Master Course
Computer Networks
IN2097

Prof. Dr.-Ing. Georg Carle

Chair for Network Architectures and Services
Department of Computer Science
Technische Universität München
http://www.net.in.tum.de

Outline

- Transport Layer
  - TCP congestion control
  - TCP variants
  - TCP throughput formula

Congestion Control

Approaches Towards Congestion Control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

Congestion Control (Van Jacobson)

- Problem: the end host does not know a lot about the network.
  - It only knows if a packet has been delivered successfully or not

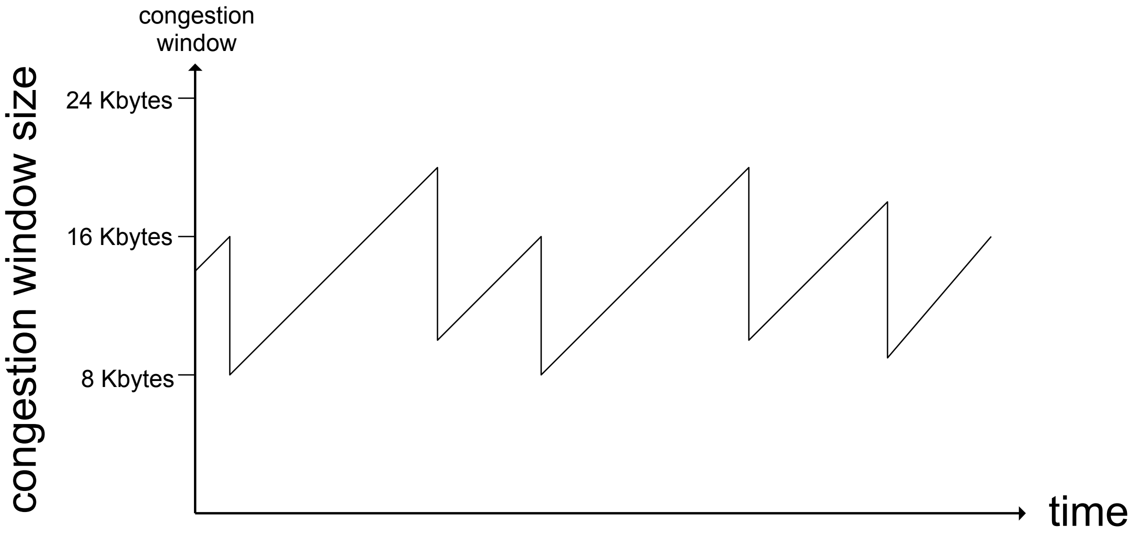
- Self clocking:
  - for every segment that leaves the network we can send a new one

- Assumption:
  - packet loss only because of congestion
  - Not true for wireless networks

TCP congestion control: additive increase, multiplicative decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



TCP Congestion Control: Details

- sender limits transmission:

  LastByteSent-LastByteAcked

  ≤ CongWin

- Roughly,

$$\boxed{\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}}$$

- **CongWin** is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**CongWin**) after loss event

three mechanisms:

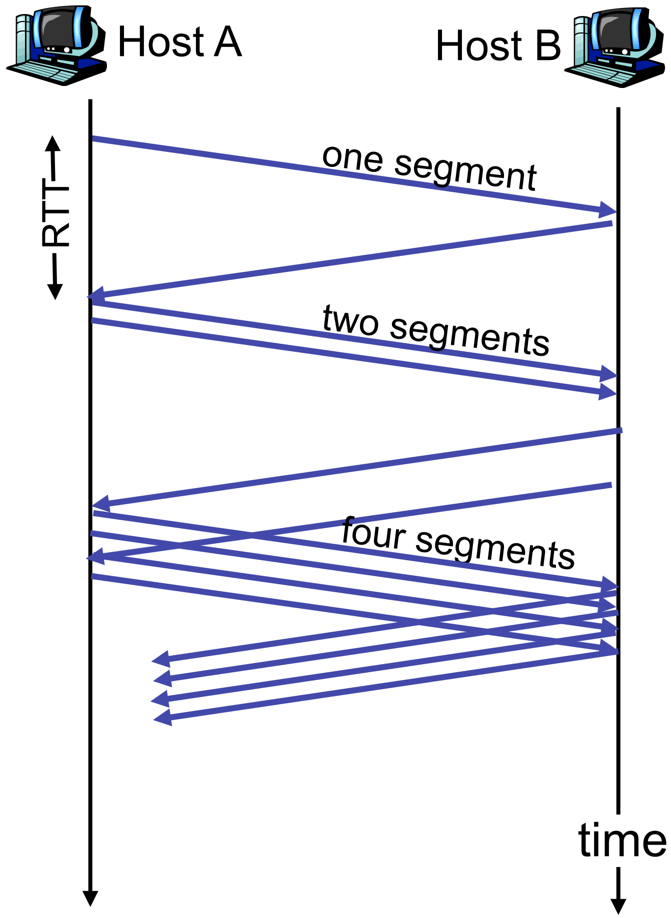
- AIMD
- slow start
- conservative after timeout events

TCP Slow Start

- When connection begins, **`CongWin`** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double **`CongWin`** every RTT
  - done by incrementing **`CongWin`** for every ACK received
- <u>Summary:</u> initial rate is slow but ramps up exponentially fast



Refinement: Inferring Loss

- After 3 dup ACKs:
  - `CongWin` is cut in half
  - window then grows linearly
- But after timeout event:
  - `CongWin` instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Philosophy:

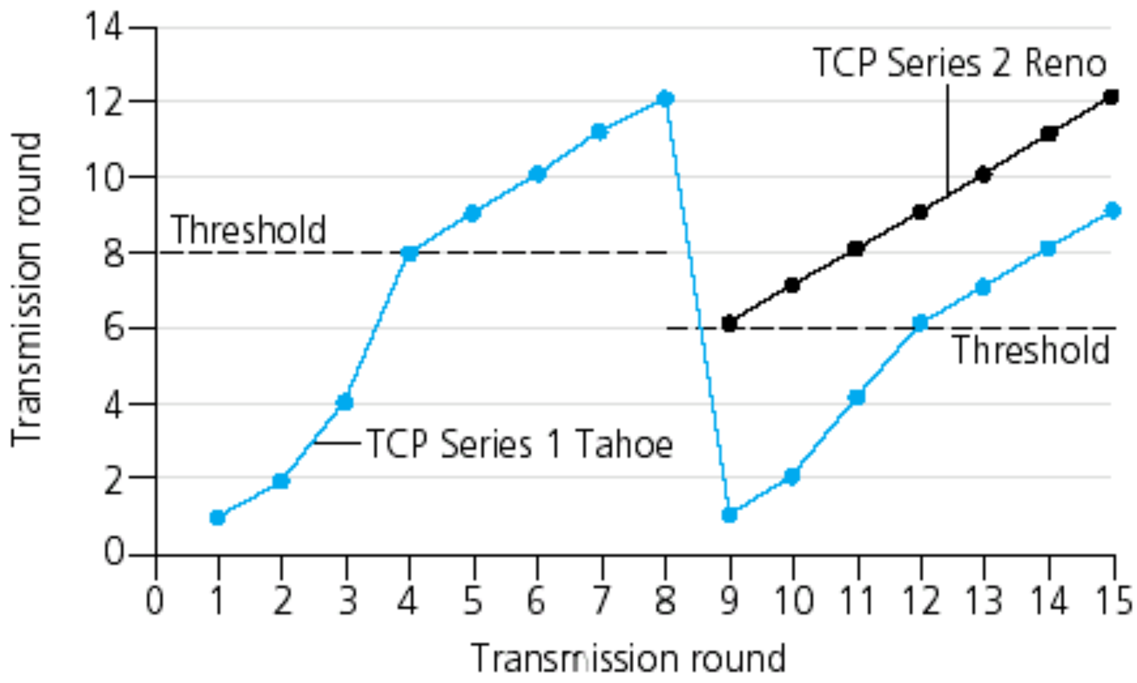
- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a “more alarming” congestion scenario

Refinement

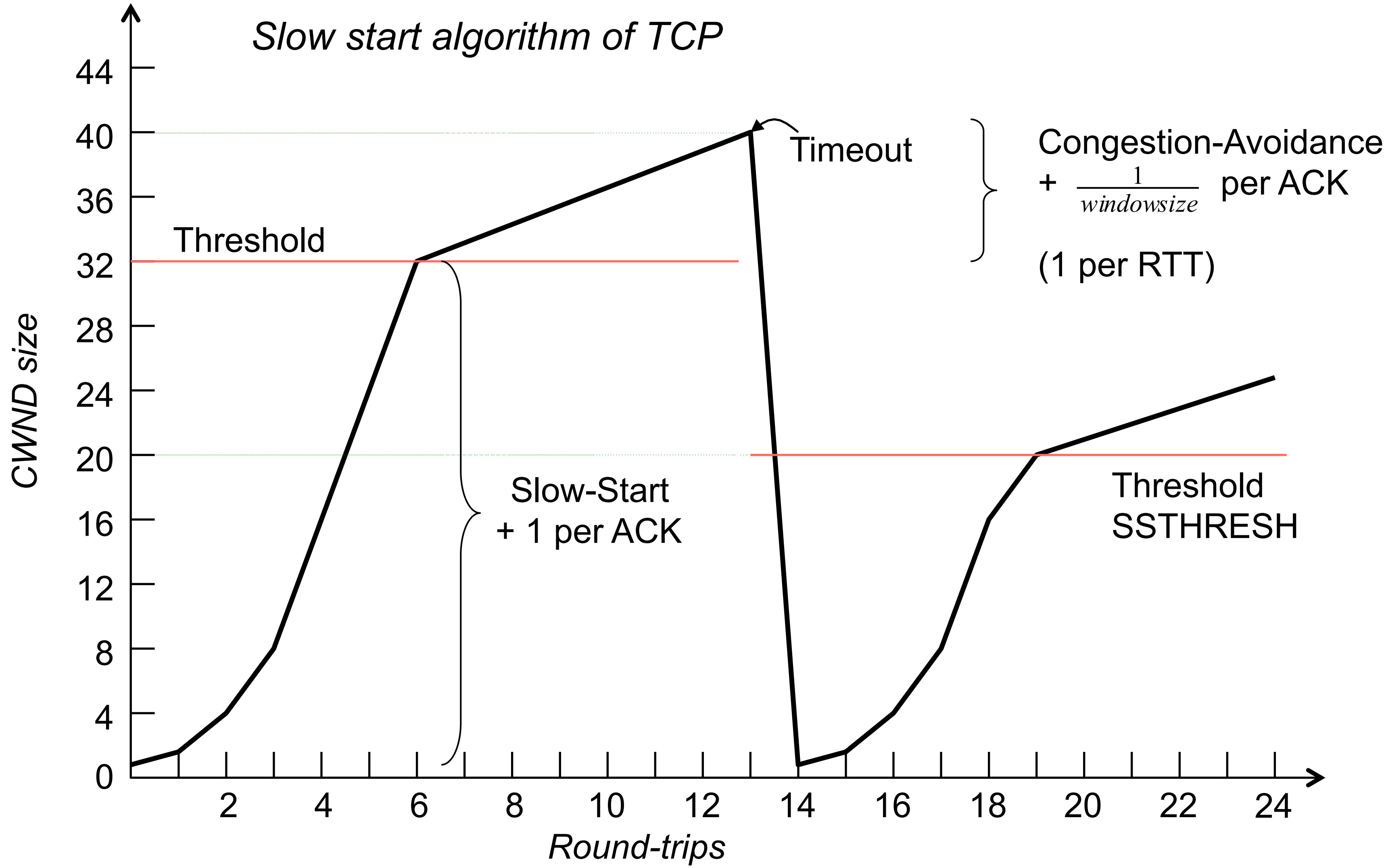
- Q: When should the exponential increase switch to linear?
- A: When CongWin gets to 1/2 of its value before timeout.

Implementation:

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

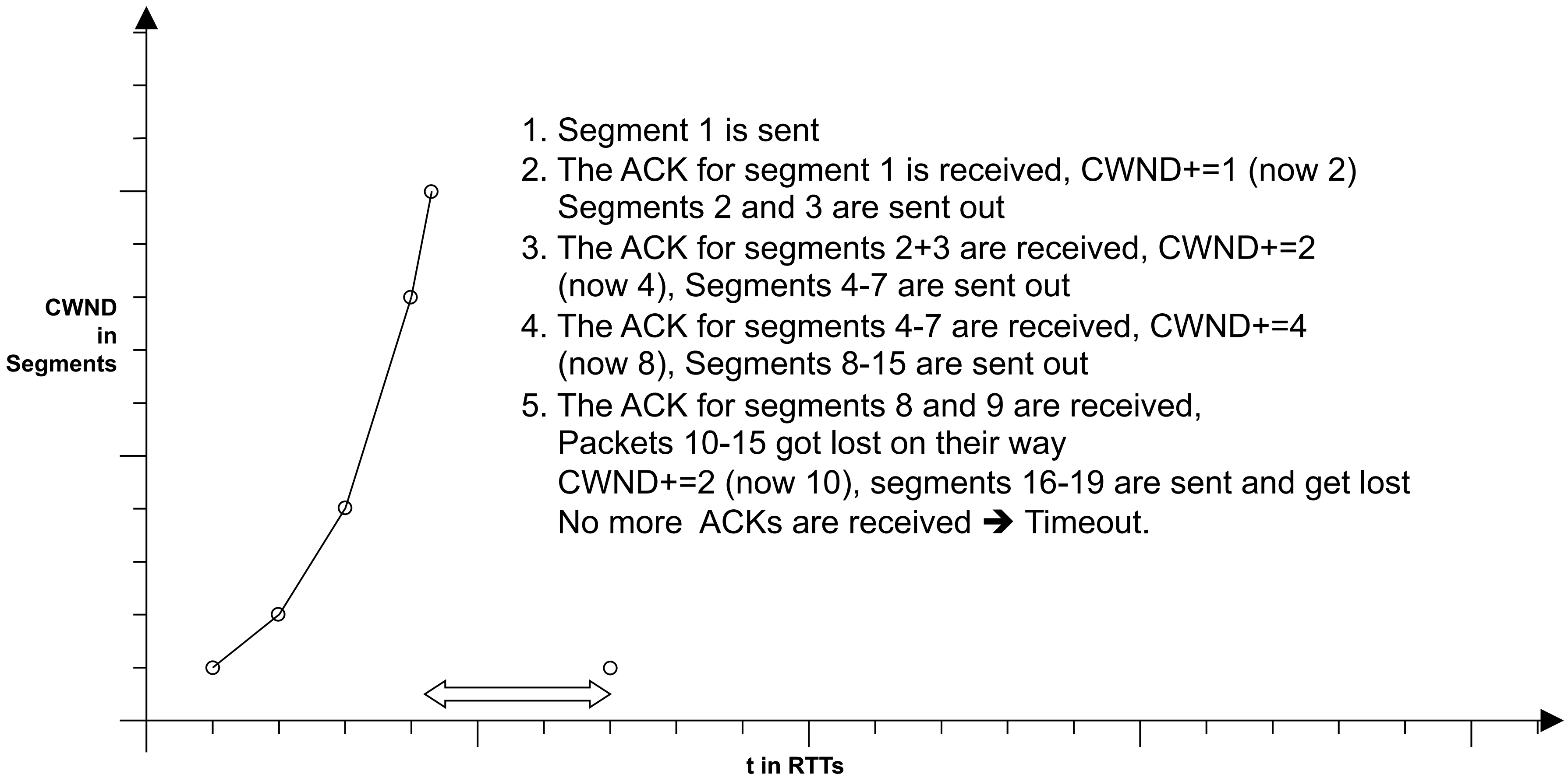


TCP Congestion Control Without Fast Retransmit



Here the window size is measured in number of packets
Real TCP uses bytes.

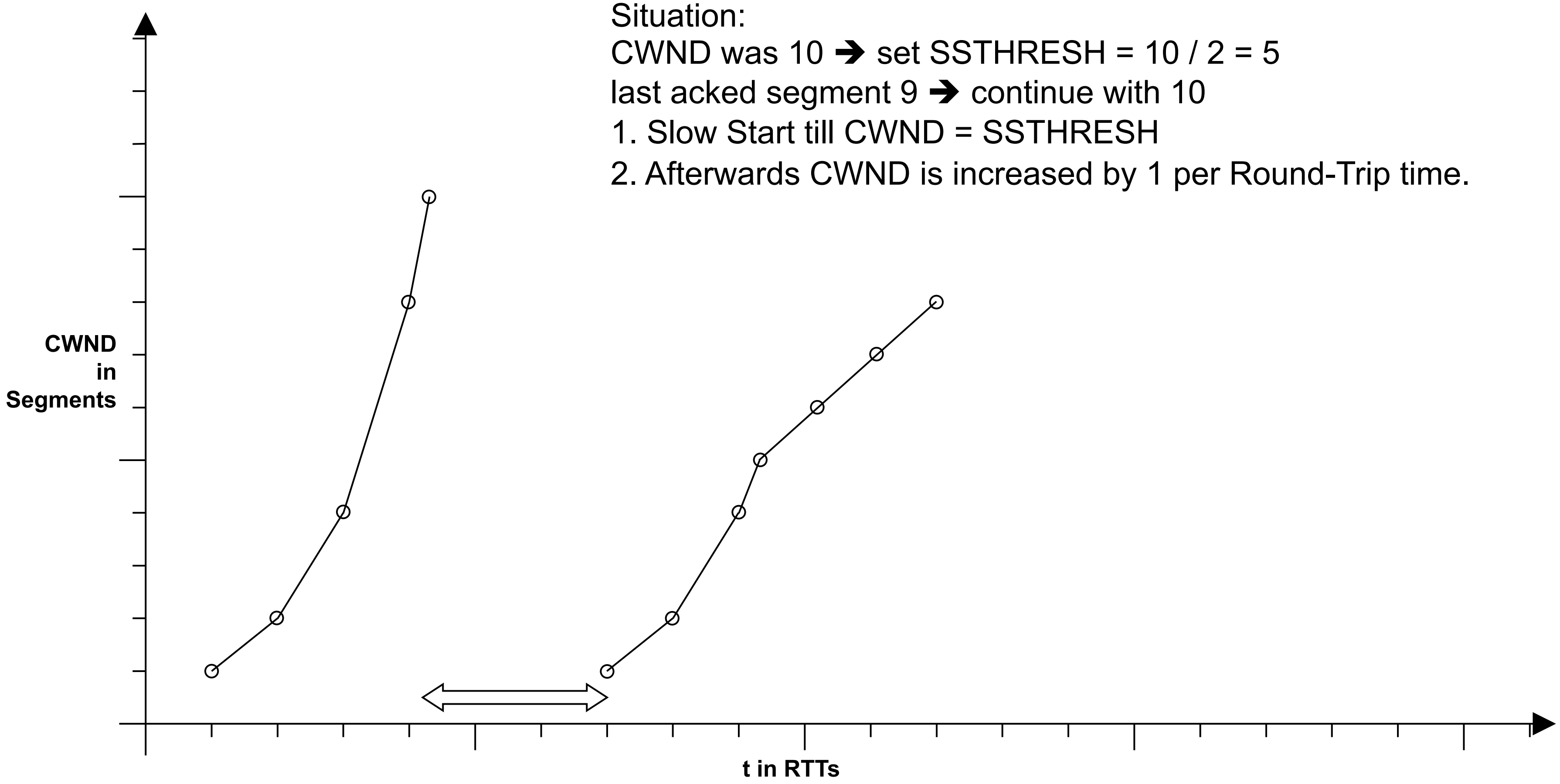
Slow Start



1. Segment 1 is sent
2. The ACK for segment 1 is received, CWND+=1 (now 2)
   Segments 2 and 3 are sent out
3. The ACK for segments 2+3 are received, CWND+=2 (now 4), Segments 4-7 are sent out
4. The ACK for segments 4-7 are received, CWND+=4 (now 8), Segments 8-15 are sent out
5. The ACK for segments 8 and 9 are received,
   Packets 10-15 got lost on their way
   CWND+=2 (now 10), segments 16-19 are sent and get lost
   No more ACKs are received ➔ Timeout.

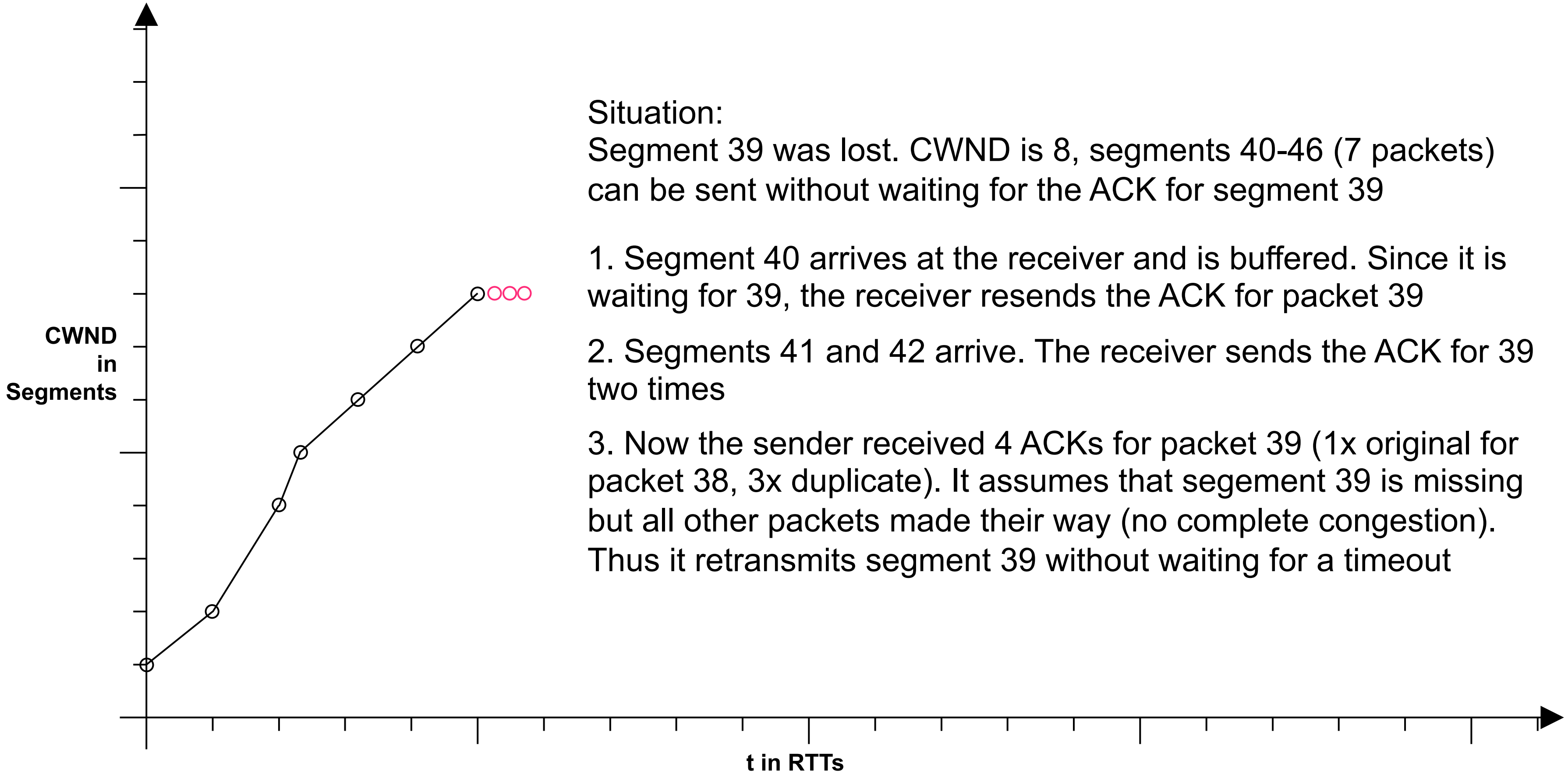
- Basic idea: packet loss indicates congestion
- Algorithm slowly approaches the limit

Slow Start and Congestion Avoidance



Situation:
CWND was 10 ➔ set SSTHRESH = 10 / 2 = 5
last acked segment 9 ➔ continue with 10
1. Slow Start till CWND = SSTHRESH
2. Afterwards CWND is increased by 1 per Round-Trip time.

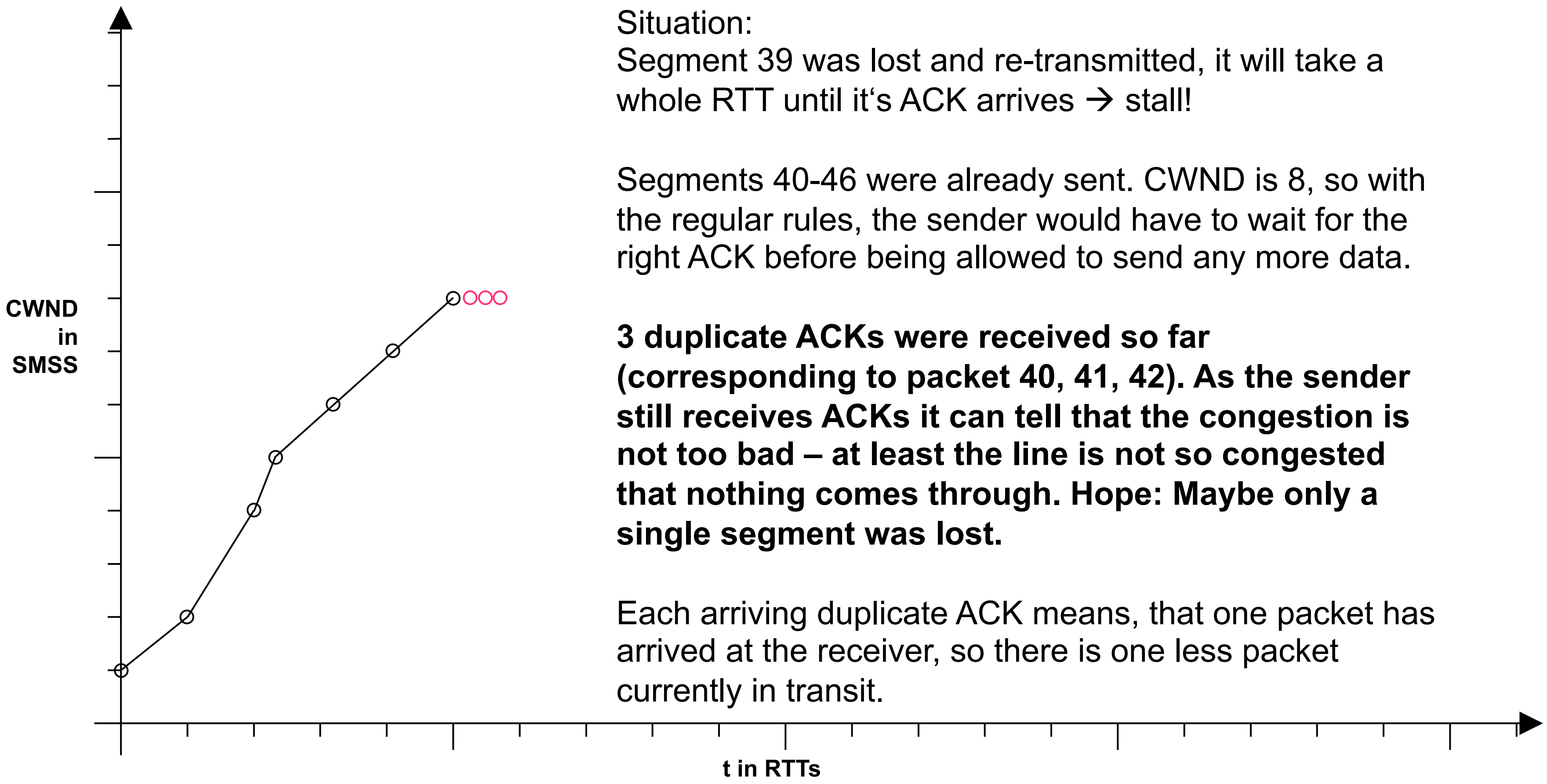
Fast Retransmit



Situation:
Segment 39 was lost. CWND is 8, segments 40-46 (7 packets) can be sent without waiting for the ACK for segment 39

1. Segment 40 arrives at the receiver and is buffered. Since it is waiting for 39, the receiver resends the ACK for packet 39
2. Segments 41 and 42 arrive. The receiver sends the ACK for 39 two times
3. Now the sender received 4 ACKs for packet 39 (1x original for packet 38, 3x duplicate). It assumes that segement 39 is missing but all other packets made their way (no complete congestion). Thus it retransmits segment 39 without waiting for a timeout

Fast Recovery I



Situation:
Segment 39 was lost and re-transmitted, it will take a whole RTT until it‘s ACK arrives → stall!

Segments 40-46 were already sent. CWND is 8, so with the regular rules, the sender would have to wait for the right ACK before being allowed to send any more data.

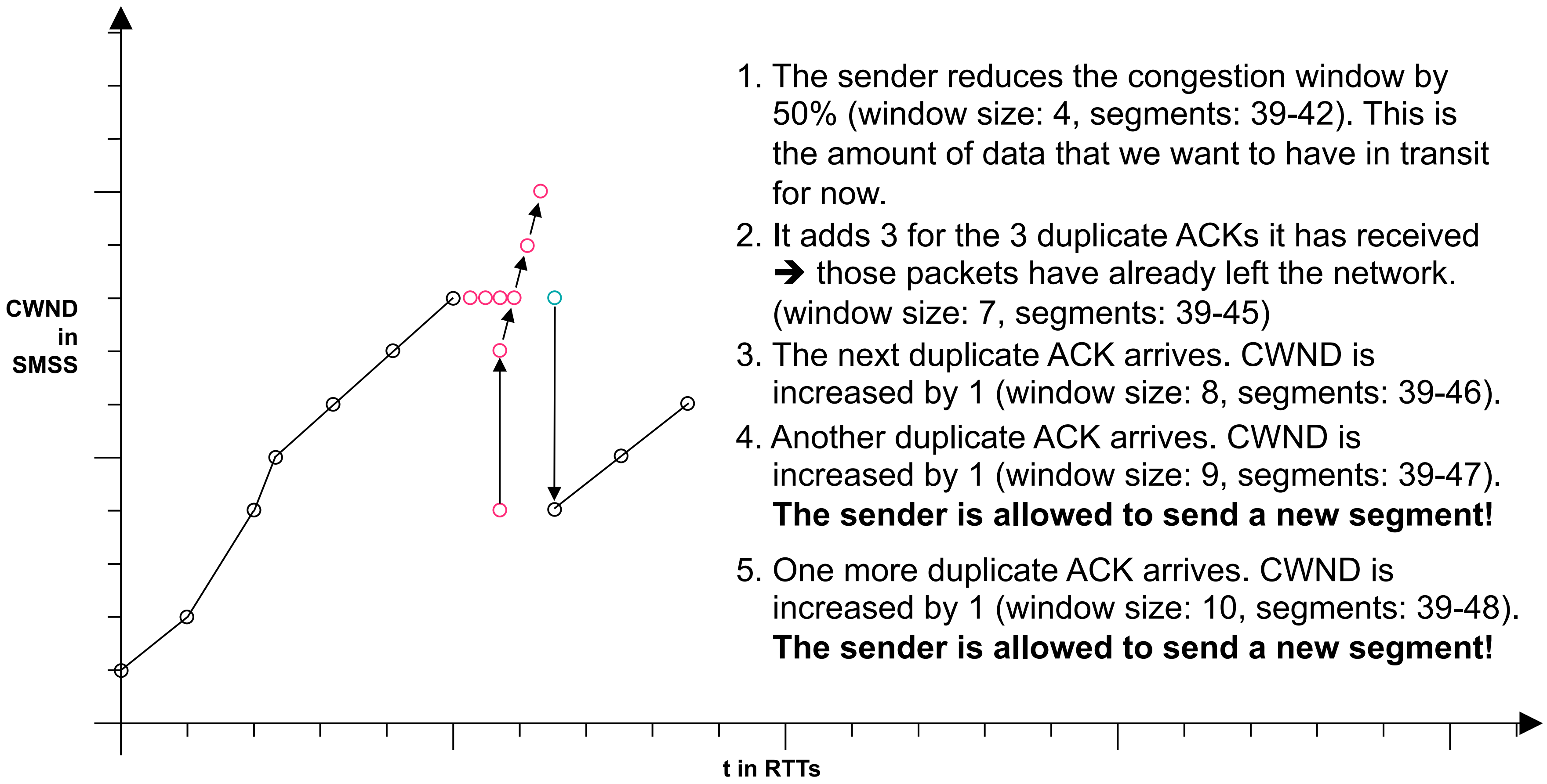
3 duplicate ACKs were received so far (corresponding to packet 40, 41, 42). As the sender still receives ACKs it can tell that the congestion is not too bad – at least the line is not so congested that nothing comes through. Hope: Maybe only a single segment was lost.

Each arriving duplicate ACK means, that one packet has arrived at the receiver, so there is one less packet currently in transit.

Idea of fast Recovery:

- Congestion is there but not too bad → Reduce the Window by 50%
- Then keep the number of segments in transit equal to the new window size even though there are no new ACKs (Compare Jacobson’s Self-Clocking).

Fast Recovery II



1. The sender reduces the congestion window by 50% (window size: 4, segments: 39-42). This is the amount of data that we want to have in transit for now.
2. It adds 3 for the 3 duplicate ACKs it has received ➔ those packets have already left the network. (window size: 7, segments: 39-45)
3. The next duplicate ACK arrives. CWND is increased by 1 (window size: 8, segments: 39-46).
4. Another duplicate ACK arrives. CWND is increased by 1 (window size: 9, segments: 39-47). **The sender is allowed to send a new segment!**
5. One more duplicate ACK arrives. CWND is increased by 1 (window size: 10, segments: 39-48). **The sender is allowed to send a new segment!**

7. The ACK for segment 39 arrives (actually it is a cummulative ACK for segment 46).
   The sender remembers the value of CWND before starting fast retransmit. CWND is set to half of the old value.
8. **The sender continues with Congestion Avoidance.**
   (window size: 4, segments 47-50)

Summary: TCP Congestion Control

- When `CongWin` is below `Threshold`, sender in slow-start phase, window grows exponentially.
- When `CongWin` is above `Threshold`, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, `Threshold` set to `CongWin/2` and `CongWin` set to `Threshold`.
- When timeout occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS.

TCP Sender Congestion Control

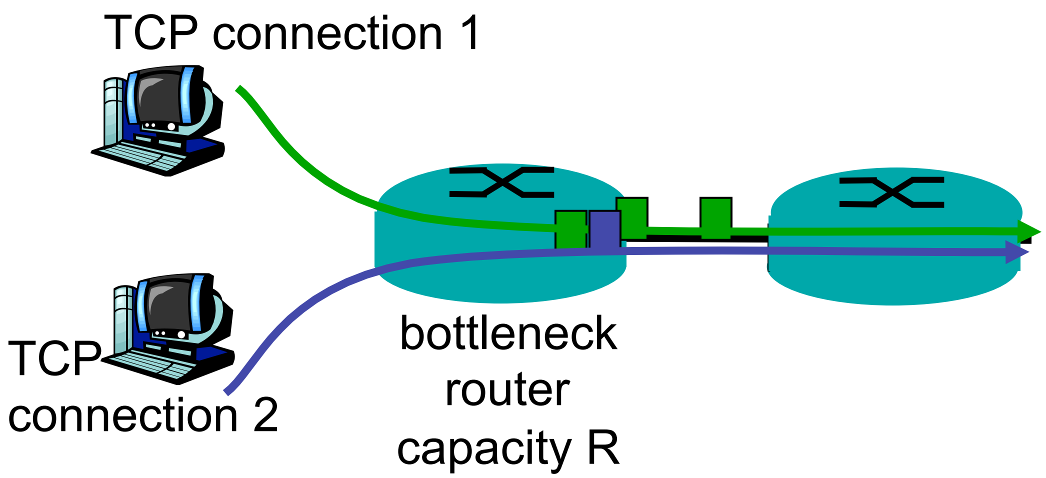
State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin := CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin := CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold := CongWin/2, CongWin := Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold := CongWin/2, CongWin := 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP summary

- Connection-oriented: SYN, SYNACK; FIN
- Retransmit lost packets; in-order data: sequence no., ACK no.
- ACKs: either piggybacked, or no-data pure ACK packets if no data travelling in other direction
- Don't overload receiver: rwin
  - rwin advertised by receiver
- Don't overload network: cwin
  - cwin affected by receiving ACKs
- Sender buffer = min { rwin, cwin }
- Congestion control:
  - Slow start: exponential growth of cwin
  - Congestion avoidance: linear groth of cwin
  - Timeout; duplicate ACK: shrink cwin
- Continuously adjust RTT estimation

TCP Fairness

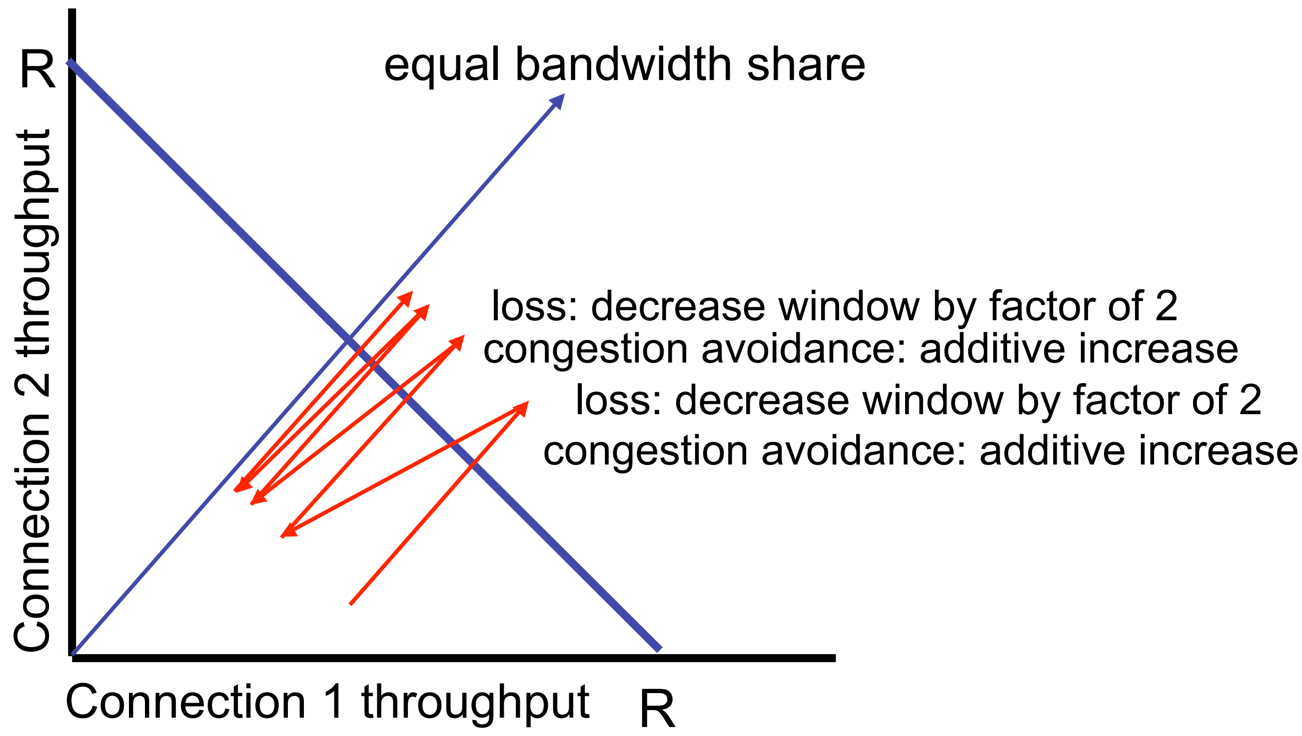
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets ~ R/2 !

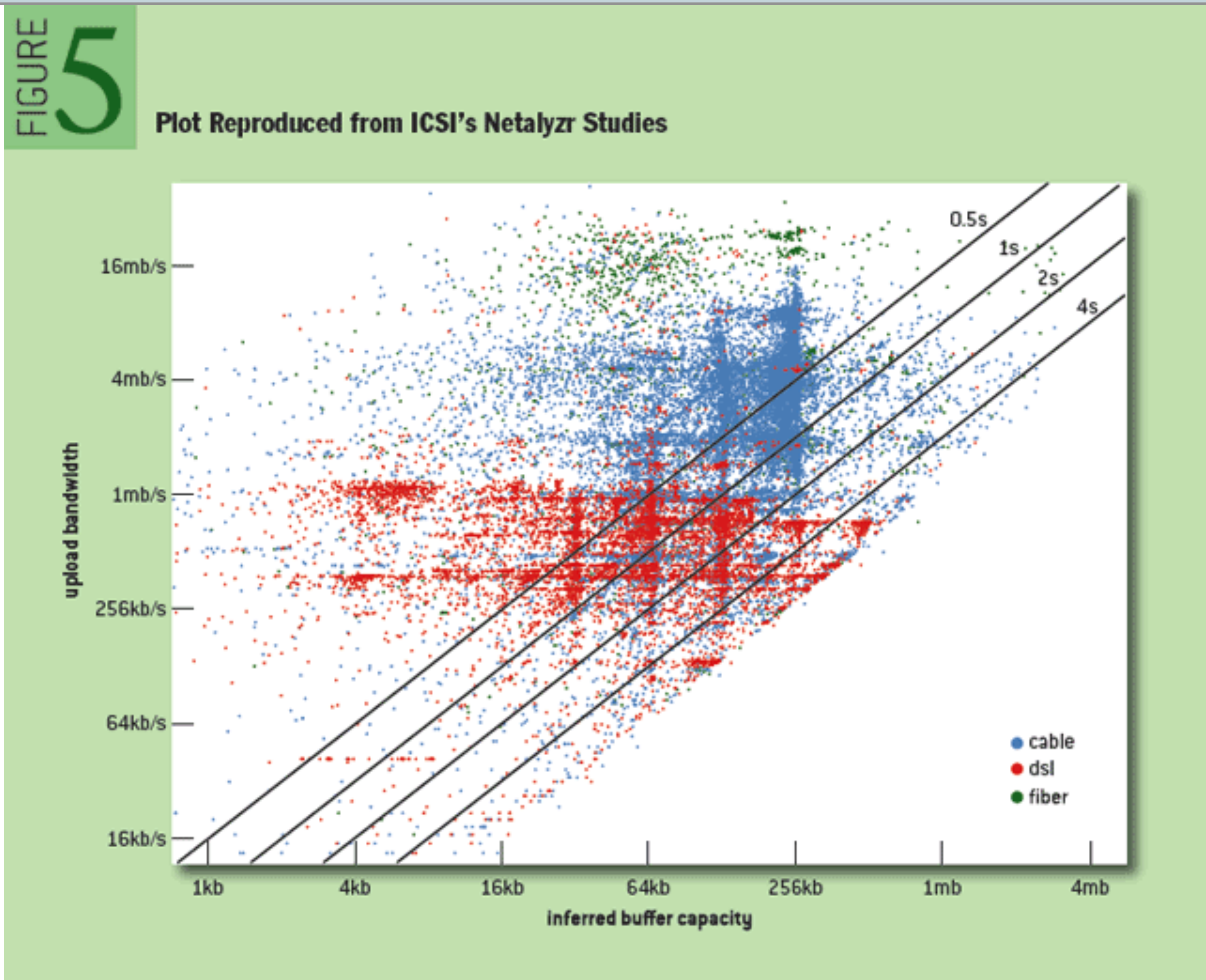
Advanced Topics

- Buffer Bloat
- TCP variants
  - TCP with ECN
  - TCP for high bandwidth long distance connections
- TCP Throughput Formula

TCP and Buffer Bloat

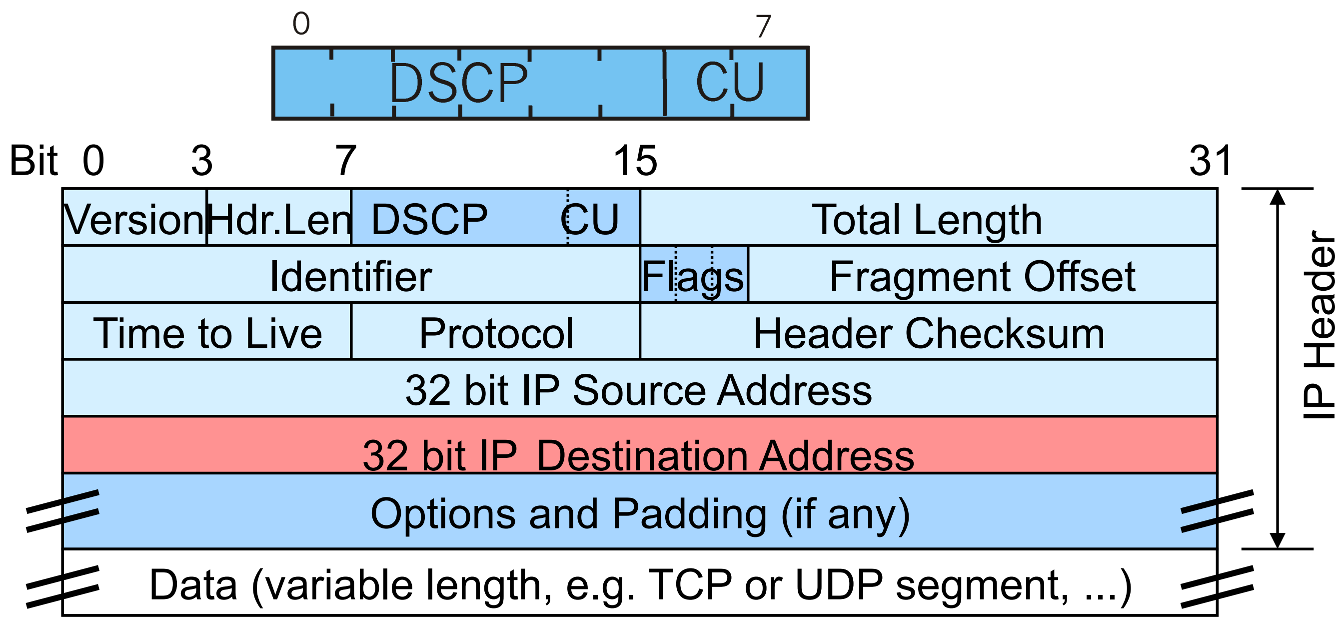
- Capacities of router queues
  - "Large queue = good: Less packet losses at bottlenecks"
  - Do you agree? What would happen to TCP?
- Effects of large buffers at bottleneck on TCP connections
  - Once queues are full: Queueing delays increase dramatically
  - TCP congestion control gets no early warning
    - No duplicate ACKS ⇨ no Fast Retransmit
    - Instead: Sudden timeouts
  - Congestion windows way too large
  - Many parallel TCP connections over same link get warning way too late
    - Synchronisation: Oscillation between "All send way too much" and "all get frightened by timeouts and send way too little"
    - Huge variations in queueing delays ⇨ DevRTT becomes very large ⇨ Timeout value becomes very large

Buffer bloat is a real-world problem



TCP with ECN

- 2 bits can be used for congestion notification:
  Explicit Congestion Notification (ECN), RFC 3168
  - 00 – Non ECN-Capable Transport, Non-ECT
  - 10 – ECN Capable Transport, ECT(0)
  - 01 – ECN Capable Transport, ECT(1)
  - 11 – Congestion Encountered, CE



ECN Operation

- Network layer with ECN
  - Endpoints supporting ECN mark packets with ECT(0) or ECT(1)
  - Routers that experience congestion rewrite ECN field to CE
- TCP with ECN
  - TCP endpoints negotiate ECN at connection setup (ECN option)
  - TCP reacts to CE as if packet is lost
  - Three flags in the TCP header
    - ECN-Echo (ECE): echoing back „Congestion Experienced“
    - Congestion Window Reduced (CWR)
- OS support
  - Linux: ECN by default is enabled when requested by incoming connections, it is not requested on outgoing connections
  - BSD: can be activated through the sysctl interface
  - Windows support: since Windows Server 2008, Windows Vista (disabled by default)

Data Center TCP (DCTCP)

- Enhancement of TCP congestion control algorithm for data center networks
- Switch marks fraction of packets corresponding to extent of congestion
- DCTCP sources estimate the and react to the extent of congestion, not just the presence of congestion as in TCP
- Publication
  - Data Center TCP (DCTCP): Mohammad Alizadeh, Balaji Prabhakar (Stanford University), Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Sudipta Sengupta, and Murari Sridharan (Microsoft Research), ACM SIGCOMM 2010
  - http://simula.stanford.edu/~alizade/Site/DCTCP.html

TCP for High Bandwidth Long Distance Connections

- Several transport protocol variants for high bandwidth long distance connections (LFNs - Long Fat Networks) exist
- Frequent property
  - Effectively use available bandwidth
  - Unfriendly – "doesn't play nicely with others"
  - Unfair to different RTT flows
  - achieves better performance than standard TCP
  - is not fair to standard TCP
- General approaches for congestion control
  - loss-based: NewReno, CUBIC
  - delay-based: Vegas, CAIA Delay Gradiant (CDG)

TCP Reno

- TCP Fast Recovery algorithm described in RFC 2581
- Implementation introduced 1990 in BSD Reno release
- Behaviour
  - sender only retransmits a packet
    - after a retransmit timeout has occurred
    - or after three duplicate acknowledgements have arrived

    triggering the Fast Retransmit algorithm.
  - a single retransmit timeout might result in the retransmission of several data packets
  - each invocation of the Fast Retransmit algorithm leads to retransmission of only a single data packet
  - problems may arrive when multiple packets are dropped from a single window

TCP NewReno

- c.f. RFC 3782 - April 2004, Proposed Standard
- „careful“ variant of Experimental RFC 2582 NewReno as default
- Properties
  - addresses problems that may arrive when multiple packets are dropped from a single window
  - with multiple packet drops, acknowledgement for retransmitted packet acks some but not all packets transmitted before the Fast Retransmit
    ⇨ „partial acknowledgment“

TCP Vegas

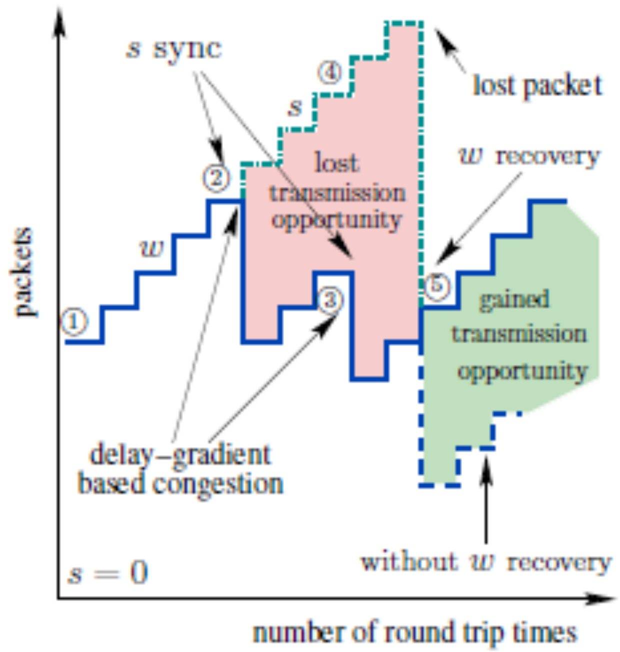
- TCP Vegas
  - by Lawrence Brakmo, Sean W. O'Malley, Larry L. Peterson at University of Arizona
  - published at SIGCOMM 1994
- Properties
  - delay-based congestion control
  - uses i^{th} RTT > min RTT + delay threshold, delay measured every RTT
  - Additive Increase Additive Decrease (AIAD) to adjust cwnd
- Properties
  - implementations available for Linux and BSD

TCP CUBIC

- CUBIC
  - Loss-based congestion control
    optimised for high bandwidth, high latency
- Properties
  - modified window-growth-control algorithm
  - window grows slowly around W_{max}
  - fast "probing" growth away from W_{max}
  - Standard TCP outperforms CUBIC's window growth function in short RTT networks.
  - CUBIC emulates standard (time-independent) TCP window adjustment algorithm, select the greater of the two windows (emulated versus cubic)
- Implemenation:
  - in Linux since kernel 2.6.19, in FreeBSD 8-STABLE

Delay Gradient TCP

- D. Hayes, G. Armitage, "Revisiting TCP Congestion Control using Delay Gradients," IFIP/TC6 NETWORKING 2011, Valencia, Spain, 9-13 May 2011 http://caia.swin.edu.au/cv/dahayes/content/networking2011-cdg-preprint.pdf
- CDG ("CAIA Delay-Gradient") modified TCP sender behaviour:
  - uses delay gradient as a congestion indicator
  - has average probability of back off independent of RTT
  - works with loss-based congestion control flows, eg NewReno
  - tolerates non-congestion packet loss, and backoff for congestion related packet loss



Multipath TCP

- **IETF Working Group Multipath TCP (mptcp)**
- Key goals
  - deployable and usable without significant changes to existing Internet infrastructure
  - usable by unmodified applications
  - stable and congestion-safe, including NAT interactions
- Objectives
  a. An architectural framework for congestion-dependent multipath transport protocols
  b. A security threat analysis for multipath TCP
  c. A coupled multipath-aware congestion control algorithm
  d. Multi-addressed multipath extensions to current TCP
  e. Application interface considerations
- TCP Extensions for Multipath Operation with Multiple Addresses, RFC6824
- MPTCP Application Interface, RFC6897

Advanced Topics

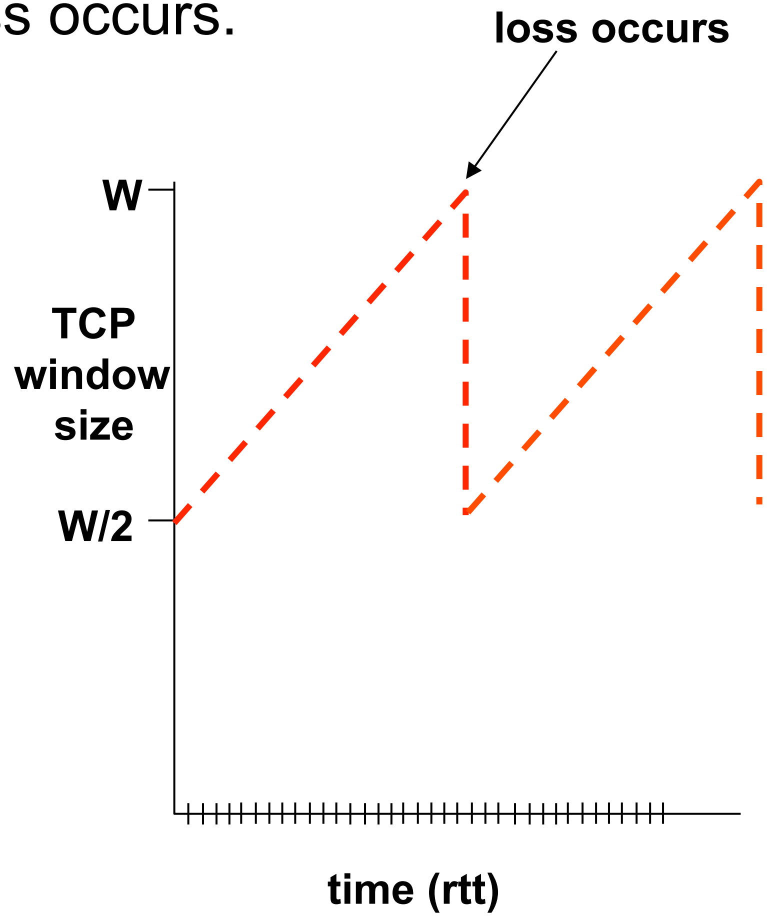
- Buffer Bloat
- TCP variants
  - TCP with ECN
  - TCP for high bandwidth long distance connections

- TCP throughput formula
  - possible usages include:
    - TCP-friendly application with UDP
    - Detection of TCP-unfriendly Flows

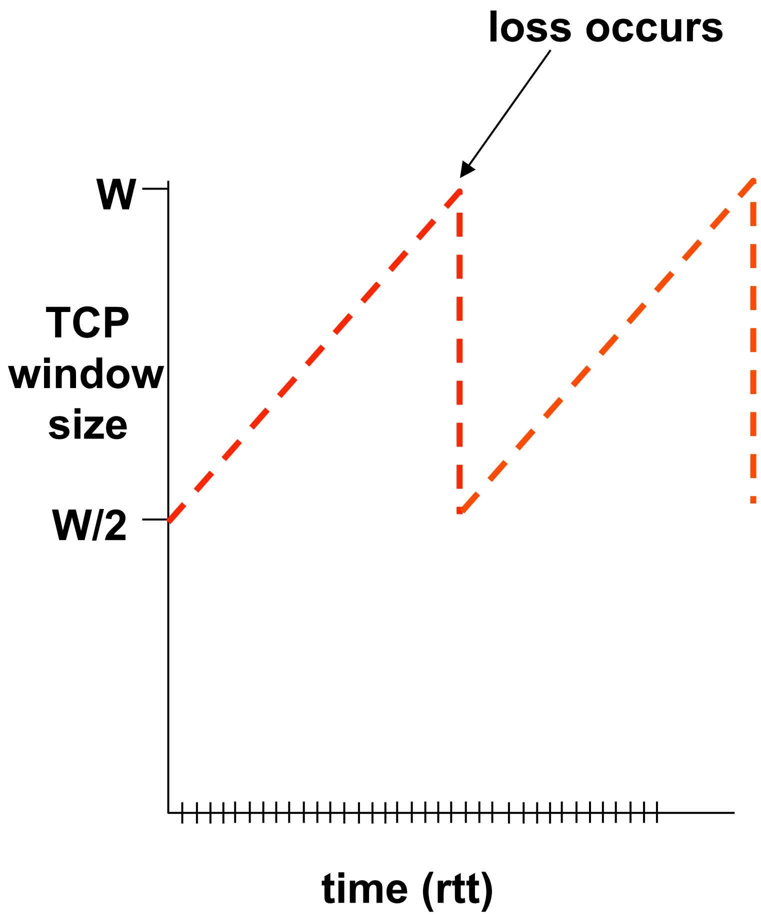
TCP throughput

- What's the average throughout of TCP as a function of window size and rtt?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/rtt
- Just after loss, window drops to W/2, throughput to W/2rtt.

⇨ Average throughout: 0.75 W/rtt

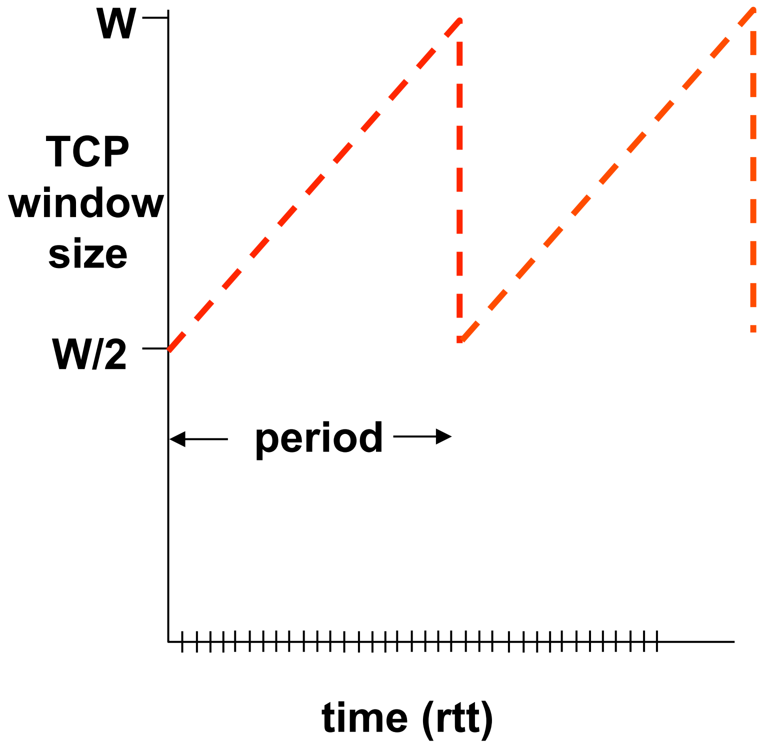


TCP throughput/loss relationship



- Idealized model:
  - W is maximum supportable window size (then loss occurs)
  - TCP window starts at W/2 grows to W, then halves, then grows to W, then halves…
  - one window worth of packets each RTT
  - to find: throughput as function of loss, RTT

TCP throughput/loss relationship



 # packets sent per "period" =

$$\frac{W}{2} + \left(\frac{W}{2} + 1\right) + \ldots + W = \sum_{n=0}^{W/2} \left(\frac{W}{2} + n\right)$$

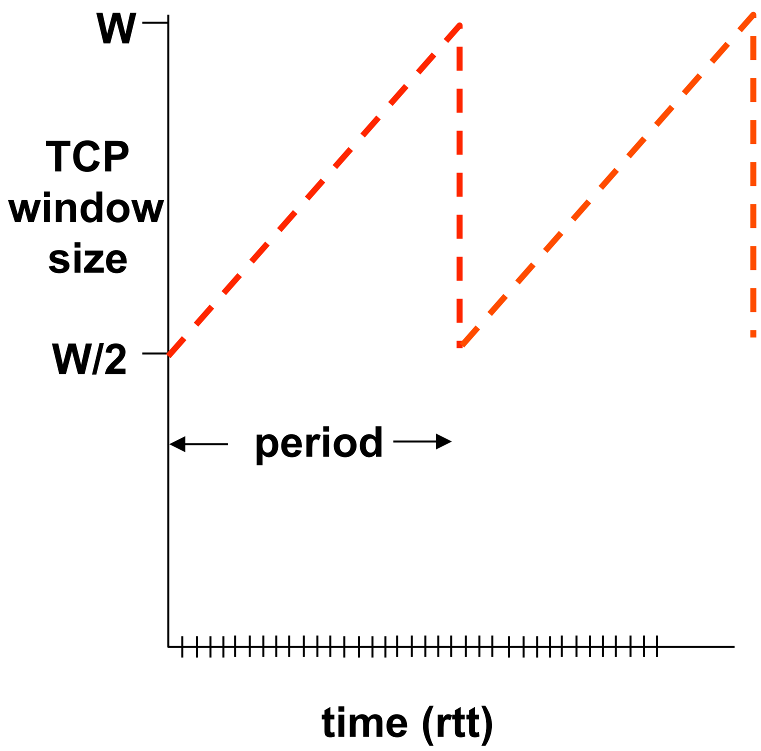
$$= \left(\frac{W}{2} + 1\right)\frac{W}{2} + \sum_{n=0}^{W/2} n$$

$$= \left(\frac{W}{2} + 1\right)\frac{W}{2} + \frac{W/2(W/2+1)}{2}$$

$$= \frac{3}{8}W^2 + \frac{3}{4}W$$

$$\approx \frac{3}{8}W^2$$

TCP throughput/loss relationship



packets sent per "period" $\approx \frac{3}{8}W^2$

1 packet lost per "period" implies:

$p_{loss} \approx \frac{8}{3W^2}$ **or:** $W = \sqrt{\frac{8}{3p_{loss}}}$

$$B = \text{avg.\_thruput} = \frac{3}{4}W \frac{\text{packets}}{\text{rtt}}$$

$$\boxed{B = \text{avg.\_thruput} = \frac{1.22}{\sqrt{p_{loss}}} \frac{\text{packets}}{\text{rtt}}}$$

B throughput formula can be extended to model timeouts and slow start