TCP Adaptation Framework in Data Centers

by

Monia Ghabadi

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2013 by Monia Ghabadi
Abstract

TCP Adaptation Framework in Data Centers

Monia Ghobadi
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2013

Congestion control has been extensively studied for many years. Today, the Transmission Control Protocol (TCP) is used in a wide range of networks (LAN, WAN, data center, campus network, enterprise network, etc.) as the de facto congestion control mechanism. Despite its common usage, TCP operates in these networks with little knowledge of the underlying network or traffic characteristics. As a result, it is doomed to continuously increase or decrease its congestion window size in order to handle changes in the network or traffic conditions. Thus, TCP frequently overshoots or undershoots the ideal rate making it a “Jack of all trades, master of none” congestion control protocol.

In light of the emerging popularity of centrally controlled Software-Defined Networks (SDNs), we ask whether we can take advantage of the information available at the central controller to improve TCP. Specifically, in this thesis, we examine the design and implementation of OpenTCP, a dynamic and programmable TCP adaptation framework for SDN-enabled data centers. OpenTCP gathers global information about the status of the network and traffic conditions through the SDN controller, and uses this information to adapt TCP. OpenTCP periodically sends updates to end-hosts which, in turn, update their behaviour using a simple kernel module.

In this thesis, we first present two real-world TCP adaptation experiments in depth: (1) using TCP pacing in inter-data center communications with shallow buffers, and (2) using Trickle to rate limit TCP video streaming. We explain the design, implementation,
limitation, and benefits of each TCP adaptation to highlight the potential power of having a TCP adaptation framework in today’s networks. We then discuss the architectural design of OpenTCP, as well as its implementation and deployment at SciNet, Canada’s largest supercomputer center. Furthermore, we study use-cases of OpenTCP using the ns-2 network simulator. We conclude that OpenTCP-based congestion control simplifies the process of adapting TCP to network conditions, leads to improvements in TCP performance, and is practical in real-world settings.
Dedication

To my mother Soori, to whom I owe everything.

تقلیم بهادرم سوری، به کسی که بسیار از اوست.
Acknowledgements

First and foremost, I would like to thank my advisor Yashar Ganjali for his support during my PhD and for guiding me through difficult times. I would also like to thank my committee members, Peter Marbach, Eyal de Lara, Angela Demke Brown, and Anees Shaikh for their valuable feedback. This thesis has benefited from collaboration with Google Inc., especially Yu-chung Cheng, Nandita Dukkipati, Ankur Jain, Glen Anderson, and Jerry Chu. Thank you for training me and giving me the industry experience I needed. I would also like to thank Chris Loken, Daniel Gruner, and Ching-Hsing Yu for help and support during my experiments at SciNet. Thanks to Elizabeth Thompson for diligently editing the final version of this thesis.

I’d like to express my deepest gratitude to my sister Kimia Ghobadi, my life-long best friend and supporter. Special thanks to Edy Garfinkiel for making the final year of my PhD memorable. I also would like to thank John Wilkes for sharing many amazing moments with me. Last but not least, thanks to my brothers Reza and Armin Ghobadi. The cost of my studies abroad was the geographical distance between us. Thanks for bringing joy into my life and making this distance bearable. This thesis is dedicated to my amazing mother Soori Sadeghi. I am who I am because of her.
Contents

1 Introduction 1
   1.1 Scientia Potentia Est (Knowledge Is Power) 4
   1.2 Contributions and Impact 6

2 Related Work 8
   2.1 Congestion Control Mechanisms 9
      2.1.1 End-to-End Congestion Control 10
      2.1.2 Active Queue Management 14
      2.1.3 Hybrid of End-to-End and AQM Congestion Control 17
   2.2 Data Center Traffic 19
      2.2.1 Communications in Data Centers 20
      2.2.2 Workload Characterization 22
   2.3 Software-Defined Networks 24

3 Trickle: Adapting TCP to Rate Limit Video Streaming 28
   3.1 Motivation 28
   3.2 YouTube Video Streaming 30
   3.3 Trickle 32
      3.3.1 The Problem: Bursty Losses 32
      3.3.2 Basic Ideas 33
      3.3.3 Challenges 34
3.3.4 Design ......................................................... 34
3.4 Experiment .................................................. 35
  3.4.1 Methodology ............................................. 35
  3.4.2 Validation ............................................... 37
3.5 Analysis ...................................................... 38
  3.5.1 Packet Losses ........................................... 39
  3.5.2 Burst Size ................................................ 41
  3.5.3 Queueing Delay ......................................... 43
  3.5.4 Rebuffering .............................................. 46
3.6 Discussion ................................................... 48

4 TCP Pacing in Data Center Networks  50
  4.1 The Tortoise And The Hare: Why Bother Pacing? ............... 51
  4.2 Data Center Traffic ......................................... 53
  4.3 Modeling the Effectiveness of Pacing .......................... 55
  4.4 Experiments ................................................ 58
    4.4.1 Functional Test ........................................ 59
    4.4.2 Setup of Experiments .................................. 60
    4.4.3 Base-Case Experiment ................................ 61
    4.4.4 Multiple Flows ....................................... 63
    4.4.5 Drop Synchronization .................................. 68
  4.5 Conclusions and Future Trends for TCP Pacing ................. 71

5 OpenTCP: TCP Adaptation in Data Center Networks  73
  5.1 OpenTCP Design Principles ................................ 73
  5.2 OpenTCP Architecture ..................................... 75
  5.3 OpenTCP Operation ......................................... 76
    5.3.1 Data Collection ....................................... 77
6 OpenTCP Evaluation

6.1 NS-2 Simulations

6.1.1 Oracle’s Estimation: Constant Bit Rate Flows

6.1.2 Oracle’s Estimation: TCP flows

6.1.3 Adaptable TCP Knobs

6.1.4 Custom Flow Priority

6.1.5 Dynamic Traffic Pattern

6.1.5.1 Opportunity: Increase $\text{init\_cwnd}$ for Mice Flows

6.1.5.2 Critical: Max-Min Fairness for Elephant Flows

6.1.5.3 Putting It Together: Dynamic Traffic Pattern

6.1.6 Using TCP Pacing to Suppress Burstiness

6.1.7 Effect of Oracle’s Time Scale

6.2 Real World Deployment

6.2.1 Traffic Characterization

6.2.2 Experiment Methodology

6.2.3 Experiment Results

6.2.4 Final Remarks

7 Conclusions and Future Work

Bibliography
List of Tables

3.1 Network statistics in DC1 and DC2 binned by each experiment group.

\[ RTT_{\text{start}} \] is the smoothed RTT (srtt) at the end of the startup phase. \ldots 37

3.2 The retransmission rate improvement bucketed by user bandwidth \ldots 41

3.3 A comparison of rebuffering frequency and rebuffering chance. \ldots \ldots 47

6.1 UDP CBR rates and start times. \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots 88

6.2 Per-switch traffic matrix as measured by the Oracle at time \( t = 30 \). \ldots 90

6.3 The approximate OpenTCP overhead for a network with \( \sim 4000 \) nodes

and \( \sim 100 \) switches for different Oracle refresh cycles. \ldots \ldots \ldots \ldots 114
List of Figures

1.1 A core link’s utilization over time. ........................................... 2
1.2 The effect of number of flows on paced TCP’s performance. ........... 3
1.3 OpenTCP uses network state and traffic statistics to adapt TCP to a network’s spatial and temporal properties. ....................... 5

2.1 TCP congestion control mechanisms: Slow start and Additive Increase Multiplicative Decrease (AIMD). ........................................... 12
2.2 Dropping Probability of RED AQM. ........................................... 15
2.3 Common data center interconnect topology. Host to switch links are GigE and links between switches are 10 GigE. ....................... 19
2.4 The partition/aggregate design pattern. ................................... 21
2.5 OpenFlow switch architecture. .................................................. 26

3.1 Time vs. sequence of bytes graph for a sample YouTube video with RTT 20ms. ................................................................. 30
3.2 Time vs. sequence of bytes graph for a YouTube video with RTT 30ms using Trickle. ............................................................. 32
3.3 CDF of goodput in throttling phase in DC1. The x-axis is in log scale. . 38
3.4 CDF of goodput in throttling phase in DC2. The x-axis is in log scale. . 39
3.5 CDF of retransmission rate in the throttling phase in DC1. .............. 40
3.6 CDF of retransmission rate in the throttling phase in DC2. .............. 40
3.7 CDF of burst size in DC1. ......................................................... 42

3.8 Time vs. sequence of bytes graph of a flow in baseline1 group. The black arrows are data bursts and the red dots are retransmits. After it experiences losses, the 64kB application writes are fragmented into smaller bursts. .............................................................. 44

3.9 Frequency of losing at least one segment as a function of burst size in DC1. 45

3.10 CDF of the smoothed RTT (srtt) samples in DC1. The average srtt of connections in Trickle is 28% and 10% lower than the connections in the baselines and shrunk-block, respectively. .................................. 46

3.11 CDF of the smoothed RTT (srtt) samples in DC2. The average srtt of connections in Trickle is 7% and 1% lower than the connections in the baselines and shrunk-block, respectively. ............................ 47

4.1 Functional testing of TBQ pacing implementation. (a) Time-sequence graph of 1 MB transfers. (b) CDF of packet inter-transmission times. 60

4.2 Experiment’s test-bed topology. .................................................. 61

4.3 Base-case (i): Under-subscribed bottleneck. (a) Bottleneck link utilization. (b) CDF of flow completion times. .................................................. 62

4.4 Base-case (ii): Over-subscribed bottleneck. (a) Bottleneck link utilization. (b) CDF of flow completion times. .............................. 64

4.5 Studying the effect of number of flows sharing the bottleneck link when the buffer size is 1.7% of BDP (64 KB) and the lower bound on inflection point (N*) is 58 flows. (a) Bottleneck link utilization. (b) Bottleneck link drop rate. (c) Average RPC completion times. (d) 99th percentile RPC completion times. ........................................ 65
4.6 Studying the effect of number of flows sharing the bottleneck link when
the buffer size is 3.4% of BDP (128 KB) and the lower bound on inflection
point ($N^*$) is 29 flows. (a) Bottleneck link utilization. (b) Bottleneck link
drop rate. (c) Average RPC completion times. (d) $99^{th}$ percentile RPC
completion times.

4.7 Studying the effect of number of flows sharing the bottleneck link when
the buffer size is 6.8% of BDP (256 KB) and the lower bound on inflection
point ($N^*$) is 14 flows. (a) Bottleneck link utilization. (b) Bottleneck link
drop rate. (c) Average RPC completion times. (d) $99^{th}$ percentile RPC
completion times.

4.8 Studying the accuracy of $N^*$ lower bound on the PoI when the buffer size
varies. (a) Bottleneck link utilization. (b) Bottleneck link drop rate. (c)
Average RPC completion times. (d) $99^{th}$ percentile RPC completion times.

4.9 (a) Bottleneck link’s drop rate. (b) The clustering probability matrix for 25
non-paced flows. (c) The clustering probability matrix for 25 paced flows.
The color of element $(i,j)$ corresponds to the probability that packets from
flow $i$ are followed by packets from flow $j$.

4.10 CDF of flows that are affected by a drop event reported by the NetFPGA
router. (a) Total number of flows is 48. (b) Total number of flows is 96
flows. (c) Total number of flows is 384.

5.1 The Oracle, SDN-enabled switches, and Congestion Control Agents are
the three components of OpenTCP.

5.2 The Oracle collects network state, and statistics through the SDN con-
troller and switches. Combining those with the CCP defined by the op-
erator, the Oracle generates CUEs which are sent to CCAs sitting on
end-hosts. CCAs will adapt TCP based on the CUEs.
5.3 Sample CCPs and CUEs. (a) Here the goal is to minimize flow completion
times by modifying the initial congestion window size. (b) This CCP
switches from the default TCP variant (TCP Reno) to CUBIC as long as
all links have a utilization below 80%.

6.1 Simple NS-2 OpenTCP topology.

6.2 Simulations Topology.

6.3 Per flow utilizations at the bottleneck link with UDP flows. Packet by
packet measurement using NS-2 trace.

6.4 Per flow utilizations at the bottleneck link with UDP flows. Oracle’s view
of the traffic with $T = 1$ second.

6.5 Per flow utilizations at the bottleneck link with TCP flows. Packet by
packet measurement using NS-2 trace.

6.6 Per flow utilizations at the bottleneck link with TCP flows. Oracle’s view
of the traffic with $T = 1$ second.

6.7 Per flow utilizations at the bottleneck link with TCP flows. Congestion
control policy: keep $TCP_0$’s utilization above 70%.

6.8 Per flow utilizations at the bottleneck link with TCP flows.

6.9 Simulation set-up.

6.10 Mice Flow case: Comparing bottleneck link’s utilization, congestion win-
dow size, bottleneck link’s queue occupancy, and flow completion times
between OpenTCP and regular TCP.

6.11 Elephant Flow case: Comparing bottleneck link’s utilization, congestion
window size, bottleneck link’s queue occupancy, and flow completion times
between OpenTCP and regular TCP.

6.12 Dynamic Traffic Pattern case: (a) Elephant flow’s utilization, (b) utiliza-
tion of four random mice flows, (c) bottleneck link’s utilization, (d) CDF
of flow completion times.
6.13 (a) Simulation Topology. (b) OpenTCP dynamically enables TCP pacing. 101
6.14 OpenTCP improves the query flows’ completion times by 50%. 101
6.15 The effect of time scale $T$ on flow completion times. 103
6.16 The topology of SciNet the data center network used for deploying and evaluating OpenTCP. 105
6.17 SciNet traffic characterization. The $x$-axis is in log scale. CDF of flow sizes excluding TCP/IP headers (left). CDF of flow completion times (right). 106
6.18 The locality pattern as seen in matrix of log(number of bytes) exchanged between server pairs. 107
6.19 CDF of flow sizes in different benchmarks used in experiments. The $x$-axis is in log scale. 109
6.20 Comparing the CDF of FCT (left) and drop rate (right) of OpenTCP$_1$, OpenTCP$_2$, and regular TCP while running the “all-to-all” IMB benchmark. 110
6.21 Congestion window distribution for OpenTCP$_1$ and unmodified TCP. 111
6.22 Comparing the CDF of FCT (left) and drop rate (right) of OpenTCP$_1$, OpenTCP$_2$, and regular TCP while running the “Flash” benchmark. 113
6.23 99$^{th}$ percentile FCT for different benchmarks. 114
6.24 Fast OpenTCP vs. OpenTCP$_2$. CDF of flow completion times (top), CDF of drop rate (bottom). 115
Chapter 1

Introduction

The Transmission Control Protocol (TCP) [73] is used in a wide range of networks (LAN, WAN, data center, campus network, enterprise network, etc.) as the de facto congestion control mechanism. Measurements reveal that 99.91% of traffic in Microsoft data centers is TCP [16], 10% of the aggregate North America Internet traffic is YouTube over TCP [11], and measurements from 10 major data centres including university, enterprise, and cloud data centers show TCP as the dominant congestion control protocol [28]. TCP is a mature protocol and has been extensively studied over a number of years. Hence, network operators trust TCP as their congestion control mechanism to maximize the bandwidth utilization of their network while keeping the network stable.

Despite, and because of, its common usage, TCP operates in these networks with little knowledge of the underlying network or traffic characteristics. As a result, it is doomed to continuously increase or decrease its congestion window size in order to handle changes in the network or traffic conditions. Thus, TCP frequently overshoots or undershoots the ideal rate, making its behaviour inefficient at times [16, 37, 77, 78, 126].

TCP is designed to operate in a diverse set of networks with different characteristics and traffic conditions. However, limiting TCP to a specific network and taking advantage of the local characteristics of that network can lead to major performance gains.
Figure 1.1: A core link’s utilization over time.

For instance, DCTCP [16] out-performs TCP in data center networks, even though the results might not be applicable in the Internet. With this mindset, one can adjust TCP (the protocol itself and its parameters) to gain better performance in specific networks (e.g. data centers).

Moreover, even focusing on a particular network, the effect of dynamic congestion control adaptation to traffic patterns is not well understood in today’s networks. Such adaptation can potentially lead to major improvements, as it provides another dimension that today’s TCP does not explore.

Example 1 - Dynamic Traffic Patterns: Figure 1.1 depicts aggregate link utilization of a core link in a backbone service provider in North America [27]. We can see that the link utilization is low for a significant period (below 50% for 6–8 hours). A pattern is seen on all the links in this network. In fact, the presented link has the highest utilization and is considered to be the bottleneck in this network.

If the network operator aims at minimizing flow completion times in this network, it makes sense to increase TCP’s initial congestion window size (init_cwnd) when the network is not highly utilized (we focus on internal traffic in this example). Ideally, the
exact value of \( \text{init\_cwnd} \) should be a function of the network-wide state (here, the number of flow initiations in the system) and how aggressively the operator wants the system to behave (congestion control policy). The operator can define a policy like the following: if link utilization is below 50\%, \( \text{init\_cwnd} \) should be increased to 20 segments instead of the default value of four segments. In other words, given the appropriate mechanisms the operator could choose the right value for the initial congestion window.

**Example 2 - Paced TCP:** Consider a data center network with small-buffer switches. The literature shows that TCP is bursty by nature and its bursts put stress on the buffers at network switches [27]. To suppress the burstiness in this network, Paced TCP [139] could be enabled in TCP flows. However, there have been several inconclusive studies on the effectiveness of pacing in mitigating TCP burstiness [12,34,115,128]. As a result, TCP pacing is not used in today’s data centers. As we will show in Chapter 4, paced TCP is beneficial only if the total number of concurrent flows is below a certain threshold (called the Point of Inflection or PoI).

Figure 1.2 compares the utilization of a bottleneck link between paced and non-paced TCP. As long as the number of flows is below a predictable point called \( N^* \), paced TCP
out-performs non-paced TCP. By increasing the number of flows the effectiveness of pacing will diminish, so much so that after the PoI, non-paced TCP out-performs paced TCP. In a large data center with thousands to millions of concurrent flows, it is almost impossible to guarantee that the number of active flows will always be less than any given threshold. Therefore, providers are reluctant to employ pacing despite its proven effectiveness. However, if we have a system that can measure the number of active flows, we can dynamically switch between paced TCP and regular TCP to ensure the system is always at its peak performance.

1.1 Scientia Potentia Est (Knowledge Is Power)

Adapting TCP requires meticulous consideration of network characteristics and traffic patterns. The fact that TCP relies solely on end-to-end measurements of packet loss or delay as the only sources of feedback from the network, means that TCP, at best, has a limited view of the network state (e.g. the trajectory of available bandwidth, congested links, network topology, and traffic). Thus, the following question arises:

Can we build a system that observes the state and dynamics of a computer network and adapts TCP’s behaviour accordingly?

If the answer to this question is positive, this will simplify tuning different TCP parameters (Example 1). It could also facilitate dynamically changing the protocol itself (Example 2). Currently, network operators use various ad-hoc solutions, as temporary adjustments of TCP to fit their network and traffic [77]. These manual tweaks open the way for misconfiguration, make debugging and troubleshooting difficult, and can result in substantial operational overhead. Moreover, making any changes to the underlying assumptions about the network or traffic requires rethinking the impact of various parameters and can result in ongoing efforts to manually adjust TCP because any proposed change should work under all conditions. Having a system that measures the state and
Chapter 1. Introduction

Figure 1.3: OpenTCP uses network state and traffic statistics to adapt TCP to a network’s spatial and temporal properties.

dynamics of the network and adapts TCP’s behaviour accordingly can address these problems.

This thesis takes the first step in addressing the need for a systematic way of adapting TCP to network and traffic conditions. We propose OpenTCP as a system for dynamic adaptation of TCP based on network and traffic conditions in Software-Defined Networks (SDNs) [98]. Figure 1.3 provides a schematic view of how OpenTCP works. OpenTCP collects data on the underlying network state (e.g. topology and routing information) as well as statistics about network traffic (e.g. link utilization and traffic matrix). Then, using this aggregated information and based on congestion control policies defined by the network operators, OpenTCP determines a specific set of adaptations for TCP.

At a high level, congestion control policies define which statistics need to be collected, which high level performance metrics the operator would like to optimize (e.g. minimize drops, maximize utilization, or minimize flow completion times), and what the constraints of the system are. OpenTCP periodically sends Congestion Update Epistles or CUEs to the end-hosts which, in turn, update their behaviour using a simple kernel module that can adapt TCP. Details about statistics, policies, and CUEs appear in Chapter 5.

Consider the following simple example. Imagine a network where all links have very
low utilization (say below 50%) at all times. If the network operator aims at minimizing
flow completion times in this network, it makes sense to increase the TCP initial con-
gestion window size, as suggested by Dukkipati et al. [49]. The exact value of the initial
congestion window will be a function of the number of flow initiations in the system (network state), and how aggressively the operator wants the system to behave (congestion control policy). For a network where dropping a few packets is not a major problem, the
operator can define a policy like the following: if all link utilizations are below 50%, the
initial congestion window size can be increased to 20 segments instead of the default value
of four. If the operator is more conservative, the window size can be set to a smaller
value (e.g. 5 segments), improving flow completion times with smaller risk of causing
packet drops. The operator can even leave it to OpenTCP to dynamically choose the
right value for the initial congestion window size.

1.2 Contributions and Impact

In this thesis, we identify the need for and the underlying requirements of a conges-
tion control adaptation mechanism in data center networks. To this end, we propose
OpenTCP as a TCP adaptation framework that works in Software-Defined Networks
(Chapter 5). We design OpenTCP as a framework for dynamic adaptation of TCP based
on network and traffic conditions in SDNs. We explain OpenTCP’s architecture which
allows network operators to define rules for adapting TCP as a function of network and
traffic conditions.

To evaluate OpenTCP’s feasibility, in Chapter 6 we implement a simulated OpenTCP
environment under the NS-2 [6] simulation package. Moreover, we deploy OpenTCP
at SciNet [8], Canada’s largest academic high-performance computing data center with
about 4000 servers. OpenTCP is used to adapt TCP parameters in these environments.
We show that using simple congestion control policies to modify TCP can improve TCP
performance. For example, using very simple changes in initial congestion window size when the network load is low, OpenTCP can lead to up to 50% reductions in average flow completion times.

To demonstrate real world examples of OpenTCP, we begin this thesis with two TCP adaptation mechanisms in real-world situations and show the impact of TCP adaptation on saving resources and improving performance. In Chapter 3 we present Trickle to rate limit TCP in YouTube video streaming. The idea is to dynamically adapt TCP’s maximum congestion window size such that TCP limits both the overall data rate and the maximum packet burst size. This technique reduces queueing and packet loss by smoothly rate-limiting TCP transfers. Then, Chapter 4 studies the impact of TCP Pacing in shallow-buffer routers in data center networks. We show that by avoiding bursty packet drops, paced TCP achieves higher throughput than non-paced TCP. Moreover, we show that for a given buffer size, the benefits of using paced TCP will diminish as we increase the number of concurrent connections beyond a certain threshold. Hence, there is a need for a dynamic framework to automatically enable/disable TCP pacing depending on network conditions. Both Trickle and TCP pacing examples serve as use-cases for OpenTCP’s potential impact in real world scenarios. While OpenTCP is not realized in today’s Internet-scale services, TCP adaptation is a common practice. We tie OpenTCP to the adaptation task and make it dynamic depending on network status, traffic patterns, and application types.
Chapter 2

Related Work

In computer networks, congestion occurs when one or more links become over-subscribed. During congestion, actions need to be taken by the transmission protocols and the network routers to ensure network stability, high throughput, and fair resource allocation to network users [96]. Modern networks use congestion control mechanisms to avoid congestion and bring the network to a stable state after episodes of congestion. Congestion control requires controlling traffic in a network and avoiding congestion collapse caused by over-subscription of any of the network’s links. In this chapter, we explain the previous work on congestion control mechanisms.

Below are a few definitions that we use throughout this thesis.

1. Flow: A network flow is a sequence of packets that share the same protocol, sender and destination IP addresses, and the sender and destination port numbers. Usually, packets in a flow follow the same path in the network [105].

2. Round-Trip Time (RTT): The round-trip time of a packet is the total time taken by the network to deliver the packet to its receiver, and to deliver its acknowledgment to the sender [40].

3. Congestion window: The congestion window is the maximum number of packets
a sender can send in one RTT before receiving any acknowledgements from the receiver. In other words, the congestion window is the number of bytes (or packets) a sender can have in flight [73].

4. Flow Completion Time: Latency of a flow is the time it takes for the flow to finish, or the flow completion time. Flow completion time usually has a direct impact on user experience, therefore users typically want their flows to complete as quickly as possible. This makes flow completion time a very important performance metric for the user [48].

2.1 Congestion Control Mechanisms

The vast literature on congestion control suggests three main types of congestion control mechanisms:

- **End-to-End congestion control**: The protocols in this scheme use end-to-end network measurements, such as packet drops or packet delay, as implicit signs of congestion.

- **Active Queue Management (AQM)**: In these schemes, the routers along a flow’s path play a more active role in measuring congestion (such as buffer occupancy) and signalling to the end systems.

- **Hybrid of End-to-End and AQM**: As the name suggests, these are hybrids of the above two schemes whereby the routers explicitly tell the receiver the state of congestion and the senders adjust their windows based on the feedback information.

Below, we describe these categories in more detail.
2.1.1 End-to-End Congestion Control

End-to-end congestion control protocols are widely used in the Internet. In these systems, the end-hosts run a congestion control protocol and probe the network to discover whether there is congestion. These protocols use a closed-loop to control congestion, employing packet drops or increases in RTT measurements as a sign of congestion in the network.

There are many proposed end-to-end congestion control protocols [41, 42, 71–73, 108]. Transmission Control Protocol (TCP) [73] along with its variations is the dominant one used in the Internet and elsewhere [16, 57, 87, 110]. TCP has the following two key functions:

1. It provides reliable and in order delivery of bytes to the higher application layer.

2. It performs congestion control, ensuring no links on the path are over-subscribed in terms of load.

In this work, we focus on the latter function.

TCP is a window-based, closed-loop congestion control protocol. It probes the network to discover whether there is congestion, taking packet drops or increases in RTT as implicit signals of congestion. It controls the number of outstanding packets in the network by keeping track of acknowledgments for data sent. Numerous Request for Comments (RFC) pages describe TCP function [21, 31, 38, 67, 74, 97, 99, 106, 123] which operating systems have implemented and enabled by default. TCP uses adaptive congestion control mechanisms that react to congestion events by limiting the sender’s transmission rate. These mechanisms allow TCP to adapt to heterogeneous network environments and varying traffic conditions, and thereby precluding severe congestion events. TCP congestion control works on end-to-end basis, whereby each connection begins with a question [53]:

*What is the appropriate sending rate for the current network path?*
Chapter 2. Related Work

The question is not answered explicitly, but each TCP connection determines the sending rate by probing the network path and altering the congestion window \((cwnd)\) based on perceived congestion. To determine the congestion window, a number of TCP variants employ various mechanisms, discussed in detail in the next section. Irrespective of congestion control variants, it is clear that throughput and latency in a network are heavily influenced by the behaviour of the transport protocol and its underlying congestion control mechanisms.

TCP variants can be differentiated in the following two ways:

1. The method to detect congestion: The most widely used TCP variants, such as Tahoe [73], Reno [124], NewReno [54], Bic [133], and Cubic [66] rely on packet drops to detect congestion; other implementations of TCP, such as CARD [76], DUAL [127], Tri-S [81], FAST [129] and Vegas [33], rely on increases in RTT to detect congestion.

2. The dynamics of TCP congestion control and avoidance: TCP variants take different approaches to increasing/decreasing their congestion window.

Below, we explain some of the most important TCP variants in more detail.

Figure 2.1 shows a typical trajectory of a TCP congestion window. A TCP connection goes through two phases: Slow start and Additive Increase Multiplicative Decrease (AIMD).

**Slow Start:** Each TCP connection starts with a pre-configured small initial congestion window (typically 4 Maximum Segment Size (MSS) [19, 21])\(^1\), and probes the network for the available bandwidth using the Slow start procedure. The goal of slow start is to keep a new sender from overflowing a network’s buffers, while expanding the congestion window and avoiding performance loss. Slow start increases the congestion

---

\(^1\)Recently ICWND10 [79] – increasing the initial congestion window further to 10 – has been proposed as an IETF draft by Google.
Figure 2.1: TCP congestion control mechanisms: Slow start and Additive Increase Multiplicative Decrease (AIMD).

window by one MSS for each new acknowledgment received; this results in the window doubling after each window’s worth of data is acknowledged. With this exponential increase, $RTT \times \log_2 W$ (where RTT stands for round-trip time) time is required to reach a window of size $W$. A connection enters slow start when the flow starts or upon experiencing a packet retransmission time-out; it exits slow start when it detects a packet loss or when the congestion window has reached a dynamically computed threshold, $ssthresh$. More specifically, $ssthresh$ is set to half the size of the congestion window at the time the packet loss was detected.

Additive Increase Multiplicative Decrease (AIMD): TCP exits slow start to enter the congestion avoidance phase, where it continues to probe for the available bandwidth, but more cautiously than in slow start. During periods when no packet losses are observed, TCP performs an Additive Increase of the window size, increasing by 1 MSS each time a full window is acknowledged (i.e., increases the congestion window as $cwnd = cwnd + 1/cwnd$ upon receiving each acknowledgment packet). The handling
of congestion indication (packet loss) varies across different TCP variants. The most commonly used variants are described below.

TCP Tahoe: Tahoe congestion control [73] was introduced in 1987 in response to a series of “congestion collapse” events (a state where the network is live-locked, performing little useful work). Upon detecting a packet loss (either through retransmission time-out or three duplicate acknowledgment packets), a Tahoe sender records the $ssthresh$ as $cwnd/2$ value (the initial value of $ssthresh$ is usually set to the receiver window size), sets $cwnd$ to 1 MSS and enters slow start. It continues slow start so long as $cwnd < ssthresh$, and is in Congestion Avoidance beyond that.

TCP Reno: This is the second version which differs from TCP Tahoe when detecting packet loss through three duplicate acknowledgment packets (an indication of a milder congestion). TCP Reno reduces the $cwnd$ by half (as opposed to reducing the window to one like in Tahoe) to achieve a higher sending rate after loss recovery [124]. The procedure implementing this is called fast-recovery [119].

TCP NewReno: TCP Reno fails to detect and recover efficiently from multiple packet losses in a window [54]. Unlike Reno, NewReno does not exit fast-recovery until all lost packets are acknowledged. Thus, it overcomes the problem faced by Reno of reducing the CWD multiple times.

TCP BIC: Binary Increase Congestion control (BIC), is an implementation of TCP with an optimized congestion control algorithm for high speed networks with high latency: so-called “long fat networks”. The BIC protocol defines a threshold target value called $last_max_cwnd$ for congestion window. BIC sends packets according to the relation between current $cwnd$ and $last_max_cwnd$; when $cwnd$ is smaller than $last_max_cwnd$, BIC regulates traffic into the network through an aggressive algorithm until $cwnd$ nears the target value at which point it steadies off its sending rate [133]. Upon congestion, BIC backs off $cwnd$ according to $cwnd ← β \times cwnd$, where $β$ is a constant parameter of the algorithm. BIC TCP is implemented and used by default in Linux kernels 2.6.8 up
Related Work

2.1.1 TCP Variants

TCP CUBIC: CUBIC is an implementation of TCP similar to BIC but is a less aggressive and more systematic derivative than BIC. BIC is too aggressive for TCP flows under short RTT or low speed networks. However, CUBIC uses a window growth function that grows slower than BIC. TCP CUBIC is implemented and used by default in Linux kernels 2.6.19 and above. BIC, CUBIC, HighSpeed TCP [52], Fast TCP [129], and Scalable TCP [89] are all examples of TCP variants which target to improve the steady-state performance of long-lived, large-bandwidth flows.

TCP Vegas: Unlike the above mentioned flavors of TCP, Vegas congestion control algorithm emphasizes packet delay, rather than packet loss, as a signal to help determine the rate at which to send packets [33]. TCP Vegas detects congestion based on increasing RTT values of the packets in the connection. The protocol relies heavily on accurate RTT measurement, which is susceptible to noise in the very low latency environments [16]. Small noisy fluctuations of latency become indistinguishable from congestion and the algorithm can over-react.

2.1.2 Active Queue Management

In traditional implementations of router queue management, the role of the routers is limited to dropping packets when a buffer becomes full (in which case the mechanism is called Drop-Tail). As explained in Section 2.1.1, TCP senders run a congestion control protocol that interprets packet loss or increases in RTT as a sign of congestion to which they react by decreasing the sending rate. Active Queue Management (AQM) mechanisms are complementary to end-to-end congestion control where routers dynamically signal congestion to sources even before the queue overflows; either explicitly by marking packets, or implicitly by dropping packets.

Random Early Detection (RED) [58], is a queue based AQM which couples congestion notification to queue size; in other words, RED drops packets randomly with a proba-
Figure 2.2: Dropping Probability of RED AQM.

ability that is proportional to the average queue length. Figure 2.2 shows the dropping probability of RED as a function of the average queue length. The objective of RED is to prevent bursts of packet drops by dropping packets early. Specifically, it sends an early congestion signal to the flow to slow down before the queue overflows, which helps in maintaining a smaller average queue length.

RED is a powerful mechanism for controlling traffic. It can provide better network utilization than Drop-Tail if properly used, but can induce network instability and major traffic disruption if not properly configured [36]. Moreover, since RED relies on queue length as an estimator of congestion, in the presence of a persistent queue, RED fails to detect the number of competing connections or the severity of congestion.

BLUE [51] is a fundamentally different AQM that uses packet loss and links idle events to manage congestion. It maintains a single probability, which it uses to mark (or drop) packets when they are queued. If the queue is continually dropping packets due to buffer overflow, BLUE increments the marking probability, thus increasing the rate at which it sends back congestion notification.

To improve RED’s ability to distinguish unresponsive users, a few variants have
been proposed, including RED with penalty box [56] and Flow Random Early Drop (FRED) [94]). These variants incur extra implementation overhead as they need to collect certain types of information: the former notes unfriendly flows while the latter checks active connections.

Another variant of RED is Stabilized RED (SRED) [93] which stabilizes the occupancy of the router queue. SRED estimates the number of active connections and identifies misbehaving flows but does not propose a simple router mechanism for penalizing these flows. By way of contrast, the “CHOose and Keep for responsive flows, CHOose and Kill for unresponsive flows” (CHOKe) [103] packet dropping scheme approximates max-min fairness for the flows that pass through a congested router. CHOKe discriminates against flows which submit more packets per second than is allowed by their fair share. As a result, aggressive flows are penalized.

Explicit Congestion Notification (ECN) [55] is an AQM, originally designed to work in conjunction with RED. ECN attempts to prevent packet drops by marking packets using a special field in the IP header. ECN is an optional feature that is only used when both endpoints support it and are willing to use it and is only effective when supported by the underlying network. When ECN is successfully negotiated, an ECN-aware router may set a mark in the IP header instead of dropping a packet in order to signal impending congestion. The receiver of the packet echoes the congestion indication to the sender, which reacts as though a packet were dropped. This behaviour is desirable as it removes the overhead of packet drops.

On the one hand, RED and its variants are queue-based AQMs whereby a backlog of packets is inherently necessitated by the control mechanism, as congestion is observed only when the queue length is positive. This creates unnecessary delay and jitter. On the other hand, flow based AQMs, such as GREEN [70], determine congestion and take action based on the packet arrival rate. GREEN measures the packet arrival rate at the link and is based on a threshold function. If the link’s estimated data arrival rate, $x$, is
above the target link capacity, $c$, the rate of congestion notification, $P$, is incremented by $\Delta P$. Conversely, if $x$ is below $c$, $P$ is decremented by $\Delta P$.

### 2.1.3 Hybrid of End-to-End and AQM Congestion Control

The end-to-end congestion control still dominates most of the Internet and data center congestion control mechanisms. However, as explained in Section 2.1.2, AQM schemes give routers a more active role in anticipating congestion. In this section we explain congestion control schemes that work co-operatively between routers and end-hosts. These schemes combine end-to-end congestion control protocols with AQM to provide robust and smooth congestion control in the network.

Explicit Control Protocol (XCP) [88] generalizes ECN by replacing the one-bit congestion feedback with a field to reflect the degree of congestion along the path. The routers explicitly tell the receiver the state of congestion and how to react to it allowing senders to adjust their windows based on the precise feedback information. XCP controls link utilization by adjusting its aggressiveness to the spare bandwidth and the feedback control delay, thus achieving stability and efficiency, even for large bandwidth-delay product networks. It controls fairness by (conservatively) managing the bandwidth distribution among flows. More specifically, it reclaims the bandwidth from flows with rates above their fair share and distributes it to flows with lower rates. XCP solves the problems with TCP in a static scenario in which there are a fixed number of flows with an infinite amount of data to send. But in practice, flows arrive randomly and transfer a finite arbitrary amount of data. In a dynamic environment with a mix of flow sizes, XCP is inefficient and unfair [47].

The Rate Control Protocol (RCP) [45] is a router co-operative congestion control algorithm designed to improve flow completion times by using routers to set the flow’s correct rate. Unlike XCP which is designed to improve the performance of long-lived flows, RCP’s main goal is to reduce the flow completion time of typical flows of typical
users in the Internet. RCP requires an end-host congestion control layer that sits between
IP and transport layer. Each RCP router needs to maintain a single fair-share rate per
link, and RCP packets carry the rate of the bottleneck link along the flow’s path. If
the router’s fair-share rate is smaller than the current rate of the bottleneck link, the
router overwrites the value in the packet. After one RTT, the source learns the fair-share
rate of the bottleneck link. RCP has been implemented in test-beds [44]; however, its
full use has not been realized as it requires changes to router hardware and end-host
implementation.

Congestion Manager (CM) has been proposed to control and manage bandwidth
sharing between transport protocols in the Internet [25, 26]. The CM maintains host-
and domain-specific path information, and orchestrates the transmissions of multiple
concurrent streams from a sender destined to the same receiver and sharing the same
congestion properties.

DCTCP [16] or Data Center TCP is proposed as a data center transport protocol
using ECN to detect congestion in routers. The routers mark every arriving packet if the
queue occupancy is greater than a fixed parameter, $K$. Then, DCTCP senders estimate
the fraction of packets that are marked, called $\alpha$; this is updated for every window of
data (roughly one RTT) as follows:

$$\alpha \leftarrow (1 - g) \times \alpha + g \times F$$

(2.1)

where $F$ is the fraction of packets marked in the last window of data, and $0 < g < 1$ is
the weight given to new samples in the estimation of $\alpha$. While traditional TCP protocols
cut the window size by a factor of 2 in response to congestion, DCTCP uses $\alpha$:

$$cwnd \leftarrow cwnd \times (1 - \frac{\alpha}{2}).$$

(2.2)

DCTCP is still unexplored in practical settings. Its analysis statically sets $K = 20$
packets for 1 Gbps and \( K = 65 \) packets for 10 Gbps ports; \( g \) is set to \( \frac{1}{16} \) in all experiments. Even though the results show promising advantages over TCP, there is no evidence that the chosen parameters can be applied in all networks. The analysis does not show that DCTCP can avoid network instability, if it is not properly configured. Moreover, to reduce the router build-up cost, many data center networks now use routers with small buffers [27] but the small-buffer routers and small RTTs in data center infrastructures lead to a disparity between the total capacity of the network and the capacity of individual queues, this means that ECN-based congestion control mechanisms can mistake the buffer occupancy for congestion in the network.

\subsection{Data Center Traffic}

While there is increasing interest in using data centers in a wide variety of scenarios, few measurement studies consider the nature of data center traffic [28,29,62,83,117].

A typical data center architecture, illustrated in Figure 2.3, consists of one or more tree-shaped layers of switches or routers [13]. A three-tiered design has a core tier in the root of the tree, an aggregation tier in the middle, and an edge tier at the leaves of the tree. A two-tiered design has only the core and the edge tiers. Typically, a two-tiered design can support 5K to 8K hosts. Recent research proposals such as VL2 [62], Fat-Tree [13], and BCube [64], combine links and switches with variants of multi-path routing such that the core of the network is not over-subscribed.

The data center environment differs significantly from wide area networks [84]. For example, RTTs are in order of a few milliseconds, and applications simultaneously need extremely high bandwidths and very low latencies. Often, there is little statistical multiplexing: a single flow can dominate a particular path. At the same time, the data center environment offers certain luxuries. The network is largely homogeneous and under a single administrative control. Thus, backward compatibility, incremental deployment, and
fairness to legacy protocols are not major concerns [16]. These properties suggest that there might be better alternatives for data center congestion control than conventional congestion control protocols. Using OpenTCP to systematically monitor the status of data center and dynamically adapt TCP, provides a framework to pursue such goals.

### 2.2.1 Communications in Data Centers

There is a wide variety of applications across data centers, ranging from customer-facing applications, including web services, file stores, authentication services, and custom enterprise applications to data intensive applications, such as MapReduce [43] and search indexing [28]. A common application structure in data centers is partition/aggregate where traffic belongs to distributed applications that perform scatter/gather types of communications, such as Google File System (GFS) [59], Panasas [130], pNFS [113], and Hadoop [114].

Many applications have a multi-layer partition/aggregate pattern workflow, as illustrated in Figure 2.4, where latency at one layer affects other layers. Requests from higher layers of the application are broken into pieces and sent to workers in lower layers. The responses of these workers are aggregated. For example, in a web search, a query might
be sent to many aggregators and workers, each responsible for a different part of the database index. Based on the replies, an aggregator might refine the query and send it out again to improve the relevance of the result. Lagging instances of partition/aggregate can thus create delays in answering queries, making the 99.9th tail latency an important factor in data centers.

Recent measurements reveal that 99% of the traffic in today’s data centers is TCP traffic \cite{16}. As explained in Section 2.1, TCP is a mature technology with many optimizations and variants, and it meets the communication needs of most applications. However, the workloads, scale, and environment of the data center is different than that of the WAN assumptions on which TCP was originally designed. For example, in contemporary operating systems such as Linux, the default Retransmission Time-Out (RTO) timer value is set to 200ms, a reasonable value for WAN, but 2-3 orders of magnitude greater than the average RTT time in the data center. As a result, there are new shortcomings that demand a fresh look at data center congestion control protocols. For example, TCP Throughput Collapse, also known as incast \cite{126}, is a behaviour of TCP that results in gross under-utilization of link capacity in data center partition/aggregate communication patterns. The incast problem arises in many typical data center applications, for example, in cluster storage \cite{59}, when storage nodes respond to requests for
Chapter 2. Related Work

2.2.2 Workload Characterization

The previous section describes the communication pattern in data centers. This section notes the previous studies analyzing the aggregate network transmission behaviour of the applications.

In [16], the authors present measurements of a 6000 server Microsoft production cluster and suggest that three types of traffic co-exist in web search data centers: throughput-sensitive large flows (1 MB to 50 MB), delay sensitive short flows (50 KB to 1 MB) and bursty query traffic (1 KB to 2 KB).

The distribution of flow sizes, number of active flows, flow utilization, and loss patterns vary among data centers. A recent study by Benson et al. [28] measures the network traffic in 10 data centers and suggests the following:

- Most flows in the data centers are small ($\leq 10KB$), with a significant fraction of them lasting under a few hundreds of milliseconds, and the number of active flows per second is under 10,000 per rack across all data centers.

- Despite the differences in the size and usage of the data centers, traffic originating from a rack in a data center is ON/OFF in nature with properties that fit heavy-tailed distributions.

- In the cloud data centers, a majority of the traffic originated by servers (80%) stays within the rack.

- Irrespective of the type, in most data centers, link utilizations are rather low in all layers except the core, where a subset of the core links often experience high
utilization. Furthermore, the exact number of highly utilized core links varies over time, but never exceeds 25% of the core links in any data center.

- Losses occur within the data centers; however, losses are not localized to links with persistently high utilization. Instead, they occur at links with low average utilization, implicating momentary spikes as the primary cause. The magnitude of losses is greater at the aggregation layer than at the edge or the core layers.

- Link utilizations are subject to time-of-day and day-of-week effects across all data centers. However, in many cloud data centers, the variations are an order of magnitude more pronounced at core links than at edge and aggregation links.

Enlightened by the workload characteristics of data centres, recent research has begun to focus on flow prioritization in data centers [17, 18, 69, 125, 131, 137]. The unique workload, scale, and environment of data centers is different than that of the WAN environment in which TCP was originally designed [37]. Specifically, unlike the long-lived streaming traffic on the Internet, the traffic patterns between data center nodes mostly consist of short-sized flows. One line of work that is similar to DCTCP [16], is the HULL (High-bandwidth Ultra-Low Latency) architecture [17] that tries to adapt congestion control algorithms to provide flow prioritization. The basic idea is to strive to keep switch queues empty using phantom queues that deliver congestion signals before network links are fully utilized and queues form at switches. While HULL and DCTCP improve latency, both are using implicit signals (such as ECN bit) to estimate the congestion in the network. These implicit signals cannot precisely estimate the correct flow rates to use so as to schedule flows to minimize latency [18]. Moreover, due to the bursty nature of traffic, keeping network queues empty is challenging and requires carefully designed rate-control and hardware packet pacing at the end-hosts, and trading off network utilization [17].

Recently, pFabric [18], tries to liberate TCP congestion control from some of its com-
plexities by proposing a minimalistic datacenter fabric design that only includes fundamental pieces of congestion control functionality. While the work shows a good potential for minimal TCP implementation via simulations, the realization of this approach in an operational data centre requires a lot of considerations and non-trivial changes to applications that run on top of TCP.

OpenTCP is orthogonal to previous works trying to improve congestion control in data centers. Our goal here is not to try to beat these proposals in a quantitative race, but instead to use information about network state and traffic conditions in order to adapt congestion control mechanism in a controlled and guided manner. OpenTCP is not meant to be a new variation of TCP but complements previous efforts by making it easy to switch between different TCP variants automatically (or in a semi-supervised manner), or to adapt TCP parameters based on network conditions, or incrementally adopt a new congestion control mechanism in a data centre. For instance, we can use OpenTCP to apply DCTCP or CUBIC in a data center environment. The decision on which variant to use is made in advance through congestion control policies defined by the network operator (Chapter 5).

2.3 Software-Defined Networks

The end-to-end argument was first put forward in the early 1980s as a core design principle of the Internet. The argument, as framed then, remains relevant and powerful: the fundamental architecture of the Internet endures, despite change in both underlying technologies and applications [39]. Nevertheless, the evolution of the Internet and the move towards cloud computing also shows the limits of foresight: the data center environment, the applications that run on top, and the interpretation of the end-to-end argument itself have all evolved.

The minimalism and simplicity of the Internet’s design has led to enormous growth
and innovation, yet the network itself remains hard to change, fragile, and hard to manage. Today, the demand to design flexible, globally controlled data center networks is steadily increasing. Software-Defined Networking (SDN) is an approach where the “control plane” and the “data plane” of the network are separate. In other words, SDN decouples the system that makes decisions about where traffic is sent (the control plane) from the underlying system that forwards traffic to the selected destination (the data plane) [9, 10]. The need for simplicity, dynamicity, and programmability in data centers is reflected in the design of several software-defined networks such as 4D [61], Ethane [35], Tesseract [134], and Openflow [98]. SDN proposes several solutions to separate control and data planes in enterprise networks. This separation simplifies modifications to the network control logic, enables the data and control planes to evolve and scale independently, and decreases the cost of the data plane element.

The 4D architecture advocates decomposition of the network function into data, dissemination, discovery, and decision planes. Tesseract implements a 4D control plane and shows the merits of 4D in practice. Ethane is a notable example of recent work which relies on the separation of data and control planes. The OpenFlow project was initiated to provide a vendor-agnostic interface with network elements to enable innovation in the control plane.

Figure 2.5 illustrates an OpenFlow switch architecture consisting of three parts:

1. a flow table, with an action associated with each flow entry, to tell the switch how to process the flow;

2. a secure channel that connects the switch to a remote control process (called the controller), allowing commands and packets to be sent between a controller and the switch; and

3. the OpenFlow protocol, which provides an open and standard way for a controller to communicate with a switch.
Figure 2.5: OpenFlow switch architecture.

The controller in OpenFlow architecture has a global view of the network state; it adds and removes flow-entries from the Flow Table. A static controller could be a simple application running on a PC to statically establish flows interconnecting a set of computers. NOX [63] is a widely-used OpenFlow controller software. There are several proposals to improve the scalability and fault tolerance of NOX and OpenFlow controllers in general [121, 135]. The controller landscape is growing rapidly, over the past years OpenDaylight [7], Floodlight [3], Beacon [2] and many more have been introduced as SDN controllers.

OpenTCP is designed to use SDNs as a platform. SDN proposals often focus on control plane design, suggesting radical departures from the standard distributed Internet architecture. However, much less effort has been put into congestion control adaptation for this new environment; most proposals focus on routing and forwarding feature and ignore the advantages that SDNs provide in congestion control domain. On the other hand, end-to-end congestion control algorithms try to share congested links efficiently and fairly among flows. In the absence of information such as the status of the network, number of active flows at a given instance, and the bottlenecks of the network, data
center administrators design and implement distributed rules to control the congestion in the network for different classes of service levels. OpenTCP bridges the gap and is designed to operate as a congestion control framework for SDNs by taking advantage of the luxuries provided in SDNs, such as programmable switches and centralized controller. In the next two chapters, we introduce two real world TCP adaptation mechanisms and then in Chapter 5, we present OpenTCP’s architecture and design.
Chapter 3

Trickle: Adapting TCP to Rate Limit Video Streaming

This chapter introduces Trickle, a server-side TCP adaptation mechanism that uses TCP to rate limit video streaming. Specifically, Trickle places an upper bound on TCP’s congestion window as a function of the streaming rate and the round-trip time. Section 3.1 offers reasons to use TCP to rate limit video streams, Section 3.2 gives an overview of YouTube video streaming, and Section 3.3 explains the design and implementation of Trickle. Sections 3.4 and 3.5 present the result of our deployment of Trickle on YouTube production data centers in Europe and India.

3.1 Motivation

YouTube is one of the most popular online video services. In Fall 2011, it was reported to account for 10% of Internet traffic in North America [11]. This vast traffic is delivered over TCP using HTTP progressive download. Because the video is delivered just-in-time to the video player, when the user cancels a video, only a limited quantity of data is discarded, conserving network and server resources. Since TCP is designed to deliver data as quickly as possible, the YouTube server, ustreamer, limits the data rate by pacing the
data into the connection. It does so by writing 64kB data blocks into the TCP socket at fixed intervals. Unfortunately, this technique, termed application pacing, causes bursts of back-to-back data packets in the network, and these bursts have several undesirable side effects. For example, they are responsible for over 40% packet losses in YouTube videos on at least one residential DSL provider [15].

This problem is not specific to YouTube videos. Similar rate limiting techniques are implemented in other popular video websites [24], and all are expected to experience similar side effects. For example, Netflix sends bursts as large as 1 to 2MB.

As an alternative to application pacing, we propose using Trickle to rate limit TCP on the server side. The key idea is to place a dynamic upper bound on the congestion window (cwnd) such that TCP itself limits both the overall data rate and the maximum packet burst size using ACK-clocking [75]. The server application periodically computes the cwnd bound from the network Round-Trip Time (RTT) and the target streaming rate, using a socket option to apply it to the TCP socket. Once set, the server application can write into the socket without a pacing timer and TCP will take care of the rest. Trickle requires minimal changes to server applications and the TCP stack. In fact, Linux already supports setting the maximum congestion window in TCP.

This chapter contributes a simple and generic technique to reduce queueing and packet loss by smoothly rate-limiting TCP transfers. It is not a special mechanism tailored only for YouTube and requires only a server-side change for easy deployment. As TCP has emerged as the default vehicle for Internet applications, many require certain kinds of throttling. The common practice, application pacing, may cause burst losses and queue spikes, but in our experiments on production YouTube data centers, we found that Trickle reduces the packet losses by up to 43% and RTTs by up to 28% over application pacing. The rest of the chapter covers the design, our experiments, and discussions of other techniques and protocols.
3.2 YouTube Video Streaming

The YouTube serving infrastructure is complicated, with many interacting components, including load balancing, hierarchical storage, multiple client types and a number of format conversions. Most are not important to the experiment at hand, but some need to be described in more detail.

All YouTube content delivery uses the same server application, ustreamer, independent of client type, video format or geographic location. Ustreamer supports progressive HTTP streaming and range requests. Most of the time, a video is delivered over a single TCP connection, but certain events, such as skipping forward or resizing the screen can cause the client to close one connection and open a new one.

The just-in-time video delivery algorithm uses two phases: a startup phase and a throttling phase. The startup phase builds up the playback buffer in the client to minimize the likelihood of player pauses due to rebuffering (buffer under-run) events. Ustreamer...
sends the first 30 to 40 seconds of video (codec time, not network time) as fast as possible into the TCP socket, like a typical bulk TCP transfer.

In the throttling phase, ustreamer uses a token bucket algorithm to compute a schedule for delivering the video. Tokens are added to the bucket at 125% of the video encoding rate and are removed as the video is delivered. The delay timer for each data block (nominally 64kB) is computed to expire as soon as the bucket has sufficient tokens. If the video delivery is running behind for some reason, the calculated delay will be zero, and the data will be written to the socket as fast as TCP can deliver it. The extra 25% added to the data rate reduces the number of rebuffering events when there are unexpected fluctuations in network capacity, without incurring too much additional discarded video.

Figure 3.1 illustrates the time-sequence graph of a packet trace from a sample YouTube video stream. The x-axis is time and the y-axis is the bytes of video. Vertical arrows represent transmitted data segments with a range of bytes at a particular time. After the flow is served for 1.4 seconds in the startup phase (in this, the first 30 seconds of the video playback), the YouTube server starts to throttle the sending of bytes to the network. During the throttling phase, every network data write is not more than one block size plus headers.

In some environments the data rate is limited by something other than the ustreamer-paced writes. For example, some video players implement their own throttling algorithms [24], especially on memory and network-constrained mobile devices. These devices generally stop reading from the TCP socket when the playback buffer is full. This is signalled back to the sender through TCP flow control using the TCP receiver window field. As a consequence, ustreamer is prevented from writing more data into the socket until the video player reads more data from the socket. In this mode, the sender behaviour is driven by the socket read pattern of the video player: sending bursts is determined by the player read size.

For short videos (less than 40 seconds) and videos traversing slow or congested links,
Figure 3.2: Time vs. sequence of bytes graph for a YouTube video with RTT 30ms using Trickle.

Figure 3.2: Time vs. sequence of bytes graph for a YouTube video with RTT 30ms using Trickle.

3.3 Trickle

3.3.1 The Problem: Bursty Losses

The just-in-time delivery described above smooths the data across the duration of each video, but it has an unfortunate interaction with TCP that causes it to send each 64kB socket write as 45 back-to-back packets.

The problem is that bursts of data separated by idle periods disrupt TCP’s self clock. For most applications TCP data transmissions are triggered by the ACKs returning from the receiver, and this provides the timing for the entire system. With YouTube, TCP typically has no data to send when the ACKs arrive; thus, when ustreamer writes the
These bursts can cause significant losses, e.g., 40% of YouTube losses in a residential ISP [15]. Similar issues have also been reported by YouTube network operations and other third parties. Worse yet, these bursts disrupt latency-sensitive applications by creating periodic queue spikes [86, 101]. The queuing time of a 64KB burst over an 1Mbps link is 512ms.

Our goal is to implement just-in-time video delivery using a mechanism that does not introduce large bursts and preserves TCP’s self clock.

3.3.2 Basic Ideas

A quick solution to the burst problem is to use smaller blocks, e.g., 16kB instead of 64kB. However, this would quadruple the overhead associated with write system calls and timers on the IO-intensive YouTube servers. A better solution would to implement a rate limit in TCP itself by for example, leveraging TCP flow control by fixing the receiver’s window ($rwin$) equal to the target streaming rate multiplied by RTT. Once the receiver fixes $rwin$, ustreamer can write the video data into the socket as fast as possible. The TCP throughput will be limited by the receive window achieving the target streaming rate.

However, this receiver based approach is not practical because YouTube does not have control of the browsers. Our solution is to set an upper-bound on $cwnd$ to $\text{target\_rate} \times RTT$, where the $\text{target\_rate}$ is the target streaming rate of a video in the throttling phase. Fortunately, Linux already provides this feature as a per-route option called $cwnd\_clamp$. We merely wrote a small kernel patch to make it available as a per-socket option.

To illustrate the smoothing effect of Trickle we show the time-sequence plot of a real YouTube connection in Figure 3.2. The initial full rate startup phase lasts 2.5 seconds. In some cases using congestion window validation [67] would force TCP to do new slow starts after idling over several Retransmission Time-Outs. It is not always useful in YouTube as the application writes are more frequent.
the throttled phase the transmission is rate-limited to 600kbps, by sending $cwnd_{clamp}$ of data per RTT. The result is a smoother line than the staggered steps (bursts) created by application pacing, shown in Figure 3.1.

### 3.3.3 Challenges

The basic idea faces two practical challenges:

**Network congestion causing rebuffering.** Following a congestion episode, ustreamer should deliver data faster than the target rate to restore the playback buffer. Otherwise, the accumulated effects of multiple congestion episodes will eventually cause rebuffering events and the codec runs out of data. The current application pacing achieves this implicitly: when TCP slows down enough to stall writes to the TCP socket, ustreamer continues to accumulate tokens. Once the network recovers, ustreamer writes data continuously until the tokens are drained, at which point the average rate for the entire throttled phase matches the target streaming rate. Clamping the $cwnd$ will not allow this type of catch-up behaviour.

**Small $cwnd$ causing inefficient transfers.** Sending data at 500kbps on a 20ms RTT connection requires an average window size of 1250 bytes, which is smaller than the typical segment size. With such a tiny window, all losses must be recovered by timeouts, since TCP fast recovery requires a window of at least four packets [20]. Furthermore, using a tiny window increases ustreamer overhead because it defeats TCP segmentation offload (TSO) and raises the interrupt processing load on the servers.

### 3.3.4 Design

Algorithm 1 presents the pseudo code of Trickle that incorporates the basic design and addresses both challenges mentioned above. After the startup phase, ustreamer determines the streaming rate, $R$, based on the video encoding rate as described in Section 3.2. When the data from the cache system arrive, ustreamer gets RTT and MSS of the con-
nection (using a socket option) to compute the upper bound of the cwnd, clamp. But before applying the clamp to the connection, ustreamer takes two precautions.

First, to deal with transient network congestion, ustreamer adds some headroom to the clamp. We used 20% headroom, similar results are achieved with 5% headroom. If the link is experiencing persistent congestion and/or does not have enough available bandwidth, ustreamer removes the cwnd clamp by setting it to infinity letting TCP stream data as fast as possible. When the goodput has reached $R$, ustreamer will start clamping again.

Second, ustreamer lower bounds clamp to 10 Maximum Segment Size (MSS). Studies show Internet paths can tolerate bursts of such size [30, 49]. However doing this increases the streaming rate beyond $R$ to 10*MSS/RTT. Hence, ustreamer throttles the data write in a fashion similar to application pacing but the burst size is strictly limited to 10 packets.

Third, ustreamer clamps the cwnd via a socket option. If the data write throttling is enabled, it throttles the write at rate $R$. Otherwise, it writes all available data into the socket.

### 3.4 Experiment

We performed live experiments to evaluate Trickle on production YouTube data centers. We begin this section with the methodology we used to compare Trickle and existing application pacing, followed by details of the data centers. Then, we present our measurement results; these validate the A/B test setup and Trickle implementation.

#### 3.4.1 Methodology

We setup A/B test experiments on selected servers in production YouTube data centers. The first goal is to evaluate if Trickle reduces the burst drops and queuing delays. The second is to determine whether the streaming quality is as good or better than current
systems. This is done by measuring the average streaming rate and the rebuffering events. We also compare it to the simplest solution, \textit{i.e.}, reducing the block size from 64kB to 16kB.

We run a 4-way experiment by splitting the servers into the following four groups.

1. Baseline1. Application pacing with 64kB block
2. Baseline2. Application pacing with 64kB block
3. Trickle.
4. shrunk-block. Application pacing with 16kB block

In order to make apples-to-apples comparison, new TCP connections (video requests) are randomly assigned to the servers in different experiment groups. Thus, each experiment group receives similar distributions of users and video requests. Using two control groups helps us estimate the confidence level of the particular metric used in our analyses.

\begin{algorithm}
\textbf{Algorithm 1:} Trickle algorithm in throttling phase
\begin{algorithmic}
\State \texttt{R = target\_rate(video\_id)}
\State \texttt{while (new data available from the cache)}
\hspace{1em} \texttt{rtt = getsockopt(TCP\_INFO)}
\hspace{1em} \texttt{clamp = rtt * R / MSS}
\hspace{1em} \texttt{clamp = 1.2 * clamp}
\hspace{1em} \texttt{goodput = delivered / elapsed}
\hspace{1em} \texttt{if goodput < R:}
\hspace{1.5em} \texttt{clamp = inf}
\hspace{1em} \texttt{if clamp < 10:}
\hspace{2em} \texttt{clamp = 10}
\hspace{2em} \texttt{write\_throttle = true}
\hspace{1em} \texttt{setsockopt(MAX\_CWND, clamp)}
\hspace{1em} \texttt{if write\_throttle:}
\hspace{1.5em} \texttt{throttles writes at rate R}
\hspace{1em} \texttt{else:}
\hspace{1.5em} \texttt{write all available data}
\end{algorithmic}
\end{algorithm}
Table 3.1: Network statistics in DC1 and DC2 binned by each experiment group. \( RTT_{\text{start}} \) is the smoothed RTT (srtt) at the end of the startup phase.

All servers use the standard Linux 2.6 kernel with CUBIC [65] congestion control. The rest of TCP configuration details can be found in [46]. For every connection, we record statistics, including video ID, IP and ports, bytes sent and retransmitted, RTTs, rebuffering events, etc in both phases. We also filter the connections that never enter the throttling phase (short video playbacks less than 30 seconds).

### 3.4.2 Validation

We ran experiments for 15 days during Fall 2011 in two data centers representing relative extremes of the user network conditions: DC1 in Western Europe and DC2 in India. We begin analysis by comparing the statistics derived for the control variables in the experiments to validate the A/B test setup. Table 3.1 summarizes these statistics in both data centers. Each experiment group has roughly the same number of flows within each data center. Note that DC1 has 16 times more flows than DC2 because DC1 has more servers participating in the experiment. A group’s average flow length and flow completion are similar across different groups in the same data center.

We also measure RTT and goodput, denoted as \( RTT_{\text{start}} \) and BW respectively, at the end of the startup phase of each connection. BW is the average goodput computed across the startup phase (typically for more than 1mB of video data). In the rest of the chapter, this easily computed metric is used to gauge the network capacity relative to
the target data rate. Since every experiment group uses the original mechanism in the startup phase, RTT_{start} and BW should be similar across groups in the same data center; the results indicate that each experiment group receives similar network and application loads. Interestingly, as we see a vast difference in user network speed in the two regions.

One of the design goals of Trickle is to maintain the same video streaming rate as the existing ustreamer application. To verify this, for each video in our experiments, we calculate the goodput during the throttling phase and ensure the distributions match between all experiment groups. Figures 3.3 and 3.4 demonstrate that the CDFs of goodput in all experiments groups match; i.e. the video streaming rate is consistent whether or not Trickle is enabled.

### 3.5 Analysis

In the following sections, we present our results. These are statistics collected during the throttling period only. As explained in Section 3.4.2, all four experiment groups use the same mechanism in the startup phase, and their results are identical.
3.5.1 Packet Losses

The most important metric is packet loss because Trickle is designed to reduce burst drops. Since losses can not be accurately estimated in live server experiments [23], we use retransmissions to approximate them. Figure 3.5 plots the CDF of flow retransmission rate in the throttling phase for DC1. As shown, the Trickle curve is consistently above all three lines indicating that it successfully and consistently lowers the retransmission rate compared to the other three groups. In Trickle, 90% of connections experience a retransmission rate lower than 0.5%; 85% when they use shrink-block; 80% in baselines. Overall Trickle effectively reduces the drop rate compared to application pacing using 64kB or 16kB block sizes.

Figure 3.6 plots the CDF of flow retransmission rate in the throttling phase for DC2. As shown, all four groups in DC2 have similar distributions because more DC2 users have insufficient bandwidth to stream at the target rate. As described in Section 3.3.4, Trickle will detect that the delivery is falling behind the target rate and stop clamping the cwnd. Connections are not rate-limited by ustreamer throttling but by network bandwidth; they behave much like bulk downloads in all experiment groups.
Figure 3.5: CDF of retransmission rate in the throttling phase in DC1.

Figure 3.6: CDF of retransmission rate in the throttling phase in DC2.
Chapter 3. Trickle: Adapting TCP to Rate Limit Video Streaming

<table>
<thead>
<tr>
<th>BW (Mbps)</th>
<th>DC1</th>
<th>DC2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% flows</td>
<td>avg. retrans. imprv.</td>
</tr>
<tr>
<td>&lt; 0.5</td>
<td>1%</td>
<td>5%</td>
</tr>
<tr>
<td>0.5 – 1</td>
<td>3%</td>
<td>23%</td>
</tr>
<tr>
<td>1 – 2</td>
<td>10%</td>
<td>40%</td>
</tr>
<tr>
<td>2 – 4</td>
<td>28%</td>
<td>53%</td>
</tr>
<tr>
<td>4 – 8</td>
<td>35%</td>
<td>56%</td>
</tr>
<tr>
<td>≥ 8</td>
<td>23%</td>
<td>53%</td>
</tr>
</tbody>
</table>

Table 3.2: The retransmission rate improvement bucketed by user bandwidth

To demonstrate the effect of bandwidth, we determine the average reduction of the retransmission rate between Trickle and baseline bucketed by the flows’ BW, as shown in Table 3.2. The average target rates are 677kbps and 604kbps in DC1 and DC2 respectively; thus, users with low bandwidth do not benefit from Trickle. However, about half of the packet losses can be avoided for high-bandwidth users in YouTube by using Trickle!

3.5.2 Burst Size

The previous results show that Trickle effectively reduces the loss rate. In this section, we demonstrate that the reduction is achieved because Trickle sends much smaller bursts. We randomly sample 1% of the flows and collect tcpdump at the server to investigate the burstiness behaviour. Following the convention of prior burst studies [30, 80], we use packets instead of bytes to measure the burst size. We use the same definition of micro-burst as in [30]: a burst is a sequence of four or more data packets with an inter-arrival time less than or equal to 1 millisecond. We use four or more packets because TCP congestion control, e.g., slow start, naturally sends bursts up to three packets.

Figure 3.7 plots the burst sizes in DC1. Intuitively, most bursts in the two baseline groups should be about 43 segments (64kB) but in reality only 20% are that size. This mismatch is due to packet losses and TCP congestion control. After a packet is lost, TCP congestion control reduces cwnd and then gradually increases it again while streaming.
Figure 3.7: CDF of burst size in DC1.

new data. These \( cwnd \) changes fragment the intermittent 64kB application writes. The TCP time-sequence graph of a sample flow in the baseline1 group in Figure 3.8 illustrates this point. The black arrows represent data bursts and the red dots are retransmits. At around 77 seconds, a large tail drop occurs causing TCP to reduce \( cwnd \) and enter slow start. At the 80 second mark, TCP \( cwnd \) has increased to about 32kB. When ustreamer writes 64KB into the socket, TCP sends a 32kB burst, followed by a series of small bursts triggered by returning ACKs. The queue overflows when the \( cwnd \) reaches 64kB at 85 seconds. In this example, one loss event has caused 15 application writes to be fragmented into many smaller bursts.

Returning to Figure 3.7, the shrunk-block curve shows steps at 12, 23, and 34 segments corresponding to 16, 32, and 48 kB block sizes respectively. This suggests that the application and/or the kernel (TCP) is bunching up the data writes. We subsequently discovered that the ustreamer token bucket implementation does not pause the data write for intervals less than 100ms. For a large portion of the flows, ustreamer continues to write 16kB blocks.

In Trickle, 94% of bursts are less than 10 segments, because DC1 users have short
RTT such that most videos require less than a 10 segment window. As described in Section 3.3.4, Trickle lower-bounds the clamp to 10 segments to avoid slow loss recovery. The remaining 6% bursts over 10 segments are contributed by either high RTT, or high resolution videos, or other factors that cause Trickle to not clamp the cwnd. In summary, over 80% of bursts in Trickle are smaller than the other mechanisms.

Next we investigate the correlation of loss and burst size. For each burst size, Figure 3.9 shows the fraction of bursts that experience at least one retransmission. Trickle reduces the burst size and reduces the chance of losses with any given burst size. One reason is that the transmission is better ack-clocked and has a smaller queue occupancy, as we will show in Section 3.5.3. Interestingly, the two baseline groups suggest a high drop rate for bursts of size 11, 17, or 35. The shrunk-block group shows that half of the bursts of size 23 and 35 experience losses. We are not able to provide a satisfactory explanation due to the lack of information at the bottlenecks which are likely to reside at the last mile. We hypothesize that the phenomenon is produced by some common buffer configurations interacting with YouTube application pacing.

3.5.3 Queueing Delay

Sending smaller bursts not only improves loss rate; it may also help reduce the maximum queue occupancy in bottleneck links. It is certainly not uncommon for users to watch online videos while surfing the Web at the same time. Since networks today are commonly over-buffered [120], shorter queue length improves the latency of interactive applications sharing the link.

We evaluate the impact of queue length by studying the RTT measurements in TCP, because there is not enough direct information from the bottleneck queues. Recall that a RTT sample in TCP includes both the propagation delay and the queuing delay. Given that the four experiment groups receive similar loads, the propagation delay distribution in each group should be close. Each video stream often has hundreds to thousands of
RTT samples, partly because Linux samples RTT per ACK packet. To reduce the sample size, we use the smoothed RTT ($srtt$) variable at the end of the connection. Since $srtt$ is a weighted moving average of all the RTT samples, it should reflect the overall queuing delay during the connection.

Figures 3.10 and 3.11 plot the CDF of the $srtt$ samples for DC1 and DC2, respectively. In DC1, on average, the $srtt$ of connections in the Trickle group is 28% smaller than the connections in the baseline groups. In DC2, the improvement over baseline is only 7%. The reason recalls the analysis in Section 3.5.1: throttling is seldom activated on the slow links in DC2. But the distribution shows that the links in India are alarmingly over-buffered: 20% of the $srtt$ samples are over 1 second and 2% are over 4 seconds! While these videos are likely being streamed in the background, the interactive applications sharing the same bottleneck queue certainly will suffer extremely high latency.

In summary, for fast networks, Trickle connections experience lower queueing delays, which should improve interactive application latencies. For slow users, the solution is to
Figure 3.9: Frequency of losing at least one segment as a function of burst size in DC1.
use Trickle but to serve at a lower rate (lower resolution video).

### 3.5.4 Rebuffering

Rebuffering occurs when a reduction in throughput in a TCP streaming session causes receiver buffer starvation. When this happens, the video player stops playing video until it receives enough packets. The rebuffering rate is an important metric in video streaming as it reflects user experience watching videos.

YouTube has a built-in mechanism to provide real-time monitoring of video playbacks. During a playback, the video player sends detailed information about user interactions to the server. The information includes the timestamps of all rebuffering events in each TCP connection.

To quantify the user perceived performance of video streaming, we use rebuffering chance and rebuffering frequency as suggested by [102]. The rebuffering chance measures the probability of experiencing rebuffering events and is defined by percentage of flows that experience at least one rebuffering event. As the name suggests, rebuffering
Figure 3.11: CDF of the smoothed RTT ($srtt$) samples in DC2. The average $srtt$ of connections in Trickle is 7% and 1% lower than the connections in the baselines and shrunk-block, respectively.

<table>
<thead>
<tr>
<th></th>
<th>DC1</th>
<th>DC2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rebuff. freq. (1/s)</td>
<td>rebuff. chance (%)</td>
</tr>
<tr>
<td>baseline1</td>
<td>0.0005</td>
<td>2.5</td>
</tr>
<tr>
<td>baseline2</td>
<td>0.0005</td>
<td>2.5</td>
</tr>
<tr>
<td>Trickle</td>
<td>0.0005</td>
<td>2.5</td>
</tr>
<tr>
<td>shrunk-block</td>
<td>0.0005</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Table 3.3: A comparison of rebuffering frequency and rebuffering chance.

frequency measures how frequently the rebuffering events occur and is defined by $r/T$, where $r$ is the number of rebuffering events and $T$ is the duration of a flow. Table 3.3 shows the average of rebuffering metrics in DC1 and DC2. DC2 users clearly have a much worse experience than DC1 users. But in both data centers the rebuffering frequency and rebuffering chance are similar across all four groups, suggesting Trickle has negligible impact on the streaming quality.

Initially the results puzzled us, as we expected Trickle to improve rebuffering by reducing burst drops. To explain the results, we studied the correlation between rebuffering and various network metrics. We found that the bandwidth deficit, the difference be-
between the target streaming rate and the bandwidth, is the key factor in rebuffering. In both DC1 and DC2, among the flows that do not have sufficient bandwidth (positive deficit), 55% to 60% have experienced at least one rebuffering event. Another possibility is a user requesting a different resolution by starting a new connection.

However, the correlation between rebuffering and the retransmission rate is less than 5%. A browser typically buffers enough video so that losses rarely cause buffer under-run. An exception occurs when the available bandwidth is too low; in this case, Trickle cannot help, as the congestion is not caused by burst drops.

3.6 Discussion

We considered other solutions to rate limit video streaming. For example, a similar idea that requires no kernel TCP change is to set the TCP send socket buffer size [107]. In the case of YouTube, the ustreamer TCP send buffer remains auto-tuned [112] during the startup phase in order to send data as fast as possible. Upon entering the throttling phase, the buffer usually is already larger than the intended clamp value. Setting a new send buffer size is not effective until the buffered amount drops below the new size, making it difficult to implement the throttling. Others control the rate by dynamically adjusting the TCP receive window at the receiver or the gateway [85, 95, 116]. In contrast Trickle is server-based making it easier to deploy in a CDN.

Another approach is to use TCP pacing [138], i.e., pacing \( cwnd \) amount of data over the RTT. While this may be the best TCP solution to suppress bursts, it is more complex to implement. Moreover, studies [30, 49] show that Internet paths can absorb only small packet bursts. Our goal is to reduce large burst drops caused by disruptions to the TCP self clock, not to eliminate bursts entirely.

Delay-based TCP congestion controls, e.g., Vegas [32], may be an interesting way to avoid burst drops by detecting queues early. It may not directly address the problem be-
cause these algorithms still require proper ACK-clocking. Even so, we plan to experiment
Trickle with delay-based congestion controls in the future.

A non-TCP approach to consider is traffic shaping underneath TCP. To our knowl-
edge most software implementations, e.g., Linux tc, can not scale to support the num-
ber of connections of YouTube servers. Doing this may require dedicated hardware; in
contrast, Trickle is a relatively simple and inexpensive solution. DCCP \cite{91} supports
unreliable delivery with congestion control based on UDP. While DCCP is suitable for
video streaming, our work focuses on improving the current HTTP / TCP model.

Trickle is motivated by Alcock and Nelson \cite{15} who identifies a YouTube burst drop
problem in residential broadband and university networks. In addition, Rao et al. show
that popular browsers also throttle video streaming for YouTube and Netflix \cite{24}. The
bilateral throttling mechanisms sometimes result in packet bursts up to several MBs.
Blanton et al. studied the correlation between burst size and losses in TCP, finding that
bursts less than 15 packets rarely experience loss but large (over 100) bursts nearly always
do \cite{30}. Allman et al. evaluate several ways to mitigate bursts created by TCP \cite{22};
we did not attempt these methods because the bursts in YouTube are created by the
application, not TCP congestion control. At this point, Trickle is actively deployed on
YouTube servers and its performance is being monitored on various networks and devices.
We are hopeful that Trickle in YouTube can help reduce the burst stress in the Internet.

Trickle requires minimal sender-side changes, thereby permitting fast deployment at
the content providers. Using OpenTCP, a service provider can generalize Trickle to rate-
limit other kinds of traffic or similar transfers. For example, OpenTCP can be configured
to disable Trickle for mobile or advertisement videos. It can also generalize Trickle to
variable bit rate video streaming to serve individual encoded blocks at different rates.
Chapter 4

TCP Pacing in Data Center Networks

In this chapter, we study the effectiveness of TCP pacing in data center networks. We confirm that by avoiding bursty packet drops, paced TCP achieves higher throughput and better flow completion times than non-paced TCP. However, in certain network conditions, non-paced TCP will out-perform paced TCP. This means that for optimal performance, TCP pacing must be turned off and on dynamically. Because of the lack of a dynamic TCP tuning framework, network operators tend to avoid enabling TCP pacing entirely.

In what follows, in Section 4.1, we give reasons for studying TCP pacing. Section 4.2 describes the data center workload and compares it to streaming traffic. Section 4.3 presents our model showing the effectiveness of pacing. Finally, Section 4.4 includes the results of our experiments to confirm our model’s validity.
4.1 The Tortoise And The Hare: Why Bother Pacing?

Throughput and latency in the Internet are heavily influenced by the behaviour of TCP. Briefly stated, TCP is a window-based congestion control scheme: during each round-trip time (RTT), every TCP source transmits packets with total size equal to its congestion window. Most TCP variants, such as Tahoe, Reno, NewReno [54], BIC [133], and CU-BIC [66], focus on the evolution of the congestion window size over time, and ignore the details of how packets are injected into the network in sub-RTT time scales. However, ignoring sub-RTT burstiness can produce bursty traffic on high bandwidth networks [139] such as data centres. This, in turn, produces more queueing delays, more packet losses, and lower throughput [12].

TCP pacing [139] addresses the problem by ensuring that bursts of packets do not cause contention in the router buffers. Specifically, TCP pacing evenly spaces the transmission of a window of packets over an entire RTT, so that data are not sent in a burst. This approach allows the sender to increase its sending rate without creating bottleneck queues [92].

Using TCP pacing is becoming more relevant in the context of data center networks, as the ever-increasing link speeds mean greater difficulty designing routers with buffer sizes equal to the bandwidth-delay product (BDP). As a result, many data center networks use switches with small buffers. Yet small-buffer switches and small RTTs in data center infrastructures create a disparity between the total capacity of the network and the capacity of individual queues, a disparity large enough to make TCP’s congestion control scheme inefficient.

In addition, as we discuss in Section 4.2, the latency-sensitive, bursty traffic in distributed systems compels us to closely consider the impact of TCP pacing in data center environments. In such networks, an important performance criterion, besides the aver-
age latency, is the 99th percentile tail latency; i.e., the completion time of the slowest flows. For example, in a search engine data center, a search query is distributed among thousands of servers; the median latency has no bearing on the result because the engine has to wait for the slowest flows to finish. In such environments, short term unfairness in TCP causes tail latency to grow, an issue that pacing may be able to address.

In this chapter, we study the effectiveness of TCP pacing in inter-data center transactions, where RTTs are higher than 1ms and bounded by the geographical distance of data center sites. We confirm that by avoiding bursty packet drops, paced TCP achieves higher throughput than non-paced TCP. However, the benefits of using paced TCP diminish as we increase the number of concurrent connections beyond a certain threshold; if the number of flows goes beyond that point, non-paced TCP can perform better than paced TCP.

We define the Point of Inflection (PoI) as the point in terms of number of concurrent flows where non-paced TCP out-performs paced TCP; we quantitatively determine its lower bound in terms of link capacity ($C$), RTT and buffer size ($B$), as $\frac{C \times RTT}{B}$. Our results yield two new insights. First, it may not be appropriate to derive conclusions about pacing by studying a network with a fixed number of users and various buffer sizes. In fact, depending upon the BDP-to-Buffer ratio, the benefits of pacing varies. Second, increasing the number of flows produces inter-flow burstiness (bursts of packets belonging to different flows) which causes drop synchronization, leading to performance degradation. In previous studies, Kulik et al. [92] report throughput improvement over simulated satellite links and propose pacing over the entire lifetime of a TCP connection. However, the simulation study by Aggarwal et al. [12] argues that, contrary to intuition, pacing has a negative impact on performance. In Section 4.4, we determine whether our model can justify these this latter contradiction.

To validate our model, we implement TCP pacing in the Linux kernel and perform several experiments in a data center test-bed. We find that while the number of concur-
rent flows is below the PoI bound, pacing offers improvements on link utilization, drop rate, average and 99th percentile flow completion times. As the number of flows passes twice the PoI bound, however, these benefits are diminished.

4.2 Data Center Traffic

A data center is commonly defined as a site where multiple servers are co-located. However, cloud services may involve multiple geographically distributed data centers. These sets of sites are connected by dedicated links and sometimes used as complete replicas of the same service.\(^1\) In this chapter, we refer to data center traffic as workloads between multiple sites, with RTTs between 1 to hundreds of milliseconds bounded by the geographical distance of data center sites.

Unlike the long-lived and non-bursty streaming traffic on the Internet, the traffic patterns between data center nodes are bursty and mostly consist of short-sized flows because traffic in such networks belongs to distributed applications that perform scatter/gather type of communications, including MapReduce [43], Google File System (GFS) [59], Panasas [130] and pNFS [113]. For example, MapReduce uses Remote Procedure Call (RPC) protocol [118] for communication between services in distributed systems to request a service from a program located in a remote computer through network. Meanwhile, in GFS, a transfer service manages all data transfers, each with one master and several workers. The master distributes jobs to the workers; each job is divided into RPC flows with few mega bytes in size. The transfer service starts the flows and continue until the transfer is finished.

The bursty properties of TCP differ for the above cases and for conventional large-sized, streaming traffic. In streaming traffic, TCP by itself starts to pace its window due to the ACK-clocking dynamics [75, 100]. Specifically, the transmission of new packets

---

\(^1\)Replication is mostly used for reducing user latency and improving serving throughput [68].
is controlled by the stream of received ACKs and, hence, there are small increases in the window size. In a data center environment, each TCP flow is a long-lived connection between a client-server pair with many short-sized RPCs within it. In other words, a TCP connection serves several back-to-back RPC flows sharing the TCP parameters (such as congestion window). As a result, within a connection, each new RPC flow starts with the congestion window of the previously finished flow; thus, there is an obvious burst of up to a congestion window’s worth of packets at the beginning of each RPC.

In streaming, bursts can still happen but will be more subtle and less frequent and the bursty behaviour will only get worse when 10 GigE network interface cards are deployed in data centers. Moreover, because data center networks have a single administrative domain and data center sites are connected to each other via dedicated links, the network administrator can control cross traffic which thereby affects the paced traffic.

Researchers have studied communication patterns in data centers in terms of high-fan-in, high-bandwidth synchronized TCP workloads, commonly known as the TCP incast problem [37, 126]. Such work is orthogonal in the sense that we are focusing on per flow burstiness which impacts packet drops and flow completion times. TCP pacing is not tied to any specific workload or TCP version: rather, pacing is about mitigating bursts, and bursts can occur regardless of TCP variant. The two major causes of high latency in data center networks are packet loss and slow recovery. TCP requires three duplicate ACKs to retransmit a lost packet, a threshold designed to avoid spurious retransmission due to packet reordering in the network. But this method is too conservative for data center environments where most losses are recovered by (conservative) timeouts. In such cases, TCP pacing may be a better option. The potential benefits of TCP pacing in data center environments include: (i) better link utilization on small-buffer routers, (ii) better short-term fairness among flows of similar RTTs by improving the worst flow latency, and (iii) smaller drop rates. Simply reducing the retransmission time-out (RTO) often makes no significant improvement since RTO activates a series of actions in the kernel,
slowing the sending speed.

Finally, TCP is slow to recover from losses incurred during short transfers [48], thereby making loss occurrence in data center transfers highly undesirable. That being said, TCP has evolved to the point where it can deal with congestion in the right conditions. In particular, bulk TCP flows can recover from packet losses quickly, and they can utilize most of the available bandwidth as long as the routers and switches have appropriate amounts of buffering (which is not always the case at data centers). However, these bulk flows can have a significant negative impact on small, latency-sensitive flows due to queuing latency and packet losses induced by the bulk flows. Thus, when evaluating TCP pacing, we need to consider the throughput and latency of the bulk flows and also the latency of small flows sharing the network.

### 4.3 Modeling the Effectiveness of Pacing

Most TCP variants inject several packets back-to-back, wait for an entire RTT to acknowledge receipt, adjust the window size and repeat. TCP pacing, on the other hand, avoids sending bursts of packets by spreading packet injections over the course of one RTT, making the traffic smoother. Thus, TCP pacing is believed to be an effective way to mitigate the impact of traffic burstiness [139].

An interesting phenomenon occurs when we increase the number of flows in the network. Even though pacing removes the burstiness resulting from back-to-back packet injections in one flow, increasing the total number of flows creates another form of burstiness: packets belonging to different flows start creating bursts in the aggregated traffic. Such bursts worsen as we increase the number of flows such that, after a certain point, the impact of pacing goes away completely, and the performance begins to drop; performance becomes worse than when we have no pacing in the network.

Consider a network with a bottleneck link capacity $C$ (packets/s) and buffer size
$B_{\text{max}}$ (packets) shared among $N$ concurrent flows with the same round-trip time $RTT$ (s).\textsuperscript{2} Previous models of TCP mostly make the simplifying assumption that $N$ and $C$ are large enough so that the average window size, $\bar{W}$, does not depend on $N$ [109,132]. In our model, we explicitly avoid such simplification so that we can derive a bound on the number of flows at which the benefits of pacing diminishes.

We consider two identical systems, one with non-paced and one with paced TCP. At steady state, on average, each flow is sending $\bar{W}$ packets in every $RTT$. Further, assume $RTT$ is divided into discrete time intervals of equal size $\delta$.

We claim that for any given buffer size, the benefit of using pacing will be less pronounced as we increase $N$. In a perfectly paced world, every flow’s packets are distributed evenly among the entire $RTT$. Hence, the probability of having one packet at time interval $\delta$ is a Bernoulli trial with $p = \frac{\bar{W} \times \delta}{RTT}$. Now, consider the aggregate of $N$ paced flows and define random variable $X$ as the total number of packet arrivals in interval $\delta$. $X$ is a binomially distributed random variable $X \sim B(N, p)$. Returning to our network, $X$ represents the burst of inter-flow packets in $\delta$ time interval. As the number of concurrent flows ($N$) increases, the fair share of each flow decreases and so does $\bar{W}$ and, hence, $p$. However, the probability distribution of $X$ becomes larger as $N$ increases, causing an increase in burstiness.

Now consider the following two systems: the best case of non-paced traffic (BNP), and the worst case of paced traffic (WP). Both systems contain the sequence of bursts of packets to the bottleneck link during one $RTT$. The best case scenario of non-paced traffic happens when $RTT$ is divided into equal intervals of $\Delta_1 = \frac{RTT}{N}$ and each flow sends $\bar{W}$ packets at the beginning of each interval. Here, we are making a simplifying assumption that each non-paced flow transmits a burst size of average length $\bar{W}$ packets clustered together. This assumption is based on the clustering of packets in non-paced

\textsuperscript{2}The $RTT$ here stands for the latency between data center sites due to their geographical distance and thus the homogeneous assumption is valid.
TCP [12].

In WP, the worst case scenario of paced traffic, $RTT$ is divided into equal intervals of $\Delta_2 = \frac{RTT}{W}$ and each flow sends one packet at the beginning of each interval. The packets from different flows make a burst of size $N$. As in BNP, we are making the realistic assumption that all paced flows are intermixed. The worst case of paced traffic happens when packets of each flow are equally spaced through the entire $RTT$ with interval $\Delta_2$.

Below we show that WP has better performance in terms of queue occupancy than does BNP as long as:

$$N < \bar{W} = \frac{C \times RTT}{B_{max}} \quad (4.1)$$

Up to this point, paced TCP out-performs non-paced TCP; past this point, however, the benefits start to diminish. Since we are comparing the best and worst case scenarios here, Equation 4.1 proves the following lower bound on the PoI.

**Theorem 1** In a network with $N$ flows, bottleneck capacity $C$, and buffer size $B_{max}$, the lower bound on the Point of Inflection, the number of flows at which non-paced TCP starts to out-perform paced TCP, is:

$$N^* = \Omega\left(\frac{C \times RTT}{B_{max}}\right) \quad (4.2)$$

In both systems, at steady state, each flow sends $\bar{W}$ packets at every $RTT$; thus, $C \times RTT = N \times \bar{W}$. The burst size in BNP and WP is $W$ and $N$, respectively.

In BNP, when the burst size of non-paced traffic is less than or equal to the available buffering capacity, there is no need to pace packets within the congestion window. Thus:
Similarly, in WP, pacing is out-performed by a non-paced system if:

\[
\begin{align*}
\frac{N}{C \times RTT} &\geq B_{\text{max}} \\
\frac{C \times RTT}{N} &\geq B_{\text{max}} \\
\frac{C \times RTT}{B_{\text{max}}} &\geq \bar{W}
\end{align*}
\]

(4.4)

In other words, when \( N = \bar{W} = \frac{C \times RTT}{B_{\text{max}}} \), WP and BNP become similar systems. Thus, to take advantage of pacing we should choose the number of flows so that \( N < \bar{W} = \frac{C \times RTT}{B_{\text{max}}} \).

### 4.4 Experiments

To validate our model, we perform several experiments using TCP pacing-enabled Linux kernels. We evaluate the impact of pacing on bottleneck link utilization, loss rate, drop synchronization, and average and 99th percentile flow completion times. We also demonstrate the existence of the PoI and the accuracy of our model in estimating the lower bound of PoI. In what follows, Section 4.4.1 describes functional tests to ensure the correctness of pacing implementation, Section 4.4.2 explains the setup for the experiments, Section 4.4.3 illustrates the basic advantages and disadvantages of pacing, and Section 4.4.4 generalizes the previous section and presents the results of a more realistic
model consisting of multiple flows.

### 4.4.1 Functional Test

Below, we describe a simple test to ensure the functionality of pacing implementation. While Section 4.4.4 extends the experiment to a more realistic real-world setting, here our functional test involves two machines: a client and a server connected through a router with large buffers with a 1 Gbps link. The latency between the client and the server is 30 ms. We use Netperf [82] to generate traffic between the two machines. Netperf is a benchmark that is close to the RPC-based communication model used in data centers. We configure Netperf so that the client sends 100 B requests to the server and the server replies with 1 MB responses. As soon as one response is completed, the client sends another request. As explained above, this scheme resembles the data center traffic pattern and is different from the widely used streaming traffic generators, such as Iperf [5], where there is only one long-lived connection. We repeat the same experiment twice: TCP pacing is either enabled or disabled at the server machines. Note that the client hosts always disable TCP pacing since there is no need to pace the ACK packets.

Figure 4.1 (a) illustrates the time-sequence graph plotted for paced and non-paced cases. The Y-axis represents packet sequence numbers, and the X-axis represents time. After the slow start phase, the congestion window is fully open, and each new RPC flow starts with a large congestion window. In the non-paced case, each 1 MB response is sent in one large burst at the beginning of RTT followed by a pause waiting to receive the ACK packets. In the paced case, packets are sent smoothly and evenly through the entire RTT.

Figure 4.1 (b) shows the CDF of packet inter-transmission times in both paced and non-paced cases after 100 RPC transfers. Here, the X-axis is the packet inter-transmission time in logarithmic scale and the Y-axis is the cumulative distribution function. Note that each 1 MB transfer is roughly 667 MTU-sized packets. Smoothing 667 packets
in an interval of 30 ms means the average packet inter-transmission time in the paced cases should be roughly 45 µs; in fact, in Figure 4.1 (b), most packets have an inter-transmission time of around 50 µs. In non-paced cases, however, packets are mostly back-to-back with an inter-transmission time of 12 µs (the rate to transmit MTU-sized packets back-to-back at the speed of 1 Gbps).

### 4.4.2 Setup of Experiments

Figure 4.2 illustrates the topology used for our experiments. TCP flows travel between a set of senders and receivers through a single bottleneck link. The bottleneck link switches have configurable buffers and use FIFO scheduling and drop tail buffer management. To monitor the bottleneck link utilization and drop rate, we read the port statistics of the switches every 12 seconds. The default capacity of the bottleneck link is 10 Gbps and the capacity of access links is 1 Gbps. We use the ANUE hardware delay emulator [1] to emulate delay in the network. ANUE receives data and stores them on a large buffer, transmitting the stored data from another port after a specified delay time has passed. The direction of flows in this topology is from server hosts to client hosts. As a result, switch$_2$ is the congestion point.
We perform experiments using TCP CUBIC, NewReno, and BIC congestion control algorithms. We use a Netperf request size of $20B$ and a response size of $1MB$, $2MB$, or a combination of $100B$ to $1MB$ flow sizes. Throughout our experiments, we limit the bottleneck bandwidth to 1, 2, and 3 $Gbps$ and use the delay emulator to emulate various ranges of RTTs, from $1ms$ to $100ms$. Our results share the same conclusions and in the following sections we use 1 $Gbps$ bottleneck bandwidth, 1 $MB$ flow sizes, 30 $ms$ RTT, and TCP CUBIC congestion control algorithm,\(^3\) unless otherwise stated. The run time for each experiment is 300 seconds. To obtain flow completion times, we use `tcpdump` to record packets on all hosts. We define RPC completion time as the difference between the time that the TCP sender starts to send the first byte of each RPC flow to the time it receives the ACK of the last byte.

### 4.4.3 Base-Case Experiment

We start with simple experiments with only one and two flows to demonstrate the potential effectiveness of TCP pacing in link utilization, packet drop, and flow completion time. While most of the results are trivial, they help us to understand the results in later sections.

\(^3\)TCP CUBIC is the default congestion control algorithm in Linux kernels since version 2.6.19.
First, consider two simple cases: (i) having no congestion in the bottleneck link by limiting the number of concurrent TCP connections to only one from $server_1$ to $client_1$, and (ii) introducing congestion in the bottleneck link by adding a second concurrent connection between $server_2$ and $client_2$.

We limit the buffer size at both switches to 1% of the BDP; i.e. 64 KB. In case (i), the capacity of the access links and the bottleneck link are equal; thus, there will be no congestion at the bottleneck link. However, in case (ii), the ratio of access to core bandwidth is two, and the bottleneck link is congested.

The two simple cases illustrate two intuitive laws of TCP pacing: (i) when there is no congestion, and hence, no loss at the bottleneck link, TCP pacing performs worse than non-paced TCP simply because there is no need to pace packets, and (ii) TCP pacing improves performance by mitigating loss events in the buffers when the bottleneck link is congested.

**Under-subscribed bottleneck:** Figure 4.3(a) compares the bottleneck link utilization of paced and non-paced cases through the entire 300 seconds of the experiment. Note that because there are no packet drops, non-paced TCP achieves 38% higher utilization
than paced TCP. This is easily explained by looking at flow completion times, illustrated in Figure 4.3(b). In our setup, the transmission time of a 1 MB RPC is 8 ms. Nearly all flows finish within one RTT (39 ms in the figure) in non-paced TCP because after the initial slow start, the congestion window becomes fully open; the following RPC calls take advantage of this fully open congestion window and send the whole 1 MB in one burst. But because TCP pacing adds delay between packets, the paced flows need 2.1 RTTs (63 ms in the figure) to finish. Even though the congestion window is fully open, the last batch of packets is sent at the end of the RTT; the sender needs another RTT to receive the ACK packets and initiate the next RPC call.

**Over-subscribed bottleneck:** We introduce congestion in the network by adding another concurrent TCP connection between a new client-server pair: \((server_2, client_2)\). Figure 4.4(a) compares paced and non-paced cases through the entire 300 seconds. With congestion in the network, paced TCP achieves higher utilization than non-paced TCP. Non-paced TCP is experiencing around 0.5% of packet drops at the bottleneck queue. As explained in Section 4.2, this phenomenon is highly undesirable in data center networks as TCP loss recovery is not efficient for small latency-sensitive transfers. The paced flows have almost zero drops. As shown in Figure 4.4(b), their flow completion time is still mostly two RTTs, whereas non-paced flows have considerably larger average and 99th percentile flow completion times.

### 4.4.4 Multiple Flows

In this section, we consider a more realistic case whereby many flows are sharing the bottleneck link. Our results confirm the model proposed in Section 4.3; for any given buffer size, as we increase the number of concurrent connections beyond the PoI, non-paced TCP out-performs paced TCP. In Section 4.3, we quantitatively determined a lower bound for PoI \((N^*)\) in terms of link capacity \((C)\), \(RTT\) and buffer size \((B)\), as

\[
\frac{C \times RTT}{B}.
\]
Figures 4.5, 4.6, and 4.7 illustrate the above claim when the buffer size is 64, 128 and 256 KB, respectively. We increase the total number of concurrent flows from 4 to 100 using all four client-server pairs in Figure 4.2 by distributing the load evenly across hosts. The capacity of the access links and the bottleneck link are both equal to 1 Gbps. As a result, the bottleneck is congested and as explained in Section 4.4.3, we expect paced TCP to have higher link utilization and lower flow completion times. However, as discussed in Section 4.3, this also depends on the BDP-to-Buffer ratio.

Figure 4.5(a) depicts the bottleneck link utilization versus the number of concurrent flows sharing the bottleneck link. Paced traffic achieves higher utilization than non-paced traffic when the number of flows is below 80 (the PoI). Pacing also has better average and 99th percentile of RPC completion times, as shown in Figures 4.5(c) and (d), respectively.

The benefits diminish as we increase the number of flows beyond 80. In this setup the BDP-to-Buffer ratio \( N^* \) is 58 flows, which is the lower bound on taking advantage of pacing, as proposed in Section 4.3. Note that we might be able to achieve some improvement in performance with pacing when the number of flows is greater than this lower bound, as shown in Figure 4.5, but the benefits have already started to diminish.
Figure 4.5: Studying the effect of number of flows sharing the bottleneck link when the buffer size is 1.7% of BDP (64 KB) and the lower bound on inflection point \( (N^*) \) is 58 flows. (a) Bottleneck link utilization. (b) Bottleneck link drop rate. (c) Average RPC completion times. (d) 99\(^{th}\) percentile RPC completion times.

Repeating the same experiment with different buffer sizes, as illustrated in Figures 4.6 and 4.7, makes it clear that for every given buffer size, as the number of flows increases, the benefits of pacing become less pronounced. Figure 4.6 illustrates the same experiment as Figure 4.5 but here the available buffer size has been doubled. Interestingly, the PoI in Figure 4.6 (48) is almost half of the PoI in Figure 4.5 (80). In Figure 4.7, when the buffer size is doubled, the PoI is reduced to 19 flows.

All three experiments are performed with buffer sizes of 1.7%, 3.4% and 6.8% of BDP, but measuring the drop rate at the bottleneck link shows that 6.8% of BDP is enough to have no packet drops even with non-paced traffic. Hence, they constitute a spectrum of tiny, small and large buffers in the context of this experiment.
In the experiments shown in Figures 4.5 and 4.6, we see that paced traffic always has a lower drop rate than non-paced traffic, even after the PoI. However, the synchronization of drops in paced TCP causes many flows to experience a loss event, leading to a dramatic degradation in performance.

An interesting question is the accuracy of the lower bound derived in Section 4.3. In other words, for various buffer sizes, what happens if we set the number of flows to be exactly $N^* = \frac{C \times RTT}{B_{max}}$? As we discussed in our model, $N^*$ is the lower bound on the PoI. We therefore, expect to see paced and non-paced TCP perform about the same when the number of flows is exactly $N^*$. Figure 4.8 confirms this by plotting the bottleneck link utilization and flow completion times for various buffer sizes when the number of
flows are chosen according to Equation 4.1. As expected, paced and non-paced cases perform similarly. Thus, the lower bound $N^*$ is a valid threshold for the PoI. Moreover, throughout the experiments presented here and in our technical report [60], we observe that the PoI is always less than twice the $N^*$ bound. In other words, in our experiments, the following inequality always holds: $N^* \leq PoI \leq 2 \times N^*$.

As mentioned in Section 4.1, our BDP-to-Buffer ratio rule can justify the seemingly contradictory conclusions in previous works. For example, Aggarwal et al. [12] study a network with 50 long-lived flows with BDP of 1250 packets and buffer size of 312 packets. Our results suggest $PoI = 8$ flows is the point of inflection in such network and increasing the number of flows to 50 diminishes pacing’s effectiveness. On the other hand, Kulik
et al. [92] study networks with BDP of 91 packets and buffer sizes of 100 and 10 packets with a single TCP flow which lies within our bound.

### 4.4.5 Drop Synchronization

In this section, we study the impact of inter-flow burstiness on drop synchronization. As mentioned in Section 4.4.4, in our experiments, paced traffic always experiences lower drop rate compared with non-paced traffic. For example, Figure 4.9(a) compares the drop rate of paced TCP with non-paced TCP in the experiment illustrated in Figure 4.5. Even after the PoI, paced TCP continues to have a lower drop rate compared with non-paced TCP. However, as we will show in this section, the synchronization of drops in
Chapter 4. TCP Pacing in Data Center Networks

Figure 4.9: (a) Bottleneck link’s drop rate. (b) The clustering probability matrix for 25 non-paced flows. (c) The clustering probability matrix for 25 paced flows. The color of element \((i, j)\) corresponds to the probability that packets from flow \(i\) are followed by packets from flow \(j\).

Paced TCP causes many of the flows to experience the loss event; hence, a dramatic degradation in performance. The negative effect of drop synchronization in the slow start phase of paced TCP has been observed and reported in previous studies of TCP pacing \([12]\). In the following, we demonstrate that drop synchronization in paced TCP is not tied to slow start phase and is a function of number of flows in the network: increasing the number of concurrent flows sharing the bottleneck increases the inter-flow burstiness, and as a result, increases the chance of many flows experiencing the drop event.

To provide the intuition for the drop synchronization phenomenon and to verify the simplifying assumptions in Section 4.3, we first measure the probability of packet clustering in paced and non-paced cases. Figures 4.9(b) and (c) illustrate the clustering matrix for one of the hosts in our topology when there are 100 flows in the bottleneck link (25 flows from each host).\(^4\) Each figure shows a 25 \(\times\) 25 colored matrix where the color of each element \((i, j)\) in the matrix corresponds to the probability that a packet from flow \(i\) is followed by a packet from flow \(j\). In Figure 4.9(b), the diagonal of the matrix has the lightest color (largest probability), and hence clearly shows that in the non-paced case, it is highly likely that packets from each flow are clustered to each other. On the

\(^4\)This corresponds to the very last data point in Figure 4.9(a).
other hand, the diameter of the matrix in Figure 4.9(c) has the darkest color (lowest probability) which shows that in paced case it is more likely that packets from different flows are intermixed with each other.

In order to measure the effect of drop synchronization, we use a NetFPGA router to count the number of flows that are affected by drop events at the bottleneck router. The NetFPGA router is configured with the “event capturing module” of the NetFPGA router design [27] which supports instrumenting the router’s output queues. We modify the event capturing module to report the flow ID of flows that experience drop events. Figure 4.10(a), (b), and (c) illustrate the CDF of flows that are affected by a packet drop event with paced and non-paced TCP when the total number of flows in the bottleneck link is 48, 96, and 384, respectively. In the case of 48 flows, illustrated in Figure 4.10(a), paced and non-paced TCP behave roughly similar to each other, with paced TCP having slightly larger probability of drop synchronization. Increasing the number of flows to 96 (Figure 4.10(b)) increases the chance of drop synchronization in paced flows. If the number of flows are further increased to 384, the synchronization effect becomes more dominant. Figure 4.10(c) illustrates that 300 flows (78% of total number of flows) are

---

5When a packet arrives, departs, or is dropped at an output queue, the current clock of the NetFPGA which has an 8 ns granularity, is recorded. Multiple events are then collected in a single event packet which is periodically sent out a specified router port.
affected by 80% of drop events in paced TCP; but, the same number of flows are only affected by 10% of drop events in non-paced TCP.

4.5 Conclusions and Future Trends for TCP Pacing

This chapter studied TCP pacing in data center networks and presents a unifying model for the general effectiveness of pacing. We show that for a given buffer size, as the number of concurrent flows are increased beyond a certain bound, the benefits of using paced TCP will start to diminish. We provide a lower bound for this point of inflection in terms of link capacity ($C$), $RTT$ and buffer size ($B$), as $\Omega(C \times \frac{RTT}{B})$. Moreover, we demonstrate that increasing the number of flows produces inter-flow burstiness (bursts of packets belonging to different flows) which causes drop synchronization and hence performance degradation. We validate our model using a novel and practical implementation of paced TCP in the Linux kernel and perform several experiments in a data center test-bed.

We showed that TCP pacing successfully mitigates the burstiness of TCP flows as long as the total number of concurrent flows falls below the PoI. In a large data center with thousands to millions of concurrent flows, it is almost impossible to guarantee that the number of active flows will always be less than any threshold. Therefore, providers are reluctant to enable pacing despite its proven effectiveness. However, if we have a system that can measure the number of active flows, we can dynamically switch between paced TCP and regular TCP to ensure the system is always at its peak performance. In Chapter 6 we show how OpenTCP can be used to enable/disable TCP pacing.

In the networking literature, TCP pacing usually refers to per-flow smoothing of congestion window size of packets within one RTT. As discussed in this chapter, having many paced flows introduces inter-flow burstiness and drop synchronization which hurts the effectiveness of pacing. We propose applying per-host TCP pacing on top of per-flow pacing to reduce the inter-flow bursts and drop events. The objective of per-host pacing,
or more precisely per-egress port pacing, is to smooth the aggregate traffic leaving an egress port. To that end, and by knowing the rate $R_i$ of each flow exiting an egress port of a host, a per-host pacer applies an inter-transmission gap of $\delta = \frac{MTU}{\sum_i R_i}$ between all packets. As a result, per-host pacing smooths the transmission of aggregate of flows leaving an egress port and alleviates the impact of inter-flow bursts. Essentially, at steady state, this results in having almost zero packet drops at the bottleneck link. We leave the study of per-egress port pacing to interested researchers in the field.
Chapter 5

OpenTCP: TCP Adaptation in Data Center Networks

This thesis takes the first step in addressing the need for a systematic way of adapting TCP to network and traffic conditions. In the past two chapters, we presented two examples of TCP adaptation and provided motivation for OpenTCP. In this chapter, we describe OpenTCP’s design decisions (Section 5.1). We then explain OpenTCP’s architecture (Section 5.2) and its operation (Section 5.3).

5.1 OpenTCP Design Principles

We introduce OpenTCP as a system for dynamic adaptation of TCP based on network and traffic conditions in Software-Defined Networks. In this section, we present OpenTCP’s design decisions.

Use Software-Defined Networks: OpenTCP mainly focuses on internal traffic in SDN-based data centers for four reasons: (1) the SDN controller already has a global view of the network (such as topology and routing information), (2) the controller can easily collect relevant statistics (such as link utilization and traffic matrix), (3) OpenTCP is deployable as a straightforward controller application in SDNs, and (4) the operator
can easily customize end-hosts’ TCP stacks.

Even though it has been designed for SDNs, OpenTCP can be deployed in traditional networks by emulating a centralized controller, and collecting network statistics. This is the approach we use to implement OpenTCP in a data center environment (as we will explain in Section 5.4). Our implementation of OpenTCP is based on OpenFlow [98], even though OpenTCP should work with any SDN.¹

**Network operator is in charge:** OpenTCP is designed to optimize TCP performance while keeping the system stable. However, all major decisions are made by the network operator, including what OpenTCP optimizes (e.g. reduced flow completion times, or maximized utilization of a specific link), the constraints that must be satisfied (e.g. stability constraints), and mapping OpenTCP updates onto changes in individual TCP sessions. By defining congestion control policies (explained in Section 5.3.2) the operator can update OpenTCP and adapt individual TCP sessions. As part of the congestion control, the operator can define a set of conditions that need to be satisfied before any modifications are made to specific TCP sessions. This, along with a monitoring component that measures the changes in the network create a controlled environment, allowing the operator to modify TCP without the risk of making the network unstable.

**Modify TCP in end-hosts:** A key decision is how we want OpenTCP to change TCP’s behaviour. We can either (i) implicitly change TCP’s behaviour by modifying the feedback provided to TCP sources (for example through ECN bits), or (ii) explicitly change it by having an agent running in the end-host that can update TCP on demand. The first option requires no change in the end-hosts, but gives us little flexibility to make the changes we want. As long as our end-hosts are under a single administrative control, changing them is not difficult. This is true for example in a data center environment. As a result, we have designed OpenTCP under the assumption that we can change end-

¹We do not tie OpenTCP to a specific OpenFlow standard as OpenTCP relies on basic features of a SDN and augments them to achieve its goals.
hosts and install a lightweight agent that can explicitly change TCP sessions. Taking advantage of today’s extendible TCP implementations, we can easily modify TCP and even introduce completely new congestion control schemes when needed.

**Keep the overhead to minimum:** OpenTCP consumes bandwidth to collect statistics and send hints to the end-hosts. It also consumes processing resources in the controller and the end-hosts. As a key factor in our design, we have kept this overhead to a minimum by utilizing restricted control paths and light-weight end-host instrumentation, and by summarizing hints.

**Two time scale control:** TCP’s congestion control updates occur on a time scale of network round-trip time (RTT). This represents sub-milliseconds in data center environments. OpenTCP modifies TCP sources on a time scale, $T$, which is several orders of magnitude slower than RTTs. The exact value of $T$ is a choice of the network operator. In order to keep the network stable, $T$ needs to be orders of magnitude slower than the network RTT. Intuitively, by slowly modifying TCP parameters, OpenTCP gives each TCP session enough time to become stable before updating its state. Moreover, by updating the system in a time scale which is orders of magnitude slower than the network RTT, OpenTCP can ensure a low overhead on the SDN controller.

### 5.2 OpenTCP Architecture

OpenTCP collects data regarding the underlying network state (e.g., topology and routing information) as well as statistics about network traffic (e.g., link utilization and traffic matrix). Then, using this aggregated information and based on policies defined by the network operator, OpenTCP decides on a specific set of adaptations for TCP. Subsequently, OpenTCP sends periodic updates to the end-hosts that, in turn, update their TCP variant using a simple kernel module. Figure 5.1 presents the schematic view of how OpenTCP works. As shown, OpenTCP’s architecture has three main components:
Figure 5.1: The Oracle, SDN-enabled switches, and Congestion Control Agents are the three components of OpenTCP.

The *Oracle* is an SDN controller application that lies at the heart of OpenTCP. It collects information about the underlying network and traffic. Then, based on *congestion control policies* defined by the network operator, it determines the appropriate changes to optimize TCP. Finally, it distributes update messages to end-hosts.

*Network switches* are typical SDN-enabled switches that in addition to switching packets, can collect traffic statistics and are used to disseminate update messages to the end-hosts.

A *Congestion Control Agent (CCA)* is a kernel module installed in each end-host. CCAs receive update messages from the Oracle via the network switches and are responsible for modifying the TCP stack at each host.

### 5.3 OpenTCP Operation

OpenTCP goes through a cycle composed of three steps: *(i)* data collection, *(ii)* optimization and CUE generation, and *(iii)* TCP adaptation. These steps are repeated at a pre-determined interval of $T$ seconds. As explained in Section 5.1, to keep the network stable, $T$ needs to be several orders of magnitude slower than the network $RTT$. In the rest of this section, we will describe each of these steps in detail.
5.3.1 Data Collection

OpenTCP relies on two types of data for its operation. First, it needs information about the overall structure of the network, including the topology, link properties like delay and capacity, and routing information. Since this information is readily available in the SDN controller, the Oracle can easily access them. Second, OpenTCP collects statistics about traffic characteristics by periodically querying flow-tables in SDN switches. In order to satisfy the low overhead design objective, our implementation of OpenTCP relies on the built-in aggregation features in SDN switches [98] to collect link level statistics. In principle, SDNs can collect flow level statistics using appropriate sampling techniques with the right update frequency [122]. The operator can choose to collect only a subset of possible statistics to further reduce the overhead of data collection. Later, we will show how this is done by using congestion control policies.

Example: Consider an SDN where each switch collects the following statistics for every link $l$: link utilization $U(l)$, number of packets transferred $P(l)$, number of live flows $F(l)$, and drop rate $D(l)$. The Oracle can directly access these statistics by sending queries to SDN switches. There are other metrics that the Oracle can measure (or estimate) even though they cannot be directly read from SDN switches. For instance, the Oracle can estimate the ratio between short-lived flows and all flows going through each link $l$, denoted by $\xi(l)$. For this, the Oracle collects packet counts for a sample of flows collected from the entire network. Since the Oracle knows the routing tables for all flows, it can easily find a good sample set. Then, by focusing on the flows going through a specific link $l$, and finding the ratio of short-lived vs. all flows in this set, the Oracle can make an estimate of $\xi(l)$. Note that since flow sampling happens only once for each link, the amortized cost of this step is low.
Figure 5.2: The Oracle collects network state, and statistics through the SDN controller and switches. Combining those with the CCP defined by the operator, the Oracle generates CUEs which are sent to CCAs sitting on end-hosts. CCAs will adapt TCP based on the CUEs.

### 5.3.2 CUE Generation

After data collection, we must determine what changes to make to TCP sources. This decision might vary from network to network depending on the network operator’s preferences. In OpenTCP, we formalize these preferences by defining a *Congestion Control Policy (CCP)*. The network operator provides OpenTCP with the CCP. At a high level, CCP defines which statistics to collect, what objective function the operator should optimize, what constraints OpenTCP must satisfy, and how the proposed updates should be mapped onto specific changes in TCP sources. Based on the CCP defined by the network operator, OpenTCP finds the appropriate changes and determines how to adapt the individual TCP sessions. These changes are sent to individual CCAs in messages we call Congestion Update Epistles (CUEs). The CCAs which are present in individual end-hosts use these CUEs to update TCP. Figure 5.2 shows how CCPs are integrated into TCP adaptation process.

**Statistics mask:** SDN switches can collect a wide range of statistics from the underlying network, including link utilization, number of flows per link, packet and byte counts, drop rates, etc. The SDN controller can query the switches directly, retrieve these statistics
and, if needed, aggregate them. In OpenTCP, the network operator can define the subset of available statistics by defining a bit mask called *statistics mask*. This mask tells the Oracle which statistics are needed. Focusing on a subset of all possible statistics, we can significantly reduce data collection overhead.

**Optimization:** The network operator can direct OpenTCP to solve an *optimization problem* defined as a set of variables, an *objective function*, and a *set of constraints*. This can be a general optimization problem (e.g. any convex optimization problem) as long as it can be defined in a canonical form that can be solved automatically. In our design of OpenTCP, we use a linear programming problem which is easily defined in the following canonical form:

\[
\begin{align*}
\text{maximize} \quad & c^T x \\
\text{subject to} : \quad & Ax \leq b
\end{align*}
\] (5.1)

Here, \(x\) represents the vector of optimization variables, \(c\) and \(b\) are vectors of known coefficients and \(A\) is a known matrix of the coefficients. The coefficients in this linear program are network and traffic related constants available to the Oracle. The objective function is directly related to what the operator is trying to achieve by changing TCP. It can be as simple as minimizing packet drops, or it can be a complex function of the input variables and coefficients. If the operator wants to adapt TCP parameters or switch to a different variant of TCP without any specific objective function in mind, she can define this optimization problem as *null*. We also define a set of constraints that are used as part of the optimization problem. If the Oracle cannot satisfy these constraints, no CUEs will be generated, and the CCAs will revert to a default version of TCP.

**Mapping rules:** Once the optimization problem is solved, we need to map its output in a way that CCAs can understand. We use the *mapping rules* defined in CCP whereby each mapping rule takes the form:

\[\text{scope:} \quad \text{(condition)} \Rightarrow \text{command(s)}\]
Figure 5.3: Sample CCPs and CUEs. (a) Here the goal is to minimize flow completion times by modifying the initial congestion window size. (b) This CCP switches from the default TCP variant (TCP Reno) to CUBIC as long as all links have a utilization below 80%.

The scope of the mapping rule is a subset of all end-hosts (or CCAs) to which this rule applies. When the Oracle generates CUEs it will use this rule only for the nodes within its scope. The condition in the rule is based on variables known to the CCA and the Oracle. If the condition contains any variables to which only the Oracle has access (for example network statistics) the Oracle replaces those with their values before sending the CUEs. Each CCA will also check this condition before applying the rule. This condition can be used to ensure changes happen only when certain stability conditions are satisfied. The CCA can fall back to a default setting (to keep the system stable) if the condition stated in the mapping rule fails. If the operator wants to apply the rule at all times, it can set the condition to be “TRUE”. Finally, each mapping rule has a command or a set of commands which the CCAs execute after checking the condition. This can be as simple as changing the value of a single parameter (like RTO), or as complex as switching between TCP variants.
**CCP Example:** Figure 5.3(a) shows a sample CCP. Here we assume there are five possible statistics that the Oracle can collect: link utilization $U(l)$, number of packets transferred $P(l)$, number of live flows $F(l)$, drop rate $D(l)$, and the ratio between short-lived flows and all flows $\xi(l)$. The statistics mask in this example is set to $[1, 0, 1, 1, 1]$; therefore the Oracle does not collect statistics for $P(l)$. Assume the network operator wants to reduce flow completion times by changing the initial congestion window size denoted by $w_0$. In this case, there is only one optimization variable. Note that in an extreme case, we can define one variable per CCA rather than one variable for all CCAs. This increases the optimization complexity, but might give better results in environments where there are significant differences between the nature of jobs running on different nodes. For the sake of simplicity we stick to one parameter in this example. The goal of the operator is to reduce flow completion times. We assume that increasing the initial congestion window does not have a major impact on long-lived flows. For short-lived flows, the improvements in flow completion times are not a linear function of $w_0$ (see Figure 5.3). However, if we limit the range of $w_0$ (say to a range of 4 to 20 segments), using linear regression, we can approximate the flow completion time for short-lived flows by $k \times (w_0 - w_d)$, where $k$ is constant and $w_d$ is the default congestion window. Based on this approximation, we can represent the objective function as “maximize $w_0$”.

In addition to determining the objective function, the operator has to define the constraints for the optimization function. A reasonable constraint would be to keep the utilization less than 70%. As this is the utilization observed every 1-10 minutes, we need to be conservative. Limiting the bandwidth ensures congestion control windows are not arbitrarily increased, causing a large drop rate in the system. We can state this constraint as: $U(l) \times (1 + w \times \xi(l) \times F(l)) \leq 70\%$ where $\xi(l) \times F(l)$ is an estimation of the number of short-lived flows in link $l$. Therefore, OpenTCP needs to solve the following linear program:
maximize \( w_0 \)
\[
s.t.: \quad 4 \leq w_0 \leq 20
\]
\[
\forall l \in L, \ U(l) \times (1 + w_0 \times \xi(l) \times F(l)) \leq 70\%
\] (5.2)

By maximizing the initial congestion window, we can reduce flow completion times, while keeping the utilization below a certain threshold will help with stability. The solution to this optimization problem is any real number between 4 and 20. The operator has defined two mapping rules in this CCP. For nodes 5 to 8 the initial congestion window size will be set to \( \lceil w_0 \rceil \) as long as the drop rate is less than 1\%. The CCA checks this condition before changing TCP. For the rest of the nodes (nodes 1 to 4) in our example, the initial congestion window size is set to 5 segments. CUE 1 presented in Figure 5.3 shows the CUE sent to one of the nodes between nodes 5 and 8 when \( \lceil w_0 \rceil = 12 \). The condition presented in the mapping rule is defined as the stability region. The CCA will enforce changes to initial congestion window size only if the packet drop rate is below 1\%. Otherwise, the system will use the default initial congestion window size for nodes 5 to 8.

### 5.3.3 TCP Adaptation

The final step in OpenTCP’s operation is to change TCP based on CUEs generated by the Oracle. After solving the optimization problem, the Oracle forms CUEs and sends them directly to CCAs sitting in the end-hosts. CUEs are distributed as broadcast and multicast messages sent to SDN switches which, in turn, deliver them to individual CCAs. OpenTCP distributes CUEs over time in order to have minimum impact on the underlying network traffic.

**CUEs have a very small overhead:** CUEs are usually very small packets. In our implementation, each CUE is usually less than 10-20 bytes. Even if we have CUEs as big as 200 bytes, and assuming we use multicasting (one message per 50 nodes), CUE
dissemination will have an average bandwidth requirement of less than 100Kb/s for a data center with 1,000,000 nodes and an update period of 10 minutes.

**CCAs are responsible for receiving and enforcing updates:** Upon receiving a CUE, each CCA will check the condition given in each mapping rule. If this condition is satisfied, the CCA immediately applies the command. Changes can be simple adaptations of TCP parameters or can require switching between different variants of TCP. If the CCA does not receive any CUEs from the Oracle for a long time (we use $2T$ where $T$ is the OpenTCP operation cycle), or if the mapping rule conditions are violated, the CCA reverts all changes and OpenTCP goes back to the default TCP settings.

**Adaptation example:** Assume the operator simply wants to switch from TCP Reno to CUBIC while keeping an eye on packet drop rates. In this case, the operator can use the CCP shown in Figure 5.3(b). Here we assume that the default TCP variant running in end-hosts is TCP Reno. The optimization problem does not have any variables or an objective function. Therefore, those are set to null. There is a constraint limiting the link utilization to 80%. If the network has a low load (i.e. as long as all links have a utilization less than 80%) the Oracle will send CUEs to all CCAs instructing them to switch to CUBIC. If network conditions change and utilization goes above 80% in any link, the Oracle will not be able to satisfy the optimization constraints. In this case, the CCAs will fall back to the default TCP variant, which is TCP Reno here.

## 5.4 Remarks

**OpenTCP without SDN Support.** OpenTCP is expressly designed for SDNs. In fact, the existence of a centralized controller with access to the global state of the network, as well as switches which can be used to collect flow and link level statistics, make SDNs the perfect platform for OpenTCP. Having said that, we can use OpenTCP on a traditional network (non-SDN) as long as we can collect the required network state and
traffic statistics. Clearly, this requires more effort and might incur a higher overhead since we do not have a centralized controller or built-in statistics collection features. In the absence of an SDN controller, we can use a dedicated node in the network to act as the Oracle. We can use SNMP to collect general link level statistics (e.g. utilization and drop rate). We can use CCAs to collect flow related information (e.g. number of active flows) at each end-host in a distributed manner and send relevant information to the Oracle. The Oracle can query the switches and CCAs to collect various statistics and if needed to aggregate them to match the statistics available in the SDNs. This needs to be done with extra care, so that the overhead remains low, and the data collection system does not have a major impact on the underlying network traffic.

**Conditions and constrains for OpenTCP.** The current implementation of OpenTCP relies on a single administrative authority to derive network and traffic statistics. To this end, we also assume no adversary in the network and that we could change the end-host stacks. Hence, OpenTCP doesn’t have to operate in a data center per say, any network that satisfies the above conditions can benefit from OpenTCP.

**Changing parameters in a running system.** Depending on the TCP parameter, making changes to TCP flows in an operational network may or may not have immediate changes in TCP behaviour. For example, adapting init_cwnd only affects new TCP flows but max_cwnd change is respected almost immediately. If the congestion window size is greater than the new max_cwnd, TCP will not drop those packets, it will deliver them and afterwards respects the new max_cwnd. Similarly, enabling TCP pacing will have an immediate effect on the delivery of packets. The operator should be aware of when and how each parameter setting will affect on going TCP sessions.

**Beyond TCP and Data Centers.** Through this chapter, we tied OpenTCP with three main components: SDN, Data Center, and TCP itself. In fact, OpenTCP is designed to adapt TCP in a data centre under a SDN, however, OpenTCP can be liberated from these components. Recall that in Chapter 3 we are adapting TCP’s maximum
congestion window size for YouTube flows. The main advantage point is to have a single administrative authority that can give us reliable insider information and has a reasonable level of control over sessions to adjust parameters. Going one more step beyond, OpenTCP doesn’t need TCP for its operation either, it is a framework to adapt any protocol based on measurements.
Chapter 6

OpenTCP Evaluation

In this chapter, we evaluate the feasibility and performance of OpenTCP. We measure the performance of a basic OpenTCP implementation by extending the NS-2 simulator and building an OpenTCP simulation environment (Section 6.1). Moreover, we explain a real-world deployment of OpenTCP at SciNet, Canada’s largest High Performance Computing (HPC) data center (Section 6.2).

6.1 NS-2 Simulations

This section introduces NS-2 OpenTCP, an NS-2 based OpenTCP implementation that embeds an OpenTCP API on top of the NS-2 simulation package. Our patch is for NS-2.31 and compatible with 2.29 and 2.30. NS-2 OpenTCP introduces new objects in

![Figure 6.1: Simple NS-2 OpenTCP topology.](image)
the NS2 environment such as OpenTCP Controller, OpenTCP switch, and Congestion Control TCP Agent based on OpenTCP’s design explained in Section 5.2. A snapshot of a sample Tcl (Tool Command Language) script that simulates a simple OpenTCP topology (Figure 6.1) is illustrated in Algorithm 2.

<table>
<thead>
<tr>
<th>Algorithm 2: A snapshot of OpenTCP NS-2 to create topology in Figure 6.1</th>
</tr>
</thead>
</table>
| # Create a simulator object  
set ns [new Simulator] |
| # Create OpenTCP Oracle  
set opentcp_oracle [new OpenTCP/Controller] |
| # Create two switches  
set sw_0 [$ns node]  
set sw_1 [$ns node] |
| # Create a link between sw_0 and sw_1  
$ns duplex-link $sw_0 $sw_1 $bottleneck_bw $latency_bottleneck $drop |
| # Create OpenTCP control path parameters to allow monitoring  
set link [new OpenTCP/Link]  
$link params [$sw_0 id] [$sw_1 id] $controlpath_bw $controlpath_latency  
$opentcp_oracle attach-link $link |
| # Create a client node and attach it to sw_0  
set n_client [$ns node]  
$ns duplex-link $n_client $sw_0 $accesslink_bw $latency $drop  
$link params [$n_client id] [$sw_0 id] $controlpath_bw $controlpath_latency  
$opentcp_oracle attach-link $link |
| # Create a TCP sink and attach it to sw_1  
set n_sink [$ns node]  
$ns duplex-link $n_sink $sw_1 $accesslink_bw $latency $drop  
$link params [$n_sink id] [$sw_1 id] $controlpath_bw $controlpath_latency  
$opentcp_oracle attach-link $link |
| # Create a TCP agent and attach it to client node  
set a_tcp [new Agent/TCP/Linux]  
$ns attach-agent $n_client $a_tcp  
$opentcp_oracle attach-agent $a_tcp [$n_client id] |

We begin by using various examples to verify the correctness of our simulation imple-
Chapter 6. OpenTCP Evaluation

88

Figure 6.2: Simulations Topology.

<table>
<thead>
<tr>
<th>Flow</th>
<th>CBR (Mbps)</th>
<th>RTT (ms)</th>
<th>start time (sec)</th>
<th>end time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$UDP_0$</td>
<td>100</td>
<td>7.66</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>$UDP_1$</td>
<td>200</td>
<td>5.4</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>$UDP_2$</td>
<td>350</td>
<td>8.3</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>$UDP_3$</td>
<td>450</td>
<td>10.4</td>
<td>20</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.1: UDP CBR rates and start times.

mentation. We use several topologies in our simulations such as a simple bus topology and then expand it to ring, fat-tree [13], and hierarchical tree to test the correctness of the Oracle’s view of the network topology and traffic matrix. To clarify the use of OpenTCP, we use topology and traffic that are not necessarily in accordance with data centers. Below, we present a representative subset of our simulations.

6.1.1 Oracle’s Estimation: Constant Bit Rate Flows

Let us begin with constant bit rate UDP flows. Assume a topology as illustrated in Figure 6.2; where four UDP flows are sharing a network with three switches. Each flow has a different rate, RTT, start time, and end time as listed in Table 6.1. The average RTT for the network is 8 ms. The buffer size at the switches is 200 packets while the
bandwidth-delay product is roughly 1041 packets.

Figure 6.3 plots the utilization of each UDP flow as a function of time at the bottleneck link (the link between switch$_0$ and switch$_1$). This utilization is measured by the NS-2 trace function with a packet by packet precision. As shown, the aggregate utilization changes as new flows join and leave the network.

At every time scale interval $T = 1$ second, the Oracle calculates traffic matrix, link utilization, drop rate, number of active flows, queue occupancy, and any other metric of interest at each switch. For example, Figure 6.4 shows the Oracle's estimation of utilization of each UDP flow as a function of time at the bottleneck link. Because the UDP flows have a constant bit rate, the Oracle has a fairly accurate view of the traffic in the network. The gap at $t = 20$ sec is because the Oracle observes the start of $UDP_3$ one second late. Clearly, having a smaller measurement time scale would ameliorate this. In today’s large-scale data centers, a time scale in the order of 100 $ms$ is feasible. This estimation is consistent in all three switches in the network. Table 6.2 shows the Oracle’s
Figure 6.4: Per flow utilizations at the bottleneck link with UDP flows. Oracle’s view of the traffic with $T = 1$ second.

Table 6.2: Per-switch traffic matrix as measured by the Oracle at time $t = 30$.

\[
\begin{bmatrix}
\text{switch}_0 \\
\text{switch}_1 \\
\text{switch}_2 \\
\end{bmatrix}
\begin{bmatrix}
\text{UDP}_0 : 100\text{Mbps} \\
\text{UDP}_1 : 0\text{Mbps} \\
\text{UDP}_2 : 350\text{Mbps} \\
\text{UDP}_3 : 450\text{Mbps} \\
\end{bmatrix} \leftrightarrow
\begin{bmatrix}
\text{UDP}_0 : 100\text{Mbps} \\
\text{UDP}_1 : 0\text{Mbps} \\
\text{UDP}_2 : 350\text{Mbps} \\
\text{UDP}_3 : 450\text{Mbps} \\
\end{bmatrix} \leftrightarrow
\begin{bmatrix}
\text{UDP}_0 : 100\text{Mbps} \\
\text{UDP}_1 : 0\text{Mbps} \\
\text{UDP}_2 : 350\text{Mbps} \\
\text{UDP}_3 : 450\text{Mbps} \\
\end{bmatrix}
\]

estimation of traffic matrix for each of the switches at time $t = 30$ sec.

6.1.2 Oracle’s Estimation: TCP flows

Next, we repeat the same experiment with TCP flows while keeping the flows’ start and end times as well as their RTTs the same. We set the capacity of access links to 1Gbps. As soon as a TCP flow starts, it will try to send as fast as it can in a best effort fashion. They consume the entire link utilization and cause congestion and packet drops in the network. Figure 6.5 plots the utilization of each TCP flow as a function of time at the bottleneck link. Again, this utilization is measured by the NS-2 trace function with a packet by packet precision. As shown, the dynamics of TCP flows keep the aggregate
utilization at 100% nearly all the time. However, the utilization of each flow is affected by the existence of other flows in the network. Because the figure is based on packet by packet arrivals, at some intervals it is difficult to observe which flow occupies more bandwidth. A time-line of the events is summarized below:

- between time $t \in [0 - 5)$, $TCP_0$ has 100% utilization since its the only flow in the network,
- between time $t \in [5 - 10)$, $TCP_1$ starts, occupies almost entire bandwidth and hence starves $TCP_0$,
- between time $t \in [10 - 20)$, $TCP_2$ starts, occupies almost entire bandwidth and hence starves $TCP_0$ and $TCP_1$,
- between time $t \in [20 - 50)$, $TCP_3$ starts and tries to occupy some bandwidth, but it eventually loses to $TCP_2$ and $TCP_2$ continues to occupy most of the bandwidth,
Figure 6.6: Per flow utilizations at the bottleneck link with TCP flows. Oracle’s view of the traffic with $T = 1$ second.

- between time $t \in [50 - 100)$, $TCP_1$ and $TCP_3$ compete for available bandwidth.

Figure 6.6 plots the Oracle’s view of the flow utilizations based on a time scale of $T = 1$ second where it is easier to observe that $TCP_0$ starts with full link utilization, but drops down to almost zero and then occupies almost 10% of the available bandwidth. Note that in this experiment, the Oracle did not make any modifications to TCP sources and acts as a monitoring unit. In OpenTCP, the Oracle uses these measurements and based on the congestion control policy it decides on which TCP parameter, or which flow to adapt. This is done using a TCP-agent that is attached to every TCP flow. Our next experiments will show examples of modified TCP sources.

6.1.3 Adaptable TCP Knobs

As explained in Chapter 5, a congestion control agent is in charge of making changes to TCP sources. These changes are based on the Oracle’s congestion control policies and its
estimation of the status of the network. The spectrum of available adaptable parameters (or “TCP knobs”) is dependent on the network operator, CCA, and adaptability of the TCP stack in the end-hosts. In this thesis, we study some of these knobs such as:

- congestion window size ($cwnd$),
- initial congestion window size ($init_{cwnd}$),
- maximum congestion window size ($max_{cwnd}$),
- Retransmission Time-Out ($RTO$),
- congestion control algorithm itself (cubic, bic, reno, etc), and
- burstiness of packets (pacing).

In the following section, we will show an example of modifying the congestion window size. Chapter 3 studies the impact of dynamically changing $max_{cwnd}$ based on network characteristics, and Chapter 4 investigates the effect of enabling TCP pacing to control the burstiness of TCP packets. Later, Section 6.2 studies the effect of adapting $init_{cwnd}$ and $RTO$ of TCP flows. Other interesting adaptable TCP knobs include Additive Increase Multiplicative Decrease (AIMD) parameters of congestion control scheme, receive window size, and retransmission behaviour. Dukkipati et al. explore weaknesses of the standard TCP retransmission algorithm and propose an algorithm to control retransmission of packets in TCP to be proportional to the number of packet drops [46].

6.1.4 Custom Flow Priority

Let us assume that the network operator in Section 6.1.2’s example is interested in giving $TCP_0$ the highest priority and defines a policy as follows:

*Keep the utilization of $TCP_0$ at least 70% of bottleneck link’s capacity. Divide the remaining 30% of bandwidth equally among other active flows.*
This policy translates into a module in the Oracle’s code where the Oracle sends CUEs to the CCAs at TCP sources to adjust their $cwnd$ so that the utilization of $TCP_0$ is kept above 700Mbps (or $cwnd \geq 446$ segments for RTT 7.66 ms). Moreover, the Oracle divides the remaining BW equally among other active flows.

Figure 6.7 illustrates the utilization of flows under this policy. The network administrator can use this policy to prioritize important flows such as advertisement flows, search results, stock related flows, peering flow, etc. The remaining less important flows could be back-up flows, map-reduce flows or other non-latency sensitive flows.

The operator can even define custom utilization for each flow and let OpenTCP drive the network. A sample congestion control policy could be:

- if $TCP_1$ exists in the network, it should have the highest priority,
- if $TCP_2$ exists in the network, it should have the lowest priority,
- if $TCP_0$ and $TCP_3$ exist in the network, and if time $t \in [20 - 40)$, $TCP_3$ gets
highest priority,

- otherwise, $TCP_3$ should occupy 20% of bandwidth and $TCP_1$ should occupy 70% of available bandwidth.

Figure 6.8 illustrates this policy in effect. When $TCP_1$ is in the network, it occupies the entire bandwidth. Moreover, $TCP_2$ has lowest bandwidth throughout the simulation, and $TCP_3$ fully utilizes the bandwidth between time 20 to 40 seconds and then shares the bandwidth with $TCP_0$ according to the given policy.

### 6.1.5 Dynamic Traffic Pattern

Recall the network in Figure 1.1 in Chapter 1 where the link utilization of a service provider had a dynamic pattern. Depending on the time of the day, the link utilization could be low for a significant period of time. Inspired by this observation, in the following we present the effect of adapting TCP to network traffic pattern to take advantage of
opportunities (when the load is low) and handling critical situations (when the load is high).

We use a simulation topology illustrated in Figure 6.9. In each simulation, we first use a default TCP with no adaptation; and in the second round, we enable OpenTCP with dynamic adaptations. We then compare the performance of the two cases.

6.1.5.1 **Opportunity: Increase init.cwnd for Mice Flows**

Let’s assume that there is only one TCP flow with size 60 KB (mice flow) and let’s compare the effect of simply increasing the init.cwnd from four to 30 segments. Here, with only one flow in the network, compared to the default TCP, we will operate at a higher instantaneous link utilization. This is because the cwnd will be elevated immediately, and because the network is empty, the utilization jumps higher. The packet drop rate will be zero in both cases and we will have a higher queue occupancy; again because the cwnd is operating at a higher value. The gain is that higher congestion window leads to faster flow completion time. Figure 6.10 compares \((S_0 - S_1)\) link utilization, flow’s
Chapter 6. OpenTCP Evaluation

Figure 6.10: Mice Flow case: Comparing bottleneck link’s utilization, congestion window size, bottleneck link’s queue occupancy, and flow completion times between OpenTCP and regular TCP.

Reducing the flow completion time is an attractive result for service providers. This is because for extremely latency sensitive traffic (such as “search”), saving RTTs translates to higher revenue. Moreover, adapting the init.cwnd is an easy change to TCP implementation which makes this policy a practical engineering solution. The effort by Dukkipati et al. lead to increasing the default init.cwnd to 10 segments instead of four segments in the most recent Linux kernels [49]. In Section 6.2, we will demonstrate more examples on the effect of increasing init.cwnd.
Chapter 6. OpenTCP Evaluation

Algorithm 3: Max-Min fairness congestion control policy.

- Allocate bandwidth to flows in order of increasing demand
- No flow receives more than available bandwidth
- Flows with unsatisfied demands split the remaining available bandwidth

6.1.5.2 Critical: Max-Min Fairness for Elephant Flows

The above strategy to increase init\(_{cwnd}\) to adapt TCP does not seem promising for fat or longed-lived (elephant) TCP flows. A more reasonable strategy could be to rate limit elephant flows. Let’s assume that the network administrator of Figure 1.1 would like to apply a rate limiting algorithm similar to the Max-Min fair-share algorithm [90]. In this policy, the administrator wants to adjust flows’ cwnd according to their demand and available bandwidth in the network. Algorithm 3 illustrates this policy: the flows with smallest demands receive their entire demand and the rest of the bandwidth is divided equally among remaining flows. The admin could measure the flows’ demands or it can be pre-configured with estimate on flow demands.

Let us repeat a similar experiment as in the above section but with a flow of size 1500 MB. In this case, because the flow size is large, it demands the entire bandwidth. But, the Max-min fair-share algorithm will not allow the cwnd to grow beyond the capacity of the network. This will avoid queue build up and packet drops. Figure 6.11 compares the link utilization, congestion window size, queue occupancy, and flow completion time for the two cases of default TCP vs. algorithmic TCP. As shown, the congestion window size at TCP case grows beyond the network’s capacity which causes the bottleneck link’s buffer to overflow and make TCP retransmit lost packets. However, in the OpenTCP case, the algorithm does not allow the cwnd to grow beyond the capacity of the network and keeps the queue occupancy below the maximum buffer size. The drop rate of the TCP case is 50 times larger than that of OpenTCP case. This leads to reducing the flow completion time by nearly 30 RTTs.
Figure 6.11: Elephant Flow case: Comparing bottleneck link’s utilization, congestion window size, bottleneck link’s queue occupancy, and flow completion times between OpenTCP and regular TCP.

6.1.5.3 Putting It Together: Dynamic Traffic Pattern

Next, we combine the two cases above to illustrate the effectiveness of TCP adaptation for a network with dynamic traffic pattern. For this purpose, we use an on-off elephant TCP flow to introduce a dynamic low-high utilization pattern as illustrated in Figure 6.12 (a). Moreover, we introduce thousands of mice TCP flows with average size 45KB randomly in the network. The utilization of four randomly selected mice TCP flows is shown in Figure 6.12 (b) and the aggregate utilization at the bottleneck link is shown in Figure 6.12 (c).

We define the following two policies for this network:

**Opportunity - Low utilization period**: Increase TCP’s *init_cwnd* to 20 segments when the bottleneck link’s utilization is below 50%.
Figure 6.12: Dynamic Traffic Pattern case: (a) Elephant flow’s utilization, (b) utilization of four random mice flows, (c) bottleneck link’s utilization, (d) CDF of flow completion times.

Critical - High utilization period: Otherwise, use the max-min algorithm explained in Section 6.1.5.2.

Figure 6.12 (d) compares the CDF of flow completion time for mice TCP flows between OpenTCP and regular TCP. OpenTCP has orders of magnitude lower flow completion time compared to TCP.

6.1.6 Using TCP Pacing to Suppress Burstiness

The literature shows that TCP is bursty by nature and its bursts put stress on the buffers at network switches [27]. Consider a data center network with small-buffer switches. To suppress the burstiness caused by TCP in this network, we can use paced TCP [138]. However, there have been several inconclusive studies on the effectiveness of pacing in mitigating TCP burstiness [12, 34, 115, 128]. As a result, TCP pacing is not used in
today’s data centers. In Chapter 4, we showed that TCP pacing successfully mitigates the burstiness of TCP flows as long as the total number of concurrent flows falls below a certain threshold [60], called the Point of Inflection. In a large data center with thousands to millions of concurrent flows, it is almost impossible to guarantee that the number of active flows will always be less than any threshold. Therefore, providers are reluctant to enable pacing despite its proven effectiveness in some settings. However, if we have a system that can measure the number of active flows, we can dynamically switch between paced TCP and regular TCP to ensure the system is always at its peak performance.
In the previous sections, we presented a synthetic and simple set-up to illustrate the need for OpenTCP. To measure the performance of OpenTCP in practice, in this section, we use a more realistic simulation that is based on real data center studies. The results of this section will be used to demonstrate OpenTCP’s potential real-world impact.

The simulation topology is illustrated in Figure 6.13; 80 hosts per rack are connected via 100 Top-of-Rack (ToR) switches that, in turn, connect to core and aggregation switches. This topology is based on SciNet’s topology that we will explain in the next section. The hosts generate TCP CUBIC traffic with flow sizes following a Pareto distribution and flow inter-arrivals following an exponential distribution. The average RTT is 20 ms and the switch buffer sizes are 400 packets. The end-hosts generate three types of TCP flows based on recent data center traffic studies [16,62,84]:

1. query traffic (mean flow size 30 KB, mean inter-arrival time 1 ms),

2. delay sensitive short messages (mean flow size 1 MB, mean inter-arrival time 100 ms), and

3. throughput sensitive long flows (mean flow size 100 MB, mean inter-arrival time 1 s).

Then, we configure OpenTCP to allow TCP pacing when the number of active flows in the network is less than the bound for pacing’s point of inflection. Figure 6.13 plots the queue occupancy of the buffer at one of the aggregation switches over time. The figure shows that by enabling pacing, OpenTCP makes a very small footprint in the switch packet buffers, compared to non-paced TCP. Moreover, Figure 6.14 (a) depicts flow completion times versus flow sizes showing that flows with same size finish earlier in OpenTCP. Figure 6.14 (b) plots the CDF of flow completion times. As the figures show, using OpenTCP reduces the average flow completion times by 50% compared to the regular TCP.
6.1.7 Effect of Oracle’s Time Scale

Utilizing OpenTCP in data centers has substantial advantages including more intelligent congestion control schemes and easier customization of current congestion control schemes. However, similar to any other architecture, OpenTCP has its own pitfalls. In OpenTCP, the Oracle periodically collects the network statistics from the switches every $T$ seconds. Selecting an appropriate time scale period for the Oracle is an important trade-off in realizing OpenTCP. If the statistics are real-time, the Oracle has a better chance to present optimum rules for the end-hosts. However, having real-time statistics is not feasible in reality. We envision a refresh rate in the order of tens of milliseconds is feasible in today’s SDNs [14, 50, 136]. In the following we present a case that shows the impact of $T$ on performance.

We use three different variants of OpenTCP with $T$ equal to 10ms, 500ms, and 1 second for a network with 100 flows, 100 packets buffer size, average RTT of 20ms. We chose two types of flows in the system: large flows (flow sizes chosen from a Pareto distribution with shape parameter 1.4 and average flow size of 5000 packets) and query flows (with average flow size of 30 packets). We used Exponential distributions to chose...
the large and query flows inter-arrivals with mean 500ms and 5ms, respectively. We define a congestion control policy to directly set the congestion window. We then used three OpenTCP variants with three different $T$ parameters: OpenTCP_1 with $T = 10ms$, OpenTCP_2 with $T = 500ms$, and OpenTCP_3 with $T = 1sec$. To further highlight this experiment, we chose a rather unrealistic congestion control policy to update OpenTCP’s current $cwnd$ so that every flow in the system gets a fair-share of the available bandwidth. Figure 6.15 compares the flow completion times for the three different $T$ values as well as with the default TCP. As shown, in OpenTCP_3, the poor choice of $T$ lead to an even worse performance compared to default TCP. This is because the Oracle’s updates were sent with old measurements and the congestion control policy was forcing the $cwnd$ of flows based on old measurements that weren’t valid anymore. An easy way to avoid this situation is to make sure OpenTCP falls back to default TCP if the measurement data is older than a predetermined threshold.

### 6.2 Real World Deployment

In this section, we evaluate the performance of large-scale OpenTCP. We deploy OpenTCP in a High Performance Computing (HPC) data center with $\sim 4000$ nodes each having 8 cores. SciNet is mainly used for scientific computations, large scale data analyses, and simulations by a large number of researchers with diverse backgrounds: from biological sciences and chemistry, to astronomy and astrophysics. Through a course of 20 days, we enabled OpenTCP and collected 200TB of packet level trace as well as flow and link level statistics.

#### Setup:

The topology of SciNet is illustrated in Figure 6.16. There are 92 racks of 42 servers. Each server connects to a Top of Rack switch (ToR) via 1Gbps Ethernet. The ToRs are Blade Network Technologies Gigabit switches. Each end-host in SciNet runs Linux 2.6.18 with TCP BIC [133] as the default TCP. SciNet consists of 3,864 nodes
Figure 6.16: The topology of SciNet the data center network used for deploying and evaluating OpenTCP.

with a total of 30,912 cores (Intel Xeon Nehalem) at 2.53GHz, with 16GB RAM per node. The ToR switches are connected to a core Myricom Myri-10G 256-Port 10GigE Chassis each having a 10Gbps connection.

**OpenTCP in SciNet:** We did not have access to a large scale SDN and could not change any hardware elements in SciNet. Therefore, we took the extra steps described in Section 5.4 to deploy OpenTCP. In our experiments, the CCAs periodically collect socket-level statistics, aggregate them, and send the results to the Oracle, a dedicated server in SciNet. The Oracle performs another level of aggregation to produce final statistics combining data from SNMP and the CCAs. Throughout our experiments, we monitor OpenTCP to ensure that we do not have a major CPU or bandwidth overhead. As Section 6.2.3 shows, the average CPU overhead of our instrumentation is less than 0.5%, and we have a negligible bandwidth requirement. We believe this is a reasonably low overhead, one which is unlikely to have a significant impact on the underlying jobs.
Evaluation overview: We begin this section by describing the workload in SciNet (Section 6.2.1). We show that, despite the differences in the nature of the jobs running in SciNet, the overall properties of the network traffic, such as flow sizes, flow completion times (FCT), traffic locality, and link utilizations are analogous to those found by previous studies of data center traffic [16, 28, 62, 117]. We take this step to ensure our results are not isolated or limited to SciNet.

6.2.1 Traffic Characterization

At SciNet, the majority of jobs are Message Passing Interface (MPI) [104] programs running on one or more servers. Jobs are scheduled through a queuing system that allows a maximum of 48 hours per job. Any job requiring more time must be broken into 48 hour chunks. Note that each node runs a single job at any point of time. We use OpenTCP to collect traces of flow and link level statistics, capturing a total of $\sim 200$TB of flow level logs over the course of 20 days.
**Workload:** We recognize two types of co-existing TCP traffic in our traces (shown in Figure 6.17): (i) MPI traffic and (ii) distributed file system flows. The majority of MPI flows are less than 1MB in size and finish within 1 second. This agrees with previous studies of data center traffic [16, 28]. It is interesting to note that only a few MPI flows (1%) last up to the maximum job time allowance of 48 hours. Unlike the MPI flows, the distributed file system traffic ranges from 20B to 62GB in size, and most (93%) finish within 100 seconds. A few large background jobs last more than 6 days. This pattern is in accordance with previous studies of data center traffic [16, 28].

**Locality:** Figure 6.18 represents the log of the number of bytes exchanged between server pairs in 24 hours. Element \((i, j)\) in this matrix represents the amount of traffic that host \(i\) sends to host \(j\). The nodes are ordered such that those within a rack are adjacent on the axes.\(^1\) The accumulation of dots around the diagonal line in the figure shows the traffic being locally exchanged among servers within a rack, a pattern that is similar to previous measurements [28, 117]. Note that this matrix is not symmetrical as

\(^1\)Virtualization is not used in this cluster.
it depicts the traffic from nodes on the $x$-axis to nodes on the $y$-axis. The vertical lines in the traffic matrix represent “token management traffic” belonging to the distributed file system. This type of traffic exchanges tokens between end-hos ts and a small set of nodes called token managers. Finally, the rectangular regions in the figure represent traffic between the nodes participating in multi-node MPI jobs.

**Utilization:** We observe two utilization patterns in SciNet. First, 80% of the time, link utilizations are below 50%. Second, there are congestion epochs in the network where both edge and core links experience high levels of utilization and thus, packet losses. Again, this observation accords with the previous studies where irrespective of the type of data center, link utilizations are known to be low [28, 117].

### 6.2.2 Experiment Methodology

We run a series of experiments to study how OpenTCP works in practice, using a combination of standard benchmarks and sample user jobs as the workload. During each experiment, we use OpenTCP to make simple modifications to end-hosts’ congestion control schemes, and measure the impact of these changes in terms of flow completion times and packet drop rates. Throughout our experiments, we set OpenTCP’s slow time scale to 1min, unless otherwise stated\(^2\) directing OpenTCP to collect statistics and send CUEs once every 1min. We also measure the overhead of running OpenTCP on CPU usage and bandwidth. In our experiments, we have used 40 nodes in 10 ToR racks and we run all the benchmarks explained above in over 10 iterations.

To evaluate the impact of the changes made by OpenTCP, we symmetrically split our nodes into two sets. Both sets run similar jobs, but we deploy OpenTCP on one of these two sets only (i.e. half the nodes). Because the two sets are comparable, we can observe the impact of OpenTCP with minimum influence from other random effects.

**Benchmarks:** In order to study different properties of OpenTCP, we use Intel MPI

\(^2\)In this environment, the fast time scale ($RTT$) is less than 1ms.
Benchmarks (IMB) as well as sample user jobs in SciNet. These benchmarks cover a wide range of computing applications, have different processing and traffic load characteristics, and stress the SciNet network in multiple ways. We also use sample user jobs (selected from the pool of jobs submitted by real users) to have an understanding of the behaviour of the system under typical user generated workload patterns. Figure 6.19 shows the CDF of flow sizes for these benchmarks. As illustrated in the figure, flow sizes range from 50B to 1GB, much like those observed in Section 6.2.1. The results presented here are for “all-to-all” IMB benchmark and “Flash” user job. We measured similar consistent results with other IMB benchmarks as well as other sample user jobs.

### 6.2.3 Experiment Results

In this section, we first define two congestion control policies for OpenTCP: OpenTCP\(_1\) and OpenTCP\(_2\). We then compare the performance of OpenTCP\(_1\), OpenTCP\(_2\), and unmodified TCP in terms of Flow Completion Times (FCT) and packet drop rates using different benchmarks. Finally, we take a closer look at the overheads of OpenTCP and the impact of changing the refresh rate.

**OpenTCP\(_1\) – Minimize FCTs**: Our first experiment with OpenTCP aims at reducing...
Figure 6.20: Comparing the CDF of FCT (left) and drop rate (right) of OpenTCP$_1$, OpenTCP$_2$, and regular TCP while running the “all-to-all” IMB benchmark.

FCTs by updating the initial congestion window size, and the retransmission time-out (RTO). In SciNet, there is a strong correlation between MPI job completion times and the tail of FCT. Therefore, reducing FCTs will have a direct impact on job completion times.

OpenTCP$_1$ collects link utilizations ($U(l)$), drop rate ($D(l)$), ratio of short-lived to long-lived flows ($\xi(l)$) and number of active flows ($F(l)$) like the CCP 1 described in Section 5.3.2. The optimization function is the same as that described by Equation 5.2; it tries to find the appropriate value for the initial congestion window size ($w_0$). The mapping rule is defined as follows:

$$\{1..N\}: \text{TRUE} \Rightarrow _{\text{init}}\text{cwnd}=[w_0], _{\text{RTO}}=2$$

Since the mapping rule is not conditional, the Oracle will send CUEs to all CCAs as long as the maximum link utilization is kept below 70% (as defined in CCP 1 in Section 5.3.2). If the link utilization goes above 70% for any of the links, the Oracle
Chapter 6. OpenTCP Evaluation

Figure 6.21: Congestion window distribution for OpenTCP₁ and unmodified TCP.

will stop sending CUEs and the CCAs will fall back to the default values of the initial congestion window size and the RTO.

**OpenTCP₂ – Minimize FCTs, limit drops:** While OpenTCP₁ aims at improving FCTs by adapting \( \text{init\_cwnd} \) and ignoring packet drops, we define a second variant called OpenTCP₂ which improves upon OpenTCP₁ by keeping the packet drop rate below 0.1%. This means that when the CCA observes a packet drop rate above 0.1% it reverts the initial congestion window size and the RTO to their default values. OpenTCP₂ introduces the following conditional mapping rule:

\[
\{1..N\}: (\_drop \leq 0.1\%) \Rightarrow \text{init\_cwnd} = [w_0], \_RTO=2
\]

This new CCP ensures that when the CCA observes a packet drop rate above 0.1% it returns the initial congestion window size and the RTO to their default values.

**Impact on flow completion times:** Figure 6.20 depicts the CDF of flow completion times for OpenTCP₁, OpenTCP₂, and unmodified TCP. Here, the tail of the FCT curve for OpenTCP₁ is almost 64% shorter than for the unmodified TCP. As mentioned above, this leads to faster job completion in our experiments. Additionally, more than 45% of the flows finish in under 260 seconds in OpenTCP₁, indicating a 62% improvement over unmodified TCP. Similarly, OpenTCP₂ helps 80% of the flows finish in a fraction of a second, a significant improvement over OpenTCP₁. This is because of OpenTCP₂'s
fast reaction to drop rate at the end-hosts. In OpenTCP$_2$, the tail of the FCT curve is 324 seconds, a 59% improvement over unmodified TCP. The FCT tail of OpenTCP$_2$ is slightly (8%) longer than OpenTCP$_1$ since it incorporates conditional CUEs and does not aggressively increase the initial congestion window. Moreover, note that the big gap between FCT of TCP and OpenTCP variants is due to nature of all-to-all benchmark that is stressing the SciNet network. In this benchmark, every process is sending and receiving to/from all other processes. This creates a TCP map-behaviour similar to TCP incast throughput collapse [126].

**Impact on congestion window size:** The improvements in FCTs are a direct result of increasing $\text{init.\_cwnd}$ in OpenTCP$_1$. Figure 6.21 presents a histogram of congestion window sizes for OpenTCP$_1$ and unmodified TCP. In the case of unmodified TCP, more than 90% of packets are sent when the source has a congestion window size of three, four, or five segments. OpenTCP$_1$ however, is able to operate at a larger range for congestion window sizes: more than 50% of packets are sent while the congestion window size is greater than 5 segments.

**Impact on packet drops:** The FCT improvements in OpenTCP$_1$ and OpenTCP$_2$ come at a price. Clearly, operating at larger congestion window sizes will result in a higher probability of congestion in the network, and thus, may lead to packet drops. As Figure 6.22 shows, the drop rate distribution for unmodified TCP has a shorter tail compared to OpenTCP$_1$ and OpenTCP$_2$. As expected, OpenTCP$_1$ introduces a considerable amount of drops in the network since the CUEs are *not* conditional and thus the CCAs do not react to packet drops. OpenTCP$_2$ has no drops for more than 81% of the flows. This is a significant improvement over unmodified TCP. However, the highest 18% of drops in OpenTCP$_2$ are worse than those in unmodified TCP, as OpenTCP$_2$ needs to observe packet drops before it can react. Some flows will take the hit in this case, and might end up with relatively large drop rates.

**The Flash Benchmark:** All the experiments described so far use the “all-to-all” IMB
Figure 6.22: Comparing the CDF of FCT (left) and drop rate (right) of OpenTCP₁, OpenTCP₂, and regular TCP while running the “Flash” benchmark.

The Flash benchmark is a selection of user jobs that we use to evaluate OpenTCP under realistic workloads. Flash solves the Sedov point-blast wave problem [111], a self-similar description of the evolution of a blast wave arising from a powerful explosion. The Flash benchmark represents a common implementation and traffic patterns in SciNet; it generates about 12,000 flows. Figure 6.22 compares the CDF of flow completion times and drop rates for OpenTCP₁, OpenTCP₂, and unmodified TCP. Like the results presented above, both OpenTCP₁ and OpenTCP₂ improve the tail of the FCT curve by at least 40%. But, OpenTCP₂ is more successful at reducing the drop rate as it has an explicit rule for controlling the drops.

**OpenTCP has low overhead:** Table 6.3 divides OpenTCP’s overhead into four categories: (i) CPU overhead associated with data aggregation and CUE generation at the Oracle, (ii) CPU overhead associated with data collection and CUE enforcement in the end-hosts, (iii) required bandwidth to transfer statistics from the end-hosts to the Oracle, and (iv) required bandwidth to disseminate CUEs to the end-hosts. We summarize
Table 6.3: The approximate OpenTCP overhead for a network with $\sim 4000$ nodes and $\sim 100$ switches for different Oracle refresh cycles.

<table>
<thead>
<tr>
<th>Oracle refresh cycle</th>
<th>1 min</th>
<th>5 min</th>
<th>10 min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle CPU Overhead (%)</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CCA CPU Overhead (%)</td>
<td>0.5</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>Data Collection (Kbps)</td>
<td>453</td>
<td>189</td>
<td>95</td>
</tr>
<tr>
<td>CUE Dissemination (Kbps)</td>
<td>530</td>
<td>252</td>
<td>75</td>
</tr>
</tbody>
</table>

Figure 6.23: 99th percentile FCT for different benchmarks.

OpenTCP’s overhead for refresh periods of 1, 5, and 10 minutes in Table 6.3. The table shows that OpenTCP has a negligible processing and bandwidth overhead, making it easy to scale OpenTCP to large clusters of hundreds of thousands of nodes. Clearly, OpenTCP’s refresh cycle plays a critical role in its overhead on networking elements.

**OpenTCP and other benchmarks:** We have conducted extensive experiments using a variety of benchmarks to evaluate OpenTCP in SciNet. In all of those, the behaviour of OpenTCP is consistent with what we present above for the All-to-all and Flash benchmarks. Figure 6.23 compares OpenTCP with unmodified TCP for different benchmarks we have used. As shown in this figure, OpenTCP consistently performs better than TCP in terms of FCT. We observe 9% (for Scatter) to 57% (for All-to-all) improvements in our experiments. The results look even better when we use more complex CCPs such as OpenTCP2.

**Fast OpenTCP:** In OpenTCP, CCAs periodically check the conditions defined in each
Chapter 6. OpenTCP Evaluation

CUE before applying changes. If the condition is not satisfied, the CCA rolls back to the default TCP settings. By default, CCAs check these conditions only when a new CUE is received (i.e. every $T$ seconds). However, unlike updates in the Oracle, checking the validity of this condition incurs a very low overhead in CCAs. Therefore, we could increase the frequency of this process. This is especially useful, when the condition checks a local stability measure (e.g. drop rate). In this case, the CCA can detect packet drops and react to them quickly. We call this variant Fast OpenTCP. By default, the CCAs in Fast OpenTCP check the CUE conditions every second. Figure 6.24 depicts the CDF of the FCT and drop rates for OpenTCP$_2$ and its fast counterpart. We see that Fast OpenTCP performs better in terms of both flow completion times and packet drop rates. We note that OpenTCP and Fast OpenTCP have the same Oracle and bandwidth overhead, and the CPU overhead in CCAs is negligible.

Figure 6.24: Fast OpenTCP vs. OpenTCP$_2$. CDF of flow completion times (top), CDF of drop rate (bottom).
6.2.4 Final Remarks

In this chapter, we evaluated the performance of an NS-2 based OpenTCP implementation as well as a real world implementation of OpenTCP. Our results show that by adaptively adjusting TCP parameters, OpenTCP can improve the performance compared to default TCP. To realize OpenTCP in practical settings, we assume that the network operator has expertise in defining the congestion control policies appropriate for the network. Moreover, to achieve pragmatic stability, there should be a monitoring system in place which alerts the operator of churns and instabilities in the network. This monitoring component should alert the controller whenever there is an oscillation between states. For example, if Oracle is making adjustments to TCP flows in time $t_1$ and immediately in time $t_1 + T$ those changes are reverted back, it is a good indication of a potential unstable condition. In this case, the operator should be notified to adjust either the congestion control policies, the stability constraints, or the overall resources in the network. One simple stability metric is number of times a rule is applied and reverted. The monitoring system can measure such stability metrics and alert the operator.

On top of adapting TCP parameters based on network and traffic conditions, OpenTCP can go beyond to include application semantics into its decision making process. For example, the target video streaming rate for our experiments in Chapter 3 can be fed to OpenTCP’s Oracle. Another application semantic could be file sizes in a transfer that OpenTCP can use to derive hints for TCP sessions.
Chapter 7

Conclusions and Future Work

In this thesis, we have discussed the potential value of OpenTCP, a dynamic congestion control framework for data center networks. OpenTCP takes advantage of the global view available to SDN controllers to find ways to adapt TCP. OpenTCP is very flexible in terms of the changes it can apply. The network operator has full control of the changes applied globally through congestion control agents. Congestion control policies tell OpenTCP what to optimize and which constraints to satisfy. How different congestion control policies impact OpenTCP is an interesting problem. In Chapters 3 and 4, we showed two real world use-cases of TCP adaptation. Due to a lack of a suitable framework, both are implemented as stand alone TCP modifications. However, they indicate the potential for using OpenTCP in today’s networks.

Our argument for the utility of OpenTCP is based on the following assumptions: 1) the network state in a data center network is highly dynamic but is measurable in a central location through network-wide data collection; 2) different TCP variants/parameters must be used under different network states to make significant performance gains or to achieve different performance goals; 3) it is practical to have a suitable set of congestion control policies where a mapping from network state to TCP configuration can be established.
In OpenTCP, the network operator has full control over the changes applied globally through the congestion control policies. Congestion control policies tell OpenTCP what to adapt and which constraints to satisfy. Although this gives freedom to the network operator, it runs the risk of instability in the network. This is because the operator might define an unstable congestion control policy or she might introduce incompatible congestion control policies in the network. Instability might also be caused if the operator chooses an unsuitable time scale to update TCP agents. OpenTCP takes steps to ensure the stability of the system in practice by switching to the default TCP settings when encountering signs of instability in the system. A more formal definition of stability along with theoretical analysis, and further experiments remains an interesting future work.
Bibliography


