCP automatically senses the network bottleneck and regulates its flow accordingly – ***TCP's self clocking behavior***

TCP self clocking behavior



(a) Flow determined by Network Congestion



(b) Flow determined by Destination System

The source has no way of knowing whether ACK rate reflects

- ✓ the status of the network (congestion control)
- ✓ or the destination (flow control)

Using TCP sliding window for congestion control

Retransmission timer management

- The value of retransmission timer (RTO) have a critical effect on TCP's reaction to congestion

- Techniques to compute RTO
  - RTT variance estimation
  - Exponential RTO Backoff
  - Karn's algorithm

RTT variance estimation

- Previously discussed method
  - Enables TCP to adapt to changes in round trip time
  - Doesn’t cope well with a situation in which the round trip time exhibits a relatively high variance
- Source of variance in RTT
  - Data rate and variance in IP datagram size =>SRTT is heavily influenced by the property of the data not the network
  - Load may change abruptly due to traffic from other sources
  - The peer TCP may not acknowledge each segment immediately due to processing delays or cumulative ACK

RTT variance estimation

- Original TCP specification tries to account for this variability

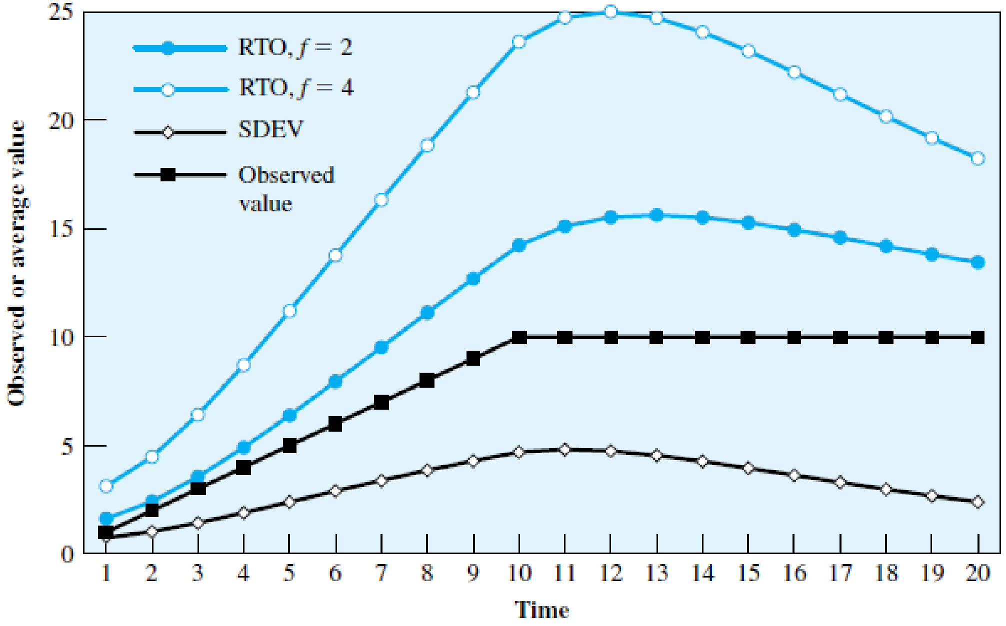
$$RTO(k+1)=\beta * SRTT(k+1) \quad \text{(often } \beta=2 \text{ is used)}$$

  - In stable condition (low variance of RTT)
    - ✓ Results in an unnecessarily high value of RTO
  - Unstable condition
    - ✓ A value of 2 may be inadequate to protect against unnecessary retransmission
- A more effective approach is to estimate the variability in RTT values and use that as input for RTO computation

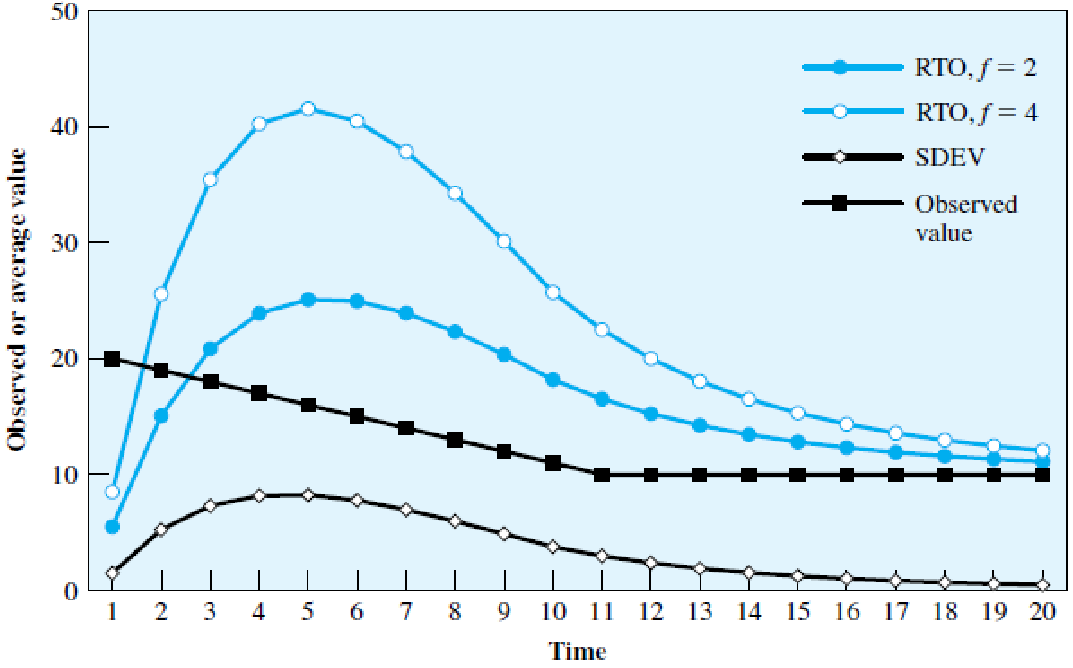
Jacobson’s algorithm

- ***Exponential smoothing, g=1-α***
  - SRTT(k+1)=(1-g)*SRTT(k) + g*RTT(k+1)
- ***Error (deviation from mean)***
  - SERR(k+1)=RTT(k+1)-SRTT(k)
- ***Estimated deviation***
  - SDEV(k+1)=(1-h)*SDEV(K) + h*|SERR(k+1)|
- Then
  - RTO(k+1)=SRTT(k+1)+f*SDEV(K+1)
  - recommended values for the coefficients
    - g=0.125, h=0.25, f=2 0r 4

Jacobson’s algorithm



(a) Increasing function



(b) Decreasing function

Two other factors

- Jacobson’s algorithm can significantly improve TCP performance, but
  - What RTO to use for retransmitted segments?
    - ✓ exponential RTO backoff algorithm
  - Which round-trip samples to use as input to Jacobson’s algorithm?
    - ✓ Karn’s algorithm

Exponential RTO backoff

- Consider the following scenario
  - There are a number of active TCP connections
  - A region of congestion develops => segments are lost or delayed past the RTO time
  - At roughly the same time many segments will be retransmitted => maintaining or even increasing the congestion
  - All source then wait a local RTO time and retransmit again
  - This pattern of behavior could ***cause a sustained condition of congestion***

Exponential RTO backoff

- Retransmission policy
  - Sending TCP entity increases its RTO each time a segment is retransmitted (backoff process)
  - This may give the congested area time to clear the current congestion

- A simple technique
  - Multiply the RTO by a constant value for each retransmission

$$\boxed{RTO(i+1) = q * RTO(i)}$$

- ✓ RTO grows exponentially with each retransmission
- ✓ q=2

karn’s algorithm

- If an ACK is received for retransmitted segment, there are 2 possibilities:
  - *The ACK is for the first transmission of the segment*
    - ✓ RRT is longer than expected
  - *The ACK is for the second transmission of the segment*
- The sending TCP entity cannot distinguish between these two
- ***How to estimate RTT?***
  - From the second transmission?
    - ✓ If the first case is true => the measured RTT will be too small
  - From the first transmission?
    - ✓ If the second case is true => the measured RTT=actual RTT + RTO

karn’s algorithm

- Do not use measured RTT to update SRTT and SDEV
- Calculate backoff RTO when a retransmission occurs
- Use backoff RTO for segments until an ACK arrives for a segment that has not been retransmitted

- *When an acknowledgment is received to an unretransmitted segment, Jacobson’s algorithm is again activated to compute future RTO values*

Windom management

- Slow start
- Dynamic window sizing on congestion
- Fast retransmit
- Fast recovery
- Limited transmit

Slow start

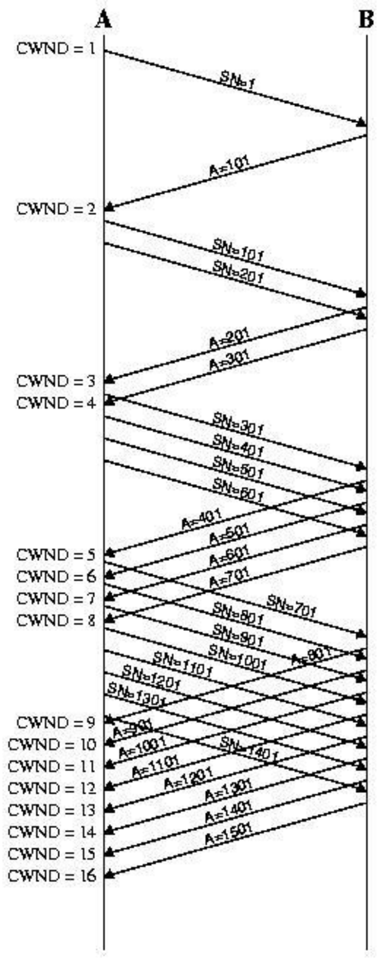
- Larger send window – the more segments are sent before an acknowledgement received
- Solution
  - TCP sender begins from relatively large but not maximum window
    - ✓ The sender might flood the network before it realized from the time out the flow was excessive
  - Gradually expanding the window until ACKs are received (slow start)

Awnd=MIN[credit, cwnd]

- ✓ awnd = allowed window in segments
- ✓ cwnd = congestion window in segments (a window used by TCP during startup and to reduce flow during period of congestion)
- ✓ credit = amount of unused credit granted in most recent ACK

Slow start

- New connection starts with a cwnd=1
- Each time an ACK to new segment is received, the value is increased by 1, up to some maximum value



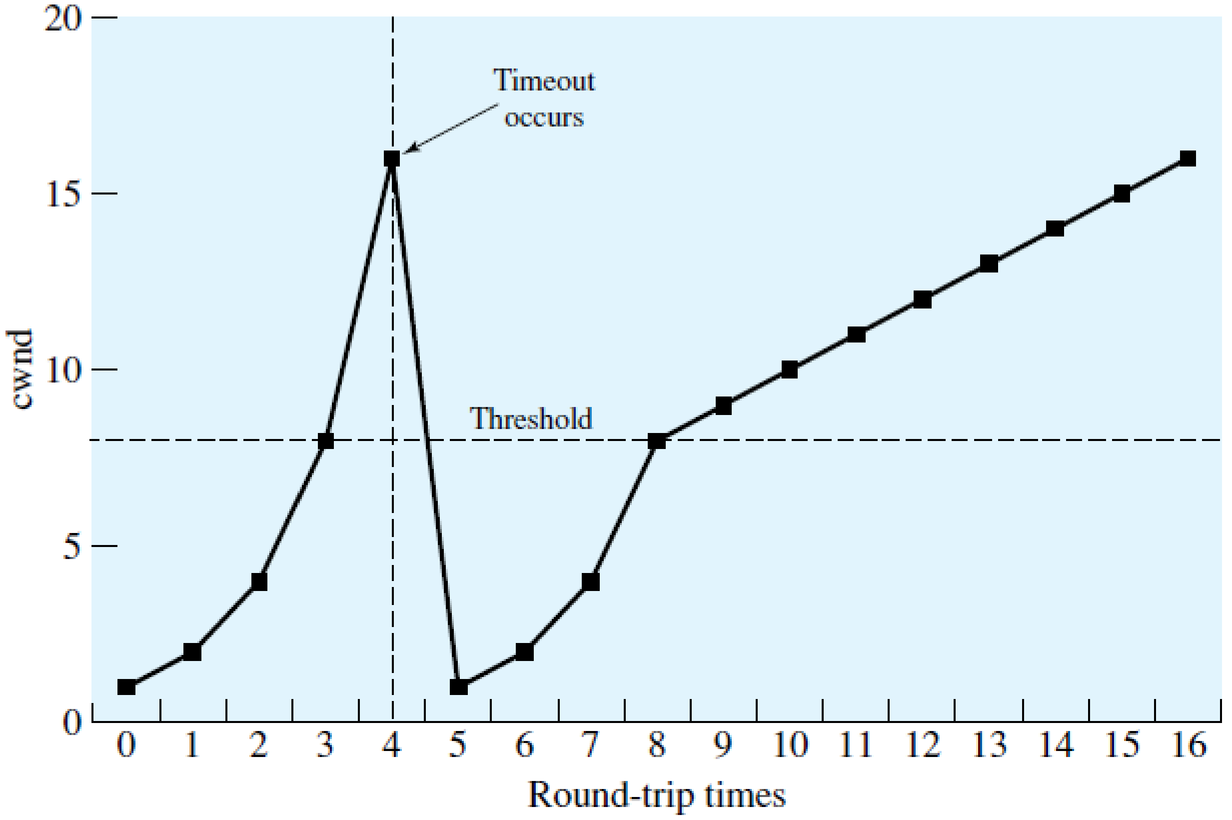
Dynamic window sizing on congestion

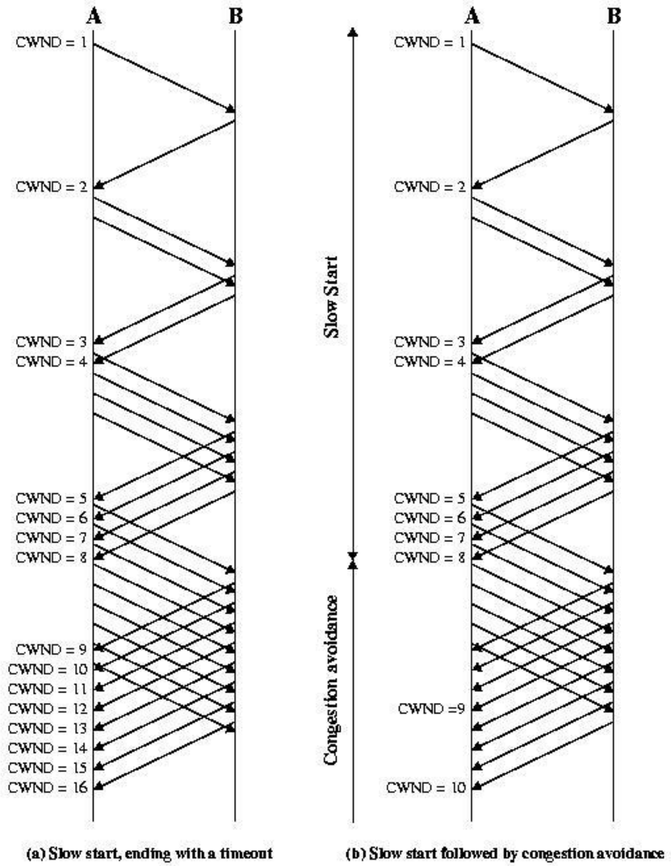
- If a segment is lost at some point
  - Signals congestion
  - Not clear how serious the congestion is
  - A wise approach would be to reset cwnd=1 and begin the slow start process all over
- Exponential growth of *cwnd* under slow start may be to aggressive
- Instead, Jacobson proposed the use ***slow start to begin with***, followed by ***a linear growth***

Dynamic window sizing on congestion

- Set *ssthresh=cwnd/2*
- Set *cwnd=1* and preform the slow start process until *cwnd=ssthresh*
- *For* cwnd$\geq$ ssthresh, increase *cwnd* by one for each round trip time

Dynamic window sizing on congestion





Fast retransmit

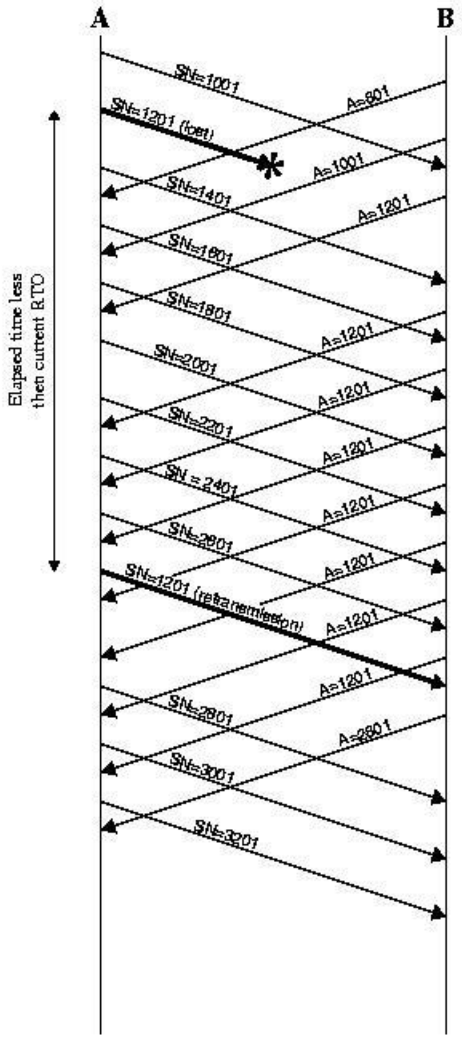
- RTO is generally noticeably longer than actual RTT
- If a segment is lost, TCP may be slow to retransmit

- Suppose that A transmit a sequence of segments
  - B receives all these segment except the first
  - B must buffer all of these incoming segments until the missing one is retransmitted
  - If retransmission is delayed, B will have to begin discarding incoming segments

Fast retransmit

- ***TCP rule***
  - If a TCP entity receives a segment out of order, it must immediately issue an ACK for the last in-order segment that was received
  - TCP repeat this ACK with each incoming segment until the missing segment arrives
- When the source TCP receives a duplicate ACK
  - The segment following the ACKed segment was delayed
  - The segment was lost
- To make sure the duplicate ACK is due to case 2
  - The sender waits until it receives three duplicate ACKs to the same segment
  - Then retransmits the lost segment

Fast retransmit



Fast recovery

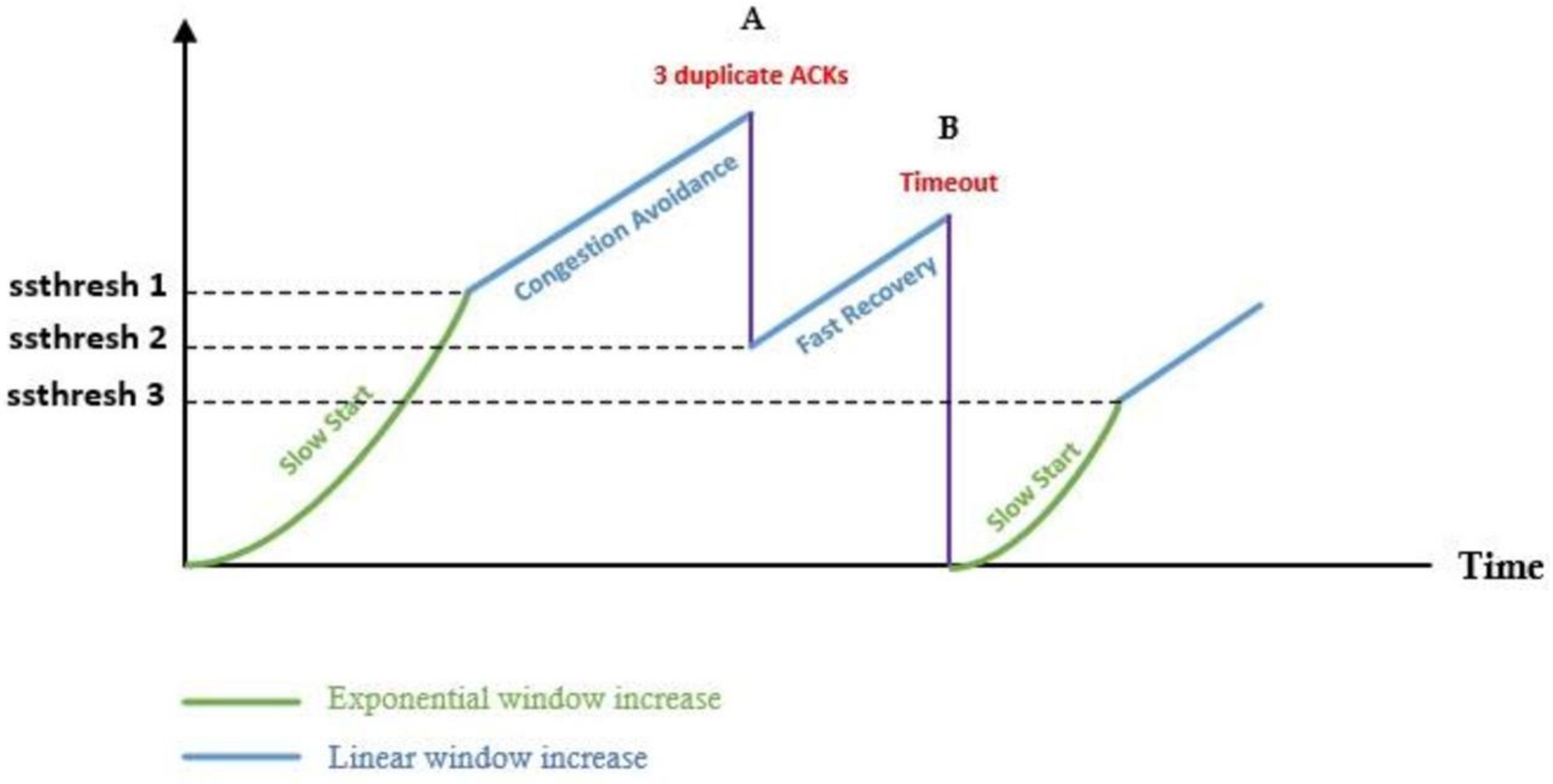
- Fast retransmit assumes that a segment was lost
  - Thus, the TCP entity should take congestion avoidance measures
  - Apply slow-start/congestion avoidance that is used when timeout occurs ?
    - ✓ Unnecessarily conservative
    - ✓ The very fact that multiple ACKs have returned indicates that data segments are getting through fairly regularly to the other side => fast recovery

- Fast recovery
  - Retransmit the lost segment
  - Cut $cwnd$ in half and then proceed with the linear increase of $cwnd$ (avoids initial exponential slow start process)

Fast recovery

- When the third duplicate ACK arrives
  - Set $ssthresh = cwnd/2$
  - Retransmit the missing segment
  - Set $cwnd = ssthresh + 3$
  - Each time an additional duplicate ACK arrives, increment cwnd by 1
  - When the next ACK arrives that acknowledge new data, set $cwnd = ssthresh$

Fast recovery



Limited transmit

- TCP implementation
  - Adaptive retransmission time and fast retransmit
- If the cwnd at the TCP sender is small, the fast retransmit mechanism may not be triggered
  - Example cwnd=3

Limited transmit

- Several questions arise
  1. Under what circumstances does sender have small congestion window?
     - ✓ Limited amount of data to send
     - ✓ The receiver impose small limit on the credit it grants
     - ✓ Small rate-delay product
  2. Is the problem common?
     - ✓ 56% retransmission due to RTO, with only 44% handled by fast retransmit
  3. If the problem is common, why not reduce number of duplicate ACKs needed to trigger retransmit?
     - ✓ Duplicate ACKs may result from segment reordering

Limited transmit

- Sender can transmit new segment when 3 conditions are met:
  - Two consecutive duplicate ACKs are received=> a total of three ACKs
  - Destination advertised window allows transmission of segment
  - Amount of outstanding data after sending is less than or equal to cwnd + 2

TCP variants

Different versions of TCP

- Implementation of TCP congestion control measures

Measure	RFC 1122	TCP Tahoe	TCP Reno	NewReno
RTT Variance Estimation	√	√	√	√
Exponential RTO Backoff	√	√	√	√
Karn's Algorithm	√	√	√	√
Slow Start	√	√	√	√
Dynamic Window Sizing on Congestion	√	√	√	√
Fast Retransmit		√	√	√
Fast Recovery			√	√
Modified Fast Recovery				√

TCP Tahoe

- ***Algorithms***
  - RTT estimator
  - Slow start
  - Dynamic window sizing (congestion avoidance)
    - ✓ Retransmit, *set ssthresh,* enter slow start phase
  - Fast retransmit
    - ✓ Based on duplicate ACKs threshold – generally set to three

Tahoe algorithm

Initially:

cwnd = 1;

ssthresh = infinite;

New ack received:

if (cwnd < ssthresh)

→ Slow Start

cwnd = cwnd + 1;

else

→ Congestion Avoidance

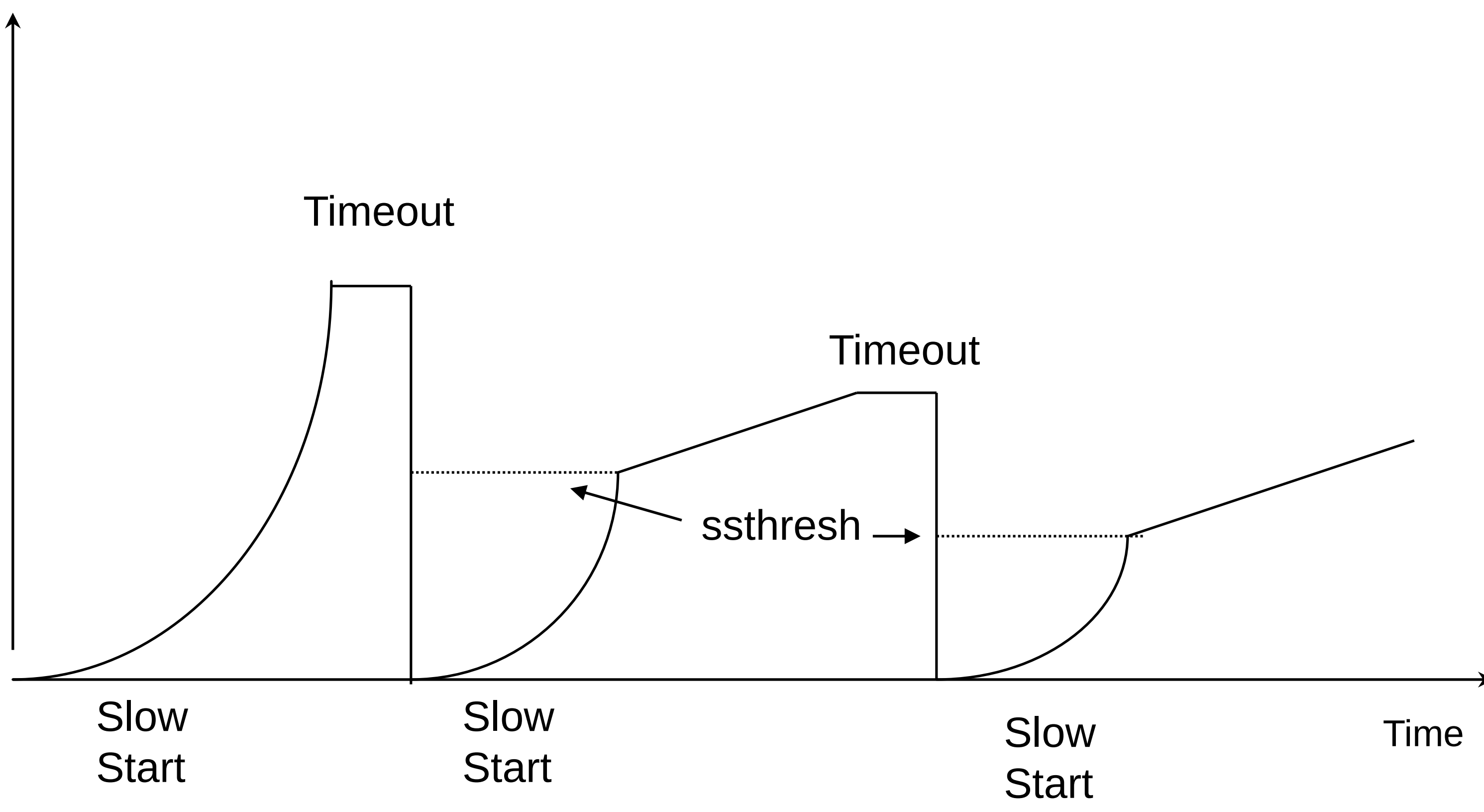
cwnd = cwnd + 1/cwnd;

Timeout:

→ Multiplicative decrease

ssthresh = cwnd/2;

cwnd = 1;



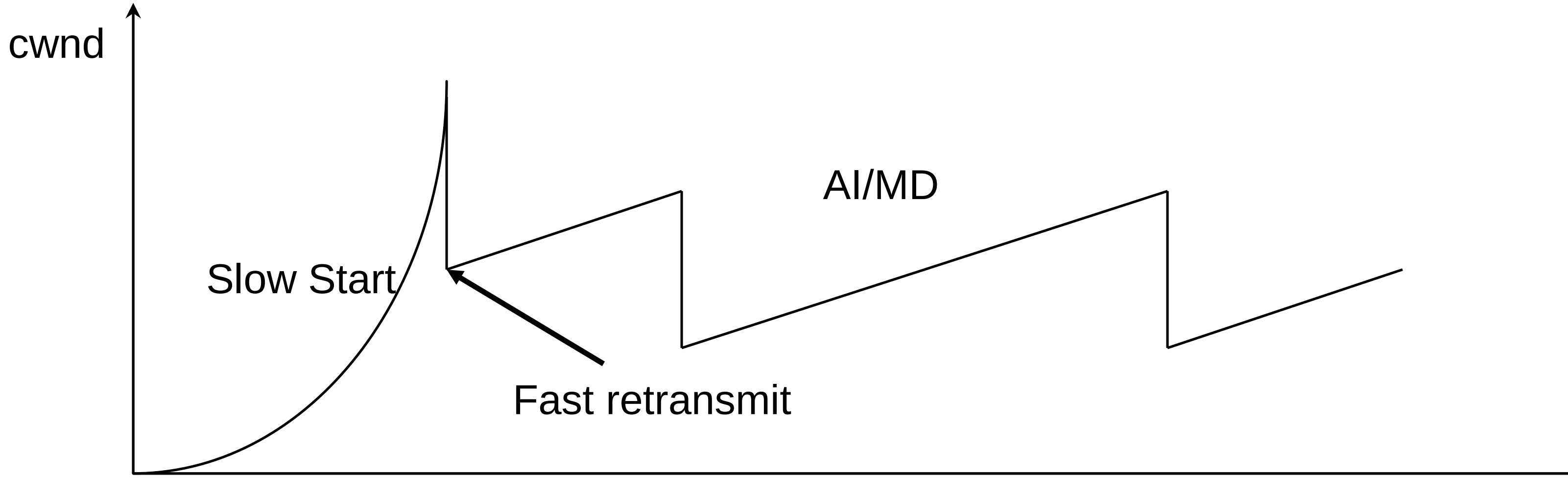
TCP Reno

- Retained the enhancements incorporated into TAHOE

- ***The fast retransmit operation is modified to include fast recovery***
  - Applies intelligent estimates of the amount of outstanding data
  - Fast recovery is entered after receiving threshold of dup ACKs
  - The sender
    - ✓ Sets $ssthresh$ = $cwnd$/2
    - ✓ retransmits one packet
    - ✓ *cwnd=ssthresh + 3*
    - ✓ "inflates" its window by the number of dup ACKs it has received
    - ✓ effectively waits until ***half a window of dup ACKs*** have been received, and then sends a new packet for each additional dup ACK that is received
    - ✓ Exists fast recovery upon receipt of an ACK for new data

TCP Reno

- Algorithm
  - If three duplicate ACKs are received
    - ✓ Set ssthresh=current cwnd/2, cwnd=cwnd/2 +3
    - ✓ Retransmit
    - ✓ If a new duplicat ACK
      - cwnd=cwnd +1
      - If cwnd > the amount of data – transmit new segment
      - Else wait
    - ✓ If fresh ACK
      - Exit fast recovery
    - ✓ If timeout
      - cwnd=1



TCP Reno

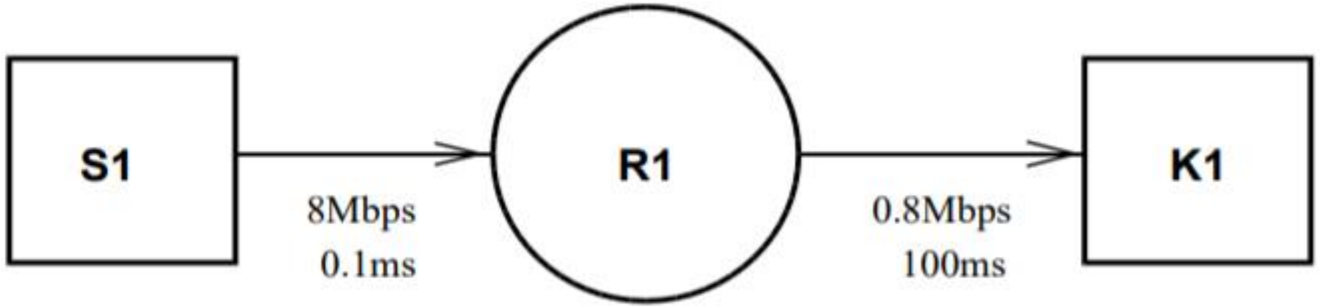
- Reno's Fast Recovery algorithm is optimized for the case when ***a single packet is dropped from a window of data***
  - Significantly improves the behavior of Tahoe TCP when a single packet is dropped from a window of data,
  - but can suffer from performance problems when multiple packets are dropped from a window of data

New-Reno TCP

- Reno doesn't improve much upon Tahoe if there are multiple packet losses in the same window
  - When multiple packet losses occur
    - ✓ Reno enters fast recovery multiple times which decreases the congestion window by half every time
- New Reno
  - TCP stores the sequence number of the highest data packet which is sent when the third duplicate ACK arrives
  - Exists fast recovery when it receives an ACK which is higher than the sequence number of the highest data packet

Example: TCP performance analysis

- Links – bandwidth capacity and delay
- Number of TCP connection from S1 to k1
  - The number of segment sent by each connection



Project – part 1

- Form a group of 3
- ***Analyzing the performance of TCP using NS3***
  - Try to understand how RTT, CWND, fast retransmit and recovery are implemented
  - Compare the different versions of TCPs integrated in NS3
  - Create a simple scenario (adapt one of NS3 TCP test setup) and analyze the performance of TCP
    - ✓ Configure the parameter (bandwidth, latency, etc.)
    - ✓ Perform tests on RTT and throughput by changing the parameters