TCP Reno

- Basic TCP congestion control
- Sliding window algorithms
- Slow start
- Congestion avoidance

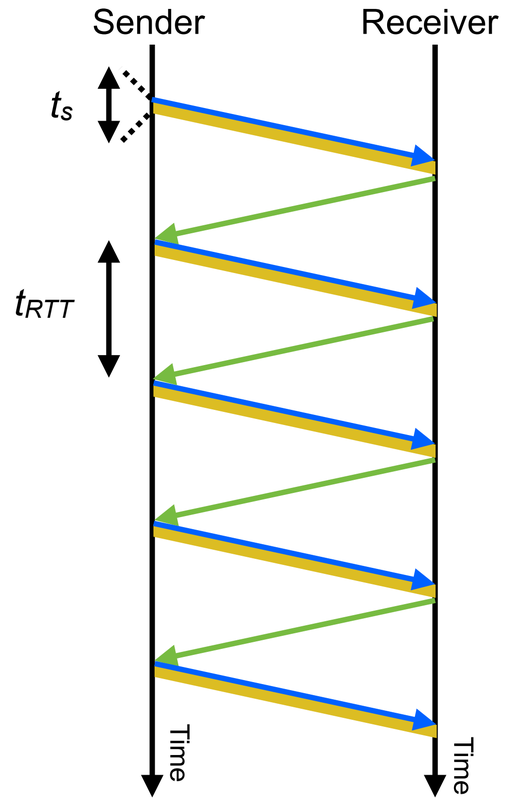
Congestion Control in TCP Reno

- TCP uses window-based congestion control
  - Maintains a **sliding window** onto the available data that determines how much can be sent according to the AIMD algorithm
  - Plus slow start and congestion avoidance
  - Gives approximately equal share of the bandwidth to each flow sharing a link

- Basic congestion control algorithm known as **TCP Reno**
  - The state of the art in TCP as of ~1990

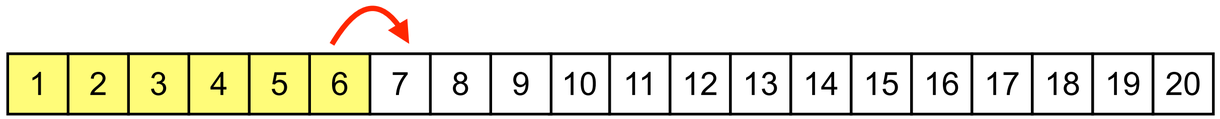
From Stop-and-Wait to Sliding Window Protocols

- Stop-and wait protocols perform poorly
  - It takes time, t_s, to send a packet
    - t_s = (packet size) / (link bandwidth)
  - Acknowledgement returns t_{RTT} seconds later
  - Link utilisation, $U = t_s / t_{RTT}$
    - Fraction of time link is sending packets → want $U \approx 1.0$
    - Assume a gigabit link sending a 1500 byte packet from Glasgow to London:
      - t_s = 1500×8 bits / 10^9 bits per second = 0.000012s
      - $t_{RTT} \approx$ 0.010 seconds
      - $U \approx$ 0.0012
      - *i.e.*, the link is in use 0.12% of the time
- Sliding window protocols improve on stop-and-wait by sending more than one packet before stopping for acknowledgement



Sliding Window Protocols Improve Link Utilisation

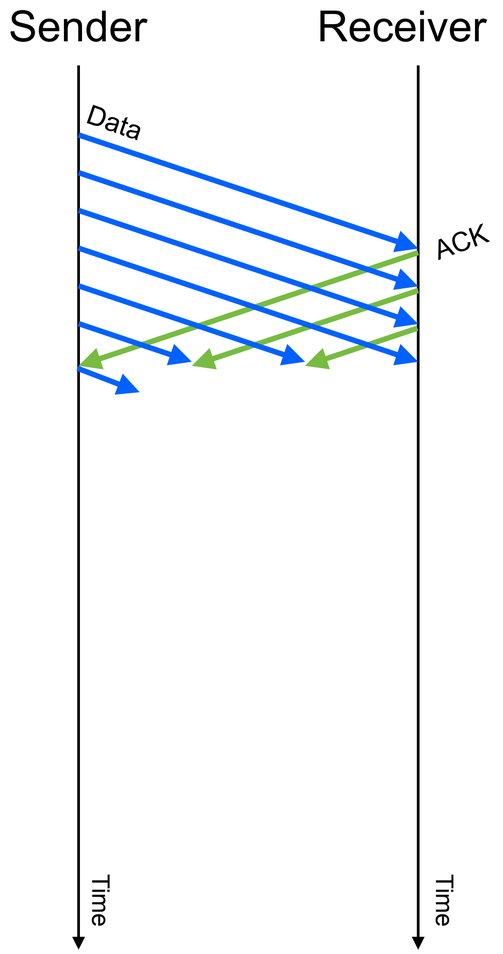
Sliding window protocols improve link utilisation using **a congestion window** – number of packets to be sent before an acknowledgement arrives



In this example, the window size is six packets → acknowledgement for packet 6 arrives just in time to release packet 7 for transmission

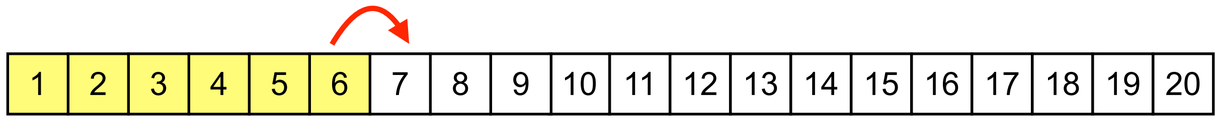
Each returning acknowledgement for new data slides the window along, releasing next packet for transmission → if window sized correctly, each acknowledgement **arrives just in time** to release next packet

What is the optimal size for the window? **bandwidth × delay** of path → **but neither is known to the sender**

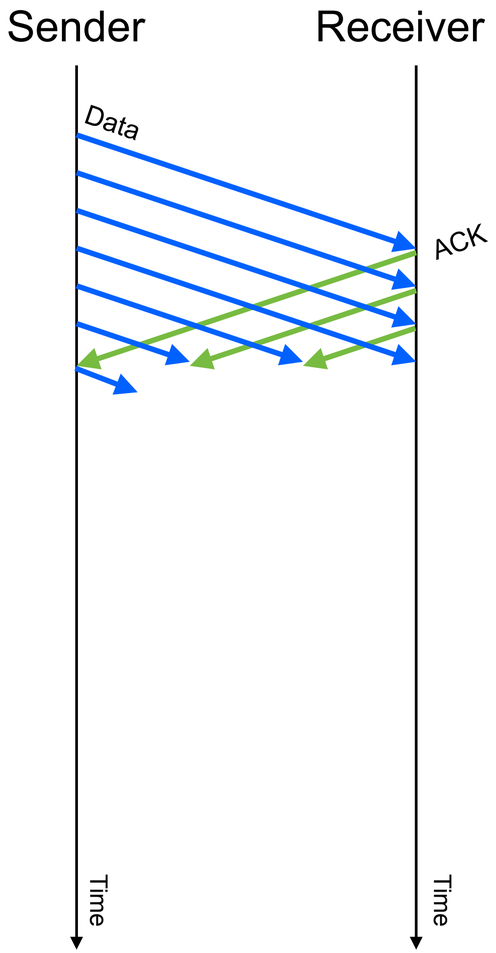


TCP Reno Congestion Control

- **TCP Reno is a sliding window protocol**
  - Optimised for not having information to know the correct window size

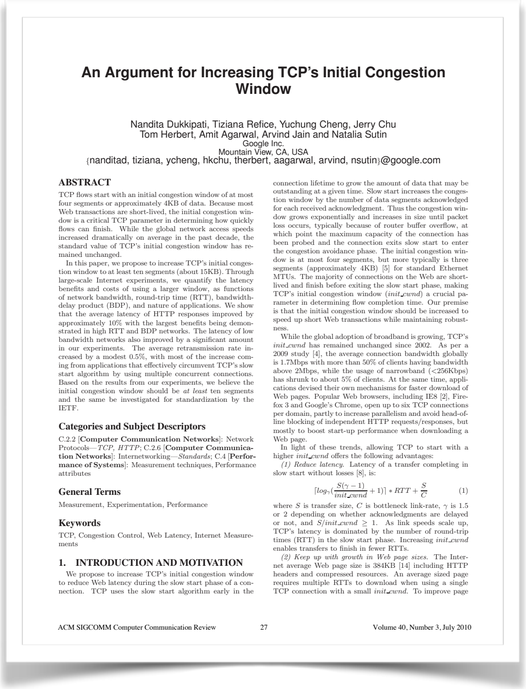


  - How to choose initial window?
  - How to find the path capacity?
    - Slow start to estimate the bottleneck link capacity
    - Congestion avoidance to probe for changes in capacity



TCP Reno: Choosing the Initial Window

- How to choose initial window size, $\mathbf{W_{init}}$?
  - No information → need to measure path capacity
  - Start with a small window, increase until congestion
  - **W_{init} = 1 packet per round-trip time (RTT) is safe**
    - Start at the slowest possible rate, equivalent to stop-and-wait, and increase
  - **W_{init}** = 3 packets per RTT
    - Traditional **TCP Reno** approach
  - **W_{init}** = 10 packets per RTT
    - **Modern TCP implementations** [RFC 6928]
    - Compromise between safety and performance – measurements show this is generally safe at present
    - Expect **W_{init}** to gradually be increased as average network performance improves



An Argument for Increasing TCP's Initial Congestion Window

Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu
Tom Herbert, Amit Agarwal, Arvind Jain and Natalia Sutin
Google Inc.
Mountain View, CA, USA
{nanditad, tiziana, ycheng, hkchu, therbert, aagarwal, arvind, nsutin}@google.com

ABSTRACT

TCP flows start with an initial congestion window of at most four segments or approximately 4KB of data. Because most Web transactions are short-lived, the initial congestion window is a critical TCP parameter in determining how quickly flows can finish. While the global network access speeds increased dramatically on average in the past decade, the standard value of TCP's initial congestion window has remained unchanged.

In this paper, we propose to increase TCP's initial congestion window to at least ten segments (about 15KB). Through large-scale Internet experiments, we quantify the latency benefits and costs of using a larger window, as functions of network bandwidth, round-trip time (RTT), bandwidth-delay product (BDP), and nature of applications. We show that the average latency of HTTP responses improved by approximately 10% with the largest benefits being demonstrated in high RTT and BDP networks. The latency of low bandwidth networks also improved by a significant amount in our experiments. The average retransmission rate increased by a modest 0.5%, with most of the increase coming from applications that effectively circumvent TCP's slow start algorithm by using multiple concurrent connections. Based on the results from our experiments, we believe the initial congestion window should be *at least* ten segments and the same be investigated for standardization by the IETF.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols—*TCP, HTTP*; C.2.6 [**Computer Communication Networks**]: Internetworking—*Standards*; C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes

General Terms

Measurement, Experimentation, Performance

Keywords

TCP, Congestion Control, Web Latency, Internet Measurements

1. INTRODUCTION AND MOTIVATION

We propose to increase TCP's initial congestion window to reduce Web latency during the slow start phase of a connection. TCP uses the slow start algorithm early in the connection lifetime to grow the amount of data that may be outstanding at a given time. Slow start increases the congestion window by the number of data segments acknowledged for each received acknowledgment. Thus the congestion window grows exponentially and increases in size until packet loss occurs, typically because of router buffer overflow, at which point the maximum capacity of the connection has been probed and the connection exits slow start to enter the congestion avoidance phase. The initial congestion window is at most four segments, but more typically is three segments (approximately 4KB) [5] for standard Ethernet MTUs. The majority of connections on the Web are short-lived and finish before exiting the slow start phase, making TCP's initial congestion window ($init\_cwnd$) a crucial parameter in determining flow completion time. Our premise is that the initial congestion window should be increased to speed up short Web transactions while maintaining robustness.

While the global adoption of broadband is growing, TCP's $init\_cwnd$ has remained unchanged since 2002. As per a 2009 study [4], the average connection bandwidth globally is 1.7Mbps with more than 50% of clients having bandwidth above 2Mbps, while the usage of narrowband (<256Kbps) has shrunk to about 5% of clients. At the same time, applications devised their own mechanisms for faster download of Web pages. Popular Web browsers, including IE8 [2], Firefox 3 and Google's Chrome, open up to six TCP connections per domain, partly to increase parallelism and avoid head-of-line blocking of independent HTTP requests/responses, but mostly to boost start-up performance when downloading a Web page.

In light of these trends, allowing TCP to start with a higher $init\_cwnd$ offers the following advantages:

(1) Reduce latency. Latency of a transfer completing in slow start without losses [8], is:

$$\lceil log_\gamma(\frac{S(\gamma-1)}{init\_cwnd}+1)\rceil * RTT + \frac{S}{C} \quad (1)$$

where S is transfer size, C is bottleneck link-rate, γ is 1.5 or 2 depending on whether acknowledgments are delayed or not, and $S/init\_cwnd \geq 1$. As link speeds scale up, TCP's latency is dominated by the number of round-trip times (RTT) in the slow start phase. Increasing $init\_cwnd$ enables transfers to finish in fewer RTTs.

(2) Keep up with growth in Web page sizes. The Internet average Web page size is 384KB [14] including HTTP headers and compressed resources. An average sized page requires multiple RTTs to download when using a single TCP connection with a small $init\_cwnd$. To improve page

ACM SIGCOMM Computer Communication Review 27 Volume 40, Number 3, July 2010

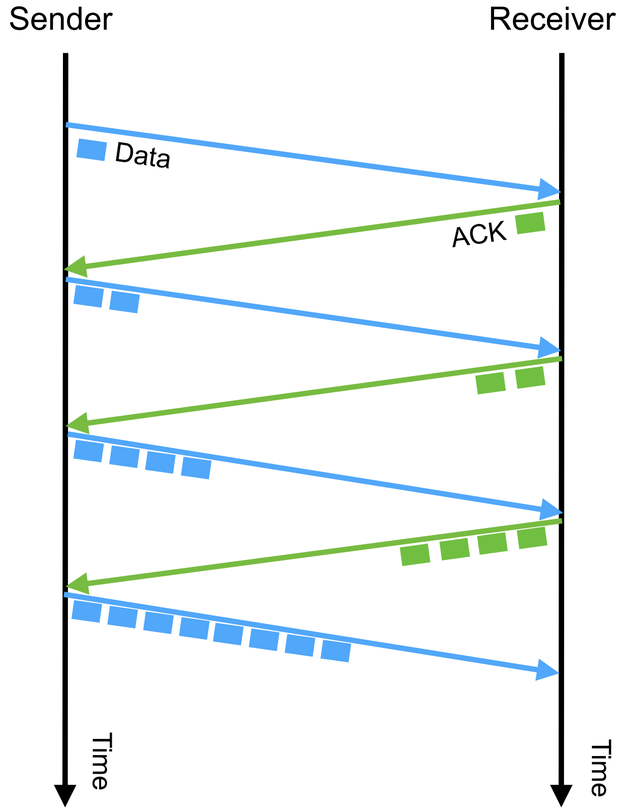
N. Dukkipati *et* al, "An argument for increasing TCP's initial congestion window", ACM Computer Communication Review, 40(3):27–33, July 2010. http://dx.doi.org/10.1145/1823844.1823848

TCP Reno: Finding the Path Capacity

- The initial window allows you to send **something**
  - Unlikely to be the optimal window size
  - How to choose the correct window size to match the link capacity?
    - **Slow start** to rapidly find the correct window size for the path
    - **Congestion avoidance** to adapt to changes in path capacity once connection is running

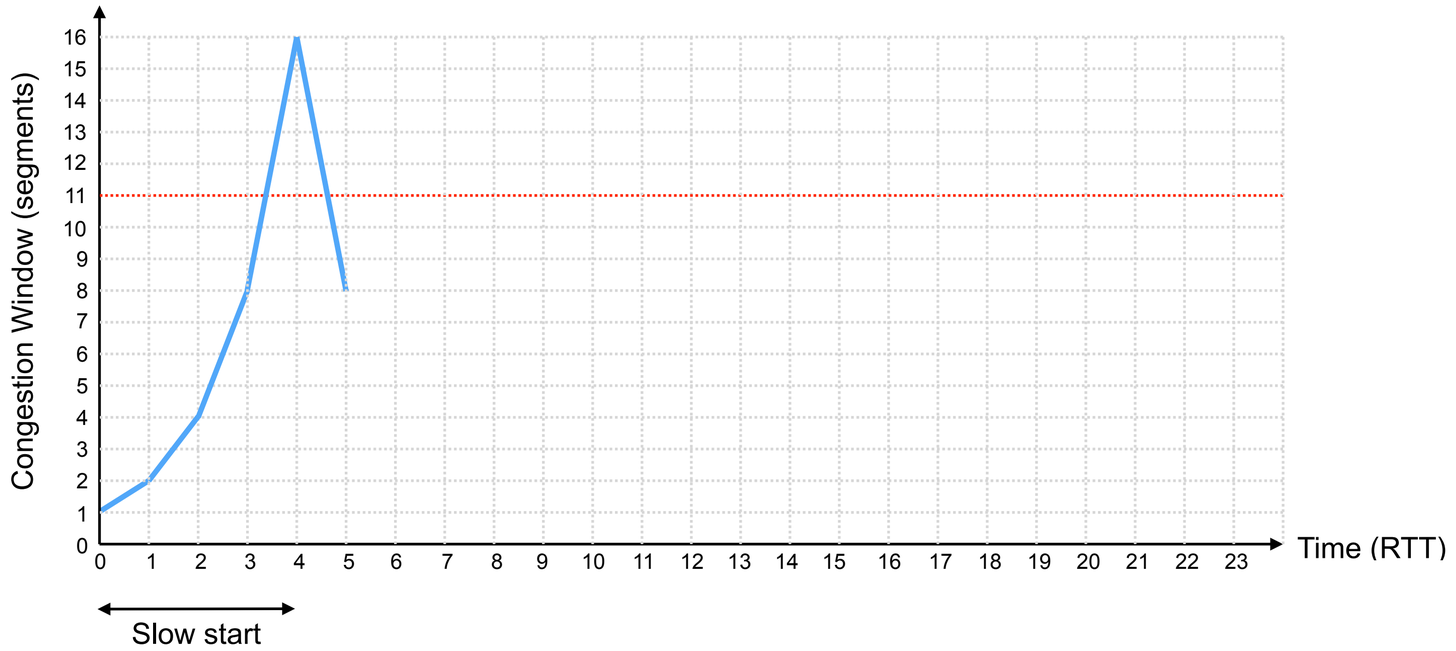
TCP Reno: Slow Start (1/2)

- Assume W_{init} = 1 packet per RTT – **slow start to connection**
  - It will be W_{init} = 3 or W_{init} = 10 in practice, but using W_{init} = 1 makes the example simpler…
- Rapidly increase window until network capacity is reached
  - Each acknowledgement for new data increases congestion window, **W**, by 1 packet per RTT → congestion window doubles each RTT
  - If a packet is lost, halve congestion window back to previous value and exit slow start



TCP Reno: Slow Start (2/2)

- TCP Reno slow start phase:
  - Starts sending slowly
  - Rapidly increases sending rate until a packet is lost
  - Resets sending rate to last known good rate when first loss occurs

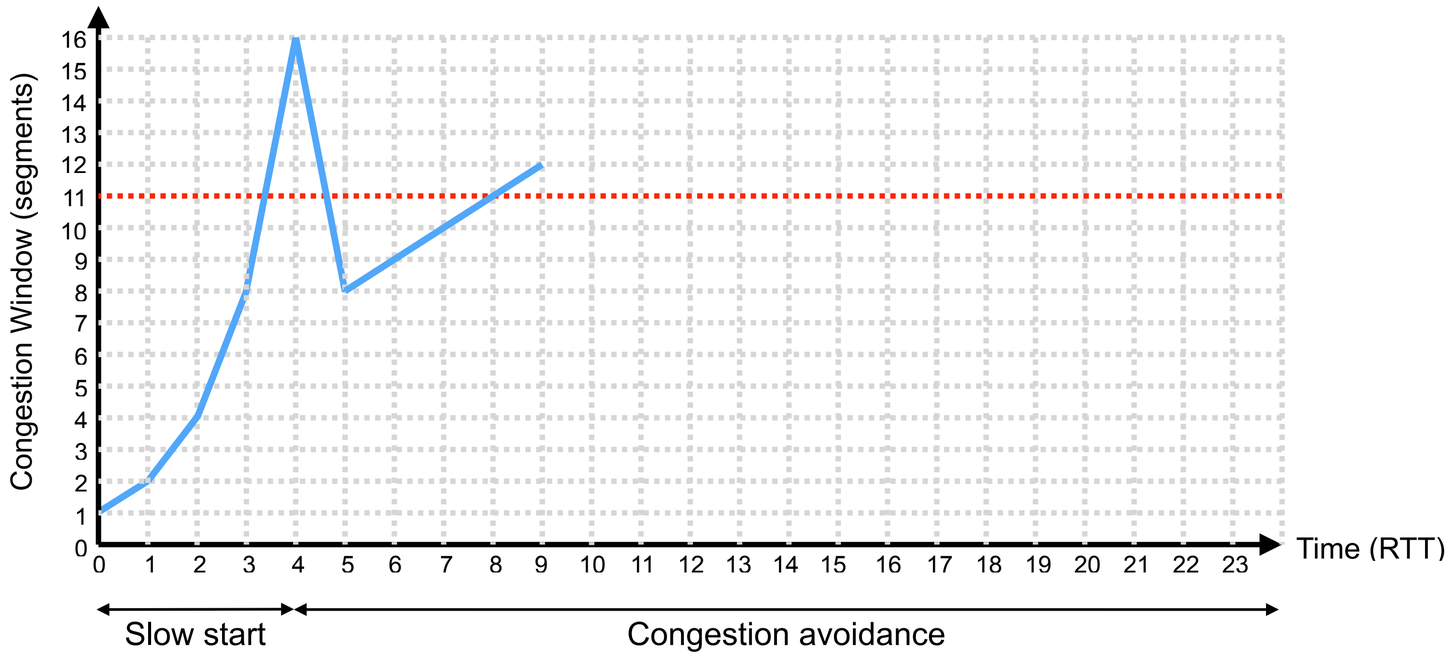


TCP Reno: Congestion Avoidance (1/4)

- After first packet is lost, TCP switches to **congestion avoidance**
  - The congestion window is now approximately the right size for the path
  - Goal is now to adapt to changes in network capacity
    - Perhaps the path capacity changes – radio signal strength changes for mobile device
    - Perhaps the other traffic changes – competing flows stop; additional cross-traffic starts
  - Additive increase, multiplicative decrease of congestion window

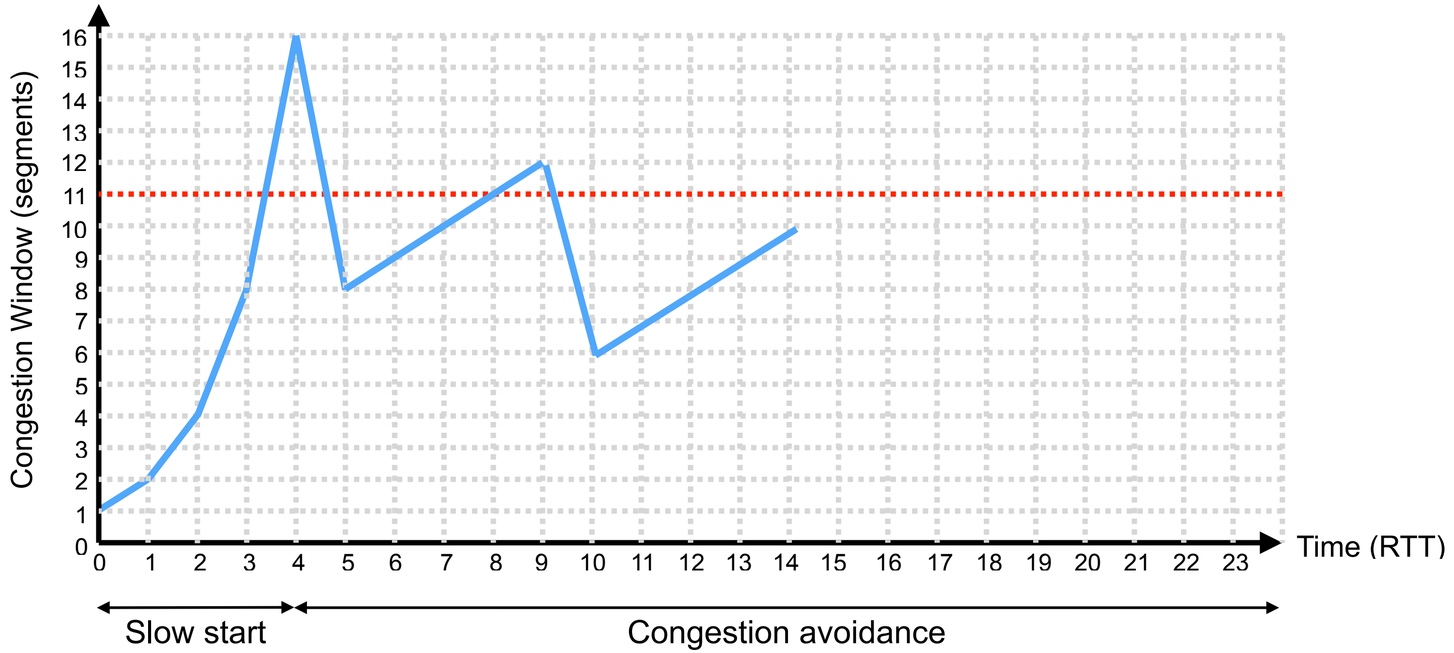
TCP Reno: Congestion Avoidance (2/4)

- If a complete window of packets is sent without loss:
  - Increase congestion window by 1 packet per RTT, then send next window worth of packets
  - Slow, linear, additive increase in window: $\mathbf{W_i = W_{i-1} + 1}$



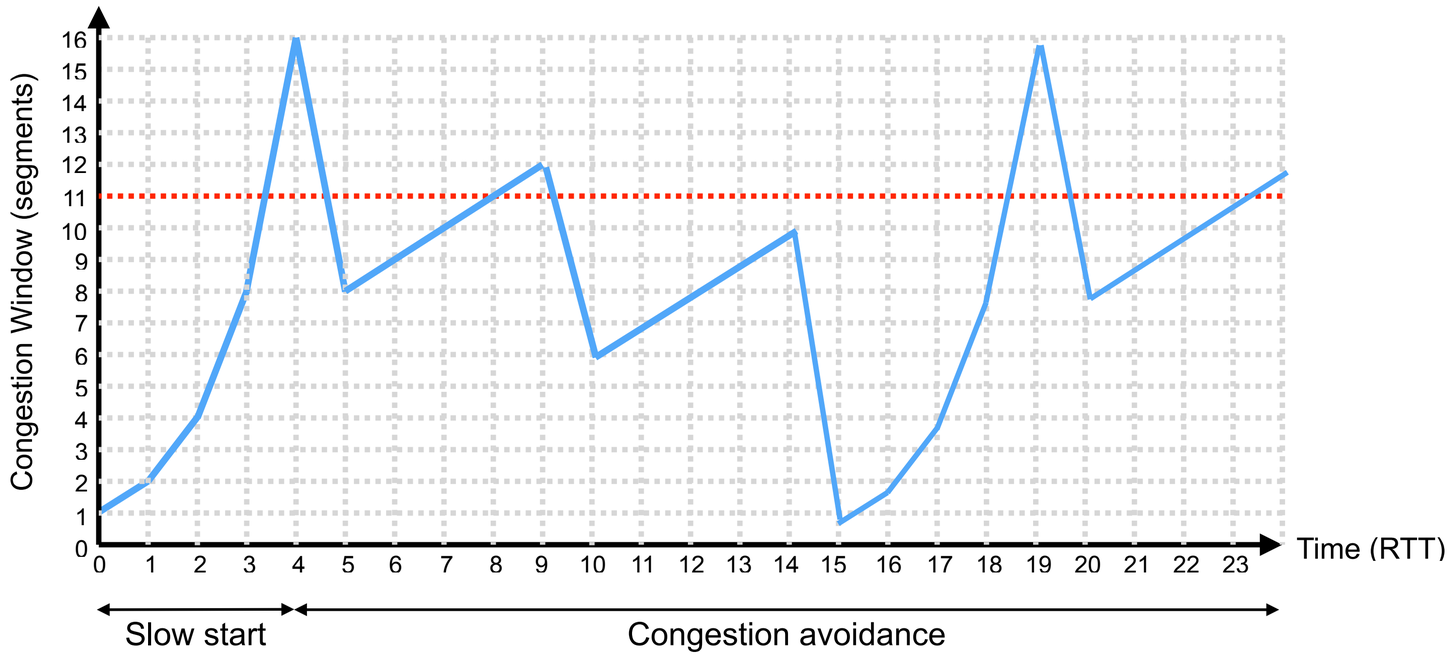
TCP Reno: Congestion Avoidance (3/4)

- If a packet is lost and detected via triple duplicate acknowledgement:
  - Transient congestion, but data still being received
  - Multiplicative decrease in window: $\mathbf{W_i = W_{i-1} \times 0.5}$
  - Rapid reduction in window allows congestion to clear quickly, avoids congestion collapse
- Then, return to additive increase until next loss

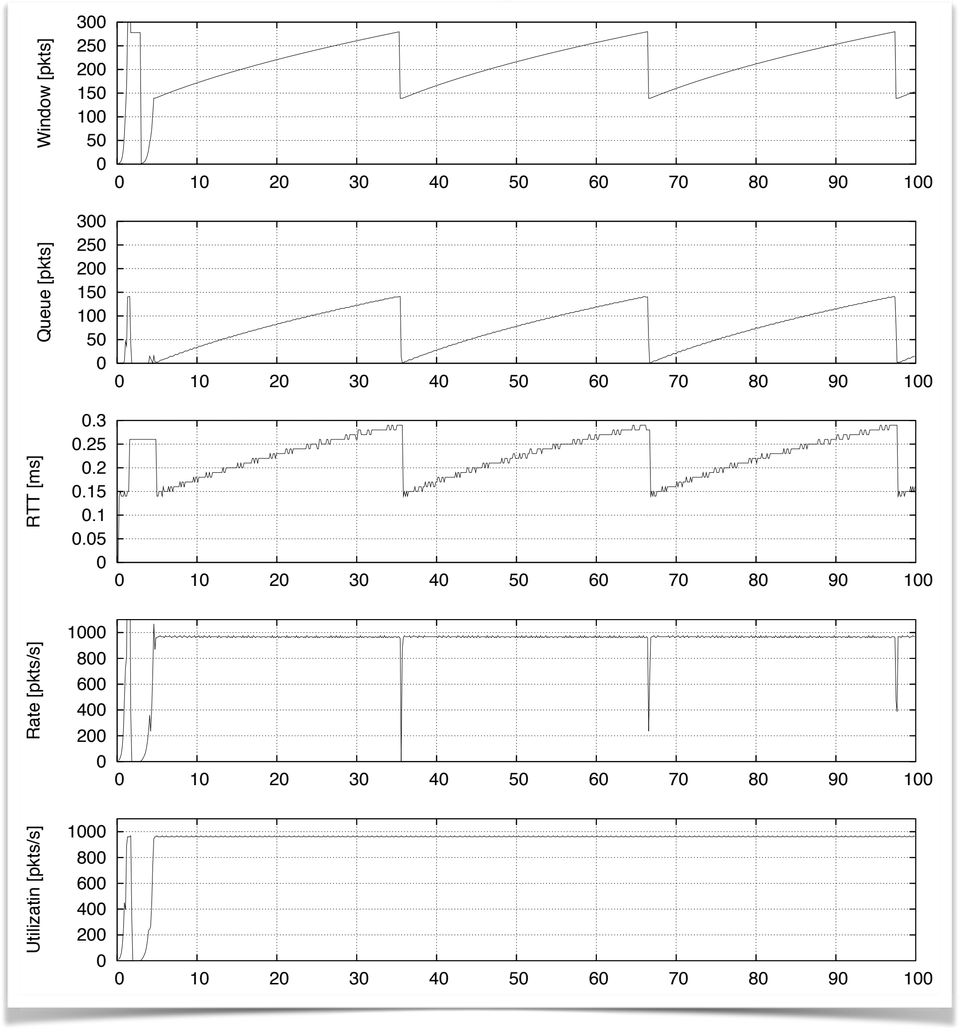


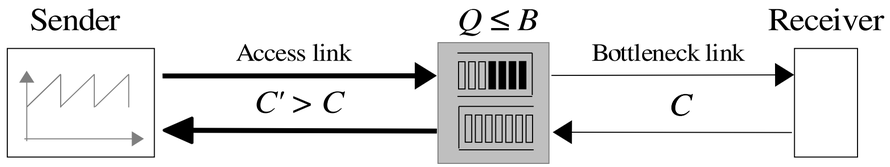
TCP Reno: Congestion Avoidance (4/4)

- If a packet is lost and detected via timeout:
  - Either receiver or path has failed – reset to W_{init} and re-enter slow start
  - How long is the timeout?
    - T_{rto} = max(1 second, average RTT + (4 x RTT variance))



Congestion Window Growth, Buffering, Throughput





Bottleneck buffer size = bandwidth × delay

- Bottleneck queue never empty
- Bottleneck link never becomes idle → sending rate varies, but receiver sees continuous flow

- Congestion window follows a “sawtooth” pattern, but received rate is constant at approximately bottleneck bandwidth

Source: G. Appenzeller, “Sizing Router Buffers”, PhD thesis, Stanford University, March 2005. http://tiny-tera.stanford.edu/~nickm/papers/guido-thesis.pdf (Figures 2.1 and 2.2)

TCP Reno: Discussion

- TCP Reno is effective at keeping bottleneck link fully utilised
  - Trades some extra delay to maintain throughput
  - Provided sufficient buffering in the network: buffer size = bandwidth × delay
  - Packets queued in buffer → delay

- Limitations:
  - Assumes packet loss is due to congestion; non-congestive loss, e.g., due to wireless interference, impacts throughput
  - Congestion avoidance phase takes a long time to use increased capacity

TCP Reno

- Basic TCP congestion control
- Sliding window algorithms
- Slow start
- Congestion avoidance