Towards A Low-Latency Future Internet

WANG ZIXIAO
B.Comp., Fudan University

**A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE**

2018

Supervisor:
Associate Professor Ben Leong Wing Lup

Examiners:
Associate Professor Chan Mun Choon
Dr. Richard Ma Tianbai
Associate Professor KyoungSoo Park, Korea Advanced Institute of Science and Technology

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Wang Zixiao
11 July 2018

Acknowledgement

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Ben Leong. He always teaches me to work hard and do the right things. Without his guidance and encouragement, I could not have completed the journey. I thank my parents. They always support and encourage me when I make mistakes or feel stressed. I also thank my lab mates, Hong Hande, Raj Joshi, Wai Kay Leong, Oana Barbu, Xu Yin, Wang Wei, Aditya Kulkarni, Daryl Seah, Ayush Mishra, Meng Xiangyun, Jin Shuaizhao and Yu Guoqing. I learnt a lot from them and they are always willing to help me. Lastly, I thank my wife Chen Lu and daughter Wang Yiru. It is them that give me the strength and confidence to finish my study. This thesis is for them.

Publications

- Shuaizhao Jin, Xiangyun Meng, Daniel Lin-Kit Wong, Zixiao Wang, Ben Leong, Yabo Dong and Dongming Lu. "Improving Neighbor Discovery by Operating at the Quantum Scale". In *Proceedings of the 15th IEEE International Conference on Mobile Ad hoc and Sensor Systems (IEEE MASS 2018).* Oct. 2018.

- Wai Kay Leong, Zixiao Wang, and Ben Leong. "TCP Congestion Control Beyond Bandwidth-Delay Product for Mobile Cellular Networks". In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2017).* Dec. 2017.

- Shuaizhao Jin, Zixiao Wang, Wai Kay Leong, Ben Leong, Yabo Dong, and Dongming Lu. "Improving Neighbor Discovery with Slot Index Synchronization". In *Proceedings of the 12th IEEE International Conference on Mobile Ad hoc and Sensor Systems (IEEE MASS 2015).* Oct. 2015.

- Yin Xu, Zixiao Wang, Wai Kay Leong, and Ben Leong. "An End-to-End Measurement Study of Modern Cellular Data Networks." In *Proceedings of the 15th Passive and Active Measurement Conference (PAM 2014).* Mar. 2014.

- Wai Kay Leong, Yin Xu, Ben Leong and Zixiao Wang. "Mitigating Egregious ACK Delays in Cellular Data Networks by Eliminating TCP ACK Clocking." In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP 2013)*, Oct. 2013.

Contents

1 Introduction 1

1.1 Low-Latency TCP In The Wild 3

1.2 Environment-Aware Congestion Control 4

1.3 Contributions 6

1.4 Organization of Thesis 7

2 Related Work 8

2.1 Loss-Based TCP 14

2.1.1 Attempts to Improve TCP Tahoe 17

2.1.2 Improving Throughput for Large-BDP Networks 20

2.2 Reducing TCP Latency 23

2.2.1 Delay-based TCP 24

2.2.2 Improving Performance with Network Information . . . 27

2.3 Achieving Low Latency & High Throughput 29

2.3.1 Forecast-based TCP 30

2.3.2 Utility Functions 31

2.3.3 Modeling the Network 33

2.4 Summary 35

3 Low-Latency TCP in the Wild 38

3.1 Census of TCP Variants 38

3.2 Performance of Low-Latency TCP 41

4 Loss-Free Congestion Control 50

4.1 Aggregate Network Model 52

4.2 Inferring the Operating Environment 57

4.3 Adapting to the Environment 60

4.4 Implementation 62

4.4.1 Receive Rate (ρ) Estimation 62

4.4.2 Propagation Delay (RTT_{min}) Estimation 63

4.4.3 Capacity (C) Estimation 64

4.4.4 Regulating the Sending Rate 64
4.4.5 Regulating $B_{target\_initial}$ 64
4.4.6 Hostile Environment Detection 66
4.4.7 Handling Losses and Network Outage 66

5 Performance Evaluation 67
5.1 Baseline: Single Flow 69
5.2 Playing Well with Others 72
5.3 Plausible Explanation for Internet Observations 76
5.3.1 **Codel-Enabled Buffer** 80
5.4 Deep Buffers 84
5.4.1 Tuning Aggressiveness 87
5.5 Mode Detection Accuracy 89
5.6 Fairness to Own Kind 93
5.7 Realistic Operating Scenarios 94
5.8 Computation Overhead 101

6 Conclusion & Future Work 103
6.1 Future Work 105
6.1.1 Distribution of TCP Variants 105
6.1.2 Understanding Benevolence and Fairness 106
6.1.3 Achieving Ultra-low Latencies 107
6.1.4 Reducing Estimation Errors 107
6.1.5 Handling Packet Losses 108

Abstract

The Transmission Control Protocol (TCP) was originally developed to provide reliable and ordered data transfer between applications running on different hosts. Physical link speeds have increased significantly over the years and the cost of memory has dropped. Internet service providers (ISPs) then started deploying fast links together with deep buffers to mitigate network variations and increase link bandwidth utilization. Although the resulting throughput improved significantly, it led to a new Bufferbloat problem. Bufferbloat occurs when deep buffers are deployed in the Internet, and loss-based TCP variants aggressively fill the deep buffers until they overflow and packets are dropped, causing retransmissions and long end-to-end latencies.

We have seen in recent times the emergence of a large number of low-latency TCP variants to address the poor latency performance of traditional loss-based TCP variants like CUBIC. While we would expect these low-latency TCP variants to compete poorly against aggressive TCP variants like CUBIC in the wild, we found that these recent low-latency TCP variants match the performance of CUBIC and even sometimes outperform CUBIC in our experiments on Amazon Web Service (AWS). After analyzing the trace data, we found that these variants were throttling CUBIC by inflicting significant losses on the network. Because most low-latency TCP variants are insensitive to packet loss, they are not affected by packet losses at the bottleneck buffers, while traditional loss-based TCP variants like CUBIC will back off by reducing *cwnd*. This allows them to take up additional available bandwidth once the loss-based TCP variants back off and obtain a larger share of the available bandwidth. We also found that the default receive window for TCP CUBIC also limits its aggressiveness and so allows low-latency variants to compete more favorably on the Internet.

Given the good performance of these low-latency variants in the wild, we expect them to eventually become more common. However, our experiments also suggest that as such variants become more common, they will cause performance degradation to competing CUBIC flows. We argue that to transition smoothly to a future Internet without causing significant degradation to existing flows, new low-latency variants need to do more to avoid inflicting unnecessary packet losses to existing CUBIC flows. To address this issue, a sender can make better choices for congestion control if it is aware of the existence and behavior of other flows sharing the same bottleneck link.

We proposed *EvaRate*, a new rate-based congestion control algorithm that incorporates a new buffer estimation technique which allows an EvaRate flow to infer its own buffer occupancy as well as that of the competing flows sharing the same bottleneck buffer. With this mechanism, an EvaRate flow is able to determine its operating environment and, when in a low-latency (or *benevolent)* environment, *collaboratively* regulate the bottleneck buffer occupancy with other EvaRate flows. By doing so, EvaRate avoids inflicting loss on the underlying network. We implemented EvaRate in the Linux kernel and performed both trace-driven and Internet experiments to verify its throughput and latency performance, as well as the effectiveness of its environment detection. Our results suggest that EvaRate represents a promising new approach for congestion control – by implementing a control loop that infers the conditions of the network environment and reacts quickly.

Finally, our approach advances the state-of-the-art in TCP congestion control by directly modeling the bottleneck link buffer and allowing flows to collaboratively regulate this buffer so as to achieve low latency while still fully utilizing the available bandwidth. We believe that our work is one potential approach to allow the Internet to transition smoothly to a low-latency future without disrupting its current operation, as the proportion of low-latency TCP variant flows continues to increase.

List of Figures

2.1 Summary of TCP design space. 13

3.1 RTT vs. *cwnd* for TCP Reno. 41
3.2 RTT vs. *cwnd* for TCP CUBIC. 42
3.3 RTT vs. *cwnd* for BBR. 42
3.4 TCP throughput for inter-continental Internet traffic. 44
3.5 Loss rate of TCP variants for inter-continental traffic. 45
3.6 TCP throughput of CUBIC and reference flows for inter-continental Internet traffic. 45
3.7 TCP loss rate of CUBIC and reference flows for inter-continental Internet traffic. 46
3.8 Throughput of CUBIC flows vs. reference flows for inter-continental traffic. 47
3.9 Impact of loss on CUBIC throughput. 48

4.1 Simplified model of network bottleneck. 53
4.2 Relationship between the sending rate and buffer occupancy. . 54
4.3 The negative-feedback loop to manage the bottleneck link buffer. 55
4.4 Time evolution of $B_{target}(t)$ [schematic]. 57
4.5 Augmented model of network bottleneck. 58

5.1 Testbed setup using Cellsim [43]. 68
5.2 Network metrics for constant-rate (16 Mbps) trace. 70
5.3 Impact of L_{max} on throughput and latency. 71
5.4 Interaction between two EvaRate flows for constant-rate (16 Mbps) trace. 73
5.5 Interaction between CUBIC and EvaRate for constant-rate (16 Mbps) trace. 74
5.6 Interaction between BBR and EvaRate for constant-rate (16 Mbps) trace. 75
5.7 Impact of different proportions of BBR for shallow buffers. . 77
5.8 Impact of different proportions of Vivace for shallow buffers. . 78

5.9 Impact of different proportions of Copa for shallow buffers. . 79
5.10 Impact of different proportions of EvaRate for shallow buffers. 79
5.11 Impact of different proportions of BBR for Codel-enabled buffers. 81
5.12 Impact of different proportions of Vivace for Codel-enabled buffers. 82
5.14 Impact of different proportions of EvaRate for Codel-enabled buffers. 82
5.13 Impact of different proportions of Copa for Codel-enabled buffers. 83
5.15 Impact of different proportions of BBR for deep buffers. . . . 84
5.17 Impact of different proportions of Copa for deep buffers. . . . 85
5.16 Impact of different proportions of Vivace for deep buffers. . . 85
5.18 Impact of different proportions of EvaRate for deep buffers. . 86
5.19 Impact of different proportions of EvaRate++ ($\alpha = 1.5$) for deep buffers. 88
5.20 Impact of different proportions of EvaRate++ ($\alpha = 1.5$) for shallow buffers. 88
5.21 Two scenarios of mode detection for Copa and EvaRate. . . . 89
5.22 Cumulative distribution of mode detection delay. 90
5.23 False positive rates for Copa and EvaRate. 91
5.24 False negative rates for Copa and EvaRate. 92
5.25 CDF of Jain's index for various TCP variants. 93
5.26 Comparison of throughput for single-flow AWS experiment. . . 95
5.27 Comparison of loss rate for single-flow AWS experiment. . . . 96
5.28 TCP throughput of background CUBIC flows and reference flows for double-flow AWS experiment. 96
5.29 TCP loss rate of background CUBIC flows and reference flows for double-flow AWS experiment. 97
5.30 TCP throughput of background CUBIC flows and reference flows with a larger receive window. 97
5.31 TCP loss rate of background CUBIC flows and reference flows with a larger receive window. 98
5.32 Mobile cellular network trace: M1. 99
5.33 Mobile cellular network trace: Starhub. 100
5.34 Mobile cellular network trace: Singtel. 100
5.35 Satellite network trace. 101
5.36 CPU utilization of state-of-the-art congestion control algorithms. 102

List of Tables

2.1 Previously Proposed TCP variants. 37

3.1 Information of the AWS servers. 43
3.2 RTT for inter-continental traffic. 43

Chapter 1

Introduction

The Transmission Control Protocol (TCP) was originally proposed to provide reliable and ordered data transfer between applications running on different hosts. File and data transfers were the major applications running on the early networks which had relatively low bandwidth, so throughput, reliability and high utilization were the main focus of TCP design. As memory became cheaper and link bandwidth increased significantly, Internet service providers (ISPs) started to deploy deep buffers to mitigate network variations while achieving link utilization. This led to a serious latency problem for traditional loss-based TCP variants, called Bufferbloat [19]. The Bufferbloat problem is essentially caused by the loss-based congestion signaling mechanism and the use of the congestion window *cwnd* to regulate the sending rate indirectly. In particular, loss-based TCP variants probe the available network bandwidth by filling the bottleneck link buffer and causing packet loss by buffer overflow. Therefore, when deep buffers are deployed in the Internet, the filling of these buffers will inflict large latencies to all the flows sharing the same bottleneck link buffer.

We have seen in recent times the emergence of a large number of low-latency TCP variants [5, 27, 2, 9] to address the issue. The idea is to use various types of metrics like delay, RTT, loss rate and throughput as signals for network congestion, instead of using packet loss. In theory, these variants are expect to be more sensitive to network variations than traditional loss-based TCP variants, and thus be able to detect and react to network congestion earlier.

However, being more sensitive to network variations or congestion suggests that these low-latency TCP variants will be less aggressive than the TCP CUBIC flows sharing the same bottleneck. TCP CUBIC is currently the default TCP for Linux operating system, and is known for its aggressiveness and poor latency performance [15]. While the bottleneck link buffer is gradually filling and before a packet loss occurs due to overflow, CUBIC is unable to detect and react early due to its loss-based congestion signal. On the other hand, the low-latency TCP variants would detect network congestion from the increased latency and reduce their aggressiveness to ease the network congestion. The result is that these low-latency TCP variants have a reduced share of the bottleneck link bandwidth.

While state-of-the-art TCP variants claim to achieve high throughput and low latency, the sensitivity to latency should theoretically lead to poor throughput performance when they compete with loss-based CUBIC. Given that we expect low-latency TCP variants to be more commonly deployed on the Internet, we are thus keen to investigate if our expectation of how they will interact with TCP CUBIC is correct, and if so, how serious the degradation of throughput will be in practice.

1.1 Low-Latency TCP In The Wild

To this end, We first investigated how recent low-latency TCP variants performed in the wild by conducting a measurement study using the servers from Amazon Web Service (AWS). We designed 2 sets of experiments to compare TCP CUBIC with state-of-the-art PropRate [27], BBR [5] , Vivace [9] and Copa [2]. The AWS servers used in the experiment were located at Australia, Asia, North America and Europe. Surprisingly, while one would expect low-latency TCP variants to contend poorly against buffer-filling TCP variants like CUBIC, we found in the single-flow experiments in the current Internet that modern low-latency TCP variants can match the performance of CUBIC and even outperform CUBIC in some experiments. This suggests that low-latency TCP variants would likely become increasingly popular over time and we foresee a future Internet where CUBIC is replaced by these variants.

In our investigations, we believe that the likely reason why the new low-latency TCP variants are contending well with CUBIC on the Internet is that the bottleneck buffers are relatively shallow, or active queue management (AQM) like Codel [33] is deployed, making the bottleneck buffers effectively shallow. Low-latency TCP variants are designed to be insensitive to packet loss and more sensitive to latency. Effectively shallow bottleneck buffers will potentially cause packet losses, while still keeping latency low. This will make CUBIC flow back off and give up bottleneck link bandwidth to low-latency TCP variants. It was also shown earlier that BBR could throttle CUBIC by inflicting massive losses [17]. We have since observed similar issues with Copa [2] and Vivace [9].

Further investigations (see Chapter 5) showed that this anomaly is also partly because of the default receive window at the receiver being too small, which prevents CUBIC from obtaining a larger share of the bandwidth but has no impact to rate-based low-latency TCP variants. However, even with a larger receive window, we can still observe that CUBIC achieves lower throughput than low-latency TCP variants, with a non-trivial loss rate.

1.2 Environment-Aware Congestion Control

The results of the measurement study suggest that new low-latency TCP variants need to avoid causing throughput degradation to existing CUBIC flows if the Internet is to transition smoothly to a benevolent future, as the proportion of low-latency TCP flows increases. It is thus obvious that detecting whether traditional aggressive loss-based flows like CUBIC exist and reacting properly to avoid harming these flows are important to achieve this goal. We refer to the current Internet operating environment where a flow will almost always encounter a competing buffer-filling flow at the bottleneck link as a *hostile* environment. We envision a future where delay-based low-latency TCP variants can co-exist without buffer overflows, and refer to it as a *benevolent* environment.

We show that this can be done by directly estimating not only the buffer occupancy for our flow, but also that of competing flows sharing the same bottleneck buffer (§4.2). By implementing a negative-feedback control loop that keeps the buffer occupancy low, *EvaRate*, an ***En V****ironment-****A****ware* ***Rate-****based* congestion control algorithm, keeps latency low and avoids overflowing the buffer and inflicting loss on competing flows. By estimating the total occupancy of the shared bottleneck link buffer, an EvaRate flow can determine whether it is operating in a hostile or benevolent environment and work together with other EvaRate flows to keep the total occupancy low (§4.3). In the process, we are able to decouple EvaRate's behavior in the two operating environments. With decoupling, we can independently tune EvaRate's behavior for each operating environment, without impacting its behavior in the other. This means that we can now have a protocol that contends well against other flows in the current Internet, while simultaneously achieving lower latencies than existing algorithms in the future benevolent environment.

We show in our experiments that EvaRate is able to keep buffer occupancy low and achieve low latencies when contending with other flows. We also demonstrate that EvaRate is able to achieve throughput comparable to the state-of-the-art TCP variants in the current Internet, for mobile networks and even for satellite networks (§5.7). EvaRate operates at an efficient point along the throughput-latency frontier. Last but not least, EvaRate is more efficient than state-of-the-art TCP variants and consumes similar CPU resources with TCP CUBIC.

EvaRate has been implemented in the Linux kernel and requires no modifications in the Internet core or to the TCP receiver. It is thus amenable to immediate deployment.

1.3 Contributions

We make 2 major contributions in this thesis. i) we show that state-of-the-art low-latency TCP variants can cause significant losses on certain conditions, due to them being insensitive to losses. This will cause unfairness to the loss-based TCP variants that share the same bottleneck link; ii) we design a new rate-based congestion control algorithm that models the bottleneck link buffer, directly manages the bottleneck buffer, detects the environment, and allows competing EvaRate flows to collaboratively regulate their sending rates to keep the overall buffer occupancy much lower.

First, our measurement study demonstrates that the new low-latency TCP variants will inflict packet losses and throttle loss-based CUBIC flows, causing unfairness to these flows. The reason that causes such aggressiveness is shallow buffer and the insensitivity to packet losses. Since the new low-latency TCP variants often perform congestion control based on a number of metrics like one-way delay, RTT and bottleneck link bandwidth, they assume that there should not be any congestion if low latency is achieved. It implies that packet losses are random losses irrelevant network congestion when packet delay is low. However, in the case of shallow buffers, the latency might never be large because queuing delay is significantly lower compared to that for a deep buffer, even if the buffer is full and buffer overflow occurs, causing packets to be dropped. As a result, the insensitivity to packet losses will inflict a lot of packet losses to other loss-based TCP variant that share the same bottleneck link buffer and force them to back off, leading to significant unfairness.

We propose EvaRate and shows that it is a viable alternative to perform congestion control by modeling and directly regulating the bottleneck link buffer, detecting the environment and information of other flows, reacting quickly and collaboratively keeping the buffer occupancy low. In the process, we are able to minimize packet losses. In addition, we show that it is necessary to distinguish benevolent and hostile environment and achieve accurate detection. We believe that there is still much room for low-latency congestion control following our work, and a smooth transition to a low-latency Internet is possible.

1.4 Organization of Thesis

The rest of this thesis is organized as follows: In Chapter 2, we discuss the related works. In Chapter 3, we present our measurement study on the distribution of different TCP variants and the performance of low-latency TCP variants on the Internet. We then describe the design and implementation of EvaRate, our rate-based TCP congestion control, in Chapter 4. Thereafter, in Chapter 5, we compare EvaRate with other state-of-the-art TCP variants in terms of throughput, latency, fairness and environment detection. Finally, we conclude and discuss possible directions for future research in Chapter 6.

Chapter 2

Related Work

In 1974, Vint Cerf and Bob Kahn proposed a protocol for packet network intercommunication to support the sharing of the resources that exist in different packet switching networks [6]. It was the prototype of the modern transmission control protocol (TCP), which defined the goal, mechanism, and components of a congestion control algorithm, and provided a basic solution based on a mechanism called the congestion window *cwnd*, which is the maximum allowed number of packets in flight. Based on this design, TCP can generally be described in terms of 3 key components: *Packet Regulation*, *Congestion Signal* and *Congestion Avoidance and Recovery*. All TCP variants, including the state-of-the-art proposals, can more or less be decoupled into these 3 components.

Packet Regulation. Packet regulation determines how a TCP sender regulates the dispatching of packets at the sender. Basically, the goal of congestion control is to probe the available link capacity, match the packet dispatching with it to fully utilize the link, and adjust the packet dispatching when congestion occurs and eases. There are mainly two approaches to measure the link capacity and regulate packet dispatching: *cwnd*-based and rate-based. A *cwnd*-based mechanism limits the maximum allowed number of packets in flight with the congestion window *cwnd*, and performs congestion control by adjusting the maximum *cwnd* in response to detected congestion. On the other hand, a rate-based mechanism dispatches packets directly at a desired sending rate, and performs congestion control by regulating the sending rate in a manner that avoids causing congestion. The *cwnd*-based mechanism has been the most popular approach for packet regulation since the invention of TCP, and is still dominant today. Its main advantage is in its simplicity and stability, making it easy to implement and predictable in performance. However, *cwnd* only indirectly controls the sending rate, and therefore, we claim is unable to react to network variations quickly enough. Also, it suffers from downlink throughput degradation problem when uplink and downlink are asymmetric and uplink is congested, due to its ACK-clocking feature [27]. A rate-based mechanism addresses this problem because the sender directly manages the sending rate and the packet dispatching is not limited by ACK packets. But a rate-based mechanism comes with 2 major challenges: (i) packet pacing and (ii) network outage. It is not easy to implement a rate-based mechanism in the kernel, because it involves packet pacing and requires clock-based interrupts. Also, a rate-based mechanism would saturate a network in the event of a network outage, since the sender is no longer ACK-clocked and will not stop sending packets when ACK packets are not received. With the availability of better hardware, these challenges have become much less serious. Most CPUs are fast enough to handle clock interrupts without significant loss in efficiency and most network links are almost lossless. Therefore, it is now practical to implement rate-based packet dispatching efficiently.

Congestion Signal. A TCP sender detects network congestion using a congestion signal. There are generally 5 metrics that can be used either independently, or in some combination, as congestion signals: packet loss, network delay, network information, utility function and buffer occupancy. Packet loss was initially used as the default congestion signal for TCP. The problem with a loss-based congestion signal is that when deep buffers are deployed in the Internet, loss-based TCP variants will fill the bottleneck link buffer to full, causing serious latency problems [33]. The problem is typically referred to as *Bufferbloat* [33]. Network delay is a natural alternative, and most delay-based TCP variants use both loss and delay as congestion signals, which means that they react not only to packet losses, but also to increased delay. Unfortunately, such TCP variants also tend to react more conservatively than both loss-based TCP variants, which makes them compete poorly with the prevailing TCP variants in the Internet. As a result, delay-based TCP variants did not find widespread adoption. A few TCP variants use only delay as a congestion signal, such as PropRate [27]. They require buffer delay be measured accurately, which is not easy in practice. There are also TCP variants that exploit network information provided by routers and switches to help detect network congestion, like DCTCP [1]. We believe that with the availability of SDN-enabled switches and programmable switches, it would be much easier to deploy such algorithms in the future. Some recent proposals use utility functions as indirect congestion signals.

Utility functions are defined functions of network metrics like throughput, delay and loss rate. A utility function defines the goodness of network status, and thus the value of a utility function indicates the level of network congestion. The drawback of such method is obvious: they are not easily explainable and is difficult to understand and predict the performance of the resulting congestion detection. In this thesis, we propose a new algorithm EvaRate, that uses the estimated bottleneck link buffer occupancy as a congestion signal, which allows EvaRate flows to collaboratively regulate the bottleneck link buffer. To the best of our knowledge, we are first to propose the use of estimated bottleneck link buffer occupancy as a congestion signal.

Congestion Avoidance and Recovery. Congestion avoidance and recovery involves regulating the send rate of the packets at the sender to achieve high network utilization while avoiding network congestion. It is the most important component in congestion control. Most *cwnd*-based TCP variants use the additive-increase and multiplicative-decrease (AIMD) mechanism or the multiplicative-increase and multiplicative-decrease (MIMD) mechanism to adjust *cwnd* and perform congestion control [7]. AIMD means that *cwnd* is incremented by 1 or a constant value x for each RTT and reduced by half on a congestion event. This worked well in the past when network bandwidth was relatively low and random losses were common. However, with the development of modern networks, link bandwidth has improved significantly and random losses are rare. It was found that the original AIMD *cwnd*-based mechanism was not able to fully utilize bottleneck link bandwidth because it was too conservative in increasing *cwnd* when no network congestion happens, and too aggressive in decreasing *cwnd* when there was network congestion. With the introduction of high bandwidth-delay product (BDP) networks, AIMD was gradually replaced with MIMD (TCP CUBIC) so that the changes to the *cwnd* were more responsive. A more recent approach to congestion control was to define and maximize certain network utility functions [8, 9, 40, 42, 48]. They determine the send rate by maximizing these network utility functions either offline or online. Offline methods such as Remy [42] train the model off-line with a large amount of data and apply the trained model at the sender. Online methods such as Vivace [9] train and improve the model dynamically on-the-run. The selection of utility functions is tricky and it is not clear if such utility function would work well in general. A final class of TCP variants attempt to perform congestion control in a more conventional way: regulating the bottleneck link buffer [5, 27] . They share a common belief that managing the bottleneck link buffer is a direct way of managing network congestion. Thus, they focus on filling and draining the bottleneck link buffer by adjusting the sending rate to match the bottleneck link bandwidth.

The literature in TCP congestion control is vast. In essence, network congestion control is about determining how we can send data packets so as to fully utilize the available network resources, while avoiding network congestion. This translates to 3 tasks: i) regulate the sending of data packets (*Packet Regulation*); ii) detect network congestion (*Congestion Signal*); iii) adjust the sending of data packets to and avoid and ease the congestion (*Congestion Avoidance and Recovery*). Hence, to understand the relation between our work and previous work, we can organize the previous algorithms according to their differences with respect to how they handle *Packet Regu-*

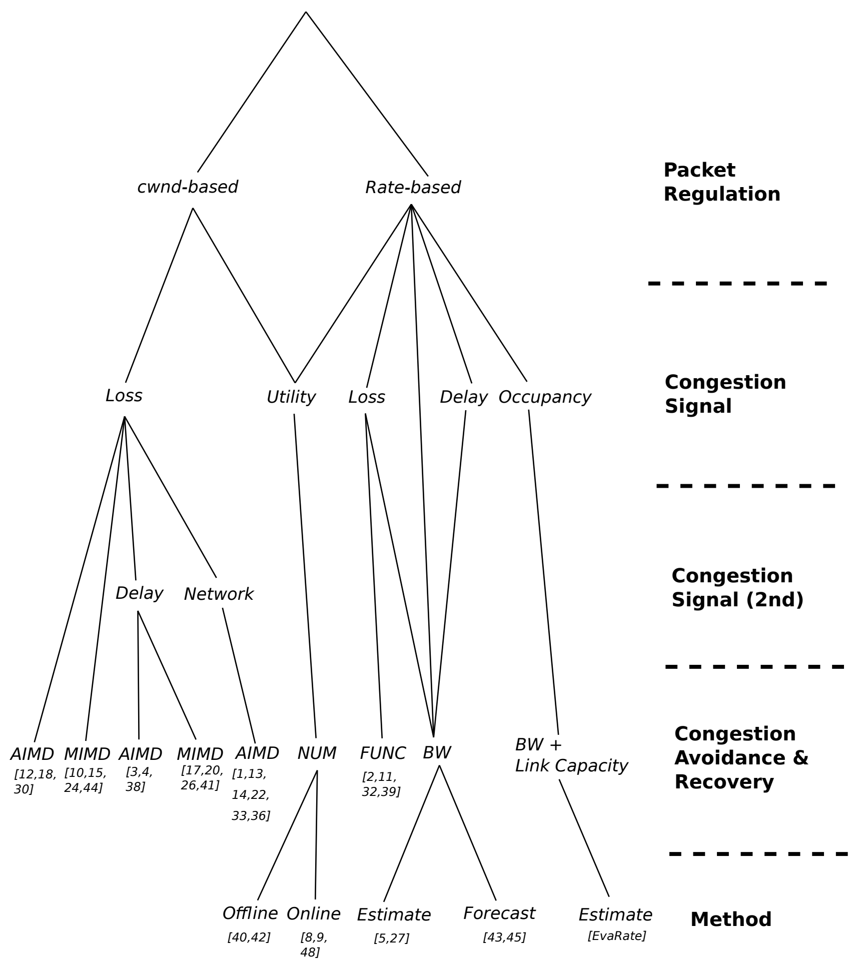


Figure 2.1: Summary of TCP design space.

lation, *Congestion Signal* and *Congestion Avoidance and Recovery*. This is summarized in Figure 2.1,

In this chapter, we first review TCP Tahoe, a *cwnd*-based and loss-based TCP variant that is the foundation of modern TCP congestion control [18]. We describe the *Slow Start*, *Congestion Avoidance* and *Congestion Recovery* states. Next, we discuss traditional *cwnd*-based TCP variants that use not only packet loss, but also network delay and network information to regulate latency. The problem of such TCP variants is: 1) they achieve low latency at a cost of lower throughput; 2) they are *cwnd*-based, which has only indirect control over the sending rate, and thus reacts to network variations slower. Finally, we describe and discuss state-of-the-art solutions that either use new congestion signals (utility function, bottleneck bandwidth and bottleneck buffer occupancy) or apply innovative congestion avoidance and recovery methods (network utilization maximization, function, and network model).

2.1 Loss-Based TCP

Van Jacobson was the first to report a series of congestion collapses and suggested that they were caused by the improper congestion control mechanism in the original 4.3 BSD (Berkeley UNIX) TCP implementation. TCP Tahoe was proposed to enhance the original TCP protocol, completing the basic mechanism and functionality of congestion control and forming the foundation of modern TCP [18]. TCP Tahoe paces the sending of data packets by adjusting the congestion window *cwnd* after an ACK packet is received or a data packet was lost. This is also known as *ACK-clocking*. It used packet loss as a signal of network congestion, and reduces *cwnd* to ease the congestion when packet loss occurs. The original 4.3 BSD TCP implementation assumed that packet losses are only caused by congestion in the network, which was not true because there were random losses in the physical layer of the networks. TCP Tahoe mainly focused to solving two problems. First, the Go-Back-N automatic repeat request mechanism in the original TCP was inefficient because a packet loss in the network could cause multiple retransmissions of subsequent packets. Secondly, letting receiver control the send window was likely to cause severe network congestion due to the receiver residing in distinct networks from the sender. In order to address these issues, 3 traffic management techniques or states were introduced to the original TCP protocol: *Slow Start*, *Congestion Avoidance* and *Fast Retransmit*. In this section, we describe the 3 basic states introduced by TCP Tahoe that all modern TCP implementations have in common.

Slow Start. As defined in the original TCP protocol, each TCP connection starts with the sender injecting a batch of packets until the number of in-flight packets reaches the window size allowed and set by the receiver. This mechanism works well when the sender and receiver are located in the same local area network (LAN), which means that the links in the route share similar characteristics, including delay, throughput and packet loss rate. Nevertheless, problems can arise when there is a mixture of both fast and slow links along the route. Under such condition, an initial burst of packets might overwhelm the network link and breakdown the whole network. Therefore, in the *Slow Start* state, a congestion window, *cwnd*, is introduced to the per-connection state [18]. When starting a new connection or restarting from a packet loss event, the system enters the *Slow Start* state, and the *cwnd* is set to 1, and is increased by 1 for each ACK packet received from the receiver. The sender sends the minimum of the receiver's advertised receive window and *cwnd*. The *Slow Start* state continues until *cwnd* reaches the *Slow Start* threshold, *ssthresh*, after which the sender enters the congestion avoidance state.

Congestion Avoidance. The congestion avoidance state regulates the sending of packets and adapts to network variations and congestion. TCP

Tahoe initially introduced additive increase multiplicative decrease (AIMD) policy to adjust the congestion window *cwnd* accordingly. When encountering a packet loss, which is a signal of network congestion, the slow start threshold *ssthresh* is set to half of the current congestion window *cwnd*, then *cwnd* is reset to one and the sender switches back to the *slow start* state. With these operations, the sender slows down the sending rate to mitigate the network congestion.

Fast Retransmit. A major problem with the timeout triggering retransmission mechanism in the original TCP protocol was that a timeout can be relatively long, which means that the sender will have to wait for a long time to resend the lost packet, thus causing degradation in network performance. TCP Tahoe uses the fast retransmit mechanism to resend any likely lost packets before the timeout event occurs. The motivation behind is that if duplicate ACKs for an in-flight packet are received, most probably this packet has been lost en route. Specifically, TCP Tahoe assumes that three duplicate ACK packets would indicate the loss of a packet, triggering the *Fast Retransmit* state. The sender performs the retransmission of the lost packet without waiting for the timeout timer to expire. Subsequently, the slow start threshold *ssthresh* is adjusted to half of the current congestion window *cwnd*, *cwnd* is reset to the initial value one, and finally it returns to the *Slow Start* state.

2.1.1 Attempts to Improve TCP Tahoe

Although TCP Tahoe defined the fundamental components of TCP implementation, several performance problems were found and inspired new TCP variants. Like TCP Tahoe, these new algorithms also used *cwnd*-based packet regulation, loss-based congestion signal, and AIMD mechanism to adjust *cwnd*.

It was observed that random packet losses were common in the network. In such a scenario, it was obviously not wise to empty the link and restart the connection, which was the case in TCP Tahoe. Furthermore, even though a packet has a high probability of being lost if three duplicate ACKs were received, it does not necessarily imply that the network congestion is serious because the subsequent three segments have been successfully received by the receiver. In this light, TCP Reno was proposed to overcome TCP Tahoe's drawback of reducing the sending rate too aggressively after duplicate ACKs are received.

TCP Reno inherited all the techniques from TCP Tahoe including *Slow Start*, *Congestion Avoidance* and *Fast Retransmit*, but the *Fast Retransmit* state was replaced with a *Fast Recovery* state. Upon receiving three duplicate ACKs, TCP Reno will halve the current congestion window *cwnd* instead of resetting it to the initial value one, set the slow start threshold *ssthresh* to the new *cwnd*, and enters the *Fast Recovery* state. In this state, TCP Reno retransmits the possibly lost packet that is indicated by the three duplicate ACKs. Then, for each duplicate ACK, *cwnd* is increased by one. This state continues until a new ACK arrives, which implies that the retransmitted packet has been received, and TCP Reno switches back to the *Congestion Avoidance* state, setting *cwnd* to *ssthresh*.

The principal limitation of TCP Reno is that it only retransmits the first lost packet in the current window, which fails to detect other possible losses within the window and thus could lead to the timeout of other lost packets. As a result, TCP Reno will only perform well when there was single packet loss within a window of data. Intuitively, TCP Reno would perform poorly if deployed in high packet-loss networks such as WiFi, and at worst it works just like TCP Tahoe.

To enhance the performance in the event of multiple packet losses, TCP NewReno was proposed in RFC 2582 [12]. TCP NewReno differs from TCP Reno only in the *Fast Recovery* state. In TCP Reno, once a fresh ACK is received, it leaves the *Fast Recovery* state and enters the *Congestion Avoidance* state. Instead, TCP NewReno takes note of each outstanding packet when it enters the *Fast Recovery* state, and does not exit until all those outstanding packets are acknowledged. Therefore, when a fresh ACK is received, there are two cases: (i) if it acknowledges all outstanding packets logged upon entering the *Fast Recovery* state, then TCP NewReno exits, sets the congestion window *cwnd* to *ssthresh*, and goes to the *Congestion Avoidance* state like TCP Reno. (ii) If it is a partial ACK, then TCP NewReno retransmits the next lost packet, and stays in the current state. Eventually all outstanding packets will be acknowledged.

A major limitation of TCP NewReno is that it takes one round-trip time (RTT) to detect each packet loss. It would be much more efficient to detect and retransmit all the lost packets in one go. To this end, a new TCP option called TCP Selective Acknowledgment (TCP SACK) was proposed in RFC 2018 as an extension of TCP Reno and TCP NewReno [31]. When enabled, TCP SACK requires packets are acknowledged selectively instead of cumulatively. Hence, each time an ACK is received, an extra piece of data is integrated into the segment header, that describes which blocks of out of order packets have arrived at the receiver. Similarly, TCP SACK enters the *Fast Retransmit* state when the sender receives 3 duplicate ACKs, and exits only when all outstanding packets upon entering the state are acknowledged, with the congestion window *cwnd* set to *ssthresh*. However, instead of transmitting all unacknowledged packets in the current window, TCP SACK retransmits all the packets that are missing.

TCP Westwood is a TCP variant that attempts to improve the TCP performance for heterogeneous networks (wired and wireless) where segment losses are often due to lower layer link errors instead of network congestion [30]. Under such condition, standard TCP such as TCP Reno will perform poorly because all segment losses are treated as a signal of network congestion, and the congestion window is reduced drastically to half of its current value, leaving much available link capacity wasted. The key idea is to select *cwnd* and *ssthresh* according to the current available link bandwidth when the congestion occurs rather than simply halving the current *cwnd*, which is too aggressive particularly when the loss is caused by a link error. Westwood uses an end-to-end method to measure the receive rate by exploiting the information in ACKs. By counting how many segments (or bytes) have been received during the last ACK and the current ACK, it's able to compute a new measurement of throughput. Westwood employs a low-pass filter to average sampled throughput and get rid of outlier noise. Having the filtered measurement of throughput *est_bw*, Westwood sets *ssthresh* and *cwnd* to be $est\_bw * RTT_{min}/seg\_size$ after three duplicate ACKs or coarse timeout expiration. This process guarantees the restart level after a loss is consistent with the available bandwidth.

2.1.2 Improving Throughput for Large-BDP Networks

With the development of faster physical links, it was found that the traditional way of increasing *cwnd* is too slow for large bandwidth-delay-product (BDP) networks. The reason is that AIMD increases its *cwnd* slowly after packet loss events, and thus it takes a long time to recover to the state before a random packet loss, causing under-utilization of the network bandwidth. This issue is especially serious in large bandwidth-delay product networks because they require an extremely high *cwnd* to fully utilize the bottleneck network bandwidth. Therefore, a number of TCP variants were proposed to address this issue by increasing *cwnd* more aggressively with multiplicative increase on an ACK packet and decreasing *cwnd* less aggressively on a packet loss event. These variants are mainly *cwnd*-base, loss-based, and use MIMD to adjust *cwnd*.

Scalable TCP (STCP) [24], HighSpeed TCP (HSTCP) [10] and H-TCP [26] were the attempts to mitigate the impact of random packet losses on the throughput performance by using MIMD mechanism. The basic idea is to make the increment and decrement of *cwnd* on each ACK and packet loss a function. For each acknowledgment in a RTT when no congestion has been detected,

$$cwnd \leftarrow cwnd + i()$$

and for the detection of segment losses in a RTT

$$cwnd \leftarrow cwnd - d()$$

For STCP, $i() = 0.01$ and $d() = 0.125 * cwnd$. This approach adapts *cwnd* independent of its size, thus making it scalable to various types of network links, such as bulk transfer in high-speed wide-area networks (WAN) where *cwnd* is usually large due to the high bandwidth-delay product (BDP). For HSTCP, $i()$ and $d()$ are two complicated functions of a threshold of *cwnd*, loss rate and the current *cwnd*. It achieves a significantly larger steady-state congestion window size compared with the standard TCP, which translates to higher average throughput and better ability of recovering from transient losses in network links.

Unlike STCP and HSTCP, the key idea of H-TCP is to make $i()$ a function of δ, the time elapsed since the last packet loss event. That says, the longer it is since the last congestion event, the faster H-TCP should increase *cwnd* to utilize available bandwidth. For the sake of backward compatibility, a threshold δ_L is introduced to separate low and high speed modes. When $\delta \leq \delta_L$, α is set to one, which is the default increase rate in the congestion avoidance state regulated in standard TCP; when $\delta \geq \delta_L$, α is set as a function of δ. The basic assumption here is that the time interval between segment losses in highspeed networks is long. Hence H-TCP is theoretically unable to handle network types with a high loss rate such as WiFi.

Xu et al. found the issue called *RTT unfairness* in other existing TCP protocols, which stems from the preference of low RTT flows over large RTT flows, because *cwnd* increase faster for a flow with lower RTT. Binary increase congestion control (BIC-TCP) was proposed as a solution to ensure linear RTT fairness under large congestion windows [44]. They proposed a new *cwnd* adjustment technique *binary search increase*, combined with a new additive increase mechanism, to address the issue of *RTT unfairness*. The basic idea is that the target window size *target_win* should be midpoint of *max_win* and *min_win*, where *max_win* is set to the last *cwnd* at which a congestion event is detected, and *min_win* is supposed to be the window size without any segment losses observed. Specifically, when segment losses occur, *max_win* will be set to the current *cwnd*, and the reduced congestion window will become the new *min_win*; when *cwnd* reaches *target_win*, *min_win* will be replaced with the current *target_win*, which is then updated based on the new *min_win* and *max_win*. In addition, to achieve TCP friendliness, BIC-TCP engages only when *cwnd* is beyond a threshold, otherwise standard TCP takes over the job.

CUBIC is an improved version of BIC-TCP and has been the default TCP algorithm in Linux kernel since 2006 after version 2.6.18 [15]. The motivation is to retain the concave and convex behavior before and after the equilibrium point respectively while simplifying the complicated *cwnd* adjustment algorithm in BIC-TCP, which adds complexity to analyzing the performance, tuning parameters for optimization, and implementing the system. The solution turns out to be a cubic function of the elapsed time since the last congestion event, as indicated by its name. It produces a smoother variation of *cwnd* as well as solves several problems remained in BIC-TCP. Firstly, it is still too aggressive to increase *cwnd* in the *additive increase* phase, particularly in lowspeed networks. CUBIC employs a cubic function to perform a smoother increase during this time period. Secondly, the real time nature of CUBIC makes it TCP friendly regardless of RTT being short or long. Additionally, it improves TCP friendliness by considering the long term convergence window size of traditional AIMD mechanism. In general, CUBIC inherits property of stability and scalability from BIC-TCP with a more a elegant solution, and enhances the TCP friendliness and RTT fairness.

2.2 Reducing TCP Latency

As interactive applications such as online games and video conferencing become more common, the poor latency performance of conventional loss-based TCP variants has become a significant drawback. It was found that the Bufferbloat problem, which causes large latencies due to the deep buffers deployed in modern networks, was getting more and more prevalent [19]. It was well-known that packet loss is a poor indicator of congestion [22], as the sender will fill the buffer to full until buffer overflow happens and packets are getting dropped, causing serious latency problem. Also it can only indicate whether or not there is congestion, without offering any information regarding the degree of congestion and how to adjust the sending rate. It was found that the Bufferbloat problem was serious in not only traditional wired networks, but also mobile cellular networks where larger buffers are deployed to absorb network variations and improve link utilization [46, 28]. This motivated a new class of congestion control mechanism that takes not only packet loss, but also network delay into consideration. These TCP variants employ both loss-based and delay-based congestion signaling.

2.2.1 Delay-based TCP

TCP Vegas is a loss-based and delay-based TCP implementation based on modifications to TCP Reno, that tried to achieve more efficient use of the available bandwidth and better prevention of packet losses due to over-utilizing network links [3].

Vegas differs from TCP Reno by applying a proactive congestion avoidance mechanism rather than the reactive one like TCP Reno. TCP Reno detects and reacts to the network congestion with segment losses as a signal of link over-utilization, and then reduces its sending rate. Therefore, basically TCP Reno oscillates between congesting network links and mitigating the congestion back and forth. Vegas, on the other hand, attempts to prevent over-utilizing the network link by comparing the measured throughput and the expected throughput, and regulating the transmission before congestion really happens.

The general idea of the new congestion avoidance mechanism is to maintain the "right" amount of extra data in the work [3]. For every RTT, Vegas computes the expected throughput by dividing the current window size by the minimum RTT for the connection which translates to the situation when the network is not congested, and calculates the current actual throughput by counting how much data is delivered within the time period of the sending and acknowledgment of a distinguished segment. Theoretically, the actual throughput should not exceed the expected throughput, otherwise the minimum RTT has to be updated to obtain a higher expected throughput. If the expected throughput exceeds the actual throughput too much, which implies more than capacity of segments have been sent, Vegas will decrease *cwnd* linearly in the next RTT. Instead, if too little is the difference, Vegas will increase *cwnd* linearly for the next RTT. The overall goal is to keep the extra data in the network neither too much nor too little. The problem of Vegas is that it is too sensitive to delay and thus competes poorly with CUBIC flows, which fill the bottleneck buffer without considering delay.

While the Linux kernel uses CUBIC [15] as its default congestion control module, Microsoft developed Compound TCP (CTCP) [41] for its WINDOWS OS. Microsoft's CTCP algorithm combines the traditional TCP Reno AIMD window algorithm with an additional delay-based window. The final *cwnd* is the sum of these two windows. The delay-based window increases when the RTT is small to quickly probe for more bandwidth. When queuing is detected from an increasing RTT, the delay-based window is decreased to keep the total `cwnd` constant. This approach combines both packet loss and RTT to detect congestion.

FAST TCP uses queuing delay as the primary congestion measure to adjust *cwnd*, while reacting to segment loss as well [20]. The authors argue that queuing delay is a better measure than segment loss. Firstly, each measurement of queuing delay provides multi-bit information rather than the single bit information by a segment loss event. Secondly, packet loss events in highspeed networks needs to be rare, making it difficult to accurately estimate the packet loss probability. Additionally, it was shown that the dynamics of queuing delay has the right scaling with respect to network capacity.

FAST TCP behaves like HSTCP, STCP and H-TCP. It updates *cwnd* every 20 ms with the measured average RTT and the minimum RTT observed so far, in that the relation between these two metrics reflect the queuing delay indirectly. It differs from the three TCP variants at the way *cwnd* evolves with regard to a delay threshold served as an indicator of the equilibrium state. FAST TCP increases *cwnd* aggressively when the delay is well below the threshold, and gradually reduces the increasing rate when the delay becomes closer to the threshold. It behaves symmetrically to reduce *cwnd* when the delay exceeds the threshold. This window adjustment mechanism achieves more stable performance around the equilibrium rather than those in HSTCP, STCP and H-TCP, which adjust *cwnd* without taking into consideration the relative position of the current state and the target state.

Martin et al. used increases in RTT as an indicator of congestion and future loss [29]. TCP Hybla [4] was developed for use in satellite connections, as they experience large RTTs. When the RTT is large, the cwnd grows at a slower rate than flows with shorter RTT. To overcome this slow growth, TCP Hybla takes as reference the RTT of a fast wired connection and increases the cwnd more aggressively to match the throughput to the reference connection. Low extra delay background transport (LEDBAT) is another algorithm that uses delay as the congestion signal [38].

Delay-based schemes have also been proposed for use in data centers [25, 32]. They perform better in data centers because the delay measurement is more accurate and reliable in a highly controlled environment. All these algorithms work by adjusting the *cwnd* which determines the maximum allowed number of outstanding unacknowledged packets.

These delay-based TCP variants are *cwnd*-based, and use delay as a supplement information to loss, so they inherit the problems of *cwnd*-based and loss-based TCP variants. In addition, being sensitive to delay means they cannot compete well with loss-based CUBIC, which prevents them from being deployed.

2.2.2 Improving Performance with Network Information

Another class of TCP variants rely on network support to improve the latency performance by detecting the network congestion earlier and reacting to it faster. Explicit congestion notification (ECN) was firstly introduced in RFC 3168 to help sender detect network congestion with an ECN bit set by routers. The sender can then react accordingly by checking the ECN bits of ACK packets to ease network congestion [36].

XCP was among the earliest algorithms that exploited ECN for congestion control [22]. In the design, an XCP router calculates the desired sending rate of each flow, and feeds the information back to the sender. The sender adjusts its *cwnd* based on the information carried in the header. Data center TCP (DCTCP) is a new TCP variant that uses multi-bit information from ECN to perform congestion control in data centers [1]. On a congestion event, DCTCP decreases *cwnd* based on the proportion of ACK packets with ECN bits marked, which performs well in data centers. Random early detection (RED [13]) and CoDel [33] are two attempts to assist senders perform congestion control at switches and routers. They try to inform senders of network congestion early and trigger congestion events to traditional *cwnd*-based TCP variants by actively dropping packets. RED starts to drop packets with a probability once the buffer occupancy is above some threshold, while CoDel starts to drop packets when they stay too long in the buffer. Both RED and Codel can effectively address the Bufferbloat problem.

ABC [14] is another recent approach that attempts to directly measure link capacity with the help of cellular base stations. In ABC's design, cellular base stations will guide mobile clients to accelerate or decelerate their sending of data packets. Although these algorithms are able to mitigate Bufferbloat problem, it is not clear how to correctly set the parameters according to various network conditions. In addition, they do not work with recently proposed low-latency TCP variants since they are loss-insensitive.

It is not surprising that we can do better if the senders have more information on the network. Unfortunately, these approaches require modifications to network switches and routers, which makes them difficult to deploy. That said, we believe that this deployment issue could be mitigated with SDN-enabled switches in the near future.

2.3 Achieving Low Latency & High Throughput

Although some earlier TCP variants managed to reduce latency, it typically came at the cost of lower throughput. State-of-the-art TCP variants attempted to achieve both high throughput and low latency. Most of them employed a rate-based instead of *cwnd*-based packet regulation, and introduced more sophisticated methods to set the sending rate. Before going through these proposals, we first review the literature for early rate-based TCP variants.

The idea of using rate information to control the sending rate of flows is not new. Padhye et al. were first to propose TCP-Friendly Rate Control (TFRC), an equation-based approach for congestion control that adjusts the send-rate based on observed loss events [35, 11]. Instead of exploiting available bandwidth as much as possible, the goal of TFRC is to keep a steady sending rate while still being responsive to network variations. TFRC uses a control equation that explicitly gives the maximum acceptable sending rate as a function of the packet size, round trip time (RTT), loss event rate, and the TCP retransmit timeout. TFRC achieves smoothly changing sending rate, which is beneficial for real time communication (RTC) applications, but at the cost of a more moderate response to transient changes in congestion as well as a sudden increase in the available bandwidth. WTCP [39] is another early rate-based algorithm designed for CPDP networks. It also uses packet loss as a congestion signal and is unlikely to perform well in fast modern networks.

Ke et al. proposed Rate-Based Pacing (RBP), which suggests pacing the sending of packets based on the current rate instead of sending them back-to-back so as to avoid multiple packet losses in the same window [23]. The motivation is that multiple packet losses will cause serious performance degradation of TCP, thus we should insert a time delay between packets so as to reduce the possibility of multiple packet losses in the same window. The time delay is a function of the round trip time (RTT) and the current congestion window size. However, this technique requires precise estimates of RTT, which are not easily available and are not actually accurate indicators of link quality in cellular data networks.

Another proposal of performing TCP congestion control using the rate information is Rate Adaptive TCP (RATCP) [21], which is not a practical approach as it requires the network to explicitly feedback the available rate to the TCP source. Also, RATCP relies on accurate estimation of round trip time (RTT), which is in general very difficult to obtain.

These approaches do not fit modern networks because they either require network support or set the sending rate based on wrong metrics, such as loss. They might work well previously, when loss was prevalent, but perform poorly today where network loss is rare.

2.3.1 Forecast-based TCP

Forecast-based TCP assumes that the network performance in the near future can be accurately predicted, and the end-to-end latency performance can be improved by adjust the sending rate based on the prediction result.

Sprout was proposed to achieve high throughput and low delay for interactive applications on cellular data networks [43]. It utilizes a doubly-stochastic process which involves a Poisson process and a Brownian motion. The receiver helps to infer the uncertainty dynamics of the network link by making use of the inter-arrival time of received packets, which enables the sender to forecast how many bytes should be sent in the next few time periods.

PROTEUS is a prediction based TCP variant optimized for real time communication (RTC) applications on cellular data networks that works under UDP [45]. The basic assumption of PROTEUS is that the network performance in the near future is predictable based on the history of the three metrics: the throughput, one-way delay, and loss rate. PROTEUS exploits regression trees to perform forecasting, and passes to applications the expected performance in terms of the throughput, one-way delay and loss rate for reaction to future variations of network conditions.

In practice, they not only consume a large amount of computing power, but also perform worse than the state-of-the-art TCP variants.

2.3.2 Utility Functions

A recent class of congestion control algorithms define network performance as utility functions, with input arguments like throughput, packet loss, delay, RTT, and loss rate. Different utility functions articulate different goals. For example, delay and RTT will have more impact on the utility in low-latency TCP variants rather than throughput-oriented TCP variants. For such TCP variants using utility functions, congestion control is achieved by mathematically optimizing (or maximizing) the associated utility functions. Thus we refer to these approaches as network-utility-maximization-based (NUM-based) TCP variants. There are mainly two ways to achieve the goal: online and offline. Online methods attempt to optimize utility functions dynamically, while offline methods train a model before deployment.

Utility-function-based TCP variants defines network performance quantitatively as a utility function of network metrics such RTT, loss rate and receive rate. To achieve better network performance, they adjust the TCP parameters such as *cwnd* and sending rate to optimize the utility function. Remy [42] uses machine learning to generate an optimal congestion-control algorithm offline based on assumptions and network data, whereas PCC [8] attempts to optimize a carefully designed utility function online. Vivace [9] uses ideas from online convex optimization literature and builds upon Allegro with considerations for minimizing latency and improving TCP friendliness along with fast and stable convergence. However, Sivaraman et al. claim that gaps need to be closed before computer-generated algorithms can be deployed in practice [40]. The jury is still out on how best to do congestion control. It is likely that there is no algorithm that is optimal for all operating scenarios and the question of whether a white-box approach or black-box approach is unlikely to be settled soon [37]. Verus is a congestion control protocol that is primarily designed for highly variable channel conditions that are hard to predict [48]. Instead of attempting to predict the cellular channel dynamics, Verus uses cues from delay variations to track channel conditions and quickly change its sending window. The key idea of the Verus is find the relationship between *cwnd* and achieved delay, and maximizes the utility function based on this profiling model.

Copa [2] shares some desirable properties with EvaRate, such as bi-modal operation and periodic draining of the buffer. However, Copa's rate control loop aims to maximize a utility function whereas EvaRate *directly* regulates the overall buffer occupancy considering other flows. We believe that this is the reason why EvaRate is able to keep the buffer occupancy low and avoid losses. Like BBR, Copa is insensitive to losses and by not reacting to losses (especially in the shallow buffer case), it inflicts loss on other loss-reacting flows. In addition, all Copa flows try to maintain 1 BDP number of packets in the bottleneck buffer, which is more than enough, especially when it is a shallow buffer. Consequently, Copa achieves higher throughput at the cost of other loss-reacting flows. EvaRate, on the other hand, is not insensitive to losses. Last but not least, the theory in Copa's design is based on the assumption that all flows sharing the same bottleneck buffer have the RTT, which allows different flows to receive the latest measurement of RTT_{min} at the same time and synchronize. However, flows have different $RTTs$ in practice and they will not synchronize and measure the actual RTT_{min} accurately.

2.3.3 Modeling the Network

BBR [5] and PropRate [27] are the two state-of-the-art schemes that take the classic model-based approach. Both BBR and PropRate use control feedback-loops to keep the buffer occupancy low and achieve optimal throughput while maintaining low latency. PropRate uses buffer delay estimates as it's control variable, while BBR uses an estimate of the bandwidth-delay product (BDP).

In particular, BBR tries to estimate the bottleneck link bandwidth and match the sending rate to the estimated bandwidth. For every 8 RTT, BBR sends at 125% of the estimated bandwidth for 1 RTT to probe potential increase of the bottleneck link bandwidth, and sends at 75% of the estimated bandwidth for 1 RTT to drain the extra packets introduced in the previous phase. BBR then match the sending rate to the newly estimated bandwidth. In contrast to BBR, PropRate measures the bottleneck link buffer delay, and oscillates the sending rate around the estimated bottleneck link bandwidth. If the buffer delay exceeds an application-specified buffer delay, PropRate sends slower than the estimated bandwidth to drain the bottleneck buffer, otherwise it sends faster to gradually fills the buffer.

It was found that BBR is very aggressive and unfair when the bottleneck link buffer is shallow, causing massive losses to itself and other flows sharing the same bottleneck buffer [16]. It is because BBR focuses on keeping 1 BDP number of packets in the buffer, without considering whether the buffer size is larger than 1 BDP. In addition, it is also due to BBR ignoring packet losses.

To the best of our knowledge, EvaRate is the first algorithm to use estimated buffer occupancy as the control variable. BBR has only a single mode of operation and thus is forced to choose aggressive parameters in order to contend successfully with the predominant CUBIC. This results in issues such as large queuing delays and massive packet loss [17]. PropRate on the other hand is designed for cellular networks with proportionally fair buffer where each PropRate sender tries to maintain a target buffer delay. With shared buffers on the Internet, two PropRate senders with different target delays will exhibit different levels of aggressiveness leading to unfairness. Because EvaRate is able to estimate the buffer occupancy of other flows, it is able to operate in two different modes. As a result, it can compete well in the presence of buffer-filling flows, while also allowing collaborative congestion control between EvaRate flows sharing the same bottleneck.

Like low-latency cellular TCP variants [43, 48, 27], EvaRate uses delay instead of loss as the congestion signal. To achieve low latency, cellular TCP variants typically operate way below the point where the bottleneck buffer overflows. In EvaRate, we are more concerned about ensuring that the buffer is not emptied to keep utilization high, without incurring large latencies, and this is more challenging because unlike mobile cellular networks, the bottleneck buffers are shared among many flows.

2.4 Summary

Based on our analysis of the literature, we argue that in modern networks, *cwnd*-based packet regulation and loss-based congestion signal do not perform well, because they only indirectly control the sending rate and there is a risk of overfilling the bottleneck buffer, causing Bufferbloat. The new rate-based TCP variants solve this problem in indirect ways: (i) they either define an ad-hoc utility function and maximize it, or (ii) try to forecast the network bandwidth. We realized that it is important to determine the operating environment, or deduce what other flows sharing the same bottleneck buffer are doing, and react accordingly. If we can achieve this, all flows sharing the same bottleneck buffer can work together to keep the number of packets in the buffer small. Copa is the first TCP variant that considers the environment, achieves performance close to that for EvaRate. However, as discussed, it is based on an ad-hoc utility function and the assumption of equal RTT for all flows, which is not a realistic assumption in practice. On the other hand, EvaRate was designed from first principles by modeling the bottleneck link buffer and using a natural control loop. The result is that EvaRate can directly estimate the number of packets in the bottleneck buffer and manage the buffer accordingly. In Table 2.1, we provide a summary of the related work on TCP.

Table 2.1: Previously Proposed TCP variants.

Algorithm	**Year**	**Key Idea(s)**
TCP Tahoe [18]	1988	*cwnd*-based, Loss, AIMD
TCP New Reno [12]	1999	*cwnd*-based, Loss, AIMD
TCP Westwood [30]	2001	*cwnd*-based, Loss, AIMD
Scalable TCP [24]	2003	*cwnd*-based, Loss, MIMD
HighSpeed TCP [10]	2003	*cwnd*-based, Loss, MIMD
BIC TCP [44]	2004	*cwnd*-based, Loss, MIMD
TCP CUBIC [15]	2008	*cwnd*-based, Loss, MIMD
TCP Vegas [3]	1994	*cwnd*-based, Loss, Delay, AIMD
TCP Hybla [4]	2004	*cwnd*-based, Loss, Delay, AIMD
LEDBAT [38]	2011	*cwnd*-based, Loss, Delay, AIMD
Fast TCP [20]	2004	*cwnd*-based, Loss, Delay, MIMD
H-TCP [26]	2004	*cwnd*-based, Loss, Delay, MIMD
Compound TCP [41]	2006	*cwnd*-based, Loss, Delay, MIMD
TCP LoLa [17]	2017	*cwnd*-based, Loss, Delay, MIMD
RED [13]	1993	*cwnd*-based, Loss, Network, AIMD
ECN [36]	2001	*cwnd*-based, Loss, Network, AIMD
XCP [22]	2002	*cwnd*-based, Loss, Network, AIMD
DCTCP [1]	2010	*cwnd*-based, Loss, Network, AIMD
Codel [33]	2012	*cwnd*-based, Loss, Network, AIMD
ABC [14]	2017	*cwnd*-based, Loss, Network, AIMD
Remy [42]	2013	*cwnd*-based, Utility, NUM, Offline
TAO [40]	2014	*cwnd*-based, Utility, NUM, Offline
Verus [48]	2015	*cwnd*-based, Utility, NUM, Online
PCC [8]	2015	Rate-based, Utility, NUM, Online
PCC Vivace [9]	2018	Rate-based, Utility, NUM, Online
WTCP [39]	1999	Rate-based, Loss, FUNC
Equation-based TCP [11]	2000	Rate-based, Loss, FUNC
TIMELY [32]	2015	Rate-based, Delay, FUNC
Copa [2]	2018	Rate-based, Delay, FUNC
Sprout [43]	2013	Rate-based, Loss, BW, Forecast
PROTEUS [45]	2013	Rate-based, BW, Forecast
BBR [5]	2016	Rate-based, BW, Estimate
PropRate [27]	2017	Rate-based, Delay, BW, Estimate

Chapter 3

Low-Latency TCP in the Wild

TCP congestion control has been studied for three decades and a large number of TCP variants have been proposed to improve network performance. In particular, reducing the latency has been a recent focus and we have seen more and more low-latency TCP variants. However, the deployment status of these new TCP variants and their performance on the current Internet is not known. In other words, we would like to know if TCP CUBIC is still the dominant TCP variant on the Internet. In this chapter, we first conducted a measurement study to classify and identify the TCP version used by Alexa Top 5,000 websites, and investigate the performance of recent low-latency TCP variants in the wild using Amazon web service (AWS) servers.

3.1 Census of TCP Variants

We conducted a measurement study to understand the deployment status of various TCP variants in the Internet. Generally, TCP variants can be classified by their level of aggressiveness, where the variants in the same group exhibit similar levels of aggressiveness. We hypothesize that in a heterogeneous Internet, measuring the distributions of different groups of TCP variants is sufficient to characterize the current Internet.

Therefore, we categorize existing TCP variants into 3 types based on their aggressiveness: *Linear*, *Polynomial* and *Rate-based*. Linear variants includes the *cwnd*-based TCP variants that increase *cwnd* linearly for each RTT, like TCP Reno and TCP NewReno [12]. Similarly, Polynomial refers to the TCP variants that increase *cwnd* more and more aggressive for each RTT, like BIC and CUBIC. Rate-based TCP variants regulate the sending without *cwnd*. Instead, they maintain a certain rate and are thus not ACK-clocked. BBR [5] and PropRate [27] are the 2 examples of Rate-based TCP.

Yang et al. developed TCP Congestion Avoidance Algorithm Identification (CAAI) [47] to identify various *cwnd*-based TCP algorithms. In particular, we measure the increase pattern of *cwnd* for the TCP variant running on a remote website as follows:

1. the receiver creates a TCP connection to a remote Web server and emulates a network environment needed to measure remote server's TCP *cwnd*;

2. the receiver measures the *cwnd* of the TCP running on the remote server by counting the number of packets received during an RTT;

3. the receiver maintains the TCP connection until it has gathered a sufficiently long trace of window sizes.

We set a small maximum segment size (MSS), and artificially increase the RTT and retransmission timeout (RTO). The small MSS is needed to increase the number of data packets for the probed web pages. This allows the receiver to collect more packet trace data for more accurate identification. A long RTT is needed to ensure that the maximum number of packets in flight is larger than *cwnd*. If not, our measurement would be constrained by the maximum number of packets in flight. To estimate current TCP *cwnd* of the remote server, the receiver emulates a sufficiently long RTT so that the remote server stops sending packets after *cwnd* of packets have been sent, and before the ACK packet of the first packet reaches the server. Thus the receiver can count the number of packets received during this RTT, which is an estimate of remote server's TCP *cwnd*. Packet loss is used to trigger different behaviors of *Congestion Recovery* state. Finally, once RTO occurs, TCP will reset *cwnd* and *ssthresh* differently so that by observing the variation of *cwnd*, the TCP variant of the remote server can be identified.

We extended this approach to identify Rate-based TCP. Our key observation is that both Linear and Polynomial TCP require a long time to probe the network bandwidth, while Rate-based can quickly measure the network bandwidth and increase the sending rate to match the network bandwidth. As a result, the number of delivered packets often increases exponentially for Rate-based TCP.

In Figures 3.1, 3.2 and 3.3, we plot the increase of *cwnd* versus RTT for Reno, CUBIC and BBR. We trigger an RTO event at 14th RTT and observe the variation of *cwnd*. Note that we can trigger the RTO event at any time as long as the sender has data to send, so there is nothing special about

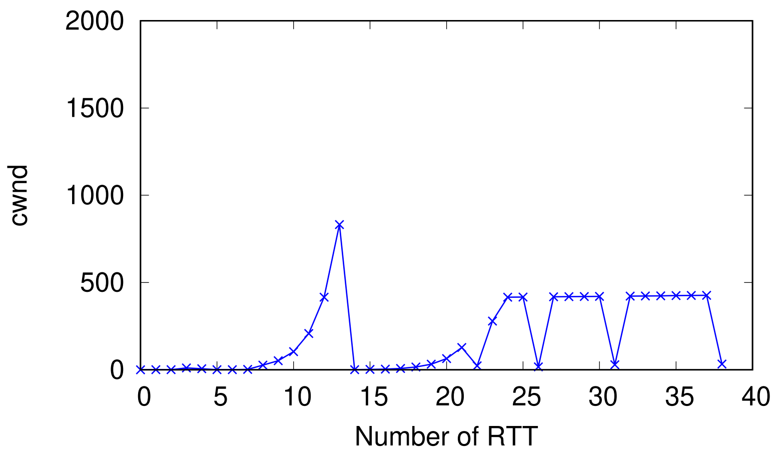


Figure 3.1: RTT vs. *cwnd* for TCP Reno.

14th RTT. We can clearly observe different levels of aggressiveness in terms of increasing *cwnd*. BBR does not maintain a *cwnd*, so what we see is the effective *cwnd*. Note also that the figure for BBR in Figure 3.3 uses a logscale for the y-axis.

We performed these measurements on the Alexa Top 5,000 websites. Among 5,000 websites, we found that 4,571 were accessible. 318 (6.96%) websites were classified as Linear and 3674 (80.4%) websites were classified as Polynomial. The remaining 579 (12.7%) were classified as Unknown because the page sizes were too small for us to discern any patterns. Among all the sites, we found YouTube's TCP variant to behave like BBR.

3.2 Performance of Low-Latency TCP

From the results in §3.1, we can see that the dominant TCP variant on the Internet today is the Polynomial buffer-filling variant TCP CUBIC [15]. As such, we would expect the recent low-latency TCP variants like BBR [5],

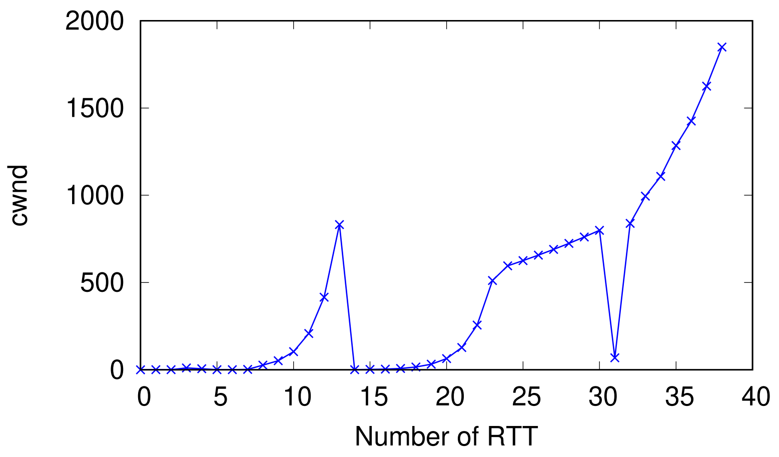


Figure 3.2: RTT vs. *cwnd* for TCP CUBIC.

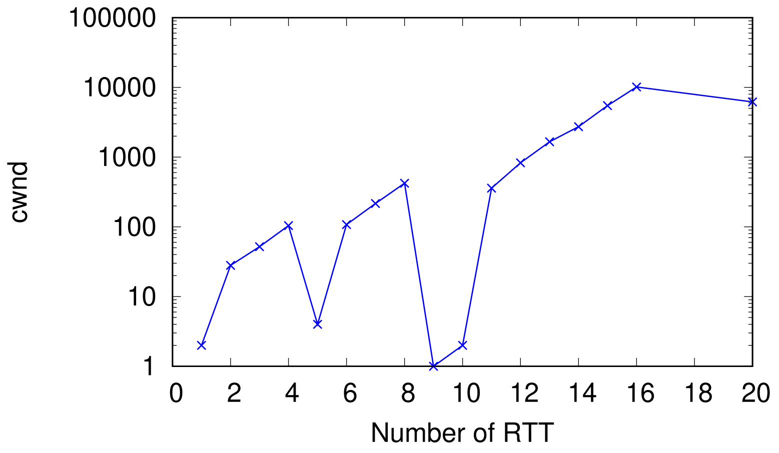


Figure 3.3: RTT vs. *cwnd* for BBR.

PropRate [27], Copa [2], and Vivace [9] to contend poorly against CUBIC. Surprisingly, when we measured the throughput and RTT of a host (in Singapore) to Amazon AWS servers located on four different continents – Australia, Asia, North America and Europe. The detailed information of the servers is in Table 3.1. We found that the effective throughput of BBR and PropRate were similar to that for CUBIC, while Copa and Vivace often outperformed CUBIC. Our results are shown in Figure 3.4 and the average RTT

Table 3.1: Information of the AWS servers.

Continent	Location	IP Address	Instance Type
Europe	London	35.177.93.102	t2.micro
North America	North California	13.57.254.199	t2.micro
Australia	Sydney	13.211.204.244	t2.micro
Asia	Mumbai	13.127.68.183	t2.micro

Table 3.2: RTT for inter-continental traffic.

Australia	Asia	North America	Europe
180 ms	80 ms	175 ms	320 ms

is shown in Table 3.2. There were only minor and statistically insignificant variations in the observed RTT for the different TCP variants.

We used `iperf` to generate the traffic for CUBIC, BBR and PropRate, and obtained the throughput for each protocol-continent pair in a round-robin manner to eliminate underlying network variations. Custom clients were used to generate traffic for Copa and Vivace (default utility functions and parameters) since only user-space UDP implementations are publicly available. We managed to obtain small error bounds after running 30 trials during the same time of the day.

It is clear that the difference in the results between the different continents was due to the difference in the RTTs. In general, CUBIC, BBR and PropRate had reduced throughput when the RTT was large, but we were not convinced that the RTT difference alone would fully explain the difference in throughput. Analyzing the traces more carefully, we discovered that one major difference between Copa and Vivace and the other variants was that the loss rates for Copa and Vivace were significantly higher when they

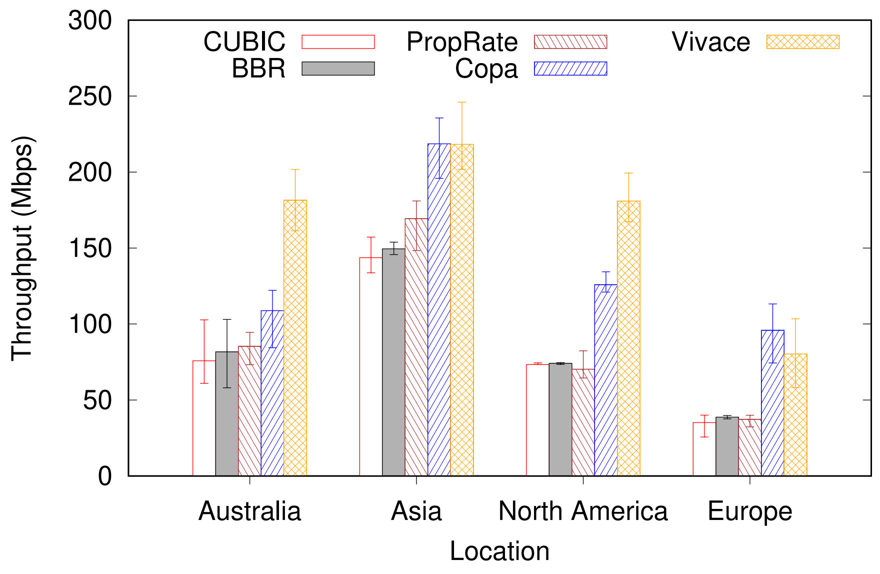


Figure 3.4: TCP throughput for inter-continental Internet traffic.

performed better as shown in Figure 3.5. On the other hand, the loss rates for CUBIC, BBR and PropRate were negligible.

We also performed another set of experiments to validate the results we obtained in the previous experiments and to further understand the sharing of bandwidth at the bottleneck links. In this new set of experiments, instead of initiating a single flow to the AWS remote servers in various continents, we started two flows to the remote servers: one CUBIC flow followed by another reference flow (for the protocol we want to test) after a delay of 20 s. After reference flow started, both flows ran for 40 s concurrently, and stopped at the same time. We calculated the throughput and loss rate for the CUBIC flow and the reference flow for the latter 40 s and plot them in Figure 3.6 and 3.7. We observed that like the single flow experiment, the low-latency TCP variants were clearly more aggressive than the background CUBIC flow, even if they started later. The loss rate of low-latency TCP

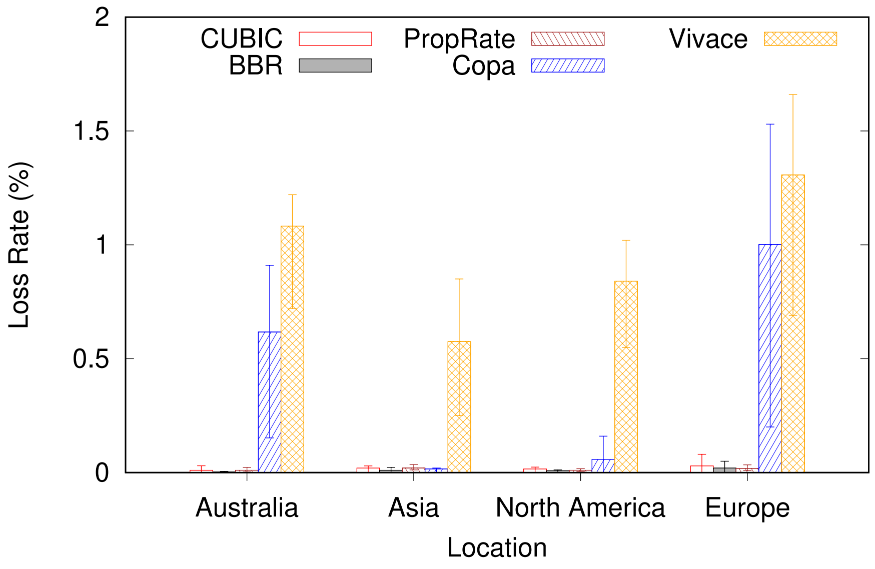


Figure 3.5: Loss rate of TCP variants for inter-continental traffic.

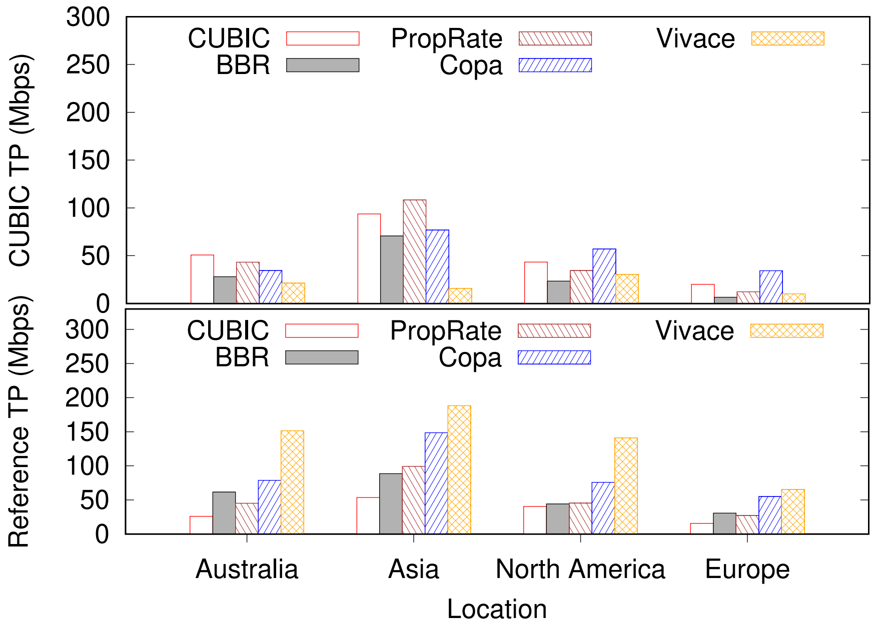


Figure 3.6: TCP throughput of CUBIC and reference flows for inter-continental Internet traffic.

variants was slightly lower than that of the single flow experiment, but still relatively significant. In addition, CUBIC behaved more friendly and was

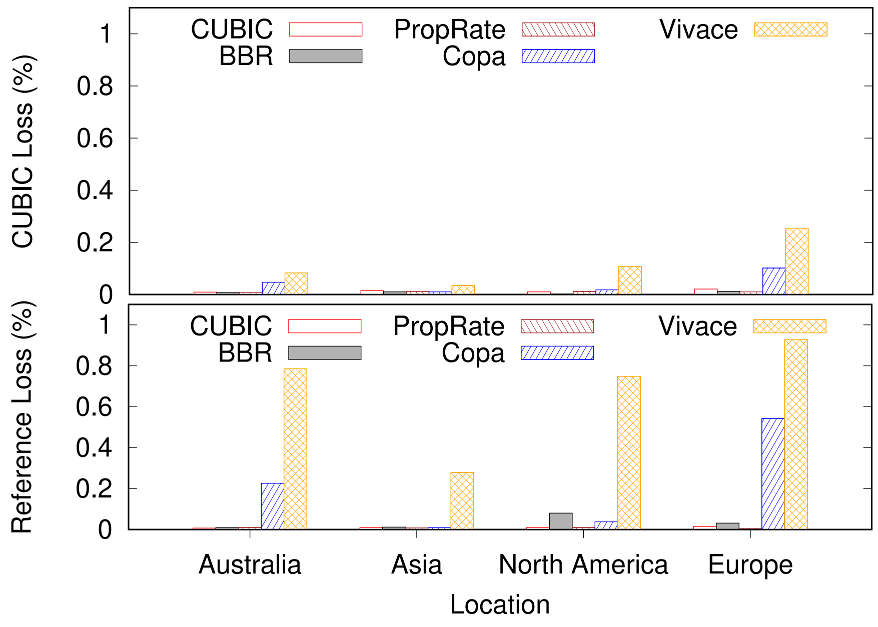


Figure 3.7: TCP loss rate of CUBIC and reference flows for inter-continental Internet traffic.

fairer to itself in this case. It is clear from Figure 3.7 that the low-latency TCP variants inflict loss on the background CUBIC flow, possibly up to 0.2%.

In Figure 3.8, we plot the throughput for the reference flow against the background CUBIC flow. We can see that CUBIC and PropRate are somewhat fairer to the background CUBIC flows than the low-latency flows. Vivace seems particularly aggressive and unfair.

Given the seemingly superior performance of Vivace and Copa, one could imagine that they could become more and more common, and the natural question is: should we be concerned with the loss that we observed? We suspect that the loss rates were likely caused by buffer overflows at some bottleneck in the core Internet. This means that flows sharing the same bottleneck will also experience loss. Given that we are currently seeing negligible

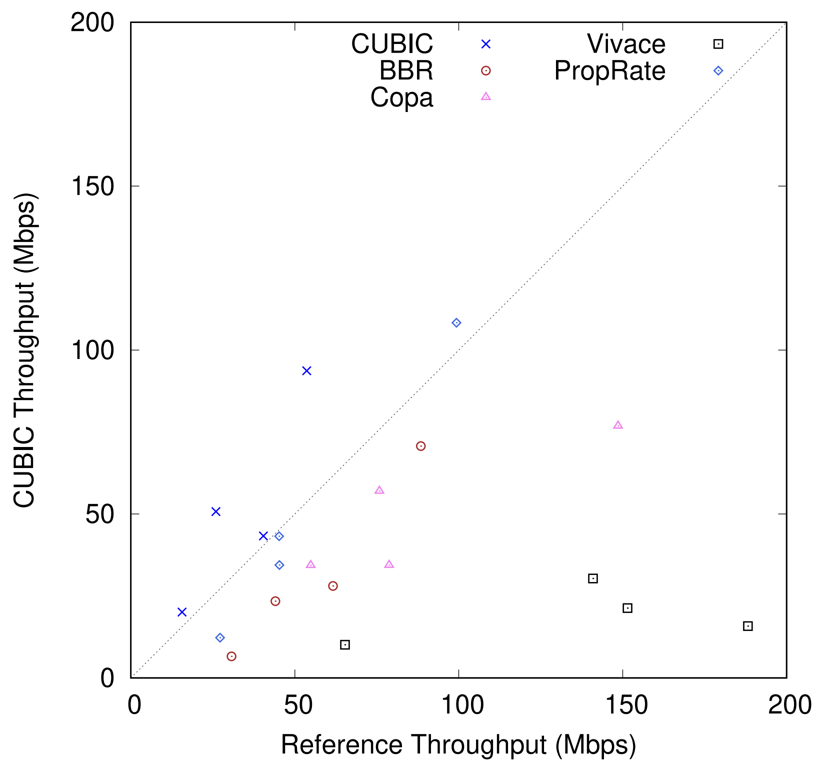


Figure 3.8: Throughput of CUBIC flows vs. reference flows for inter-continental traffic.

loss with CUBIC, the intrinsic characteristic of the Internet might change and it might become more lossy if Vivace and Copa become more common. To understand the likely impact of increasing loss rates, we performed a simple experiment, where we measured the throughput of a TCP flow on a 32 Mbps link as we varied the loss rates of the bottleneck link from 0 to 1% for both a large RTT (300 ms) and a low RTT (80 ms) link. We also investigated the impact of a shallow (roughly 1 × BDP) buffer and a deep (2,000 packet) buffer. The results are shown in Figure 3.9.

Given that CUBIC is throttled by losses, it should not be surprising that even a slight increase in loss rates from zero to 0.1% can result in the

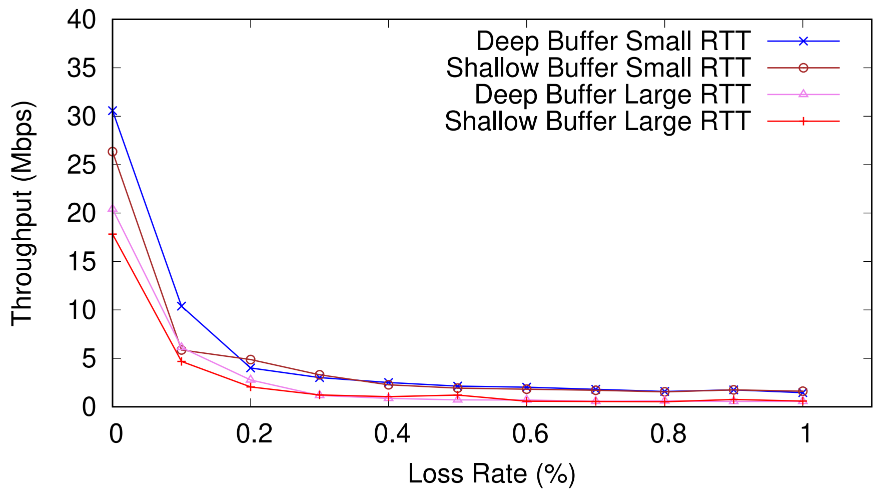


Figure 3.9: Impact of loss on CUBIC throughput.

significant degradation of CUBIC's throughput. In other words, it would not be surprising if the superior performance that we observed for both Copa and Vivace on the Internet is due to them successfully throttling CUBIC with packet losses. If so, this would be extremely troubling because it means that low-latency variants like Vivace and Copa are actually TCP unfriendly!

Unfortunately, low-latency TCP is fundamentally incompatible with CUBIC: it is impossible for a TCP variant to keep the latency low when the bottleneck buffer is saturated by a CUBIC flow to the point of buffer overflow. Intuitively, it is also clear that a TCP variant which tends to occupy a larger proportion of the buffer will be likely more aggressive and take up a larger share of the available bandwidth. However, what Copa and Vivace seem to suggest is that it is possible to still contend successfully with CUBIC by causing packet losses.

We felt it is philosophically problematic for the evolution of the Internet to try to achieve low latencies while simultaneously causing packet losses in the network. In this thesis, we argue that there is a better way: (i) we can work on keeping the number of packets in the buffer small at all times to keep losses low (and avoid changing the character of the Internet); and (ii) flows that can detect the total occupancy of the shared bottleneck buffer attempt to keep buffer occupancy low *collaboratively*.

Chapter 4

Loss-Free Congestion Control

From the results of our measurement study in Chapter 3, we hypothesize that Copa and Vivace perform well in practice because the buffers on the Internet are relatively shallow and Copa and Vivace are insensitive to loss compared to conventional loss-based flows like CUBIC. This means that CUBIC will back off quite conservatively, leaving enough free bandwidth to the low-latency TCP variants. But we have also seen in Figure 3.9 that a small loss rate can cause significant degradation to CUBIC flows. Hence to operate effectively in the current Internet without degrading competing CUBIC flows, a new low-latency TCP should be designed to minimize the loss it causes to competing flows to allow the Internet to transition smoothly to a benevolent future. Our key insight is that we can directly estimate not only the buffer occupancy for our flow (§4.1), but also that of competing flows sharing the same bottleneck buffer (§4.2). By implementing a control loop that keeps the buffer occupancy low, EvaRate keeps latency low and avoids overflowing the buffer and inflicting loss on competing flows.

Like existing TCP variants, EvaRate adopts a *slow start* phase where it tries to rapidly increase the sending rate to match the available bandwidth. Also, like the previous algorithms [27], we send 10 initial packets and wait for the corresponding ACKs to have an initial estimate of the receive rate. However, instead of doubling the sending rate after every RTT (like CUBIC or BBR), we increase the rate approximately only 1.5 times every RTT until we do not observe a significant increase in the receive rate. This is because, while doubling the sending rate would allow us to converge to the available bandwidth faster, it also risks overshooting the available bandwidth by a significant amount. Once we detect that the current sending rate is higher than the estimated receive rate, we switch to congestion avoidance mode. EvaRate goes back to the slow start phase only when the retransmission timeout (RTO) expires, upon which EvaRate discards the measured information of the network and sends a burst of packets to reprobe the network.

In *congestion avoidance mode*, EvaRate adopts a control loop which uses the estimated real-time buffer occupancy as a feedback signal to regulate the sending rate. The sending rate is adjusted such that the flow's buffer occupancy converges to a target occupancy, B_{target}. The initial B_{target} is determined by the maximum tolerable latency specified by the application, while the real-time buffer occupancy is estimated from the number of packets in flight, RTT and the estimated receive rate (see §4.1).

To achieve full link utilization, we only need to ensure that the buffer occupancy is non-zero. At the same time, we want to keep the buffer occupancy low in order to minimize the buffer delay. To do so, we adopt a time-varying B_{target} which decreases, or *decays*, exponentially with some half-life.

As B_{target} decays to a low value, the buffer occupancy becomes closer to zero due to the sending rate control loop trying to match the buffer occupancy to B_{target}. At this point, the buffer is at the risk of emptying and we know that B_{target} might have decayed to a value that is too low. Therefore, we increase B_{target} and resume the decay process. This mechanism has two additional benefits: (i) it allows newly joined flows to grab a share of the bottleneck bandwidth, and (ii) in benevolent environments, it allows EvaRate flows to collaboratively maintain the total buffer occupancy low.

In the following sections, we describe EvaRate's design in detail. We first explain the buffer-occupancy-based control loop used to regulate the sending rate (§4.1). We then describe how EvaRate infers the operating environment using an augmented model of the buffer (§4.2). Finally, we describe how EvaRate operates differently in the two operating scenarios (§4.3).

4.1 Aggregate Network Model

Consider a simplified model of the network pipe between two end hosts where there is a bottleneck buffer somewhere along the pipe (see Figure 4.1). The simplification and assumption of an aggregate bottleneck link is also used by many TCP variants, such as BBR and Copa [5, 2]. While the outgoing link of a buffer typically has a fixed capacity in wired networks, sharing the bottleneck link with other flows will make it seem to our flow that the bottleneck bandwidth is a time-varying function $\rho(t)$. We note that $\rho(t)$ can be measured at the receiver as the receive rate. We denote $B(t)$ as the number of packets in the bottleneck link buffer, t_{buff} as the time a packet

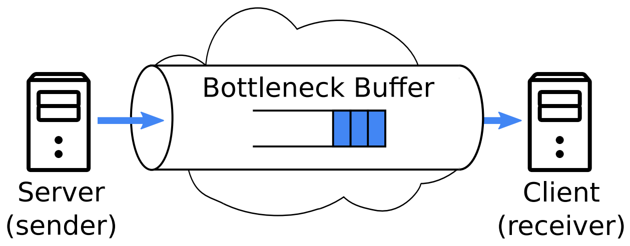


Figure 4.1: Simplified model of network bottleneck.

spends in this buffer and L as the end-to-end latency (RTT) between the sender and the receiver. Assume that we can measure the minimum RTT (denoted as RTT_{min}) i.e. the two-way propagation delay between the sender and receiver (refer §4.4.2 for details), then:

$$t_{buff} \approx L - RTT_{min} \tag{4.1}$$

And since $t_{buff} = \frac{B}{\rho}$,

$$B \approx \rho(L - RTT_{min}) \tag{4.2}$$

Equation (4.2) articulates the relationship between buffer occupancy (B) and end-to-end latency (L). In other words, to achieve a desired latency no more than L, the buffer occupancy needs to be kept below $\rho(L - RTT_{min})$.

In Figure 4.2, we illustrate the relationship between the sending rate and the variation in the buffer occupancy, where $S(t)$ is the sending rate, and $\rho(t)$ is the time-varying receive rate at time t. It is clear that:

$$B(t) = \int S(t) - \rho(t)\, dt \tag{4.3}$$

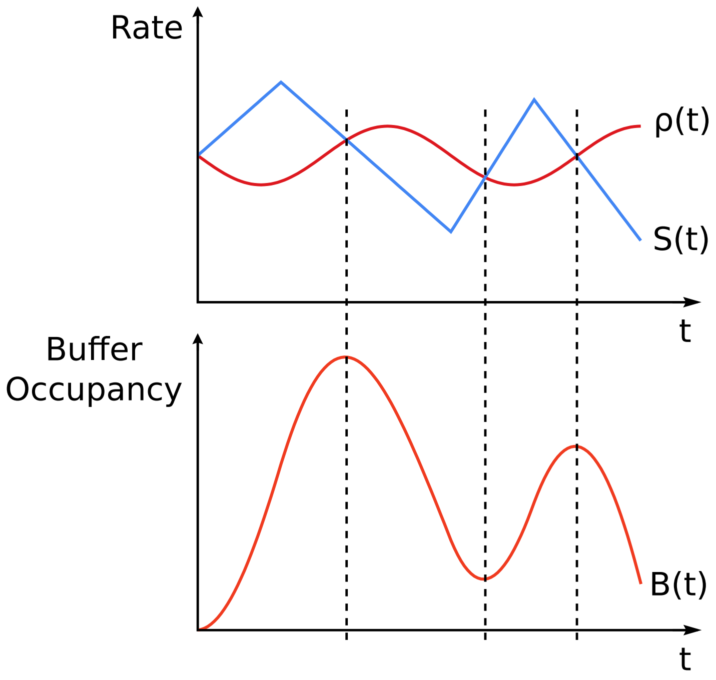


Figure 4.2: Relationship between the sending rate and buffer occupancy.

Basically, when we send packets at a rate above the receive rate, the buffer fills; on the other hand, if we send at a rate below the receive rate, the buffer drains.

Our key insight is that, in practice, we can estimate the number of packets in the buffer, $B(t)$, rather accurately as the difference between the number of unacknowledged packets in flight, P_{if}, and the number of *on-wire* packets in the pipe. The former is known by the sender and the latter can be estimated as the bandwidth delay product $\rho \times RTT_{min}$, where the receive rate ρ is an estimate of the bandwidth, while RTT_{min} is an estimate of the two-way propagation delay between the sender and the receiver. Therefore, the buffer occupancy $B(t)$ is given by,

$$B(t) \approx P_{if}(t) - \rho RTT_{min} \tag{4.4}$$

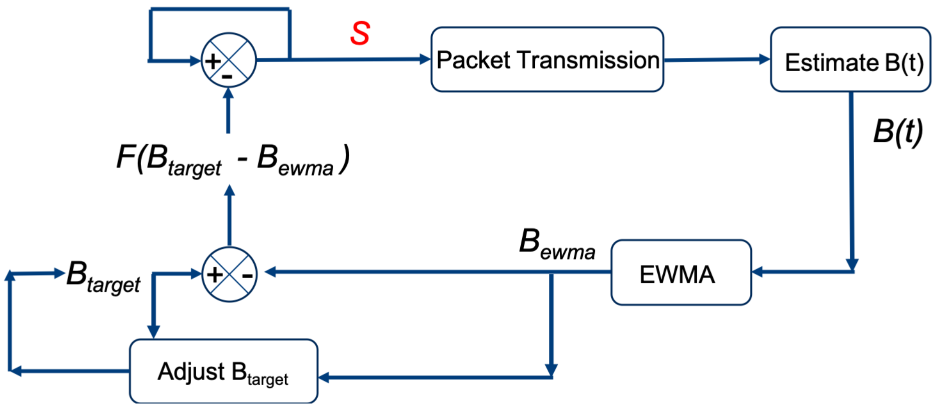


Figure 4.3: The negative-feedback loop to manage the bottleneck link buffer.

Given $B(t)$ and the receive rate ρ (see §4.4.1), we can increase buffer occupancy by setting $S(t) > \rho$ and drain the buffer by setting $S(t) < \rho$. The goal of the control loop is to adjust the sending rate such that the bottleneck buffer occupancy $B(t)$ converges to a target buffer occupancy, B_{target}. To do so, we use a negative-feedback loop as shown in Figure 4.3 to update $S(t)$ as follows:

$$S(t) = \alpha\rho(1 - k\frac{B(t) - B_{target}}{B_{target}}) \tag{4.5}$$

where α is an aggression factor that can make EvaRate more aggressive in the presence of buffer-filling flows and k is a constant that controls the rate of convergence to the target buffer occupancy B_{target} (see §4.4.4). Basically, given the measured receive rate ρ, we want to send faster than ρ when $B(t) < B_{target}$, and slower than ρ when $B(t) > B_{target}$. Note that the negative-feedback loop's aggressiveness in updating $S(t)$ is proportional to the difference between $B(t)$ and B_{target}.

To determine an initial value for B_{target}, EvaRate assumes the application will specify the maximum tolerable round-trip latency , L_{max}. For example, L_{max} for real-time communication applications like Skype and FaceTime would be low (say 100 ms), while that for throughput-intensive applications could be potentially larger. Then, using Equation (4.2), we can set the initial B_{target} to be equal to the buffer occupancy corresponding to L_{max}:

$$B_{target\_initial} = \rho(L_{max} - RTT_{min}) \tag{4.6}$$

However, our goal is not to converge to this upper bound but to minimize the buffer occupancy without emptying the buffer. Maintaining a low buffer occupancy is essential for achieving low latency and preventing packet loss [27], especially in networks with shallow buffers. To this end, we make B_{target} decay with a half-life equivalent to $2 \times RTT$, which is roughly the minimum time required by the sender to detect the impact of changes made to the sending rate. In other words,

$$B_{target}(t) = B_{target\_initial} e^{-\frac{t.ln(2)}{2RTT}} \tag{4.7}$$

$B_{target}(t)$ will be allowed to decay according to (4.7), until we find that $B(t)$ falls below a threshold that puts the buffer at the risk of emptying (see details in §4.4.5). This suggests that $B_{target}(t)$ has become too low and we will reset $B_{target\_initial}$ as follows (refer §4.4.5 for details):

$$B_{target\_initial} \rightarrow min(\rho \frac{RTT_{max} - RTT_{min}}{2}, 2B_{target}(t)) \tag{4.8}$$

In Figure 4.4, we illustrate the time evolution of $B(t)$ and $B_{target}(t)$. As

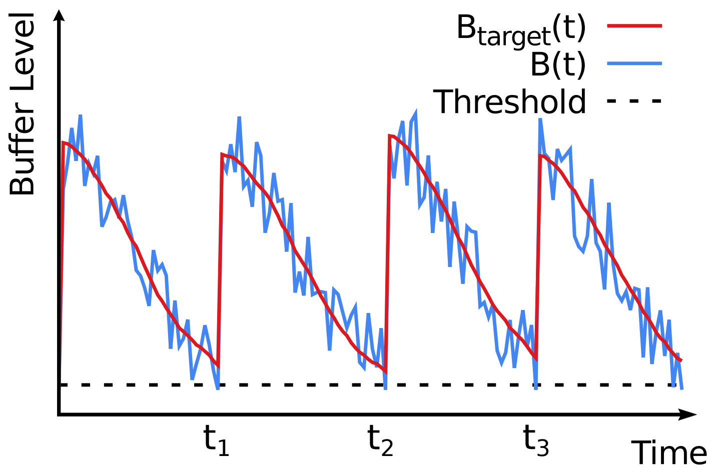


Figure 4.4: Time evolution of $B_{target}(t)$ [schematic].

$B_{target}(t)$ decays, the negative-feedback loop (Equation (4.5)) updates the sending rate so that $B(t)$ tracks $B_{target}(t)$. However, when $B_{target}(t)$ is very low, $B(t)$ will drop below the risk threshold (i.e. at times t_1, t_2 and t_3). At these points, $B_{target}(t)$ is reset according to Equation (4.8).

Note that our basic control loop makes *no assumptions* about the capacity of the bottleneck link. We are able to make a decision on the appropriate sending rate from just the observed receive rate and measured RTT. Therefore, we can fully utilize the available bandwidth by keeping the buffer not empty, and achieve low latency by keeping the buffer occupancy low.

By ensuring that the buffer is not emptied, we fully utilize the available bandwidth; by keeping the buffer occupancy low, we achieve low latency.

4.2 Inferring the Operating Environment

Our key insight to distinguish between the two environments is that, in a *hostile* environment, the buffer occupancy of the competing flows is more

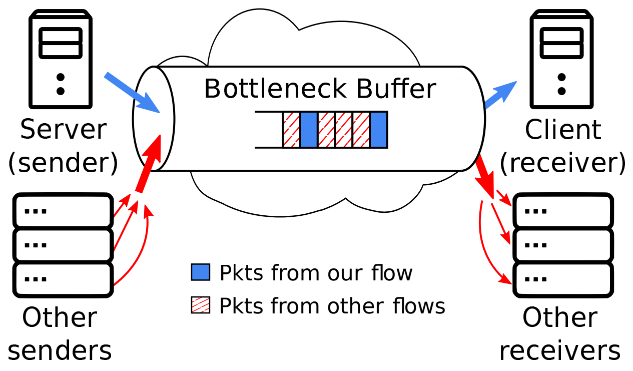


Figure 4.5: Augmented model of network bottleneck.

volatile compared to a *benevolent* environment. To capture the buffer occupancy of the competing flows, we consider an augmented and more realistic model of the network shown in Figure 4.5. In this model, we consider two flows: (i) our flow, and (ii) an aggregate flow consisting of all the competing flows sharing the same bottleneck buffer. We denote the sending rates of the two flows with $S(t)$ and $S'(t)$, respectively. Each flow has packets in the buffer, which we denote with $B(t)$ and $B'(t)$, respectively.

Assume that we are able to estimate (i) the link capacity C, (ii) the number of packets of our flow in the bottleneck buffer i.e. B; and (iii) the overall buffer delay t_{buff}. We know that $t_{buff} = \frac{B+B'}{C}$, so we can estimate B' as follows:

$$B' = t_{buff}C - B \tag{4.9}$$

Since we already have an estimate for B (see §4.1), it remains for us to estimate t_{buff} and C.

Estimating Buffer Delay t_{buff}. There is no way to directly measure the buffer delay t_{buff} without having access to the bottleneck buffer. However, it is possible to achieve a good estimate of t_{buff} indirectly based on the information available at the sender. In particular, for an end-to-end link, the RTT of packets is roughly the sum of the round-trip propagation delay, RTT_{min} and the buffer delay t_{buff}. Therefore,

$$t_{buff} = RTT - RTT_{min} \tag{4.10}$$

Since the occupancy of the bottleneck buffer is highly volatile, we use an exponential-weighted moving-average (EWMA) filter to smooth our estimate of t_{buff}.

Estimating Maximum Available Link Capacity C. In principle, we could measure C if we were able to send large number of packets in a short burst. However, doing this reliably in practice is improbable because the Internet backbones typically have large capacities and they are traversed by a large number of flows. Instead, we opt for an approximation and take C to be the largest instantaneous receive rate ρ during a time window (see §4.4.3).

Hostile vs. Benevolent. Existing TCP variants generally have very little information about competing flows. The main novelty of EvaRate is that it uses the inferred buffer occupancy B' of competing flows to determine whether the operating environment is *hostile* or *benevolent*. In a *benevolent* environment, B' remains relatively stable compared to a *hostile* environment. This is because in the *benevolent* environment, all the flows try to utilize the link while minimizing the buffer occupancy. On the other hand, CUBIC-like flows keep filling the buffer and halving the congestion window when there is packet loss. Both the aggressive filling of the buffer and the halving of the congestion window leads to a noticeable variance in B'. In practice, EvaRate decides the environment is hostile if the variance of the B' estimate over time is large (see §4.4.6). There is a possibility of a false positive, i.e. EvaRate might decide it is facing a hostile environment even when it is not. However, the only consequence of such an occurrence is that EvaRate would have a slightly higher buffer occupancy than necessary.

4.3 Adapting to the Environment

Once an EvaRate flow detects its operating environment, it adjusts its sending rate control loop accordingly.

Hostile Environment. Recall that in a *hostile* environment, the bottleneck buffer is saturated by aggressive CUBIC-like flow(s), causing the bottleneck link buffer to fill and making all flows suffer from long latency or even packet losses. In such situation, it is beneficial to help ease the congestion by remaining less aggressive to add little burden to the bottleneck link. So the priority is to avoid getting starved by CUBIC and remain less aggressive than CUBIC. It turns out that it is sufficient to adjust the sending rate according to Equation (4.5) to prevent EvaRate from getting starved by CUBIC. And doing so, EvaRate will only have a small number of packets in the buffer at any time. In competition with CUBIC flows, the CUBIC flows would have significantly higher buffer occupancy and hence take up a much larger share of the bandwidth. Nevertheless, EvaRate performs very well on the Internet (§5.7) so we chose to not make EvaRate behave more aggressively. It remains as work in progress to come up with a good analytical model for the interactions between CUBIC and EvaRate.

Benevolent Environment. When all competing flows at the bottleneck are attempting to minimize latency, we can estimate B' accurately. This means that EvaRate flows can work together to keep the overall buffer occupancy low. We do this by modifying the control loop in Equation (4.5) to use the estimated total buffer occupancy $B + B'$. More specifically, the sending rate is now controlled by the following equation:

$$S(t) = \rho(1 - k\frac{B(t) + B'(t) - B_{target}}{B_{target}}) \tag{4.11}$$

where, $B(t)$ is the flow's own buffer occupancy and $B'(t)$ is the estimated buffer occupancy of the competing flows. This control loop manages to keep the latency low even if the competing flows are not EvaRate flows, but other low-latency variants like BBR. The impact is that EvaRate tends to be even less aggressive than when it operates in the presence of CUBIC. Unsurprisingly, EvaRate is less aggressive than BBR (see §5.2).

According to Equation (4.7), a set of EvaRate flows exhibit the same behavior with the decay of $B_{target}(t)$. When they detect the overall buffer level, i.e. $B(t) + B'(t)$, falling below a threshold, they reset B_{target} according to Equation (4.8). While this might suggest that synchronization across flows could happen, it doesn't occur in practice because there is a measurement lag of approximately 1 RTT where some flow would have already increased its sending rate and caused the buffer to fill before the other flows react. The convergence rate is related to the rate of B_{target} decaying and resetting but it is a trade-off because if the oscillations are too fast, the system will become unstable. It takes time to measure the effect of a change before the next change can be made.

4.4 Implementation

We implemented EvaRate in the latest Linux kernel v3.19 as a kernel module, with about 2,500 lines of code. In this section, we provide an overview of some of the more important implementation details.

4.4.1 Receive Rate (ρ) Estimation

Unlike previous algorithms [43, 46], we have chosen to perform receive rate estimation at the sender instead of the receiver to avoid modifications to the receiver's TCP stack and to ensure that EvaRate is compatible with existing TCP implementations. To estimate the receive rate, the sender relies on the TCP timestamp option at the receiver which by default is enabled on all major operating systems (Windows, MacOS, Linux, Android, etc.). With the TCP timestamp option enabled, the TCP receiver will send ACK packets with the `TSval` set to its current time. This timestamp corresponds to the receiving time of the data packet at the receiver. From the ACK number and the timestamp value, the sender can determine the number of bytes received by the receiver and the time taken to receive this data. The only caveat to our method is that the sender needs to know TCP timestamp granularity at the receiver. Our current implementation uses 4 ms as the receiver's timestamp granularity since it is currently the default on most OS's. However, we have separately developed a simple technique to detect the receiver's timestamp granularity within one RTT. The details of which are omitted in the interest of space.

After making a change to the sending rate, we need to let the change take effect before making the next change. Hence, we update the estimated receive rate at intervals of one RTT. The sender designates one packet as a "signal packet" and, upon receiving its ACK, calculates the number of acknowledged bytes (B_{curr}) and the duration elapsed (D_{curr}) to estimate the instantaneous receive rate $\hat{\rho} = \frac{B_{curr}}{D_{curr}}$; it then sends out a new "signal packet". $\hat{\rho}$ can be volatile, so we use $\bar{\rho}$ which an exponentially weighted moving average (EWMA) of $\hat{\rho}$:

$$\bar{\rho} = \alpha\bar{\rho} + (1-\alpha)\hat{\rho} \tag{4.12}$$

where, α is set to $\frac{1}{8}$ in our implementation. We use $\hat{\rho}$ for Equation (4.4) and for our C estimate (see §4.4.3) because we want an instantaneous estimate of the buffer occupancy $B(t)$ and the maximum available link capacity respectively. All other equations in our algorithm use $\bar{\rho}$ as the receive rate estimate.

4.4.2 Propagation Delay (RTT_{min}) Estimation

Like for the receive rate, the sender continuously measures the network RTT using the timestamps of the sent packets and corresponding received ACKs. The minimum observed RTT since the beginning of the connection is used as an estimate for the round-trip propagation delay RTT_{min} between the sender and the receiver. If an EvaRate flow shares a bottleneck buffer with other flows *after* the other flows have already started, the estimated RTT_{min}

would tend to be higher than the actual round-trip propagation delay. This has only a minor impact since this results in a lower initial B_{target}, which will eventually oscillate anyway.

4.4.3 Capacity (C) Estimation

To estimate the maximum available link capacity, we use $\hat{\rho}$ instead of $\bar{\rho}$, as we want to estimate the upper bound. However, C may change over time due to various reasons, e.g. shifting of the network bottleneck. So, instead of taking the largest $\hat{\rho}$ observed since the beginning of a flow as the estimate for C, we instead take the largest $\hat{\rho}$ observed within the last $10RTT$ time window. This allows us to adapt to changes in C and potentially eliminate spurious $\hat{\rho}$ estimates (if any).

4.4.4 Regulating the Sending Rate

Recall that in Equation (4.5), the parameter k controls the rate of convergence of the control loop. The rate of convergence is basically the aggressiveness in matching $B(t)$ to B_{target}. In our implementation, we found that $k = \frac{1}{2}$ works well in practice. With this setting, $S(t)$ has a maximum value of $\frac{3}{2}\bar{\rho}$ when $B(t) = 0$, where $\bar{\rho}$ is the estimated receive rate. This limits the sending rate $S(t)$ to $\frac{3}{2}\bar{\rho}$ and prevents the sender from becoming overly aggressive.

4.4.5 Regulating $B_{target\_initial}$

Equation (4.6) initializes $B_{target\_initial}$ using the application-specified target average latency L_{max}. It is possible that L_{max} is less than the network's measured RTT_{min}. In such a case, $B_{target\_initial}$ obtained from Equation (4.6) would be negative. We handle this case by setting $B_{target\_initial}$ to 10 packets, since we also sent out a burst of 10 packets during *slow start*. It is also possible that the bottleneck buffer is shallow and the B_{target} based on the application-specified L_{max} is too large and unachievable. In such case, attempting to fill the buffer to B_{target} can cause a lot of losses due to buffer overflow and hurt other flows. We found that $\frac{RTT_{max}-RTT_{min}}{2}$, which is half of the maximum observed buffer occupancy is a safer and more reasonable basis to reset B_{target}. Therefore, we use the minimum of $2B_{target}$ and $\frac{RTT_{max}-RTT_{min}}{2}$ as the new B_{target}, as shown in Equation (4.8).

It it not possible to implement exponential decay in the kernel precisely since we do not have access to floating point operations. As a result, our implementation approximates Equation (4.7) by interpolation as follows:

$$e^{-\frac{t.ln(2)}{2RTT}} \approx \begin{cases} (-\frac{1}{4RTT}t+1), & t \leq 2RTT \\ (-\frac{3}{16RTT}+\frac{7}{8}), & t \leq 4RTT \\ \frac{1}{8}, & \text{otherwise} \end{cases}$$

Also, recall that B_{target} is reset using Equation (4.8) when $B(t)$ falls below a certain threshold. This is because of allowing $B(t)$ to fall to zero risks link under-utilization. Our current implementation sets this threshold at 10 packets because it works well in practice. In the long term, we intend to study how this threshold should be set as a function of network volatility.

4.4.6 Hostile Environment Detection

From §4.2, we know that the defining characteristic of a hostile environment is a large standard deviation in the estimated B', which is the buffer occupancy of the competing traffic. In our current implementation, we compute the standard deviation of B' in time windows of 100 seconds and decide that we are operating in a hostile environment if the standard deviation exceeds 50 packets.

4.4.7 Handling Losses and Network Outage

Notwithstanding that EvaRate is a delay-based congestion control algorithm, it is still important to handle losses appropriately to avoid inflicting losses on competing flows. First, when EvaRate detects more than 1 packet loss during the *Slow Start* phase, it instantly switches to *Congestion Avoidance* mode. Second, EvaRate limits the maximum number of packets in flight to no more than the sum of the estimated bandwidth-delay product and B_{target}. This helps mitigate the impact of network outages or sudden drop in bandwidth.

In addition, packet losses will cause the ACK number to stop increasing and thus need to be handled. Since SACK is enabled, we simply retransmit dropped packets. We can obtain reasonably good send rate estimates by assuming that each duplicate ACK corresponds to one MSS packet received. Also, when enabled, SACK blocks in the ACKs can be used to accurately determine the exact number of bytes received.

Chapter 5

Performance Evaluation

In this chapter, we evaluate EvaRate using both trace-driven emulations and real network experiments to demonstrate that EvaRate not only works in practice, but that it can also improve the latency performance of the whole network when incrementally deployed. To compare EvaRate to previous algorithms under the same network conditions, most of our experiments were performed with trace-driven emulation. We used a trade-driven emulation tool Cellsim, developed by Winstein et al. [43] and subsequently enhanced by Leong et al. [27]. Apart from emulating the network delay and bandwidth according to a network trace, Cellsim also maintains separate uplink and downlink tail-drop buffers that allows us to verify the ground-truth buffer occupancy.

Our testbed setup is shown in Figure 5.1. Multiple TCP senders running on different physical machines connect via an Ethernet switch and Cellsim to a TCP receiver. The bottleneck link and the corresponding buffer is emulated by Cellsim which runs on a physical machine with two network interfaces. We

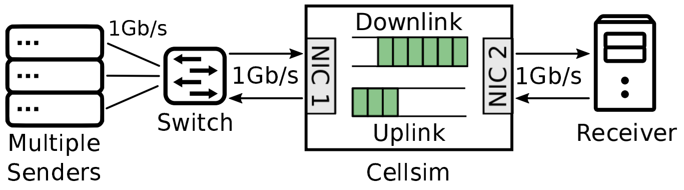


Figure 5.1: Testbed setup using Cellsim [43].

generated some constant bandwidth traces and also used publicly-available cellular traces from Leong et al. [27].

First, we conducted trace-driven emulations to study whether EvaRate is able to manage the bottleneck link buffer as we expect in an ideal environment, where the background link capacity is fixed and only a single EvaRate flow runs in the network. We show that EvaRate works as designed and it is able to probe the bottleneck link buffer occupancy and keep the buffer occupancy low.

Next, we conducted 2-flow experiments to investigate how EvaRate interacts with itself, CUBIC and BBR. Our results suggest that EvaRate is fair to itself, and that 2 EvaRate flows can collaboratively keep the bottleneck link buffer. Also, we show that EvaRate acquires a small share of the throughput when competing with CUBIC, which demonstrates that while it is not as aggressive as CUBIC, we can indeed avoid starvation in a hostile environment. We also show with trace-driven emulation that BBR, Vivace and Copa can trigger a large number of losses, while EvaRate still manages to keep the buffer occupancy low for deep buffers. We then evaluated the mode detection in various scenarios and found that EvaRate's mode detection has lower false positive and false negative rate than Copa's, at the cost of a slightly longer detection time.

Finally, we conducted Internet experiments to compare the performance of EvaRate, to other state-of-the-art TCP variants. We show that as EvaRate is gradually deployed in the Internet, the latency and loss of the whole network reduces. On the contrary, widespread deployment of BBR, Copa and Vivace could potentially make the latency and loss rate worse. Also, we show that EvaRate is able to achieve good performance on the Internet even though it is clearly not as aggressive as CUBIC. This result was counter-intuitive.

5.1 Baseline: Single Flow

We first performed trace-based emulations with constant-rate traces of different link capacities (4 Mbps, 8 Mbps, 16 Mbps, 32 Mbps and 64 Mbps) to verify that our implementation of EvaRate behaves in a manner consistent with our network model and control loop. In these experiments, an EvaRate *sender* initiates a transmission to a *receiver* using `iperf` for a duration of 60 seconds. The results were all within expectations and similar, so we only plot the network metrics for the 16 Mbps trace in Figure 5.2. The buffer size was set to 2,000 packets (but it does not matter since it was never filled) and L_{max} was set to 100 ms.

We see in Figure 5.2 that *slow start* works as expected and quickly ramps up to probe the network capacity. We can see the sending rate significantly overshoots the available capacity. This explains our decision to increase the sending rate by 50% every RTT, as doubling it would have made it even worse. We can see that the buffer estimate $B(t)_{est}$ is very close to the actual buffer occupancy $B(t)_{act}$, and that B_{target} is oscillating, thereby

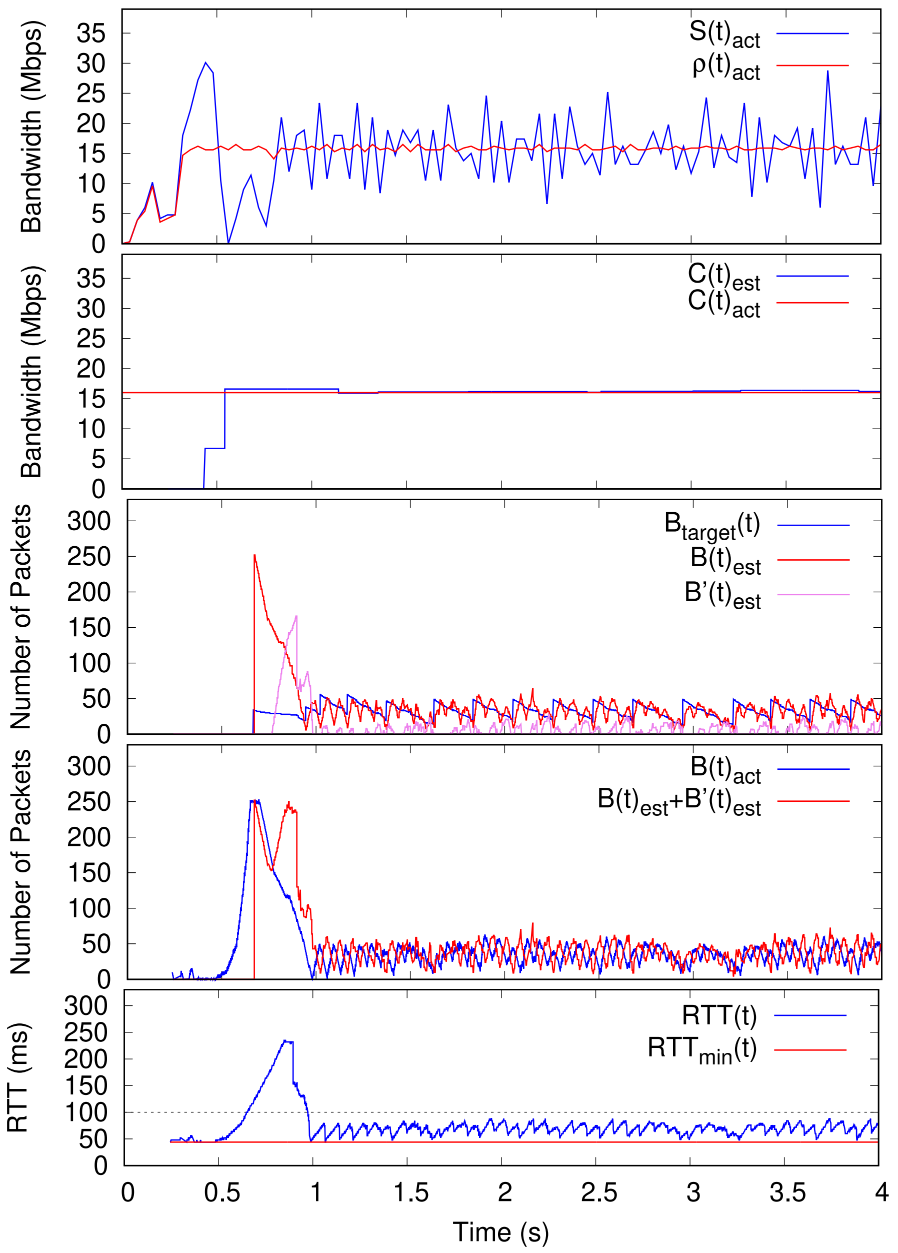
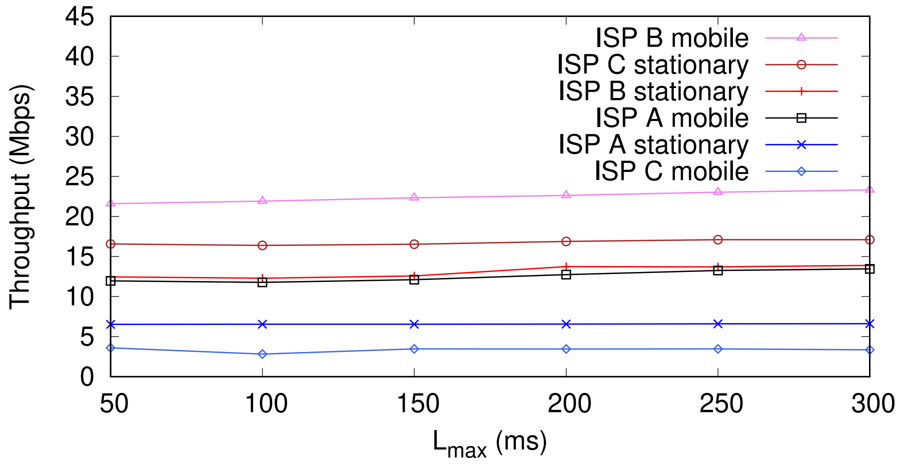


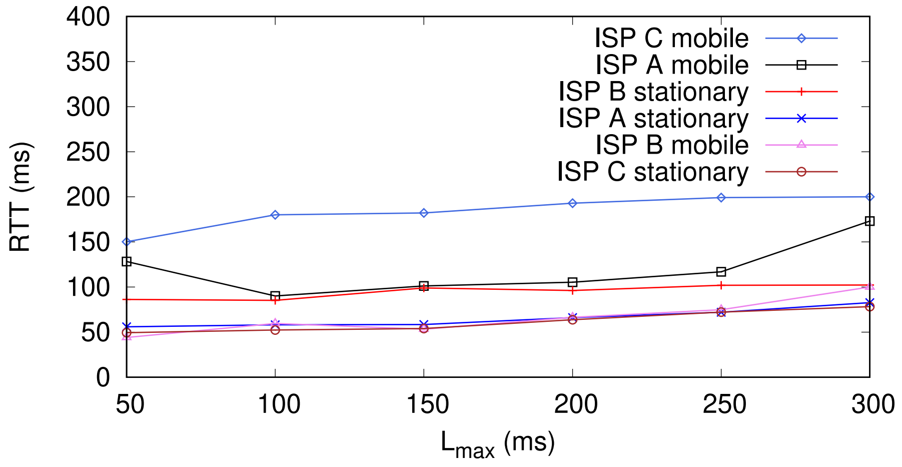
Figure 5.2: Network metrics for constant-rate (16 Mbps) trace.

causing $B(t)_{act}$ to oscillate in step. RTT remains comfortably below L_{max} (100 ms).

Impact of L_{max}. In Figure 5.3, we show how L_{max} affects the average



(a) Throughput.



(b) *RTT*.

Figure 5.3: Impact of L_{max} on throughput and latency.

RTT for different cellular traces. We see that L_{max} has minimal impact on RTT. The resulting RTT tends to be much lower than L_{max} and seems more dependent on the network volatility rather than L_{max}. It is not shown in the figures, but while we might see some slight increase in throughput for a higher L_{max}, the impact of L_{max} on average throughput is marginal at best.

5.2 Playing Well with Others

Next, we investigate how EvaRate interacts with itself, CUBIC, and BBR. CUBIC is currently the dominant TCP variant on the Internet and BBR is the state-of-the-art delay-based algorithm. For each protocol, we run a 60 seconds experiment using a constant-rate trace with a bottleneck link capacity of 2000 KB/sec. The first flow starts at 0 seconds while the second flows starts after 20 seconds.

EvaRate vs EvaRate. In Figure 5.4, we plot network metrics measured for two EvaRate flows sharing the same 2000 KB/s bottleneck link. This scenario illustrates the idealized benevolent environment. Both flows are able to estimate their own buffer occupancy $B(t)_{est}$ and the total buffer occupancy $B(t)_{est} + B'(t)_{est}$ accurately and consistently. The standard deviation of $B'(t)_{est}$ is small from the perspective of both flows and they both operate in benevolent mode. The two flows are able to collaboratively keep the total buffer occupancy low and the RTT below 100 ms.

EvaRate vs CUBIC. In Figure 5.5, we plot the interaction between an EvaRate flow and a CUBIC flow sharing the same bottleneck buffer. When CUBIC reduces its congestion window upon packet loss, EvaRate is able to momentarily measure C and obtain a good estimate of the total buffer occupancy $B(t)_{est}+B'(t)_{est}$. This does not last very long as we refresh C and our estimate of total buffer occupancy starts to diverge, while our estimate of $B'(t)$ starts to drop significantly. The large standard deviation of $B'(t)$ is interpreted as a signal that EvaRate is operating in a *hostile* environment. In general, the interaction between CUBIC and EvaRate seems to exhibit a

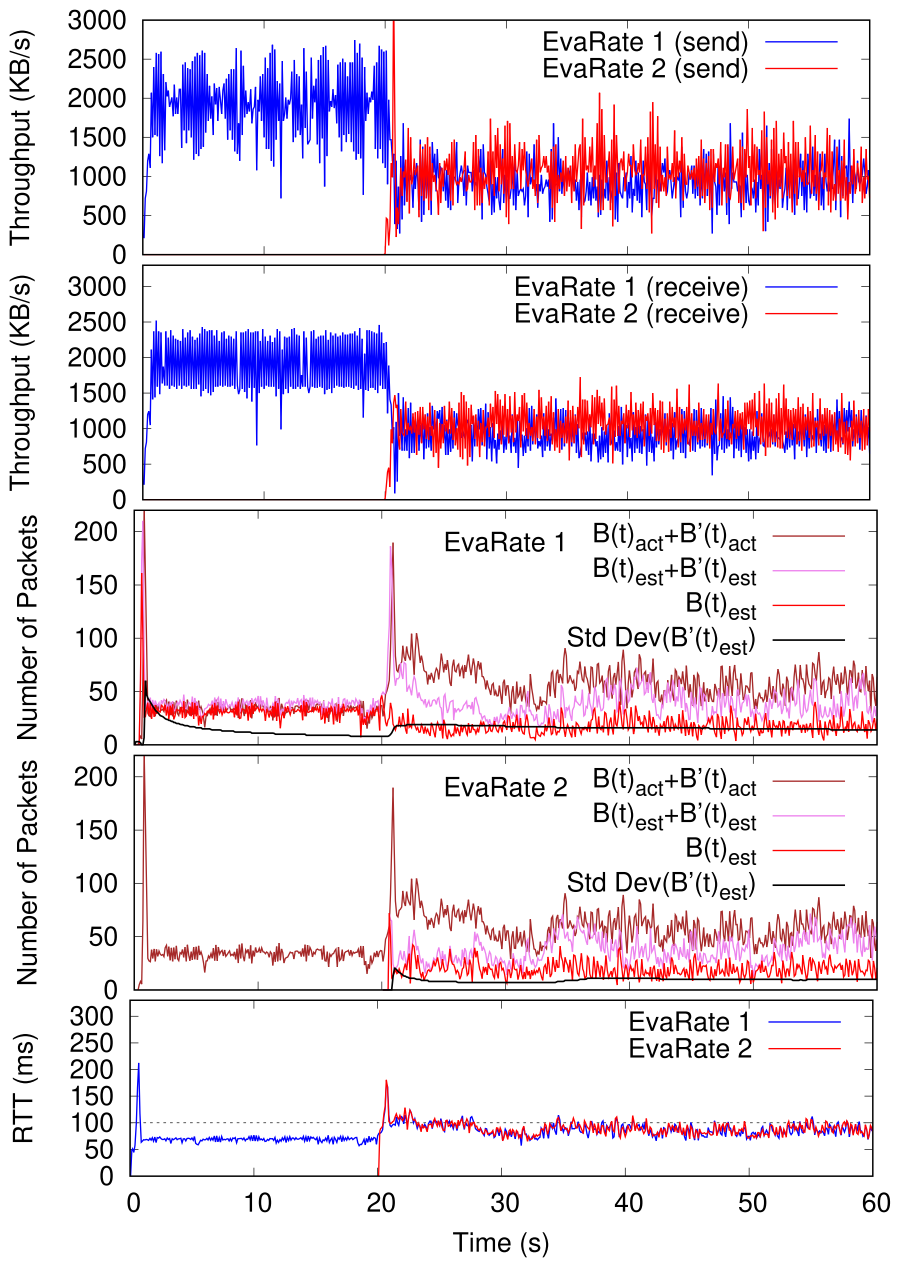


Figure 5.4: Interaction between two EvaRate flows for constant-rate (16 Mbps) trace.

significant amount of randomness. The only constant factor among them is a high standard deviation in $B'(t)$. CUBIC tends to keep the buffer very full and it is difficult for EvaRate to get a fair share of the available bandwidth,

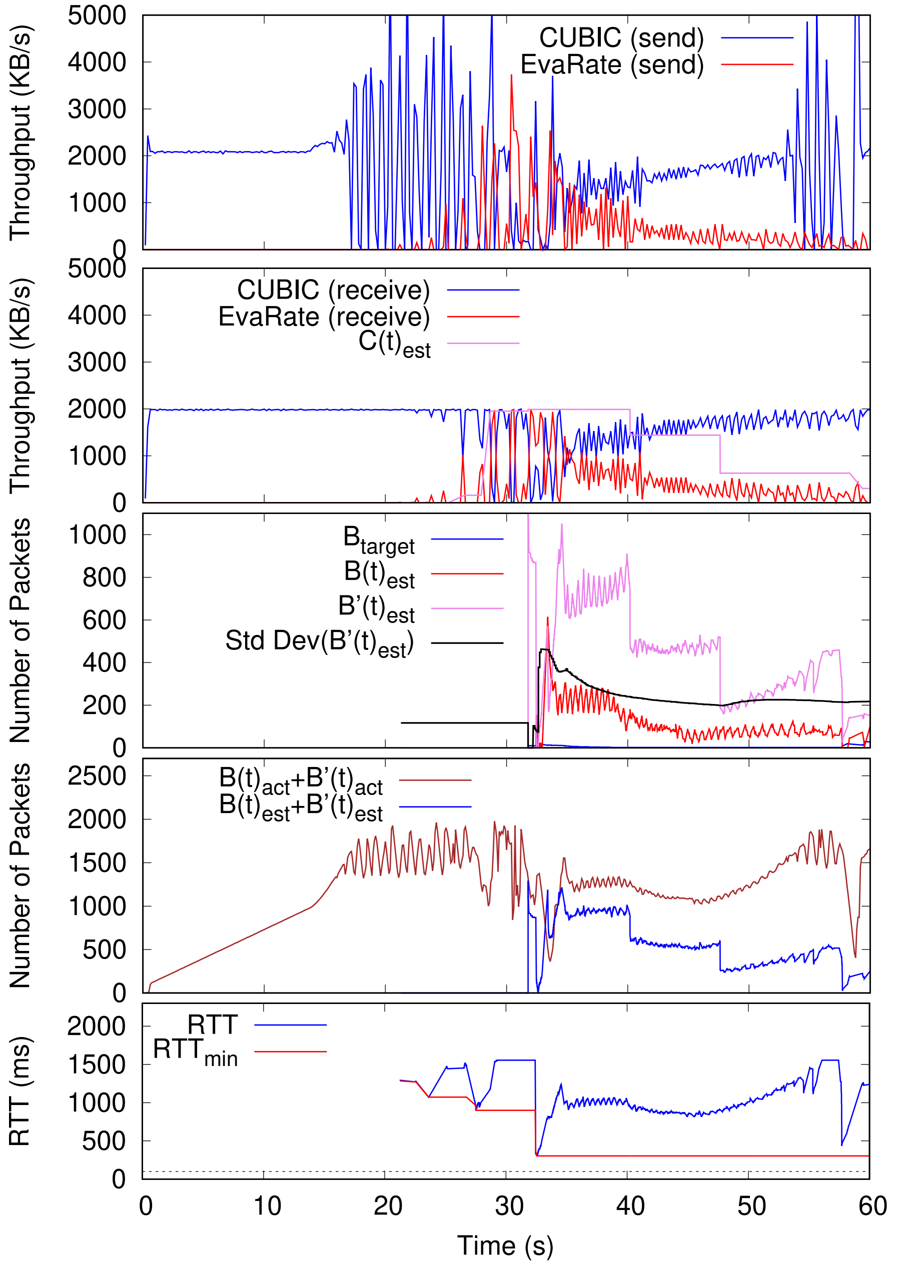


Figure 5.5: Interaction between CUBIC and EvaRate for constant-rate (16 Mbps) trace.

but our control loop does successfully prevent flow starvation.

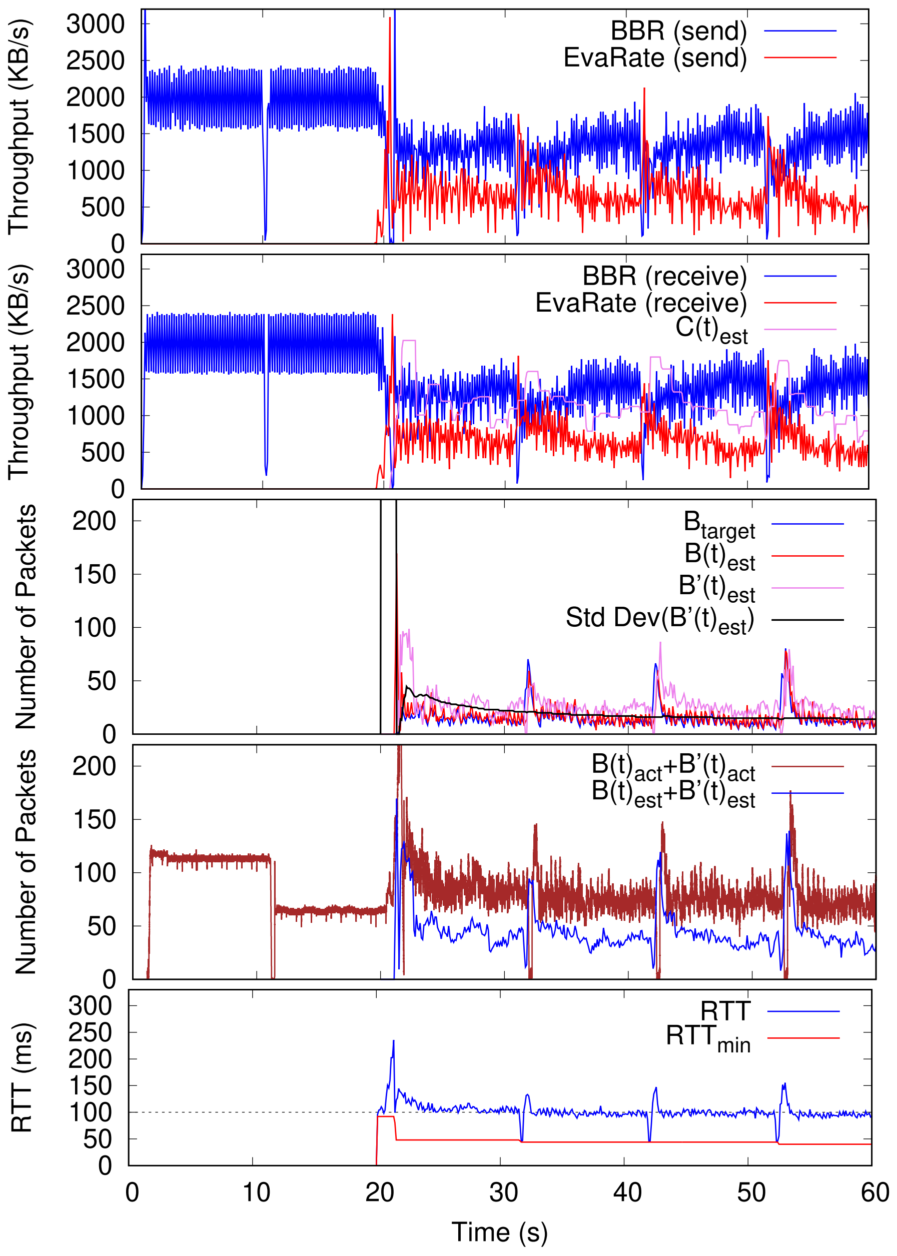


Figure 5.6: Interaction between BBR and EvaRate for constant-rate (16 Mbps) trace.

EvaRate vs BBR. In Figure 5.6, we plot the interaction between an EvaRate flow and a BBR flow sharing the same bottleneck buffer. Unlike CUBIC, BBR periodically (every 10 seconds) stops sending for 400 ms, allowing EvaRate to periodically get a good measurement of the link capacity C, thereby improving the estimate of the overall buffer occupancy. BBR is clearly more aggressive than EvaRate, but we are able to estimate the buffer occupancy for both flows accurately. The standard deviation of B' is small so EvaRate operates in *benevolent* mode. However, while seemingly *benevolent*, BBR's higher aggressiveness results in it having a higher number of packets in the buffer than EvaRate in the steady state, which results in overall higher RTT.

Looking at these results, we can also make some predictions on how EvaRate will react to multiple BBR flows. Unlike EvaRate, BBR does not attempt to coordinate across flows and each BBR flow will maintain its own small but non-negligible number of packets (˜100) in the buffer. If there are many BBR flows sharing the same buffer, there can be a large and constant number of packets in the buffer at any time, so the overall latency will not be low. We verify this in the next section.

5.3 Plausible Explanation for Internet Observations

To understand the observations in the Internet experiments (see §3.2) and answer the question of what the future might look like as the dominant TCP variant changes from CUBIC to new low-latency TCP variants, we ran an experiment where 10 flows share a common 32 Mbps bottleneck link with a shallow 210-packet ($1 \times$ BDP) buffer. The RTT was set at 80 ms. At first,

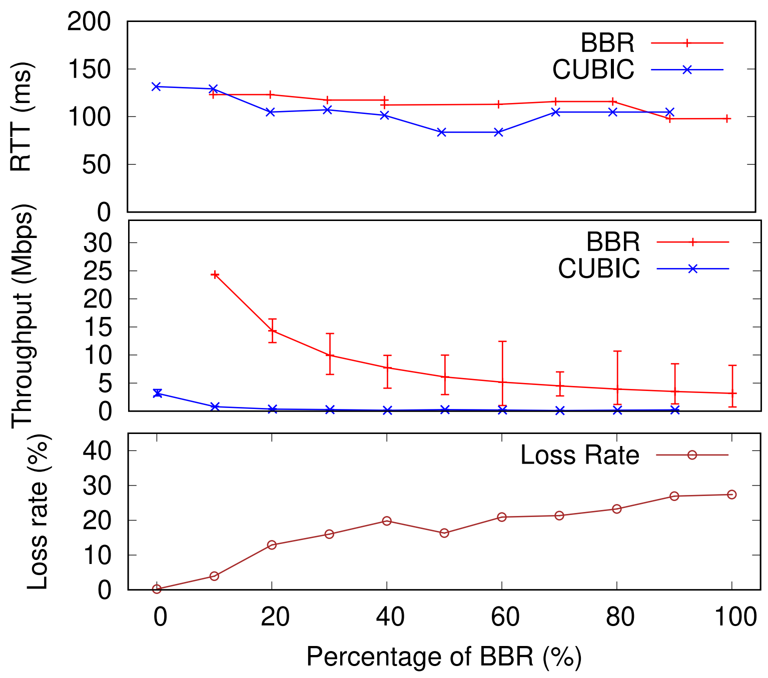


Figure 5.7: Impact of different proportions of BBR for shallow buffers.

all 10 flows were CUBIC and we measured the throughput, RTT and loss rates. Then we repeated the experiment by replacing the CUBIC flows with a low latency flow (one of BBR, Copa, Vivace, or EvaRate) flows one at a time until all 10 flows were the latter variants. We plot the average values for RTT and throughput for the individual flows as well as the loss rate for all the 10 flows in Figures 5.7 to 5.10. The error bars are used to indicate the highest and lowest values for individual flows to provide us a sense of the fairness across the flows.

BBR. We make the following observations about BBR from Figure 5.7: (i) as highlighted in [17], BBR is much more aggressive than CUBIC. Just one BBR flow can take up 25 Mbps of the available bandwidth when competing with 9 CUBIC flows; (ii) as expected, RTT remains high as long as there

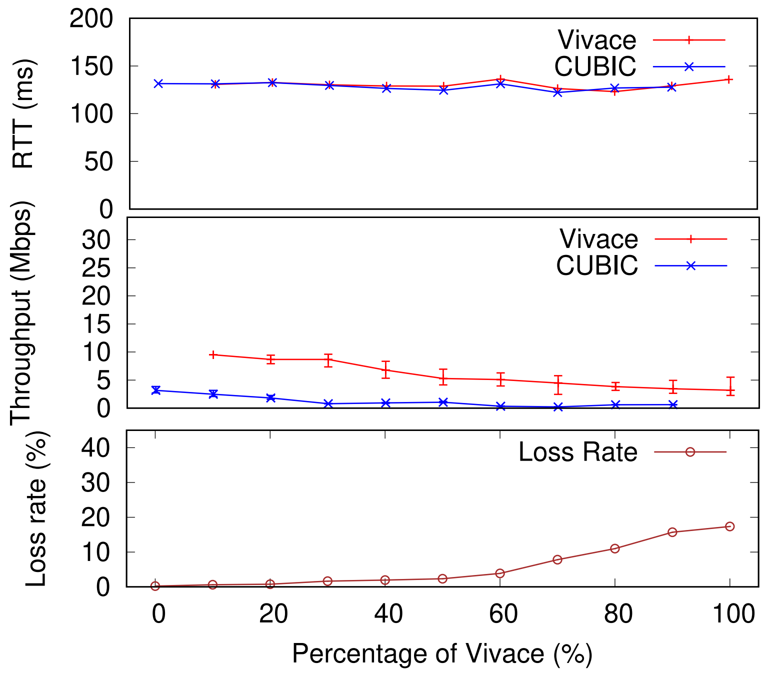


Figure 5.8: Impact of different proportions of Vivace for shallow buffers.

is one CUBIC flow; and (iii) the distribution of throughput is not very fair even when all flows are BBR flows.

Vivace and Copa. We make the following observations from Figures 5.8 and 5.9: (i) both Vivace and Copa take up a larger share of the throughput than CUBIC; (ii) loss rates increase as the number of Vivace and Copa flows increases. At 10 flows, the loss rates for both Vivace and Copa are quite significant at around 10%.

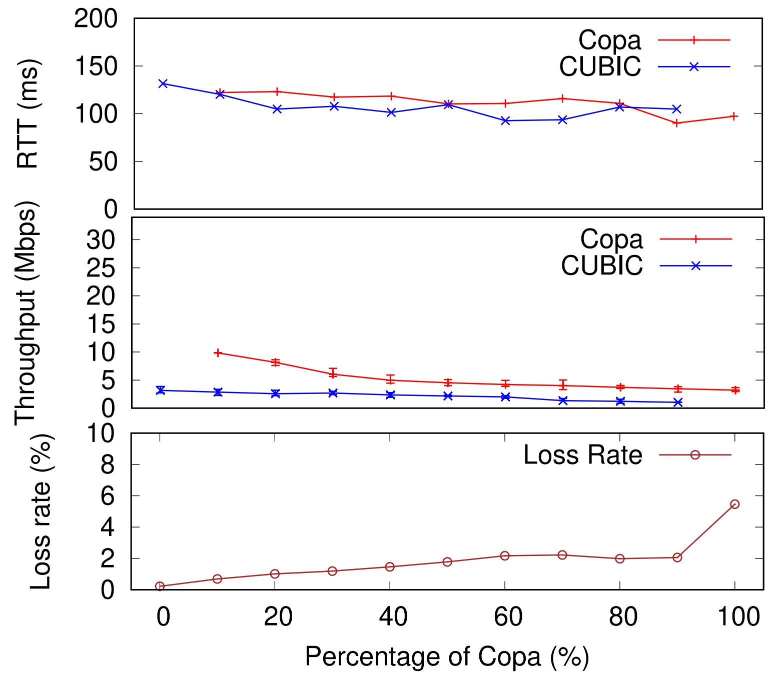


Figure 5.9: Impact of different proportions of Copa for shallow buffers.

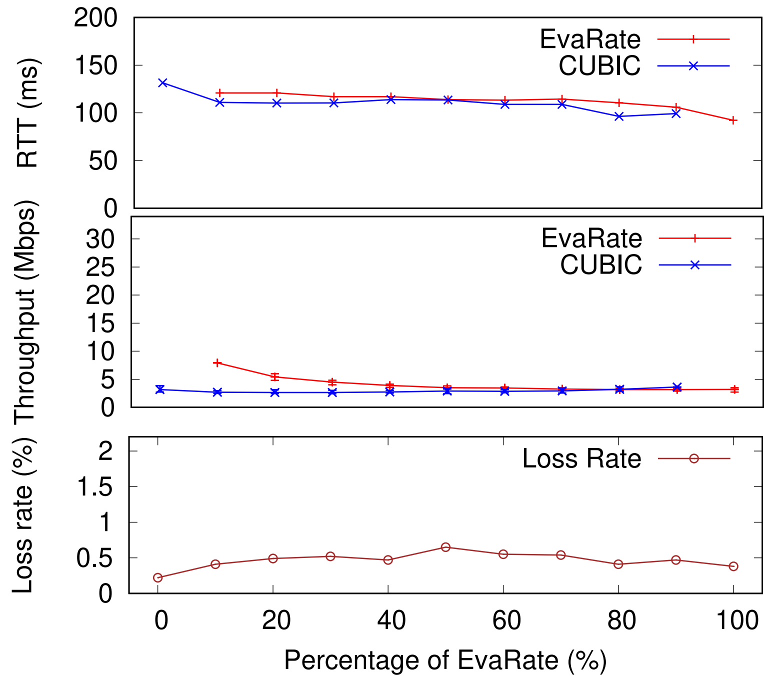


Figure 5.10: Impact of different proportions of EvaRate for shallow buffers.

EvaRate. From Figure 5.10, we see that: (i) EvaRate hardly contributes to the RTT; the resulting RTT is determined by the competing CUBIC flows. (ii) EvaRate achieves a fairer share of the bandwidth when contending with CUBIC, unlike the other variants that tend to throttle CUBIC with increased losses. (iii) EvaRate seems to increase loss rates slightly when there are more EvaRate flows, but the increase is not significant even with the majority being EvaRate flows. When the system is 100% EvaRate, loss rate is actually lower than that with some CUBIC flows.

5.3.1 Codel-Enabled Buffer

Codel is a queuing policy in switches to maintain the queuing delay under a certain level [33]. Since Codel is commonly deployed on the Internet, we also investigated the impact of BBR, Vivace, Copa and EvaRate in Codel-enabled switches. We set the queuing delay threshold to be 70 ms, which corresponds to the buffer size in the previous shallow-buffer experiments when the link capacity and RTT_{min} are 32 Mbps and 80 ms. In Figure 5.14, we can see that Codel has similar effect on EvaRate flows. However, in Figures 5.11, 5.12 and 5.13, we observe larger $RTTs$ for BBR, Vivace and Copa flows compared to the result in the shallow-buffer experiments. The reason is that Codel is essentially a drop-head queue instead of a drop-tail queue, which means that the packets at the tail will not be instantly dropped when the buffer delay exceeds the threshold. As a result, more aggressive flows will flood the bottleneck buffer and experience higher latencies. The study reveals that Codel is mostly equivalent to using a shallow buffer, except

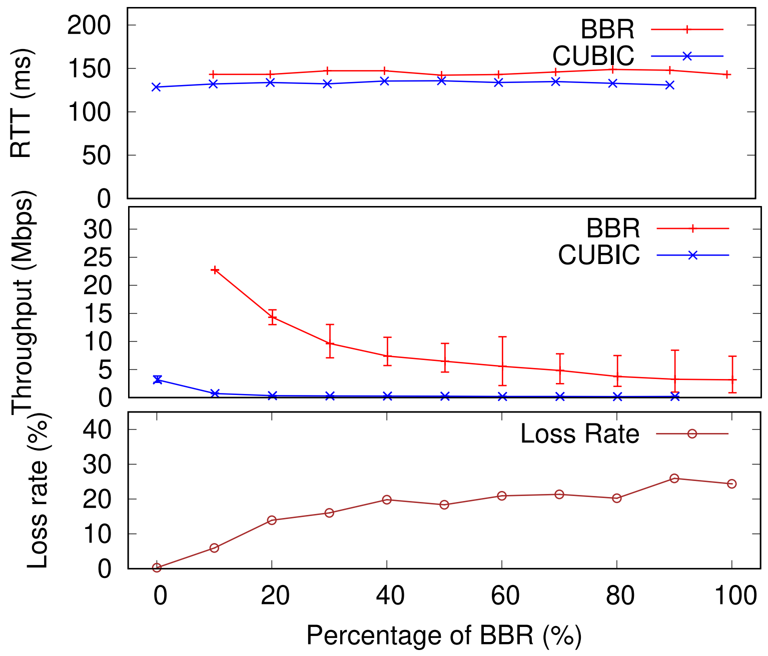


Figure 5.11: Impact of different proportions of BBR for Codel-enabled buffers.

it is a drop-head queue.

Our results cannot fully replicate what we observed with AWS servers on the Internet because we assume all steady state flows and we do not know the parameters of the core Internet routers. They do however suggest that there is a need for more in-depth study on how the newly proposed low-latency flows will impact the Internet as the proportion of these flows increases relative to that of CUBIC.

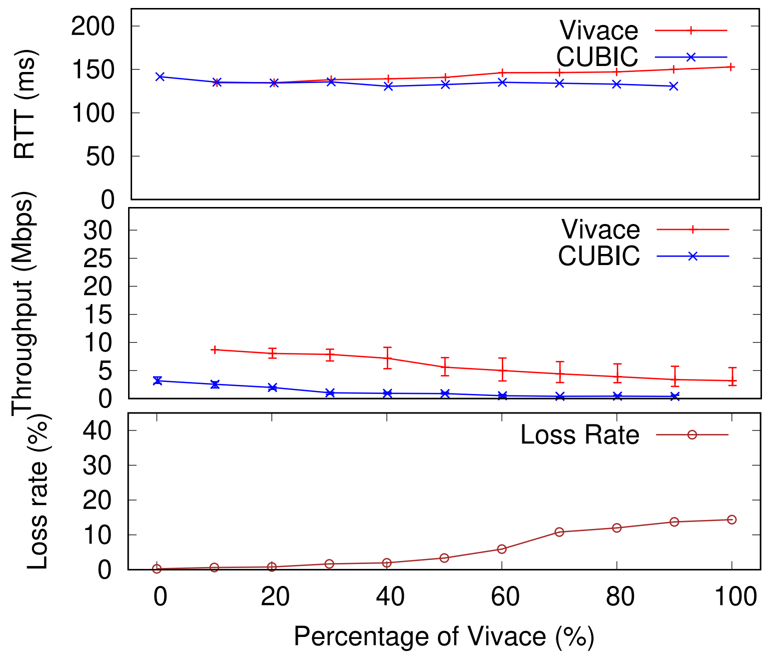


Figure 5.12: Impact of different proportions of Vivace for Codel-enabled buffers.

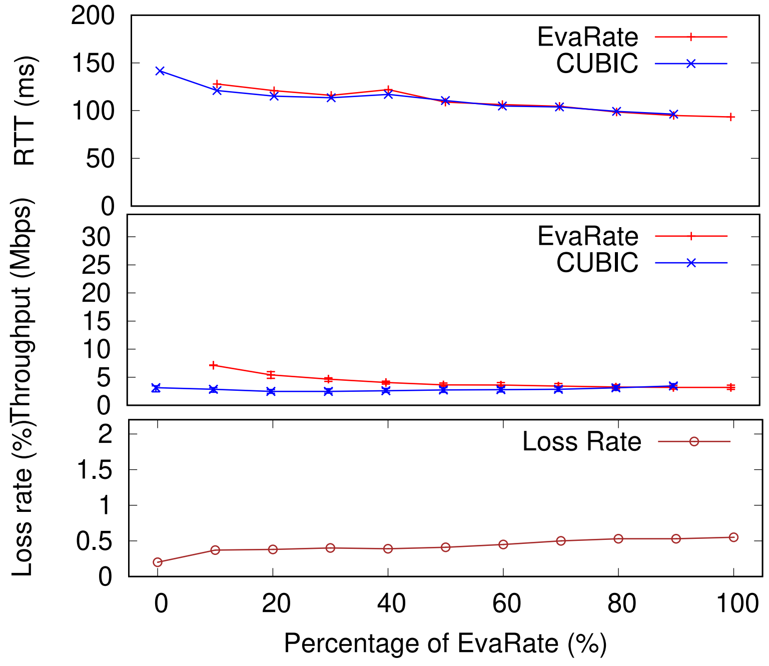


Figure 5.14: Impact of different proportions of EvaRate for Codel-enabled buffers.

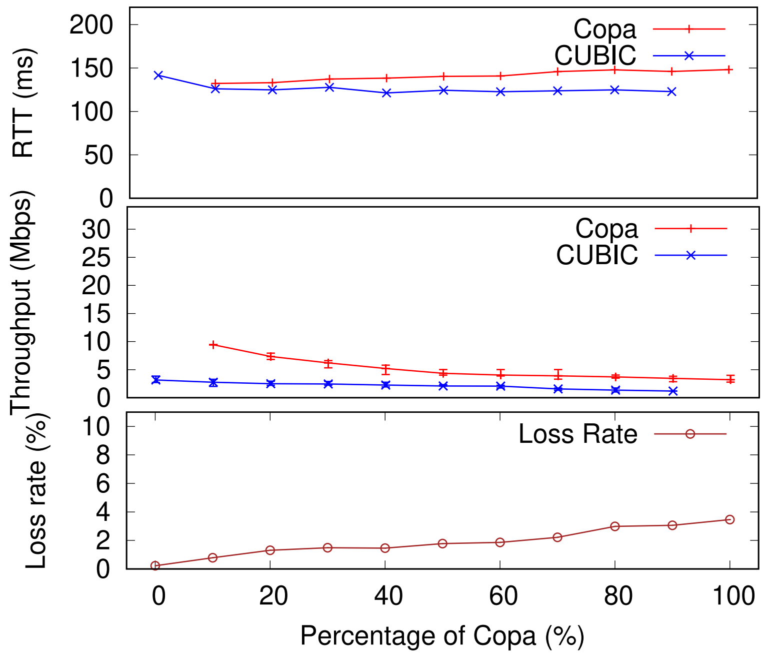


Figure 5.13: Impact of different proportions of Copa for Codel-enabled buffers.

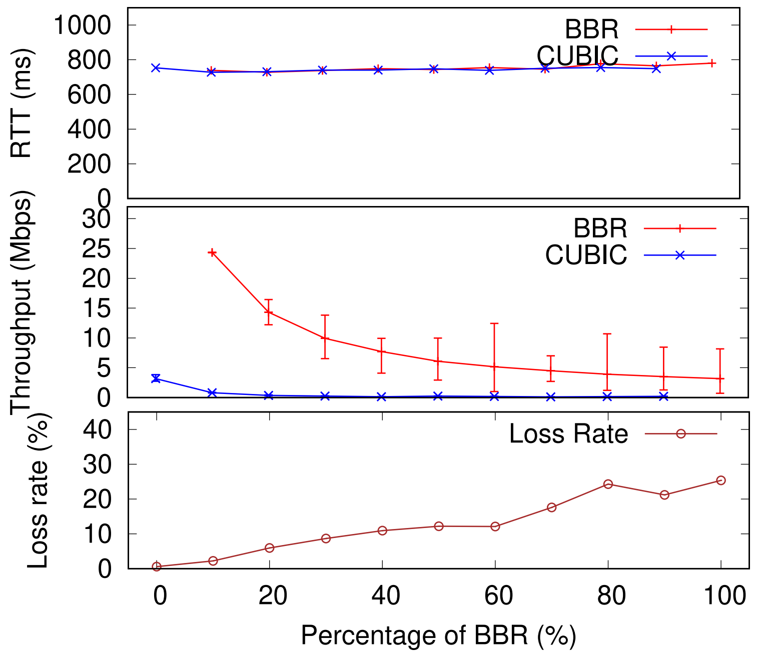


Figure 5.15: Impact of different proportions of BBR for deep buffers.

5.4 Deep Buffers

The results in §5.3 suggest that a plausible reason why Vivace and Copa seem to compete well with CUBIC on the Internet is that bottleneck link buffers are relatively shallow, Vivace and Copa are not sensitive to packet losses and thus can take the share of the CUBIC flows that back off due to packet losses. We decided that it was important to also investigate and understand the impact of deep buffers. To do so, we repeated the experiments in §5.3, but with a relatively deep buffer of 2,000 packets. The results are shown in Figures 5.15 to 5.18.

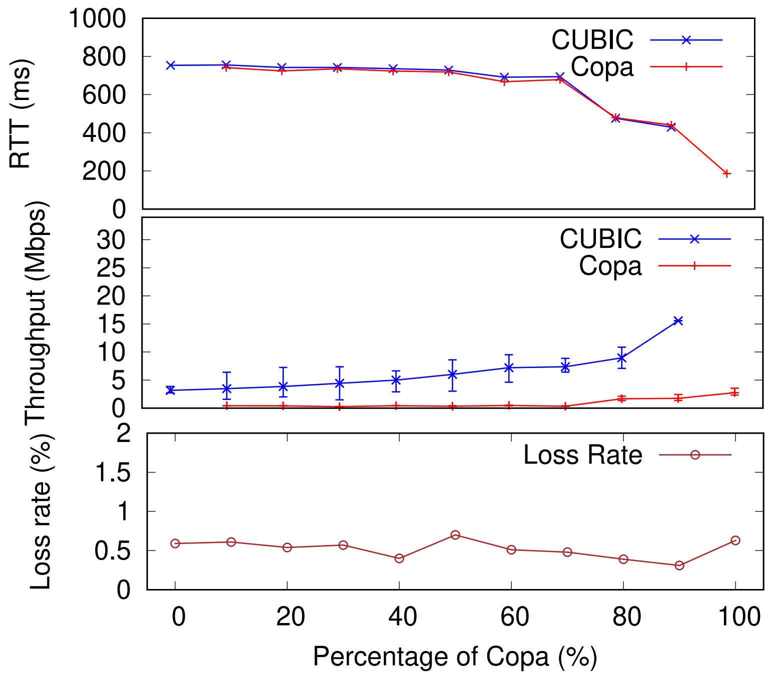


Figure 5.17: Impact of different proportions of Copa for deep buffers.

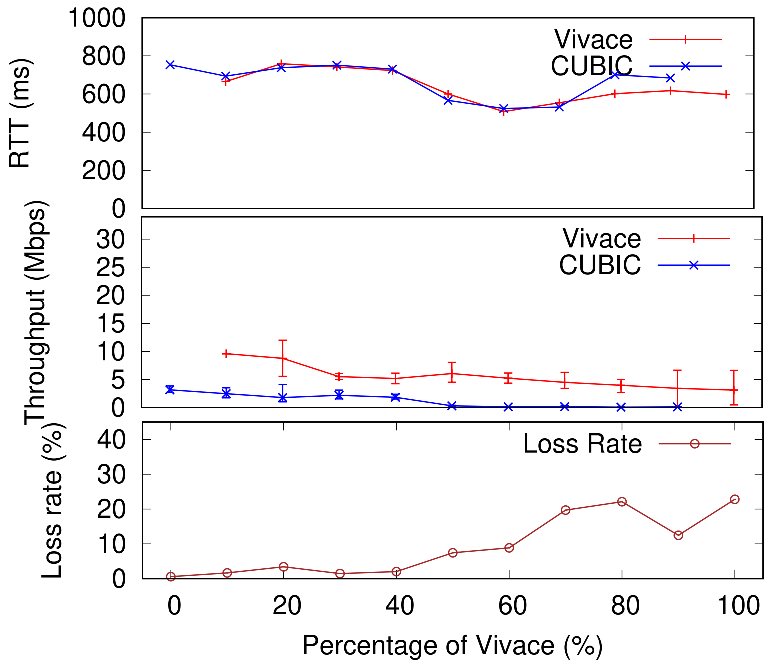


Figure 5.16: Impact of different proportions of Vivace for deep buffers.

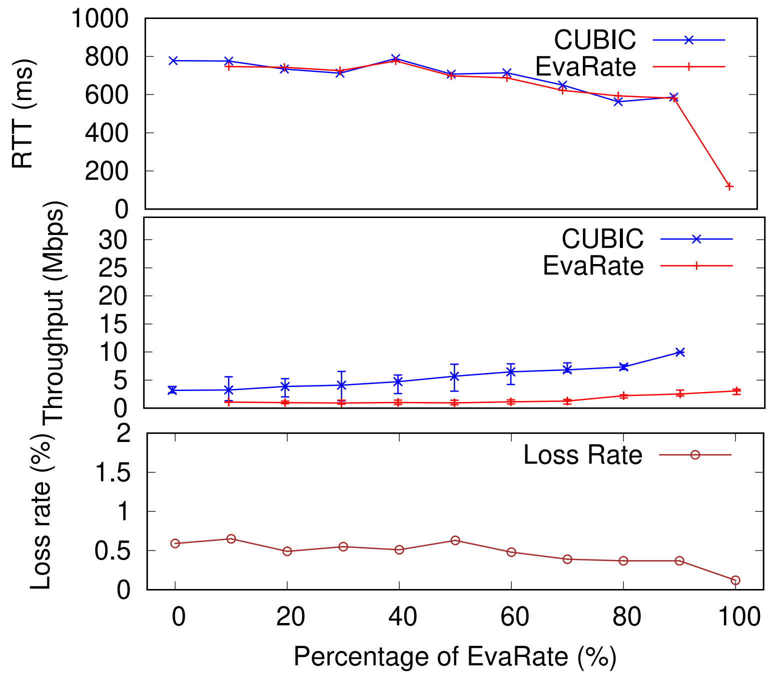


Figure 5.18: Impact of different proportions of EvaRate for deep buffers.

Figures 5.15 and Figure 5.16 show that BBR and Vivace are still able to contend well with CUBIC but as expected, the RTT is capped at 800 ms because of the buffer-filling behavior of CUBIC. The loss rates for both BBR and Vivace are significant as the proportion of these flows increases relative to CUBIC. This suggests that the evaluations of new low-latency flows need to take into account the "stacking" of multiple flows more carefully. In contrast, we see from Figure 5.17 that Copa and EvaRate are relatively well-behaved and the RTT falls as the proportion of these flows increases relative to CUBIC. The loss rates are also low with loss falling to zero when all the flows are EvaRate flows. Unfortunately, both Copa and EvaRate now contend poorly against CUBIC. It is plausible that Copa could potentially be tuned to contend better with CUBIC, but unlike what was claimed in [2], Copa in its default configuration does not perform well under all scenarios.

5.4.1 Tuning Aggressiveness

Next, we demonstrate the usefulness of decoupling the congestion control algorithm into two different modes of operation depending on whether it is operating in the presence of buffer-filling flows. Note that there is an aggression factor α in Equation (4.5). α can be used to make EvaRate more aggressive in the presence of buffer-filling flows, or in a hostile environment. Specifically, we set $\alpha = 1.5$ when EvaRate is in a hostile environment to increase its aggressiveness. We call this variant EvaRate++. Note that we believe it is beneficial to the whole network to stay less aggressive in a hostile environment. Thus EvaRate++ is only to demonstrate that the design of EvaRate allows it to compete well in a hostile environment by adjusting α.

In Figure 5.19, we see that EvaRate++ is able to achieve a fair share of the bandwidth when contending with CUBIC in the presence of deep buffers. In other words, if we can detect the operating environment accurately, we can tune EvaRate to match *any* desired level of aggressiveness just by adjusting α. This, however, comes at a cost. We see in Figure 5.20 that EvaRate++ has effectively replicated BBR's behavior and caused loss rates to increase to about 8% for a network with shallow buffers. We are not in favor of inflicting loss on the underlying network, so α is set at 1 for the default EvaRate implementation. This also suggests that EvaRate could be enhanced to dynamically set α to an appropriate value greater than 1 when operating in a hostile environment with deep buffers. This remains as future work.

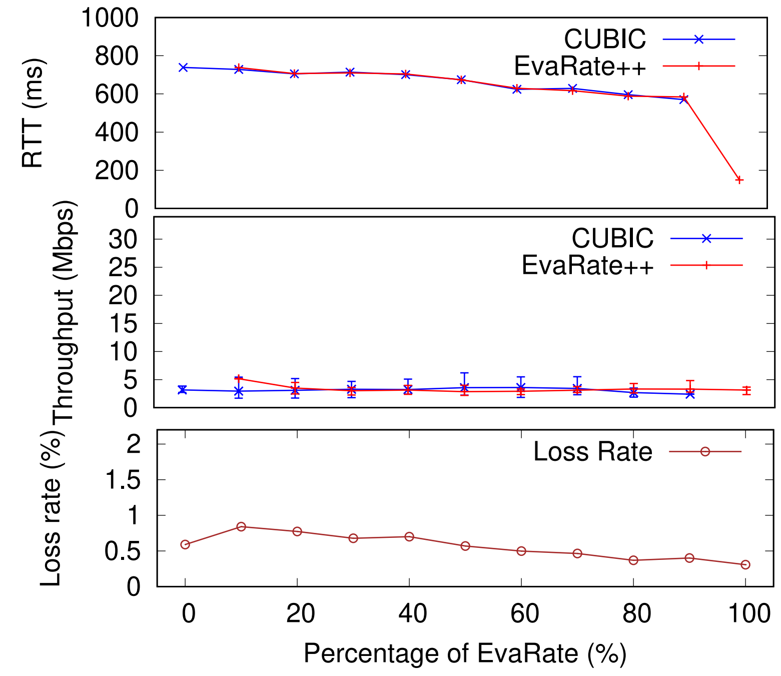


Figure 5.19: Impact of different proportions of EvaRate++ ($\alpha = 1.5$) for deep buffers.

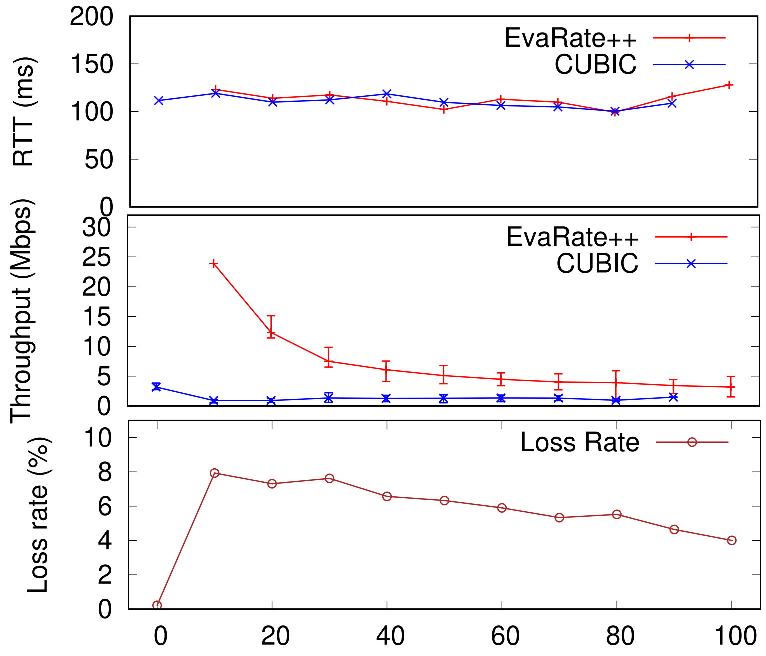


Figure 5.20: Impact of different proportions of EvaRate++ ($\alpha = 1.5$) for shallow buffers.

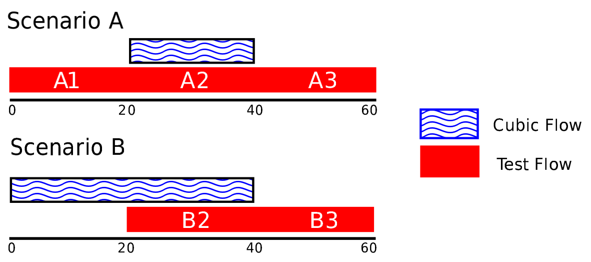


Figure 5.21: Two scenarios of mode detection for Copa and EvaRate.

5.5 Mode Detection Accuracy

To decouple the congestion control algorithm into two operating modes, we need to be able to detect the presence of buffer fillers accurately. The natural question is therefore: how accurate is EvaRate's simple standard-deviation-based mode detection mechanism (§4.4.6)? It turns out that Copa also attempts to detect the presence of buffer fillers by inferring whether a queue is empty at least once every 5 RTT [2].

To investigate the accuracy of these mode detection mechanisms, we considered the 2 scenarios illustrated in Figure 5.21. In Scenario A, we start a test flow and 20 s later, we start a CUBIC flow that persists for 20 s. In Scenario B, we start a CUBIC flow and then the test flow 20 s later. The first CUBIC flow then ends after another 20 s. The goal of these scenarios is to cause the system to change from buffer filling to non-buffer filling and vice versa, and we measured the time taken for the 2 algorithms to detect the changes.

In Figure 5.22, we plot the detection delays for the different scenarios. Our results show that in general, Copa has lower detection delays than EvaRate. Copa concludes that there are no buffer-filling flows (Scenarios A1, A3 and B3) in less than 0.5 s. It is also quite fast in detecting a new buffer filling flow

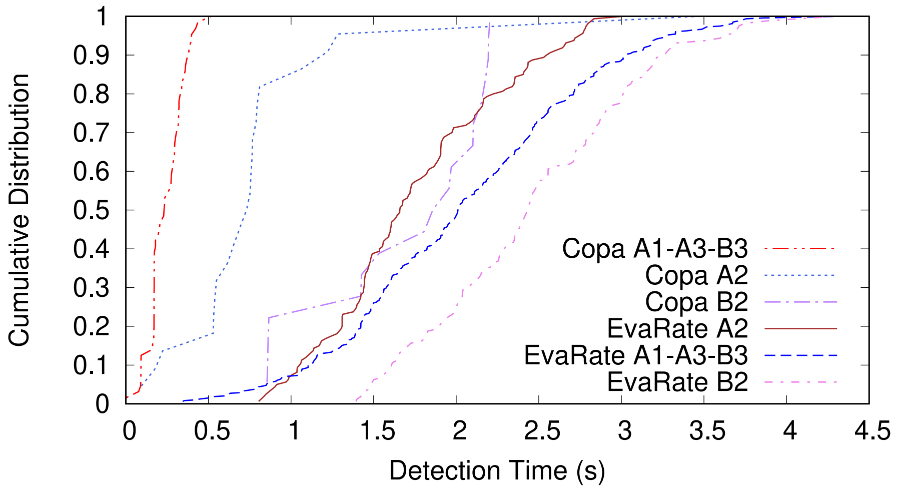


Figure 5.22: Cumulative distribution of mode detection delay.

(Scenario A2). What we found, however, was that Copa took significantly longer (around 2 s) to detect the presence of an existing buffer-filling flow (Scenario B2). EvaRate has a comparable delay to Copa in this scenario (Scenario B2).

The delays in Figure 5.22 do not however reflect the full picture. We found that both Copa and EvaRate were constantly evaluating the operating environment and even after detecting a change correctly, both algorithms will occasionally switch back to the wrong mode (even if there was no actual change in the background). In Figures 5.23 and 5.24, we plot the false positive and false negative rates respectively. A false positive means that the algorithm decides that it is operating in the presence of buffer fillers when it is not, and a false negative means that an algorithm mistakenly believes that it is operating in a benevolent environment even though there is CUBIC-like traffic in the background. It is clear from Figures 5.23 and 5.24 that Copa has traded off detection accuracy for lower detection delays, while EvaRate's

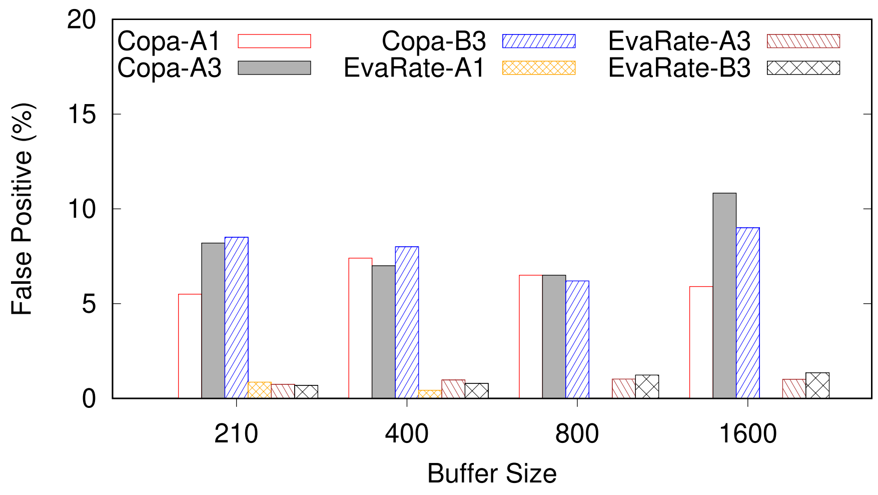


Figure 5.23: False positive rates for Copa and EvaRate.

detection accuracy is very much higher in spite of the longer detection delay.

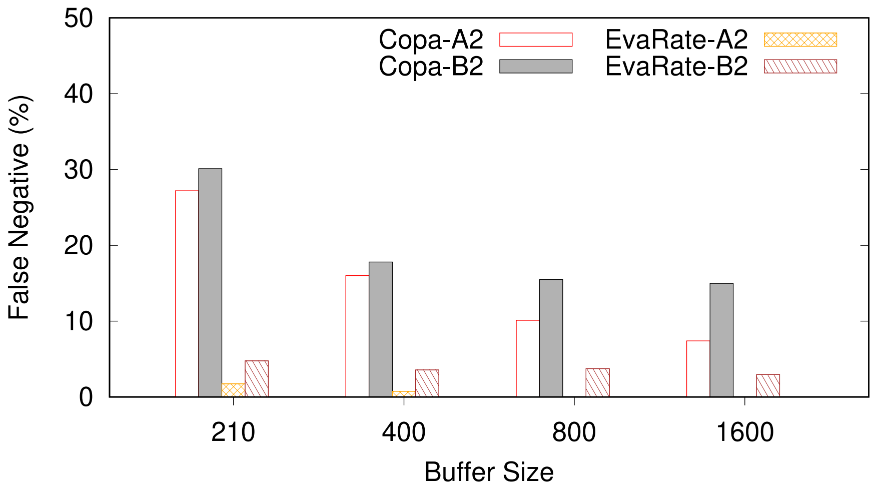


Figure 5.24: False negative rates for Copa and EvaRate.

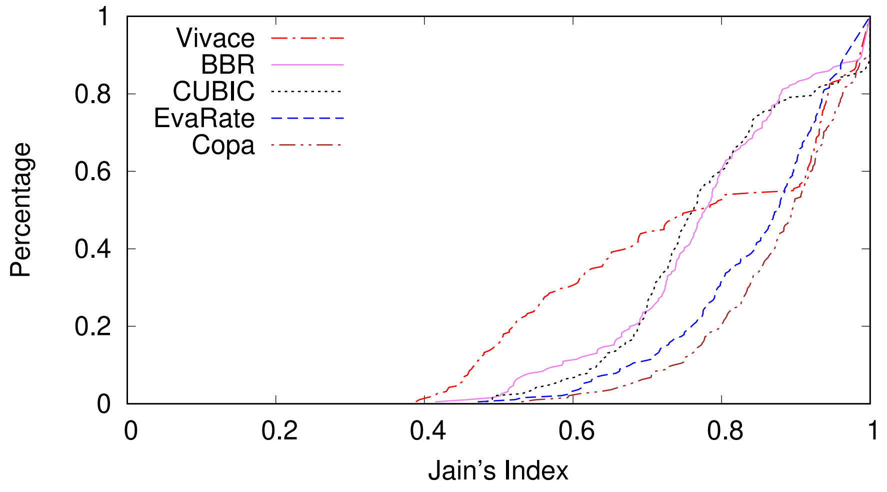


Figure 5.25: CDF of Jain's index for various TCP variants.

5.6 Fairness to Own Kind

We replicated the experiment in [2] to understand how EvaRate compares with CUBIC, BBR, Copa and Vivace in terms of being fair to other flows of the same kind. In this experiment, the bottleneck bandwidth was set to 100 Mbps, the buffer size to 333 packets ($1 \times$ BDP), the RTT was set to 40 ms. For each run, 10 flows are started one at a time at an interval of 1 s, and each flow runs for 10 s. We calculated the throughput of each flow at a window granularity of 100 ms and plot the resulting cumulative distribution of Jain's fairness for all algorithms in Figure 5.25. We can see that EvaRate achieves a similar level of fairness to other flows of the same kind as Copa, and is fairer than the others. However, we note that we had earlier shown in §5.3 that some algorithms can potentially throttle CUBIC under certain scenarios, so the fairness of an algorithm to flows of its own kind is only one facet of fairness.

5.7 Realistic Operating Scenarios

To understand how EvaRate would perform in more realistic operating scenarios, we evaluated EvaRate by repeating the AWS-based experiments described in §3. We also compared the performance of EvaRate to other algorithms for mobile cellular networks and satellite networks using Cellsim.

AWS Experiments. We plot the results for the single-flow experiments in Figures 5.26 and 5.27. We found that the throughput for EvaRate is generally comparable to Copa, but is generally lower than Vivace. However, the loss rate result shown in Figure 5.27 suggests that EvaRate achieves the throughput performance at a much smaller cost of loss rate. It turns out that the loss rate for EvaRate is similar to CUBIC for most source-destination pairs.

We also performed another set of double-flow experiments to validate the results we obtained in the previous experiments and to further understand the sharing of bandwidth at the bottleneck links. In this new set of experiments, instead of initiating a single flow to the AWS remote servers in various continents, we started two flows to the remote servers: one CUBIC flow followed by another reference flow (for the protocol we want to test) after a delay of 20 s. After reference flow started, both flows ran for 40 s concurrently, and stopped at the same time. We calculated the throughput and loss rate for the CUBIC flow and the reference flow for the latter 40 s and plot the results in Figure 5.28 and 5.29. We observed that like the single flow experiment, the low-latency TCP variants were clearly more aggressive than the background CUBIC flow, even if they started later. The loss rate of

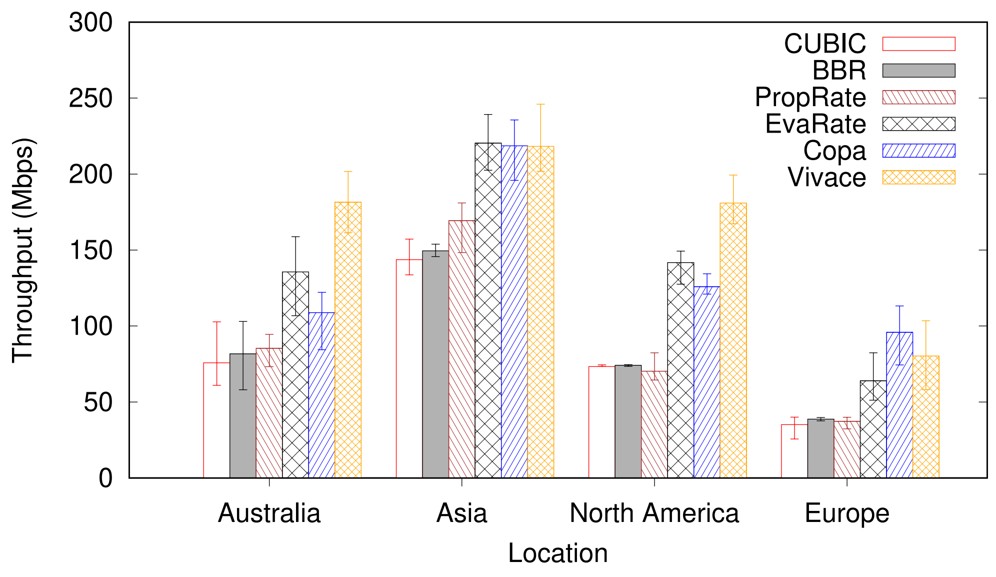


Figure 5.26: Comparison of throughput for single-flow AWS experiment.

low-latency TCP variants was slightly lower than that of the single-flow experiment, but still relatively significant. On the other hand, EvaRate behaves more TCP-friendly than the recent low-latency TCP variants and achieves better fairness and global throughput performance, which comes at a lower loss rate as shown in Figure 5.29.

We found that the default TCP receive window was relatively too small and we thought that maybe this might have prevented *cwnd*-based TCP variants like TCP CUBIC from sending enough data to obtain a larger share of the bottleneck link buffer. We thus repeated the 2-flow AWS experiments by increasing the receive window from the default 200 KB to a much larger value of 40 MB, and plot the results in Figure 5.30. In general, CUBIC generally obtains a larger share of the available bandwidth, though at a higher loss rate. We can see that a larger TCP receive window has more impact for flows with long RTTs and has minimal impact on those with short RTTs. It

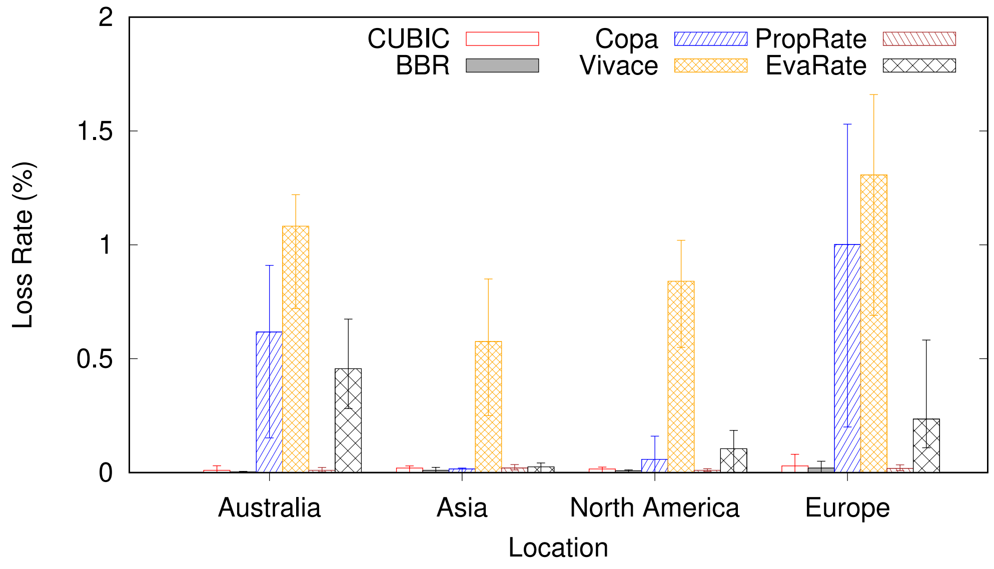


Figure 5.27: Comparison of loss rate for single-flow AWS experiment.

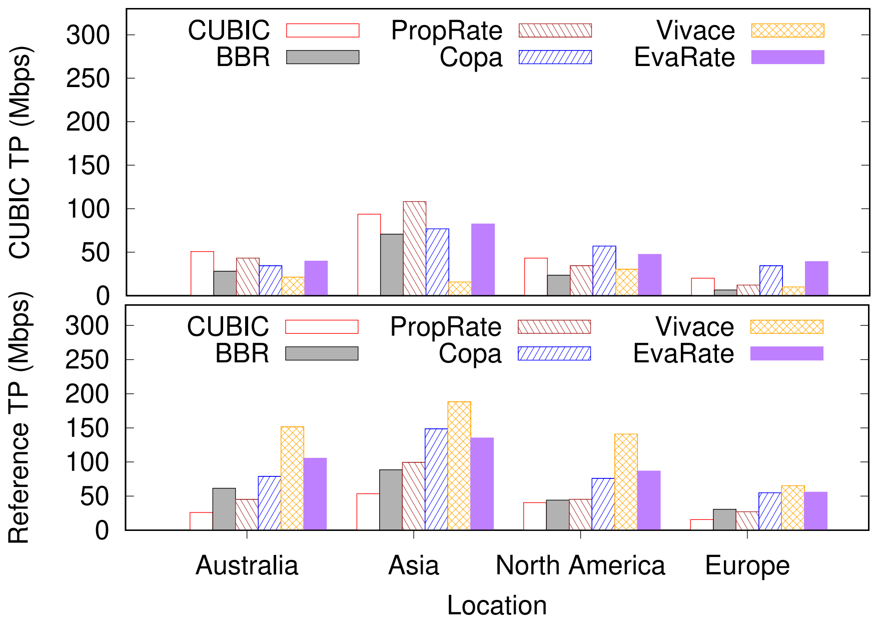


Figure 5.28: TCP throughput of background CUBIC flows and reference flows for double-flow AWS experiment.

was interesting for us to note that the current default CUBIC receive window effectively helps low-latency variant compete better against CUBIC.

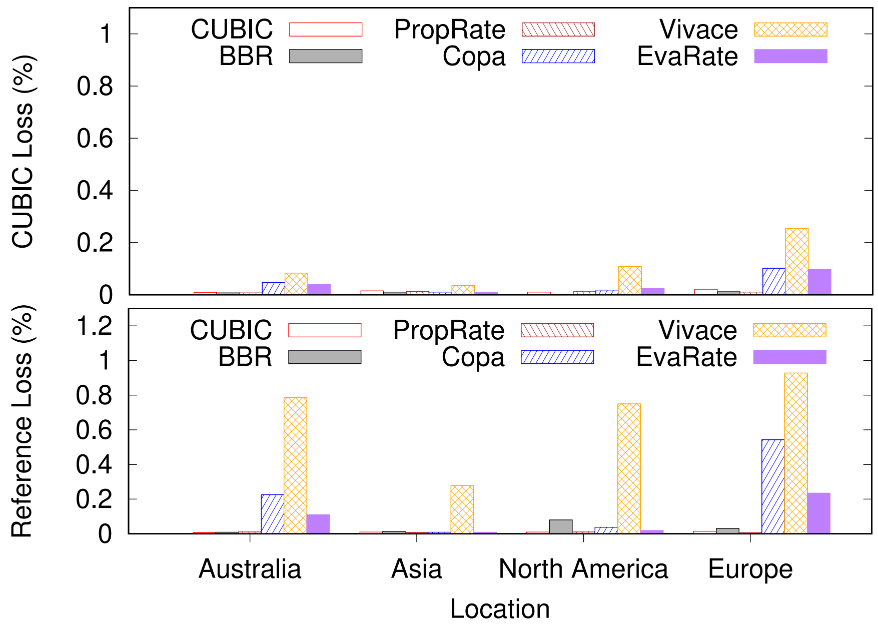


Figure 5.29: TCP loss rate of background CUBIC flows and reference flows for double-flow AWS experiment.

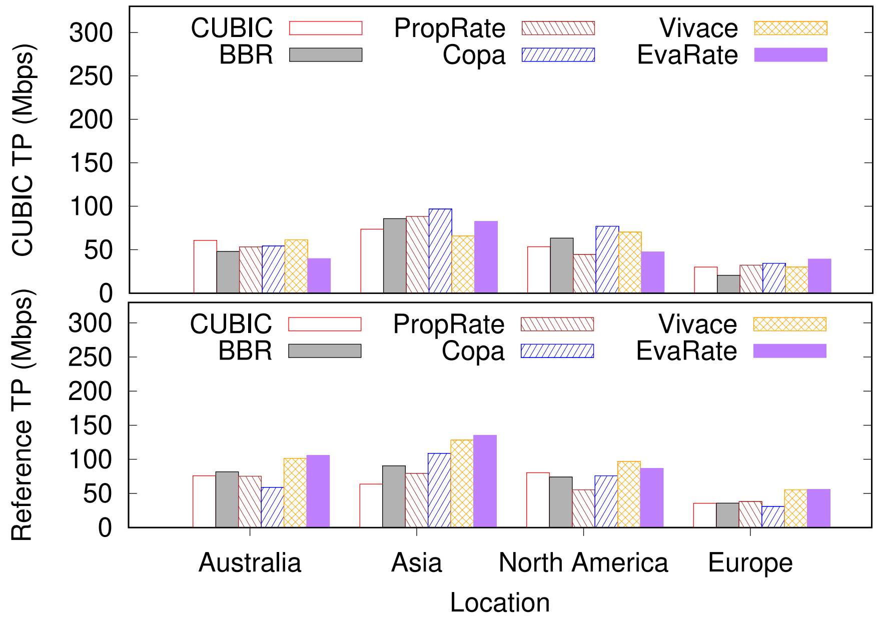


Figure 5.30: TCP throughput of background CUBIC flows and reference flows with a larger receive window.

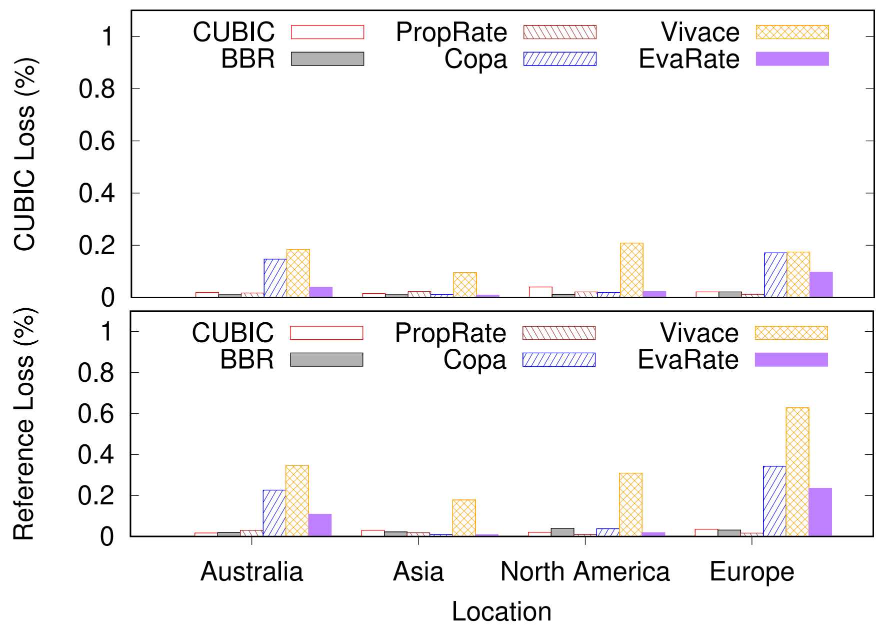


Figure 5.31: TCP loss rate of background CUBIC flows and reference flows with a larger receive window.

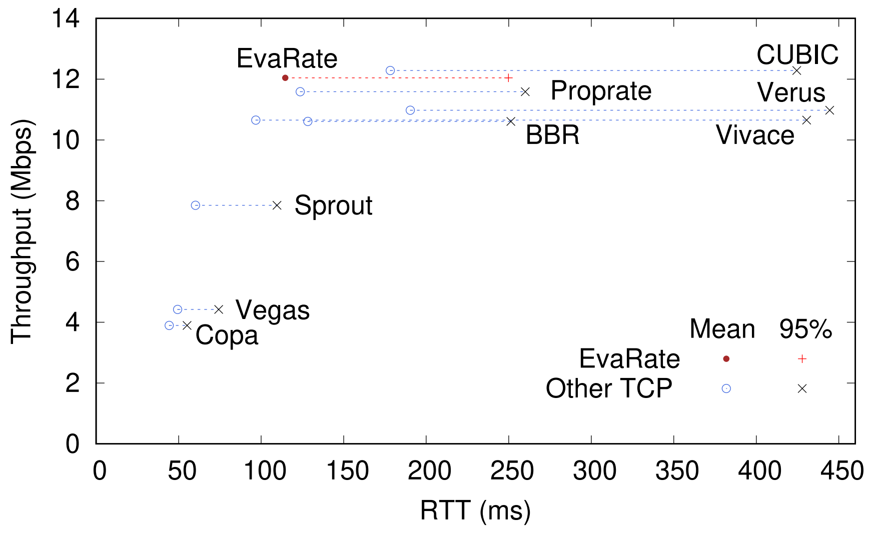


Figure 5.32: Mobile cellular network trace: M1.

Mobile Cellular Networks. For the mobile cellular network, we used a publicly-available set of cellular traces collected by Leong et al. [27]. The traces are collected using 3 local ISPs in Singapore (M1, Starhub and Singtel) on a shuttle bus moving around the university campus. We plot the results in Figures 5.32 to 5.34. We can see that although not specially designed for mobile cellular networks, EvaRate operates at an efficient point along the throughput-latency frontier, with a throughput close to CUBIC and a much lower latency than CUBIC. Also, in the result of Singtel trace, in which network condition varies significantly, Vivace becomes very aggressive and causes a much higher latency than other low-latency TCP variants. EvaRate manages to keep the overall latency low, although the throughput is lower than CUBIC, PropRate and Vivace.

Satellite Networks. For the satellite network, we use the measurement results from the WINDS satellite system: 42 Mbps link capacity, 800 ms RTT, $1 \times$ BDP buffer size and a 0.74% stochastic loss rate [34]. In each experiment,

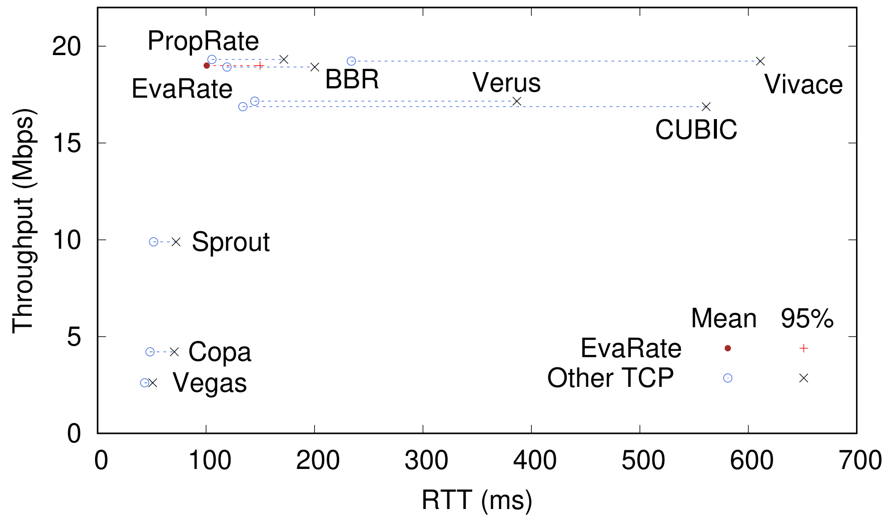


Figure 5.33: Mobile cellular network trace: Starhub.

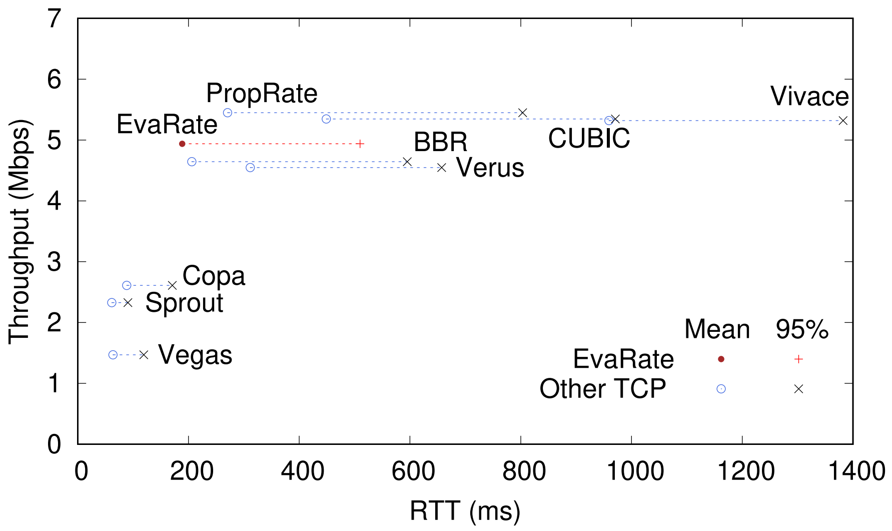


Figure 5.34: Mobile cellular network trace: Singtel.

a sender starts the connection and transmits data for 60 seconds. The results are presented in Figure 5.35. It is clear from these results that it is difficult to compare TCP variants directly because different algorithms are optimized for different trade-offs.

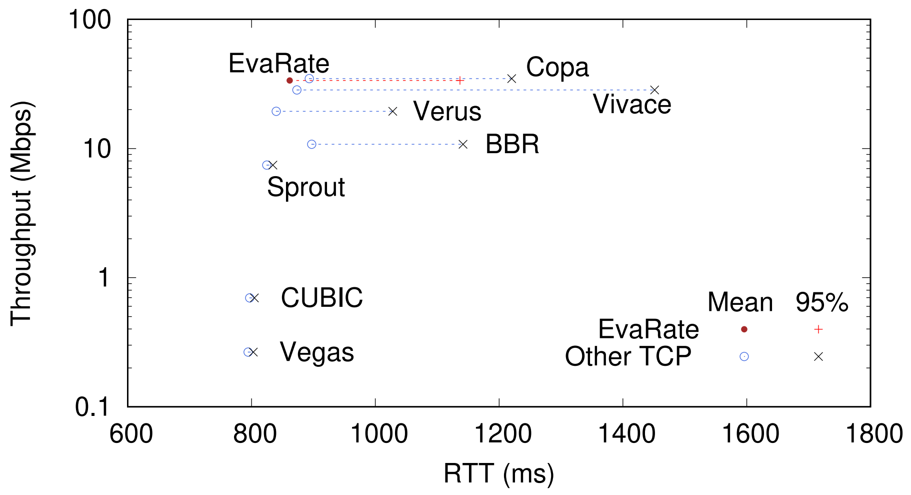


Figure 5.35: Satellite network trace.

5.8 Computation Overhead

In Figure 5.36, we compare the measured CPU overhead of EvaRate to the state-of-the-art algorithms for various Intel processors. We used `iperf` for CUBIC, BBR, PropRate and EvaRate, and custom clients for Verus, Vivace and Copa to generate TCP traffic and measured the CPU utilization ratio at the sender for three different CPUs. Our results show that EvaRate has low overheads comparable to those of CUBIC. It is worth noting however that Verus, Copa and Vivace are implemented using UDP in user space. It is likely that a kernel implementation of these algorithms would be more efficient.

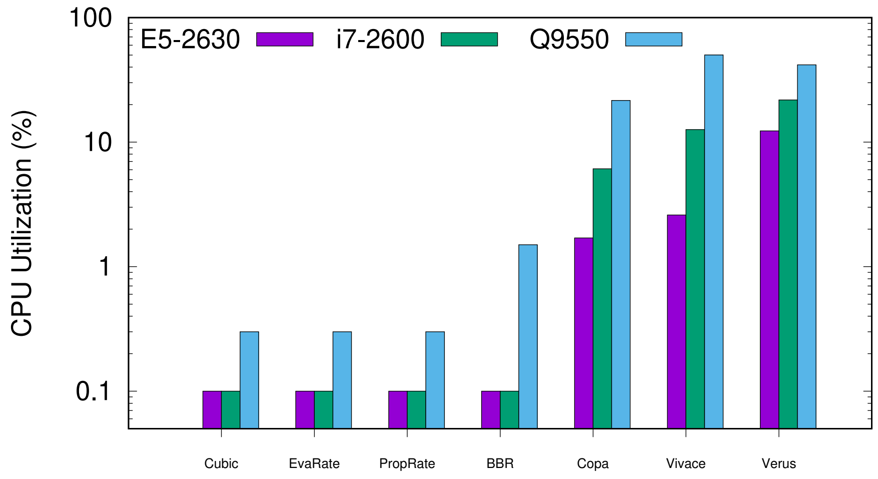


Figure 5.36: CPU utilization of state-of-the-art congestion control algorithms.

Chapter 6

Conclusion & Future Work

In this thesis, we first proposed an algorithm to detect and identify the TCP variant deployed at remote web servers. With the algorithm, we performed a measurement study to determine the distribution of various TCP variants in the Internet. Our results suggests that TCP CUBIC is still the most dominant TCP variant deployed in the Internet. In addition, we also found that BBR, a low-latency TCP variant proposed by Google, has been deployed "in the wild."

Given that TCP CUBIC is loss-based and aggressive, we investigated how the recently proposed low-latency TCP variants would interact with CUBIC. We found in our AWS experiments that, surprisingly, the new low-latency TCP variants were able to match, and even outperform CUBIC in terms of throughput, but at a higher loss rate. With further analysis, we concluded that the new low-latency achieved a high throughput by inflicting packet losses to other flows sharing the same bottleneck link buffer, due to them being insensitive to packet loss.

Based on the measurement study, we argue that it is not sufficient for a low-latency TCP variant, like BBR, Copa and Vivace, to be insensitive to losses. To operate effectively in the current Internet, it should also avoid causing significant degradation to existing flows to allow the Internet to transition smoothly to a benevolent future. Our key insight is that we can directly estimate not only the buffer occupancy for our flow, but also that of competing flows sharing the same bottleneck buffer. With our new approach to estimate the buffer occupancy of our own flow and the competing flows, EvaRate is able to detect and distinguish hostile environment and benevolent environment based on the variance of the buffer occupancy. EvaRate flows will collaboratively manage the bottleneck link buffer in a benevolent environment, and keep minimum number of packets in the buffer in a hostile environment. By designing and implementing a negative-feedback control loop that keeps the buffer occupancy low, EvaRate keeps latency low and avoids overflowing the buffer and inflicting loss on competing flows.

We then extensively evaluate the performance of EvaRate using both trace-driven emulations and Internet experiments. We made 3 major observations: i) EvaRate performs exactly as designed in the single-flow experiments, successfully keeping the buffer occupancy low and non-zero; ii) In the evolution experiments where BBR, Vivace, Copa and EvaRate are progressively deployed, we find that BBR, Vivace and Copa inflict packet losses on other CUBIC flows in shallow buffers, but EvaRate stays friendly to other competing CUBIC flows; iii) In the AWS Internet experiments, we find that BBR, Vivace and Copa outperform CUBIC by inflicting losses to competing CUBIC flows, while EvaRate achieves slightly lower throughput but is friendly to competing CUBIC flows.

Philosophically, we do not believe that there is a best or optimal congestion control algorithm. The best algorithm likely depends on the context. With sufficient information, an approach like Remy would probably work. That said, we can probably all agree that there is no good reason to require buffer overflows and packet losses for congestion control and it is timely to move beyond CUBIC.

EvaRate highlights a new point in the congestion control design space where instead of having a single response to observed network metrics, we incorporate an environment inference step to allow us to decouple the handling of the existing CUBIC-like TCP variants from that of a future low-latency utopia (or *benevolent* regime). It is a place that we all intuitively know should exist, and yet to the best of our knowledge, few have studied this "Never Never Land of TCPs" where low-latency transport protocols can live happily ever after.

6.1 Future Work

While we have investigated a number of issues related to EvaRate, there is still much room to explore. We highlight a few possibilities.

6.1.1 Distribution of TCP Variants

Although we estimated the distribution of 3 classes of TCP variants on the Internet in Chapter 3, we can do more to design an algorithm to identify the exact TCP variant of remote web servers. This will not only allow us to determine the deployment progress of state-of-the-art TCP variants, but also potentially provide valuable information for the parameter setting of controlled network simulation experiments. Also, with a more detailed measurement study, we can better understand the reason why low-latency TCP variants compete better than CUBIC in the real Internet.

6.1.2 Understanding Benevolence and Fairness

We have an intuitive understanding of benevolence and we have devised a simple way to detect it. However, we have found that benevolence is a concept that describes an operating environment more than an algorithm. For example, we found that while one BBR flow is benevolent to EvaRate, a shared bottleneck with many BBR flows is hardly benevolent.

Similarly, fairness should consider not only the share of bandwidth but also loss rate. We found that Vivace can be fair in terms of throughput, but at a very high loss rate. Essentially the reason is that if everyone behaves aggressive, each one will get similar share of throughput. Thus, we need a new definition of fairness to better handle this situation.

EvaRate seems to be benevolent to its own kind, but it is not clear whether this property holds if we scale up to a large number of flows. There is scope to study and perhaps define this concept of benevolence more precisely, and perhaps eventually develop an algorithm that is ”provably-benevolent.”

6.1.3 Achieving Ultra-low Latencies

We have seen that EvaRate was not able to achieve the low latencies reported for PropRate [27]. Leong et al. had earlier observed that in order to achieve ultra-low latencies, it is sometimes necessary to allow the buffer to empty for some periods of time. EvaRate attempts to keep the buffer full at all times, which explains why there seems to be a lower bound on how low RTT can go. We are confident that EvaRate can be tuned to achieve even lower latencies (potentially in a cellular network) by allowing the buffer to be emptied for some duration of time. However, unlike PropRate, our feedback loop is not directly related to the latency, so it is not straightforward to make EvaRate converge to a desired (low) latency even if the network conditions allow for it.

6.1.4 Reducing Estimation Errors

We can estimate our own buffer occupancy B accurately. However, the maximum available capacity C is a quantity that is extremely hard to measure or estimate accurately in a hostile environment. If we could find a technique to measure C accurately and faster, we would significantly improve our estimate of B', the buffer occupancy of competing flows. This information could potentially be used to improve fairness when contending with other (non-EvaRate) flows in a hostile environment. BBR has an interesting mechanism where it would stop sending for 400 ms every 10 s. Something similar might also be helpful for improving the measurement of network metrics for EvaRate.

6.1.5 Handling Packet Losses

The loss and fairness issues reported in this thesis suggest that packet loss should not be ignored. Instead, we need a better mechanism to understand what packet losses really mean to modern networks, and distinguish random packet losses and packet losses due to buffer overflow. Not reacting to packet losses is likely to cause a large number of packet drops for shallow buffers. We are looking into how EvaRate can best respond to packet losses and perhaps react differently to different packet loss patterns.

Bibliography

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of SIGCOMM '10*, 2010.

[2] Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based Congestion Control for the Internet. In *Proceedings of NSDI '18*, 2018.

[3] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques For Congestion Detection and Avoidance. In *Proceedings of SIGCOMM '94*, 1994.

[4] Carlo Caini and Rosario Firrincieli. TCP Hybla: A TCP Enhancement For Heterogeneous Networks. *International Journal of Satellite Communications and Networking*, 2004.

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *Queue*, 14(5):50:20–50:53, 2016.

[6] Vint Cerf and Robert E. Kahn. A Protocol for Packet Network Intercommunication. *Communications, IEEE Transactions on*, 22(5):637–648, 1974.

[7] Dah-Ming Chiu and Raj Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17(1):1 – 14, 1989.

[8] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proceedings of NSDI '15*, 2015.

[9] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *Proceedings of NSDI '18*, 2018.

[10] Sally Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Best Current Practice), 2003.

[11] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based Congestion Control For Unicast Applications. In *Proceedings of SIGCOMM '00*, 2000.

[12] Sally Floyd and Thomas R. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, 1999.

[13] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[14] Prateesh Goyal, Mohammad Alizadeh, and Hari Balakrishnan. Rethinking Congestion Control for Cellular Networks. In *Proceedings of HotNets '17*, 2017.

[15] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[16] Mario Hock, Roland Bless, and Martina Zitterbart. Experimental Evaluation of BBR Congestion Control. In *Proceedings of ICNP '17*, 2017.

[17] Mario Hock, Felix Neumeister, Martina Zitterbart, and Roland Bless. TCP LoLa: Congestion Control for Low Latencies and High Throughput. In *Proceedings of LCN '17*, 2017.

[18] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of SIGCOMM '88*, 1988.

[19] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Tackling Bufferbloat in 3G/4G Networks. In *Proceedings of IMC '12*, 2012.

[20] Cheng Jin, David Wei, and Steven Low. Fast TCP: Motivation, Architecture, Algorithms, Performance. In *Proceedings of INFOCOM '04*, 2004.

[21] Aditya Karnik and Anurag Kumar. Performance of TCP Congestion Control with Explicit Rate Feedback: Rate Adaptive TCP (RATCP). In *Proceedings of Globecom '00*, 2000.

[22] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of SIGCOMM '02*, 2002.

[23] Jun Ke and Carey Williamson. Towards a Rate-Based TCP Protocol for the Web. In *Proceedings of MASCOT '00*, 2000.

[24] Tom Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, 2003.

[25] Changhyun Lee, Chunjong Park, Keon Jang, Sue B Moon, and Dongsu Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proceedings of ATC '15*, 2015.

[26] Douglas Leith and Robert Shorten. H-TCP: TCP For High-Speed and Long-Distance Networks. In *Proceedings of PFLDnet '04*, 2004.

[27] Wai Kay Leong, Zixiao Wang, and Ben Leong. TCP Congestion Control Beyond Bandwidth-Delay Product for Mobile Cellular Networks. In *Proceedings of CoNEXT '17*, 2017.

[28] Wai Kay Leong, Yin Xu, Ben Leong, and Zixiao Wang. Mitigating egregious ACK delays in cellular data networks by eliminating TCP ACK clocking. In *Proceedings of ICNP '13*, 2013.

[29] Jim Martin, Arne Nilsson, and Injong Rhee. Delay-based Congestion Avoidance For TCP. *IEEE/ACM Transactions on Networking*, 11(3):356–369, 2003.

[30] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of MobiCom '01*, 2001.

[31] Sally Floyd Matt Mathis, Jamshid Mahdavi and Allyn Romanow. TCP Selective Acknolwdgement Options. RFC 2018, 1996.

[32] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of SIGCOMM '15*, 2015.

[33] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *Queue*, 10(5):20–34, 2012.

[34] Hiroyasu Obata, Kazuya Tamehiro, and Kenji Ishida. Experimental Evaluation of TCP-STAR for Satellite Internet over WINDS. In *Proceedings of ISADS '11*, 2011.

[35] Jitendra Padhye, Jim Kurose, Don Towsley, and Rajeev Koodli. A Model Based TCP-Friendly Rate Control Protocol. In *Proceedings of NOSSDAV '99*, 1999.

[36] K. K. Ramakrishnan, Sally Floyd, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, 2001.

[37] Michael Schapira and Keith Winstein. Congestion-Control Throwdown. In *Proceedings of HotNets '17*, 2017.

[38] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). IETF Working Draft, 2012.

[39] Prasun Sinha, Narayanan Venkitaraman, Raghupathy Sivakumar, and Vaduvur Bharghavan. WTCP: A Reliable Transport Protocol for Wireless Wide-Area Networks. In *Proceedings of MobiCom '99*, 1999.

[40] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An Experimental Study of the Learnability of Congestion Control. In *Proceedings of SIGCOMM '14*, 2014.

[41] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *Proceedings of INFOCOM '06*, 2006.

[42] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-generated Congestion Control. In *Proceedings of SIGCOMM '13*, 2013.

[43] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of NSDI '13*, 2013.

[44] Lisong Xu, K. Harfoush, and Injong Rhee. Binary Increase Congestion Control (BIC) For Fast Long-Distance Networks. In *Proceedings of INFOCOM '04*, 2004.

[45] Qiang Xu, Sanjeev Mehrotra, Zhuoqing Mao, and Jin Li. PROTEUS: Network Performance Forecast for Real-time, Interactive Mobile Applications. In *Proceeding of MobiSys '13*, 2013.

[46] Yin Xu, Wai Kay Leong, Ben Leong, and Ali Razeen. Dynamic Regulation of Mobile 3G/HSPA Uplink Buffer with Receiver-Side Flow Control. In *Proceedings of ICNP '12*, 2012.

[47] Peng Yang, Wen Luo, Lisong Xu, Jitender Deogun, and Ying Lu. TCP Congestion Avoidance Algorithm Identification. In *Proceedings of ICDCS '11*, 2011.

[48] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive Congestion Control for Unpredictable Cellular Networks. In *Proceedings of SIGCOMM '15*, 2015.