University of Glasgow

Lowering Latency

Networked Systems (H) – Lecture 6

2019-2020

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866,

Lecture Outline

- Key factor limiting performance is **latency**
  - Connection establishment latency
    - Reduce number of RTTs to improve connection setup time
    - Reduce number of connections needed
    - TCP + TLS 1.3 → QUIC
      - Overlapping connection establishment and security handshake
      - 0-RTT connection re-establishment with idempotent data
      - Stream multiplexing within a single connection
  - Data transfer latency
    - Impact of TCP congestion control
    - Impact of delay-based congestion control
    - Impact of explicit congestion notification
    - Impact of the speed of light

→ Lectures 2-4

→ This lecture

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

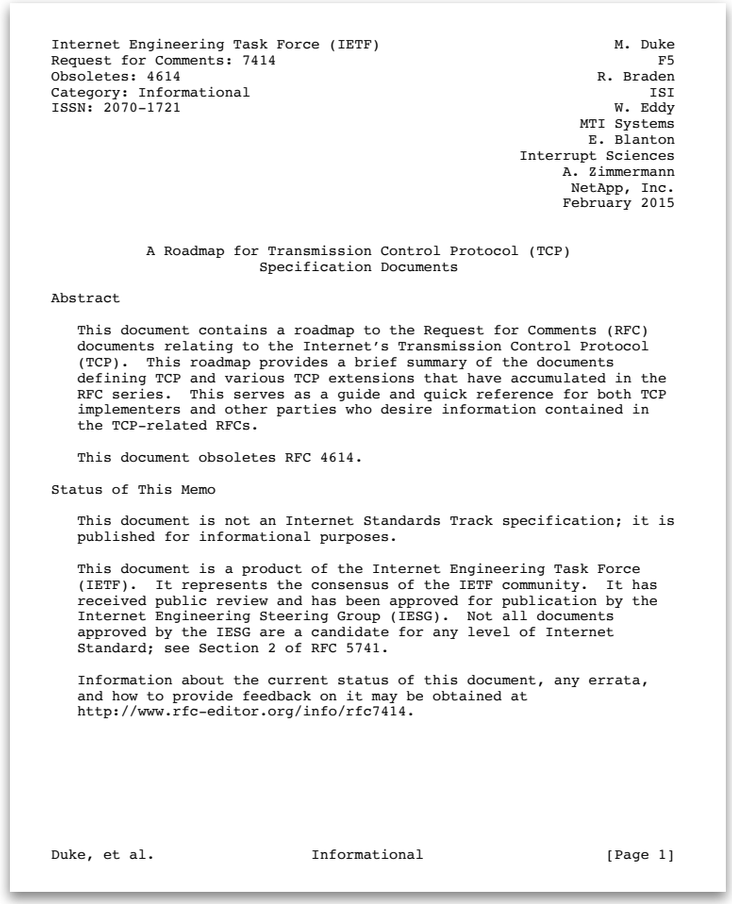
TCP Congestion Control

CC BY ND

TCP Congestion Control

- Congestion control principles
- Loss-based congestion control
  - How to effectively make use of network capacity
    - TCP Reno
    - TCP Cubic
- Delay-based congestion control
  - Reducing TCP-induced latency
    - TCP Vegas
    - TCP BBR
- Explicit congestion notification

- TCP is a complex, **highly optimised**, protocol
  - This lecture is a **very** simplified review of complex issues



Internet Engineering Task Force (IETF) — M. Duke, F5
Request for Comments: 7414
Obsoletes: 4614 — R. Braden, ISI
Category: Informational — W. Eddy, MTI Systems
ISSN: 2070-1721 — E. Blanton, Interrupt Sciences
A. Zimmermann, NetApp, Inc.
February 2015

A Roadmap for Transmission Control Protocol (TCP) Specification Documents

Abstract

This document contains a roadmap to the Request for Comments (RFC) documents relating to the Internet's Transmission Control Protocol (TCP). This roadmap provides a brief summary of the documents defining TCP and various TCP extensions that have accumulated in the RFC series. This serves as a guide and quick reference for both TCP implementers and other parties who desire information contained in the TCP-related RFCs.

This document obsoletes RFC 4614.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

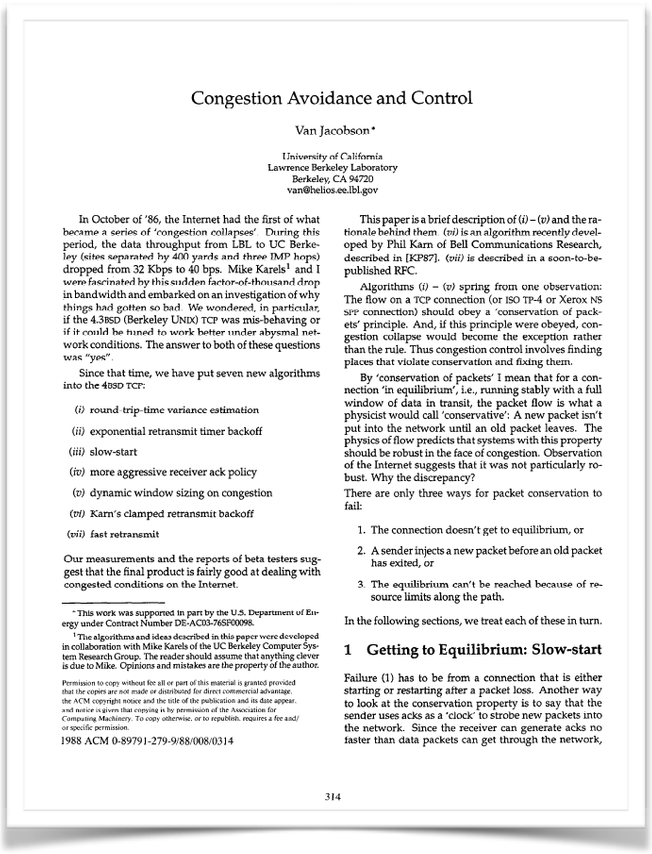
Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7414.

Duke, et al. — Informational — [Page 1]

M. Duke, R. Braden, W. Eddy, E. Blanton, and A. Zimmermann, "A Roadmap for TCP Specification Documents", IETF, RFC 7414, February 2015 (https://tools.ietf.org/html/rfc7414) (57 pages, referencing ~150 other documents)

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Congestion Control Principles



Congestion Avoidance and Control

Van Jacobson*

University of California
Lawrence Berkeley Laboratory
Berkeley, CA 94720
van@helios.ee.lbl.gov

In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and three IMP hops) dropped from 32 Kbps to 40 bps. Mike Karels[1] and I were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. We wondered, in particular, if the 4.3BSD (Berkeley UNIX) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was "yes".

Since that time, we have put seven new algorithms into the 4BSD TCP:

(*i*) round-trip-time variance estimation
(*ii*) exponential retransmit timer backoff
(*iii*) slow-start
(*iv*) more aggressive receiver ack policy
(*v*) dynamic window sizing on congestion
(*vi*) Karn's clamped retransmit backoff
(*vii*) fast retransmit

Our measurements and the reports of beta testers suggest that the final product is fairly good at dealing with congested conditions on the Internet.

This paper is a brief description of (*i*) - (*v*) and the rationale behind them. (*vi*) is an algorithm recently developed by Phil Karn of Bell Communications Research, described in [KP87]. (*vii*) is described in a soon-to-be-published RFC.

Algorithms (*i*) - (*v*) spring from one observation: The flow on a TCP connection (or ISO TP-4 or Xerox NS SPP connection) should obey a 'conservation of packets' principle. And, if this principle were obeyed, congestion collapse would become the exception rather than the rule. Thus congestion control involves finding places that violate conservation and fixing them.

By 'conservation of packets' I mean that for a connection 'in equilibrium', i.e., running stably with a full window of data in transit, the packet flow is what a physicist would call 'conservative': A new packet isn't put into the network until an old packet leaves. The physics of flow predicts that systems with this property should be robust in the face of congestion. Observation of the Internet suggests that it was not particularly robust. Why the discrepancy?

There are only three ways for packet conservation to fail:

1. The connection doesn't get to equilibrium, or
2. A sender injects a new packet before an old packet has exited, or
3. The equilibrium can't be reached because of resource limits along the path.

In the following sections, we treat each of these in turn.

1 Getting to Equilibrium: Slow-start

Failure (1) has to be from a connection that is either starting or restarting after a packet loss. Another way to look at the conservation property is to say that the sender uses acks as a 'clock' to strobe new packets into the network. Since the receiver can generate acks no faster than data packets can get through the network,

* This work was supported in part by the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

[1] The algorithms and ideas described in this paper were developed in collaboration with Mike Karels of the UC Berkeley Computer System Research Group. The reader should assume that anything clever is due to Mike. Opinions and mistakes are the property of the author.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1988 ACM 0-89791-279-9/88/008/0314

V. Jacobson, "Congestion avoidance and control", Proceedings of the SIGCOMM Conference, Stanford, CA, USA, August 1988. ACM. http://dx.doi.org/10.1145/52324.52356

- Key congestion control principles:
  - Packet loss as a congestion signal
  - Conservation of packets in flight
  - Additive increase/multiplicative decrease



Source: PARC

Van Jacobson

- Practical TCP congestion control:
  - Robust IETF standards for TCP
  - Embodies congestion control principles
  - **Extremely high performance**



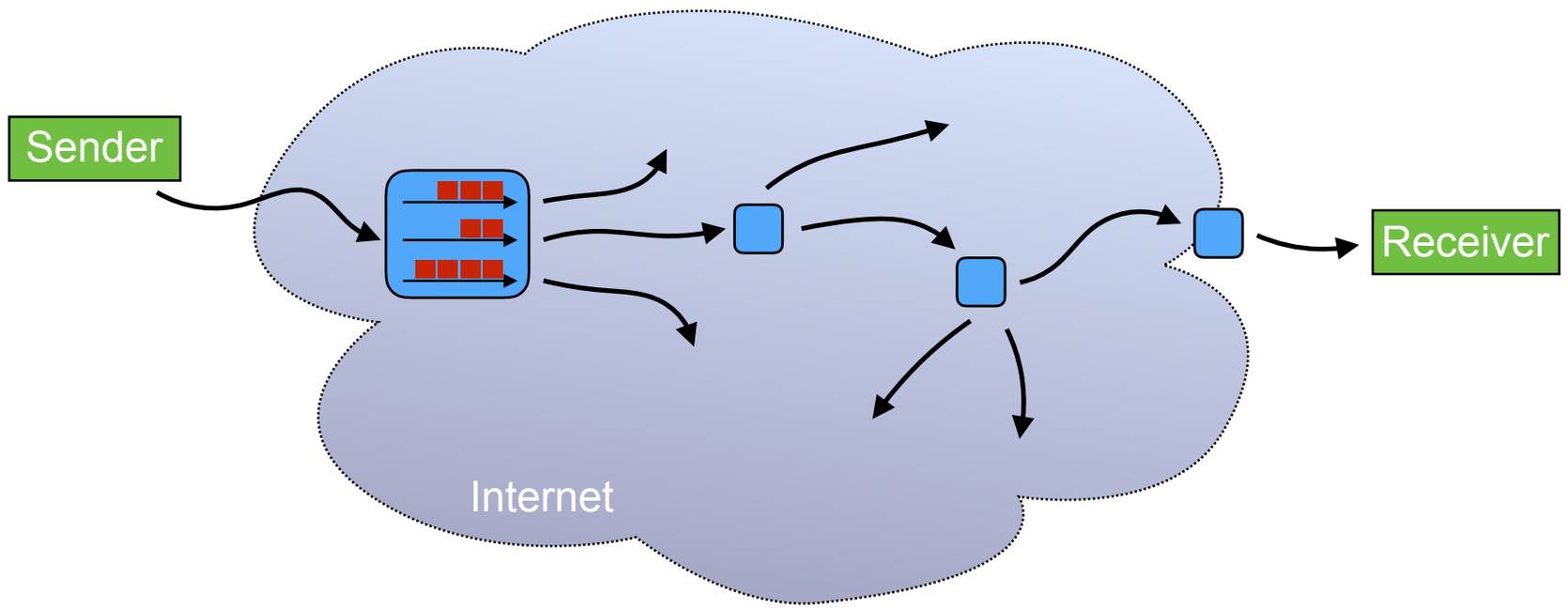
Source: Sally Floyd

Sally Floyd

https://www.nytimes.com/2019/09/04/science/sally-floyd-dead.html

CC BY ND

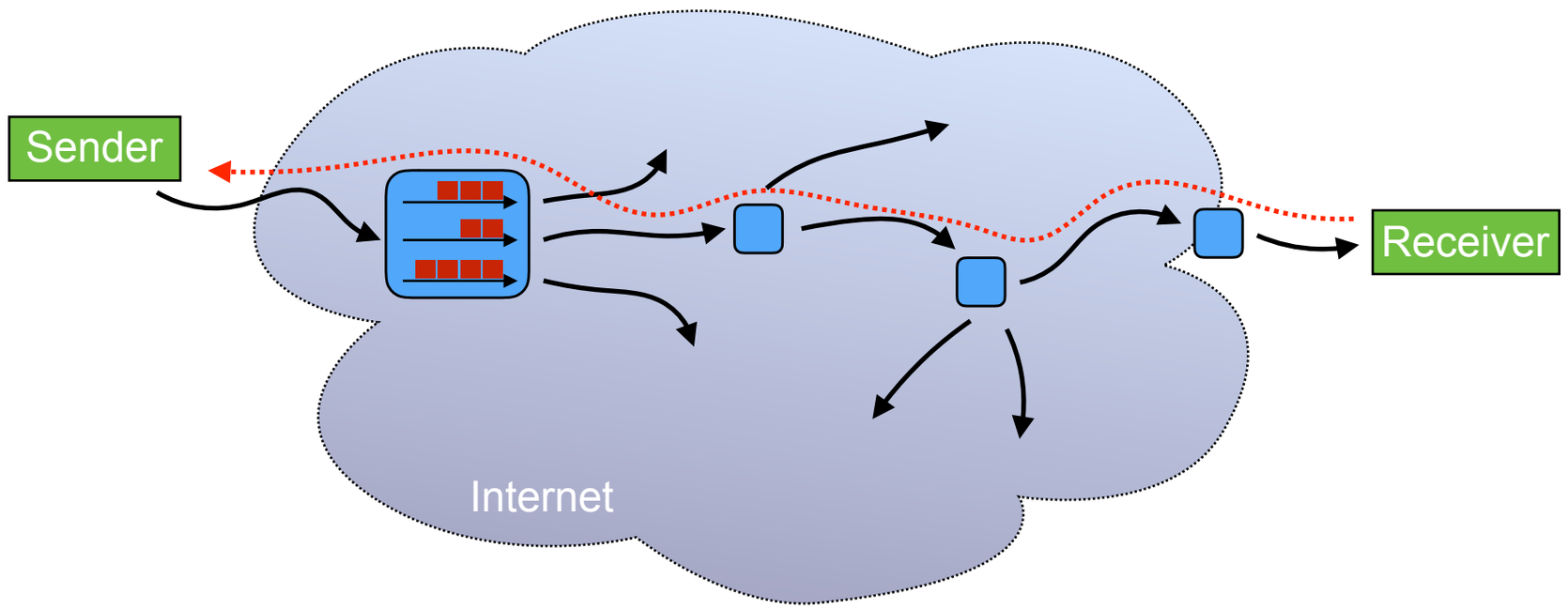
Key Principles: Packet Loss as a Congestion Signal



- Data flows from sender to receiver through a series of **IP routers**
- Routers perform two functions:
  - **Routing:** receive packets, determine appropriate route to destination (→ Lecture 9)
  - **Forwarding**: enqueue packets on outgoing link for delivery
    - Queues shrinks if packets are being forwarded faster than they arrive
    - Queues grows if packets arrive faster than they can be forwarded onwards
    - Queues have maximum size → **packets discarded if the queue is full**
    - Congestion control can use this packet loss as a congestion signal

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

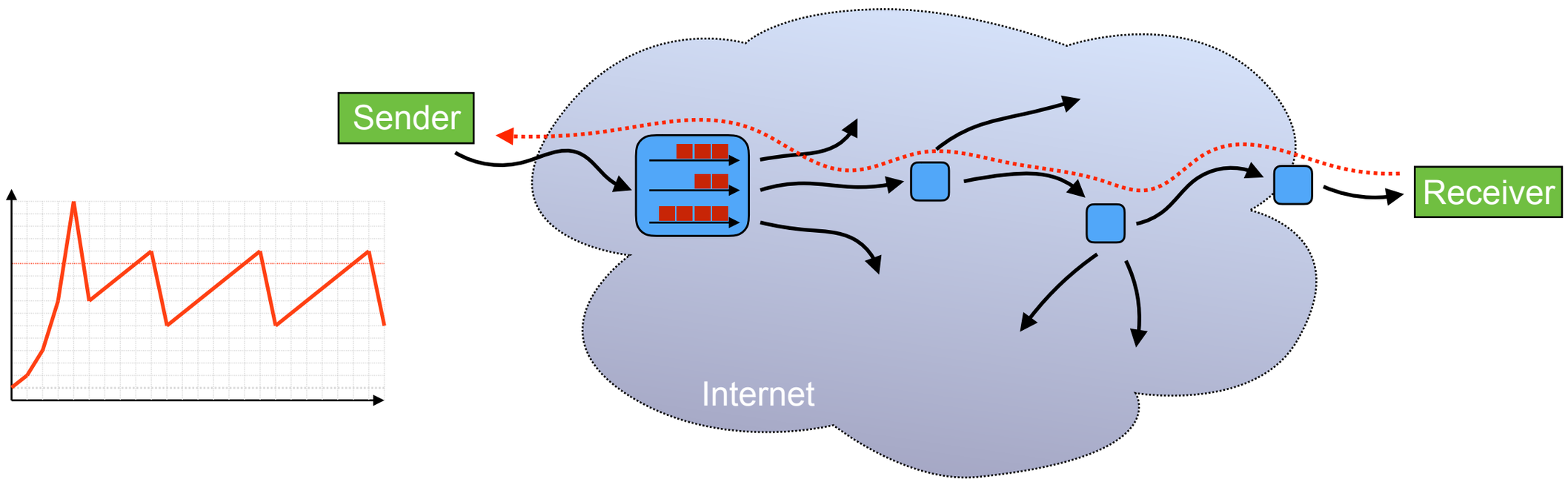
Key Principles: Conservation of Packets



- Acknowledgements returned by receiver
  - Send one packet for each acknowledgement received
  - Total number of packets in transit is constant
    - System is in equilibrium; queues neither increase nor decrease in size
    - **ACK clocking** – each acknowledgement “clocks out” the next packet
- Automatically reduces sending rate if network gets congested and delivers packets more slowly

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Key Principles: AIMD Algorithms



- Sending rate follows additive increase/multiplicative decrease (AIMD) algorithm
  - Start slowly, increase gradually to find equilibrium
    - Add a small amount to the sending speed each time interval without loss
    - For a window-based algorithm $w_i = w_{i-1} + \alpha$ each RTT, where $\alpha = 1$ typically
  - Respond to congestion rapidly
    - Multiply sending window by some factor $\beta < 1$ each interval loss seen
    - For a window-based algorithm $w_i = w_{i-1} \times \beta$ each RTT, where $\beta = 1/2$ typically
  - Faster reduction than increase → stability; TCP Reno

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Reno

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Congestion Control in TCP Reno

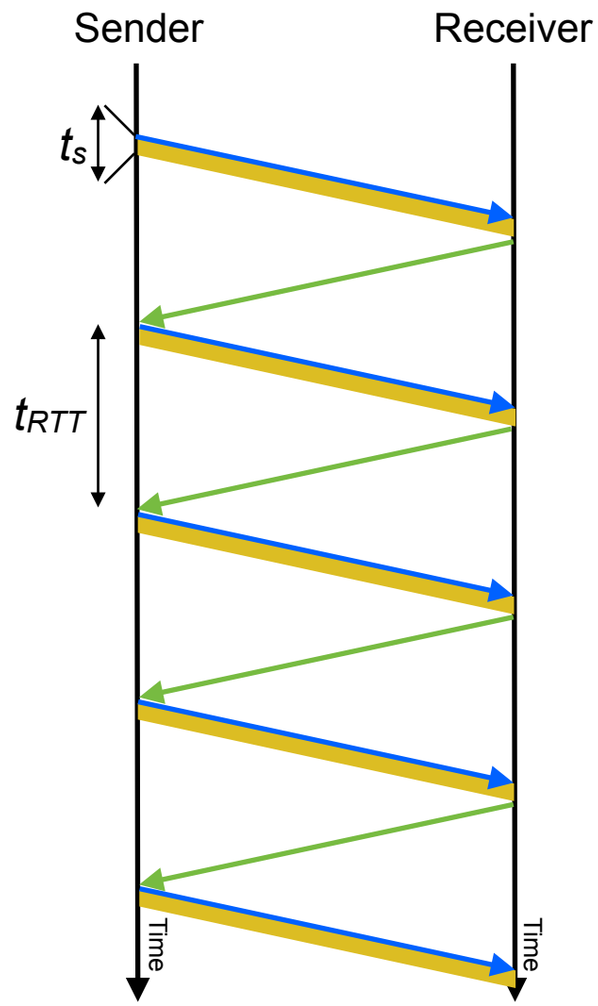
- TCP uses window-based congestion control
  - Maintains a **sliding window** onto the available data that determines how much can be sent according to the AIMD algorithm
  - Plus slow start and congestion avoidance
  - Gives approximately equal share of the bandwidth to each flow sharing a link

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

cc BY ND

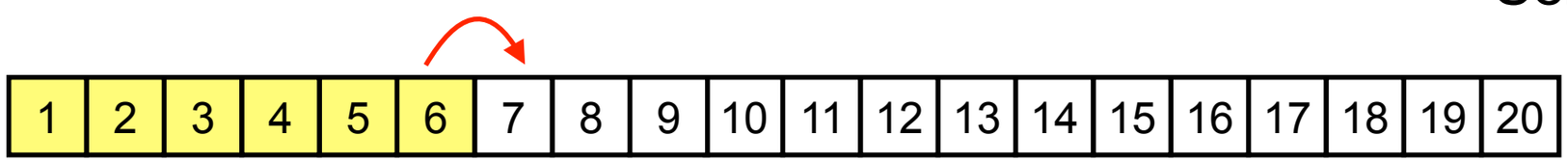
Sliding Window Protocols

- 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
  - TCP Reno is a sliding window protocol



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Sliding Window Protocols Improve Link Utilisation

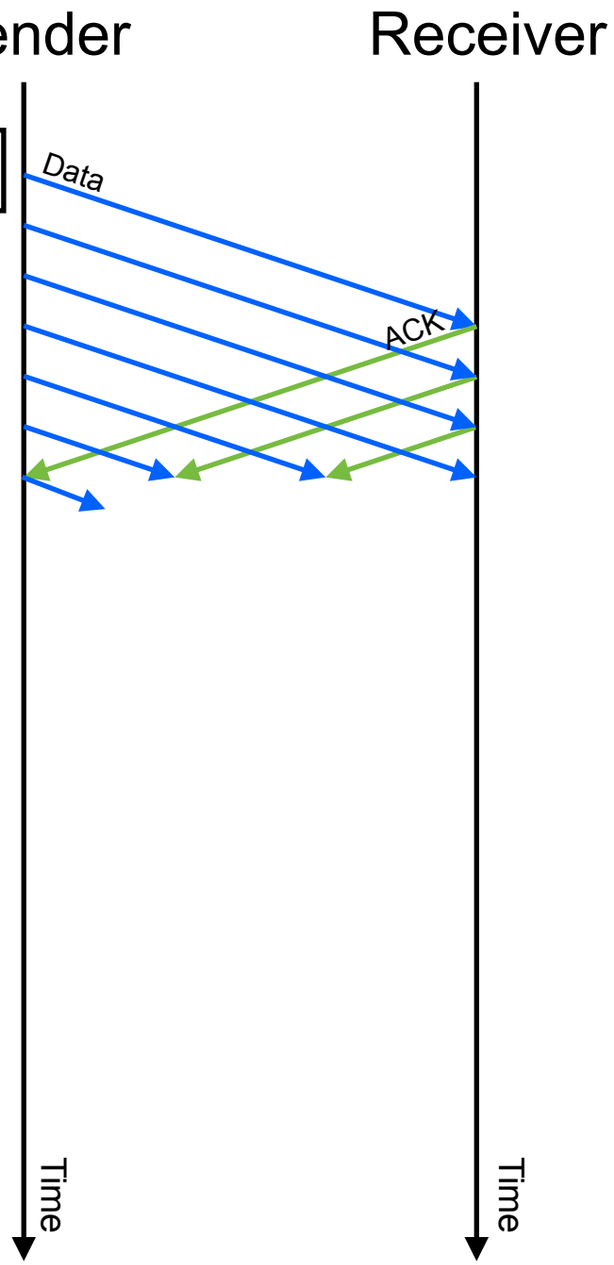


Sliding window protocols improve link utilisation using **a congestion window** – number of packets to be sent before an acknowledgement arrives

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

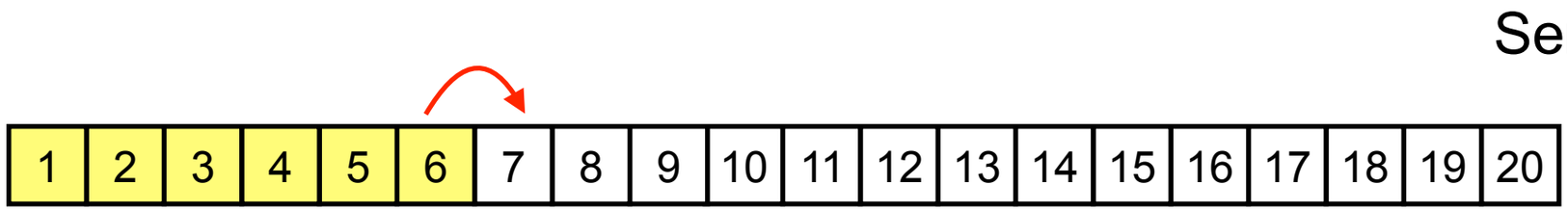
Each returning acknowledgement for new data slides the window along, releasing next packet for transmission → if window sized correctly, each acknowledgement **arrives just in time** to release next packet

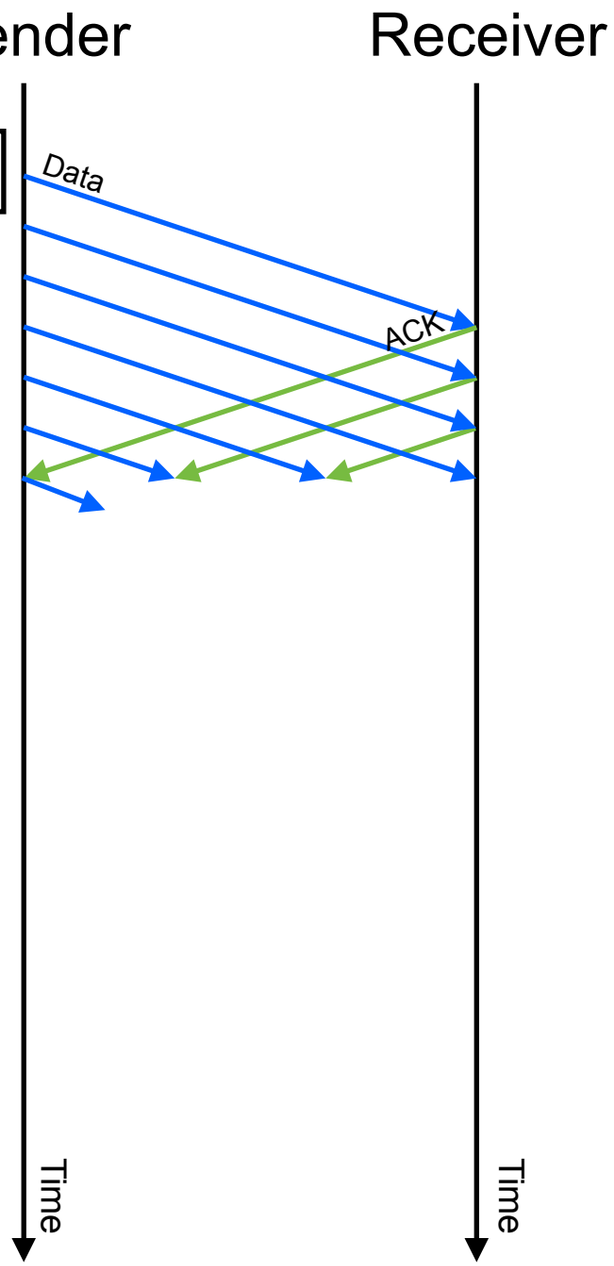
What is the optimal size for the window? **bandwidth × delay** of path
→ **but neither is known to the sender**



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Reno Congestion Control



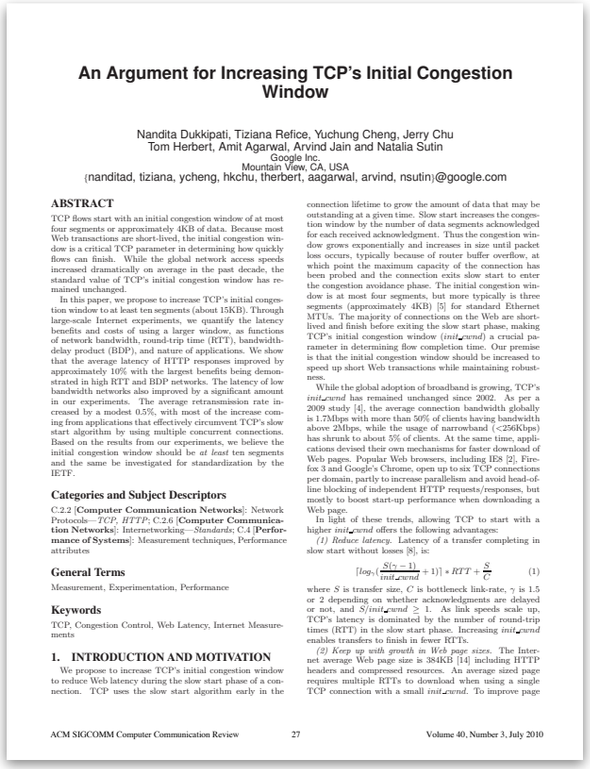


- **TCP Reno is a sliding window protocol**
  - Optimised for not having information to know the correct window size
  - How to choose initial window?
  - How to find the path capacity?
    - Slow start to estimate the bottleneck link capacity
    - Congestion avoidance to probe for changes in capacity

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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
    - **$\mathbf{W_{init}}$ = 1 packet per round-trip time (RTT) is safe**
      - Start at the slowest possible rate, equivalent to stop-and-wait, and increase
    - **$\mathbf{W_{init}}$ =** 3 packets per RTT
      - Traditional **TCP Reno** approach
  - **$\mathbf{W_{init}}$** = 10 packets per RTT
    - **Modern TCP implementations** [RFC 6928]
    - Compromise between safety and performance – measurements show this is generally safe at present
    - Expect **$\mathbf{W_{init}}$** 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

N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. Computer Communication Review, 40(3):27–33, July 2010. http://dx.doi.org/10.1145/1823844.1823848

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

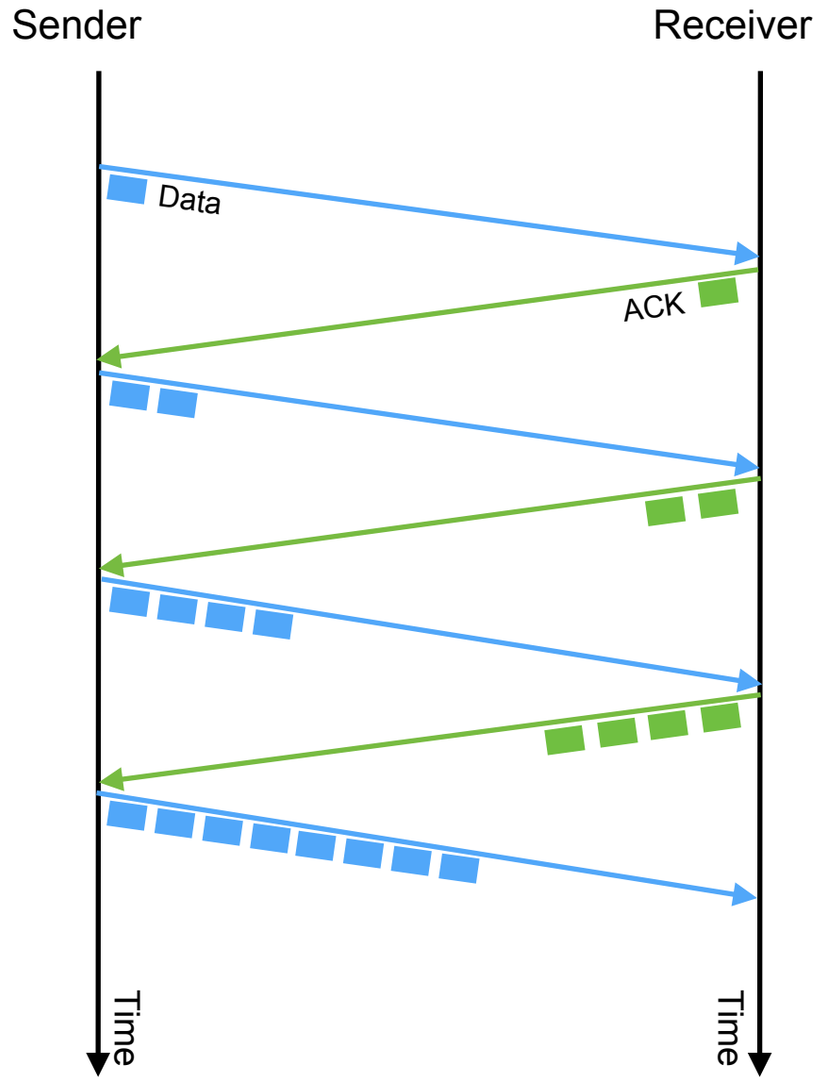
TCP Reno: Finding the Path Capacity

- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Reno: Slow Start (1/2)

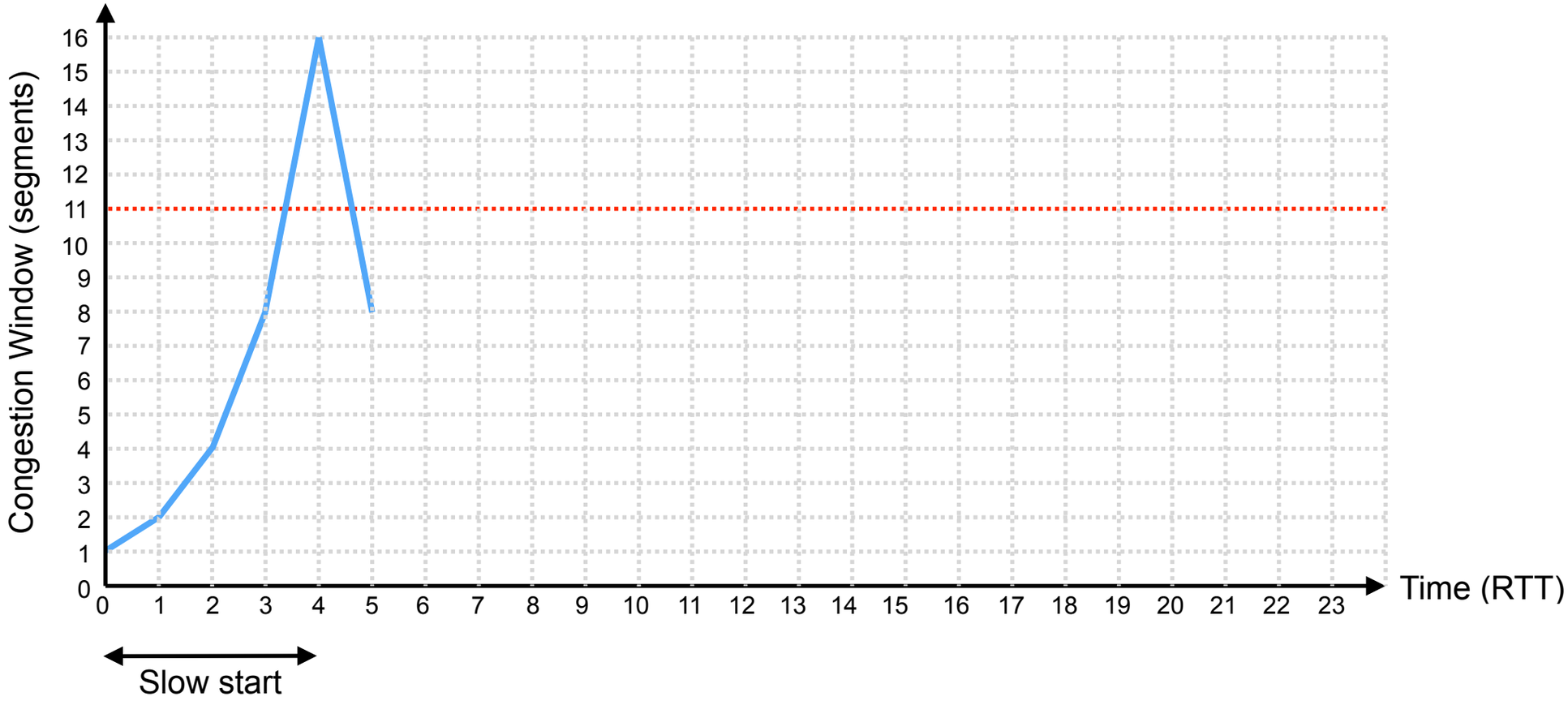
- Assume W_{init} = 1 packet per RTT – **a slow start to the connection**
  - It will be W_{init} = 3 or W_{init} = 10 in practice, but using W_{init} = 1 makes the example simpler…
- Rapidly increase window until network capacity is reached
  - Each acknowledgement for new data increases congestion 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



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

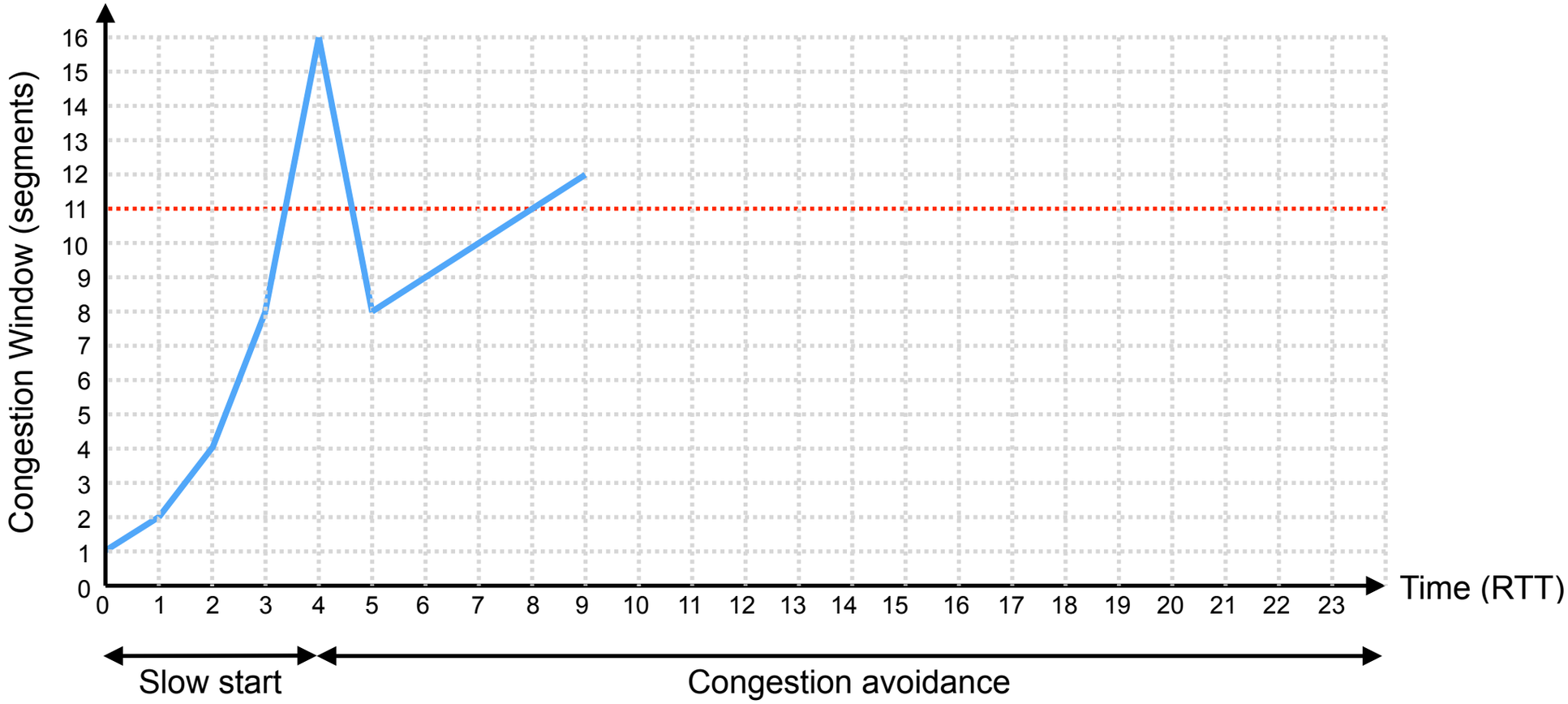
TCP Reno: Congestion Avoidance (1/4)

- 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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Reno: Congestion Avoidance (2/4)

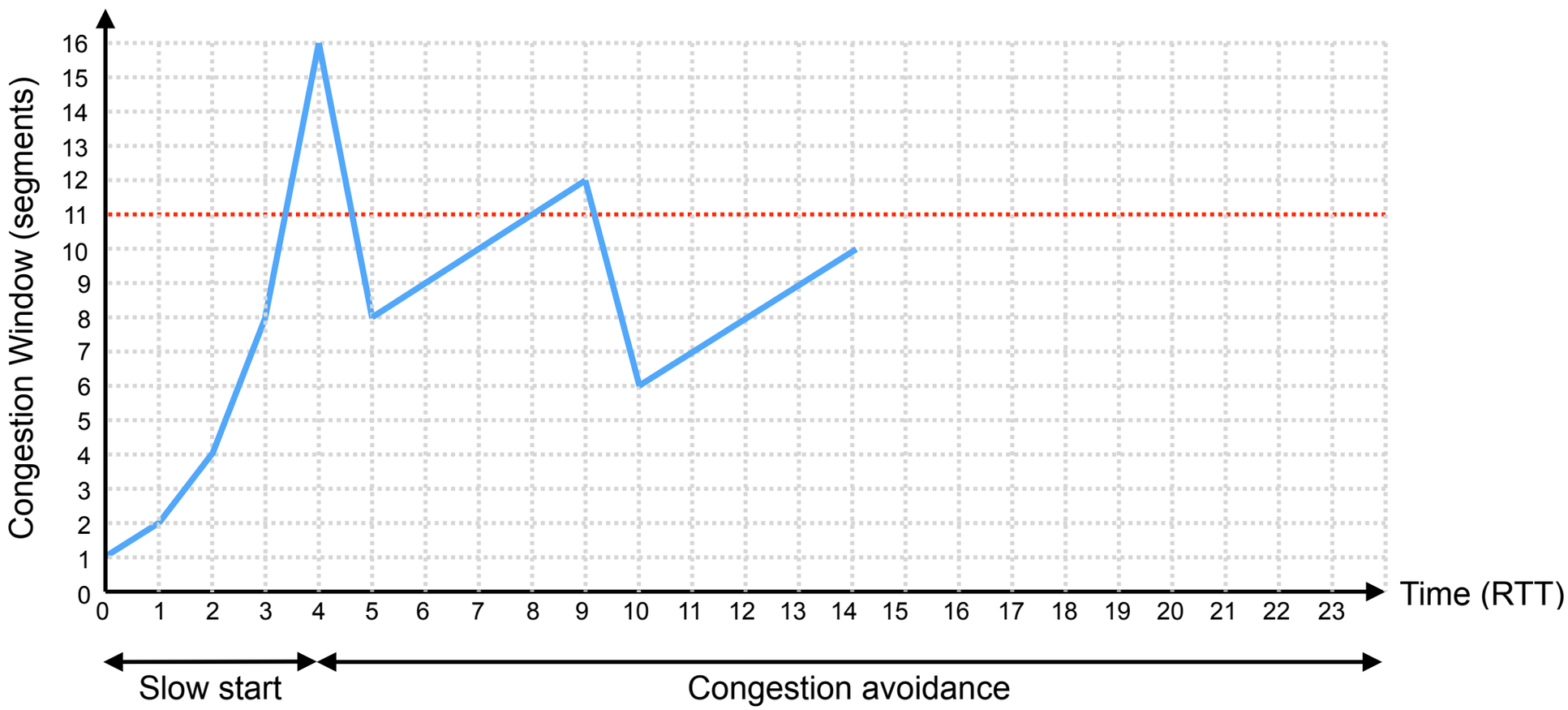
- If a complete window of packets is sent without loss:
  - Increase congestion window by 1 packet per RTT, then send next window worth of packets
  - Slow, linear, additive increase in window: $\mathbf{W_i = W_{i-1} + 1}$



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Reno: Congestion Avoidance (3/4)

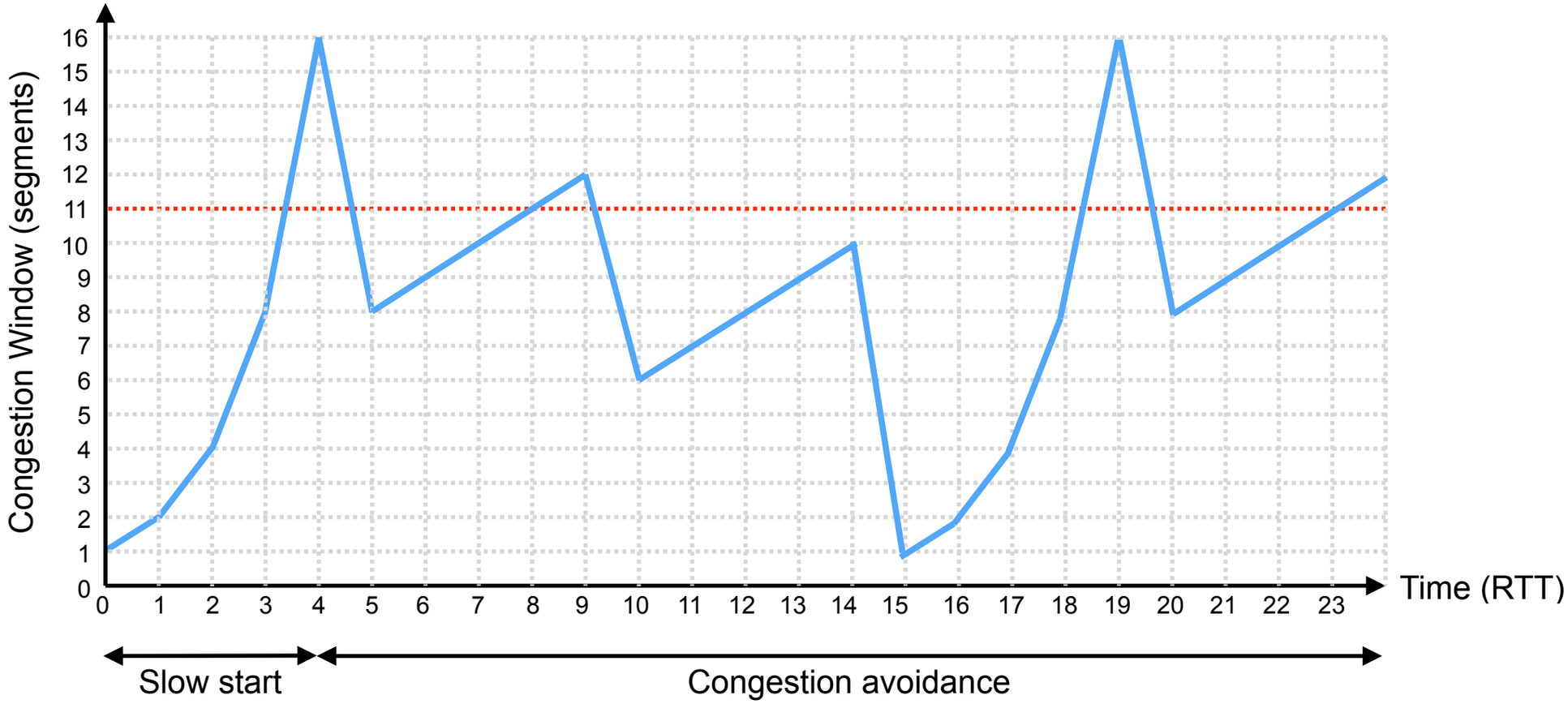
- If a packet is lost and detected via triple duplicate acknowledgement:
  - Transient congestion, but data still being received
  - Multiplicative decrease in window: $\mathbf{W_i = W_{i-1} \times 0.5}$
  - Rapid reduction in window allows congestion to clear quickly, avoids congestion collapse
- Then,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



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

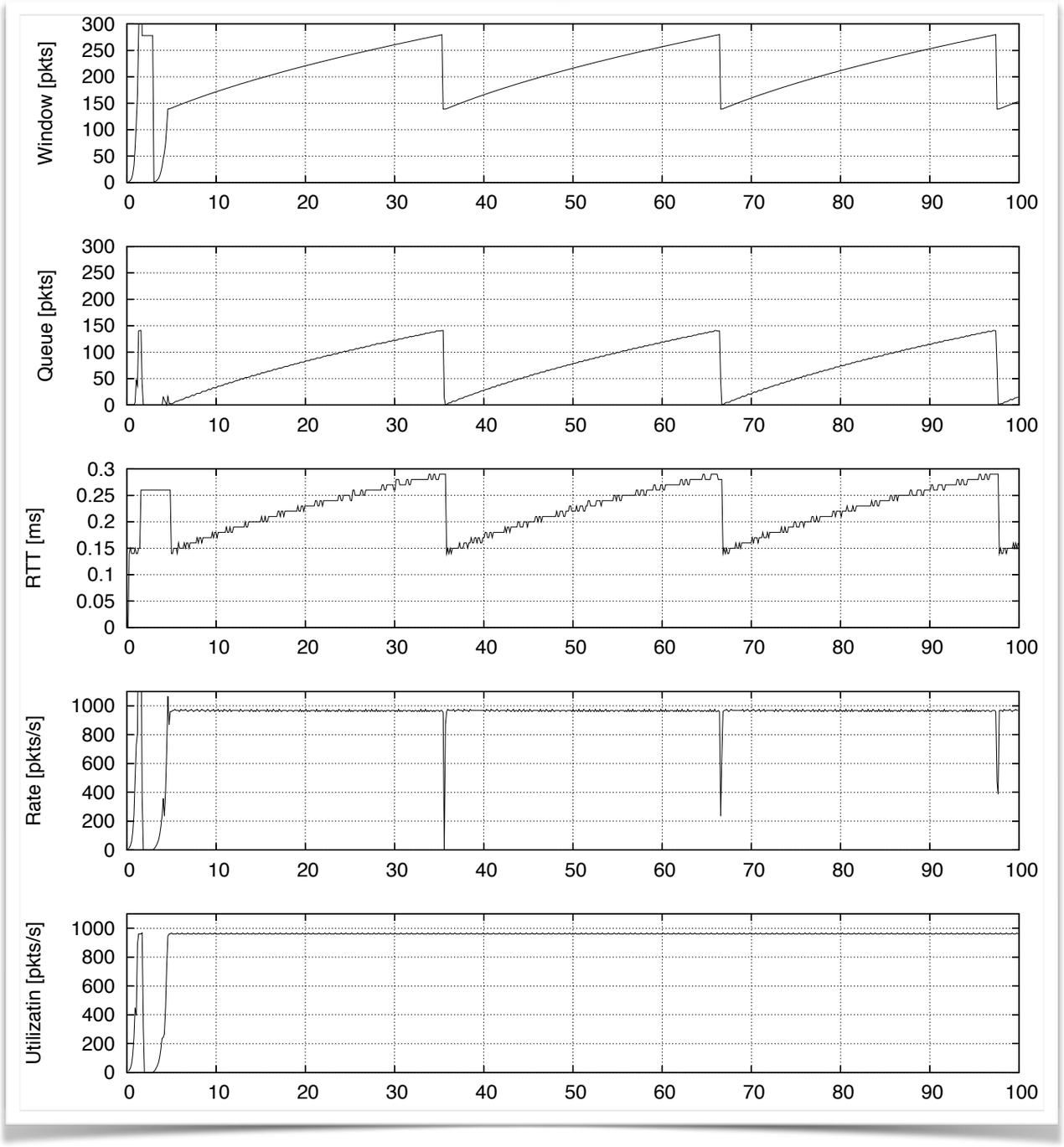
TCP Reno: Congestion Avoidance (4/4)

- 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))

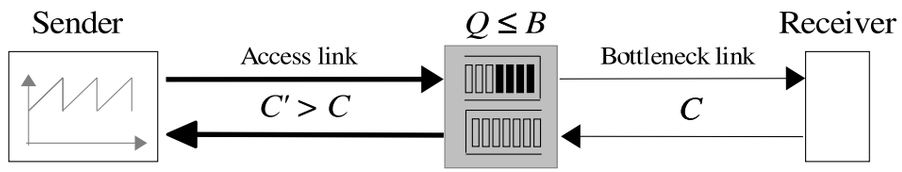


Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Congestion Window Evolution, Buffering, and Throughput



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)



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

CC BY ND

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

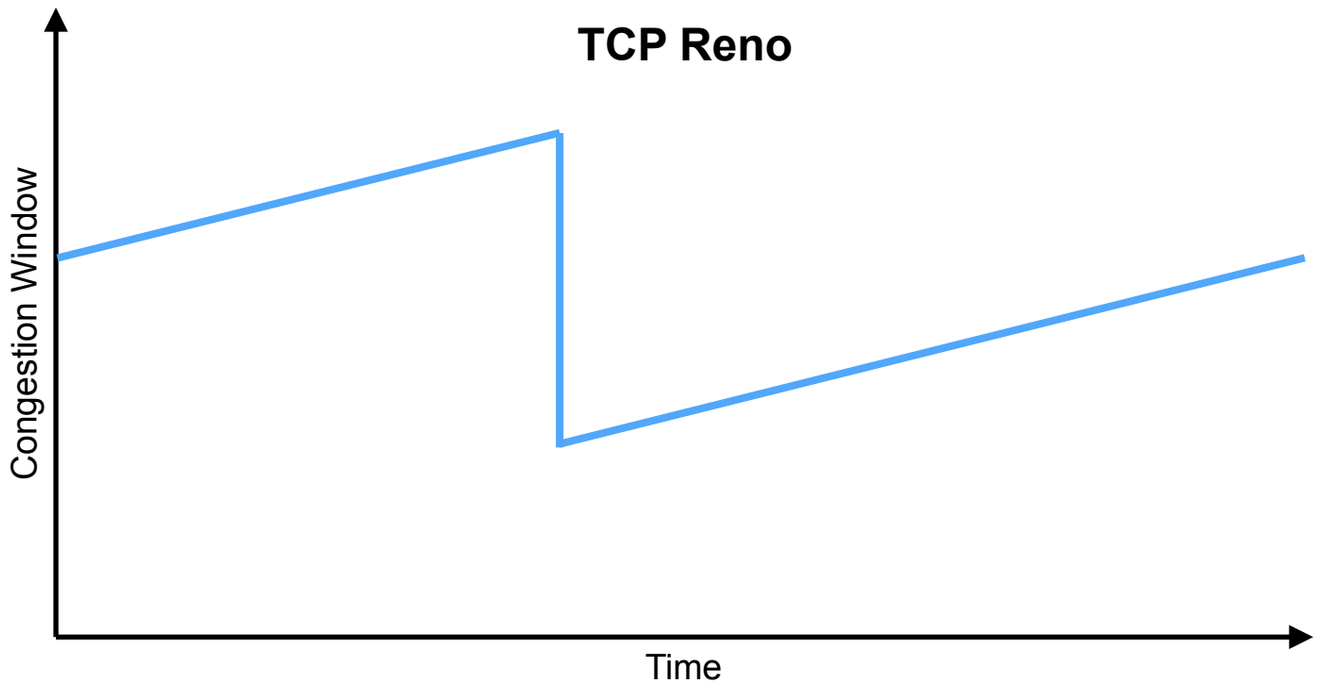
TCP Cubic

- Improving TCP performance on fast, long-distance, networks

BY ND

TCP Performance on Fast Long-distance Networks

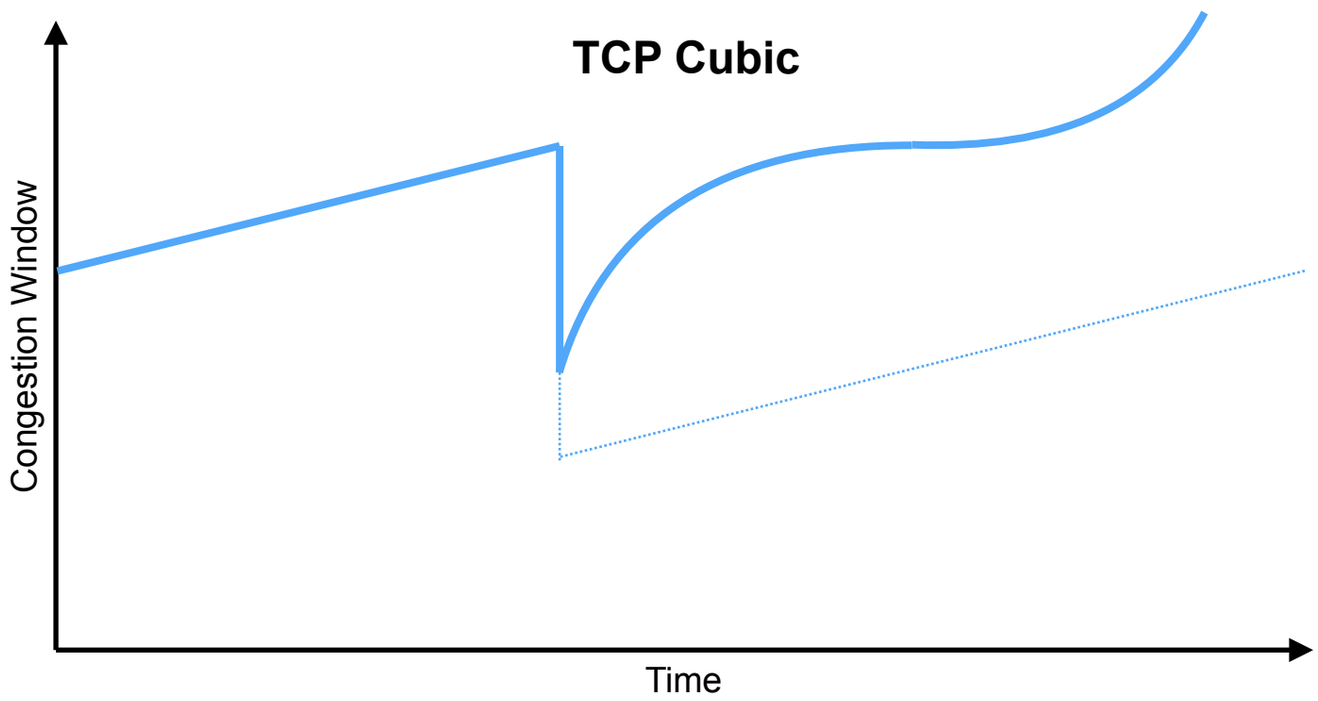
- TCP Reno can perform poorly on **fast long-distance networks**
  - e.g., multi-gigabit per second inter-continental links
  - Path with 10Gbps bandwidth and 100ms RTT requires congestion window ~100,000 packets
  - In congestion avoidance, one packet lost and detected via triple duplicate ACK halves the window, then increase by 1 packet per RTT → 50,000 RTTs to recover sending rate
    - Approximately 1.4 hours with 100ms RTT!

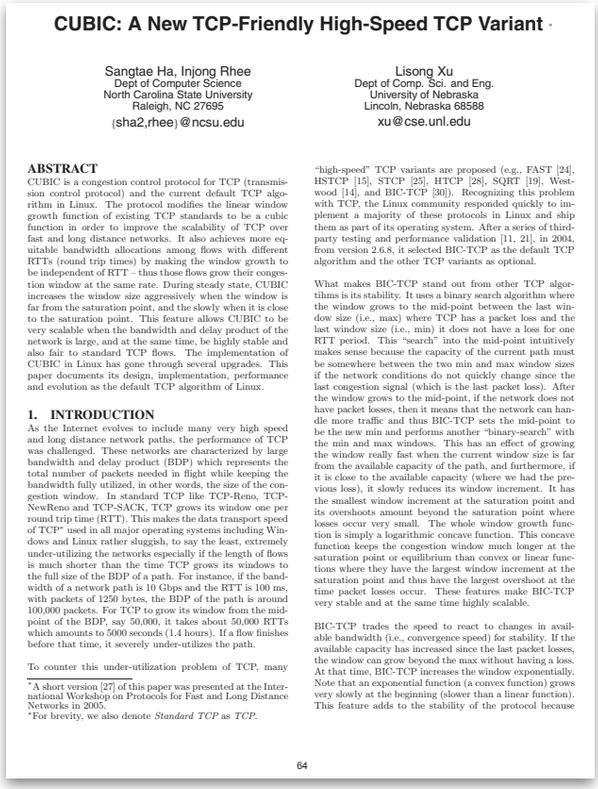


Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Cubic

- **TCP Cubic** changes the congestion control algorithm
  - During congestion avoidance, increases congestion window faster than TCP Reno on fast long-distance networks
  - Rapidly increases congestion window after packet loss
  - Slows rate of increase as window approaches the largest successfully achieved window





CUBIC: A New TCP-Friendly High-Speed TCP Variant *

Sangtae Ha, Injong Rhee
Dept of Computer Science
North Carolina State University
Raleigh, NC 27695
{sha2,rhee}@ncsu.edu

Lisong Xu
Dept of Comp. Sci. and Eng.
University of Nebraska
Lincoln, Nebraska 68588
xu@cse.unl.edu

ABSTRACT
CUBIC is a congestion control protocol for TCP (transmission control protocol) and the current default TCP algorithm in Linux. The protocol modifies the linear window growth function of existing TCP standards to be a cubic function in order to improve the scalability of TCP over fast and long distance networks. It also achieves more equitable bandwidth allocations among flows with different RTTs (round trip times) by making the window growth to be independent of RTT – thus those flows grow their congestion window at the same rate. During steady state, CUBIC increases the window size aggressively when the window is far from the saturation point, and the slowly when it is close to the saturation point. This feature allows CUBIC to be very scalable when the bandwidth and delay product of the network is large, and at the same time, be highly stable and also fair to standard TCP flows. The implementation of CUBIC in Linux has gone through several upgrades. This paper documents its design, implementation, performance and evolution as the default TCP algorithm of Linux.

1. INTRODUCTION
As the Internet evolves to include many very high speed and long distance network paths, the performance of TCP was challenged. These networks are characterized by large bandwidth and delay product (BDP) which represents the total number of packets needed in flight while keeping the bandwidth fully utilized, in other words, the size of the congestion window. In standard TCP like TCP-Reno, TCP-NewReno and TCP-SACK, TCP grows its window one per round trip time (RTT). This makes the data transport speed of TCP* used in all major operating systems including Windows and Linux rather sluggish, to say the least, extremely under-utilizing the networks especially if the length of flows is much shorter than the time TCP grows its windows to the full size of the BDP of a path. For instance, if the bandwidth of a network path is 10 Gbps and the RTT is 100 ms, with packets of 1250 bytes, the BDP of the path is around 100,000 packets. For TCP to grow its window from the mid-point of the BDP, say 50,000, it takes about 50,000 RTTs which amounts to 5000 seconds (1.4 hours). If a flow finishes before that time, it severely under-utilizes the path.

To counter this under-utilization problem of TCP, many

*A short version [27] of this paper was presented at the International Workshop on Protocols for Fast and Long Distance Networks in 2005.
*For brevity, we also denote *Standard TCP* as *TCP*.

"high-speed" TCP variants are proposed (e.g., FAST [24], HSTCP [15], STCP [25], HTCP [28], SQRT [19], Westwood [14], and BIC-TCP [30]). Recognizing this problem with TCP, the Linux community responded quickly to implement a majority of these protocols in Linux and ship them as part of its operating system. After a series of third-party testing and performance validation [11, 21], in 2004, from version 2.6.8, it selected BIC-TCP as the default TCP algorithm and the other TCP variants as optional.

What makes BIC-TCP stand out from other TCP algorithms is its stability. It uses a binary search algorithm where the window grows to the mid-point between the last window size (i.e., max) where TCP has a packet loss and the last window size (i.e., min) it does not have a loss for one RTT period. This "search" into the mid-point intuitively makes sense because the capacity of the current path must be somewhere between the two min and max window sizes if the network conditions do not quickly change since the last congestion signal (which is the last packet loss). After the window grows to the mid-point, if the network does not have packet losses, then it means that the network can handle more traffic and thus BIC-TCP sets the mid-point to be the new min and performs another "binary-search" with the min and max windows. This has an effect of growing the window really fast when the current window size is far from the available capacity of the path, and furthermore, if it is close to the available capacity (where we had the previous loss), it slowly reduces its window increment. It has the smallest window increment at the saturation point and its overshoots amount beyond the saturation point where losses occur very small. The whole window growth function is simply a logarithmic concave function. This concave function keeps the congestion window much longer at the saturation point or equilibrium than convex or linear functions where they have the largest window increment at the saturation point and thus have the largest overshoot at the time packet losses occur. These features make BIC-TCP very stable and at the same time highly scalable.

BIC-TCP trades the speed to react to changes in available bandwidth (i.e., convergence speed) for stability. If the available capacity has increased since the last packet losses, the window can grow beyond the max without having a loss. At that time, BIC-TCP increases the window exponentially. Note that an exponential function (a convex function) grows very slowly at the beginning (slower than a linear function). This feature adds to the stability of the protocol because

64

S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", ACM SIGOPS Operating System Review, Vol. 54, Num. 5, pages 64-74, July 2008 https://dx.doi.org/10.1145/1400097.1400105

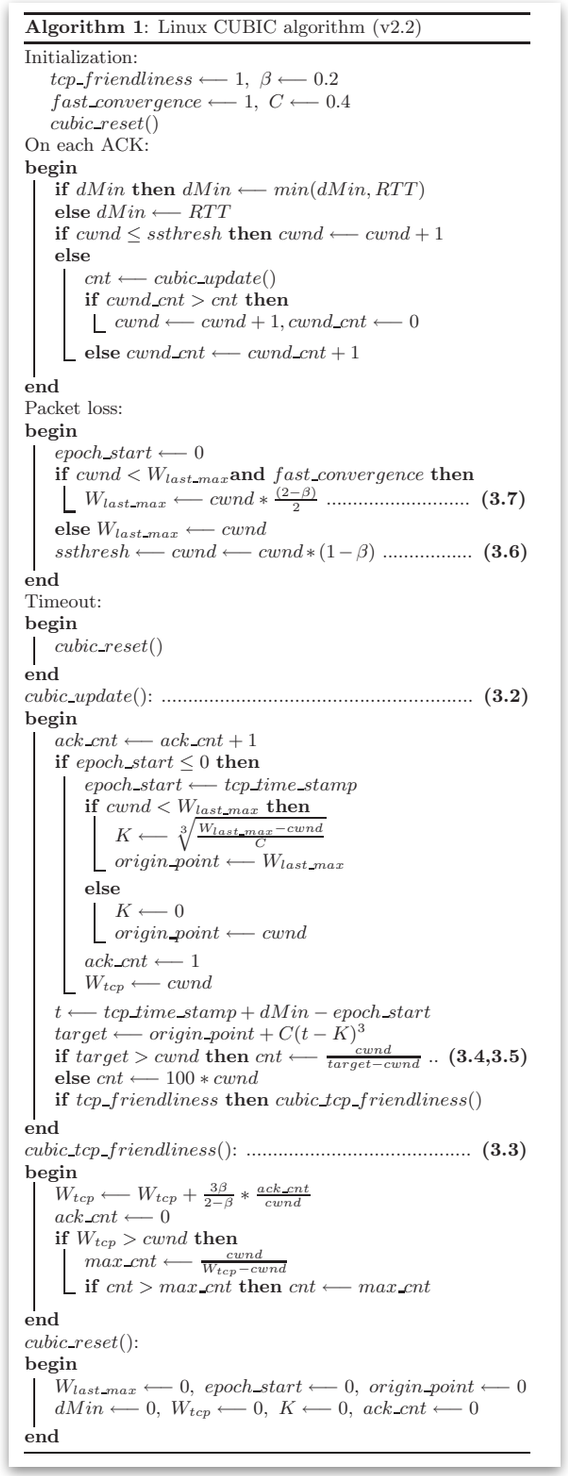
I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", IETF RFC 8312, February 2018.

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Cubic Congestion Control

- **On packet loss** during congestion avoidance, TCP Cubic reduces congestion window: $W_i = W_{i-1} \times 0.7$
  - TCP Reno uses $W_i = W_{i-1} \times 0.5$
  - TCP Cubic is more aggressive

- **After packet loss** during congestion avoidance, TCP Cubic increases the congestion window as: $W_{cubic} = C\,(t - K)^3 + W_{max}$
  - W_{max} is the maximum window size reached before the loss
  - t is the time since the packet loss
  - K is the time it will take to increase the window back to W_{max}, assuming no further packet losses
  - C = 0.4 is a constant that controls fairness to TCP Reno
- **Many** additional details included to ensure fairness with TCP Reno on slower, shorter-RTT, networks



Algorithm 1: Linux CUBIC algorithm (v2.2)

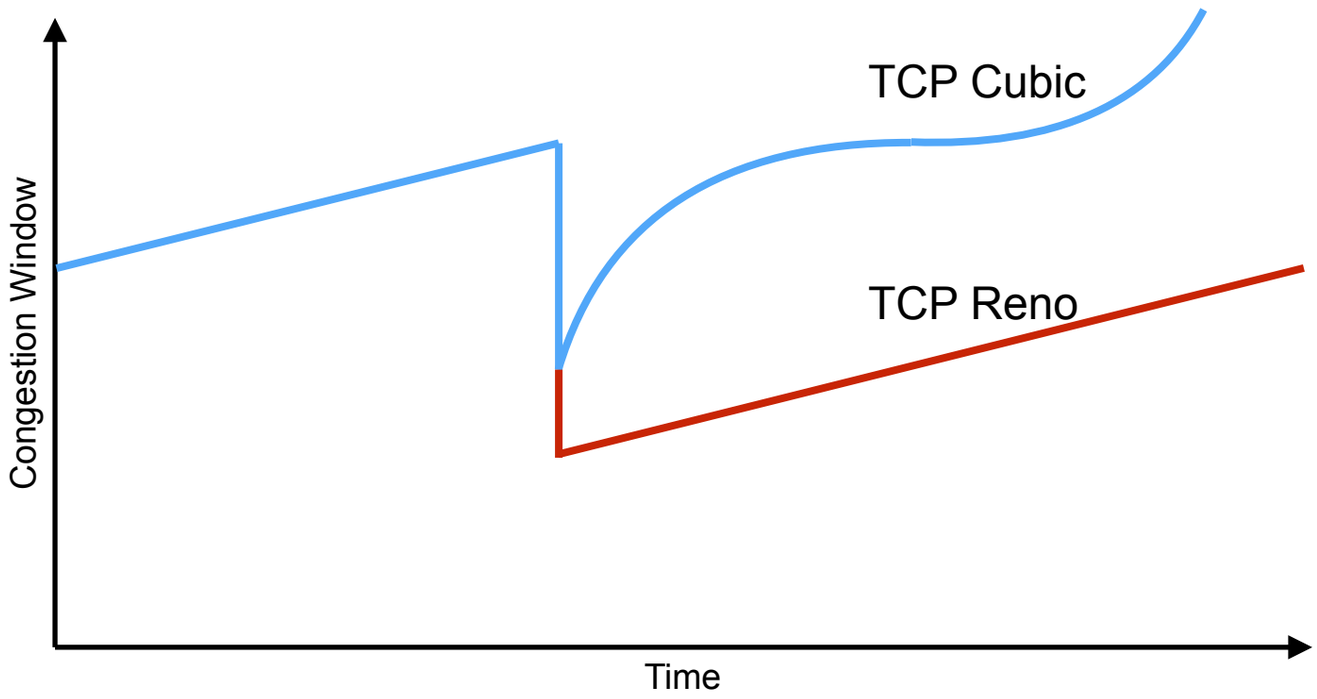
Initialization:
  $tcp_friendliness \longleftarrow 1,\ \beta \longleftarrow 0.2$
  $fast_convergence \longleftarrow 1,\ C \longleftarrow 0.4$
  $cubic_reset()$
On each ACK:
begin
  if $dMin$ **then** $dMin \longleftarrow min(dMin, RTT)$
  else $dMin \longleftarrow RTT$
  if $cwnd \leq ssthresh$ **then** $cwnd \longleftarrow cwnd + 1$
  else
    $cnt \longleftarrow cubic_update()$
    if $cwnd_cnt > cnt$ **then**
      $cwnd \longleftarrow cwnd + 1, cwnd_cnt \longleftarrow 0$
    else $cwnd_cnt \longleftarrow cwnd_cnt + 1$
end
Packet loss:
begin
  $epoch_start \longleftarrow 0$
  if $cwnd < W_{last_max}$ **and** $fast_convergence$ **then**
    $W_{last_max} \longleftarrow cwnd * \frac{(2-\beta)}{2}$ ..... **(3.7)**
  else $W_{last_max} \longleftarrow cwnd$
  $ssthresh \longleftarrow cwnd \longleftarrow cwnd * (1 - \beta)$ ..... **(3.6)**
end
Timeout:
begin
  $cubic_reset()$
end
$cubic_update()$: ..... **(3.2)**
begin
  $ack_cnt \longleftarrow ack_cnt + 1$
  if $epoch_start \leq 0$ **then**
    $epoch_start \longleftarrow tcp_time_stamp$
    if $cwnd < W_{last_max}$ **then**
      $K \longleftarrow \sqrt[3]{\frac{W_{last_max} - cwnd}{C}}$
      $origin_point \longleftarrow W_{last_max}$
    else
      $K \longleftarrow 0$
      $origin_point \longleftarrow cwnd$
    $ack_cnt \longleftarrow 1$
    $W_{tcp} \longleftarrow cwnd$
  $t \longleftarrow tcp_time_stamp + dMin - epoch_start$
  $target \longleftarrow origin_point + C(t - K)^3$
  if $target > cwnd$ **then** $cnt \longleftarrow \frac{cwnd}{target - cwnd}$ .. **(3.4,3.5)**
  else $cnt \longleftarrow 100 * cwnd$
  if $tcp_friendliness$ **then** $cubic_tcp_friendliness()$
end
$cubic_tcp_friendliness()$: ..... **(3.3)**
begin
  $W_{tcp} \longleftarrow W_{tcp} + \frac{3\beta}{2-\beta} * \frac{ack_cnt}{cwnd}$
  $ack_cnt \longleftarrow 0$
  if $W_{tcp} > cwnd$ **then**
    $max_cnt \longleftarrow \frac{cwnd}{W_{tcp} - cwnd}$
    if $cnt > max_cnt$ **then** $cnt \longleftarrow max_cnt$
end
$cubic_reset()$:
begin
  $W_{last_max} \longleftarrow 0,\ epoch_start \longleftarrow 0,\ origin_point \longleftarrow 0$
  $dMin \longleftarrow 0,\ W_{tcp} \longleftarrow 0,\ K \longleftarrow 0,\ ack_cnt \longleftarrow 0$
end

S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", ACM SIGOPS Operating System Review, Vol. 54, Num. 5, pages 64-74, July 2008 https://dx.doi.org/10.1145/1400097.1400105

BY ND

TCP Cubic vs Reno

- TCP Cubic is the default in most modern operating systems
  - **Much more complex** than TCP Reno
    - Core response is relatively straight-forward
    - Much complexity to ensure fairness with TCP Reno in the typical operating regime of TCP Reno
  - **Improves performance** for networks with longer RTT and higher bandwidth
- Both algorithms use packet loss as congestion signal and eventually fill router buffers
  - Trade latency for throughput



QUIC congestion control algorithm is based on TCP Reno for initial version; expected to migrate to some algorithm based on TCP Cubic

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

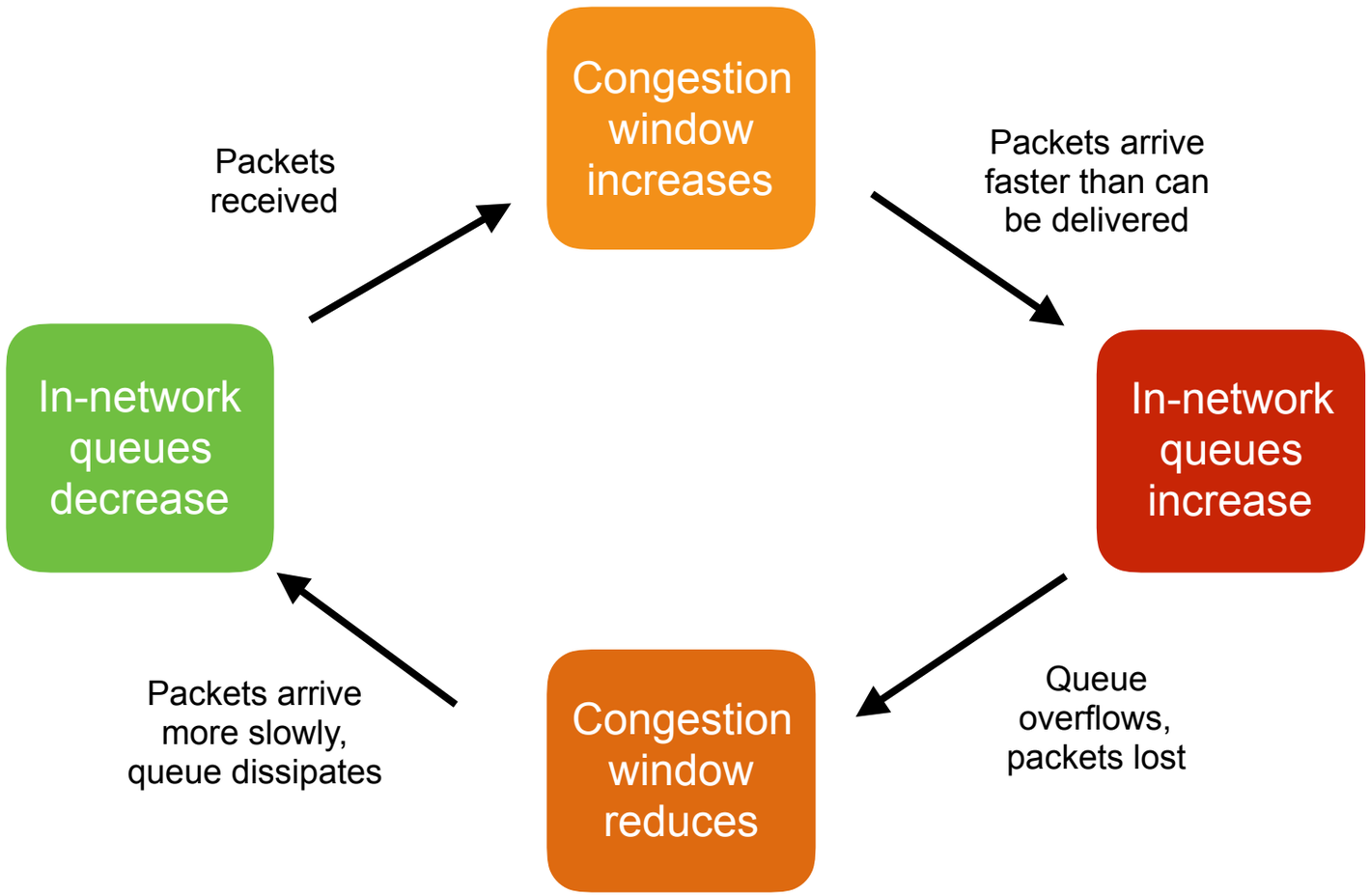
Delay-based Congestion Control

- Traditional TCP causes latency
- TCP Vegas
- TCP BBR

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Impact of TCP on Latency

- TCP Reno and TCP Cubic **both** aim to fill the network



Algorithms use packet loss as congestion signal → increase congestion window until queue overflows and packet is lost

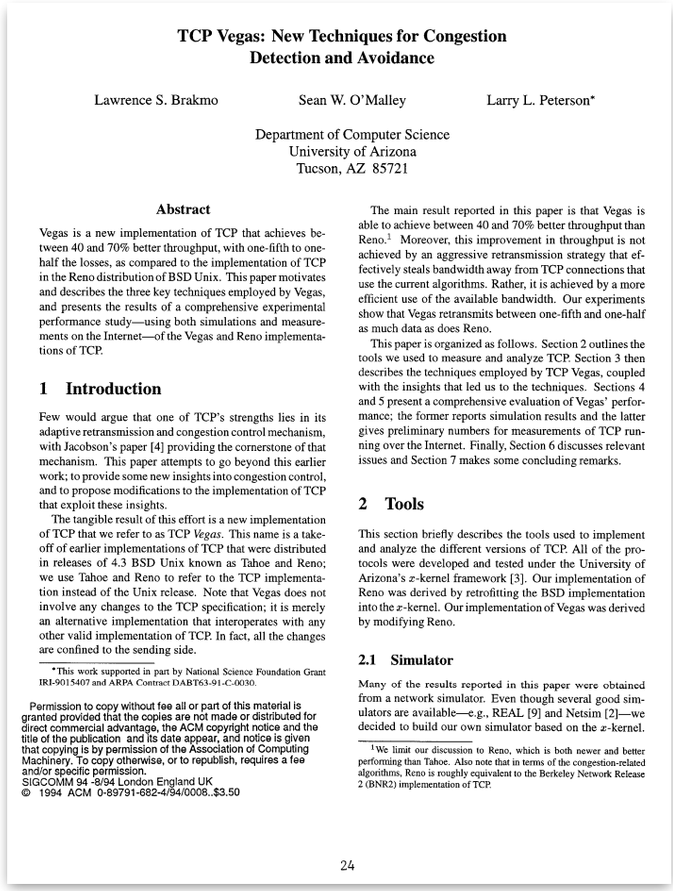
No matter how much big the queue, TCP Reno or Cubic **will** cause it to overflow, if given enough data to send

Packets waiting in queues within network adds latency → increase RTT

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Vegas: Reducing Latency

- Key insight: if sending faster than the network can deliver, packets will be queued
  - TCP Reno and Cubic wait until the queue overflows and packets are lost before slowing down
  - **TCP Vegas** watches for the increase in delay as the queue starts to fill up → slows down before the queue overflows
    - Queues are smaller → lower latency
    - Packets are not lost, so don't need retransmission
- Only affects congestion avoidance; slow start unchanged



TCP Vegas: New Techniques for Congestion Detection and Avoidance

Lawrence S. Brakmo Sean W. O'Malley Larry L. Peterson*

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

Vegas is a new implementation of TCP that achieves between 40 and 70% better throughput, with one-fifth to one-half the losses, as compared to the implementation of TCP in the Reno distribution of BSD Unix. This paper motivates and describes the three key techniques employed by Vegas, and presents the results of a comprehensive experimental performance study—using both simulations and measurements on the Internet—of the Vegas and Reno implementations of TCP.

1 Introduction

Few would argue that one of TCP's strengths lies in its adaptive retransmission and congestion control mechanism, with Jacobson's paper [4] providing the cornerstone of that mechanism. This paper attempts to go beyond this earlier work; to provide some new insights into congestion control, and to propose modifications to the implementation of TCP that exploit these insights.

The tangible result of this effort is a new implementation of TCP that we refer to as TCP *Vegas*. This name is a take-off of earlier implementations of TCP that were distributed in releases of 4.3 BSD Unix known as Tahoe and Reno; we use Tahoe and Reno to refer to the TCP implementation instead of the Unix release. Note that Vegas does not involve any changes to the TCP specification; it is merely an alternative implementation that interoperates with any other valid implementation of TCP. In fact, all the changes are confined to the sending side.

*This work supported in part by National Science Foundation Grant IRI-9015407 and ARPA Contract DABT63-91-C-0030.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGCOMM 94 -8/94 London England UK
© 1994 ACM 0-89791-682-4/94/0008..$3.50

The main result reported in this paper is that Vegas is able to achieve between 40 and 70% better throughput than Reno.[1] Moreover, this improvement in throughput is not achieved by an aggressive retransmission strategy that effectively steals bandwidth away from TCP connections that use the current algorithms. Rather, it is achieved by a more efficient use of the available bandwidth. Our experiments show that Vegas retransmits between one-fifth and one-half as much data as does Reno.

This paper is organized as follows. Section 2 outlines the tools we used to measure and analyze TCP. Section 3 then describes the techniques employed by TCP Vegas, coupled with the insights that led us to the techniques. Sections 4 and 5 present a comprehensive evaluation of Vegas' performance; the former reports simulation results and the latter gives preliminary numbers for measurements of TCP running over the Internet. Finally, Section 6 discusses relevant issues and Section 7 makes some concluding remarks.

2 Tools

This section briefly describes the tools used to implement and analyze the different versions of TCP. All of the protocols were developed and tested under the University of Arizona's *x*-kernel framework [3]. Our implementation of Reno was derived by retrofitting the BSD implementation into the *x*-kernel. Our implementation of Vegas was derived by modifying Reno.

2.1 Simulator

Many of the results reported in this paper were obtained from a network simulator. Even though several good simulators are available—e.g., REAL [9] and Netsim [2]—we decided to build our own simulator based on the *x*-kernel.

[1] We limit our discussion to Reno, which is both newer and better performing than Tahoe. Also note that in terms of the congestion-related algorithms, Reno is roughly equivalent to the Berkeley Network Release 2 (BNR2) implementation of TCP.

L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In Proceedings of the SIGCOMM Conference, pages 24–35, London, UK, August 1994. ACM. https://dx.doi.org/10.1145/190314.190317

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

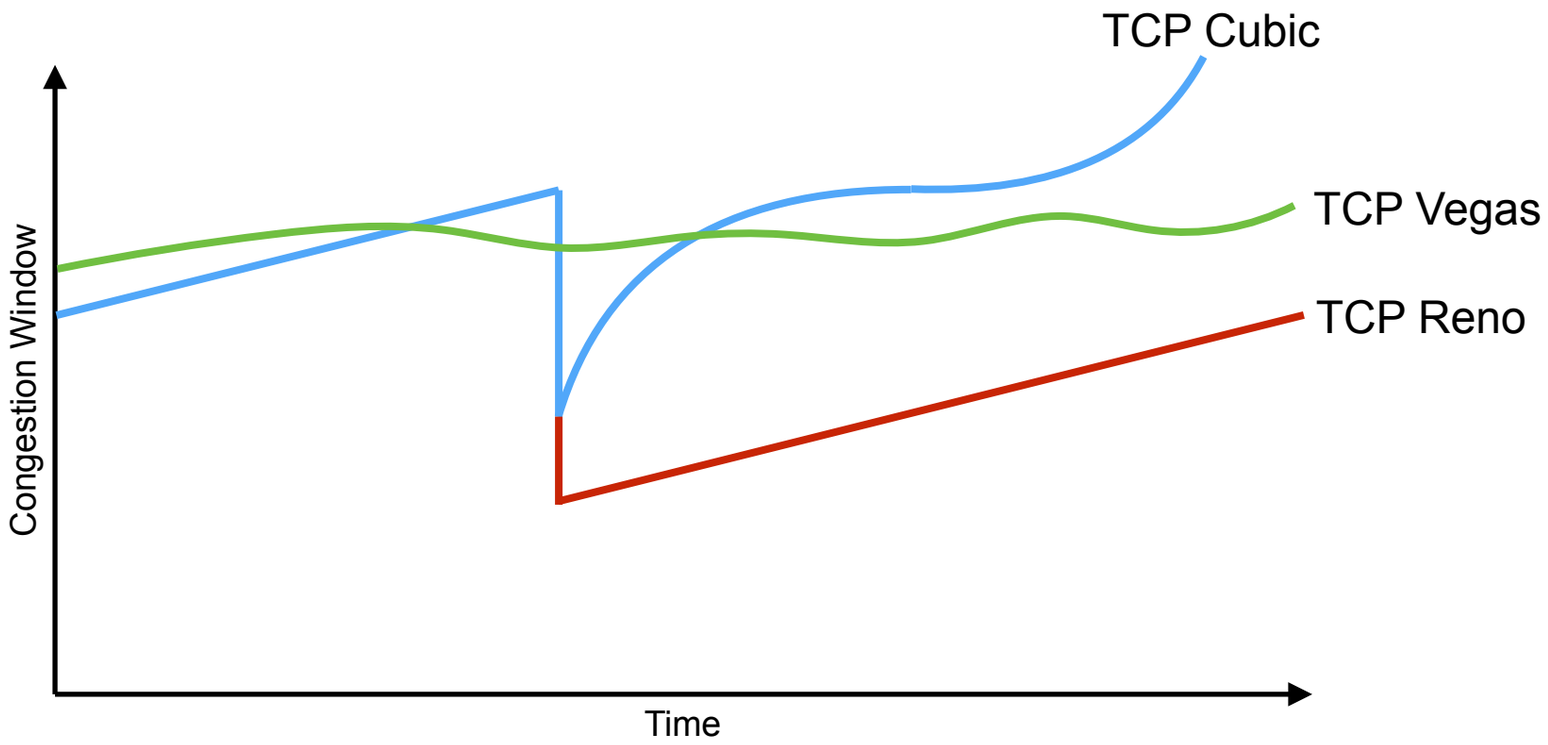
TCP Vegas Congestion Control (1/2)

- Measure **BaseRTT**
  - The smallest time between sending a packet and getting its acknowledgement
  - BaseRTT will be from a packet delivered without much queueing
- Calculate **ExpectedRate** = $W/_{BaseRTT}$
  - The congestion window, W, determines how many packets are sent each RTT
  - If the network can support this sending rate, the complete window should be delivered within the RTT
- Measure **ActualRate**
  - Bytes sent divided by actual RTT, for each packet
  - Measure how fast packets are actually received, based on acknowledgements returned
  - ActualRate ≤ ExpectedRate since packets can't be delivered faster than they're sent

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Vegas Congestion Control (2/2)

- Compare ExpectedRate with ActualRate and adjust window:
  - If ExpectedRate - ActualRate < R_1 then additive increase to window
    - Data arriving at close to expected rate, can likely send faster
  - If ExpectedRate - ActualRate > R_2 then additive decrease to window
    - Data arriving slower than it's being sent, slow down
  - Parameters R_1 and R_2, where $R_1 < R_2$, critical to performance



Additive increase and decrease in size window → smoother changes in sending rate than Reno or Cubic

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Vegas: Limitations

- Delay-based congestion control is a good idea in principle
  - Reduces latency
  - Reduces packet loss
- But loss- and delay-based congestion control don’t cooperate
  - TCP Reno and TCP Cubic aggressively increase queue sizes
  - Increases RTT and reduces ActualRate as measured by TCP Vegas
  - Forces TCP Vegas to slow down; cycle repeats → TCP Vegas sending rate drops to zero over time

- TCP Vegas is not used, because it can’t be deployed alongside TCP Reno or TCP Cubic

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP BBR

- Is it possible to develop a delay-based congestion control algorithm for TCP that can be deployed on a network alongside TCP Reno and TCP Cubic?
- **Maybe** – TCP BBR is a current attempt
  - BBR = Bottleneck Bandwidth and RTT
  - Proposal from Google – measures RTT and bandwidth of the bottleneck link, directly sets congestion window
- Used by YouTube in some cases, but **highly experimental**
  - Recent work by Ranysha Ware and Justine Sherry shows serious fairness problems with TCP BBR v1
    - https://dl.acm.org/doi/10.1145/3355369.3355604
    - https://vimeo.com/showcase/6531379/video/369121357#t=990s
- TCP BBR v2 in development, may solve these problems – this is an active research area



BBR Congestion-Based Congestion Control

MEASURING BOTTLENECK BANDWIDTH AND ROUND-TRIP PROPAGATION TIME

NEAL CARDWELL
YUCHUNG CHENG
C. STEPHEN GUNN
SOHEIL HASSAS YEGANEH
VAN JACOBSON

By all accounts, today's Internet is not moving data as well as it should. Most of the world's cellular users experience delays of seconds to minutes; public Wi-Fi in airports and conference venues is often worse. Physics and climate researchers need to exchange petabytes of data with global collaborators but find their carefully engineered multi-Gbps infrastructure often delivers at only a few Mbps over intercontinental distances.[6]

These problems result from a design choice made when TCP congestion control was created in the 1980s—interpreting packet loss as "congestion."[13] This equivalence was true at the time but was because of technology limitations, not first principles. As NICs (network interface controllers) evolved from Mbps to Gbps and memory chips from KB to GB, the relationship between packet loss and congestion became more tenuous.

Today TCP's loss-based congestion control—even with the current best of breed, CUBIC[11]—is the primary cause of these problems. When bottleneck buffers are large,

acmqueue | september-october 2016 20

N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. ACM Queue, 14(5), September 2016. https://dl.acm.org/doi/10.1145/3012426.3022184

CC BY ND

Explicit Congestion Notification (ECN)

- ECN for TCP and IP

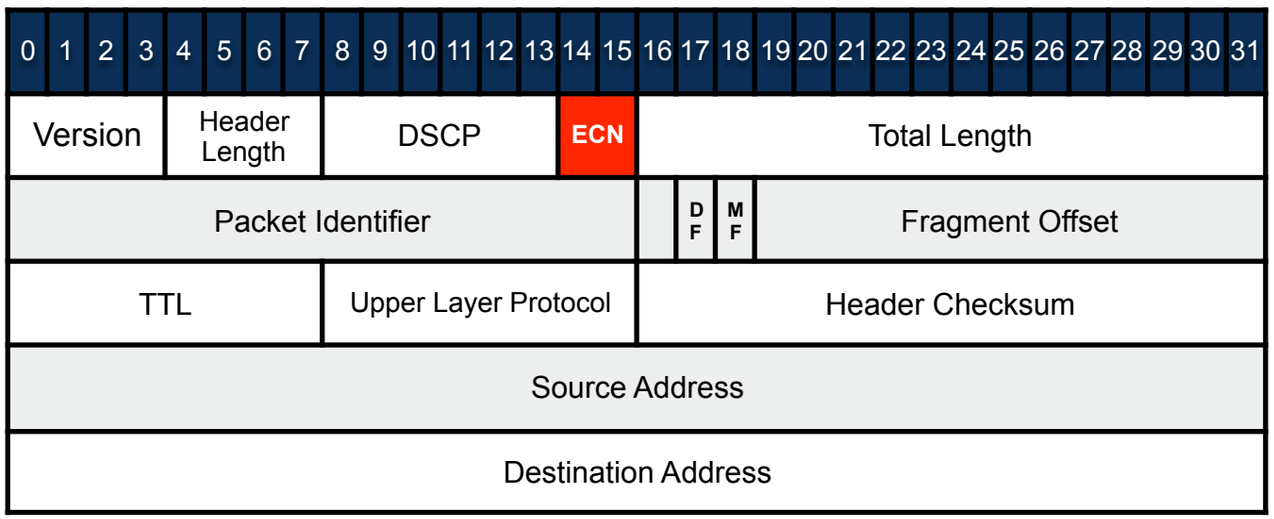
BY ND

Explicit Congestion Notification

- TCP has inferred congestion through measurement
  - TCP Reno and Cubic → packets lost when queues overflow
    - Problematic because it increases delay
    - Problematic because of non-congestive loss on wireless links
  - TCP Vegas → increase in delay as queues start to fill
    - Conceptually a good idea, but difficult to deploy when competing with TCP Reno and Cubic
- **Why not have the network tell TCP congestion is occurring?**
  - **E**xplicit **C**ongestion **N**otification (ECN) field in IP header
  - Tell TCP to slow down if it's overloading the network

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

ECN and IP



00 = Not ECN capable
01 = ECN capable transport
10 = *Reserved*
11 = Congestion Experienced (CE)

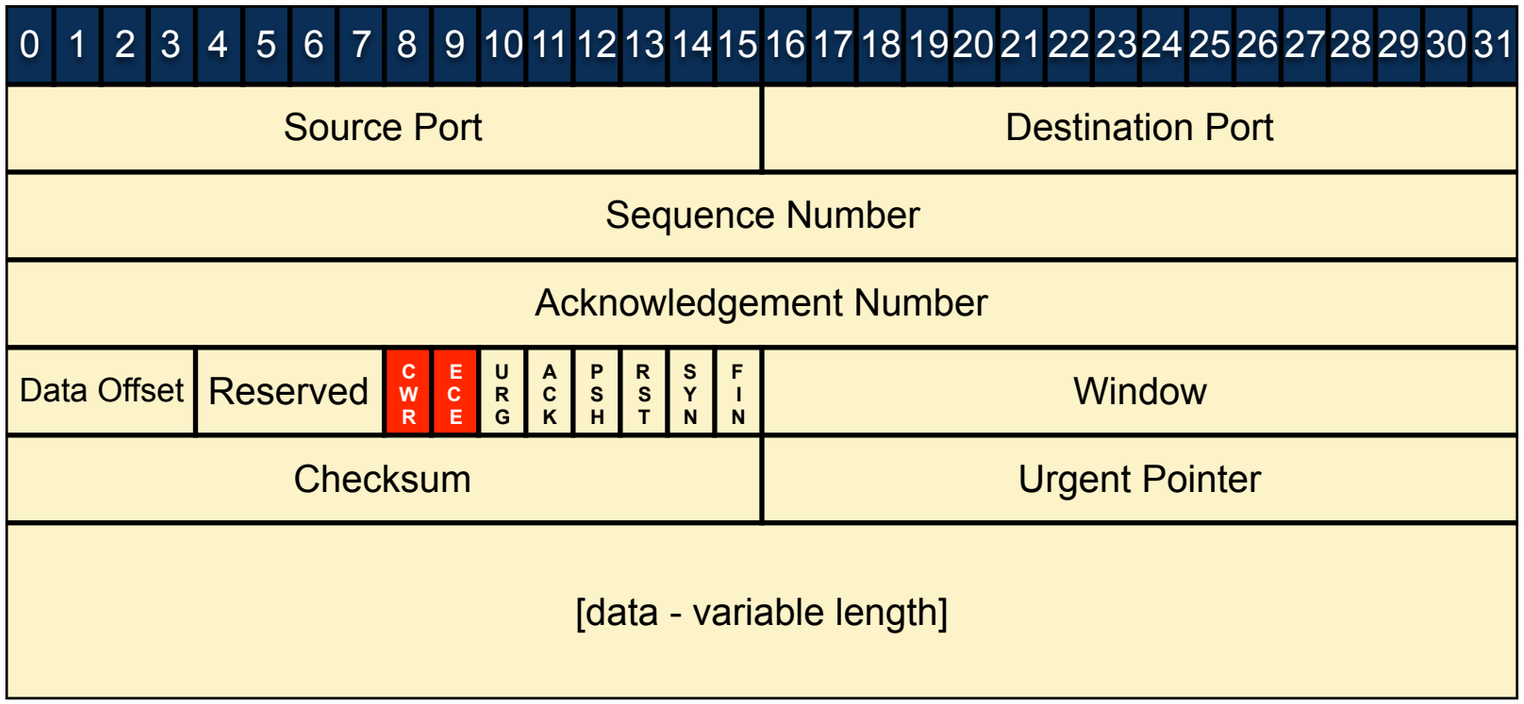
K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", IETF RFC 3168, September 2001. https://datatracker.ietf.org/doc/rfc3168/

- If sender supports ECN, set ECN bits to 01 on transmission
  - If sender doesn't support ECN, set the ECN bits to 00 to disable
- If congestion is occurring, congested **router** sets the ECN bits to 11
  - Indicates that a queue of packets for some link is getting full, but has not yet overflowed – an **early signal of congestion set by the network**
  - If the queue overflows, packets are dropped as usual
- Provides a congestion signal from the network to the receiver

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

ECN and TCP (1/3)

- Receiver may get a TCP segment within an IP packet marked **ECN Congestion Experienced (ECN-CE)**
- ECN Echo (ECE) field in **TCP header** allow it to signal this back to the sender

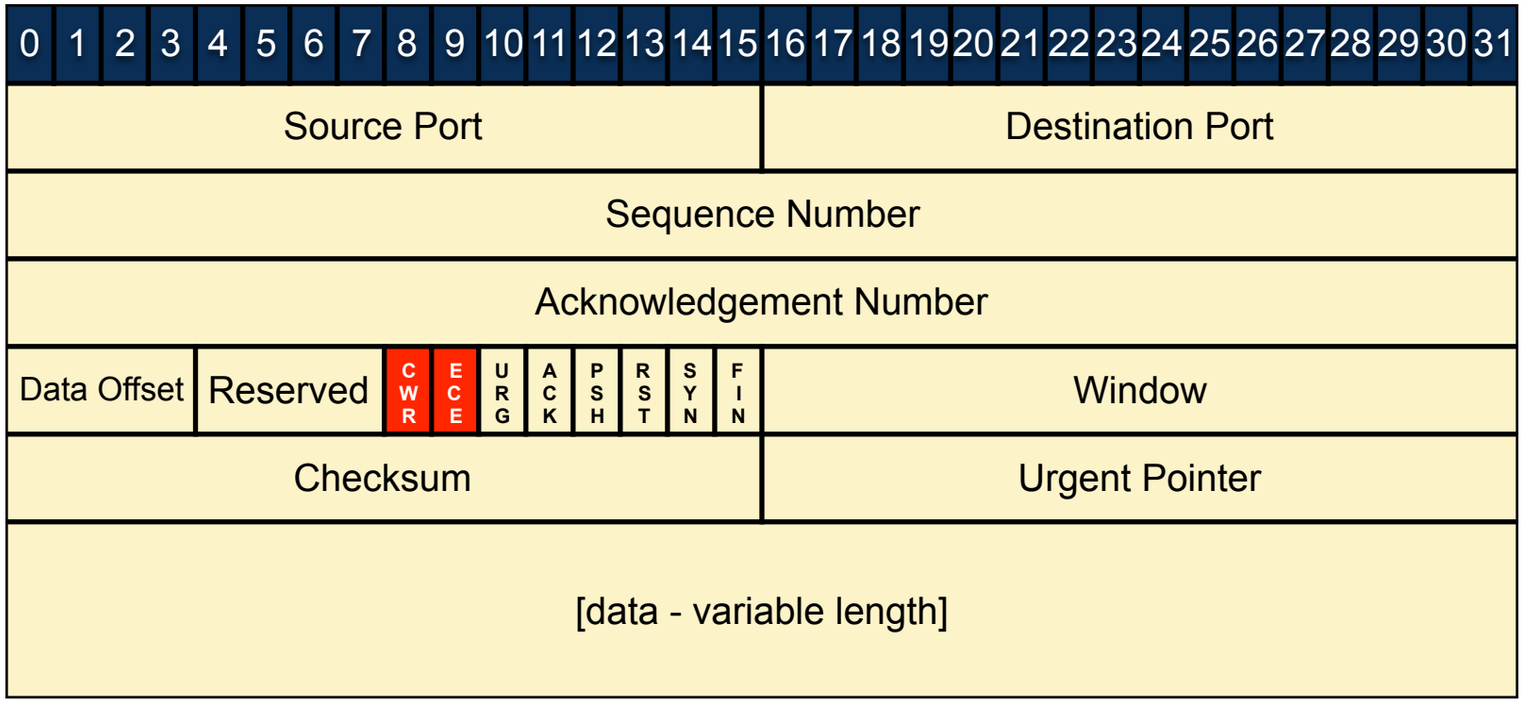


- ECE = 1 on next TCP acknowledgement packet sent by receiver after ECN-CE received
- Tells the TCP sender that congestion has occurred

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

ECN and TCP (2/3)

- Sender reduces congestion window on receiving acknowledgement with ECE = 1 **as if the packet had been lost**
- Sets CWR = 1 in next packet it sends to inform the receiver it has done so



- ECN-CE marked packets arrived at the receiver, so no need to retransmit them
- If packets are lost, TCP sender reacts in the usual way

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

ECN and TCP (3/3)

- ECN lets TCP react to congestion before packet loss occurs
  - Routers signal congestion **before** queues overflow
  - Smaller queues → reduces latency
  - Works independent of choice of TCP congestion control algorithm
- Deployment issues:
  - TCP endpoints need to be updated to support ECN → largely done
    - **Most** endpoints now support ECN and set ECN Capable Transport in IP packets
      - ECN was widely implemented but disabled-by-default for many years, due to concerns that it wouldn't pass through firewalls
      - Apple forced firewalls to be fixed: iOS 9 enabled ECN for 5% of connection; iOS 10 for 50%; iOS 11 and later for all connections
  - Relatively few routers support ECN – endpoints can react to explicit congestion signals, but network is currently unlikely to provide them
    - Starting to change – e.g., recent DOCSIS Cable Modems support ECN
    - Will take time, but latency benefits as ECN is enabled throughout the network

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Light Speed?

- Impact of propagation delay on latency

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Light Speed?

- Two factors impact latency for data transfer
  - The time packets spend in queues within the network
    - Impact of TCP congestion control algorithm
    - ECN to signal congestion before queues overflow
  - **The time packets spend propagating down the links between routers**
    - Speed at which light propagates through an optical fibre
    - Speed at which electrical signals propagate down a wire
    - Speed at which radio signals propagate through the air

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Reducing Propagation Delay (1/3)

- Physically shorter links have lower propagation delay
  - Significant reduction in latency by laying new cables that follow great circle routes between popular destinations
  - e.g., ocean floor cable from Japan to Europe via the north pole, rather than Pacific Ocean – US – Atlantic Ocean



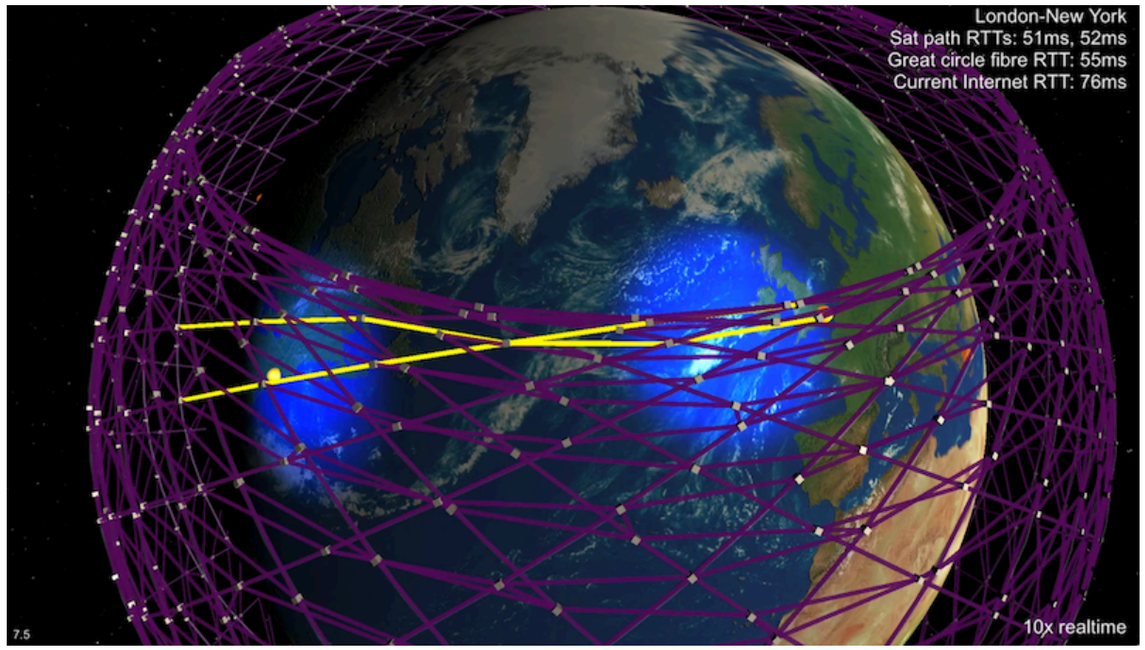
https://www.scientificamerican.com/article/an-internet-cable-will-soon-cross-the-arctic-circle/

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Reducing Propagation Delay (2/3)



Source: Wikipedia



Source: Mark Handley

- Signals propagate at the speed of light **in the transmission medium**
  - Speed of light in optical fibre ~200,000 km/s
  - Speed of light in vacuum 299,792 km/s
- SpaceX Starlink deploying ~4,000 low earth orbit communications satellites
  - ~47% reduction in latency since signals sent through vacuum rather than optical fibre
  - With careful routing, satellite path can follow close to the most direct great circle route
  - http://nrg.cs.ucl.ac.uk/mjh/starlink/

CC BY ND

Reducing Propagation Delay (2/3)

- What application cares so much about reducing latency to spend $billions launching >4,000 satellites? Or new undersea cables?
  - High frequency share trading

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Lowering Latency

- TCP Reno and TCP Cubic give high bandwidth transmission – but cause high queueing delays
- New congestion control algorithms might reduce queueing delays
- ECN will reduce queueing delays
- Significant investment in reducing propagation delays

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow