TCP congestion control

TPC congestion control

TLC Networks Group
firstname.lastname@polito.it
http://www.tlc-networks.polito.it/

TCP protocol

• TCP (Transmission Control Protocol)
• Already reviewed
  – Fundamentals
  – Port mechanism
    • Socket
  – Header format
TCP congestion control

References

- Richard Stevens: TCP Illustrated
- RFC 793 (1981)
  - Transmission Control Protocol
  - Updated by RFC 3168 (ECN) RFC 6093, RFC 6528
- RFC 7323 (updates RFC 1323 in 1992)
  - TCP Extensions for High Performance
- RFC 5681 (obsoletes RFC 2581):
  - TCP Congestion Control
- RFC 6582 (obsoletes RFC 3782 and RFC 2582):
  - The NewReno Modification to TCP's Fast Recovery Algorithm
- RFC 2883 (obsoletes RFC 2018 defined in 1996):
  - An Extension to the Selective ACKnowledgement (SACK) Option for TCP
- RFC 6298 (obsoletes RFC 2988):
  - Computing TCP's Retransmission Timer

TCP connection opening
(three-way handshake)

- Three-way handshake
- Client executes an active open, server executes a passive open
- ISN (initial sequence number) is randomly generated
- A SYN uses one sequence number

source (client)                        destination (server)

SYN / ISN<sub>c</sub>  <MSS<sub>c</sub>>

SYN / ISN<sub>s</sub>  <MSS<sub>s</sub>>

ACK / ISN<sub>c</sub>+1

ACK / ISN<sub>s</sub>+1

time
TCP connection closing (half-close)

source (client)   
\[\text{close request from application}\]  
\[\text{FIN / Seq. Numb}_{c}\]  
\[\text{ACK (Seq.Numb}_{c+1}\]  
\[\text{EOF to higher layers}\]

destination (server)  
\[\text{close request from application}\]  
\[\text{FIN / Seq. Numb}_{s}\]  
\[\text{ACK (Seq. Numb}_{s+1}\]  
\[\text{EOF to higher layers}\]

Connection management: client

invia SYN

SYN_sent

Send ACK

established

closed

Time_wait

Wait for 30 s

FIN_sent

send data; client side ends connection sending FIN

FIN_wait_1

Re-send if timeout expires.

FIN_wait_2

receive ACK

receive FIN send ACK

10 minutes if server idle

Re-send ACK If a new FIN arrives

Re-send if Timeout expires (3s+backoff)
Notes

- The **Timed_wait** state avoids that old segments belonging to closed connections may interfere with new connections.
- **Timed_wait** should be “aligned” to TTL, today a timer set to 30s is used.
- During the **Timed_wait** state, socket (ports) cannot be used.
- BSD implementation passes from **FIN_wait_2** to **closed** in 10 minutes, of the server does not send any data in the meantime.

Connection management: server

- Server application creates socket
- Receive SYN, send SYN & ACK
- Re-send if timeout expires
- Receive FIN, send ACK
- (client connection closed)
- Receive ACK
- Re-send if timeout expires
- (3s+backoff)
- Closed
- Last ACK
- Send FIN
- Send ACK
- Established
- Listen
TCP State Transition Diagram

- Example of TCP connections opened by the client and closed by the server

Self-clocking behavior

- Segments are spaced within a RTT according to the bottleneck link rate
TCP transmitter

- Fragments data application in segments
- Computes and transmits checksum over header and data
- Window with Go BACK N retransmission (but!)
- Activates timer when sending segments:
  - Unacknowledged segments induce retransmissions after a timeout expiration
- Like any window protocol, transmission speed ruled by window size
  - Flow and congestion control

TCP receiver

- Discards segments with CRC errors
- Stores out of sequence segments
  - Selective repeat like behaviour
- Re-orders out of sequence segments
  - Delivers an ordered and correct data stream to application process
- Cumulative ACKs
- Declares in the window field of the TCP header the amount of available buffer space to control transmitter sending rate (flow control)
TCP congestion control

TCP receiver

- In sequence and correct segment
  - Store the segment (eventually passing it to higher layer protocols) and send a cumulative ACK

- Duplicate segment
  - Discard the segment and send a cumulative ACK with the number of the last segment received in sequence

- Segment with checksum error
  - Discard the segment; no ACK sent

- Out of sequence segment
  - Store the segment (non mandatory, but de facto standard) and send a cumulative ACK with the number of the last segment received in sequence (duplicate ACK)

Transmitter window

Segment sequence number
Segments transmitted, ACK received
Segments transmitted, ACK not received
Segments not yet transmitted
Segments that cannot be transmitted
Maximum admissible window size
Available window
Transmitter window dynamics

- When an ACK referring to a new segment is received, the transmitter window:
  - Move to the right by the segment size
  - It is possible to transmit a new segment
- When a new segment is transmitted, the available window is reduced by a segment
- If the available window goes to zero, segment transmission is stopped

Flow and congestion control

- For any window protocol, the transmission bit rate in absence of errors is:
  \[ \frac{\text{Transmission window}}{\text{Round trip time}} \]
- “Short” connections obtain higher bit rate
- To regulate transmission bit rate (objective of both flow and congestion control), control
  - Round trip time (delay ACK transmission)
    - Generates retransmissions
  - Transmitter window size
Flow and congestion control

- TCP: transmitter bit rate regulated by both:
  - Flow control
  - Congestion control
- Flow control: avoid to saturate a slow receiver
- Congestion control: avoid to saturate the network (more precisely, one link which becomes the bottleneck link)
  - Data are stored in node buffer
    - Under congestion
      - Buffer occupancy increases
      - Round trip increases, and bit rate decreases
      - Normally it is not enough to control congestion

Flow and congestion control

- TCP transmitter window size is regulated:
  - Flow control: receiver declares the available window size (available receiver buffer)
  - Congestion control: the transmitter computes a congestion window (cwnd) value as a function of segment losses detected by missing ACKs
    - Timeout expiration
    - Duplicate ACKs
- The actual transmitter window size is the minimum between the two above values
TCP congestion control

TCP flow control

- TCP receiver explicitly declares the available buffer space (time variable)
  - Window field in the TCP segment header
- TCP transmitter window (amount of data sent without receiving ACKs) never exceeds the declared receiver window size (in bytes)

TCP congestion control

- Originally (<1988) TCP was relying only on the window control operated by the receiver to enforce flow control
  - Relatively lightly loaded networks
  - TCP connection limited by the receiver speed
- Congestion effect is segment drops, which implies throughput reduction due to frequent retransmissions
- Goals of congestion control
  - Adjusting to the bottleneck bandwidth
  - Adjusting to bandwidth variations
  - Fairly sharing bandwidth between flows
  - Maximizing throughput
TCP congestion control

• Besides the limitation imposed by the receiver through the receiver advertised window (adv_wnd), the TCP transmitter controls the network congestion through the congestion window (cwnd) value
• TCP transmitter can send up to n segments without receiving an ACK, where
  \[ n = \min(\text{adv}_\text{wnd}, \text{cwnd}) \]
• Several versions of TCP congestion control defined to compute cwnd
  – Reno (NewReno)
  – SACK
  – BIC and CUBIC
  – Many others (Tahoe, Vegas, Westwood, ECN)

Congestion Window (cwnd)

• Limits amount of in transit data
• Measured in bytes

\[ \text{wnd} = \min(\text{cwnd}, \text{adv}_\text{wnd}) \]

\[ \text{effective}_\text{wnd} = \text{wnd} - (\text{last}_\text{byte}_\text{sent} - \text{last}_\text{byte}_\text{acked}); \]
TCP congestion control

- Obvious idea
  - Try to adapt rate to available resources
  - Increase when not congested
  - Decrease rate when congestion detected

- Issues
  - How much to decrease/increase and how?
  - How to detect congestion
    - Timeout expiration
    - Duplicate ACKs
  - Need to probe for available bandwidth
    - How to start?
    - Put the network in congestion status to detect available bandwidth
  - Must work for greedy source but also for … (e.g. telnet)
  - Timeout setting
    - Need to estimate RTT (random process)

Utilization and Fairness

- More than full utilization (congestion)
- Less than full utilization
- Equal throughput (fairness)
- Ideal point
  - Max efficiency
  - Perfect fairness
- Max throughput
**TCP congestion control**

**Multiplicative Increase**
**Additive Decrease**

- Not stable!
- Moves away from fairness

**Additive Increase**
**Additive Decrease**

- Stable
- But does not converge to fairness
TCP congestion control

**Multiplicative Increase**
**Multiplicative Decrease**

- Stable
- Does not converge to fairness

---

**Additive Increase**
**Multiplicative Decrease**

- Stable
- Converges to ideal working point
- AIMD algorithm
TCP congestion control algorithm

- Tahoe version
- Maintains an additional variable (besides cwnd and adv_wnd)
  - ssthresh: threshold
  - Heuristically set to represent an “optimal” window value
- Two phases of congestion control
  - Slow start (cwnd < ssthresh)
    - Probe for bottleneck bandwidth
  - Congestion avoidance (cwnd >= ssthresh)
    - AIMD
- Note: algorithm description assumes for simplicity that each TCP segment has a size equal to 1 MSS

Slow Start algorithm

- Main ideas
  - Run when cwnd<ssthresh
  - Starts at slow pace but increase fast
- At connection startup
  - cwnd = 1 segment (more precisely, cwnd=1MSS)
  - ssthresh = adv_wnd
- For each in sequence ACK received, cwnd = cwnd + 1MSS
- Exponential window growth
  - For each RTT, cwnd size doubles
  - Not slow!
- Continues until
  - ssthresh is reached
  - A segment is lost
TCP congestion control

**Slow Start algorithm**

![Diagram of Slow Start algorithm with Host A and Host B, showing RTT, 1 segment, 2 segments, and 4 segments over time.]

**Slow Start: example**

![Graph showing Slow Start example with data points at different time intervals.]
TCP congestion control

### Congestion Avoidance algorithm

- **Main ideas**
  - Run when cwnd>ssthresh
  - Slow down window growth but keep increasing to probe for additional available bandwidth
- **For each in sequence ACK received**
  - \[ cwnd = cwnd + \frac{1}{cwnd} \] or
  - \[ cwnd = cwnd + \frac{\text{MSS}}{cwnd} \] (in byte)
- **Linear window growth**
  - Every RTT, the window increases by 1 MSS in absence of losses
  - ADDITIVE increase
- **Continues until a segment is lost**

---

### Congestion Avoidance: example
TCP congestion control

If one segment is lost...

• …congestion indication
  – Transmitter bit rate overcame available bit rate

• Main ideas:
  – TCP transmitter re-send the missing segment if
    the proper ACK is not received before timeout
    expiration (all segments lost is a severe
    congestion scenario)
  – Reset the window value (cwnd=1)
  – Set the threshold to half the current window to
    ensure a fast cwnd increase
    • ssthresh = max(min(cwnd,adv_wnd)/2,2),

Summary

1) cwnd = 1 MSS
   ssthresh = adv_wnd
2) cwnd = cwnd + 1 for each ACK until
   cwnd > ssthresh (goto 3)
   if timeout expires:
     ssthresh = min(cwnd,adv_wnd)/2
     cwnd = 1
     goto 2)
3) cwnd = cwnd + 1/cwnd for each ACK
   if timeout expires:
     ssthresh = min(cwnd,adv_wnd)/2
     cwnd = 1
     goto 2)
Fast Retransmit and Fast Recovery

- Further modification to the congestion control algorithm proposed in 1990 (RFC 2001, Stevens)
- It allows the “immediate” retransmission of a single segment lost (Fast Retransmit)
  - Single segment loss is an indication of mild congestion
- …and avoids to re-start the algorithm in the Slow Start phase when a single segment was lost (Fast Recovery)
Fast Retransmit

- Observe duplicate ACKs
  - If few duplicate ACKs, it may be an out of order segments delivery
  - If more duplicate ACKs are lost, strong indication of segment loss
    - However, since duplicate ACKs are received at the transmitter, other segments were received, which implies mild congestion
- If three duplicate ACKs are received, retransmit the missing segment without waiting for timeout expiration (Fast Retransmit).

Fast Retransmit: example
Fast Recovery

• When congestion detected, go into congestion avoidance phase, and avoids slow start
• When the 3rd duplicate ACK is received:
  – ssthresh = min(cwnd,adv_wnd)/2
  – re-transmit the missing segment
  – cwnd=ssthresh+3
    • To keep constant the number of segments in the pipe
• For each successive duplicate ACK
  – cwnd=cwnd+1
  – enable segment transmission also during Fast Recovery
• When an ACK confirms the missing segment:
  – cwnd=ssthresh
  – cwnd=cwnd+1/cwnd for each correct and in sequence ACK
TCP congestion control

Summary

- slow start
- congestion avoidance
- ssthresh

Gain with respect to SS + CA

Summary

- slow start
- congestion avoidance
- ssthresh

Gain with respect to SS + CA
Fast Retransmit and Fast Recovery

- At steady state, cwnd oscillates around the optimal window size
- TCP always forces packet drops

TCP versions

- TCP Tahoe (Included in 4.3BSD Unix)
  - Originally proposed by Van Jacobson
    - Slow start
    - Congestion avoidance
    - Fast retransmit
- TCP Reno (Proposed in 1990)
  - All TCP Tahoe algorithms
  - Adds
    - Fast-recovery
    - Delayed ACKs
    - Header prediction to improve performance in HW
TCP Reno: Delayed ACK

• Motivations to delay ACK transmission
  – To reduce the number of ACKs sent (reduce control traffic)
  – The application may create data as a response to received segment
    • Exploit piggybacking to send ACKs
  – The receiver may empty the reception buffer, declaring larger available window rwnd

• Disadvantages
  – Modify connection RTT (Round Trip Time)
  – Window growth is slowed down

Delayed ACK: RFC

• The delayed ACK algorithm specified in [Bra89] SHOULD be used by a TCP receiver. When used, a TCP receiver MUST NOT excessively delay acknowledgments. Specifically, an ACK SHOULD be generated for at least every second full-sized segment, and MUST be generated within 500ms of the arrival of the first unacknowledged segment.

• Out-of-order data segments SHOULD be acknowledged immediately, to accelerate loss recovery.
**Delayed ACK: algorithm**

- ACKs are sent
  - either every 2 received segments
    - Window growth halved
  - or 200ms after segment reception

- Immediate ACK transmission only for out-of-sequence segments
  - Send ACK for the last segment in order and correctly received
    - Generates duplicate ACKs

---

**TCP ACK generation**

[RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order segment arrival, no gaps,</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next</td>
</tr>
<tr>
<td>everything else already acked</td>
<td>segment, send ACK</td>
</tr>
<tr>
<td></td>
<td>immediately send single cumulative ACK</td>
</tr>
<tr>
<td>in-order segment arrival, no gaps,</td>
<td>send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>one delayed ACK pending</td>
<td></td>
</tr>
<tr>
<td></td>
<td>immediate ACK if segment starts at lower end of gap</td>
</tr>
<tr>
<td>out-of-order segment arrival,</td>
<td></td>
</tr>
<tr>
<td>higher-than-expect seq. # gap detected</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>arrival of segment that partially or</td>
<td></td>
</tr>
<tr>
<td>completely fills gap</td>
<td></td>
</tr>
</tbody>
</table>
TCP NewReno

- RFC2582, proposed in 1999
- Solves the TCP-Reno problem
  - Multiple segment drops make useless the fast recovery-fast retransmit mechanism
- Considers partial ACKs reception during a Fast Recovery as a signal of loss of another segment
  - Retransmits immediately
- A new variable, named recovery, is needed
- When ACK received
  - The Fast Recovery phase is declared ended

TCP NewReno

- When the 3\textsuperscript{rd} consecutive duplicate ACK is received:
  - \textit{ssthresh} = \textit{min}(\textit{cwnd},\textit{advWnd})/2
  - Recovery=highest sequence number transmitted
  - Retransmit the missing segment
  - \textit{cwnd}=\textit{ssthresh}+3
- For each successive duplicate ACK
  - \textit{cwnd}=\textit{cwnd}+1
  - Send new segments if possible
TCP NewReno

- When an ACK which confirms the missing segment is received:
  - If ACK > recovery, then
    - cwnd = ssthresh
    - Fast Recovery procedure ends
  - Else (partial ACK)
    - Shrink transmission window by an amount equal to the confirmed segment size
    - cwnd = cwnd + 1
    - Send new segments if cwnd permits

TCP SACK

- Introduces selective acknowledge in ACK
- Must be negotiated by TCP transmitter and receiver
- Exploits Option field in TCP header to transport SACK information
- More than one segment per RTT can be retransmitted
TCP congestion control

TCP SACK

- A block represents a contiguous sequence of segments correctly received and buffered at the receiver.
- The receiver sends SACK info only if some out of sequence segments were received.
- May be used to indicate duplicated segments.

TCP variants today

- The most popular version try to address three key problems
  - TCP poor performance on high bandwidth-delay product network
    - How much time is needed to increase cwnd on a 1Gbps link from half utilization to full utilization?
      - Using 1500-byte PDU and 100 ms RTT
      - Full utilization cwnd = 1Gbps/1500byte \( \approx \) 8333
      - Half utilization cwnd = 8333/2 = 4166
      - cwnd is increased by 1 for each RTT
        - 4167 RTT is needed to fully utilize the link
        - 4167 RTT * 100ms(RTT time) = 6.95 minutes
  - TCP throughput depends on RTT
    - Keep a separate delay based window (MS Windows solution)
  - Vast majority of Internet traffic is short flows
    - Most TCP flows never leave slow start!
    - Increase initial cwnd to 10 (Google driven)
TCP today

• Compound TCP (Windows)
  – Based on Reno
  – Uses two congestion windows: delay based and loss based
  – Thus, it uses a compound congestion controller

• TCP CUBIC (Linux)
  – Enhancement of BIC (Binary Increase Congestion Control)
  – Window size controlled by cubic function
  – Parameterized by the time $T$ since the last dropped packet

High Bandwidth-Delay Product

• Key Problem: TCP performs poorly when
  – The capacity of the network (bandwidth) is large
  – The delay (RTT) of the network is large
  – Or, when bandwidth * delay is large
    • $b \times d =$ maximum amount of in-flight data in the network
    • a.k.a. the bandwidth-delay product

• Why does TCP perform poorly?
  – Slow start and additive increase are slow to converge
  – TCP is ACK clocked
    • i.e. TCP can only react as quickly as ACKs are received
    • Large RTT $\rightarrow$ ACKs are delayed $\rightarrow$ TCP is slow to react
TCP congestion control

Poor Performance of TCP Reno

Goals

- Fast window growth
  - Slow start and additive increase are too slow when bandwidth is large
  - Want to converge more quickly
- Maintain fairness with other TCP variants
  - Window growth cannot be too aggressive
- Improve RTT fairness
  - TCP Tahoe/Reno flows are not fair when RTTs vary widely
- Simple implementation
**Compound TCP Implementation**

- Default TCP implementation in Windows
- Key idea: split cwnd into two separate windows
  - Traditional, loss-based window
  - New, delay-based window
- \( \text{wnd} = \min(cwnd + dwnd, \text{adv}_\text{wnd}) \)
  - cwnd is controlled by AIMD
  - dwnd is the delay window
- Rules for adjusting dwnd:
  - If RTT is increasing, decrease dwnd (dwnd >= 0)
  - If RTT is decreasing, increase dwnd
  - Increase/decrease are proportional to the rate of change

**Compound TCP Example**

- Advantages: fast ramp up, more fair to flows with different RTTs
- Disadvantage: must estimate RTT
**TCP CUBIC Implementation**

- Default TCP implementation in Linux
- Make window size growth independent of RTT
  - Use elapsed real time since the last loss event
- Replace AIMD with cubic function

\[
cwnd = C \times \left(T \times \frac{3\sqrt{cwnd_{max}\beta}}{C}\right)^3 + cwnd_{max}
\]

- \(C\) \rightarrow a constant scaling factor
- \(\beta\) \rightarrow a constant fraction for multiplicative decrease
- \(T\) \rightarrow time since last packet drop
- \(cwnd_{max}\) \rightarrow cwnd when last packet dropped

**TCP CUBIC Example**

- Less wasted bandwidth due to fast ramp up
- Stable region and slow acceleration help maintain fairness
  - Fast ramp up is more aggressive than additive increase
**Timeout setting and RTT estimation**

- The timeout value is essential to obtain an efficient TCP behavior
- Timeout cannot be smaller than 200ms (delayed ACK and transmitter clock granularity)
- The timeout should be a function of connection RTT, which is time variable depending on network load
- A round trip time estimate is needed to set a proper timeout value

**Timeout setting**

- For each transmitted segment, compute the time difference M between segment transmission and ACK reception
  - Instantaneous RTT sample
- RTT estimate by weighting through an exponential filter with coefficient $\alpha$:
  - $\text{RTT} = \alpha \cdot \text{RTT} + (1-\alpha) \cdot M$ ($\alpha = 0.875$)
- Timeout (RTO) set to:
  - $\text{RTO} = \beta \cdot \text{RTT}$ ($\beta > 1$, typically 2)
Problems in RTT estimate

- Re-transmitted segment: RTT estimate?

RTT estimation may increase without bound if associating the ACK to the first segment transmission!!

Problems in RTT estimate

- Re-transmitted segment: RTT estimate?

RTT estimation too small if associating the ACK to segment re-transmission!
TCP congestion control

**Exponential backoff on the timeout value**

- RTT samples of re-transmitted segment may provide a wrong estimation.
- Karn algorithm:
  - RTT estimate is not modified unless an ACK for a non-retransmitted segment is received.
    - Not enough! Indeed, if then RTT increase, a new RTT estimate is never obtained since all segment are re-transmitted.
  - Increase timeout value according to an exponential backoff algorithm for each lost segment, since the RTT estimate is not reliable.
    - Sooner or later the timeout will assume a value larger than the current RTT; and a new RTT estimate is obtained.

**Problems in RTT estimate**

- Delay variations may create fluctuations on RTT estimate.
  - Use more complex formulas to estimate RTT.
  - Take into account the average estimation error.
    - timeout = average + 4 * standard_deviation.
Jacobson/Karels Algorithm

- New proposal for RTT estimation
  - Diff = SampleRTT - EstimatedRTT
  - EstimatedRTT = EstimatedRTT + (δ Diff)
  - Deviation = Deviation + δ(|Diff| - Deviation)
  - Where δ ranges from 0 to 1

- Standard deviation is considered when computing RTO
  - RTO = μ EstimatedRTT + φ Deviation
  where μ = 1 and φ = 4

Notes on RTT estimate

- Estimate is always constrained by timer granularity (10ms on recent systems, 200ms on older systems)
  - The RTT may be comparable with timer granularity (RTT=100-200ms for long distance connections)

- Accuracy in RTT estimation is fundamental to obtain an efficient congestion control (avoids useless re-transmissions or excessively long waits)
Timeout setting: problems

- Initial value?
- Since an RTT estimation is missing, the initial timeout value is chosen according to a conservative approach
  - Initial timeout set to 1s (RFC6298)
- TCP connections are very sensitive to the first segment loss since the timeout value is large

Silly Window Syndrome

- Excessive overhead problem due to
  - Slow receivers or
  - Transmitter sending only small segments
- If the receiver buffer fills up, the receiver declares increasingly smaller advWnd
- The transmitter sends tinygrams if the applications generates few data (e.g., telnet application)
TCP connections for telnet traffic

- Telnet application
  - When pressing a key on the terminal keyboard
  - A TCP segment TCP of 1B is sent in a dedicated IP datagram: (20B+20B)header +1B data

- Even worse, if local echo disabled, 4 1B segments are sent: key + ACK + echo + ACK

- Exploiting piggybacking of the first ACK on the echo segment, one segment is saved
  - Delayed ACK helps

Silly Window Syndrome avoidance

- At the receiver side:
  - Declare the new available receiver window only if equal to
    - 1 MSS or
    - Half of the receiver buffer
  - Delayed acknowledgment

- At transmitter side:
  - Nagle algorithm
Nagle algorithm (RFC 896)

- When opening the connection, all data in the transmission buffer are sent
- Then, wait for
  - at least 1 MSS data in the transmission buffer or
  - ACK reception
- A host never has more than one tinygram without an ACK

Nagle algorithm

- When running a telnet application, successive characters following the first one are collected in a single segment, sent after receiving the first ACK
- Ftp, smtp, http connections are not penalized
- The number of tinygrams is drastically reduced
- Is congestion friendly
  - Being ACK clocked, when the network is lightly loaded ACKs are frequently and fastly received and segment transmission is speeded-up
  - When network becomes congested, ACKs are delayed and less segments are sent