# Congestion Control in Highly Variable Networks

by

Prateesh Goyal

B.Tech., Indian Institute of Technology Bombay (2016)
S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

© Massachusetts Institute of Technology 2022. All rights reserved.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
November 12, 2021

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Hari Balakrishnan
Fujitsu Chair Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Mohammad Alizadeh
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

**Congestion Control in Highly Variable Networks**

by

Prateesh Goyal

Submitted to the Department of Electrical Engineering and Computer Science
on November 12, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

Modern applications place an enormous demand on networks to deliver high throughput and low delay. To support applications, computer networks are evolving rapidly. Several new network environments such as datacenter and wireless networks have emerged recently and become prominent. While bandwidth has been increasing steadily in these network environments, they also exhibit significant variability in network conditions. For example, the capacity of a cellular link varies with time. Deployed congestion control solutions struggle to adapt to these variations, and their performance is far from optimal in many environments: the feedback used by these schemes is often imprecise or fails to capture variations in the network conditions fast enough.

To improve performance, we need *accurate* and *timely* feedback. To this end, we advocate designing separate feedback mechanisms tailored specifically to the nuances of each network environment. Understanding how conditions are varying in each environment can help us unravel what kind of information about the network conditions can improve adaption to such variations. Additionally, the feedback mechanism should be *practical* and only involve changes that are within the administrative and hardware constraints of the given network environment. Following this philosophy, this dissertation contributes separate high performance congestion control solutions for three prominent network environments: (1) Wireless Networks; (2) Datacenter Networks; (3) Wide-area Internet.

*ABC* is a simple explicit congestion control protocol for network paths with wireless links. ABC adapts to variations in the link capacity quickly and accurately. Compared to deployed schemes, ABC either achieves 50% higher throughput for similar delays or 3× lower delays for similar throughput.

*BFC* is a practical per-hop per-flow flow control architecture for datacenter networks with bursty traffic. Compared to deployed schemes, BFC responds to congestion faster, and achieves 2.3 - 60× lower tail latency for short flows and 1.6 - 5× better average completion time for long flows.

*Nimbus* proposes a new feedback mechanism, elasticity detection, to robustly characterize the nature of cross-traffic competing a flow. Nimbus enables low delay conges-

tion control in the Internet without any router modifications. Compared to deployed schemes, Nimbus achieves 40-50 ms lower delays in the Internet for similar throughput.

Thesis Supervisor: Hari Balakrishnan
Title: Fujitsu Chair Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Mohammad Alizadeh
Title: Associate Professor of Electrical Engineering and Computer Science

*To my family*

# Acknowledgments

I am very grateful to have Professors Hari Balakrishnan and Mohammad Alizadeh as my advisors. They have been crucial in my growth as a researcher. They taught me how to do research, refine my ideas and make them more impactful, and to communicate and present my work effectively. Mohammad has been the one constant across all my PhD projects. I am thankful for the countless number of hours he spent mentoring and helping me. His tenacity and sheer dedication continues to inspire me. I will dearly miss our philosophical discussions and late night writing sessions. Hari has been a fatherly figure to me, mentoring me and encouraging me through the ups and the downs of my PhD, providing me invaluable career and life advice on innumerable occasions. I hope to replicate his enthusiasm and optimism in the face of adversity.

I am thankful to Professor Thomas E. Anderson for serving on my thesis committee, and being an incredible collaborator and mentor. Working closely with him on BFC and several other projects has been a sheer joy. I am grateful to my undergraduate advisor Professor S. Sudarshan for introducing me to the world of research. I consider myself fortunate to have been surrounded by amazing teachers throughout my education. My IIT JEE coaching teachers, Ashish Arora, Dhruv Kumar Banerjee and Abhijeet Jha, instilled in me the importance of hard work and dedication, and a continued enthusiasm for science. I have also been fortunate to mentor and learn from two very bright interns, Anup Agarwal and Preey Shah. I take immense pride in the work we did together.

I am grateful to all my other collaborators over the years: Ahmed Saeed, Akshay Narayan, Andreas Pavlogiannis, Anirudh Sivaraman, Deepti Raghavan, Frank Cangialosi, Georgios Nikolaidis, Kevin Zhao, Krishnendu Chatterjee, Mehrdad Khani, Radhika Mittal, Ravi Netravali, Ravichandra Addanki, Srinivas Narayana, Venkat Arun, Vibhaalakshmi Sivaraman and Vikram Nathan. I am inspired by your creativity, enthusiasm, intelligence, dedication, and I thank you all for your help. Each and every one of you has taught me something; how to write papers and build well

engineering systems are just two of them. You truly have helped me become a better researcher.

At MIT, I found a community within my labmates in the NMS group and other denizens of the Stata Center: Ahmed Saeed, Akshay Narayan, Anirudh Sivaraman, Arash Nasr, Arjun Balasingam, Darsh Shah, Deepak Vasisht, Frank Cangialosi, Hongzi Mao, Kapil Vaidya, Lei Yang, Manya Ghobadi, Mehrdad Khani, Pantea Karimi, Parimarjan Negi, Peter Iannucci, Pouya Hamadanian, Prafull Sharma, Radhika Mittal, Ravi Netravali, Ravichandra Addanki, Srinivas Narayana, Venkat Arun, Vibhaalakshmi Sivaraman, Vikram Nathan and many others. Thank you for all the stimulating conversations, the technical feedback, listening to my rants, and for being a friend. I will miss hanging out with you all.

I will be forever indebted to my parents, Rekha Goyal and Mukesh Goyal, and my sister, Antulika Goyal, for their unconditional love and unwavering faith in me. I dedicate this dissertation to my family.

# Contents


**1 Introduction** **27**

1.1 Motivation . . . 27

1.1.1 Modern Networks are Highly Variable . . . 28

1.1.2 Existing Solutions are not Enough . . . 30

1.2 Accurate, Timely, and Practical Feedback Mechanisms for Congestion Control . . . 31

1.3 Key Contributions . . . 33

1.3.1 Accel-Brake Control (ABC) . . . 33

1.3.2 Backpressure Flow Control (BFC) . . . 36

1.3.3 Nimbus . . . 38

1.3.4 Beyond this Dissertation . . . 40

1.4 Previous Papers and Organization . . . 42

**2 ABC: A Simple Explicit Congestion Controller for Wireless Networks** **43**

2.1 Introduction . . . 43

2.2 Motivation . . . 46

2.3 Design . . . 49

2.3.1 The ABC Protocol . . . 50

2.3.1.1 ABC Sender . . . 50

2.3.1.2 ABC Router . . . 51

2.3.1.3 Fairness . . . 53

2.3.1.4 Stability Analysis . . . 55

2.4 Coexistence . . . 56
2.4.1 Deployment with non-ABC Routers . . . 56
2.4.2 Multiplexing with ECN Bits . . . 57
2.4.3 Non-ABC Flows at an ABC Router . . . 59
2.5 Estimating Link Rate . . . 60
2.5.1 Wi-Fi . . . 60
2.5.2 Cellular Networks . . . 64
2.6 Discussion . . . 65
2.7 Evaluation . . . 66
2.7.1 Prototype ABC Implementation . . . 66
2.7.2 Experimental Setup . . . 67
2.7.3 Performance . . . 67
2.7.4 Coexistence with Various Bottlenecks . . . 72
2.7.5 Fairness among ABC and non-ABC Flows . . . 73
2.7.6 Additional Experiments . . . 76
2.8 Related Work . . . 77
2.9 Conclusion . . . 79

**3 Backpressure Flow Control** **81**
3.1 Introduction . . . 81
3.2 Motivation . . . 84
3.2.1 Limits of End-to-End Congestion Control . . . 86
3.2.2 Existing Solutions are Insufficient . . . 87
3.2.3 Revisiting Per-hop, Per-Flow Flow Control . . . 89
3.3 Design . . . 91
3.3.1 Design Constraints . . . 92
3.3.2 A Strawman Proposal . . . 93
3.3.3 Backpressure Flow Control (BFC) . . . 94
3.3.3.1 Assigning Flows to Queues . . . 95
3.3.3.2 Backpressure Mechanism . . . 97

3.4 Tofino2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 101
3.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
3.6 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
3.6.1 Tofino2 Evaluation . . . . . . . . . . . . . . . . . . . . . . . 104
3.6.2 Simulation-based Evaluation . . . . . . . . . . . . . . . . . . . 107
3.6.2.1 Setup . . . . . . . . . . . . . . . . . . . . . . . . . . 107
3.6.2.2 Performance . . . . . . . . . . . . . . . . . . . . . . . 109
3.6.3 Stress-testing BFC . . . . . . . . . . . . . . . . . . . . . . . 113
3.6.4 Understanding the Limits of BFC . . . . . . . . . . . . . . . . . . 115
3.6.5 Dynamic Queue Assignment . . . . . . . . . . . . . . . . . . . . 118
3.6.6 Additional Experiments . . . . . . . . . . . . . . . . . . . . . . 119
3.6.6.1 Multiple Traffic Classes . . . . . . . . . . . . . . . . . 120
3.6.6.2 Impact of Spatial Locality . . . . . . . . . . . . . . . . 122
3.6.6.3 Using TCP Slow-start . . . . . . . . . . . . . . . . . . 122
3.6.6.4 Cross Data Center Traffic . . . . . . . . . . . . . . . . 123
3.6.6.5 Dynamic vs. Stochastic Queue Assignment . . . . . . 125
3.6.6.6 Size of Flow Table . . . . . . . . . . . . . . . . . . . . 127
3.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

**4 Elasticity Detection: A Building Block for Internet Congestion Control 129**

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
4.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
4.3 Cross-traffic Estimation . . . . . . . . . . . . . . . . . . . . . . . . . 133
4.3.1 Estimating the Rate of Cross-traffic . . . . . . . . . . . . . . . . . 134
4.3.2 Elasticity Detection: Principles . . . . . . . . . . . . . . . . . . 136
4.3.3 Elasticity Detection: Practice . . . . . . . . . . . . . . . . . . . 138
4.3.4 Setting Parameters for Elasticity Detection . . . . . . . . . . . . 140
4.4 NimbusCC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
4.4.1 Mode Switching . . . . . . . . . . . . . . . . . . . . . . . . . . 143

4.4.2 Multiple NimbusCC Flows . . . . . . . . . . . . . . . . . . . . . 144
4.5 Visualizing NimbusCC . . . . . . . . . . . . . . . . . . . . . . . . . 147
4.6 Discussion and Limitations . . . . . . . . . . . . . . . . . . . . . . . 149
4.7 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
4.7.1 Performance Benefits from Elasticity Detection . . . . . . . . . 153
4.7.2 Robustness of Elasticity Detection . . . . . . . . . . . . . . . . 158
4.7.3 Performance When Assumptions Do Not Hold . . . . . . . . . . 162
4.7.4 Elasticity Detection with Multiple NimbusCC Flows . . . . . . 165
4.7.5 Testing on Internet Paths . . . . . . . . . . . . . . . . . . . . 167
4.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168

**5 Conclusion 171**
5.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172

**A ABC: Stability Analysis 175**

**B ABC: Miscellaneous Results 181**
B.1 BBR Overestimates the Sending Rate . . . . . . . . . . . . . . . . . . 181
B.2 Wi-Fi Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
B.3 Low Delays and High Throughput . . . . . . . . . . . . . . . . . . . . 181
B.4 ABC vs Explicit Control Schemes . . . . . . . . . . . . . . . . . . . . 182

**C BFC: Impact of Pause Threshold 185**

**D BFC: Deadlock Prevention 189**

**E BFC: Incremental Deployment 191**

**F BFC: Miscellaneous Results 193**
F.1 Comparison with Homa . . . . . . . . . . . . . . . . . . . . . . . . . 193
F.2 Parameter Sensitivity for Comparison Schemes . . . . . . . . . . . . . 198
F.3 Reducing Contention for Queues . . . . . . . . . . . . . . . . . . . . . 200
F.4 Incast Flow Performance . . . . . . . . . . . . . . . . . . . . . . . . 201

**G Nimbus: Miscellaneous Results** **203**

G.1 Nimbus Helps Cross Traffic . . . . . . . . . . . . . . . . . . . . . . . . 203

G.2 Cross-traffic Congestion Control Protocols . . . . . . . . . . . . . . . . 204

G.2.1 Multiple Elastic Flows using Different Congestion Control Protocols. . . . . . . . . . . . . . . . . . . . . . . . . . . 204

G.2.2 NimbusCC & Cubic v. BBR . . . . . . . . . . . . . . . . . . . 204

G.2.3 Elastic Flows, No ACK Clocking . . . . . . . . . . . . . . . . 206

G.3 Copa Mode Switching Errors . . . . . . . . . . . . . . . . . . . . . . . 206

G.3.1 CBR Cross Traffic . . . . . . . . . . . . . . . . . . . . . . . . 207

G.3.2 Elastic Cross Traffic . . . . . . . . . . . . . . . . . . . . . . . 208

G.4 Buffer size, RTT, and AQM . . . . . . . . . . . . . . . . . . . . . . . . 210

G.5 Using Different CC Algorithms with NimbusCC . . . . . . . . . . . . 210

# List of Figures


1-1 **Variations in link capacity for a cellular link.** . . . . . . . . . . . . . 28

1-2 **Variations in fairshare rate of a datacenter flow across different link speeds. The fair-share rate ($f(t)$) for a link of capacity $C$ shared by $N(t)$ flows is $C/N(t)$.** . . . . . . . . . . . . . . . . . . . . . . . 29

1-3 **Logical switch components in per-hop, per-flow flow control.** . . . 36

2-1 **Performance on a emulated cellular trace** — The dashed blue in the top graph represents link capacity, the solid orange line represents the achieved throughput. Cubic has high utilization but has very high delays (up to 1500 milliseconds). Verus has large rate variations and incurs high delays. Cubic+CoDel reduces queuing delays significantly, but leaves the link underutilized when capacity increases. ABC achieves close to 100% utilization while maintaining low queuing delays (similar to that of Cubic+CoDel). . . 46

2-2 **Feedback** — Calculating $f(t)$ based on enqueue rate increases $95^{th}$ percentile queuing delay by 2×. . . . . . . . . . . . . . . . . . . . . . . . . 52

2-3 **Fairness among competing ABC flows** — 5 flows with the same RTT start and depart one-by-one on a 24 Mbit/s link. The additive-increase (AI) component leads to fairness. . . . . . . . . . . . . . . . . . . . . . . . . 55

2-4 **Coexistence with non-ABC bottlenecks** — When the wired link is the bottleneck, ABC becomes limited by $w_{\text{cubic}}$ and behaves like a Cubic flow. When the wireless link is the bottleneck, ABC uses $w_{abc}$ to achieve low delays and high utilization. . . . . . . . . . . . . . . . . . . . . . . . . 57

2-5 **Inter-ACK time v. batch (A-MPDU) size** — Inter-ACK times for a given batch size exhibits variation. The solid black line represents the average Inter-ACK time. The slope of the line is $S/R$, where $S$ is the frame size in bits and $R$ is the link rate in bits per second. . . . . . . . . . . . . 62

2-6 **Wi-Fi Link Rate Prediction** — ABC router link rate predictions for a user that was not backlogged and sent traffic at multiple different rates over three different Wi-Fi links. Horizontal lines represent the true link capacity, solid lines summarize the ABC router's link capacity prediction (each point is an average over 30 seconds of predictions), and the dashed slanted line represents the prediction rate caps. ABC's link rate predictions are within 5% of the ground truth across most sending rates (given the prediction cap). 63

2-7 **ABC vs. previous schemes on three Verizon cellular network traces** — In each case, ABC outperforms all other schemes and sits well outside the Pareto frontier of previous schemes (denoted by the dashed lines). . . 68

2-8 **$95^{th}$ percentile per-packet delay across 8 cellular link traces** — On average, ABC achieves similar delays and 50% higher utilization than Copa and Cubic+Codel. PCC and Cubic achieve slightly higher throughput than ABC, but incur 380% higher $95^{th}$ percentile delay than ABC. . . . . . . . 69

2-9 **Throughout and mean delay on Wi-Fi** — For the multi-user scenario, we report the sum of achieved throughputs and the average of observed $95^{th}$ percentile delay across both users. We consider three versions of ABC (denoted ABC _*) for different delay thresholds. All versions of ABC outperform all prior schemes and sit outside the pareto frontier. . . . . . . . 71

2-10 **Coexistence with non-ABC bottlenecks** — ABC tracks the ideal rate closely (fair share) and reduces queuing delays in the absence of cross traffic (white region). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

2-11 **Coexistence among ABC flows** — ABC achieves similar aggregate utilization and delay irrespective of the number of connections. ABC outperforms all previous schemes. . . . . . . . . . . . . . . . . . . . . . . . . . 74

2-12 **RTT unfairness** . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

2-13 **Coexistence with non-ABC flows** — Across all scenarios, the standard deviation for ABC flows is small and the flows are fair to each other. Compared to RCP's Zombie List strategy, ABC's max-min allocation provides better fairness between ABC and non-ABC flows. With ABC's strategy, the difference in average throughput of ABC and Cubic flows is under 5%. . . 75

2-14 **Impact of propagation delay on performance** — On a Verizon cellular network trace with different propagation delays, ABC achieves a better throughput/delay tradeoff than all other schemes. . . . . . . . . . . . . 77

2-15 **ABC's robustness to flow size** — With a single backlogged ABC flow and multiple concurrent application-limited ABC flows, all flows achieve high utilization and low delays. . . . . . . . . . . . . . . . . . . . . . . . . . 78

2-16 **Impact of $\eta$** — Performance of ABC with various values of $\eta$ (target utilization). $\eta$ presents a trade-off between throughput and delay. Same setup as Fig. 2-1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

3-1 **Hardware trends for top-of-the-line data center switches from Broadcom** — Switch capacity and link speed have been growing rapidly, but buffer size is not keeping up with increases in switch capacity. . . . . . . . . . 85

3-2 Cumulative bytes contributed by different flow sizes for three different industry workloads. The three vertical lines show the BDP for a 10 Gbps, 40 Gbps, and 100 Gbps network, assuming a 12 $\mu$s RTT. . . . . . . . . . 85

3-3 Mean percent change in fair-share rate as a function of workload, delay, and bandwidth. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

3-4 **Number of active flows for different load, link speed, and scheduling policy** — Lines correspond to different loads. Flow sizes are from the Google distribution with lognormal ($\sigma = 2$) inter-arrival times. . . . . . 91

3-5 **Logical switch components in per-hop, per-flow flow control.** . . . 92

3-6 **Testbed topology** — The colored lines show the path for different flow groups. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

3-7 **Queue length and under-utilization** — 2 flows are competing at a 100 Gbps link. Cell size is 176 bytes. BFC achieves high utilization and low buffering. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

3-8 **Congestion spreading** — Dynamic queue assignment reduces HoL blocking, improving FCTs on average and at the tail. . . . . . . . . . . . . . . . 106

3-9 Google distribution with 55% load + 5% 100-1 incast. BFC tracks the ideal behavior, improves FCTs, and reduces buffer occupancy. For FCT slowdown, both the x and y axis are log scaled. . . . . . . . . . . . . . . . . . . . . . 109

3-10 FCT slowdown and buffer occupancy for Google distribution with 60% load. For all the schemes, PFC was never triggered. Part (c) shows the CDF of active flows at a port with and without incast, with the vertical bar showing the total number of queues per port. . . . . . . . . . . . . . . . . . . . . . 110

3-11 FCT slowdown ($99^{th}$ percentile) for Facebook distribution with and without incast. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

3-12 Average FCT slowdown for long flows, and $99^{th}$ percentile tail FCT slowdown for small flows, as a function of load. . . . . . . . . . . . . . . . . . . . . . 113

3-13 Average FCT slowdown for long flows, and $99^{th}$ percentile tail FCT slowdown for small flows, as a function of incast degree. . . . . . . . . . . . . . . . 114

3-14 Median FCT slowdown for mice flows in the presence of long-running flows. 116

3-15 $99^{th}$ percentile FCT slowdown when combined with congestion control. Facebook workload, same setup as Fig. 3-11. . . . . . . . . . . . . . . . . . . 117

3-16 FCT slowdown ($99^{th}$ percentile) and buffer occupancy of HPCC variants, using the setup in Fig. 3-11a. . . . . . . . . . . . . . . . . . . . . . . . 118

3-17 Multiple traffic classes with BFC, reporting $99^{th}$ percentile FCT slowdown for the Facebook workload, 60% load, and no incast. . . . . . . . . . . . . 121

3-18 Impact of spatial locality. FCT slowdown ($99^{th}$ percentile) for Facebook distribution with and without incast. . . . . . . . . . . . . . . . . . . . . 122

3-19 Impact of using slow start on median and 99th percentile tail latency FCT slowdown, for the Facebook flow size distribution with and without incast (setup the same as Fig. 3-11). With incast, DCTCP + SS (slow start) reduces the tail FCT, but it increases median FCTs by up to 2 ×. In the absence of incast, DCTCP + SS increases both the tail and median FCT for short and medium flows. . . . . . . . . . . . . . . . . . . . . . . . . 123

3-20 Performance in cross data center environment where two data center are connected by a 200 $\mu$s link, for the Facebook workload (60% load) with no incast traffic. The left figure shows the 99th percentile FCT slowdown for intra-data-center flows. The right figure shows the average utilization of the link connecting the two data centers. . . . . . . . . . . . . . . . . . . . . . 124

3-21 Performance of BFC with stochastic queue assignment, for the workload in Fig. 3-11a. BFC + Stochastic incurs more queue collisions leading to worse tail latency especially for small flows compared to BFC + Dynamic. . . . . 126

3-22 FCT slowdown (99th percentile) for BFC for different size flows as a function of the size of the flow table (as a multiple of the number of queues in the switch). The other experiments in the dissertation use a flow table of 100X. Further reducing the size of the flow table hurts small flow performance. . 126

4-1 **Network model** — The time-varying total rate of cross-traffic is $z(t)$. The bottleneck link rate is $\mu$. The sender's transmission rate is $S(t)$, and the rate of traffic received by the receiver is $R(t)$. . . . . . . . . . . . . . . . 135

4-2 **Instantaneous delay measurements do not reveal elasticity** — The bottom plot shows the total queueing delay (orange) and the self-inflicted delay (green). The experiment contains one background Cubic flow in the elastic region (30–90 s) and CBR cross-traffic in the inelastic region (90–150 s). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136

4-3 **Cross-traffic's reaction to pulses** — The pulses change the inter packet spacing for cross-traffic. Elastic traffic reacts to these changes after a RTT, while inelastic traffic does not. . . . . . . . . . . . . . . . . . . . . . . 137

4-4 **Cross-traffic FFT for elastic and inelastic traffic** — Only the FFT for elastic traffic has a pronounced peak at $f_p$ (5 Hz). . . . . . . . . . . . 139

4-5 **Distribution of elasticity with varying elastic fraction of cross-traffic** — The cross-traffic consists of an elastic Cubic flow and inelastic Poisson-distributed traffic with different rates. Completely inelastic cross-traffic has $\eta$ close to zero, while completely elastic cross-traffic exhibits a high $\eta$. Cross-traffic with some elastic fraction also exhibits high elasticity ($\eta > 2$). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140

4-6 **Asymmetric sinusoidal pulse** — The pulse has period $T = 1/f_p$. The positive half-sine lasts for $T/4$ with amplitude $\mu/4$, and the negative half-sine lasts for the remaining duration, with amplitude $\mu/12$. The two half-sines cancel out each other over one period. . . . . . . . . . . . . . . . . . . . . 141

4-7 Performance on a 96 Mbit/s Mahimahi link with 50 ms delay and 2 BDP of buffering while varying the rate and type of cross-traffic as denoted at the top of the graph. $x$M denotes $x$ Mbit/s of inelastic Poisson cross-traffic. $y$T denotes $y$ long-running Cubic cross-flows. The solid black line indicates the correct time-varying fair-share rate that the protocol should achieve given the cross-traffic. For each scheme, the solid line shows throughput and the dotted line shows queuing delay. The cross-traffic contains elastic flows from 20–120 s. For Nimbus and Copa, the red shaded regions indicate times spent in the wrong mode (e.g., delay-controlling with elastic cross-traffic). . . . . 148

4-8 Performance of NimbusCC on a cross traffic workload derived from a packet trace collected at a WAN router. . . . . . . . . . . . . . . . . . . . . . . . 155

4-9 The elasticity metric closely tracks elastic cross traffic (ground truth measured independently from the rate of ACK-clocked flows). Green-shaded regions indicate inelastic periods. . . . . . . . . . . . . . . . . . . . . . . . 157

4-10 Mean throughput and queuing delay (lower delay on the right) with video cross traffic. NimbusCC achieves similar throughput as Cubic but reduces delays and performs better than the other schemes. Copa and Vegas achieve low throughput. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158

4-11 Nimbus is robust to variations in link bandwidth and fraction of traffic controlled by it. The accuracy is high even when the fraction of traffic under control is small. Increasing pulse size increases robustness. . . . . . . . . . 159

4-12 Nimbus is more accurate than Copa when (i) inelastic cross traffic occupies a large fraction of the link (left); (ii) elastic cross traffic has higher RTT than the flow's RTT (right). . . . . . . . . . . . . . . . . . . . . . . . . 160

4-13 **Impact of $\eta_{thresh}$** — With a high $\eta_{thresh}$, NimbusCC operates in delay-controlling mode more often, reducing delays but losing throughput against elastic cross traffic (see the $10^{\text{th}}$ percentile in the throughput profile, shown in red) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162

4-14 **Impact of incorrect $\mu$** — When the error is high, all the traffic is classified as elastic and NimbusCC operates in TCP-competitive mode. . . . . . . . 163

4-15 **Multiple competing NimbusCC flows** — Multiple NimbusCC flows achieve fair sharing of a bottleneck link (top graph). There is at most one pulser flow at any time; identified by its rate variations. Together, the flows achieve low delays by staying in delay mode for most of the duration (bottom graph). The red background shading shows when a NimbusCC flow was (incorrectly) in competitive mode . . . . . . . . . . . . . . . . . . . . 165

4-16 **Multiple NimbusCC flows and other cross traffic** — There are 3 NimbusCC flows throughout. Cross traffic in 30-90 s is elastic and made up of 3 Cubic flows. Cross traffic in 90-150 s is inelastic and made up of a 96 Mbit/s constant bit-rate stream. NimbusCC flows achieve their fair share (top) while achieving low delays in the absence of elastic cross traffic (bottom). 166

4-17 **Performance on three example Internet paths** — The $x$ axis is inverted; better performance is up and to the right. On paths with buffering and no drops, ((a) and (b)), NimbusCC achieves the same throughput as BBR and Cubic but reduces delays significantly. On paths with significant packet drops (c), Cubic suffers but NimbusCC achieves high throughput. . 167

4-18 **Paths with queuing** — NimbusCC reduces the RTT compared to Cubic and BBR (upto 50ms), at similar throughput. . . . . . . . . . . . . . . . . 168

B-1 **Comparison with BBR** — BBR overshoots the link capacity, causing excessive queuing. Same setup as Fig. 2-1. . . . . . . . . . . . . . . . . . . 182

B-2 **Throughput and $95^{th}$ percentile delay for a single user in WiFi** — We model changes in MCS index as bownian motion, with values changing every 2 seconds. We limit the MCS index values to be between 3 and 7. ABC outperforms all other schemes. . . . . . . . . . . . . . . . . . . . . . 182

B-3 **Utilization and mean per-packet delay across 8 different cellular network traces** — On average, ABC achieves similar delays and 50% higher utilization than Copa and Cubic+Codel. BBR, PCC, and Cubic achieve slightly higher throughput than ABC, but incur 70-240% higher mean per-packet delays. . . . . . . . . . . . . . . . . . . . . . . . . . 183

B-4 **ABC vs explicit flow control** — ABC achieves similar utilization and $95^{th}$ percentile per-packet delay as XCP and $XCP_w$ across all traces. Compared to RCP and VCP, ABC achieves 20% more utilization. . . . . . . . 184

B-5 **Time series for explicit schemes** — We vary the link capacity every 500ms between two rates 12 Mbit/sec and 24 Mbit/sec.The dashed blue in the top graph represents bottleneck link capacity. ABC and $XCP_w$ adapt quickly and accurately to the variations in bottleneck rate, achieving close to 100% utilization. RCP is a rate base protocol and is inherently slower in reacting to congestion. When the link capacity drops, RCP takes time to drain queues and over reduces its rates, leading to under-utilization. . . . . 184

C-1 Impact of pause threshold ($Th$) on the metric of worst case inefficiency. Increasing $Th$ reduces the maximum value for the fraction of time $f$ can run out of packets at the bottleneck. . . . . . . . . . . . . . . . . . . . . . . . 187

E-1 FCT slowdown ($99^{th}$ percentile) and buffer occupancy distribution for two BFC variants. When NICs don't respond to backpressure (BFC - NIC), BFC experiences moderate increased buffering. Using sampling to reduce recirculation (BFC + sampling) has marginal impact on performance. . . . 192

F-1 FCT slowdown on an oversubscribed clos topology. With packet spraying, Homa encounters minimal congestion in the core and outperforms other schemes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 194

F-2 BFC's dynamic queueue assignment achieves a better approximation of the SRF scheduling policy. BFC-SRF achieves close to optimal FCTs. . . . . . 196

F-3 FCT slowdown with 100-1 incast. Collisions in BFC-SRF can cause priority inversions hurting FCTs . . . . . . . . . . . . . . . . . . . . . . . . . . 197

F-4 **Parameter sensitivity for comparison schemes** — $99^{th}$ percentile FCT slowdown for the Facebook workload, 60% load without incast. Sensitivity to the choice of parameters in HPCC, DCTCP, and ExpressPass. . . . . . 199

F-5 FCT slowdown for short and long flows as a function of incast degree. The x axis is not to scale. By isolating incast flows, BFC + IncastLabel reduces collisions and achieves the best performance. . . . . . . . . . . . . . . . 200

F-6 FCT slowdown for incast traffic. Slowdown is defined per flow. BFC reduces the FCT for incast flows compared to other feasible schemes. Setup from Fig. 3-9. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 201

G-1 Using NimbusCC reduces the p95 FCT of cross-flows relative to BBR at all flow sizes, and relative to Cubic for short flows. Vegas provides low cross-flow FCT, but its own rate is low. . . . . . . . . . . . . . . . . . . . . . . . . 204

G-2 Performance with WAN cross traffic consisting of an equal mix of Cubic, NewReno and BBR flows. The deviation profile of NimbusCC is similar to that of Cubic, however, NimbusCC reduces delays. . . . . . . . . . . . . . 205

G-3 **NimbusCC's performance against BBR is similar to that of Cubic—** Both NimbusCC and Cubic compete against 1 BBR flow on a 96 Mbit/s link. For various buffer sizes, NimbusCC achieves the same throughput as Cubic. 205

G-4 By modifying the pulse frequency, Nimbus correctly classifies PCC-Vivace, a rate-based elastic protocol, as elastic. . . . . . . . . . . . . . . . . . . . 206

G-5 When the CBR traffic is low (a), Copa classifies the traffic as non buffer-filling and is able to achieve low queuing delays. But when the CBR traffic occupies a high fraction (c), Copa incorrectly classifies the traffic as buffer-filling, resulting in higher queuing delays. In both the situations (b and d), the elasticity detector correctly classifies the traffic as inelastic and NimbusCC achieves low queuing delays. . . . . . . . . . . . . . . . . . . . . . 207

G-6 **Queuing delay and throughput dynamics for elastic cross traffic** — When the elastic cross traffic increases fast enough (a), Copa classifies it as buffer-filling and is able to achieve its fair share. But when the elastic cross traffic increases slowly (c), Copa incorrectly classifies the traffic as non-buffer-filling, achieving less than its fair share. In both the situations (b and d), Nimbus correctly classifies the traffic as elastic and NimbusCC achieve its fair share. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 209

G-7 **NimbusCC's versatility**— NimbusCC with different combinations of delay-controlling and TCP-competitive algorithms. . . . . . . . . . . . . . . . . 211

# List of Tables

1.1 Dissertation overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

2.1 RTT unfairness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

3.1 For a shared 100 Gbps link, BFC achieves close to ideal throughput (40%) for the long flow, with low tail queuing delay. . . . . . . . . . . . . . . . . 86

4.1 Average queuing delay (in ms) in the inelastic region, and deviation from fairshare throughput in elastic and inelastic regions from Fig. 4-7. NimbusCC is the only scheme to achieve close to fair-share throughput and low delays. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

4.2 Classification by Nimbus. . . . . . . . . . . . . . . . . . . . . . . . . . . 150

4.3 Performance on time-varying links. . . . . . . . . . . . . . . . . . . . . . 164

4.4 Performance on a topology with multiple bottlenecks. . . . . . . . . . . . 164

4.5 Impact of $\kappa$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166

F.1 Per-packet queuing delay for scheduled traffic in the core. . . . . . . . . . 195

# Chapter 1

# Introduction

> Change is the only constant.
>
> ---
>
> Heraclitus

## 1.1 Motivation

Congestion control is one of the most prominent problems in computer networking. Early solutions for congestion control primarily focussed on solving the problem of "congestion collapse" [73] in wide area networks (WANs). The goal was to keep the networking running in the event of congestion. Since the 1990's, computer networks have undergone a tremendous transformation. This transformation necessitates that we think of congestion control in a new light. To understand why, we need to understand why and how computer networks have evolved.

New and emerging applications place immense demand on computer networks to deliver high throughput, low latency communication. To support applications, several new network environments have emerged recently and become prominent. Large-scale distributed computing have lead to the development of modern datacenter networks that connect millions of machines together. The need for high speed, low latency mobile Internet connectivity has lead to advancements in both Wi-Fi and Cellular networks. Even traditional WANs have transformed; we have seen a continuous in-

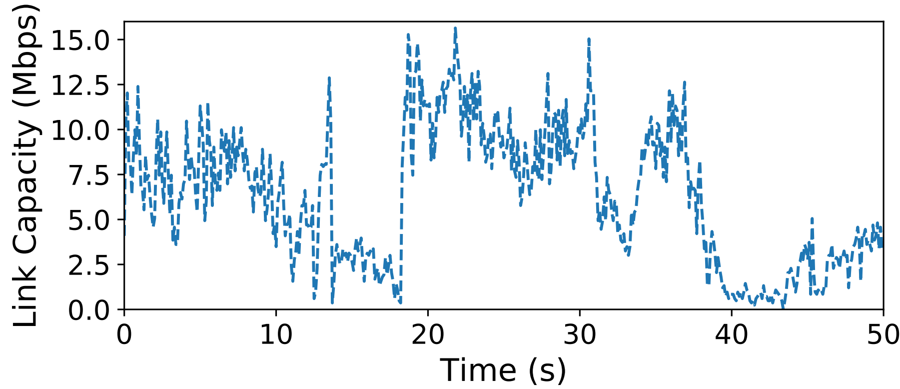



Figure 1-1: **Variations in link capacity for a cellular link.**

crease in link speeds and an increase in heterogeneity of protocols managing the network. These different environments differ from traditional networks in many aspects, but they share a common characteristics: they are highly variable. Congestion control is challenging in such environments.

Congestion control aims to provide high throughput and low latency for data transfers. At a high level, congestion control aims to admit as much traffic as possible into the network without causing any congestion. Typically, the sender (or the router) continuously adjusts the transfer rate of a flow based on feedback from the network. Feedback signals such as round trip time (RTT), packet drops, etc. provide information about the state of the network. It is difficult to adapt the transfer rate in modern networks with rapidly varying network conditions.

### 1.1.1 Modern Networks are Highly Variable

In this dissertation, we target the three most prominent, highly variable networks: (1) Wireless networks; (2) Datacenter networks; (3) Wide-area Internet.

**Wireless Networks:** Unlike wired links, wireless links (both Cellular and Wi-Fi) can exhibit great variations in link capacity over time. Within a fraction of a second, the link capacity can double or reduce to half. These variations can occur for several reasons. For example, if the cellular receiver is moving, then the instantaneous channel quality can change causing variations in the link capacity. Similarly, if the number of users competing at a cellular base station is changing, then the frequency bands

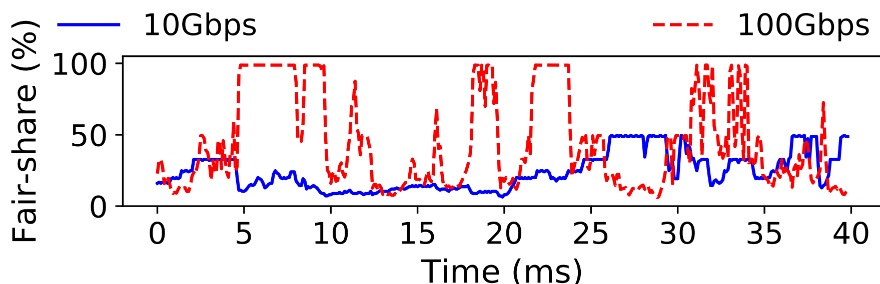



Figure 1-2: **Variations in fairshare rate of a datacenter flow across different link speeds. The fair-share rate ($f(t)$) for a link of capacity $C$ shared by $N(t)$ flows is $C/N(t)$.**

and time slots allocated to a user can change causing variations in the link capacity. Fig. 1-1 shows the variations in link capacity for a Verizon LTE link.

The key challenge for congestion control on such links is to adapt to variations in the link capacity. Sending traffic below the link capacity causes underutilization and loss of throughput. Exceeding the link capacity builds up a queue at the router, degrading latency.

**Datacenter Networks:** Modern datacenters have high speed links (up to 400 Gbps). Majority of the traffic is composed of very short flows that last only a few RTTs on such high speed links. Cross-traffic competing with a flow can be very bursty in such settings. The appropriate transfer rate for a flow can change significantly on RTT timescales. Fig. 1-2 shows variations in the fairshare rate of a long running flow at a single datacenter link when competing with cross-traffic across different linkspeeds. The cross-traffic flow sizes are derived from an industry workload, and the average cross-traffic load is 60%. As link speeds increase, cross-traffic flows arrive and depart more quickly, the fairshare rate of the long flow fluctuates more significantly.

Adapting to traffic conditions quickly and limiting congestion is critical in datacenters. In particular, datacenter operators often care about the tail latency of flows, even a small amount of queue build up can degrade performance [41]. Further, datacenter switches often have small buffers, slow response to congestion can cause packet drops which can be particularly expensive in such settings [150].

**Wide-area Internet:** The last decade has seen widespread deployment of many

new congestion control protocols such as Cubic [63], Compound [133], BBR [36], etc. Congestion control in the Internet is no longer governed by a single solution. A flow on the Internet can now end-up competing with flows which are using different congestion control protocols. The throughput and latency of the flow is not only dependent on its own congestion control but also on the protocols used by competing flows which are unknown to the sender.

This uncertainty and heterogeneity is a deterrent to achieving both high throughput and low delays in the Internet. A flow can use a deployed congestion control protocol to compete appropriately with the cross-traffic. However, the deployed protocols often fill up the buffers and incur high delays. To solve this, researchers have proposed many "delay-controlling" protocols (e.g., Vegas [31], FAST [140], LED-BAT [121], Copa [22]) that can provide lower delays compared to these deployed protocols. However, the throughput of such delay-controlling protocols is dismal when competing against deployed protocols.

Our thesis is that, to overcome the throughput-delay trade-off, congestion control should take into account the *nature* of competing cross-traffic flows. The salient aspect of this nature is cross-traffic *elasticity*. If the cross-traffic is elastic, i.e., it is trying to grab more bandwidth at the bottleneck, then the sender should use a "competitive" congestion control protocol to compete appropriately without necessarily attempting to reduce delays. Otherwise, the sender can safely use a delay-controlling protocol to reduce delays without worrying about losing throughput.

### 1.1.2 Existing Solutions are not Enough

Existing congestion control solutions are far from optimal in these highly variable networks. There are two reasons. First, these solutions make congestion control decisions based on feedback which is inaccurate/imprecise. Such feedback signals do not provide all the necessary information required to adapt the transfer rate. For example, packet drops or explicit congestion notification (ECN) [118] marks only signal congestion and do not provide any information about the degree to which the bottleneck link is underutilized. In wireless networks, when the link capacity opens

up and network becomes underutilized, such signals do not provide information to the sender on how to increase the rate and match the link capacity. Similarly, in wide-area Internet, existing feedback mechanisms don't provide any information on the nature of cross-traffic that can help guide congestion control decisions. Because of imprecise feedback, existing schemes face a trade-off in both these environments: they either achieve low latency or high throughput but not both.

Second, typically the sender adjusts the sending rate based on feedback signals echoed by the receiver. There is a delay of one end-to-end RTT in the feedback. In datacenter networks, the state of the network and the appropriate sending rate can change significantly within the feedback delay. Acting on such stale information is particularly bad for tail latency of flows. To get around this problem, datacenter operators often run their network at low utilization wasting usable bandwidth.

## 1.2 Accurate, Timely, and Practical Feedback Mechanisms for Congestion Control

Our thesis is that we can improve congestion control performance by using *accurate*, *timely*, and *practical* feedback mechanisms that help adapt well to variations in the network conditions. A one-size-fits-all mechanism that ignores the differences in these different network environments is inefficient in many scenarios. Instead, we advocate for separate feedback mechanisms for different environments that take into account the variability, the hardware capabilities, and the administrative constraints of the given environment. We consider the following related factors in designing new feedback mechanisms:

**Who can react to the feedback?** Ideally, we want to leverage as much router/switch support as possible for congestion control. However, typically only the sender can adjust the transfer rate of a flow. In datacenter networks, it is also feasible to control the transfer rate at the switches.

**Who can generate the feedback?** In the wide-area Internet, it is challenging to

modify the routers to generate any additional feedback at the routers. In such cases, it is still possible to generate feedback at the sender using only end-to-end measurements such as packet rates and delays (§1.3.3). In wireless networks, it might be possible to modify the wireless router (e.g., the cellular base station) to generate feedback. In datacenter networks, all the switches in the network can generate feedback.

**How to communicate the feedback?** An obvious first choice is for the router to specify the feedback in the packet header itself. However, using an arbitrary number of additional feedback bits in the packet header might not be feasible in wireless networks. In such cases, we need to come up with parsimonious feedback signals that fit within the existing bits allocated for feedback in the packet header. Alternatively, in datacenter networks it is even feasible to generate out of band packets for feedback. However, we need to be cognizant of the bandwidth and computational overhead in communicating the feedback.

**What kind of feedback mechanisms can improve performance?** A good starting point is identifying the main factors causing variations in the network conditions and focusing on factors that existing schemes don't address well (§1.1.2). In the wide-area Internet, feedback on whether the cross-traffic is competing to grab more bandwidth can guide congestion control decisions at the sender and reduce delays. In wireless networks, explicit feedback on how to adapt to the variations in the link capacity can improve performance. In datacenter networks with bursty traffic, hop-by-hop per-flow congestion control – where each switch generates backpressure feedback for flows causing congestion to throttle their transfer rates at the previous-hop (upstream) switch – can enable faster response to congestion and reduce tail latency.

**How to compute and react to the feedback?** Computing feedback often involves using underlying information such as the queue size or the enqueue/dequeue rate at the router, the ACK arrival rate at the sender, etc. Using the right information can have a substantial impact on performance. For example, in wireless networks, we show that using the dequeue rate at the router can help us compute more accurate feedback compared to existing schemes that ignore it [81, 132]. Further, we also need

| Scheme | ABC [58, 57] | BFC [60] | Nimbus [59] |
|---|---|---|---|
| Environment | Wireless | Datacenter | Wide-area Internet |
| Variability | Link capacity | Cross-traffic (load) | Cross-traffic (nature) |
| Feedback Mechanism | Accurate, single-bit explicit feedback | Per-hop per-flow flow control | Elasticity detection |
| Benefit | Fast adaptation to variations in the link capacity | Faster response to congestion | Low delay congestion control without compromising throughput |

Table 1.1: Dissertation overview

to take into account the capabilities of the underlying hardware. For example, per-hop per-flow congestion control is challenging on modern datacenter switches in-part because they have limited memory for maintaining state required to compute the feedback.

## 1.3 Key Contributions

We apply the aforementioned philosophy to propose three high performance, practical congestion control solutions for highly variable networks:

1. ABC: A simple explicit congestion controller for network paths with wireless links.
2. BFC: A practical congestion control architecture for datacenter networks that achieves an approximation of per-hop per-flow flow control.
3. Nimbus: A robust end-to-end technique for wide-area Internet to detect whether the cross-traffic competing with a flow is elastic or not.

Table 1.1 presents a brief overview.

### 1.3.1 Accel-Brake Control (ABC)

The principal challenge for congestion control on wireless links is to track variations in the link capacity. Congestion control protocols like Cubic [63], NewReno [66], and

BBR [36] adjust the sending rate of a flow based on traditional feedback signals such as ECN marks, packet drops, RTT, etc. from the network. Such protocols are good at inferring congestion to reduce the sending rate. However, when the wireless link capacity increases and the link becomes underutilized, these conventional feedback signals do not reveal any information about the extent of underutilization. In such periods, the sender has to resort to some sort of blind increase in the rate which can degrade performance. If the rate increase is slow, the throughput suffers. But if it overshoots the capacity, it causes large queuing delays. Using active queue management (AQM) schemes like PIE [112] or Codel [111] at the router is also not enough as these schemes do not provide any feedback on how to increase the rate when the link is underutilized.

Explicit congestion control protocols like XCP [81] and RCP [132] can improve performance by enabling the bottleneck wireless router to signal both rate increases and decreases to the sender via packet headers. However, current explicit protocols have two key limitations. First, existing schemes face deployment challenges as they require multi-bit per packet feedback, and consequently major changes to the packet header format and end-points. Next, these schemes were designed for fixed capacity links and their performance is sub-optimal on time-varying links. The primary contribution of ABC is two general techniques that overcome these limitations.

**Signal rate increase and decrease using a single-bit:** In ABC, the wireless router marks each packet with an accelerate or a brake. The accel-brake marks are echoed back to the sender via acknowledgements (ACKs). On receiving an "accelerate" in an ACK, the sender increases its congestion window by one and sends out two packets in response to the ACK. On receiving a "brake", the sender reduces its congestion window by 1 and sends nothing in response the ACK. By controlling the fraction of packets that are marked accelerate, the ABC router can vary the sender's rate over a wide dynamic range: from zero to double the current window within a RTT. Such a dynamic range allows for fast adaptation to changes in the bottleneck link capacity. To ease deployment, ABC can reuse the existing ECN bit in the packet header for accel-brake feedback. ABC doesn't require any packet header modifications.

**Compute accurate feedback using the dequeue rate:** Informally, the goal of an explicit congestion control protocol is to match the rate of packets arriving at the bottleneck router (enqueue rate) to the link capacity. Existing approaches compare the instantaneous enqueue rate at the router to the link capacity to compute feedback. ABC proposes a novel control loop for computing the accel-brake feedback: an ABC router instead uses the dequeue rate of packets leaving the router. This change is rooted in the simple observation that, dequeue rate at the router along with feedback marked by the ABC router provides an accurate prediction of enqueue rate one RTT in the future. Comparing this future enqueue rate to the link capacity allows for faster convergence to the bottleneck link capacity.

**Coexistence with non-ABC routers:** For practicality, there is another challenge we need to overcome. An ABC flow may encounter both ABC and non-ABC routers on its path. For example, a Wi-Fi user's traffic may traverse both a Wi-Fi router (running ABC) and an ISP router (not running ABC); either router could be the bottleneck at any given time. The ABC flow must adapt to any congestion on the non-ABC router. To solve this, we propose a simple dual-congestion control loop solution where the ABC sender maintains two congestion windows. $w_{abc}$ obeys the accel-brake feedback and tracks the available bandwidth on the ABC router. $w_{nonabc}$ tracks the available bandwidth on the non-ABC router using a standard control loop (e.g., Cubic) based on legacy feedback signals such as packet drops. The ABC sender simply uses the minimum of the two windows.

We evaluate ABC using a Wi-Fi implementation and trace-driven emulation of cellular links. ABC achieves 30-40% higher throughput than Cubic+Codel [111] for similar delays, and 2.2× lower delays than BBR on a Wi-Fi path. On cellular network paths, ABC achieves 50% higher throughput than Cubic+Codel, and 2× lower delay than XCP for similar throughput.

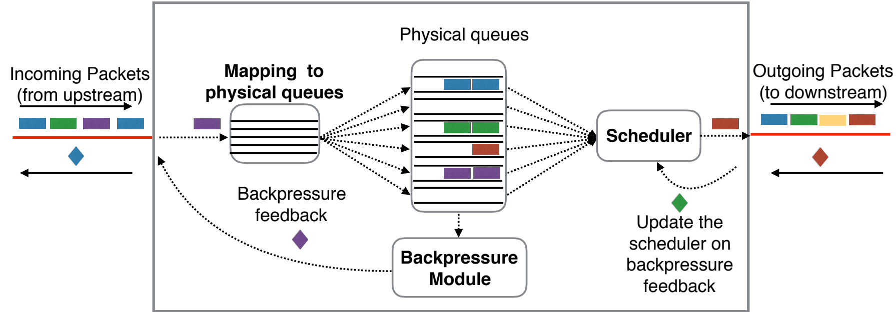



Figure 1-3: **Logical switch components in per-hop, per-flow flow control.**

### 1.3.2 Backpressure Flow Control (BFC)

Achieving low tail latency and high throughput is challenging in datacenter networks with bursty traffic. Most deployed congestion control protocols like DCQCN [150], HPCC [96], and DCTCP [14] adjust the sending rate based on feedback signals from the network. Typically, it takes one RTT for the sender to learn about the congestion and reduce its rate. Bursty traffic coupled with slow reaction can cause queue build up and packet loss at the switches, both of which hurt tail latency.

BFC revisits the old idea of per-hop per-flow flow control. Ideally, each flow would get its own queue at the switch. When a flow builds up a queue at the switch, the switch generates backpressure feedback for the flow and sends it to the previous-hop (upstream) switch. On receiving feedback, the upstream switch throttles the queue corresponding to the flow without affecting other competing flows.

Fig. 1-3 shows the basic components of a per-hop, per-flow flow control scheme. Per-hop control implies faster response to congestion, per-flow queue and feedback implies no head-of-line (HoL) blocking.

However, existing per-hop per-flow flow control schemes [19, 91] are not practical on existing datacenter switches. Modern programmable switches have three key constraints: (1) Limited memory for bookkeeping; (2) Limited queues for per-flow control, flows might have to share a queue and incur HoL blocking; (3) Support for only simple constant-time per-packet operations. The main contribution of BFC is a set of three simple ideas that together achieve an approximation of per-hop per-flow flow control on a modern programmable switch.

**Track only active flows:** Existing schemes need per-flow state and dedicated queues for all connections going through the switch. At any given time, many of these connections might be dormant and not have any packets in the switch. In contrast, a BFC switch only tracks state for and assigns a queue to active flows that have packets queued at the switch. Additionally, we show that the number of active flows is modest when the switch uses certain BFC compliant scheduling policies such as fair queueing or shortest flow first.

**Dynamically assign flows to queues:** A BFC switch tracks empty queues available at the switch. When a new flow arrives, the switch assigns it an empty queue if one is available, otherwise the flow is assigned a queue at random. As long as the number of active flows is less than the number of queues, no two flows share a queue and there is no HoL blocking. In contrast, stochastically assigning flows to queues like SFQ [103] can cause flows to share a queue even when there are empty queues available at the switch, degrading latency.

**Communicate state across switches:** In BFC, each switch marks the packet header with the current queue assignment. The subsequent switch thus knows the queue assignment at the previous hop (upstream queue). If packets coming from an upstream queue cause congestion at the switch, the switch simply signals the upstream switch to throttle the upstream queue. Alternatively, the switch can track all the flows causing congestion and signal the upstream switch the set of flows causing congestion. The upstream switch can then perform lookups to identify and throttle the queues associated with these flows. In contrast, communicating queue assignment across switches allows for a simpler mechanism, the BFC switch only maintains and updates the per-upstream queue state.

We demonstrate BFC's feasibility by implementing it on Tofino2 [72], a state-of-the-art P4-based programmable hardware switch. We evaluate BFC using large-scale ns3 [5] simulations with traces from real workloads. Compared to deployed end-to-end schemes, BFC achieves 2.3 - 60× lower tail latency for short flows and 1.6 - 5× better average completion time for long flows.

### 1.3.3 Nimbus

In the past, researchers have proposed several *delay-controlling* congestion control protocols to reduce delays in the Internet [31, 22, 140]. These schemes reduce their rates as delays increase to limit queuing. Despite the advantages, these protocols haven't seen widespread deployment. Typically, routers in the Internet use FIFO droptail queues that are shared by competing flows. The throughput of delay-controlling schemes is dismal when competing against deployed *buffer-filling* protocols such as Cubic [63] and NewReno [66] that must fill buffers to elicit congestion signals (packet losses or ECN). The reason is that buffer-filling senders steadily increase their rates, causing queuing delays to rise; in response to increasing delays, a competing delay-controlling flow will reduce its rate. The buffer-filling flow then grabs this freed-up bandwidth. The throughput of the delay-controlling flow plummets, but delays don't reduce.

Deploying AQM schemes [112, 111] or isolating flows intro separate queues [130, 103] at the router can also help achieve both low delays and high throughput. However, since such solutions require router modifications, they have not seen much adoption in the Internet. Instead, we ask the following question: Given the current state of the Internet, is it possible to achieve the benefits of delay-controlling protocols without compromising on throughput?

We believe that additional feedback on the *nature* of the cross-traffic competing with a flow can guide congestion control decisions at the sender and help answer this question. The key challenge is to extract this information without any modifications to the router. To this end, we propose a novel algorithm, Nimbus, that uses only end-to-end measurements to rigorously characterize whether the cross-traffic at the bottleneck link is *elastic* or not. Formally, an elastic flow adjusts its rate based on the available bandwidth at the bottleneck link. By definition, any backlogged flow using a congestion control protocol is elastic. By contrast, an inelastic flow's rate is independent of the available bandwidth. Examples include application-limited traffic (e.g., video streams where the available bandwidth exceeds the maximum video bitrate),

short TCP transfers, constant bit rate traffic, etc. The cross-traffic is considered elastic if it contains *any* elastic flows, otherwise, it is inelastic.

Elasticity detection can serve as a building block for low delay congestion control. When Nimbus deems cross-traffic is inelastic, the sender can *safely* use a delay-controlling protocol to reduce delays without worrying about losing throughput. If the the cross-traffic is elastic, the sender can use a standard TCP-competitive protocol like Cubic to compete appropriately against other flows without attempting to reduce delays. Nimbus can support various delay-controlling (e.g., Vegas, Copa) and TCP-competitive protocols (e.g., Cubic, NewReno).

Characterizing elasticity has another advantage: *robustness*. By definition, all elastic cross-traffic flows respond to variations in the available bandwidth regardless of the underlying congestion control protocol. Nimbus leverages this universal property for robust elasticity detection. The key idea is: Modulate the sending rate to create variations in the available bandwidth and observe if the cross-traffic responds to these variations. The detection mechanism is not tied to the specifics of protocols or RTT of the cross-traffic flows and is robust under a variety of network and traffic conditions. In our experiments, Nimbus achieves at least 85% detection accuracy even when cross-traffic is a combination of a varying number of elastic flows and highly-varying inelastic flows, or when cross-traffic is composed of multiple elastic flows with different RTTs and congestion control protocols.

The detection technique is composed of three components.

**1. Estimate the cross-traffic rate:** The sender continuously estimates the aggregate rate of the cross-traffic based on end-to-end measurements of the flow's sending and receiving rate.

**2. Tickle the cross-traffic:** The sender continuously modulates the sending rate with sinusoidal pulses at a fixed frequency (e.g., 5 Hz). These pulses create variations in the available bandwidth at the bottleneck link. An elastic flow will respond to these induced variations by adapting its rate. In contrast, for inelastic cross-traffic, there will be no reaction in the cross-traffic rate.

**3. Monitor cross-traffic response in the frequency domain:** The sender com-

putes a fast fourier transform (FFT) of the estimated cross-traffic rate. If the cross-traffic rate oscillates at the pulsing frequency, then the cross-traffic rate is deemed elastic, otherwise it is classifies as inelastic. An alternative approach might be to see if there is an inverse correlation between the sending rate and the cross-traffic rate. However, this approach is problematic: cross-traffic rate responds to the sending rate variations after an RTT, calculating correlation is brittle as the cross-traffic RTT is unknown and the cross-traffic can contain multiple flows with different RTTs.

Our results on emulated and real-world paths show that congestion control using elasticity detection achieves throughput comparable to Cubic, but with delays that are 50-70 ms lower when cross-traffic is inelastic.

### 1.3.4 Beyond this Dissertation

Many of the ideas presented in the aforementioned works go beyond congestion control in the specific network environment they were originally developed for. These ideas are general and can serve as building blocks for congestion control in other environments, scheduling, network monitoring, etc. For example:

- ABC's insight on computing timely feedback using the dequeue rate also applies to explicit congestion control in datacenter networks [96].
- BFC's dynamic queue assignment can help switches/routers achieve a better approximation of various scheduling policies (e.g., fair queuing) that the network operator wants to enforce.
- Elasticity detection can be used a monitoring tool to shed light on traffic behaviour on Internet paths or aid selective deployment of AQM schemes on routers with elastic traffic to reduce delays.

We also contribute a few related solutions for congestion control that are not included in the main body of this dissertation:

**CCP (Hotnets 2017 [108], SIGCOMM 2018 [109]):** To ease implementation of complex congestion control algorithms at the sender, we propose moving out the

control functions from the datapath and placing them in a separate user space agent which we call the congestion control plane (CCP). The datapath summarizes feedback information about packet RTTs, losses, etc. via a well-defined interface to algorithms running in the off-datapath CCP. The algorithms use this information to control the datapath's congestion window or sending rate. We leverage CCP for Nimbus's Linux TCP implementation which otherwise would have been cumbersome given the lack of support for signal processing libraries in the Linux kernel datapath.

**Annulus (SIGCOMM 2020 [122]):** Annulus is a lightweight congestion control scheme for flows traversing Wide Area Networks (WANs) but with senders located in a datacenter. Such WAN flows share datacenter links with the (intra) datacenter traffic. Typically, the WAN RTT is long and the available bandwidth at the datacenter links varies on a shorter timescale. Traditional congestion control for WAN flows using feedback signals that are delayed by the WAN RTT can lead to congestion and packet drops at the datacenter switches. To solve this, Annulus revisits the idea of dual congestion control loop mentioned earlier (§1.3.1). One control loop tracks the available bandwidth on the WAN links. The other control loops uses direct signals from the datacenter switches (QCN packets [9]) with shorter feedback delay to quickly adapt to variations within the datacenter.

**Bundler (EuroSys 2021 [35]):** Bundler is a new kind of middlebox that is designed for controlling a *bundle* of flows between two Internet sites (e.g., MIT and Harvard). Right now, the bottleneck link typically lies somewhere in the Internet where the network operator has no control. Bundler enforces an aggregate sending rate for the bundle at the sender site's middlebox (*sendbox*). The rate limit moves the bottleneck to the sendbox where the network operator can now enforce various policies (e.g., scheduling, traffic shaping, etc.) to improve performance. To determine the aggregate rate, Bundler uses Nimbus. If the cross-traffic competing with the bundle (at the original bottleneck) is elastic, Bundler does not attempt to enforce any aggregate rate as it can hurt bundle's throughput. Otherwise, Bundler uses a delay-controlling protocol to limit queuing in the network and move the bottleneck to the sendbox.

## 1.4 Previous Papers and Organization

The chapters of this dissertation are based on the following papers:

Chapter 2 revises two papers:

1. Prateesh Goyal, Mohammad Alizadeh, Hari Balakrishnan. Rethinking Congestion Control for Cellular Networks. In Proc. of ACM HotNets, 2017 [58].

2. Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, Hari Balakrishnan. ABC: A Simple Explicit Congestion Controller for Wireless Networks. In Proc. of USENIX NSDI, 2020 [58].

Chapter 3 revises: Prateesh Goyal, Preey Shah, Kevin Zhao, Mohammad Alizadeh, Thomas E. Anderson. Backpressure Flow Control. In Proc. of USENIX NSDI, 2022 [60].

Chapter 4 revises: Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, Hari Balakrishnan. Elasticity Detection: A Building Block for Internet Congestion Control [59].

# Chapter 2

# ABC: A Simple Explicit Congestion Controller for Wireless Networks

## 2.1 Introduction

This chapter presents a new explicit congestion control protocol for network paths with wireless links. Congestion control on such paths is challenging because of the rapid time variations of the link capacity. Explicit control protocols like XCP [81] and RCP [132] can in theory provide superior performance on such paths compared to end-to-end [63, 66, 36, 31, 142, 146, 22, 42] or active queue management (AQM) [111, 112] approaches (§2.2). Unlike these approaches, explicit control protocols enable the wireless router to directly specify a target rate for the sender, signaling both rate decreases and rate increases based on the real-time link capacity.

However, current explicit control protocols have two limitations, one conceptual and the other practical. First, existing explicit protocols were designed for fixed-capacity links; we find that their control algorithms are sub-optimal on time-varying wireless links. Second, they require major changes to packet headers, routers, and endpoints to deploy on the Internet.

Our contribution is a simple and deployable protocol, called Accel-Brake Control (ABC), that overcomes these limitations. In ABC (§2.3), a wireless router marks each packet with one bit of feedback corresponding to either *accelerate* or *brake* based on a

measured estimate of the current link rate. Upon receiving this feedback via an ACK from the receiver, the sender increases its window by one on an accelerate (sends two packets in response to the ACK), and decreases it by one on a brake (does not send any packet). This simple mechanism allows the router to signal a large dynamic range of window size changes within one RTT: from throttling the window to 0, to doubling the window.

Central to ABC's performance is a novel control algorithm that helps routers provide very accurate feedback on time-varying links. Existing explicit schemes like XCP and RCP calculate their feedback by comparing the current *enqueue rate* of packets to the link capacity. An ABC router, however, compares the *dequeue rate* of packets from its queue to the link capacity to mark accelerates or brakes. This change is rooted in the observation that, for an ACK-clocked protocol like ABC, the current dequeue rate of packets at the router provides an accurate prediction of the future incoming rate of packets, one RTT in advance. In particular, if the senders maintain the same window sizes in the next RTT, they will send one packet for each ACK, and the incoming rate in one RTT will be equal to the current dequeue rate. Therefore, rather than looking at the current enqueue rate, the router should signal changes based on the anticipated enqueue rate in one RTT to better match the link capacity. The impact of this subtle change is particularly significant on wireless links, since the enqueue and dequeue rates can differ significantly when the link capacity varies.

ABC also overcomes the deployability challenges of prior explicit schemes, since it can be implemented on top of the existing explicit congestion notification (ECN) [118] infrastructure. We present techniques that enable ABC to co-exist with non-ABC routers, and to share bandwidth fairly with legacy flows traversing a bottleneck ABC router (§2.4).

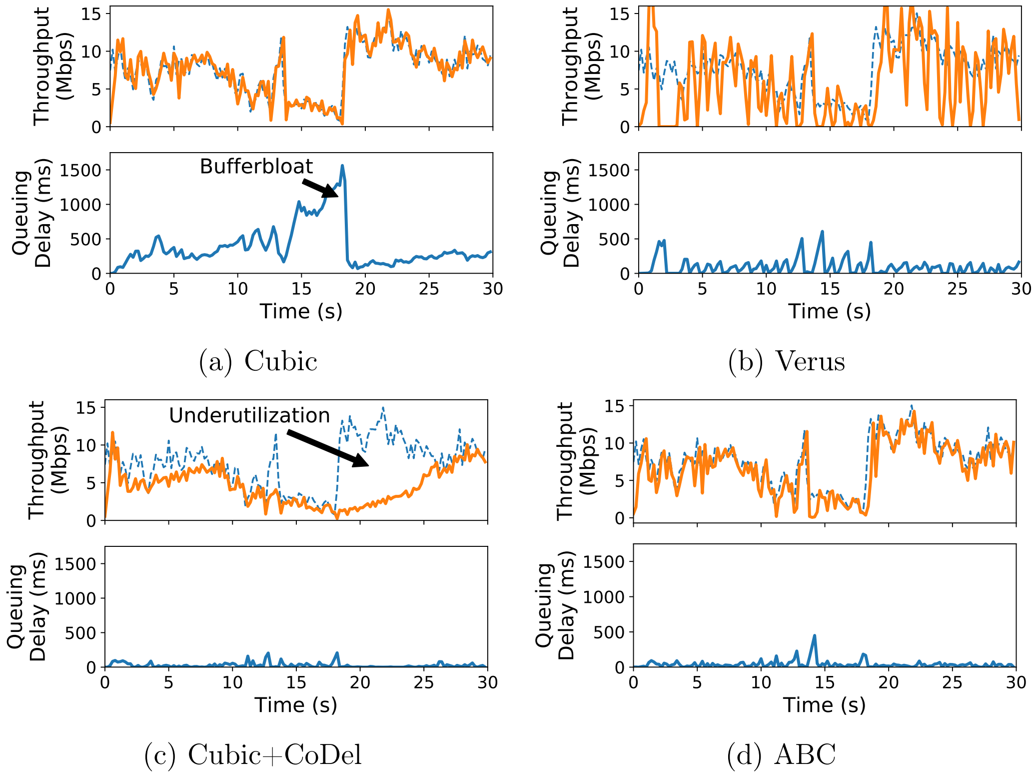
We have implemented ABC on a commodity Wi-Fi router running OpenWrt [46]. Our implementation (§2.5.1) reveals an important challenge for implementing explicit protocols on wireless links: how to determine the link rate for a user at a given time? The task is complicated by the intricacies of the Wi-Fi MAC's batch scheduling and

| Scheme | Norm. Utilization | Norm. Delay (95%) |
|---|---|---|
| ABC | 1 (78%) | 1 (242ms) |
| XCP | 0.97 | 2.04 |
| Cubic+Codel | 0.67 | 0.84 |
| Copa | 0.66 | 0.85 |
| Cubic | 1.18 | 4.78 |
| PCC-Vivace | 1.12 | 4.93 |
| BBR | 0.96 | 2.83 |
| Sprout | 0.55 | 1.08 |
| Verus | 0.72 | 2.01 |

block acknowledgements. We develop a method to estimate the Wi-Fi link rate and demonstrate its accuracy experimentally. For cellular links, the 3GPP standard [1] shows how to estimate the link rate; our evaluation uses emulation with cellular packet traces.

We have experimented with ABC in several wireless network settings. Our results are:

1. In Wi-Fi, compared to Cubic+Codel, Vegas, and Copa, ABC achieves 30-40% higher throughput with similar delays. Cubic, PCC Vivace-latency and BBR incur 70%–6× higher $95^{th}$ percentile packet delay with similar throughput.
2. The results in emulation over 8 cellular traces are summarized below. Despite relying on single-bit feedback, ABC achieves 2× lower $95^{th}$ percentile packet delay compared to XCP.
3. ABC bottlenecks can coexist with both ABC and non-ABC bottlenecks. ABC flows achieve high utilization and low queuing delays if the bottleneck router is ABC, while switching to Cubic when the bottleneck is a non-ABC router.
4. ABC competes fairly with both ABC and non-ABC flows. In scenarios with both ABC and non-ABC flows, the difference in average throughput of ABC and non-ABC flows is under 5%.




(a) Cubic

(b) Verus

(c) Cubic+CoDel

(d) ABC

Figure 2-1: **Performance on a emulated cellular trace** — The dashed blue in the top graph represents link capacity, the solid orange line represents the achieved throughput. Cubic has high utilization but has very high delays (up to 1500 milliseconds). Verus has large rate variations and incurs high delays. Cubic+CoDel reduces queuing delays significantly, but leaves the link underutilized when capacity increases. ABC achieves close to 100% utilization while maintaining low queuing delays (similar to that of Cubic+CoDel).

## 2.2 Motivation

Link rates in wireless networks can vary rapidly with time; for example, within one second, a wireless link's rate can both double and halve [142].[1] These variations make it difficult for transport protocols to achieve both high throughput and low delay. Here, we motivate the need for explicit congestion control protocols that provide feedback to senders on both rate increases and decreases based on direct knowledge of the wireless link rate. We discuss why these protocols can track wireless link rates more accurately than end-to-end and AQM-based schemes. Finally, we discuss deployment challenges for explicit control protocols, and our design goals for a deployable explicit protocol for wireless links.

[1]We define the link rate for a user as the rate that user can achieve if it keeps the bottleneck router backlogged (see §2.5).

**Limitations of end-to-end congestion control:** Traditional end-to-end congestion control schemes like Cubic [63] and NewReno [66] rely on packet drops to infer congestion and adjust their rates. Such schemes tend to fill up the buffer, causing large queuing delays, especially in cellular networks that use deep buffers to avoid packet loss [142]. Fig. 2-1a shows performance of Cubic on an LTE link, emulated using a LTE trace with Mahimahi [110]. The network round-trip time is 100 ms and the buffer size is set to 250 packets. Cubic causes significant queuing delay, particularly when the link capacity drops.

Recent proposals such as BBR [36], PCC-Vivace [42] and Copa [22] use RTT and send/receive rate measurements to estimate the available link rate more accurately. Although these schemes are an improvement over loss-based schemes, their performance is far from optimal on highly-variable links. Our experiments show that they either cause excessive queuing or underutilize the link capacity (e.g., see Fig. 2-7). Sprout [142] and Verus [146] are two other recent end-to-end protocols designed specifically for cellular networks. They also have difficulty tracking the link rate accurately; depending on parameter settings, they can be too aggressive (causing large queues) or too conservative (hurting utilization). For example, Fig. 2-1b shows how Verus performs on the same LTE trace as above.

The fundamental challenge for any end-to-end scheme is that to estimate the link capacity, it must utilize the link fully and build up a queue. When the queue is empty, signals such as the RTT and send/receive rate do not provide information about the available capacity. Therefore, in such periods, all end-to-end schemes must resort to some form of "blind" rate increase. But for networks with a large dynamic range of rates, it is very difficult to tune this rate increase correctly: if it is slow, throughput suffers, but making it too fast causes overshoots and large queuing delays.[2] For schemes that attempt to limit queue buildup, periods in which queues go empty (and a blind rate increase is necessary) are common; they occur, for example, following a sharp increase in link capacity.

[2] BBR attempts to mitigate this problem by periodically increasing its rate in short pulses, but our experiments show that BBR frequently overshoots the link capacity with variable-bandwidth links, causing excessive queuing (see Appendix B.1).

**AQM schemes do not signal increases:** AQM schemes like RED [49], PIE [112] and CoDel [7] can be used to signal congestion (via ECN or drops) before the buffer fills up at the bottleneck link, reducing delays. However, AQM schemes do not signal rate increases. When capacity increases, the sender must again resort to a blind rate increase. Fig. 2-1c shows how CoDel performs when the sender is using Cubic. Cubic+CoDel reduces delays by 1 to 2 orders of magnitude compared to Cubic alone but leaves the link underutilized when capacity increases.

Thus, we conclude that, both end-to-end and AQM-based schemes will find it difficult to track time-varying wireless link rates accurately. Explicit control schemes, such as XCP [81] and RCP [132] provide a compelling alternative. The router provides multiple bits of feedback per packet to senders based on direct knowledge of the wireless link capacity. By telling senders precisely how to increase or decrease their rates, explicit schemes can quickly adapt to time-varying links, in principle, within an RTT of link capacity changes.

**Deployment challenges for explicit congestion control:** Schemes like XCP and RCP require major changes to packet headers, routers, and endpoints. Although the changes are technically feasible, in practice, they create significant deployment challenges. For instance, these protocols require new packet fields to carry multi-bit feedback information. IP or TCP options could in principle be used for these fields. But many wide-area routers drop packets with IP options [50], and using TCP options creates problems due to middleboxes [67] and IPSec encryption [83]. Another important challenge is co-existence with legacy routers and legacy transport protocols. To be deployable, an explicit protocol must handle scenarios where the bottleneck is at a legacy router, or when it shares the link with standard end-to-end protocols like Cubic.

**Design goals:** In designing ABC, we targeted the following properties:

1. *Control algorithm for fast-varying wireless links:* Prior explicit control algorithms like XCP and RCP were designed for fixed-capacity links. We design ABC's control algorithm specifically to handle the rapid bandwidth variations and packet transmission behavior of wireless links (e.g., frame batching at the

MAC layer).

2. *No modifications to packet headers:* ABC repurposes the existing ECN [118] bits to signal both increases and decreases to the sender's congestion window. By spreading feedback over a sequence of 1-bit signals per packet, ABC routers precisely control sender congestion windows over a large dynamic range.
3. *Coexistence with legacy bottleneck routers:* ABC is robust to scenarios where the bottleneck link is not the wireless link but a non-ABC link elsewhere on the path. Whenever a non-ABC router becomes the bottleneck, ABC senders ignore window increase feedback from the wireless link, and ensure that they send no faster than their fair share of the bottleneck link.
4. *Coexistence with legacy transport protocols:* ABC routers ensure that ABC and non-ABC flows share a wireless bottleneck link fairly. To this end, ABC routers separate ABC and non-ABC flows into two queues, and use a simple algorithm to schedule packets from these queues. ABC makes no assumptions about the congestion control algorithm of non-ABC flows, is robust to the presence of short or application-limited flows, and requires a small amount of state at the router.

Fig. 2-1d shows ABC on the same emulated LTE link. Using only one bit of feedback per packet, the ABC flow is able to track the variations in bottleneck link closely, achieving both high throughput and low queuing delay.

## 2.3 Design

ABC is a window-based protocol: the sender limits the number of packets in flight to the current congestion window. Window-based protocols react faster to the sudden onset of congestion than rate-based schemes [24]. On a wireless link, when the capacity drops and the sender stops receiving ACKs, ABC will stop sending packets immediately, avoiding further queue buildup. In contrast, a rate-based protocol would take time to reduce its rate and may queue up a large number of packets at the bottleneck link in the meantime.

ABC senders adjust their window size based on explicit feedback from ABC routers. An ABC router uses its current estimate of the link rate and the queuing delay to compute a *target rate*. The router then sets one bit of feedback in each packet to guide the senders towards the target rate. Each bit is echoed to a sender by a receiver in an ACK, and it signals either a one-packet increase ("accelerate") or a one-packet decrease ("brake") to the sender's congestion window.

## 2.3.1 The ABC Protocol

We now present ABC's design starting with the case where all routers are ABC-capable and all flows use ABC. We later discuss how to extend the design to handle non-ABC routers and scenarios with competing non-ABC flows.

### 2.3.1.1 ABC Sender

On receiving an "accelerate" ACK, an ABC sender increases its congestion window by 1 packet. This increase results in two packets being sent, one in response to the ACK and one due to the window increase. On receiving a "brake," the sender reduces its congestion window by 1 packet, preventing the sender from transmitting a new packet in response to the received ACK. As we discuss in §2.3.1.3, the sender also performs an additive increase of 1 packet per RTT to achieve fairness. For ease of exposition, let us ignore this additive increase for now.

Though each bit of feedback translates to only a small change in the congestion window, when aggregated over an RTT, the feedback can express a large dynamic range of window size adjustments. For example, suppose a sender's window size is $w$, and the router marks accelerates on a fraction $f$ of packets in that window. Over the next RTT, the sender will receive $w \cdot f$ accelerates and $w - w \cdot f$ brakes. Then, the sender's window size one RTT later will be $w + wf - (w - wf) = 2wf$ packets. Thus, in one RTT, an ABC router can vary the sender's window size between zero ($f = 0$) and double its current value ($f = 1$). The set of achievable window changes for the next RTT depends on the number of packets in the current window $w$; the larger $w$,

the higher the granularity of control.

In practice, ABC senders increase or decrease their congestion window by the number of newly acknowledged bytes covered by each ACK. Byte-based congestion window modification is a standard technique in many TCP implementations [17], and it makes ABC robust to variable packet sizes and delayed, lost, and partial ACKs. For simplicity, we describe the design with packet-based window modifications.

#### 2.3.1.2 ABC Router

**Calculating the target rate:** ABC routers compute the target rate $tr(t)$ using the following rule:

$$tr(t) = \eta\mu(t) - \frac{\mu(t)}{\delta}(x(t) - d_t)^+, \quad (2.1)$$

where $\mu(t)$ is the link capacity, $x(t)$ is the observed queuing delay, $d_t$ is a pre-configured delay threshold, $\eta$ is a constant less than 1, $\delta$ is a positive constant (in units of time), and $y^+$ is $\max(y, 0)$. This rule has the following interpretation. When queuing delay is low ($x(t) < d_t$), ABC sets the target rate to $\eta\mu(t)$, for a value of $\eta$ slightly less than 1 (e.g., $\eta = 0.95$). By setting the target rate a little lower than the link capacity, ABC aims to trade a small amount of bandwidth for large reductions in delay, similar to prior work [77, 15, 92]. However, queues can still develop due to rapidly changing link capacity and the 1 RTT of delay it takes for senders to achieve the target rate. ABC uses the second term in Equation (2.1) to drain queues. Whenever $x(t) > d_t$, this term reduces the target rate by an amount that causes the queuing delay to decrease to $d_t$ in at most $\delta$ seconds.

The threshold $d_t$ ensures that the target rate does not react to small increases in queuing delay. This is important because wireless links often schedule packets in batches. Queuing delay caused by batch packet scheduling does not imply congestion, even though it occurs persistently. To prevent target rate reductions due to this delay, $d_t$ must be configured to be greater than the average inter-scheduling time at the router.

ABC's target rate calculation requires an estimate of the underlying link capacity, $\mu(t)$. In §2.5, we discuss how to estimate the link capacity in cellular and WiFi

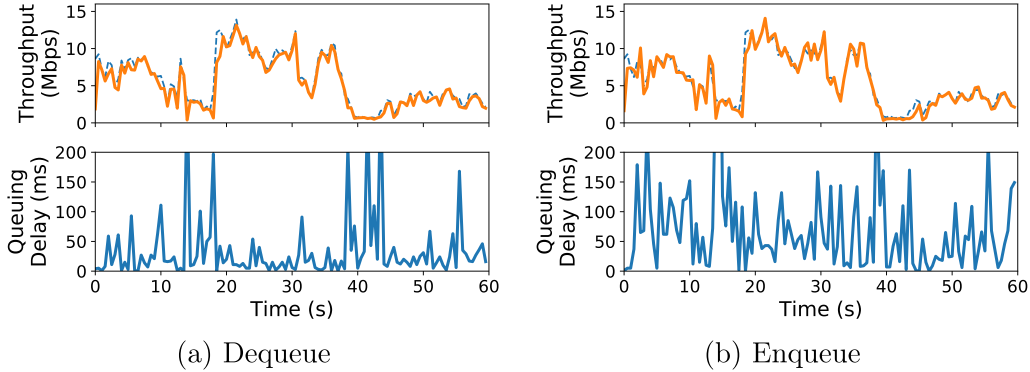



Figure 2-2: **Feedback** — Calculating $f(t)$ based on enqueue rate increases $95^{th}$ percentile queuing delay by 2×.

networks, and we present an implementation for WiFi.

**Packet marking:** To achieve a target rate, $tr(t)$, the router computes the fraction of packets, $f(t)$, that should be marked as accelerate. Assume that the current *dequeue rate*—the rate at which the router transmits packets—is $cr(t)$. If the accelerate fraction is $f(t)$, for each packet that is ACKed, the sender transmits $2f(t)$ packets on average. Therefore, after 1 RTT, the *enqueue rate*—the rate at which packets arrive to the router—will be $2cr(t)f(t)$. To achieve the target rate, $f(t)$ must be chosen such that $2cr(t)f(t)$ is equal to $tr(t)$. Thus, $f(t)$ is given by:

$$f(t) = \min\left\{\frac{1}{2}\cdot\frac{tr(t)}{cr(t)}, 1\right\}. \tag{2.2}$$

An important consequence of the above calculation is that $f(t)$ is computed based on the *dequeue* rate. Most explicit protocols compare the enqueue rate to the link capacity to determine the feedback (e.g., see XCP [81]).

ABC uses the dequeue rate instead to exploit the ACK-clocking property of its window-based protocol. Specifically, Equation (2.2) accounts for the fact that when the link capacity changes (and hence the dequeue rate changes), the rate at the senders changes automatically within 1 RTT because of ACK clocking. Fig. 2-2 demonstrates that computing $f(t)$ based on the dequeue rate at the router enables ABC to track the link capacity much more accurately than using the enqueue rate.

ABC recomputes $f(t)$ on every dequeued packet, using measurements of $cr(t)$ and $\mu(t)$ over a sliding time window of length $T$. Updating the feedback on every packet allows ABC to react to link capacity changes more quickly than schemes that use

periodic feedback updates (e.g., XCP and RCP).

Packet marking can be done deterministically or probabilistically. To limit burstiness, ABC uses the deterministic method in Algorithm 1. The variable `token` implements a token bucket that is incremented by $f(t)$ on each outgoing packet (up to a maximum value `tokenLimit`), and decremented when a packet is marked accelerate. To mark a packet accelerate, `token` must exceed 1. This simple method ensures that no more than a fraction $f(t)$ of the packets are marked accelerate.

```
token = 0;
for each outgoing packet do
    calculate f(t) using Equation (2.2);
    token = min(token +f(t), tokenLimit);
    if packet marked with accelerate then
        if token > 1 then
            token = token − 1;
            mark accelerate;
        else
            mark brake;
```

**Algorithm 1:** Packet marking at an ABC router.

**Multiple bottlenecks:** An ABC flow may encounter multiple ABC routers on its path. An example of such a scenario is when two smartphone users communicate over an ABC-compliant cellular network. Traffic sent from one user to the other will traverse a cellular uplink and cellular downlink, both of which could be the bottleneck. To support such situations, an ABC sender should send traffic at the smallest of the router-computed target rates along their path. To achieve this goal, each packet is initially marked accelerate by the sender. ABC routers may change a packet marked accelerate to a brake, but not vice versa (see Algorithm 1). This rule guarantees that an ABC router can unilaterally reduce the fraction of packets marked accelerate to ensure that its target rate is not exceeded, but it cannot increase this fraction. Hence the fraction of packets marked accelerate will equal the minimum $f(t)$ along the path.

#### 2.3.1.3 Fairness

Multiple ABC flows sharing the same bottleneck link should be able to compete fairly with one another. However, the basic window update rule described in §2.3.1.1 is

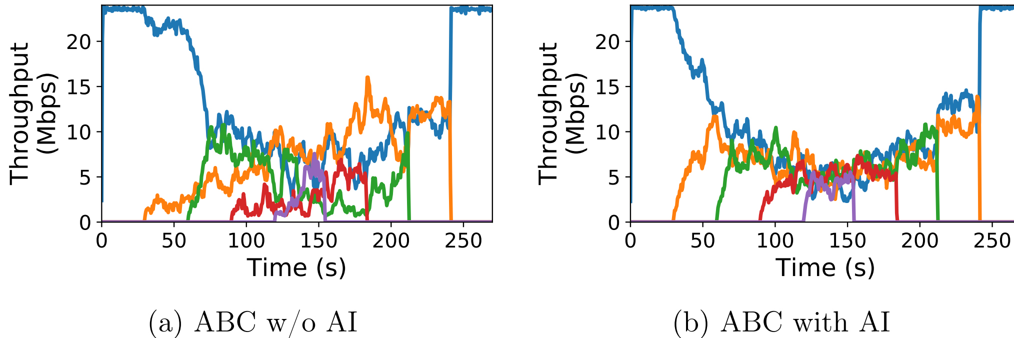
a multiplicative-increase/multiplicative-decrease (MIMD) strategy,[3] which does not provide fairness among contending flows (see Fig. 2-3a for an illustration). To achieve fairness, we add an additive-increase (AI) component to the basic window update rule. Specifically, ABC senders adjust their congestion window on each ACK as follows:

$$w \leftarrow \begin{cases} w + 1 + 1/w & \text{if accelerate} \\ w - 1 + 1/w & \text{if brake} \end{cases} \tag{2.3}$$

This rule increases the congestion window by 1 packet each RTT, in addition to reacting to received accelerate and brake ACKs. This additive increase, coupled with ABC's MIMD response, makes ABC a multiplicative-and-additive-increase/multiplicative-decrease (MAIMD) scheme. Chiu and Jain [37] proved that MAIMD schemes converge to fairness (see also [12]). Fig. 2-3b shows how with an AI component, competing ABC flows achieve fairness.

To give intuition, we provide a simple informal argument for why including additive increase gives ABC fairness. Consider $N$ ABC flows sharing a link, and suppose that in steady state, the router marks a fraction $f$ of the packets accelerate, and the window size of flow $i$ is $w_i$. To be in steady state, each flow must send 1 packet on average for each ACK that it receives. Now consider flow $i$. It will send $2f + 1/w_i$ packets on average for each ACK: $2f$ for the two packets it sends on an accelerate (with probability $f$), and $1/w_i$ for the extra packet it sends every $w_i$ ACKs. Therefore, to be in steady state, we must have: $2f + 1/w_i = 1 \implies w_i = 1/(1 - 2f)$. This shows that the steady-state window size for all flows must be the same, since they all observe the same fraction $f$ of accelerates. Hence, with equal RTTs, the flows will have the same throughput, and otherwise their throughput will be inversely proportional to their RTT. Note that the RTT unfairness in ABC is similar to that of schemes like Cubic, for which the throughput of a flow is inversely proportional to its RTT. In §2.7.5, we show experiments where flows have different RTTs.

[3] All the competing ABC senders will observe the same accelerate fraction, $f$, on average. Therefore, each flow will update its congestion window, $w$, in a multiplicative manner, to $2fw$, in the next RTT.




(a) ABC w/o AI (b) ABC with AI

Figure 2-3: **Fairness among competing ABC flows** — 5 flows with the same RTT start and depart one-by-one on a 24 Mbit/s link. The additive-increase (AI) component leads to fairness.

#### 2.3.1.4 Stability Analysis

ABC's stability depends on the values of $\eta$ and $\delta$. $\eta$ determines the target link utilization, while $\delta$ controls how long it will take for a queue to drain. In Appendix A, we prove the following result for a fluid model of the ABC control loop.

**Theorem 1.** *Consider a single ABC link, traversed by $N$ ABC flows. Let $\tau$ be the maximum round-trip propagation delay of the flows. ABC is* globally asymptotically stable *if*

$$\delta > \frac{2}{3} \cdot \tau. \tag{2.4}$$

*Specifically, if $\mu(t) = \mu$ for $t > t_0$ (i.e., the link capacity stops changing after some time $t_0$), the enqueue/dequeue rate and the queuing delay at the ABC router will converge to certain values $r^*$ and $x^*$ that depend on the system parameters and the number of flows. In all cases: $\eta\mu < r^* \leq \mu$.*

This stability criterion is simple and intuitive. It states that $\delta$ should not be much smaller than the RTT (i.e, the feedback delay). If $\delta$ is very small, ABC reacts too forcefully to queue build up, causing under-utilization and oscillations.[4] Increasing $\delta$ well beyond $2/3\tau$ improves the stability margins of the feedback loop, but hurts responsiveness. In our experiments, we used $\delta = 133$ ms for a propagation RTT of 100 ms.

[4]Interestingly, if the sources do not perform additive increase or if the additive increase is sufficiently "gentle," ABC is stable for any value of $\delta$. See the proof in Appendix A for details.

## 2.4 Coexistence

An ABC flow should be robust to presence of non-ABC bottlenecks on its path and share resources fairly with non-ABC flows sharing the ABC router.

### 2.4.1 Deployment with non-ABC Routers

An ABC flow can encounter both ABC and non-ABC routers on its path. For example, a Wi-Fi user's traffic may traverse both a Wi-Fi router (running ABC) and an ISP router (not running ABC); either router could be the bottleneck at any given time. ABC flows must therefore be able to detect and react to traditional congestion signals—both drops and ECN—and they must determine when to ignore accelerate feedback from ABC routers because the bottleneck is at a non-ABC router.

We augment the ABC sender to maintain two congestion windows, one for tracking the available rate on ABC routers ($w_{\text{abc}}$), and one for tracking the rate on non-ABC bottlenecks ($w_{\text{nonabc}}$). $w_{\text{abc}}$ obeys accelerates/brakes using Equation (2.3), while $w_{\text{nonabc}}$ follows a rule such as Cubic [63] and responds to drop and ECN signals.[5] An ABC sender must send packets to match the lower of the two windows.Our implementation mimics Cubic for the non-ABC method, but other methods could also be emulated.

With this approach, the window that is not the bottleneck could become large. For example, when a non-ABC router is the bottleneck, the ABC router will continually send accelerate signals, causing $w_{\text{abc}}$ to grow. If the ABC router later becomes the bottleneck, it will temporarily incur large queues. To prevent this problem, ABC senders cap both $w_{\text{abc}}$ and $w_{\text{nonabc}}$ to $2\times$ the number of in-flight packets.

Fig. 2-4 shows the throughput and queuing delay for an ABC flow traversing a path with an ABC-capable wireless link and a wired link with a droptail queue. For illustration, we vary the rate of the wireless link in a series of steps every 5 seconds. Over the experiment, the bottleneck switches between the wired and wireless links several times. ABC is able to adapt its behavior quickly and accurately. Depending on

[5] We discuss how ABC senders distinguish between accelerate/brake and ECN marks in §2.4.2.

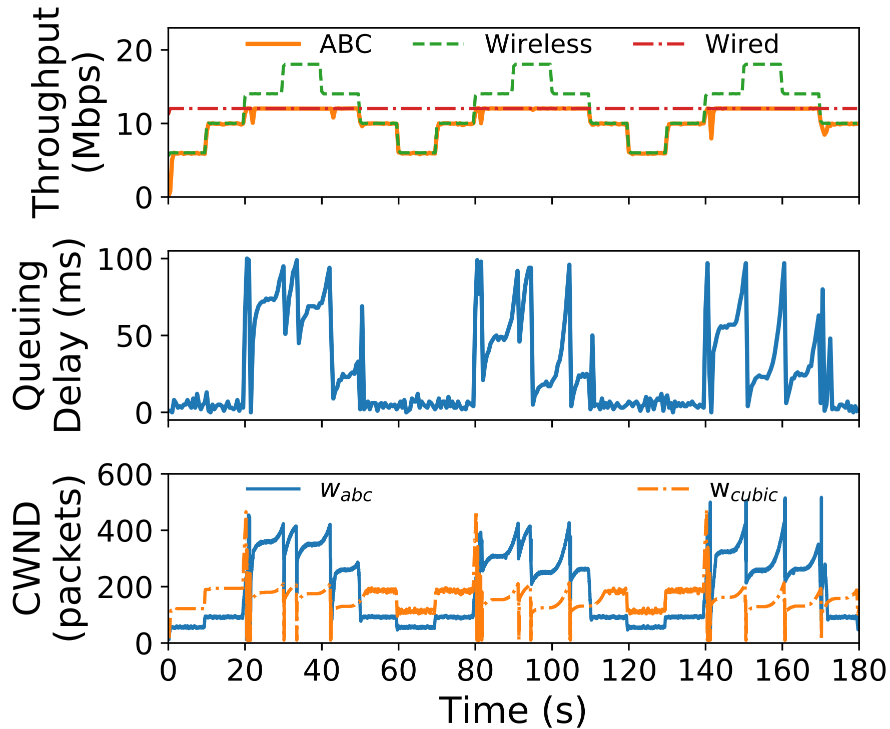



Figure 2-4: **Coexistence with non-ABC bottlenecks** — When the wired link is the bottleneck, ABC becomes limited by $w_{\text{cubic}}$ and behaves like a Cubic flow. When the wireless link is the bottleneck, ABC uses $w_{abc}$ to achieve low delays and high utilization.

which link is the bottleneck, either $w_{\text{nonabc}}$ (i.e., $w_{\text{cubic}}$) or $w_{\text{abc}}$ becomes smaller and controls the rate of the flow. When the wireless link is the bottleneck, ABC maintains low queuing delay, whereas the queuing delay exhibits standard Cubic behavior when the wired link is the bottleneck. $w_{\text{cubic}}$ does not limit ABC's ability to increase its rate when the wireless link is the bottleneck. At these times (e.g., around the 70 s mark), as soon on $w_{\text{abc}}$ increases, the number of in-flight packets and the cap on $w_{\text{cubic}}$ increases, and $w_{\text{cubic}}$ rises immediately.

### 2.4.2 Multiplexing with ECN Bits

IP packets have two ECN-related bits: ECT and CE. These two bits are traditionally interpreted as follows:

| ECT | CE | Interpretation |
|---|---|---|
| 0 | 0 | Non-ECN-Capable Transport |
| 0 | 1 | ECN-Capable Transport ECT(1) |
| 1 | 0 | ECN-Capable Transport ECT(0) |
| 1 | 1 | ECN set |

Routers interpret both 01 and 10 to indicate that a flow is ECN-capable, and routers change those bits to 11 to mark a packet with ECN. Upon receiving an ECN mark (11), the receiver sets the *ECN Echo (ECE)* flag to signal congestion to the sender. ABC reinterprets the ECT and CE bits as follows:

| ECT | CE | Interpretation |
|---|---|---|
| 0 | 0 | Non-ECN-Capable Transport |
| 0 | 1 | **Accelerate** |
| 1 | 0 | **Brake** |
| 1 | 1 | ECN set |

ABC send all packets with accelerate (01) set, and ABC routers signal brakes by flipping the bits to 10. Both 01 and 10 indicate an ECN-capable transport to ECN-capable legacy routers, which will continue to use (11) to signal congestion.

With this design, receivers must be able to echo both standard ECN signals and accelerates/brakes for ABC. Traditional ECN feedback is signaled using the ECE flag. For ABC feedback, we repurpose the NS (nonce sum) bit, which was originally proposed to ensure ECN feedback integrity [45] but has been reclassified as historic [89] due to lack of deployment. Thus, it appears possible to deploy ABC with only simple modifications to TCP receivers.

**Deployment in proxied networks:** Cellular networks commonly split TCP connections and deploy proxies at the edge [137, 120]. Here, it is unlikely that any non-ABC router will be the bottleneck and interfere with the accel-brake markings from the ABC router. In this case, deploying ABC may not require any modifications to today's TCP ECN receiver. ABC senders (running on the proxy) can use either 10 or 01 to signal an accelerate, and routers can use 11 to indicate a brake. The TCP receiver can echo this feedback using the ECE flag.

### 2.4.3 Non-ABC Flows at an ABC Router

ABC flows are potentially at a disadvantage when they share an ABC bottleneck link with non-ABC flows.[6] If the non-ABC flows fill up queues and increase queuing delay, the ABC router will reduce ABC's target rate. To ensure fairness in such scenarios, ABC routers isolate ABC and non-ABC packets in separate queues.

We assume that ABC routers can determine whether a packet belongs to an ABC flow. In some deployment scenarios, this is relatively straightforward. For example, in a cellular network deployment with TCP proxies at the edge of the network [137, 120], the operator can deploy ABC at the proxy, and configure the base station to assume that all traffic from the proxy's IP address uses ABC. Other deployment scenarios may require ABC senders to set a predefined value in a packet field like the IPv6 flow label or the IPv4 IPID.

The ABC router assigns weights to the ABC and non-ABC queues, respectively, and it schedules packets from the queues in proportion to their weights. In addition, ABC's target rate calculation considers only ABC's share of the link capacity (which is governed by the weights). The challenge is to set the weights to ensure that the average throughput of long-running ABC and non-ABC flows is the same, no matter how many flows there are.

Prior explicit control schemes address this problem using the TCP loss-rate equation (XCP) or by estimating the number of flows with Zombie Lists (RCP). Relying on the TCP equation requires a sufficient loss rate and does not handle flows like BBR. RCP's approach does not handle short flows. When one queue has a large number of short flows (and hence a low average throughput), RCP increases the weight of that queue. However, the short flows cannot send faster, so the extra bandwidth is taken by long-running flows in the same queue, which get more throughput than long-running flows in the other queue (see §2.7.5 for experimental results).

To overcome these drawbacks, a ABC router measures the average rate of the $K$ largest flows in each queue using the Space Saving Algorithm [104], which requires

[6] ABC and non-ABC flows may also share a non-ABC link, but in such cases, ABC flows will behave like Cubic and compete fairly with other traffic.

$\mathcal{O}(K)$ space. It considers any remaining flow in either queue to be short, and it calculates the total rate of the short flows in each queue by subtracting the rate of the largest $K$ flows from the queue's aggregate throughput. ABC uses these rate measurements to estimate the rate demands of the flows. Using these demands, ABC periodically computes the max-min fair rate allocation for the flows, and it sets the weight of each of the two queues to be equal to the total max-min rate allocation of its component flows. This algorithm ensures that long-running flows in the two queues achieve the same average rate, while accounting for demand-limited short flows.

To estimate the demand of the flows, the ABC router assumes that the demand for the top $K$ flows in each queue is X% higher than the current throughput of the flow, and the aggregate demand for the short flows is the same as their throughput. If a top-$K$ flow is unable to increase its sending rate by X%, its queue's weight will be larger than needed, but any unfairness in weight assignment is bounded by X%. Small values of $X$ limit unfairness but can slow down convergence to fairness; our experiments use $X = 10\%$.

## 2.5 Estimating Link Rate

We describe how ABC routers can estimate the link capacity for computing the target rate (§2.3.1.2). We present a technique for Wi-Fi that leverages the inner workings of the Wi-Fi MAC layer, and we discuss options for cellular networks.

### 2.5.1 Wi-Fi

We describe how an 802.11n access point (AP) can estimate the average link rate. For simplicity, we first describe our solution when there is a single user (client) connected to the AP. Next, we describe the multi-user case.

We define *link rate* as the potential throughput of the user (i.e., the MAC address of the Wi-Fi client) if it was backlogged at the AP, i.e., if the user never ran out of packets at the AP. In case the router queue goes empty at the AP, the achieved throughput will be less than the link rate.

**Challenges:** A strawman would be to estimate the link rate using the physical layer bit rate selected for each transmission, which would depend on the modulation and channel code used for the transmission. Unfortunately, this method will overestimate the link rate as the packet transmission times are governed not only by the bitrate, but also by delays for additional tasks (e.g., channel contention and retransmissions [28]). An alternative approach would be to use the fraction of time that the router queue was backlogged as a proxy for link utilization. However, the Wi-Fi MAC's packet batching confounds this approach. Wi-Fi routers transmit packets (frames) in batches; a new batch is transmitted only after receiving an ACK for the last batch. The AP may accumulate packets while waiting for a link-layer ACK; this queue buildup does not necessarily imply that the link is fully utilized. Thus, accurately measuring the link rate requires a detailed consideration of Wi-Fi's packet transmission protocols.

**Understanding batching:** In 802.11n, data frames, also known as MAC Protocol Data Units (MPDUs), are transmitted in batches called A-MPDUs (Aggregated MPDUs). The maximum number of frames that can be included in a single batch, $M$, is negotiated by the receiver and the router. When the user is not backlogged, the router might not have enough data to send a full-sized batch of $M$ frames, but will instead use a smaller batch of size $b < M$. Upon receiving a batch, the receiver responds with a single Block ACK. Thus, at a time $t$, given a batch size of $b$ frames, a frame size of $S$ bits,[7] and an ACK inter-arrival time (i.e., the time between receptions of consecutive block ACKs) of $T_{IA}(b, t)$, the current dequeue rate, $cr(t)$, may be estimated as

$$cr(t) = \frac{b \cdot S}{T_{IA}(b, t)}. \tag{2.5}$$

When the user is backlogged and $b = M$, then $cr(t)$ above will be equal to the link capacity. However, if the user is not backlogged and $b < M$, how can the AP estimate the link capacity? Our approach calculates $\hat{T}_{IA}(M, t)$, the estimated ACK inter-arrival time *if the user was backlogged and had sent $M$ frames in the last batch.*

We estimate the link capacity, $\hat{\mu}(t)$, as

[7] For simplicity, we assume that all frames are of the same size, though our formulas can be generalized easily for varying frame sizes.

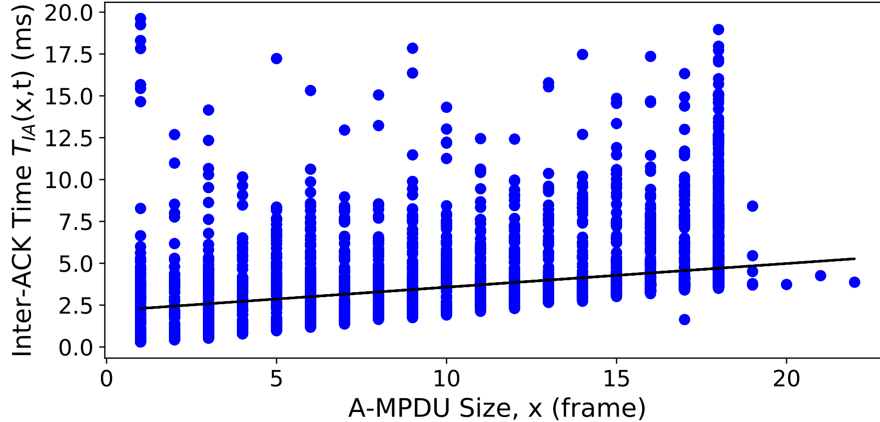



Figure 2-5: **Inter-ACK time v. batch (A-MPDU) size** — Inter-ACK times for a given batch size exhibits variation. The solid black line represents the average Inter-ACK time. The slope of the line is $S/R$, where $S$ is the frame size in bits and $R$ is the link rate in bits per second.

$$\hat{\mu}(t) = \frac{M \cdot S}{\hat{T}_{IA}(M,t)}. \tag{2.6}$$

To accurately estimate $\hat{T}_{IA}(M,t)$, we turn to the relationship between the batch size and ACK inter-arrival time. We can decompose the ACK interval time into the batch transmission time and "overhead" time, the latter including physically receiving an ACK, contending for the shared channel, and transmitting the physical layer preamble [55]. Each of these overhead components is independent of the batch size. We denote the overhead time by $h(t)$. If $R$ is the bitrate used for transmission, the router's ACK inter-arrival time is

$$T_{IA}(b,t) \;=\; \frac{b \cdot S}{R} + h(t). \tag{2.7}$$

Fig. 2-5 illustrates this relationship empirically. There are two key properties to note. First, for a given batch size, the ACK inter-arrival times vary due to overhead tasks. Second, because the overhead time and batch size are independent, connecting the average values of ACK inter-arrival times across all considered batch sizes will produce a line with slope $S/R$. Using this property along with Equation (2.7), we can estimate the ACK inter-arrival time for a backlogged user as

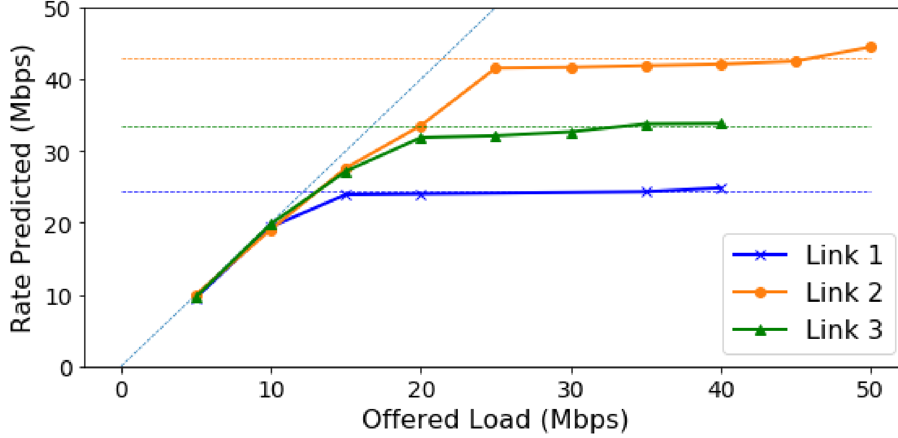



Figure 2-6: **Wi-Fi Link Rate Prediction** — ABC router link rate predictions for a user that was not backlogged and sent traffic at multiple different rates over three different Wi-Fi links. Horizontal lines represent the true link capacity, solid lines summarize the ABC router's link capacity prediction (each point is an average over 30 seconds of predictions), and the dashed slanted line represents the prediction rate caps. ABC's link rate predictions are within 5% of the ground truth across most sending rates (given the prediction cap).

$$\begin{aligned} \hat{T_{IA}}(M,t) &= \frac{M \cdot S}{R} + h(t) \\ &= T_{IA}(b,t) + \frac{(M-b)\cdot S}{R}. \end{aligned} \quad (2.8)$$

We can then use $\hat{T}_{IA}(M,t)$ to estimate the link capacity with Equation (2.6). This computation is performed for each batch transmission when the batch ACK arrives, and passed through a weighted moving average filter over a sliding window of time $T$ to estimate the smoothed time-varying link rate. $T$ must be greater than the inter-ACK time (up to 20 ms in Fig. 2-5); we use $T = 40$ ms. Because ABC cannot exceed a rate-doubling per RTT, we cap the predicted link rate to double the current rate (dashed slanted line in Fig. 2-5).

To evaluate the accuracy of our link rate estimates, we transmit data to a single client through our modified ABC router (§2.7.1) at multiple different rates over three Wi-Fi links (with different modulation and coding schemes). Fig. 2-6 summarizes the accuracy of the ABC router's link rate estimates. With this method, the ABC Wi-Fi router is able to predict link rates within 5% of the true link capacities.

**Extension to multiple users.** In multi-user scenarios, each receiver will negotiate

its own maximum batch size ($M$) with the router, and different users can have different transmission rates. We now present two variants of our technique for (1) when the router uses per-user queues to schedule packets of different users, and (2) when the users share a single FIFO (first-in first-out) queue at the router.

*Per-user queues.* In this case each user calculates a separate link rate estimate. Recall that the link rate for a given user is defined as the potential throughput of the user if it was backlogged at the router. To determine the link rate for a user $x$, we repeat the single-user method for the packets and queue of user $x$ alone, treating transmissions from other users as overhead time. Specifically, user $x$ uses Equations (2.8) and (2.6) to compute its link rate ($\hat{\mu}_x(t)$) based on its own values of the bit rate ($R_x$) and maximum batch size ($M_x$). It also computes its current dequeue rate ($cr_x(t)$) using Equation (2.5) to calculate the accel-brake feedback. The inter-ACK time ($T_{IA_x}(b, t)$), is defined as the time between the reception of consecutive block-ACKs for user $x$. Thus, the overhead time ($h_x(t)$) includes the time when other users at the same AP are scheduled to send packets. Fairness among different users is ensured via scheduling users out of separate queues.

*Single queue.* In this case the router calculates a single aggregate link rate estimate. The inter-ACK time here is the time between two consecutive block-ACKs, regardless of the user to which the block-ACKs belong to. The router tries to match the aggregate rate of the senders to the aggregate link rate, and uses the aggregate current dequeue rate to calculate accel-brake feedback.

### 2.5.2 Cellular Networks

Cellular networks schedule users from separate queues to ensure inter-user fairness. Each user will observe a different link rate and queuing delay. As a result, every user requires a separate target rate calculation at the ABC router. The 3GPP cellular standard [1] describes how scheduling information at the cellular base station can be used to calculate per-user link rates. This method is able to estimate capacity even if a given user is not backlogged at the base station, a key property for the target rate estimation in Equation (2.1).

## 2.6 Discussion

We discuss practical issues pertaining to ABC's deployment.

**Delayed Acks:** To support delayed ACKs, ABC uses byte counting at the sender; the sender increases/decreases its window by the new bytes ACKed. At the receiver, ABC uses the state machine from DCTCP [14] for generating ACKs and echoing accel/brake marks. The receiver maintains the state of the last packet (accel or brake). Whenever the state changes, the receiver sends an ACK with the new state. If the receiver is in the same state after receiving $m$ packets (the number of ACKs to coalesce), then it sends a delayed ACK with the current state. Our TCP implementation and the experiments in §2.7 use delayed ACKs with $m = 2$.

**Lost ACKs:** ABC's window adjustment is robust to ACK losses. Consider a situation where the sender receives a fraction $p < 1$ of the ACKs. If the accelerate fraction at the router is $f$, the current window of the sender is $w_{abc}$, then in the next RTT, the change in congestion window of the sender is $fpw_{abc} - (1 - f)pw_{abc} = (2f - 1)pw_{abc}$. As a result, lost ACKs only slow down the changes in the congestion window, but whether it increases or decreases doesn't depend on $p$.

**ABC routers don't change prior ECN marks:** ABC routers don't mark accel-brake on incoming packets that contain ECN marks set by an upstream non-ABC router. Since packets with ECN set can't convey accel-brake marks, they can slow down changes in $w_{abc}$ (similar to lost ACKs). In case the fraction of packets with ECN set is small, then, the slow down in changes to $w_{abc}$ will be small. If the fraction is large, then the non-ABC router is the likely bottleneck, and the sender will not use $w_{abc}$.

**ECN routers clobbering ABC marks:** An ECN router can overwrite accel-brake marks. The ABC sender will still track the non-ABC window, $w_{nonabc}$, but such marks can slow down adjustment to the ABC window, $w_{abc}$.

**ABC on fixed-rate links:** ABC can also be deployed on fixed-rate links. On such links, its performance is similar to prior explicit schemes like XCP.

## 2.7 Evaluation

We evaluate ABC by considering the following properties:

1. **Performance:** We measure ABC's ability to achieve low delay and high throughput and compare ABC to end-to-end schemes, AQM schemes, and explicit control schemes (§2.7.3).
2. **Multiple bottlenecks:** We test ABC in scenarios with multiple ABC bottlenecks and mixtures of ABC and non-ABC bottlenecks (§2.7.4).
3. **Fairness:** We evaluate ABC's fairness while competing against other ABC and non-ABC flows (§2.7.5).
4. **Additional considerations:** We evaluate how ABC performs with application-limited flows and different network delays. We also demonstrate ABC's impact on a real application's performance (§2.7.6).

### 2.7.1 Prototype ABC Implementation

**ABC transport:** We implemented ABC endpoints in Linux as kernel modules using the pluggable TCP API.

**ABC router:** We implemented ABC as a Linux queuing discipline (qdisc) kernel module using OpenWrt, an open source operating system for embedded networked devices [46]. We used a NETGEAR WNDR 3800 router configured to 802.11n. We note that our implementation is portable as OpenWrt is supported on many other commodity Wi-Fi routers.

ABC's WiFi link rate estimation exploits the inner workings of the MAC 802.11n protocol, and thus requires fine-grained values at this layer. In particular, the ABC qdisc must know A-MPDU sizes, Block ACK receive times, and packet transmission bitrates. These values are not natively exposed to Linux router qdiscs, and instead are only available at the network driver. To bridge this gap, we modified the router to log the relevant MAC layer data in the cross-layer socket buffer data structure (skb) that it already maintains per packet.

### 2.7.2 Experimental Setup

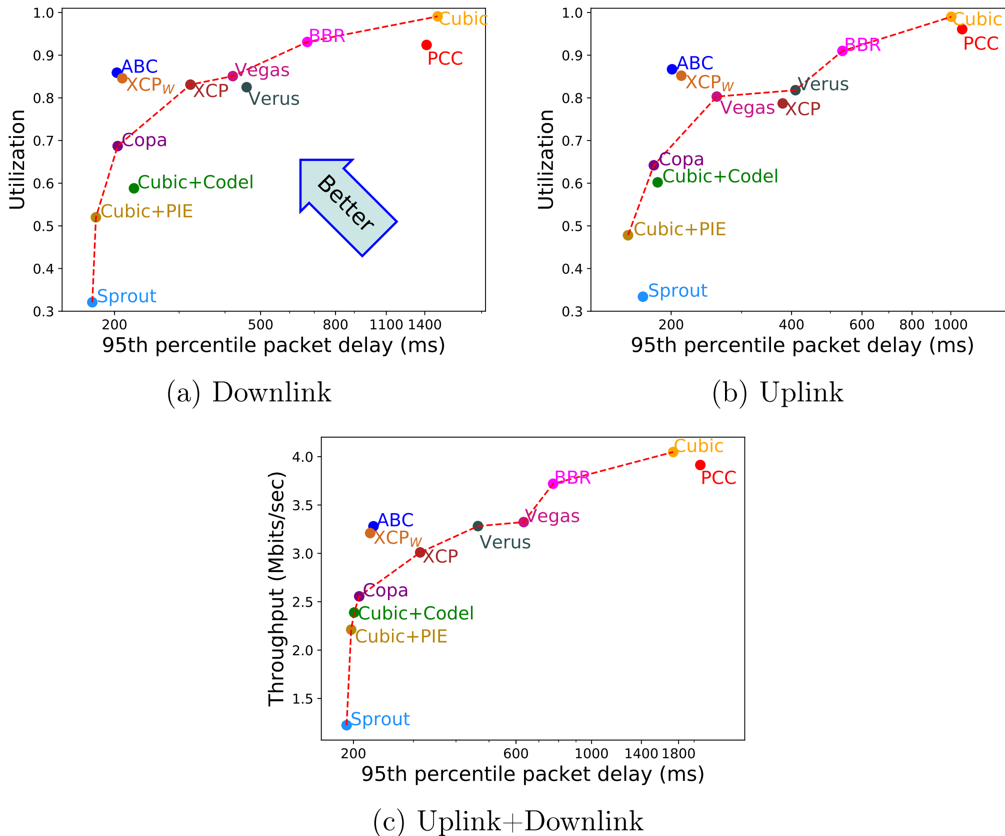
We evaluated ABC in both Wi-Fi and cellular network settings. For Wi-Fi, experiments we used a live Wi-Fi network and the ABC router described in §2.7.1. For cellular settings, we use Mahimahi [110] to emulate multiple cellular networks (Verizon LTE, AT&T, and TMobile). Mahimahi's emulation uses packet delivery traces (separate for uplink and downlink) that were captured directly on those networks, and thus include outages (highlighting ABC's ability to handle ACK losses).

We compare ABC to end-to-end protocols designed for cellular networks (Sprout [142] and Verus [146]), loss-based end-to-end protocols both with and without AQM (Cubic [63], Cubic+Codel [111], and Cubic+PIE [112]), recently-proposed end-to-end protocols (BBR [36], Copa [22], PCC Vivace-Latency (referred as PCC)) [42]), and TCP Vegas [31]), and explicit control protocols (XCP [81], RCP [132] and VCP [143]). We used TCP kernel modules for ABC, BBR, Cubic, PCC, and Vegas; for these schemes, we generated traffic using iperf [134]. For the end-to-end schemes that are not implemented as TCP kernel modules (i.e., Copa, Sprout, Verus), we used the UDP implementations provided by the authors. Lastly, for the explicit control protocols (i.e., XCP, RCP, and VCP), we used our own implementations as qdiscs with Mahimahi to ensure compatibility with our emulation setup. We used Mahimahi's support of Codel and Pie to evaluate AQM.

Our emulated cellular network experiments used a minimum RTT of 100 ms and a buffer size of 250 MTU-sized packets. Additionally, ABC's target rate calculation (Equation (2.1)) used $\eta = 0.98$ and $\delta = 133$ ms. Our Wi-Fi implementation uses the link rate estimator from §2.5, while our emulated cellular network setup assumes the realistic scenario that ABC's router has knowledge of the underlying link capacity [1].

### 2.7.3 Performance

**Cellular:** Fig. 2-7a and 2-7b show the utilization and $95^{th}$ percentile per packet delay that a single backlogged flow achieves using each aforementioned scheme on two Verizon LTE cellular link traces. ABC exhibits a better (i.e., higher) throughput/delay



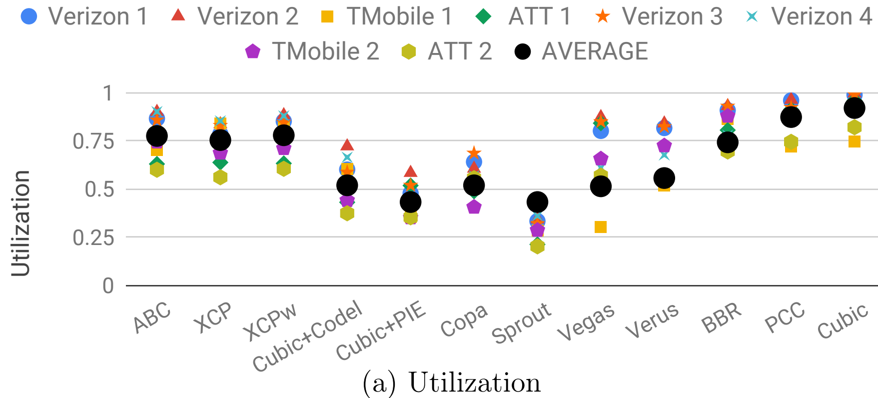

(a) Downlink (b) Uplink

(c) Uplink+Downlink

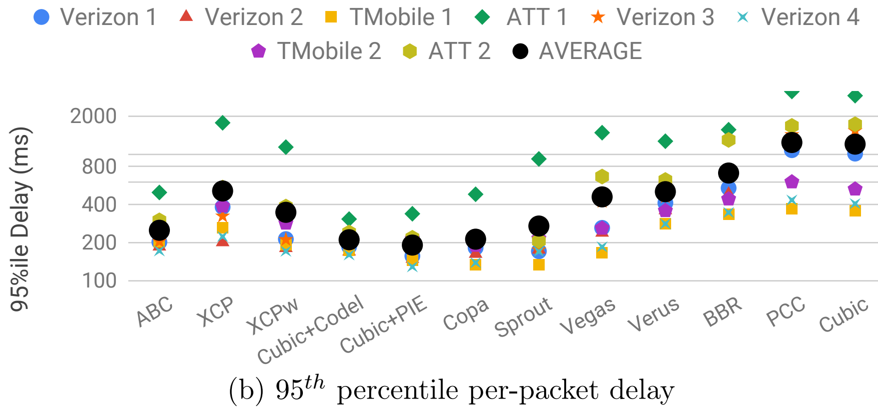
Figure 2-7: **ABC vs. previous schemes on three Verizon cellular network traces** — In each case, ABC outperforms all other schemes and sits well outside the Pareto frontier of previous schemes (denoted by the dashed lines).

tradeoff than all prior schemes. In particular, ABC sits well outside the Pareto frontier of the existing schemes, which represents the prior schemes that achieve higher throughput or lower delay than any other prior schemes.

Further analysis of Fig. 2-7a and 2-7b reveals that Cubic+Codel, Cubic+PIE, Copa, and Sprout are all able to achieve low delays that are comparable to ABC. However, these schemes heavily underutilize the link. The reason is that, though these schemes are able to infer and react to queue buildups in a way that reduces delays, they lack a way of quickly inferring increases in link capacity (a common occurrence on time-varying wireless links), leading to underutilization. In contrast, schemes like BBR, Cubic, and PCC are able to rapidly saturate the network (achieving high utilization), but these schemes also quickly fill buffers and thus suffer from high queuing delays. Unlike these prior schemes, ABC is able to quickly react to *both*




(a) Utilization




(b) $95^{th}$ percentile per-packet delay

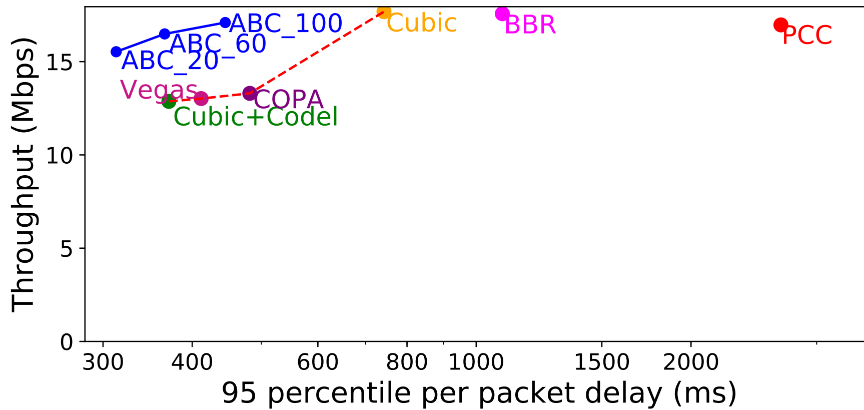
Figure 2-8: **$95^{th}$ percentile per-packet delay across 8 cellular link traces** — On average, ABC achieves similar delays and 50% higher utilization than Copa and Cubic+Codel. PCC and Cubic achieve slightly higher throughput than ABC, but incur 380% higher $95^{th}$ percentile delay than ABC.

increases and decreases in available link capacity, enabling high throughput and low delays.

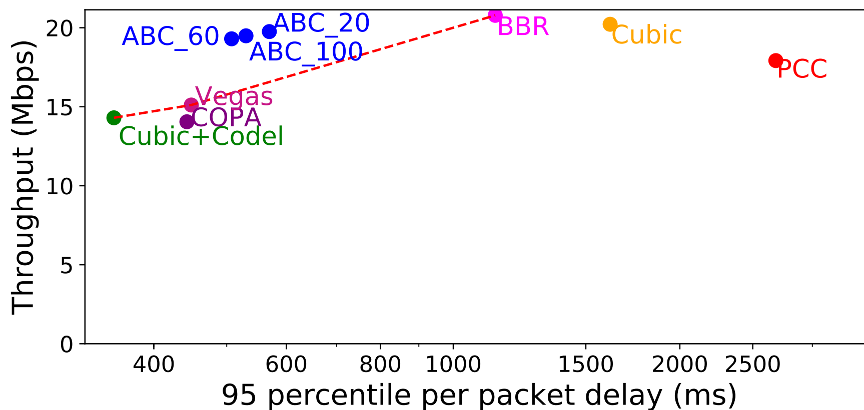
We observed similar trends across a larger set of 8 different cellular network traces (Fig. 2-8). ABC achieves 50% higher throughput than Cubic+Codel and Copa, while only incurring 17% higher $95^{th}$ percentile packet delays. PCC and Cubic achieve slightly higher link utilization values than ABC (12%, and 18%, respectively), but incur significantly higher per-packet delays than ABC (394%, and 382%, respectively). Finally, compared to BBR, Verus, and Sprout, ABC achieves higher link utilization (4%, 39%, and 79%, respectively). BBR and Verus incur higher delays (183% and 100%, respectively) than ABC. Appendix B.3 shows mean packet delay over the same conditions, and shows the same trends.

**Comparison with Explicit Protocols:** Fig. 2-7 and 2-8 also show that ABC outperforms the explicit control protocol, XCP, despite not using multi-bit per-packet feedback as XCP does. For XCP we used $\alpha = 0.55$ and $\beta = 0.4$, the highest permissible stable values that achieve the fastest possible link rate convergence. XCP achieves similar average throughput to ABC, but with 105% higher $95^{th}$ percentile delays. This performance discrepancy can be attributed to the fact that ABC's control rule is better suited for the link rate variations in wireless networks. In particular, unlike ABC which updates its feedback on every packet, XCP computes aggregate feedback values ($\phi$) only once per RTT and may thus take an entire RTT to inform a sender to reduce its window. To overcome this, we also considered an improved version of XCP that recomputes aggregate feedback on each packet based on the rate and delay measurements from the past RTT; we refer to this version as XCP$_w$ (short for XCP wireless). As shown in Fig. 2-7 and Fig. 2-8, XCP$_w$ reduces delay compared to XCP, but still incurs 40% higher $95^{th}$ percentile delays (averaged across traces) than ABC. We also compared with two other explicit schemes, RCP and VCP, and found that ABC consistently outperformed both, achieving 20% more utilization on average. (Appendix B.4).

**Wi-Fi:** We performed similar evaluations on a live Wi-Fi link, considering both single and multi-user scenarios. We connect senders to a WiFi router via Ethernet. Each sender transmits data through the WiFi router to one receiver. All receivers' packets share the same FIFO queue at the router. In this experiment, we excluded Verus and Sprout as they are designed specifically for cellular networks. To mimic common Wi-Fi usage scenarios where endpoints can move and create variations in signal-to-noise ratios (and thus bitrates), we varied the Wi-Fi router's bitrate selections by varying the MCS index using the Linux `iw` utility; we alternated the MCS index between values of 1 and 7 every 2 seconds. In Appendix B-2, we also list results for an experiment where we model MCS index variations as Brownian motion—results show the same trends as described below. This experiment was performed in a crowded computer lab with contention from other Wi-Fi networks. We report average performance values across three, 45 second runs. We considered three different ABC delay threshold ($d_t$)




(a) Single user




(b) Two users, shared queue

Figure 2-9: **Throughout and mean delay on Wi-Fi** — For the multi-user scenario, we report the sum of achieved throughputs and the average of observed $95^{th}$ percentile delay across both users. We consider three versions of ABC (denoted ABC _*) for different delay thresholds. All versions of ABC outperform all prior schemes and sit outside the pareto frontier.

values of 20 ms, 60 ms, and 100 ms; note that increasing ABC's delay threshold will increase both observed throughput and RTT values.

Fig. 2-9 shows the throughput and $95^{th}$ percentile per-packet delay for each protocol. For the multi-user scenario, we report the sum of achieved throughputs and the average $95^{th}$ percentile delay across all users. In both the single and multi-user scenarios, ABC achieves a better throughput/delay tradeoff than all prior schemes, and falls well outside the Pareto frontier for those schemes. In the single user scenario, the ABC configuration with $d_t = 100$ ms achieves up to 29% higher throughput than Cubic+Codel, Copa and Vegas. Though PCC-Vivace, Cubic and BBR achieve slightly

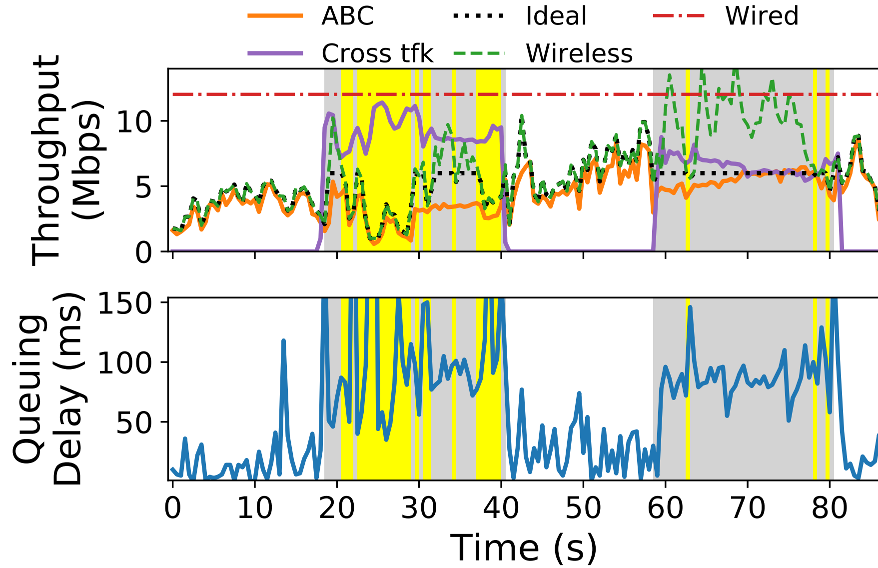



Figure 2-10: **Coexistence with non-ABC bottlenecks** — ABC tracks the ideal rate closely (fair share) and reduces queuing delays in the absence of cross traffic (white region).

higher throughput (4%) than this ABC configuration, their delay values are considerably higher (67%-6×). The multi-user scenario showed similar results. For instance, ABC achieves 38%, 41% and 31% higher average throughput than Cubic+Codel, Copa and Vegas, respectively.

### 2.7.4 Coexistence with Various Bottlenecks

**Coexistence with ABC bottlenecks:** Fig. 2-7c compares ABC and prior protocols on a network path with two cellular links. In this scenario, ABC tracks the bottleneck link rate and achieves a better throughput/delay tradeoff than prior schemes, and again sits well outside the Pareto frontier.

**Coexistence with non-ABC bottlenecks:** Fig. 2-10 illustrates throughput and queuing delay values for an ABC flow traversing a network path with both an emulated wireless link and an emulated 12 Mbits/s fixed rate (wired) link. The wireless link runs ABC, while the wired link operates a droptail buffer. ABC shares the wired link with on-off cubic cross traffic. In the absence of cross traffic (white region), the wireless link is always the bottleneck. However, with cross traffic (yellow and grey regions), due to contention, the wired link can become the bottleneck. In this case,

ABC's fair share on the wired link is half of the link's capacity (i.e., 6 Mbit/s). If the wireless link rate is lower than the fair share on the wired link (yellow region), the wireless link remains the bottleneck; otherwise, the wired link becomes the bottleneck (grey region).

The black dashed line in the top graph represents the ideal fair throughput for the ABC flow throughout the experiment. As shown, in all regions, ABC is able to track the ideal rate closely, even as the bottleneck shifts. In the absence of cross traffic, ABC achieves low delays while maintaining high link utilization. With cross traffic, ABC appropriately tracks the wireless link rate (yellow region) or achieves its fair share of the wired link (grey region) like Cubic. In the former cross traffic scenario, increased queuing delays are due to congestion caused by the Cubic flow on the wired link. Further, deviations from the ideal rate in the latter cross traffic scenario can be attributed to the fact that the ABC flow is running as Cubic, which in itself takes time to converge to the fair share [63].

### 2.7.5 Fairness among ABC and non-ABC Flows

**Coexistence among ABC flows:** We simultaneously run multiple ABC flows on a fixed 24 Mbits/s link. We varied the number of competing flows from 2 to 32 (each run was 60 s). In each case, the Jain Fairness Index [78] was within 5% from the ideal fairness value of 1, highlighting ABC's ability to ensure fairness.

Fig. 2-11 shows the aggregate utilization and delay for concurrent flows (all flows running the same scheme) competing on a Verizon cellular link. We varied the number of competing flows from 1 to 16. ABC achieves similar aggregate utilization and delay across all scenarios, and, outperforms all other schemes. For all the schemes, the utilization and delay increase when the number of flows increases. For ABC, this increase can be attributed to the additional packets that result from additive increase (1 packet per RTT per flow). For other schemes, this increase is because multiple flows in aggregate ramp-up their rates faster than a single flow.

**RTT unfairness:** We simultaneously ran 2 ABC flows on a 24 Mbits wired bottle-

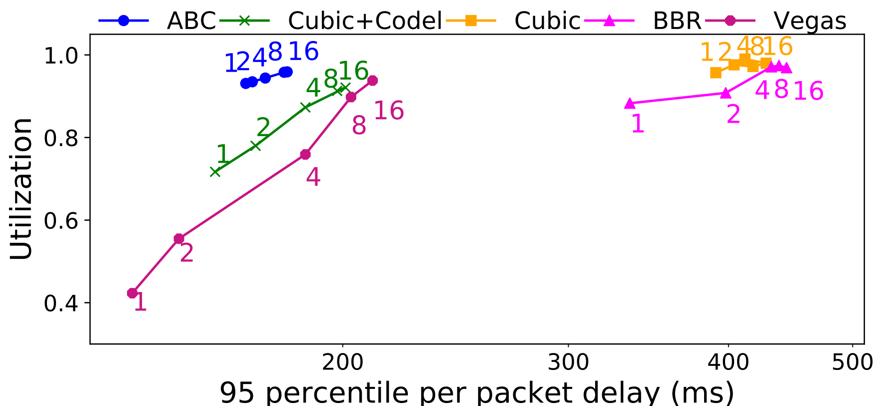



Figure 2-11: **Coexistence among ABC flows** — ABC achieves similar aggregate utilization and delay irrespective of the number of connections. ABC outperforms all previous schemes.

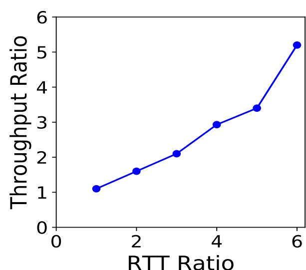



Figure 2-12: **RTT unfairness**

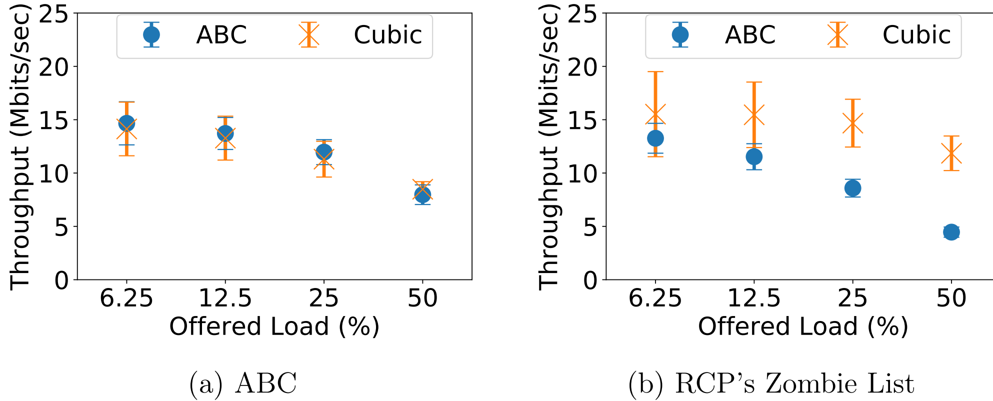
| RTT (ms) | Tput (Mbps) |
|---|---|
| 20 | 6.62 |
| 40 | 4.94 |
| 60 | 4.27 |
| 80 | 3.0 |
| 100 | 2.75 |
| 120 | 2.40 |

Table 2.1: RTT unfairness

neck. We varied the RTT of flow 1 from 20ms to 120ms. RTT of flow 2 was fixed to 20ms. Fig. 2-12 shows the ratio of the average throughput of these 2 flows (average throughput of flow 2 / flow 1, across 5 runs) against the ratio of their RTTs (RTT of flow 1 / flow 2). Increasing the RTT ratio increases the throughput ratio almost linearly and the throughput is inversely proportional to the RTT. Thus, the unfairness is similar to existing protocols like Cubic.

Next, we simultaneously ran 6 ABC flows. The RTT of the flows vary from 20ms to 120ms. Table 2.1 shows the RTT and the average throughput across 5 runs. Flows with higher RTTs have lower throughput. However, note that the flow with the highest RTT (120ms) still achieves ∼35 % of the throughput as flow with the lowest RTT (20ms).

**Coexistence with non-ABC flows:** We consider a scenario where 3 ABC and 3 non-ABC (in this case, Cubic) long-lived flows share the same 96 Mbits/s bottleneck link. In addition, we create varying numbers of non-ABC short flows (each of size 10




(a) ABC

(b) RCP's Zombie List

Figure 2-13: **Coexistence with non-ABC flows** — Across all scenarios, the standard deviation for ABC flows is small and the flows are fair to each other. Compared to RCP's Zombie List strategy, ABC's max-min allocation provides better fairness between ABC and non-ABC flows. With ABC's strategy, the difference in average throughput of ABC and Cubic flows is under 5%.

KB) with Poisson flow arrival times to offer a fixed average load. We vary the offered load values, and report results across 10 runs (40 seconds each). We compare ABC's strategy to coexist with non-ABC flows to RPC's Zombie list approach (§2.4.3).

Fig. 2-13 shows the mean and standard deviation of throughput for long-lived ABC and Cubic flows. As shown in Fig. 2-13a, ABC's coexistence strategy allows ABC and Cubic flows to fairly share the bottleneck link across all offered load values. Specifically, the difference in average throughput between the ABC and Cubic flows is under 5%. In contrast, Fig. 2-13b shows that RCP's coexistence strategy gives higher priority to Cubic flows. This discrepancy increases as the offered load increases, with Cubic flows achieving 17-165% higher throughput than ABC flows. The reason, as discussed in §2.4.3, is that long-lived Cubic flows receive higher throughput than the average throughput that RCP estimates for Cubic flows. This leads to unfairness because RCP attempts to match average throughput for each scheme.Fig. 2-13 also shows that the standard deviation of ABC flows is small (under 10%) across all scenarios. This implies that in each run of the experiment, the throughput for each of the three concurrent ABC flows is close to each other, implying fairness across ABC flows. Importantly, the standard deviation values for ABC are smaller than those for Cubic. Thus, ABC flows converge to fairness faster than Cubic flows do.

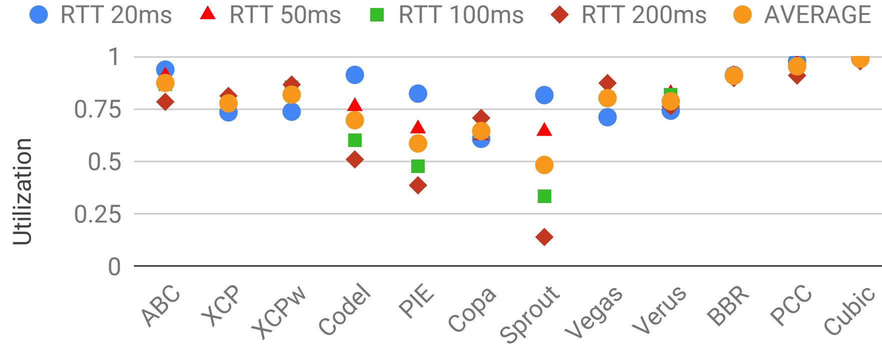
### 2.7.6 Additional Experiments

**Perfect future capacity knowledge:** We considered a variant of ABC, PK-ABC, which knows an entire emulated link trace in advance. This experiment reflects the possibility of resource allocation predictions at cellular base stations. Rather than using an estimate of the current link rate to compute a target rate (as ABC does), PK-ABC uses the expected link rate 1 RTT in the future. On the same setup as Fig. 2-7b, PK-ABC reduces $95^{th}$ percentile per-packet-delays from 97 ms to 28 ms, compared to ABC, while achieving similar utilization (∼90%).

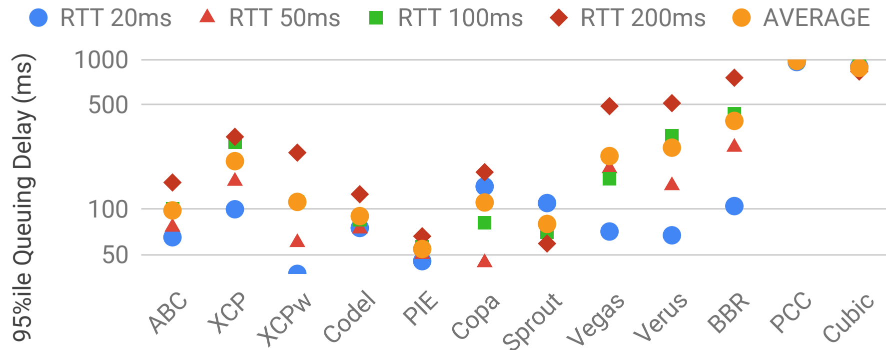
**ABC's improvement on real applications:** We evaluated ABC's improvement for real user-facing applications on a multiplayer interactive game, Slither.io [8]. We loaded Slither.io using a Google Chrome browser which ran inside an emulated cellular link with a background backlogged flow. We considered three schemes for the backlogged flow: Cubic, Cubic+Codel, and ABC. Cubic fully utilizes the link, but adds excessive queuing delays hindering gameplay. Cubic+Codel reduces queuing delays (improving user experience in the game), but underutilizes the link. Only ABC is able to achieve both high link utilization for the backlogged flow and low queuing delays for the game. A video demo of this experiment can be viewed at `https://youtu.be/Dauq-tfJmyU`.

**ABC's sensitivity to network latency:** Thus far, our emulation experiments have considered fixed minimum RTT values of 100 ms. To evaluate the impact that propagation delay has on ABC's performance, we used a modified version of the experimental setup from Fig. 2-8. In particular, we consider RTT values of 20 ms, 50 ms, 100 ms, and 200 ms. Fig. 2-14 shows that, across all propagation delays, ABC is still able to outperform all prior schemes, again achieving a more desirable throughput/latency trade off. ABC's benefits persist even though schemes like Cubic+Codel and Cubic+PIE actually improve with decreasing propagation delays. Performance with these schemes improves because bandwidth delay products decrease, making Cubic's additive increase more aggressive (improving link utilization).

**Application-limited flows:**We created a single long-lived ABC flow that shared a




(a) Utilization




(b) per-packet queuing delay

Figure 2-14: **Impact of propagation delay on performance** — On a Verizon cellular network trace with different propagation delays, ABC achieves a better throughput/delay tradeoff than all other schemes.

cellular link with 200 application-limited ABC flows that send traffic at an aggregate of 1 Mbit/s. Fig. 2-15 shows that, despite the fact that the application-limited flows do not have traffic to properly respond to ABC's feedback, the ABC flows (in aggregate) are still able to achieve low queuing delays and high link utilization.

**Impact of $\eta$:** Fig. 2-16 shows the performance of ABC with various values of $\eta$ on a Verizon cellular trace. Increasing $\eta$ increases the link utilization, but, also increases the delay. Thus, $\eta$ presents a trade-off between throughput and delay.

## 2.8 Related Work

Several prior works have proposed using LTE infrastructure to infer the underlying link capacity [144, 98, 75]. CQIC [98] and piStream [144] use physical layer information at the receiver to estimate link capacity. However, these approaches have

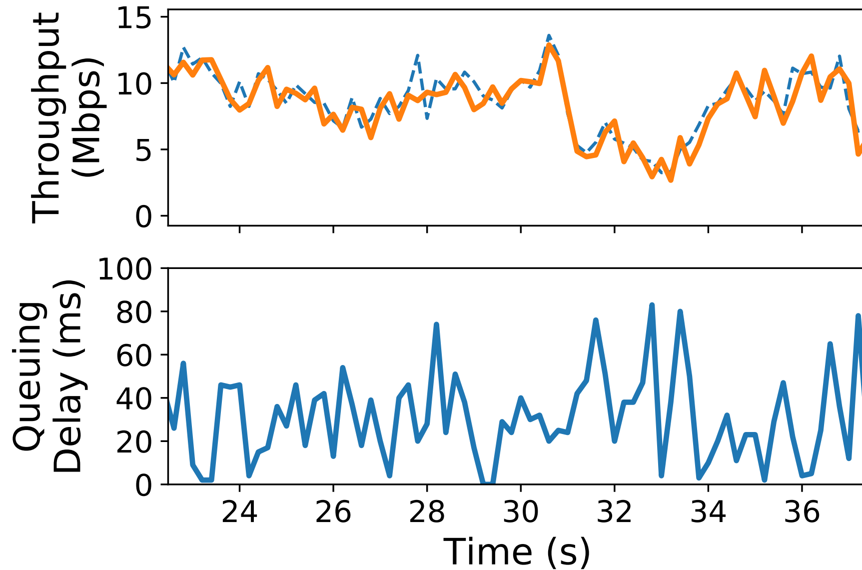



Figure 2-15: **ABC's robustness to flow size** — With a single backlogged ABC flow and multiple concurrent application-limited ABC flows, all flows achieve high utilization and low delays.

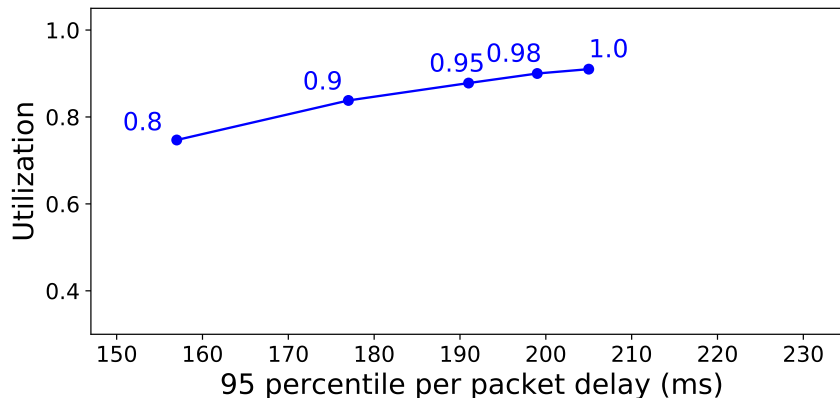



Figure 2-16: **Impact of** $\eta$ — Performance of ABC with various values of $\eta$ (target utilization). $\eta$ presents a trade-off between throughput and delay. Same setup as Fig. 2-1.

several limitations that lead to inaccurate estimates. CQIC's estimation approach considers historical resource usage (not the available physical resources) [144], while piStream's technique relies on second-level video segment downloads and thus does not account for the short timescale variations in link rate required for per-packet congestion control. These inaccuracies stem from the opacity of the base station's resource allocation process at the receiver. ABC circumvents these issues by accurately estimating link capacity directly at the base station.

In VCP [143], router classifies congestion as low, medium, or high, and signals the sender to either perform a multiplicative increase, additive increase, or multiplicative

decrease in response. Unlike an ABC sender, which reacts to ACKs individually, VCP senders act once per RTT. This coarse-grained update limits VCP's effectiveness on time-varying wireless paths. For instance, it can take 12 RTTs to double the window. VCP is also incompatible with ECN, making it difficult to deploy.

In BMCC [117, 115], a router uses ADPM [20] to send link load information to the receiver on ECN bits, relying on TCP options to relay the feedback from the receiver to the sender. MTG proposed modifying cellular base stations to communicate the link rate explicitly using a new TCP option [75]. Both approaches do not work with IPSec encryption [83], and such packet modifications trigger the risk of packets being dropped silently by middleboxes [67]. Moreover, unlike ABC, MTG does not ensure fairness among multiple flows for a user, while BMCC has the same problem with non-BMCC flows [115, 116].

XCP-b [10] is a variant of XCP designed for wireless links with unknown capacity. XCP-b routers use the queue size to determine the feedback. When the queue is backlogged, the XCP-b router calculates spare capacity using the change in queue size and uses the same control rule as XCP. When the queue goes to zero, XCP-b cannot estimate spare capacity, and resorts to a blind fixed additive increase. Such blind increase can cause both under-utilization and increased delays (§2.2.)

Although several prior schemes (XCP, RCP, VCP, BMCC, XCP-b) attempt to match the current enqueue rate to the capacity, none match the future enqueue rate to the capacity, and so do not perform as well as ABC on time-varying links.

## 2.9 Conclusion

ABC is a simple new explicit congestion control protocol for time-varying wireless links. ABC routers use a single bit to mark each packet with "accelerate" or "brake", which causes senders to slightly increase or decrease their congestion windows. Routers use this succinct feedback to quickly guide senders towards a desired target rate. ABC outperforms the best existing explicit flow control scheme, XCP, but unlike XCP, ABC does not require modifications to packet formats or user devices,

making it simpler to deploy. ABC is also incrementally deployable: ABC can operate correctly with multiple ABC and non-ABC bottlenecks, and can fairly coexist with ABC and non-ABC traffic sharing the same bottleneck link. We evaluated ABC using a WiFi router implementation and trace-driven emulation of cellular links. ABC achieves 30-40% higher throughput than Cubic+Codel for similar delays, and 2.2× lower delays than BBR on a Wi-Fi path. On cellular network paths, ABC achieves 50% higher throughput than Cubic+Codel.

# Chapter 3

# Backpressure Flow Control

## 3.1 Introduction

Single and multi-tenant data centers have become one of the largest and fastest growing segments of the computer industry. Data centers are increasingly dominating the market for all types of high-end computing, including enterprise services, parallel computing, large scale data analysis, fault-tolerant middleboxes, and global distributed applications [56, 105, 18]. These workloads place enormous pressure on the data center network to deliver, at low cost, ever faster throughput with low tail latency even for highly bursty traffic [41, 147].

Although details vary, almost all data center networks today use a combination of endpoint congestion control, FIFO queues at switches, and end-to-end feedback of congestion signals like delay or explicit switch state to the endpoint control loop.[1] As link speeds continue to increase, however, the design of the control loop becomes more difficult. First, more traffic fits within a single round trip, making it more difficult to use feedback effectively. Second, traffic becomes increasingly bursty, so that network load is not a stable property except over very short time scales. And third, switch buffer capacity is not keeping up with increasing link speeds (Fig. 3-1), making it even more challenging to handle traffic bursts. Most network operators run their networks

[1] In this chapter, we refer to schemes that rely on feedback signals delayed by an entire round-trip-time as *end-to-end* schemes, to contrast them with hop-by-hop mechanisms.

at very low average load, throttle long flows at far below network capacity, and even then see significant congestion loss.

Instead, we propose a different approach. The key challenge for data center networks, in our view, is to efficiently allocate buffer space at congested network switches. This becomes easier and simpler when control actions are taken per flow and per hop, rather than end-to-end. Despite its advantages, per-hop per-flow flow control appears to require per-flow state at each switch, even for quiescent flows [19, 91], something that is not practical at data center scale.

Our primary contribution is to show that per-hop per-flow flow control can be *approximated* with a limited amount of switch state and modest number of switch queues, using only simple constant-time switch operations on a modern programmable switch. Instead of all flows, we only need state and dedicated queues for *active flows*—those flows with queued packets. We show that, with switch-level fair queueing or shortest flow scheduling, the number of active flows is modest for typical data center workloads, even in the tail of the distribution. The tradeoff is that performance can degrade when the number of active flows exceeds the number of queues. In practice, we advocate combining per-hop flow control with end-to-end congestion control to avoid pathological behavior. However, to better illustrate the benefits and limitations of our approach, our description and experiments focus on comparing pure per-hop control with pure end-to-end control.

We have implemented our approach, *Backpressure Flow Control (BFC)*, on Tofino2 a state-of-the-art P4-based programmable switch ASIC supporting 12.8 Tbps of switching capacity [72]. Tofino2 has 32-128 independently pausable queues at each output port. Our implementation uses less than 10% of the dedicated stateful memory on Tofino2. All per-packet operations are implemented entirely in the dataplane; BFC runs at full switch capacity.

To evaluate performance, we run large-scale ns-3 [5] simulations using synthetic traces drawn to be consistent with measured workloads from Google and Facebook data centers [107] on an oversubscribed multi-level Clos network topology. We synthetically add incast to these workloads to represent a challenging scenario for both

end-to-end and per-hop approaches. We consider both throughput and tail latency performance for short, medium, and long flows.

For our simulated workloads, BFC improves both latency for short flows and throughput for long flows. Compared to a wide set of deployed end-to-end systems, including DCTCP [14], DCQCN [150], and HPCC [96], BFC achieves 2.3 - 60× better tail flow completion times (FCTs) for short flows, and 1.6 - 5× better average performance for long flows. ExpressPass [38] achieves 35% better short flow tail latency, but 17× worse average case performance for long flows. We also show that BFC performs close to an idealized fair queueing system with unbounded buffers and switch queues, but with limited queues and far smaller buffers. BFC can be combined with other switch scheduling algorithms such as priority scheduling among traffic classes. Unlike other receiver-driven schemes like Homa [107], BFC does not assume knowledge of flow sizes and does not rely on packet spraying (which is difficult to deploy in practice). With packet spraying, Homa outperforms BFC, but without it we show BFC outperforms Homa and can enforce shortest remaining flow first scheduling more accurately.

Our specific contributions are:

- A discussion of the fundamental limits of end-to-end congestion control for high bandwidth data center networks.

- A practical protocol for per-hop per-flow flow control, called BFC, that uses a small, constant amount of state and limited number of switch queues to achieve near-optimal tail-latency performance for typical data center workloads.

- An implementation and proof-of-concept evaluation of BFC on a commercial switch. To our knowledge, this is the first implementation of a per-hop per-flow flow control scheme for a multi-Tbps switch.

## 3.2 Motivation

Over the last decade, researchers and data center operators have proposed a variety of congestion control algorithms for data centers, including DCTCP [14], Timely [106], Swift [90], DCQCN [150], and HPCC [96]. The primary goals of these protocols are to achieve high throughput, low tail packet delay, and high resilience to bursts and incast traffic patterns. Operationally, these protocols rely on *end-to-end* feedback loops, with senders adjusting their rates based on congestion feedback signals echoed by the receivers. Irrespective of the type of signal (e.g., ECN marks, multi-bit INT information [96, 85], delay), the feedback delay for these schemes is a network round-trip time (RTT). This delay has an important role in the performance of end-to-end schemes. In particular, senders require at least one RTT to obtain feedback, and therefore face a hard tradeoff in deciding the starting rate of a flow. They can either start at a high rate and risk causing congestion, or start at a low rate and risk under-utilizing the network. Moreover, even after receiving feedback, senders can struggle to determine the right rate if the state of the network (e.g., link utilization and queuing delay) changes quickly compared to the RTT.

We argue that three trends are making these problems worse over time, and will make it increasingly difficult to achieve good performance with end-to-end protocols.

**Trend 1: Rapidly increasing link speed.** Fig. 3-1 shows the switch capacity of top-of-the-line data center switches manufactured by Broadcom [33, 113, 139]. Switch capacity and link speeds have increased by a factor of 10 over the past six years with no signs of stopping.

**Trend 2: Most flows are short.** Fig. 3-2 shows the byte-weighted cumulative distribution of flow sizes for three industry data center workloads [107]: (1) All applications in a Google data center, (2) Hadoop cluster in a Facebook center, and (3) a WebSearch workload. Each point is the fraction of all bytes sent that belong to flows smaller than a threshold for that workload. For example, for the Google workload, flows that are shorter than 100 KB represent nearly half of all bytes. As link speed increases, a growing fraction of traffic belongs to flows that complete quickly relative

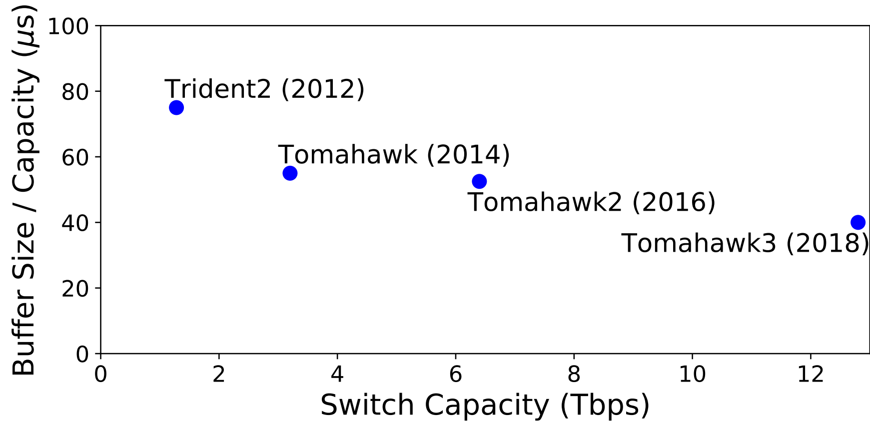



Figure 3-1: **Hardware trends for top-of-the-line data center switches from Broadcom** — Switch capacity and link speed have been growing rapidly, but buffer size is not keeping up with increases in switch capacity.

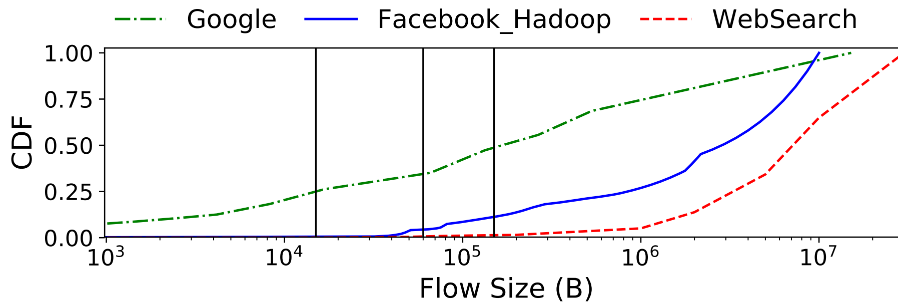



Figure 3-2: Cumulative bytes contributed by different flow sizes for three different industry workloads. The three vertical lines show the BDP for a 10 Gbps, 40 Gbps, and 100 Gbps network, assuming a 12 $\mu$s RTT.

to the RTT. For example, most Facebook Hadoop traffic is likely to fit within one round trip within the next decade. While some have argued that data center flows are increasing in size [11], the trend is arguably in the opposite direction with the growing use of RDMA for fine-grained remote memory access.

**Trend 3: Buffer size is not scaling with switch capacity.** Fig. 3-1 shows that the total switch buffer size relative to its capacity has decreased by almost a factor of 2 (from 75 $\mu$s to 40 $\mu$s) over the past six years. With smaller buffers relative to link speed, buffers now fill up more quickly, making it more difficult for end-to-end congestion control to manage those buffers.

| Scheme | Throughput (%) | 99% Queuing Delay ($\mu$s) |
|---|---|---|
| BFC | 37.3 | 1.2 |
| HPCC | 22.9 | 23.9 |
| DCQCN | 10.0 | 30.4 |

Table 3.1: For a shared 100 Gbps link, BFC achieves close to ideal throughput (40%) for the long flow, with low tail queuing delay.

### 3.2.1 Limits of End-to-End Congestion Control

This combination — very fast links, short flows, and inadequate buffers — creates the perfect storm for end-to-end congestion control protocols. Flows that complete within one or a few RTTs (which constitute an increasingly larger fraction of traffic) either receive no feedback, or last for so few feedback cycles that they cannot find the correct rate [80]. For longer flows, the rapid arrival and departure of cross-traffic creates significant fluctuations in available bandwidth at RTT timescales, making it difficult to find the correct rate. The result is loss of throughput and large queue buildup. Insufficient switch buffering further exacerbates these problems, leading to packet drops or link-level pause events (PFC [141]) that spread congestion upstream.

To understand these issues, we consider an experiment with a long-lived flow competing on a single link against cross-traffic derived from the Google, Facebook, and WebSearch workloads. We repeat the experiment at 10, 40, and 100 Gbps, with the average load of the cross-traffic flows set to be 60% of the link capacity in each case. Fig. 3-3 plots the relative change in the fair-share rate of the long-lived flow over different time intervals.[2] Congestion control protocols struggle to track the fair-share rate when it varies significantly over their feedback delay (typically an RTT). As link speeds increase or flows become shorter, the fair-share rate changes more rapidly (since flows arrive and finish more quickly), and hence congestion control becomes more difficult.

Table 3.1 considers one configuration in detail, with a single long flow sharing a 100 Gbps link with cross-traffic drawn from the Facebook distribution at 60% average load. The minimum RTT (hence, feedback delay) is $8\,\mu$s. We consider both the

[2]The fair-share rate ($f(t)$) for a link of capacity $C$ shared by $N(t)$ flows is $C/N(t)$. The relative change in $f(t)$ over time interval $I$ is given by $\mid \frac{f(t+I)-f(t)}{f(t)} \mid$.

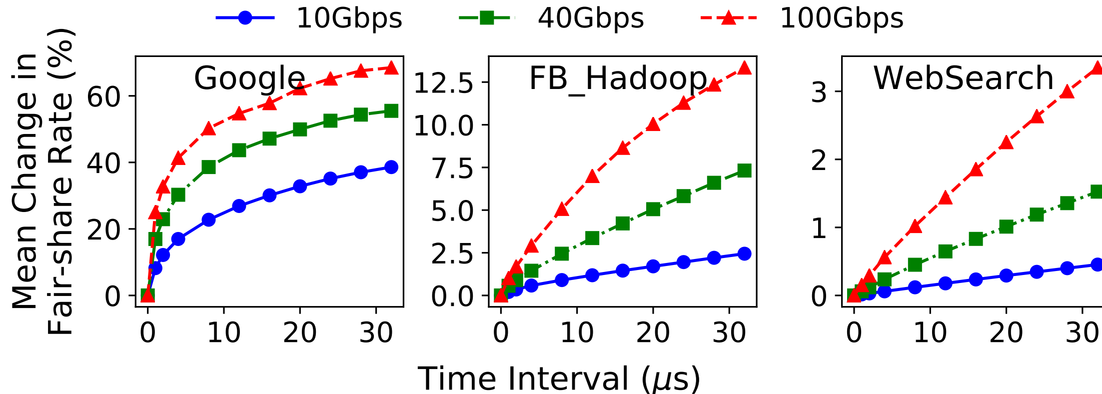



Figure 3-3: Mean percent change in fair-share rate as a function of workload, delay, and bandwidth.

single packet ($99^{th}$ percentile) queuing delay and throughput for the long flow, for our approach (BFC) and two end-to-end protocols (DCQCN and HPCC). BFC is able to achieve close to the maximum possible throughput for the long-lived flow (40%) with low tail delay, while the end-to-end protocols fall short in both respects.

### 3.2.2 Existing Solutions are Insufficient

There are several existing solutions that go beyond end-to-end congestion control. We briefly discuss the most prominent of these approaches and why they are insufficient to deal with the challenges described above.

**Priority flow control (PFC).** One approach to handling increased buffer occupancy would be to use PFC, a hop-by-hop flow control mechanism.[3] With PFC, if the packets from a particular input port start building up at a congested switch (past a configurable threshold), the switch sends a "pause" frame upstream, stopping that input from sending more traffic until the switch has a chance to drain stored packets. This prevents switch buffers from being overrun. Unfortunately, PFC has a side effect: head-of-line (HoL) blocking [150]. For example, incast traffic to a single server can cause PFC pause frames to be sent one hop upstream towards the source of the traffic. This stops *all* the traffic traversing the paused link, even those flows that

[3]For simplicity, we focus on the case where there is congestion among the traffic at a particular priority level.

are destined to other uncongested egress ports. These flows will be delayed until the packets at the congested port can be drained. Worse, as packets queue up behind a PFC, additional PFC pause frames can be triggered at upstream hops, widening the scope of HoL blocking.

**Switch scheduling.** Several efforts use switch scheduling to overcome the negative side-effects of elephant flows on the latency of short flows. These proposals range from approximations of fair queuing (e.g., Stochastic Fair Queuing [103], Approximate Fair Queuing [123]) to scheduling policies that prioritize short flows (e.g., pFabric [16], QJump [61], Homa [107]). Our work is orthogonal to the choice of switch scheduling policy, and we present results with priority scheduling and shortest flow first. Scheduling by itself does nothing to reduce buffer occupancy; buffers can fill, causing packet drops or HoL blocking, regardless of scheduling.

**Receiver-based congestion control.** Because sender-based congestion control schemes generally perform poorly on incast workloads, some researchers have proposed shifting to a scheme where the receiver prevents congestion by explicitly allocating credits to senders for sending traffic. Three examples are NDP [64], pHost [51] and Homa [107]. BFC makes fewer assumptions than these approaches. Homa, for example, assumes knowledge of the flow size distribution and flow length, so that it can assign flows to near-optimal priority queues; this is unavailable with today's TCP socket interface and not all applications know flow lengths in advance [23, 135]. Homa uses packet spraying to achieve better load balancing, so that congestion primarily occurs at the last hop, where the receiver has complete visibility. However, congestion-free operation of the core is difficult to engineer for widely deployed oversubscribed and asymmetric networks [124, 151, 149]. Packet spraying can also cause packet reordering, which is incompatible with high-speed end host software and hardware packet handling [101, 82]. Other proposals suggest collecting credits generated by a flow's receiver (congestion-controlled by all switches on the flow's path) before sending [38]; at high link speeds, the network state changes rapidly over the feedback delay, making it difficult for the receiver to determine the right rate for credits, similar to sender-based protocols.

### 3.2.3 Revisiting Per-hop, Per-Flow Flow Control

Our approach is inspired by work in the early 90s on hop-by-hop credit-based flow control for managing gigabit ATM networks [19, 91]. Credit-based flow control was also introduced by multiprocessor hardware designs of the same era [87, 95, 32]. In these systems, each switch methodically tracks its buffer space, granting permission to send at an upstream switch if and only if there is room in its buffer. In ATM, packets of different flows are buffered in separate queues and are scheduled according to the flows' service requirements. The result is a network that has no congestion loss by design.

An ideal realization of such a per-hop, per-flow flow control scheme has several desirable properties:

**(1) Fast reaction:** When a flow starts experiencing congestion at a switch, the upstream switch can reduce its rate within a 1-Hop RTT, instead of the end-to-end RTT that it takes for standard congestion control schemes. Likewise, when capacity becomes available at a switch, the upstream switch can increase its rate within a 1-Hop RTT (provided the upstream switch has packets from that flow). Assuming a hardware implementation, the 1-hop RTT consists of the propagation latency and the switch pipeline latency — typically 1-2 $\mu$s.[4] This is substantially smaller than the typical end-to-end RTT in data centers (e.g., 10-20 $\mu$s), which in addition to multiple switch hops includes the latency at the endpoints.

**(2) Buffer pooling:** During traffic bursts, a per-hop per-flow flow control mechanism throttles traffic upstream from the bottleneck. This enables the bottleneck switch to tap into the buffers of its upstream neighbors, thereby significantly increasing the ability of the network to absorb bursts.

**(3) No HoL blocking:** Unlike PFC, there is no HoL blocking or congestion spreading with per-hop per-flow flow control, because switches isolate flows in different queues and perform flow control for each of them separately.

**(4) Simple control actions:** Flow control decisions in a per-hop per-flow flow

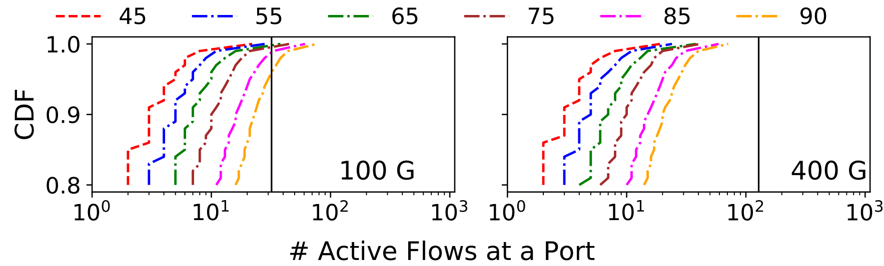
[4]For example, a 100 m cable has a propagation latency of 500 ns, and a typical data center switch has a pipeline latency around 500 ns [33, 26].

control system are simpler to design and reason about than end-to-end congestion control algorithms because: *(i)* whether to send or pause a flow at a switch depends only on feedback from the immediate next-hop switch (as opposed to multiple potential points of congestion with end-to-end schemes), *(ii)* concerns like fairness are dealt with trivially by scheduling flows at each switch, and therefore flow control can focus exclusively on the simpler task of managing buffer occupancy and ensuring high utilization.

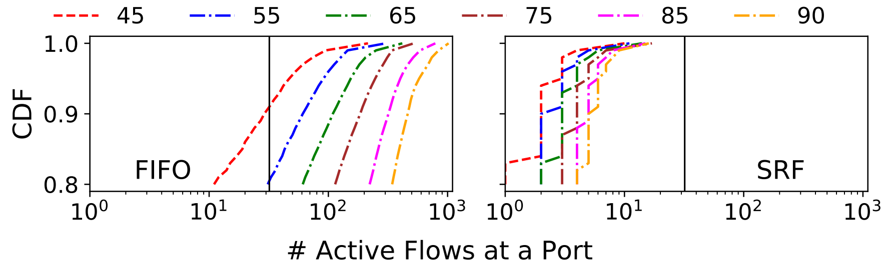
Despite these compelling properties, per-hop per-flow flow control schemes have not been widely deployed, in part because of their high implementation complexity and resource requirements. ATM schemes require per-connection state and large buffers, which are not feasible in today's data center switches. We observe, however, that per-connection switch state is not actually required. Indeed, much of the time, per-connection state is for flows that have no packets queued at the switch, and therefore don't need to be flow controlled.

We define an *active flow* to be a flow with one or more packets queued at the switch. A result of queuing theory is that the number of active flows is surprisingly small for a switch using fair queuing [86, 88]. In particular, for an M/G/1-PS (Processor Sharing) queue with Poisson flow arrivals operating at average load $\rho < 1$, the number of active flows has a geometric distribution with mean $\frac{\rho}{1-\rho}$, independent of the link speed or the flow size distribution. Even at load $\rho = 0.9$, the expected number of active flows is only 9. The intuition behind this fact is that a fair queued switch will tend to process short flows quickly, completing them and keeping the number of active flows small.

Data center network workloads are often more bursty than Poisson, leading to longer queues and more active flows. However, the basic principle still holds. Fig. 3-4 shows the cumulative distribution of the number of active flows for a single bottleneck link operating at different loads and link speeds, using the Google flow size distribution and (bursty) log-normal flow inter-arrival times. The upper graph assumes fair queuing and includes a vertical bar for the number of queues per port on Tofino2. At 100 Gbps, the number of active flows significantly exceeds the number of queues




(a) Number of active flows vs. link speed, with fair queuing




(b) Number of active flows vs. scheduling policy, 100G

Figure 3-4: **Number of active flows for different load, link speed, and scheduling policy** — Lines correspond to different loads. Flow sizes are from the Google distribution with lognormal ($\sigma = 2$) inter-arrival times.

only for loads above 85%, and then only modestly; importantly, the distribution is invariant to link speed, and the trend is for faster links to have more queues. The result holds even more strongly with shortest remaining flow first (SRF) scheduling. By contrast, with FIFO queuing, even a single long flow can cause a large number of small flows to back up behind it, and therefore the number of active flows is much larger.

## 3.3 Design

Our goal is to design a practical system for per-hop, per-flow flow control for data center networks. We first describe the constraints on our design (§3.3.1). We then sketch a plausible strawman proposal that surprisingly turns out to not work well at all (§3.3.2), and we use that as motivation for our design (§3.3.3).

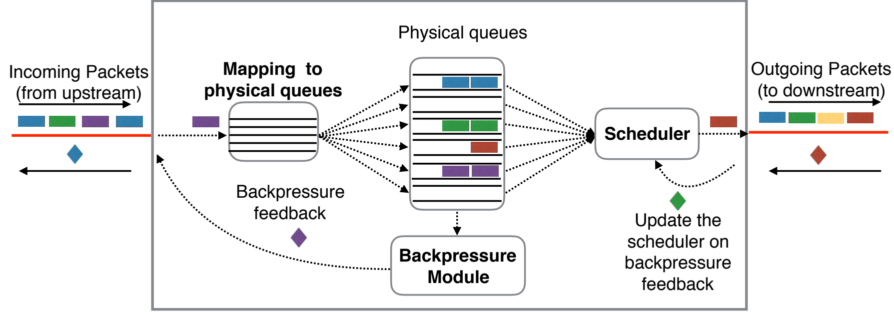



Figure 3-5: **Logical switch components in per-hop, per-flow flow control.**

### 3.3.1 Design Constraints

Fig. 3-5 shows the basic components of a per-hop, per-flow flow control scheme (per port). (1) *Mapping to physical queues:* When a packet arrives at the switch, the switch routes the packet to an egress port and maps it to a FIFO queue at that port. This assignment of flows to queues must be consistent, that is, respect packet ordering. (2) *Backpressure module:* Based on queue occupancy, the switch generates backpressure feedback for some flows and sends it upstream. (3) *Scheduler:* The scheduler at each egress port forwards packets from queues while respecting backpressure feedback from the downstream switch.

ATM per-hop per-flow flow control systems [19, 91] roughly followed this architecture, but they would be impractical for modern data centers. First, they assumed per-flow queues and state, but modern switches have a limited number of queues per egress port [123, 29] and modest amounts of table memory [30, 39]. In particular, it is not possible to maintain switch state for all live connections. Second, earlier schemes did not attempt to minimize buffer occupancy. Instead, they sent backpressure feedback only when the switch was about to run out of buffers. On a buffer-constrained switch, this can result in buffer exhaustion — buffers held by straggler flows can prevent other flows from using those buffers at a later time.

**Hardware assumptions.** Modern data center switches have made strides towards greater flexibility [126, 21], but they are not infinitely malleable and have real resource constraints. We make the following assumptions based on the capabilities of Tofino2.

1. We assume the switch is programmable and supports stateful operations.Tofino2

can maintain millions of register entries, and supports simple constant-time per-packet operations to update the state at line rate [125].

2. The switch has a limited number of FIFO queues per egress port, meaning that flows must be multiplexed onto queues. Tofino2 has 32/128 queues per 100/400G port. The assignment of flows to queues is programmable. The scheduler can use deficit round-robin or priorities among queues, but packets within a queue are forwarded in FIFO order.
3. Each queue can be independently paused and resumed without slowing down forwarding from other queues. When we pause a queue, that pauses *all* of the flows assigned to that queue. The switch can pause/resume each queue directly within the dataplane.

### 3.3.2 A Strawman Proposal

We originally thought stochastic fair queuing [103] with per-queue backpressure might meet our goals: use a hash function on the flow header to consistently assign the packets of each flow to a randomly-chosen FIFO queue at its egress port, and pause a queue whenever its buffer exceeds the 1-hop bandwidth-delay product (BDP). For simplicity, use the same hash function at each switch.

This strawman needs only a small amount of state for generating the backpressure feedback and no state for queue assignment. However, with even a modest number of active flows, the birthday paradox implies that there is a significant chance that any specific flow will land in an already-occupied FIFO queue. These collisions hurt latency for two reasons: (1) The packets for the flow will be delayed behind unrelated packets from other flows; for example, a short flow may land behind a long flow. (2) Queue sharing can cause HoL blocking. If a particular flow is paused (because it is congested downstream), all flows sharing the same queue will be delayed.

To prevent collisions from affecting tail latency performance, the strawman requires significantly more queues than active flows. For example, at an egress port with $n$ active flows, to achieve fewer than 1% collisions, we would need roughly $100n$ queues.

### 3.3.3 Backpressure Flow Control (BFC)

Our design achieves the following properties:

**Minimal HoL blocking:** We assign flows to queues dynamically. As long as the number of active flows at an egress is less than the number of queues, (with high probability) no two flows share a queue and there is no HoL blocking. When a new flow arrives at the switch, it is assigned to an empty queue if one is available, sharing queues only if all are in use.

**Low buffering and high utilization:** BFC pauses a flow at the upstream when the queue occupancy exceeds a small threshold. BFC's pause threshold is set aggressively to reduce buffering. With coarse pausing like PFC, pausing aggressively hurts utilization, but BFC only pauses those flows causing congestion (except when collisions occur). The remaining flows at the upstream can continue transmitting, avoiding under-utilization.

**Hardware feasibility:** BFC does not require per-flow state, and instead uses an amount of memory proportional to the number of physical queues in the switch. To allow efficient lookup of the state associated with a flow, the state is stored in a flow table, an array indexed using a hash of the flow identifier. The size of this array is set in proportion to the number of physical queues. In our Tofino2 implementation, it consumes less than 10% of the dedicated stateful memory. Critically, the mechanism for generating backpressure and reacting to it is simple and the associated operations can be implemented entirely in the dataplane at line rate.

**Generality:** BFC does not make assumptions about the network topology or where congestion can occur, and does not require packet spraying like NDP [64] or Homa [107]. Furthermore, it does not assume knowledge of flow sizes or deadlines. Such information can be incorporated into BFC's design to improve small flow performance (see Appendix F.1), at a cost in deployability.

**Idempotent state:** Because fiber packets can be corrupted in flight [151], BFC ensures that pause and resume state is maintained idempotently, in a manner resilient to packet loss.

#### 3.3.3.1 Assigning Flows to Queues

To minimize sharing of queues and HoL blocking, we dynamically assign flows to empty queues. As long as the flow is active (has packets queued at the switch), subsequent packets for that flow will be placed into the same FIFO queue. Each flow has a unique 5-tuple of the source and destination addresses, port numbers, and protocol; we call this the flow identifier (FID). BFC uses the hash of the FID to track a flow's queue assignment. To simplify locating an empty queue, BFC maintains a bit map of empty queues. When the last packet in a queue is scheduled, BFC resets the corresponding bit for that queue.

With dynamic queue assignment, a flow can be assigned to different queues at different switches. To pause a flow, BFC pauses the queue the flow came from at the upstream switch (called the upstream queue). The pause applies to all flows sharing the same upstream queue with the paused flow. We describe the pause mechanism in detail in §3.3.3.2. The packet scheduler uses deficit round robin to implement fair queuing among the queues that are not paused.

Since there is a limited number of queues, it is possible that all queues have been allocated when a new flow arrives, at which point HoL blocking is unavoidable. For hardware simplicity, we assign the flow to a random queue in this case. Packets assigned to the same queue are scheduled in FIFO order. The number of active flows is usually small (§3.2.3), but in certain settings, such as incast, it can exceed the number of queues. BFC's behavior is similar to stochastic fair queuing in such scenarios in that it incurs HoL blocking. BFC still outperforms existing protocols like DCQCN and HPCC except in the most extreme cases (see §3.6.3, §3.6.4). Even during a large scale incast, BFC can leverage the large number of upstream queues feeding traffic to a bottleneck switch to (1) absorb larger bursts, and (2) limit congestion spreading. In particular, when flows involved in an incast are spread among multiple upstream ports, BFC assigns these flows to separate queues at those ports. As long as the total number of flows does not exceed the total number of queues across *all* of the upstream ports, BFC will not incur HoL blocking at the upstream switches. As

the size of the network increases and the fan-in to each switch gets larger, there will be even more queues at the upstream switches to absorb an incast, further reducing congestion spreading.

**Mechanism:** To keep track of queue assignment, BFC maintains an array indexed by the egress port of a flow and the hash of the FID. All flows that map to the same index are assigned to the same queue. We maintain the following state per entry: the physical queue assignment (`qAssignment`), and the number of packets in the queue from the flows mapped to this entry (`size`). The pseudocode is as follows (we defer switch-specific implementation issues to §3.4):

```
On Enqueue(packet):
    key = <packet.egressPort, hash(packet.FID)>
    if flowTable[key].size == 0:
        reassignQueue = True:
    flowTable[key].size += 1
    if reassignQueue:
        if empty q available at packet.egressPort:
            qAssignment = emptyQ
        else:
            qAssignment = randomQ
        flowTable[key].qAssignment = qAssignment
    packet.qAssignment = flowTable[key].qAssignment

On Dequeue(packet):
    key = <packet.egressPort, hash(packet.FID)>
    flowTable[key].size -= 1
```

In the flow table, if two flows map to the same index they will use the same queue (collision). Since flows going through different egress ports cannot use the same queue, the index also includes the egress port. Index collisions in the flow table can hurt performance. These collisions decrease with the size of the table, but the flow table cannot be arbitrarily large as the switch has a limited stateful memory. In our design, we set the size of the flow table to 100 $\times$ the number of queues in the switch. This ensures that if the number of flows at an egress port is less than the number of

queues, then the probability of index collisions is less than 1%. If the number of flows exceeds the number of queues, then the index collisions do not matter as there will be collisions in the physical queues regardless. Tofino2 has 4096 queues in aggregate, and hence the size of the flow table is 409,600 entries, which is less than 10% of the switch's dedicated stateful memory.

While using an array is not memory efficient, accessing state involves simple operations. Existing solutions for maintaining flow state either involve slower control plane operations, or are more complex [114, 25]. In the future, if the number of queues increases substantially, we can use these solutions for the flow table; however at the moment, the additional complexity is unnecessary.

#### 3.3.3.2 Backpressure Mechanism

BFC pauses a flow if the occupancy of the queue assigned to that flow exceeds the pause threshold $Th$. To pause/resume a flow, the switch could signal the flow ID to the upstream switch, which can then pause/resume the queue associated with the flow. While this solution is possible in principle, it is difficult to implement on today's programmable switches. The challenge is that, on receiving a pause, the upstream switch needs to perform a lookup to find the queue assigned to the flow and some additional bookkeeping to deal with cases when a queue has packets from multiple flows (some of which might be paused and some not).

We take a different approach. Switches directly signal to the upstream device to pause/resume a specific queue. Each upstream switch/source NIC inserts its local queue number in a special header field called `upstreamQ`. The downstream switch uses this information to pause the queue at the upstream.

**Mechanism:** Recall that, in general, multiple flows can share a queue in rare cases. This has two implications. First, we track the queue length (and not just the `flowTable.size`) and use that to determine if the flow's upstream queue should be paused. Second, each upstream queue can, in general, have flows sending packets to multiple queues at multiple egresses. We pause an upstream queue if *any* of its flows are assigned a congested queue, and we resume when *none* of its flows have

packets at a congested queue (as measured at the time the packet arrived at the switch).

We monitor this using a Pause Counter, an array indexed by the ingress port and the `upstreamQ` of a packet. The upstream queue is paused if and only if its Pause Counter at the downstream switch is non-zero. On enqueue of a packet, if its flow is assigned a queue that exceeds the pause threshold, we increment the pause counter at that index by 1. When this packet (the one that exceeded $Th$) leaves the switch we decrement the counter by 1. Regardless of the number of flows assigned to the `upstreamQ`, it will be resumed only once all of its packets that exceeded the pause threshold (when the packet arrived) have left the switch.

```
On Enqueue(packet):
    key = <packet.ingressPort, packet.upstreamQ>
    if packet.qAssignment.qLength > Th:
        packet.metadata.counterIncr = True
        pauseCounter[key] += 1
        if pauseCounter[key] == 1:
            //Pause the queue at upstream
            sendPause(key)
```

```
On Dequeue(packet):
    key = <packet.ingressPort, packet.upstreamQ>
    if packet.metadata.counterIncr == True:
        pauseCounter[key] -= 1
        if pauseCounter[key] == 0:
            //Resume the queue at upstream
            sendResume(key)
```

To minimize bandwidth consumed in sending pause/resumes, we only send a pause packet when the pause counter for an index goes from 0 to 1, and a resume packet when it goes from 1 to 0. For reliability against pause/resume packets being dropped, we also periodically send a bitmap of the queues that should be paused at the upstream (using the pause counter). Additionally, the switch uses a high priority queue for processing the pause/resume packets. This reduces the number of queues available for dynamic queue assignment by 1, but it eliminates performance degradation due

to delayed pause/resume packets.

The memory required for the pause counter is small compared to the flow table. For example, if each upstream switch has 128 queues per egress port, then for a 32-port downstream switch, the pause counter is 4096 entries.

**Pause threshold.** BFC treats any queue buildup as a sign of congestion. BFC sets the pause threshold $Th$ to 1-Hop BDP at the queue drain rate. Let $N_{active}$ be the number of *active queues* at an egress, i.e. queues with data to transmit that are not paused, $HRTT$ be the 1-Hop RTT to the upstream, and $\mu$ be the port capacity. Assuming fair queuing as the scheduling policy, the average drain rate for a queue at the egress is $\mu/N_{active}$. The pause threshold $Th$ is thus given by $(HRTT) \cdot (\mu/N_{active})$. When the number of active queues increases, $Th$ decreases. In asymmetric topologies, egress ports can have different link speeds; as a result, we calculate a different pause threshold for every egress based on its speed. Similarly, ingress ports can have different 1-Hop RTTs. Since a queue can have packets from different ingresses, we use the max of $HRTT$ across all the ingresses to calculate $Th$. We use a pre-configured match-action table indexed with $N_{active}$ and $\mu$ to compute $Th$.

BFC does not guarantee that a flow will never run out of packets due to pausing. First, a flow can be paused unnecessarily if it is sharing its upstream queue with other paused flows. Second, a switch only resumes an upstream queue once all its packets (that exceeded the pause threshold when they arrived) have left the downstream switch. Since the resume takes an $HRTT$ to take effect, a flow can run out of packets at the downstream switch for an $HRTT$, potentially hurting utilization. However, this scenario is unlikely — a pause only occurs when a queue builds up, typically because multiple flows are competing for the same egress port. In this case, the other flows at the egress will have packets to occupy the link, preventing under-utilization.

We might reduce the (small) chance of under-utilization by resuming the upstream queue earlier, for example, when a flow's queue at the downstream drops below $Th$, or more precisely, when *every* queue (with a flow from the same upstream queue) drops below $Th$. Achieving this would require extra bookkeeping, complicating the design.

Increasing the pause threshold would reduce the number of pause/resumes generated, but only at the expense of increased buffering (Fig. 3-7). In Appendix C, we analyze the impact of $Th$ on under-utilization and peak buffer occupancy in a simple model, and we show that a flow runs out of packets at most 20% of the time when $Th$ is set to 1-hop BDP. Our evaluation results show that BFC achieves much better throughput than this worst case in practice (Table 3.1, §3.6).

**Sticky queue assignment:** Using `upstreamQ` for pausing flows poses a challenge. Since a switch does not know the current queue assignment of a flow at the upstream, it uses the `upstreamQ` conveyed by the last packet of the flow to pause a queue. However, if a flow runs out of packets at the upstream switch (e.g., because it was bottlenecked at the downstream switch but not the upstream), then its queue assignment may change for subsequent packets, causing it to temporarily evade the pause signal sent by the downstream switch. Such a flow will be paused again when the downstream receives packets with the new `upstreamQ`. The old queue will likewise be unpaused when its last packet (that exceeded $Th$) departs the downstream switch.

To reduce the impact of such queue assignment changes, we add a timestamp to the flow table state, updated whenever a packet is enqueued or dequeued. A new queue assignment only happens if the `size` value in the flow table is 0, and the timestamp is older than a "sticky threshold" (i.e., the entry in the flow table has had no packets in the switch for at least this threshold). Since with BFC's backpressure mechanism a flow can run out of packets for an $HRTT$, we set the sticky threshold to a small multiple of $HRTT$ (2 $HRTT$).

While sticky queue assignments reduce the chance that a backlogged flow will change queues, it doesn't completely eliminate it (e.g., packets from the same flow may arrive slower than this interval due to an earlier bottleneck). Such situations are rare, and we found that BFC performs nearly identically to an ideal (but impractical) variant that pauses flows directly using the flow ID without sticky queue assignments.

## 3.4 Tofino2 Implementation

We implemented BFC in Tofino2, a to-be-released P4-based programmable switch ASIC with a Reconfigurable Match Table (RMT) architecture [29]. A packet in Tofino2 first traverses the ingress pipeline, followed by the traffic manager (TM) and finally the egress pipeline. Tofino2 has four ingress and four egress RMT pipelines. Each pipeline has multiple stages, each capable of doing stateful packet operations. Ingress/egress ports are statically assigned to pipelines.

*Bookkeeping:* The flow table and pause counter are both maintained in the ingress pipeline. The flow table contains three values for each entry and is thus implemented as three separate register arrays (one for each value), updated one after the other.

*Multiple pipelines:* The flow table is *split* across the four ingress pipelines, and the size of the table in each ingress pipeline is 25 $\times$ the number of queues. During normal operation, packets of an active flow arrive at a single ingress pipeline (same ingress port). Since the state for a flow only needs to be accessed in a single pipeline, we can split the flow table. However, splitting can marginally increase collisions if the incoming flows are distributed unevenly among the ingress pipelines. Similarly, the pause counter is split among the ingress pipelines. An ingress pipeline contains the pause counter entries corresponding to its own ingress ports.

*Gathering queue depth information:* We need queue depth information in the ingress pipeline for pausing and dynamic queue assignment. Tofino2 has an inbuilt feature tailored for this task. The TM can communicate the queue depth information for all the queues in the switch to all the ingress pipelines without consuming any additional ingress cycles or bandwidth. The bitmap of empty queues is periodically updated with this data, with a different rotating starting point per pipeline to avoid new flows from being assigned to the same empty queue.

*Communicating from egress to ingress pipeline:* The enqueue operations described earlier are executed in the ingress pipeline when a packet arrives. Dequeue operations should happen at the egress but the bookkeeping data structures are at the ingress. To solve this, in the egress pipeline, we mirror packets as they exit and recirculate the

header of the mirrored packet back to the ingress pipeline it came from. The dequeue operations are executed on the recirculated packet header.

Recirculating packets involves two constraints. First, the switch has dedicated internal links for recirculation, but the recirculation bandwidth is limited to 12% of the entire switch capacity. Second, the recirculated packet consumes an additional ingress cycle. The switch has a cap on the number of packets it can process every second (pps capacity).

Most workloads have an average packet size greater than 500 bytes [27], and Tofino2 is designed with enough spare capacity in bandwidth and pps to handle header recirculation for every packet for those workloads (with room to spare). If the average packet size is much smaller, we can reduce recirculations by sampling packets for recirculation (described in Appendix E).

Recirculation is not fundamental to BFC. For example, Tofino2 has native support for PFC bookkeeping in the TM. Likewise, if BFC bookkeeping was implemented in the TM, it would not need recirculation. Similarly, in switches with a disaggregated RMT architecture [39] where the same memory can be accessed at both the ingress and egress, there is no need for recirculation.

## 3.5 Discussion

**Guaranteed losslessness.** BFC does not guarantee losslessness. In particular, a switch in BFC pauses an `upstreamQ` only after receiving a packet from it. This implies an `upstreamQ` can send packets for up to an $HRTT$ to the bottleneck switch before being paused, even if the switch is congested. In certain mass incast scenarios, this might be sufficient to trigger drops. Using credits [19, 91] could address this at the cost of added complexity. We leave an investigation of such prospective variants of BFC to future work. In our evaluation with realistic switch buffer sizes, BFC never incurred drops except under a 2000-to-1 incast (§3.6.3) and even then only 0.007% of the packets were dropped.

**Deadlocks:** Pushback mechanisms like PFC have been shown to be vulnerable

to deadlocks in the presence of cyclic buffer dependencies (CBD) or misbehaving NICs [70, 62]. BFC NICs do not generate any backpressure and as a result cannot cause deadlocks. Since NICs always drain, in the absence of CBD, BFC cannot have deadlocks (see Appendix D for a formal proof). A downstream switch in BFC *will* resume an `upstreamQ` if it drains all the packets sent by the `upstreamQ`. If a downstream is not deadlocked, it will eventually drain packets from the upstream, and as a result, the corresponding upstream cannot be deadlocked.

To prevent CBD, we can reuse prior approaches for deadlock prevention. These approaches can be classified into two categories. The first is to redesign routing protocols to avoid installing routes that might cause CBD [129, 128]. The other is to identify a subset of possible ingress/egress pairs that are provably CBD free, and only send pause/resume along those pairs [71]. For a fat-tree topology, this would allow up-down paths but not temporary loops or detour routes [97]. In BFC, we use the latter approach. Given a topology, we pre-compute a match action table indexed by the ingress and egress port, and simply elide the backpressure pause/resume signal if it is disallowed. See Appendix D for details.

**Incremental deployment:** In a full deployment, BFC would not require end-to-end congestion control. In a partial deployment, we advocate some form of end-to-end congestion control, such as capping the number of inflight packets of a flow. A common upgrade strategy is to upgrade switches more rapidly than server NICs. If only switches and not NICs are running BFC, capping inflight packets prevents a source NIC from overrunning the buffers of the first hop switch. The same strategy can be used for upgrading one cluster's switches before the rest of the data center [149]. In our evaluation, we show incremental deployment would have some impact on buffer occupancy at the edge but minimal impact on performance (Appendix E).

## 3.6 Evaluation

We present a proof-of-concept evaluation of our Tofino2 implementation. To compare performance of BFC against existing schemes, we perform large scale ns-3 [5]

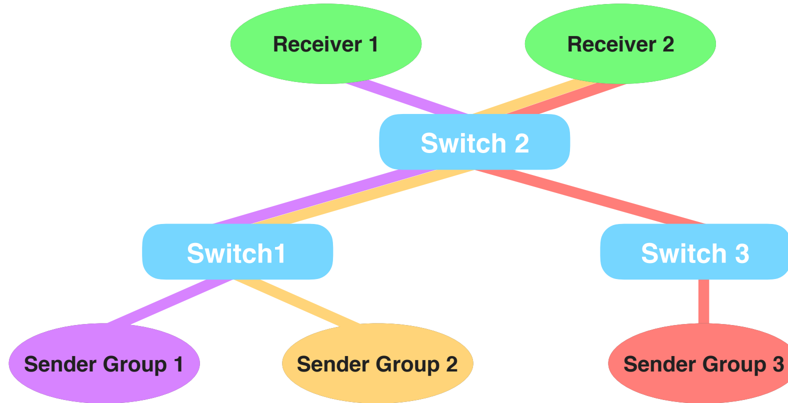



Figure 3-6: **Testbed topology** — The colored lines show the path for different flow groups.

simulations.

### 3.6.1 Tofino2 Evaluation

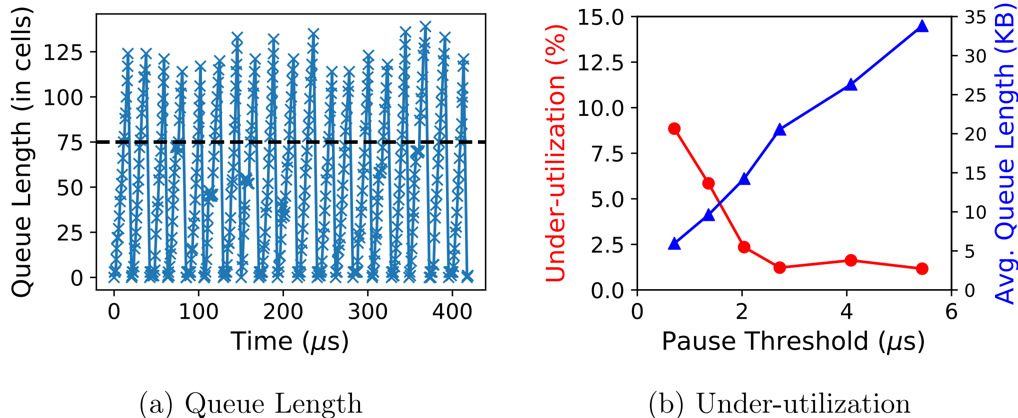
**Testbed:** For evaluation, we were able to gain remote access to a Tofino2 switch. Using a single switch, we created a simple multi-switch topology (Fig. 3-6) by looping back packets from the egress port back into the switch. All the ports are 100 Gbps, each port has 16 queues.[5] The experiments include three groups of flows.

- Sender Group 1 → Switch 1 → Switch 2 → Receiver 1.
- Sender Group 2 → Switch 1 → Switch 2 → Receiver 2.
- Sender Group 3 → Switch 3 → Switch 2 → Receiver 2.

To generate traffic we use the on-chip packet generator with no end-to-end congestion control.

**Low buffering, high utilization:** Fig. 3-7a shows the queue length for a flow when two flows are competing at a link (a group 2 flow is competing with a group 3 flow at the switch 2 → receiver 2 link). The pause threshold is shown as a horizontal black line. BFC's pausing mechanism is able to limit the queue length near the pause threshold ($Th$). The overshoot from $Th$ is for two reasons. First, it takes an $HRTT$

[5]For 100 Gbps ports, Tofino2 has 32 queues, but in loopback mode only 16 queues are available.




(a) Queue Length

(b) Under-utilization

Figure 3-7: **Queue length and under-utilization** — 2 flows are competing at a 100 Gbps link. Cell size is 176 bytes. BFC achieves high utilization and low buffering.

for the pause to take effect. Second, Tofino2 has small hardware queues after the egress pipeline, and a pause from the downstream cannot pause packets already in these hardware queues.

Notice that the queue length goes to 0 temporarily. Recall that a downstream switch only resumes the `upstreamQ` when it has drained all the packets from the `upstreamQ` that exceeded $Th$. As a result, a flow at the downstream can run out of packets for an $HRTT$. This can cause under-utilization when the queues for the two flows go empty simultaneously. We repeat the above experiment but vary the pause threshold. Fig. 3-7b shows the average queue length and the under-utilization of the congested link. With a pause threshold of 2 $\mu$s, BFC achieves close to 100% utilization with an average queue length of 15 KB.

**Queue assignment and congestion spreading:** We next evaluate the impact of queue assignment on HoL blocking and performance. We evaluate three different queue assignment strategies with BFC's backpressure mechanism: (1) "BFC + single": All flows are assigned to a single queue (similar to PFC); (2) "BFC + stochastic": Flows are assigned to queues using stochastic hashing; (3) "BFC + dynamic": Dynamic queue assignment as described in §3.3.3.1.

The setup consists of two group 1 flows, eight group 3 flows, and a number of group 2 flows varied between four to twenty. All flows are 1.5 MB in size. The experiment is designed such that for group 2 and 3 flows, the bottleneck is the switch 2 $\rightarrow$ receiver

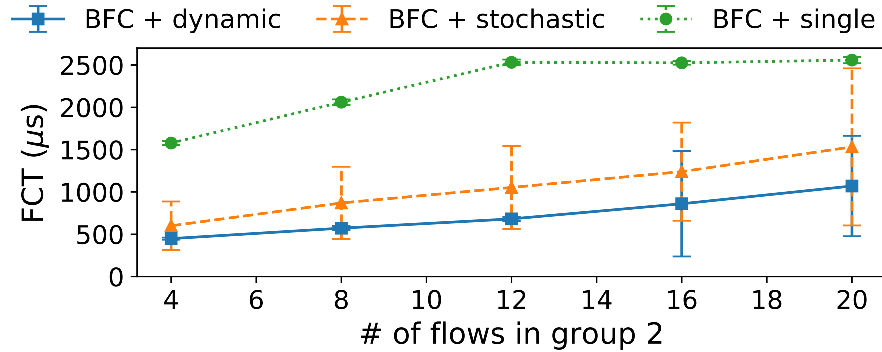



Figure 3-8: **Congestion spreading** — Dynamic queue assignment reduces HoL blocking, improving FCTs on average and at the tail.

2 link. The bottleneck for group 1 flows is the switch 1 $\rightarrow$ switch 2 link. Switch 2 will pause queues at switch 1 in response to congestion from group 2 flows. Notice that group 1 and group 2 flows are sharing the switch 1 $\rightarrow$ switch 2 link. If a group 1 flow shares a queue with a group 2 flow (a collision), the backpressure due to the group 2 flow can slow down the group 1 flow, causing HoL blocking and increasing its flow completion time (FCT) unnecessarily.

Fig. 3-8 shows the average FCT for group 1 flows across four runs. The whiskers correspond to one standard deviation in the FCT. BFC + single achieves the worst FCT as group 1 and 2 flows always share a queue. With stochastic assignment, the FCT is substantially lower, but the standard deviation in FCT is high. In some runs, group 1 and 2 flows don't share a queue and there is no HoL blocking. In other runs, due to the stochastic nature of assignment, they do share a queue (even when there are other empty queues), resulting in worse performance. With dynamic assignment, BFC achieves the lowest average FCT and the best tail performance. In particular, the standard deviation is close to 0 when the number of flows at the switch 1 $\rightarrow$ switch 2 link (group 1 + group 2 flows) is lower than the number of queues. In such scenarios, group 1 flows consistently incur no collisions. When the number of flows exceed the queues, collisions are inevitable, and the standard deviation in FCT increases.

### 3.6.2 Simulation-based Evaluation

We also implemented BFC in ns-3 [5]. For DCQCN we use [6], for ExpressPass we use [2], and for all other schemes we use [4].

#### 3.6.2.1 Setup

**Network Topology:** We use a Clos topology with 128 leaf servers, 8 top of the rack (ToR) switches and 8 Spine switches (2:1 over subscription). Each Spine switch is connected to all the ToR switches, each ToR has 16 servers, and each server is connected to a single ToR. All links are 100 Gbps with a propagation delay of 1 us. The maximum end-to-end base round trip time (RTT) is 8 $\mu$s and the 1-Hop RTT is 2 $\mu$s. The switch buffer size is set to 12 MB. Relative to the ToR switch capacity of 2.4 Tbps, the ratio of buffer size to switch capacity is 40 $\mu$s, the same as Broadcom's Tomahawk3 from Fig. 3-1. We use an MTU of 1 KB. Unless specified otherwise, we use Go-Back-N for retransmission, flow-level ECMP for load balancing, and the standard shared buffer memory model implemented in existing switches [33].

**Comparisons:** *HPCC:* HPCC uses explicit link utilization information from the switches to reduce buffer occupancy and drops/PFCs at the congested switch. We use the parameters from the paper, $\eta = 0.95$ and $maxStage = 5$. The dynamic PFC threshold is set to trigger when traffic from an input port occupies more than 11% of the free buffer (as in the HPCC paper). We use the same PFC thresholds for DCQCN and DCTCP.

*HPCC-PFC:* This version replaces PFC with perfect retransmission. On a packet drop, the switch informs the sender directly, which then retransmits the dropped packet. We choose this (potentially impractical) strategy to provide a bound on the performance that can be achieved using any retransmission scheme.

*DCQCN:* DCQCN uses ECN bits and end-to-end control to manage buffer use at the congested switch. The ECN threshold triggers before PFC ($K_{min}$ = 100KB and $K_{max}$ = 400KB).

*DCTCP:* The ECN threshold is same as DCQCN. Flows start at line rate to avoid

degradation in FCTs from slow-start.

*ExpressPass:* In ExpressPass, senders transmit data based on credits generated by the receiver. These credits are rate-limited at the switches to avoid congestion. We chose $\alpha = 0.5, w_{init} = 0.0625$ and a credit buffer size of 16 credits. The ExpressPass simulator does not follow a shared buffer model; instead it assumes dedicated per-port buffers. To eliminate drops, we supplied a high per-port buffer value of 75 MB. There is no PFC.

*BFC:* We use 32 physical queues per port (consistent with Tofino2) and our flow table has 76K entries. The flow table takes 400 KB of memory. We chose per-flow fair queuing as our scheduling mechanism; all the comparison schemes strive for per-flow fairness, thus, fair queuing provides for a just comparison.

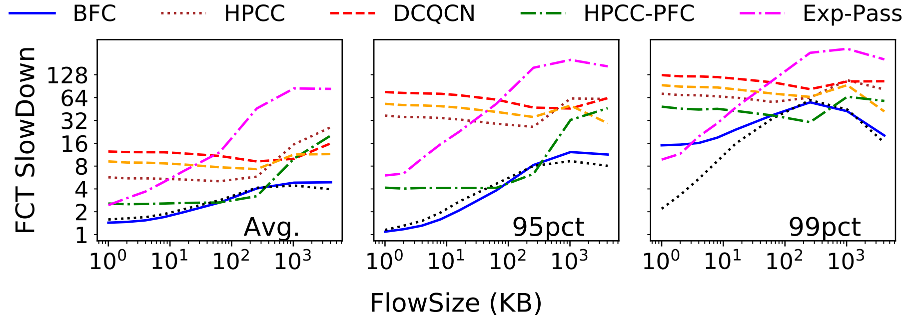
*Ideal-FQ:* To understand how close BFC comes to optimal performance, we simulate ideal fair queuing with infinite buffering at each switch. The NICs cap the in-flight packets of a flow to 1 BDP. Note that infinite buffering is not realizable in practice; its role is to bound how well we could possibly do.

**Sensitivity to parameters:** All systems were configured to achieve full throughput for a single flow on an unloaded network. For end-to-end schemes, the choice of parameters governs the trade-off between the performance of short flows (through reduced queuing) and long flows (higher link utilization). We perform parameter sensitivity analysis for HPCC, DCTCP and ExpressPass in Appendix F.2.

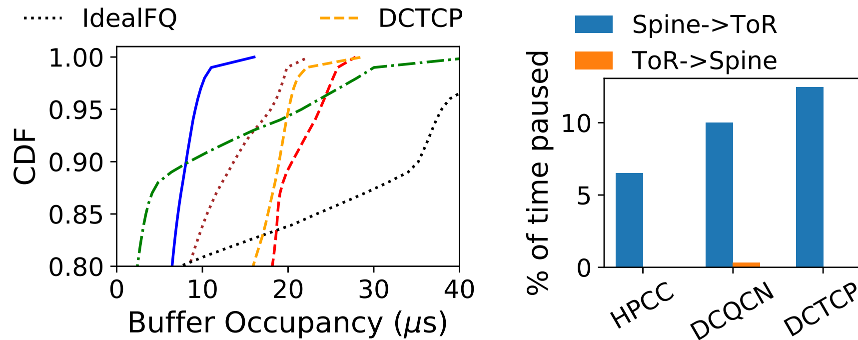
**Performance metrics:** We consider three performance metrics: (1) FCT normalized to the best possible FCT for the same size flow, running at link rate (referred as the FCT slowdown); (2) Overall buffer occupancy at the switch; (3) Throughput of individual flows.

**Workloads:** We synthesized a trace to match the flow size distributions from the industry workloads discussed in Fig. 3-2: (1) Aggregated workload from all applications in a Google data center; (2) a Hadoop cluster at Facebook (FB_Hadoop). The flow arrival pattern is open-loop and follows a *bursty* log-normal inter-arrival time distribution with $\sigma = 2$.[6] For each flow arrival, the source-destination pair is

[6]Most prior work evaluates using Poisson flow arrivals [38, 107], but we use the more bursty




(a) FCT




(b) Buffer occupancy (c) PFC Time

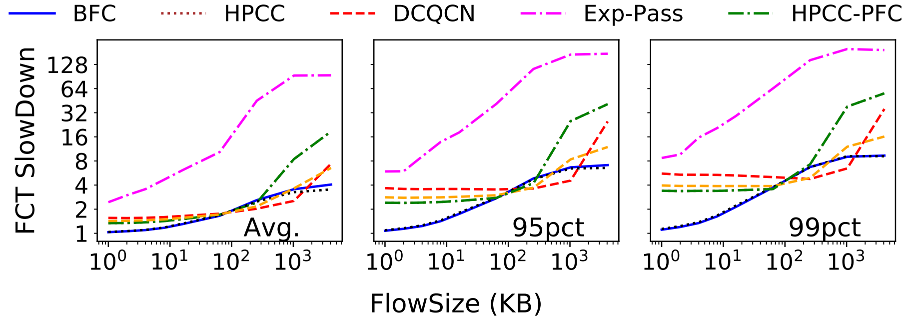
Figure 3-9: Google distribution with 55% load + 5% 100-1 incast. BFC tracks the ideal behavior, improves FCTs, and reduces buffer occupancy. For FCT slowdown, both the x and y axis are log scaled.

derived from a uniform distribution. We consider scenarios with and without incast, different traffic load settings, and incast ratios. Since our topology is oversubscribed, on average links in the core (Spine-ToR) will be more congested than the ToR-leaf server links. In our experiments, by X% load we mean X% load on the links in the core.

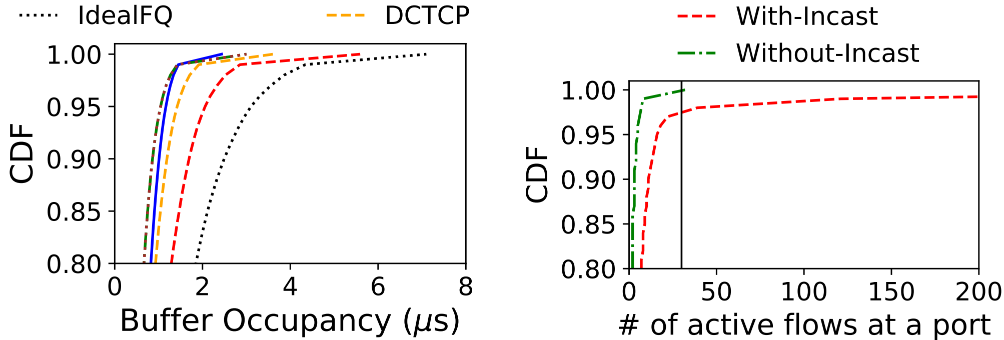
#### 3.6.2.2 Performance

Fig. 3-9 and 3-10 show our principal results. The flow sizes are drawn from the Google distribution and the average load is set to 60% of the network capacity. For Fig. 3-9 (but not Fig. 3-10), 5% of the traffic (on average) is from incast flows. The incast degree is 100-to-1 and the size is 20 MB in aggregate. A new incast event starts every 500 $\mu$s. Since the best-case completion time for an incast is 1.6

Lognormal as it provides a more challenging case for BFC.




(a) FCT




(b) Buffer Occupancy

(c) Active Flows

Figure 3-10: FCT slowdown and buffer occupancy for Google distribution with 60% load. For all the schemes, PFC was never triggered. Part (c) shows the CDF of active flows at a port with and without incast, with the vertical bar showing the total number of queues per port.

ms (20 MB/100 Gbps), multiple incasts coexist simultaneously in the network. We report the FCT slowdowns at the average, $95^{\text{th}}$ and $99^{\text{th}}$ percentile, the tail buffer occupancy (except for ExpressPass simulations which do not follow the shared buffer model), and the fraction of time links were paused due to PFC. We report the FCT slowdowns for the incast traffic separately in Appendix F.4.

Out of all the schemes, DCQCN is worst on latency for small flow sizes, both at the average and the tail. Compared to DCQCN, DCTCP improves latency as it uses per-ACK feedback instead of periodic feedback via QCN. However, the frequent feedback is not enough, and the performance is far from optimal (Ideal-FQ). The problem is that both DCQCN and DCTCP are slow in responding to congestion. Since flows start at line rate, a flow can build up an entire end-to-end bandwidth-delay product (BDP) of buffering (100 KB) at the bottleneck before there is any possibility of reducing its rate. The problem is aggravated during incast events. The

bottleneck switch can potentially accumulate one BDP of packets per incast flow (10 MB in aggregate for 100-to-1 incast).

Both protocols have low throughput for long flows. When capacity becomes available, a long flow may fail to ramp up quickly enough, reducing throughput and shifting its work to busier periods where it can impact other flows. Moreover, on sudden onset of congestion, a flow may not reduce its rate fast enough, slowing short flows.

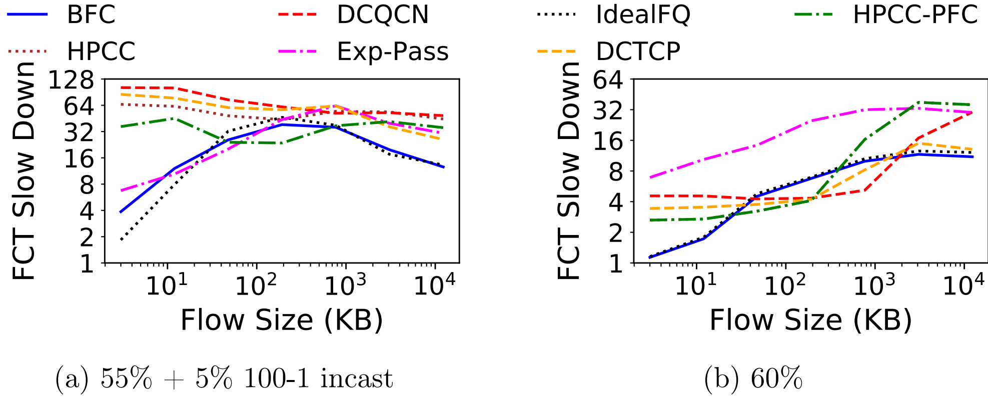
HPCC improves on DCQCN and DCTCP by using link utilization instead of ECN and a better control algorithm. Compared to DCQCN and DCTCP, HPCC reduces tail latency, tail buffer occupancy, and PFC pauses (in case of incast). Compared to BFC, however, HPCC has 5-30× worse tail latency for short flows with incast, and 2.3-3× worse without. Long flows do worse with HPCC than DCQCN and DCTCP since HPCC deliberately targets 95% utilization and very small queues to improve tail latency for short flows.

With ideal retransmission, HPCC performance improves, especially for short and medium flows. However, HPCC without PFC has higher tail buffer occupancy and suffers packet loss. Compared to BFC, overall performance is still worse for both long and short flows.

Across all systems, ExpressPass achieves the worst throughput for long flows. In ExpressPass, the receiver can generate unnecessary credits for an additional RTT before learning that a flow is finished. These credits are considered "wasted" as the sender cannot transmit packets in response, and can therefore cause link under-utilization. Credit waste and the corresponding under-utilization increase with faster link speeds and/or when the flow sizes get shorter (see §6.3 and §7 in [38]).

Ideal-FQ achieves lower latency than all the schemes, but its buffer occupancy can grow to an unfeasible level.

BFC achieves the best FCTs (both average and tail) among all the schemes. Without incast, BFC performance closely tracks optimal. With incast, incoming flows exhaust the number of physical queues, triggering HoL blocking and hurting tail latency. This effect is largest for the smallest flows at the tail. Fig. 3-10c shows the CDF of the number of active flows at a port. In the absence of incast, the number



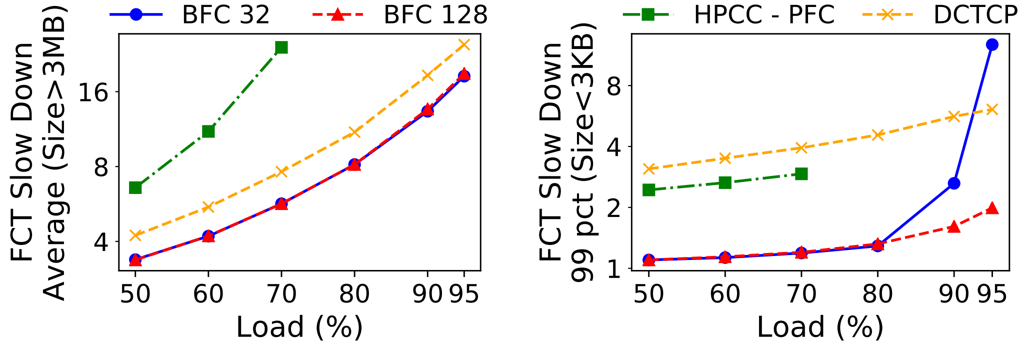

(a) 55% + 5% 100-1 incast (b) 60%

Figure 3-11: FCT slowdown ($99^{th}$ percentile) for Facebook distribution with and without incast.

of active flows is smaller than the total queues 99% of the time, and collisions are rare. With incast, the number of active flows increases, causing collisions. However, the tail latency for short flows with BFC is still 5-30× better than existing schemes. BFC also improves the performance of incast flows, achieving 2× better FCTs at the tail compared to HPCC (see Appendix F.4).

Note that, compared to BFC and Ideal-FQ, latency for medium flows (200-1000KB) is slightly better with existing schemes. Because they slow down long flows relative to perfect fairness, medium flows have room to get through more quickly. Conversely, tail slowdown is better for long flows than medium flows with BFC and Ideal-FQ. Long flows achieve close to the long term average available bandwidth, while medium flows are more affected by transient congestion.

**Another workload:** We repeated the experiment in Fig. 3-9 and Fig. 3-10 with the Facebook distribution. Fig. 3-11 shows the $99^{th}$ percentile FCT slowdown. The trends in the FCT slowdowns are similar to that of the Google distribution, except that ExpressPass performs better since it incurs fewer wasted credits (as a percentage) for the Facebook workload, which has larger flows. We omit other statistics presented earlier in the interest of space, but the trends are similar to Fig. 3-9 and 3-10. Henceforth, all the experiments use the Facebook workload.




(a) Average FCT for long flows

(b) Tail FCT for short flows

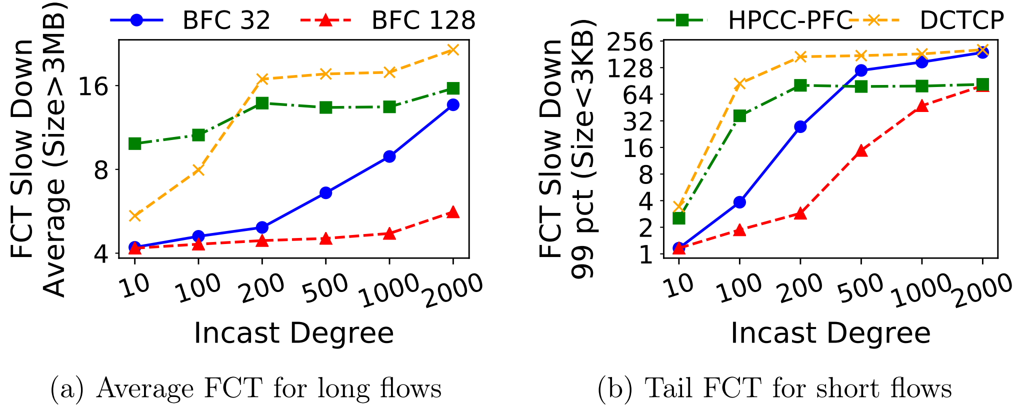
Figure 3-12: Average FCT slowdown for long flows, and $99^{th}$ percentile tail FCT slowdown for small flows, as a function of load.

### 3.6.3 Stress-testing BFC

In this section we stress-test BFC under high load and large incast degree. Flow arrivals follow a bursty log-normal distribution ($\sigma = 2$). We evaluate BFC under two different queue configurations: (1) 32 queues per port (BFC 32); (2) 128 queues per port (BFC 128). We show the average slowdown for long flows ($>$ 3MB) and $99^{th}$ percentile slowdown for short flows ($<$ 3KB).

**Load:** Fig. 3-12 shows the performance as we vary the average load from 50 to 95% (without incast). HPCC only supports loads up to 70%. At higher loads, it becomes unstable (the number of outstanding flows grows without bound), in part due to the overhead of the INT header (80 B per-packet). All other schemes were stable across all load values.

At loads $\leq$ 80%, BFC 32 achieves both lower tail latency (Fig. 3-12b) for short flows and higher throughput for long flows (Fig. 3-12a). The tail latency for short flows is close to the perfect value of 1. At higher loads, flows remain queued at the bottleneck switch for longer periods of time, raising the likelihood that we run out of physical queues, leading to head of line blocking. This particularly hurts tail performance for short flows as they might be delayed for an extended period if they are assigned to the same queue as a long flow. At the very high load of 95%, the HoL blocking degrades tail latency substantially for BFC 32. However, it still achieves good link utilization, and the impact of collisions is limited for long flows.




(a) Average FCT for long flows

(b) Tail FCT for short flows

Figure 3-13: Average FCT slowdown for long flows, and $99^{th}$ percentile tail FCT slowdown for small flows, as a function of incast degree.

Increasing the number of queues reduces collisions and the associated HoL blocking. BFC 128 achieves better tail latency for short flows at load $\geq 90\%$.

**Incast degree:** If the size of an incast is large enough, it can exhaust physical queues and hurt performance. Fig. 3-13 shows the effect of varying the degree of incast on performance. The average load is 60% and includes a 5% incast. The incast size is 20 MB in aggregate, but we vary the degree of incast from 10 to 2000.

For throughput, both BFC 32 and BFC 128 perform well as long as the incast degree is moderate compared to the number of queues. Both start to degrade when the incast degree exceeds 8× the number of queues per port. Till this point, BFC can leverage the FanIn from the larger number of upstream queues (and greater aggregate upstream buffer space) to keep the incast from impeding unrelated traffic. As the incast degree scales up further, BFC 32 is able to retain some of its advantage relative to HPCC and DCTCP.

For high incast degree, the tail latency for short flows becomes worse than HPCC. The tail is skewed by the few percent of small requests that happen to go to the same destination as the incast. (Across the 128 leaf servers in our setup, several servers are the target of an incast at any one time, and these also receive their share of normal traffic.) As the incast degree increases, more small flows share physical queues with incast flows, leading to more HoL blocking.

### 3.6.4 Understanding the Limits of BFC

This section further investigates the impact of large numbers of active flows on BFC's performance through controlled microbenchmarks. We also show that adding a simple end-to-end flow control mechanism on top of pure BFC helps alleviate the performance impairments caused by large numbers of flows.

Collisions hurt performance in two ways. Consider a congested port $X$. First, at $X$, the packets of a short flow can get stuck behind the packets of a long flow sharing the same queue, increasing the FCT. Such performance degradation occurs when the number of active flows exceeds the number of queues at $X$. Second, $X$ can pause an upstream queue. Unrelated flows sharing this upstream queue will get paused even though they are not going through the congested port $X$ (congestion spreading). BFC can leverage the larger number of upstream queues at the upstream switches to limit congestion spreading (§3.3.3.1). Typically, congestion spreads only once the number of flows at the congested port exceeds the total number of upstream queues. As a result, in larger topologies with more upstream switches, congestion spreading is harder to create.

To illustrate these issues, we conduct experiments on our standard topology (§3.6.2.1) where we create different numbers of long-running elephant flows destined to the same receiver (Receiver $A$). All elephant flows start at the beginning of the experiment. We then create two groups of short flows: (1) destined to the same receiver $A$ (referred as "direct" mice flows), and (2) destined to a different receiver $B$ in the same rack as receiver $A$ (referred to as "indirect" mice flows). The aggregate load for each group of mice flows is 3% of the link capacity, and the size of the mice flows is 1 KB. Fig. 3-14 shows the median FCT slowdown for mice flows as we vary the number of long-running flows. We show results for BFC with 32 and 128 queues, and also IdealFQ (described in §3.6.2.1) for reference. As expected, for direct mice flows, the FCT degrades when the number of long-running flows exceeds the number of queues. For indirect flows, the degradation only happens when long flows exceed $8\times$ the number of queues, since the topology has 8 spine switches connected to each

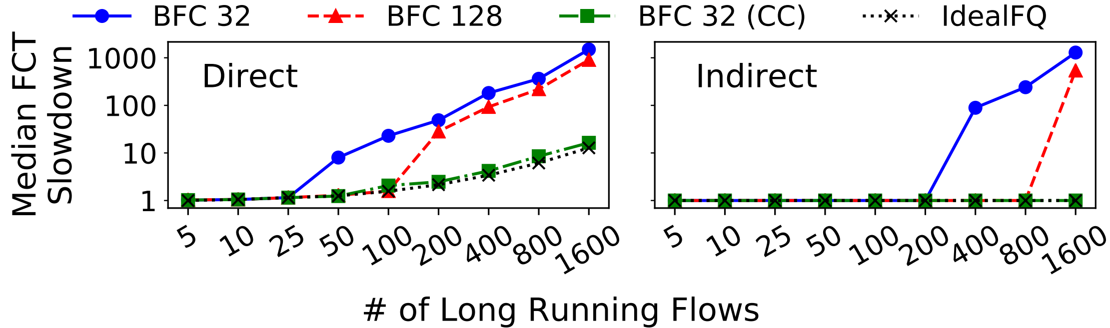



Figure 3-14: Median FCT slowdown for mice flows in the presence of long-running flows.

ToR switch. In this case, some indirect mice flows get paused unnecessarily because they share an upstream queue with a paused long-running flow.

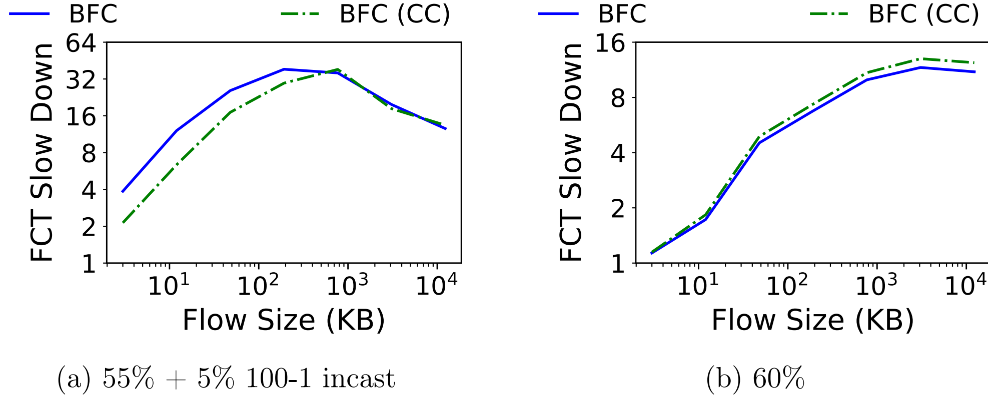
**Combining end-to-end congestion control with BFC:** In the previous experiment, each long-running flow can build up to 1 Hop-BDP of buffering before getting paused. With $N$ long-running flows, in the worst case, a mice flow experiencing a collision can get stuck behind $N\times$ 1-Hop BDP of buffering. BFC can use a simple end-to-end congestion control mechanism to reduce this buffering and limit HoL blocking. This mechanism is helpful in scenarios with persistently large numbers of active flows. As our evaluations showed (§3.6.3), even in workloads with high load and occasional large-scale incast, pure BFC (with no end-to-end control) performs well except in extreme cases.

Augmenting BFC with end-to-end control is simple. The main goal of the end-to-end control is to prevent flows from sending an excessively large number of packets into the network. Importantly, the end-to-end mechanism need not try to accurately control queuing, react quickly to bursts, or achieve fairness — typical requirements for low-latency data center congestion control protocols — since BFC already achieves these goals.

As an example, we implemented a simple delay-based congestion control that tries to maintain the end-to-end RTT at a certain threshold ($\texttt{RTT}_{Target}$). We chose a high $\texttt{RTT}_{Target}$ value of 2.5× base RTT to avoid hurting the throughput of long flows, exploiting the fact that it isn't necessary to tightly control queuing in BFC. The sender's window ($w$) is adjusted following Algorithm 2.

$\texttt{RTT}_{Target} = 2.5\times$ Base RTT;
$w$ =1 BDP;
**for** *each Acknowledgement* **do**
  **if** $RTT > \texttt{RTT}_{Target}$ **then**
    $w = w - \frac{\texttt{RTT - RTT}_{Target}}{RTT}$
  **else**
    $w = w + \frac{\texttt{RTT}_{Target} - RTT}{RTT}$

**Algorithm 2:** Simple end-to-end congestion control

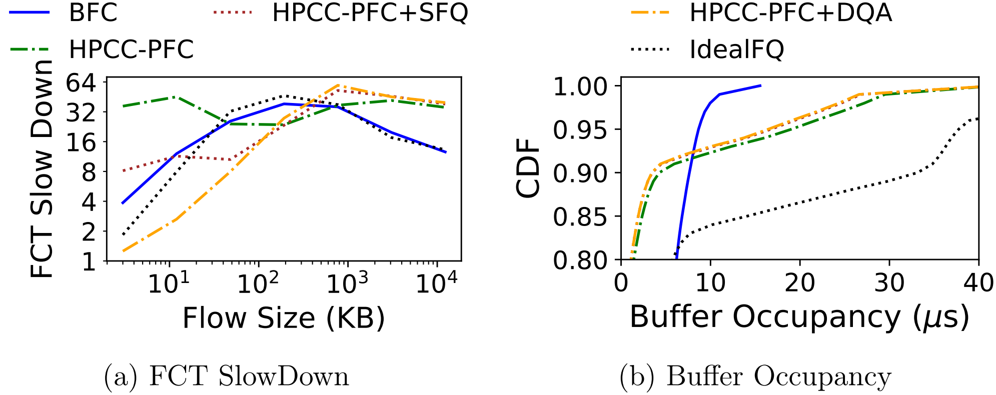



(a) 55% + 5% 100-1 incast (b) 60%

Figure 3-15: $99^{th}$ percentile FCT slowdown when combined with congestion control. Facebook workload, same setup as Fig. 3-11.

With the above rule, the window of a sender roughly goes from $w \rightarrow w \times \frac{\texttt{RTT}_{Target}}{RTT}$ within an RTT. Fig. 3-14 shows the performance with this variant (BFC 32 (CC)). The performance is close to IdealFQ in all the cases. To check if this change negatively affected the overall behavior of BFC, we repeat the principle experiment in Fig. 3-11 (Facebook workload) with BFC 32 (CC). Fig. 3-15 shows the $99^{th}$ percentile FCT slowdowns. The FCTs of long flows are similar to that of the original BFC (within 10%). However, in the presence of incast, adding congestion control improves the $99^{th}$ percentile FCT of short flows and the peak buffer occupancy by 30%. While using end-to-end congestion control can improve performance under frequent collisions (and we advocate supplementing BFC with such a mechanism in practice), in this dissertation we focus on BFC without any such mechanism to better understand the core benefits and limitations of BFC in its purest form.

In Appendix F.3, we experiment with a variant of BFC where the sender labels incast flows explicitly (similar to the potential optimization in [107]). All the incast




(a) FCT SlowDown (b) Buffer Occupancy

Figure 3-16: FCT slowdown ($99^{th}$ percentile) and buffer occupancy of HPCC variants, using the setup in Fig. 3-11a.

flows at an egress port are assigned to the same queue. This frees up queues for non-incast traffic and reduces collisions substantially under large incasts.

### 3.6.5 Dynamic Queue Assignment

We next consider the effect of applying BFC's dynamic queue assignment separately from the backpressure mechanism. For this, we modified HPCC with idealized retransmission (HPCC-PFC) to add stochastic fair queuing (HPCC-PFC+SFQ) and dynamic queue assignment (HPCC-PFC+DQA). To match BFC, we use 32 physical queues with HPCC. We repeat the experiment from Fig. 3-11a, showing tail slowdown and buffer occupancy for the HPCC variants, BFC, and IdealFQ in Fig. 3-16.

Adding SFQ to HPCC improves short flow latency by isolating them from long flows in different queues, but it still suffers from more collisions (and thus higher tail latency for short flows) than DQA. DQA on its own, however, has no benefit for long flows: since HPCC is unable to adapt to rapid changes in the number of flows (and the fair-share rate), it is unable to fully utilize the link for long flows, even with DQA. Moreover, both HPCC-PFC+SFQ and HPCC-PFC+DQA build deep buffers and experience drops at the same rate as HPCC-PFC. Notice that HPCC's lower throughput for long flows favors short flows to such an extent that HPCC-PFC+DQA achieves better tail latency for short flows than both BFC and IdealFQ.

### 3.6.6 Additional Experiments

In this section, we present a more complete set of simulation results for BFC. We first summarize those results, and then present them.

**Priority scheduling:** Data center operators often classify traffic into multiple classes and use scheduling priorities to ensure performance for the most time-sensitive traffic. We repeat the experiment in Fig. 3-11b but with traffic split equally among four priority traffic classes, and show that BFC performs well in this case. See §3.6.6.1 for details.

**Spatial locality:** We repeat the experiment in Fig. 3-11 with spacial locality in source-destination pairs such that the average load on all links across the network is same. The trends in performance are similar. See §3.6.6.2 for details.

**Slow-start:** We evaluate the impact of using TCP slow-start instead of starting flows at line rate. We repeat the experiment in Fig. 3-11 and compare the original DCTCP with slow start (DCTCP + SS) and our modified DCTCP where flows start at the line rate. With incast, DCTCP + SS reduces buffer occupancy by reducing the intensity of incast flows, improving tail latency. However, it also increases median FCTs by up to 2×. Flows start at a lower rate, taking longer to ramp up to the desired rate. In the absence of incast, it increases both the tail and median FCT for short flows. See §3.6.6.3 for details.

**Performance in asymmetric topologies:** BFC makes no assumption about the topology, link speeds and link delays. We evaluate the performance of BFC in a multi-data-center topology. BFC achieves low FCT for flows within the data center, and high link utilization for the inter-data-center links (see §3.6.6.4).

**Dynamic vs. stochastic queue assignment in BFC:** We repeat the experiment in Fig. 3-11a but use stochastic hashing to statically assign flows to physical queue instead. With stochastic assignment, the number of collisions in physical queues increases, hurting FCTs (see §3.6.6.5).

**Size of flow table:** Reducing the size of the flow table can increase index collisions in the flow table, potentially hurting FCTs. We repeat the experiment in Fig. 3-11a

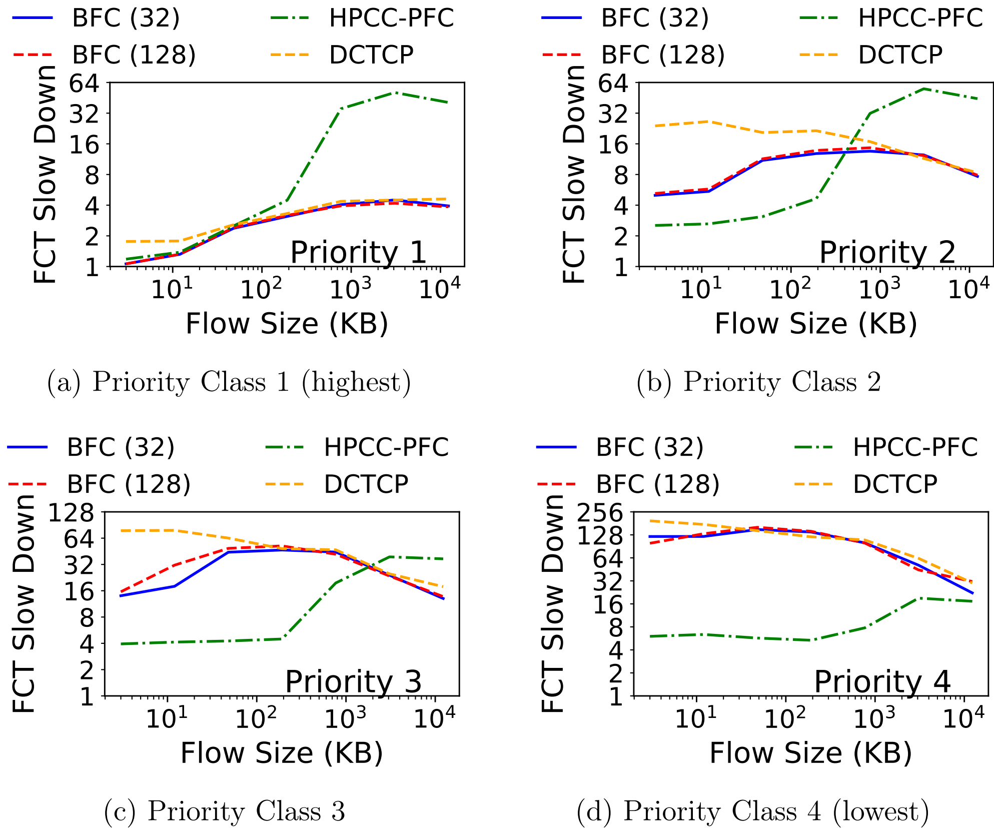
and evaluate the impact of size of flow table. Reducing the size partly impacts the short flow FCTs (see §3.6.6.6).

#### 3.6.6.1 Multiple Traffic Classes

Many data center operators allocate network traffic into a small number of priority traffic classes to ensure that mission critical traffic is delivered with low tail latency, while other traffic is delivered according to its quality of service needs. BFC has a simple extension to support priority groups. To avoid priority inversion where a flow at one priority can be stalled behind a flow of a lower priority, we assume queues at a port are statically assigned to different priority levels. The switch performs dynamic queue assignment for each class independently. A flow with priority $X$ is only assigned to physical queues associated with that priority. Queues at the same priority level follow fair scheduling.

Statically partitioning physical queues among traffic classes could make it more likely for traffic within a class to run out of queues and suffer degraded performance with collisions and HoL blocking. On the other hand, high priority traffic is preferentially scheduled, leading to short queues and few active flows. Collisions will be more likely at lower priority traffic classes, where performance is already degraded. Priority scheduling results in rapid and extreme changes in the available rate for these background classes. Relative to end-to-end control, per-hop backpressure can more easily utilize rapidly changing spare capacity.

To test how BFC behaves with multiple traffic classes, we repeat the experiment in Fig. 3-11b: Facebook workload, 60% load, and no incast. We configure the system with 4 priority classes, each with equal load (15% each, 60% in aggregate). We allocate physical queues evenly to each traffic class. We consider configurations with 32 and 128 queues per port (8 or 32 queues per class). We also show results for HPCC and DCTCP. In this study, DCTCP marks packets based on per-class queueing, while HPCC uses switch aggregates. Fig. 3-17 shows the $99^{th}$ percentile FCT slowdown for different priority classes. BFC achieves good performance across all traffic classes and flow sizes. In particular, BFC achieves up to 5× better tail latency for short flows




(a) Priority Class 1 (highest)

(b) Priority Class 2

(c) Priority Class 3

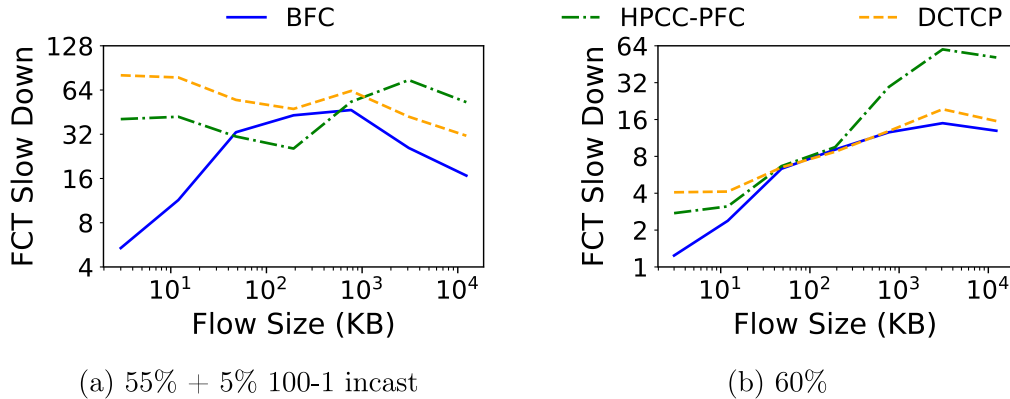
(d) Priority Class 4 (lowest)

Figure 3-17: Multiple traffic classes with BFC, reporting $99^{th}$ percentile FCT slowdown for the Facebook workload, 60% load, and no incast.

than DCTCP. At the lowest priority level, DCTCP's short flow tail latency converges to that of BFC. For low priority flows, tail latency is primarily governed by time spent waiting to be scheduled at the switch.

HPCC's performance is somewhat anomalous. Long flows suffer priority inversion, where long flows at high priority achieve significantly worse service than short flows at lower priority. In HPCC, long flows back off in an attempt to keep queues empty. The (transient) extra capacity left by such long flows can be used by short flows traffic at all priority levels, improving performance for these short flows.

BFC has only slightly better performance with 32 vs. 8 queues per priority level, indicating that collisions did not have much impact. For high priority traffic, the setup is equivalent to running our experiment with just one traffic class at 15% load and a small number of queues—even modest numbers of active queues are unlikely at such low load. Lower priority traffic can run out of queues, but they gain the




(a) 55% + 5% 100-1 incast

(b) 60%

Figure 3-18: Impact of spatial locality. FCT slowdown ($99^{th}$ percentile) for Facebook distribution with and without incast.

benefit of being able to take immediate advantage when the high priority queues are empty. In other words, work conserving behavior is more important for background traffic than the number of queues. We acknowledge this is just one study, and there are likely scenarios where BFC's performance could suffer when using multiple traffic classes.

One obvious improvement is to split queues dynamically among classes rather than statically. But in the long run, we strongly believe that the number of queues per port is likely to continue to grow to whatever is needed to deliver good performance.

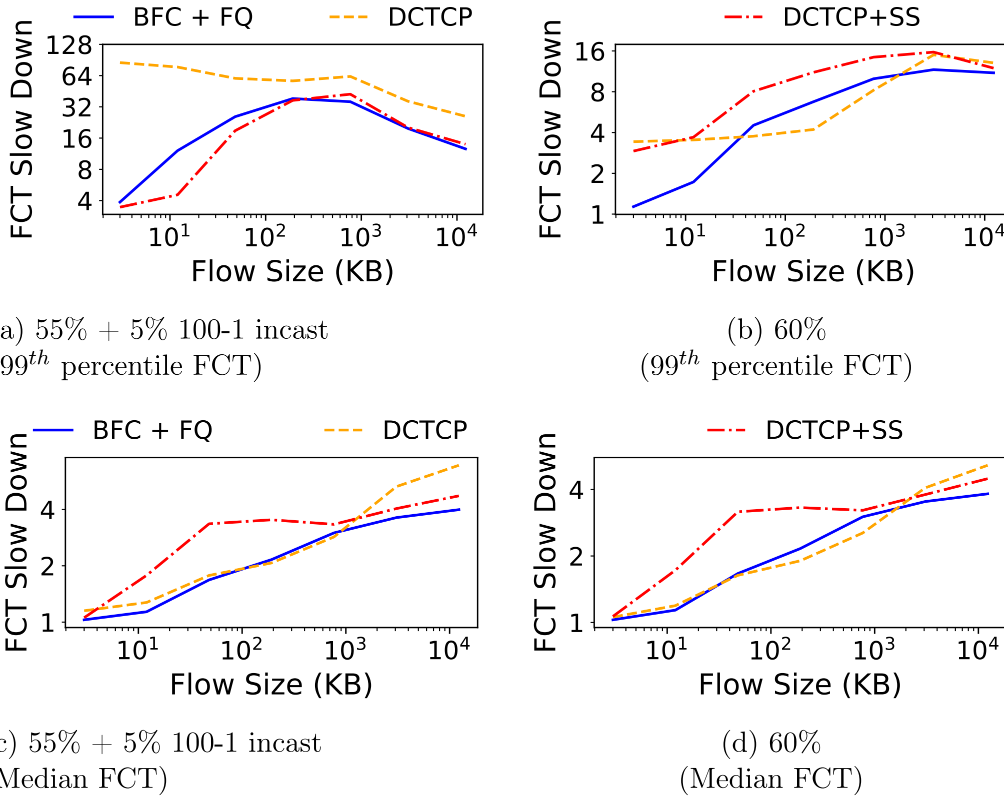
#### 3.6.6.2 Impact of Spatial Locality

We repeated the experiment from Fig. 3-11 with spatial locality in source-destination pairs such that the average load on all links across the network is same. Fig. 3-18 shows the $99^{th}$ percentile slowdowns. The trends are similar to Fig. 3-11.

#### 3.6.6.3 Using TCP Slow-start

We also evaluate the impact of using TCP slow-start instead of starting flows at line rate in Figure 3-19. We compare the original DCTCP with slow start (DCTCP + SS) with an initial window of 10 packets versus the modified DCTCP used so far (initial window of the BDP). The setup is same as Fig. 3-11.

With incast, DCTCP + SS reduces buffer occupancy by reducing the intensity of incast flows, improving tail latency (Fig. 3-19a). However, slow start increases the




(a) 55% + 5% 100-1 incast ($99^{th}$ percentile FCT)

(b) 60% ($99^{th}$ percentile FCT)

(c) 55% + 5% 100-1 incast (Median FCT)

(d) 60% (Median FCT)

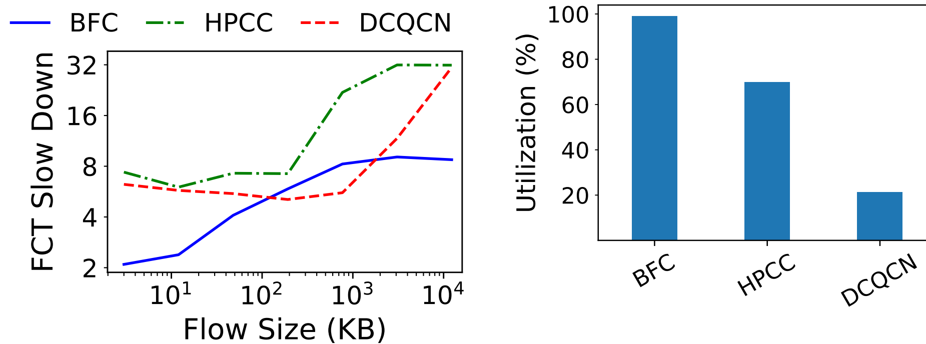
Figure 3-19: Impact of using slow start on median and 99th percentile tail latency FCT slowdown, for the Facebook flow size distribution with and without incast (setup the same as Fig. 3-11). With incast, DCTCP + SS (slow start) reduces the tail FCT, but it increases median FCTs by up to 2 ×. In the absence of incast, DCTCP + SS increases both the tail and median FCT for short and medium flows.

median FCT substantially (Fig. 3-19c). Flows start at a lower rate, taking longer to ramp up to the desired rate. For applications with serially dependent flows, an increase in median FCTs can impact the performance substantially.

In the absence of incast, slow start increases both the tail (Fig. 3-19b) and median (Fig. 3-19d) FCT for the majority of flow sizes. In particular, short flows are still slower than with BFC, as slow start does not remove burstiness in buffer occupancy in the tail.

#### 3.6.6.4 Cross Data Center Traffic

For fault tolerance, many data center applications replicate their data to nearby data centers (e.g., to a nearby metro area). We evaluate the impact of BFC on managing




(a) $99^{th}$ percentile FCT

(b) Utilization at the interconnect.

Figure 3-20: Performance in cross data center environment where two data center are connected by a 200 $\mu$s link, for the Facebook workload (60% load) with no incast traffic. The left figure shows the 99th percentile FCT slowdown for intra-data-center flows. The right figure shows the average utilization of the link connecting the two data centers.

cross-data center congestion in such scenarios. We consider the ability of different systems to achieve good throughput for the inter-data-center traffic, and we also consider the impact of the cross-data-center traffic on tail latency of local traffic, as the larger bandwidth-delay product means more data is in-flight when it arrives at the bottleneck.

We created a Clos topology with 64 leaf servers, and 100 Gbps links and 12 MB switch buffers. Two gateway switches connect the data centers using a 200 Gbps link with 200 $\mu$s of one-way delay (i.e. the base round trip delay of the link is 400 $\mu$s), or roughly equivalent to the two data centers being separated by 50 km assuming a direct connection. The experiment consists of intra-data-center flows derived from the Facebook distribution (60% load). Additionally, there are 20 long-lived inter-data-center flows in both the directions.

Fig. 3-20a shows the 99th percentile tail latency in FCT slowdown for intra-data-center flows for BFC, HPCC and DCQCN.[7] Fig. 3-20b shows the average utilization of the link connecting the two data centers (interconnect), a proxy for the aggregate throughput of the long-lived inter-data-center flows. BFC is better for both types of flows. With BFC, the link utilization of the wide area interconnect is close to 100%, while neither HPCC nor DCQCN can maintain the link at full utilization,

[7]Data center operators have developed specialized protocols for better inter-data center link management [36]; comparing those to BFC is future work.

even with ample parallelism. This is likely a consequence of slow end-to-end reaction of the inter-data-center flows [122]. The congestion state on the links within a data center is changing rapidly because of the shorter intra-data-center flows. By the time an inter-data-center flow receives congestion feedback and adjusts its rate, the congestion state in the network might have already changed. When capacity becomes available, the inter-data-center flows can fail to ramp up quickly enough, hurting its throughput.

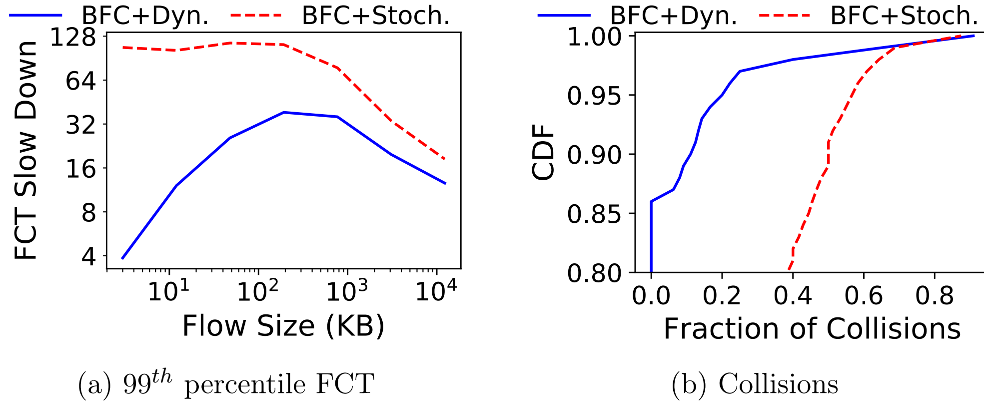
Relative to the single data center case (cf. Fig. 3-11b), tail latency FCTs are worse for all three protocols, but the relative advantage of BFC is maintained. Where HPCC has better tail latency than DCQCN in the single data center case for both short and medium-sized flows, once inter-data-center traffic is added, HPCC becomes worse than DCQCN. With bursty workloads, on the onset of congestion, the long-lived flow will take an end-to-end RTT to reduce its rate, and can build up to 1 BDP (or 500 KB) of buffering, hurting the tail latency of intra-data-center traffic. This has less of an impact on DCQCN because it utilizes less of the inter-data-center bandwidth in the first place.

In contrast, BFC reacts at the scale of the hop-by-hop RTT. Even though inter-data-center flows have higher end-to-end RTTs, on switches within the data center, BFC will pause/resume flows on a hop-by-hop RTT timescale (2 $\mu$s). As a result, with BFC, tail latencies of intra-data-center flows are relatively unaffected by the presence of inter-data-center flows, while the opposite is true of HPCC.

#### 3.6.6.5 Dynamic vs. Stochastic Queue Assignment

To understand the importance of dynamically assigning flows to physical queues, we repeated the experiment in Fig. 3-11a with a variant of BFC, BFC + Stochastic, where we use stochastic hashing to statically assign flows to physical queues (as in SFQ). In BFC (referred as BFC + Dynamic here), the physical queue assignment is dynamic. To isolate the effect of changing the physical queue assignment, the pause thresholds are the same as BFC + Dynamic.

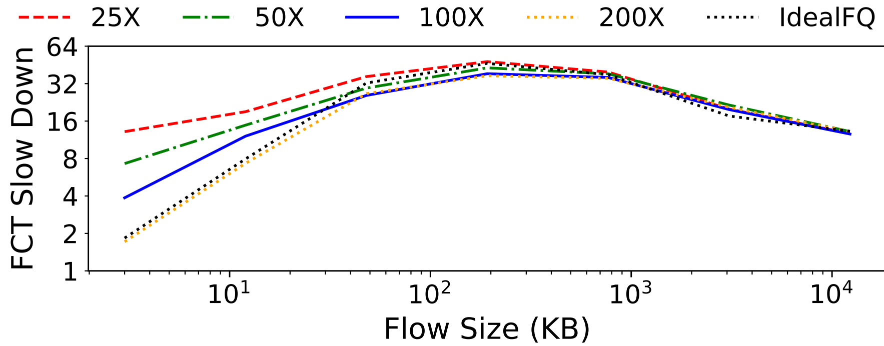
Fig. 3-21a shows the tail latency. Compared to BFC, tail latency for BFC +




(a) $99^{th}$ percentile FCT

(b) Collisions

Figure 3-21: Performance of BFC with stochastic queue assignment, for the workload in Fig. 3-11a. BFC + Stochastic incurs more queue collisions leading to worse tail latency especially for small flows compared to BFC + Dynamic.




(a) $99^{th}$ percentile FCT

Figure 3-22: FCT slowdown (99th percentile) for BFC for different size flows as a function of the size of the flow table (as a multiple of the number of queues in the switch). The other experiments in the dissertation use a flow table of 100X. Further reducing the size of the flow table hurts small flow performance.

Stochastic is much worse for all flow sizes. Without the dynamic queue assignment, flows are often hashed to the same physical queue, triggering HoL blocking and hurting tail latency, even when there are unoccupied physical queues. Fig. 3-21b is the CDF of such collisions. BFC+Stochastic experiences collisions in a high fraction of cases and flows end up being paused unnecessarily. Such flows finish later, further increasing the number of active flows and collisions. Even with incast, the number of active flows in BFC is smaller than the number of physical queues most of the time.

#### 3.6.6.6 Size of Flow Table

We repeated the experiment in Fig. 3-11a, but varied the size of the flow table (as a function of the number of queues in the switch). The default in the rest of the dissertation uses a flow table of 100X. Fig. 3-22 shows the tail latency as a function of flow size, for both smaller and larger flow tables. Reducing the size of the flow table increases the index collisions in the flow table. Each flow table collision means that those flows are necessarily assigned to the same physical queue. Tail latency FCTs degrade as a result, particularly for small flows and for smaller table sizes. This experiment shows that increasing the size of the flow table would moderately improve short flow tail latency for BFC.

## 3.7 Conclusion

In this chapter, we presented Backpressure Flow Control (BFC), a practical congestion control architecture for data center networks. BFC provides per-hop per-flow flow control, but with bounded state, constant-time switch operations, and careful use of buffers. Switches dynamically assign flows to physical queues, allowing fair scheduling among competing flows and use selective backpressure to reduce buffering with minimal head of line blocking. Relative to existing end-to-end congestion control schemes, BFC improves short flow tail latency and long flow utilization for networks with high bandwidth links and bursty traffic. We demonstrate BFC's feasibility by implementing it on Tofino2, a state-of-art P4-based programmable hardware switch. In simulation, compared to several deployed end-to-end schemes, BFC achieves 2.3 - 60$\times$ lower tail latency for short flows and 1.6 - 5$\times$ better average completion time for long flows.

# Chapter 4

# Elasticity Detection: A Building Block for Internet Congestion Control

## 4.1 Introduction

Achieving high throughput and low delay has been a key goal of congestion control research for decades. An important category of proposals is *delay-controlling* congestion control protocols. To minimize delays while avoiding "bufferbloat" [54], these schemes (e.g., Vegas [31], FAST [140], LEDBAT [121], Sprout [142], Copa [22]) reduce their rates as delays increase, unlike *buffer-filling* methods like Cubic [63], NewReno [66], and Compound [127] that must fill buffers to elicit congestion signals (packet losses or ECN). Delay-controlling protocols offer a deployable path towards reducing queuing delay in the Internet; unlike active queue management [112, 49] or packet scheduling mechanisms [103, 130], they do not require changes to routers.

There is, however, a major obstacle to deploying delay-controlling protocols on the Internet: their throughput suffers when competing against flows that compete for bandwidth more aggressively (e.g., Cubic [63], NewReno [66], BBR [36], etc.) at a shared bottleneck. For example, a Cubic flow steadily increases its rate in the absence of packet loss or ECN, causing queuing delays to rise; in response to these increasing delays, a competing delay-controlling flow will reduce its rate. The Cubic flow then grabs this freed-up bandwidth. The throughput of the delay-controlling

flow plummets, but delays don't reduce.

Is it possible to achieve the benefits of a delay-controlling protocol without lowering throughput? In this paper, we present a practical design to achieve precisely this goal. The key ingredient of our approach is a new method, **Nimbus**, to detect whether competing traffic at a bottleneck link is *elastic* or not using only end-to-end delay and rate measurements. We define a flow to be elastic if it increases its rate when it senses that more bandwidth is available at the shared bottleneck, and decreases it otherwise. All other flows are *inelastic*. Correspondingly, the cross traffic as a whole is elastic if it contains *any* elastic flows, and it otherwise inelastic.

A congestion-controlled flow backlogged at the transport layer is elastic. However, many flows on the Internet (even congestion-controlled flows) are not backlogged; examples include application-limited flows, short TCP flows that fit within the initial congestion window, constant bitrate (CBR) flows, and even video streams when the available bandwidth exceeds the maximum video bitrate. Such flows do not react to changes in available bandwidth and are thus inelastic.

Our key observation is that when Nimbus deems cross traffic to be inelastic, the sender can use a delay-controlling protocol to reduce delays for both the sender and the cross traffic without worrying about losing throughput. Otherwise, it can switch to a *TCP-competitive* protocol like Cubic (or whatever is considered dominant) to compete well without attempting to reduce delays.

Elasticity is a basic property of a backlogged congestion-controlled flow and does not depend on specifics such as its congestion control algorithm or round-trip time (RTT). We use this property to design a robust elasticity detector. A Nimbus sender modulates its rate with sinusoidal pulses to create small traffic fluctuations at the bottleneck at a specific frequency (e.g., 5 Hz). It concurrently estimates the rate of the cross traffic based on its own send and receive rates, and measures its *frequency response* (FFT) to determine if the cross traffic's rate oscillates at the same frequency. If it does, then the sender concludes that the cross traffic contains elastic flows; otherwise, it is inelastic.

In the future, if delay-controlling protocols become widely deployed, Nimbus might

miss out on some opportunities to control delays when competing against delay-controlling elastic cross-traffic flows. For example, if an elastic cross-traffic flow uses Copa (a delay-controlling scheme), then in principle it is possible to achieve low delay and high throughput by also running Copa (or some other protocol compatible with Copa). However, this would require knowing the congestion control protocol used by the cross traffic, so we could pick a compatible protocol. For instance, simply using a delay-controlling scheme like Vegas against Copa is insufficient, as it would lead to throughput loss. We sidestep this challenge by focusing on detecting elasticity, which suffices to ensure no throughput loss compared to the prevalent deployed algorithm(s). We leave detecting other properties of cross traffic (like the congestion control protocol), which could expand the set of scenarios where we can reduce delays, to future work.

**Key results:** We demonstrate the benefits of using elasticity detection for congestion control with NimbusCC, a congestion controller that uses Nimbus to switch between TCP-competitive and delay-controlling modes. We implement NimbusCC using CCP [109] in Linux. NimbusCC can support various protocols in each mode. We report results using Vegas, Copa (default "delay" mode), and BasicDelay (a new method that uses our cross traffic rate estimator), as examples of delay-controlling protocols, and Cubic and NewReno as examples of TCP-competitive protocols. Our experimental results show that:

1. Nimbus is robust to a variety of cross traffic conditions, achieving more than 85% detection accuracy even when the cross traffic is a varying mix of inelastic and elastic flows of different sizes, and when it includes multiple flows with different RTTs or congestion control protocols. These results hold across a wide range of buffer sizes, RTTs, bottleneck link rates, active queue management (AQM) schemes, flow sizes, and fractions of cross traffic.
2. NimbusCC achieves throughput within 10% of the ideal value against elastic traffic made up of a variable number of TCP flows, whereas Copa is 54% lower. NimbusCC also achieves 60 ms lower mean delay than Cubic against Poisson-distributed inelastic cross traffic.

3. When cross traffic is modeled from a flow-size distribution measured at a WAN link [34], NimbusCC achieves throughput comparable to Cubic and BBR, but with 50 ms lower median delay. Copa has slightly better (5 ms) median delay but achieves 40% lower throughput than NimbusCC and Cubic whenever cross traffic is substantially elastic. Similar results hold when the cross traffic contains elastic flows using different congestion control protocols.
4. On 25 Internet paths, NimbusCC achieved a throughput at least as high as Cubic with lower delays on 60% of the paths and similar delays on the other 40%. Compared to BBR, NimbusCC's throughput was 10% lower, but the mean packet delay was 40–50 ms lower.

Elasticity detection is a general technique and while we explore its use in congestion control, we believe it could be used to solve other problems in the future — e.g., for aggregate traffic control between sites [35] and in tools like speedtest to inform users not only of the rate and delay, but also the nature of the cross traffic on particular paths (and hence whether using a different congestion control protocol could improve throughput or delay).

## 4.2 Related Work

Copa [22] aims to maintain a bounded number of packets in the bottleneck queue. Copa induces a periodic pattern of sending rate that nearly empties the queue once every 5 RTTs. This helps Copa flows obtain an accurate estimate of the minimum RTT and the queuing delay. In addition, Copa uses this pattern to detect the presence of non-Copa flows: it expects the queue to be nearly empty at least once every 5 RTTs, provided only Copa flows with similar RTTs share the bottleneck link. If this does not occur, Copa switches to a TCP-competitive mode.

This method is sensitive to variations in cross-traffic (e.g., arrival/departure of flows), the control protocol used by the cross-traffic flows, and even their RTTs. For these reasons, we find that Copa suffers from both false positives (increased delay) and, more importantly, false negatives (lower throughput) (see §4.5, §4.7.1

and §4.7.2). Unlike Copa, Nimbus does not look for a specific pattern in the RTTs. Instead it directly estimates elasticity by measuring whether the cross-traffic reacts to rate fluctuations over a few seconds in the frequency domain. This method is more robust and can be applied to any combination of TCP-competitive and delay-controlling algorithms, whereas Copa's approach relies on the specific dynamics of its rate controller.

BBR [36] estimates the bottleneck bandwidth $b$ and minimum RTT $d$. It paces traffic at $b$ while capping the number of in-flight packets to $2 \times b \times d$. To estimate the bottleneck, BBR periodically increases its rate over $b$ for about one RTT and then reduces it for the following RTT. BBR uses this sending-rate pattern to obtain estimates of $b$; specifically, it tests if the observed rate exceeds the current estimate $b$ in the rate-increase phase. However, BBR doesn't use these pulses to infer the nature of cross-traffic.

PCC-Vivace [42] uses an online learning algorithm to adapt its sending rate to maximize a utility function that incorporates the achieved rate, delay, and loss rate. Our experiments (§4.5, §4.7.1) show that Vivace cannot achieve both low delay with inelastic cross-traffic and compete fairly with elastic TCP flows. Compound TCP [133] maintains both a loss-based window and a delay-based window, and transmits data based on the sum of the two windows. Compound does not attempt to switch between two modes, and therefore it incurs high queuing delays due to its loss-based window.

## 4.3 Cross-traffic Estimation

We present a simple new method to estimate the total rate of cross-traffic at the sender (§4.3.1). Then, we show how to detect whether the cross-traffic contains *any* elastic flows, describing the key principles (§4.3.2) and a practical method (§4.3.3).

Figure 4-1 shows our network model and introduces some notation. A sender communicates with a receiver over a single bottleneck link of rate $\mu$. The bottleneck link is shared with cross-traffic, consisting of an unknown number of flows, each of

which is either elastic or inelastic. $S(t)$ and $R(t)$ denote the time-varying sending and receiving rates, respectively, while $z(t)$ is the total rate of the cross-traffic.

**Operating regime:** Our technique requires some degree of traffic persistence. The sender must be able to create sufficient pulses and observe the impact on cross traffic over a period of time. Thus, it is best suited for long flows. Fortunately, it is for such transfers that delay-controlling schemes are useful, because short flows are unlikely to cause significant queueing delay [54]. The detector is designed for a single bottleneck link with a stable rate, and uses a link-rate estimator similar to BBR's. When these conditions do not hold, the detector can become inaccurate and have false positives. Our detector conservatively classifies cross traffic as elastic in these cases (with high likelihood, discussed in §4.6 and §4.7.3). When applied to congestion control, the detector will thus choose a TCP-competitive mode in these scenarios. Thus, while such false positives might cause the sender to miss out on opportunities to reduce delay, they will not cause it to lose throughput.

Our technique is most effective when the elastic flows react on a timescale of a few RTTs. If an elastic flow is slower to react, it can go undetected with short pulses. By using long pulses Nimbus can detect such sluggish elastic flows but at the cost of some congestion. Since, majority of elastic traffic on the Internet reacts on RTT timescales (e.g., ACK-clocked flows), we use short pulses. For ease of exposition, we describe Nimbus in the context of detecting ACK-clocked flows, but the technique applies more generally (e.g., correctly classifying fast-reacting rate-based flows).

### 4.3.1 Estimating the Rate of Cross-traffic

In Fig. 4-1, the total traffic into the bottleneck queue is $S(t) + z(t)$, of which the receiver sees $R(t)$. As long as the bottleneck link is busy (i.e., its queue is not empty), and the router treats all traffic the same way, the ratio of $R(t)$ to $\mu$ must be equal to the ratio of $S(t)$ and the total incoming traffic, $S(t) + z(t)$.[1] Using this property, we

[1]This property holds even when the bottleneck link is dropping packets as long as the drop rate is the same for the sender-to-receiver flow and the cross-traffic.

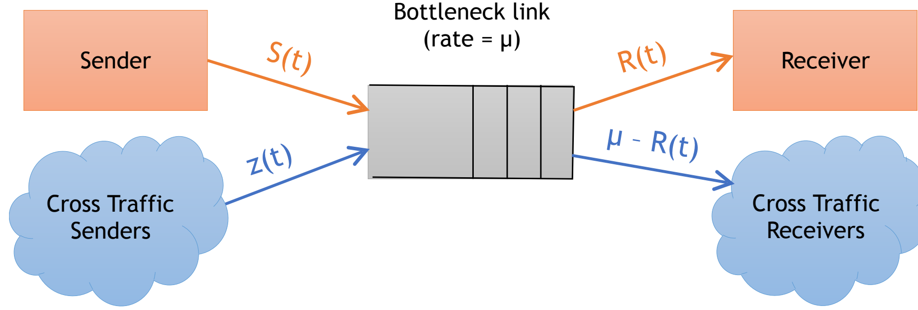



Figure 4-1: **Network model** — The time-varying total rate of cross-traffic is $z(t)$. The bottleneck link rate is $\mu$. The sender's transmission rate is $S(t)$, and the rate of traffic received by the receiver is $R(t)$.

propose the following estimator for $z(t)$:

$$\hat{z}(t) = \mu \frac{S(t)}{R(t)} - S(t). \tag{4.1}$$

We estimate $S(t)$ and $R(t)$ by considering $n$ packets at a time:

$$S_{i,i+n} = \frac{n_{bytes}}{s_{i+n} - s_i}, \qquad R_{i,i+n} = \frac{n_{bytes}}{r_{i+n} - r_i}, \tag{4.2}$$

where $n_{bytes}$ is the number of bytes in the $n$ packets, $s_k$ is the time at which the sender sends packet $k$, $r_k$ is the time at which the sender receives the ACK for packet $k$, and the units of the rates are bytes per second. $S(t)$ and $R(t)$ must be measured over the *same* $n$ packets.

The above quantities can be calculated using the timestamps of the first and the last packet and hence are unaffected by delayed acknowledgements (delayed ACKs). Our implementation in CCP [109] reuses the Linux kernel's measurements of $S(t)$ and $R(t)$ over the last RTT—the same method used by the BBR implementation [84]. Like BBR [36], we use the maximum received rate to estimate $\mu$, taking care to avoid incorrect estimates due to ACK compression.[2]

We have conducted several tests with various patterns of cross-traffic to evaluate the effectiveness of this $z(t)$ estimator (including scenarios with packet drops and delayed ACKs). The overall error is small: the 50th and 95th percentiles of the

[2]A variety of other techniques [68, 43, 44, 93, 74, 94, 100] could also be used to estimate $\mu$.

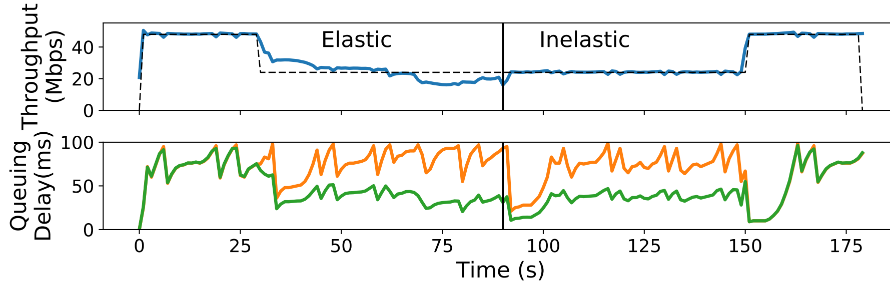



Figure 4-2: **Instantaneous delay measurements do not reveal elasticity** — The bottom plot shows the total queueing delay (orange) and the self-inflicted delay (green). The experiment contains one background Cubic flow in the elastic region (30–90 s) and CBR cross-traffic in the inelastic region (90–150 s).

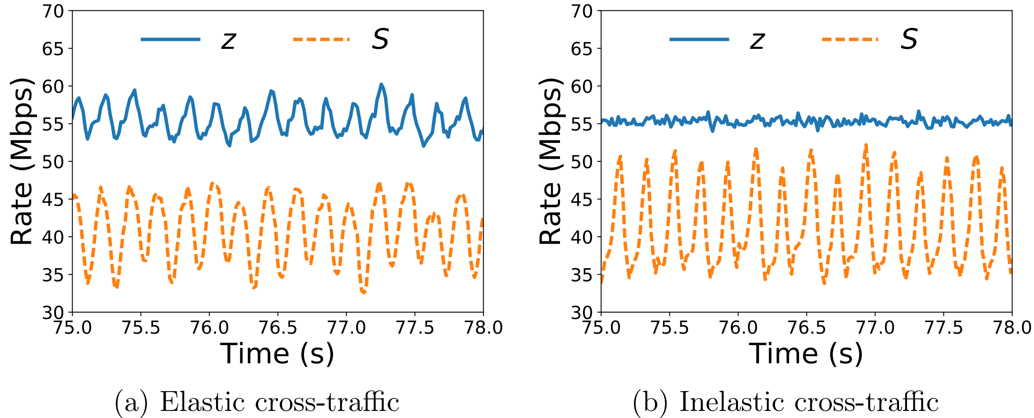
relative error are 1.3% and 7.5%, respectively. Unlike prior work on estimating cross-traffic rate [131, 76, 69], our method is in-band and does not use any probe packets; however, it relies on the property that the sender is persistently backlogged.

### 4.3.2 Elasticity Detection: Principles

We now turn to designing an online estimator for a sender to determine if the cross-traffic includes *any* elastic flows.[3] A strawman approach might attempt to detect elastic flows by estimating the contribution of the cross-traffic to queueing delay. For example, the sender can estimate its own contribution to the queueing delay—i.e., the "self-inflicted" delay—and if the total delay is significantly higher than the self-inflicted delay, conclude that the cross-traffic is elastic.

This scheme does not work. To see why, consider the experiment in Figure 4-2, where a Cubic flow shares a link with elastic and inelastic traffic in two separate time periods. The self-inflicted queueing delay for the Cubic flow (green, bottom figure) looks the same in the elastic and inelastic phases. The reason is that a flow's share of the queue occupancy is proportional to its throughput, which is roughly the same in the two phases (top figure). Because the Cubic flow gets 50% of the bottleneck link, its self-inflicted delay is roughly half of the total queueing delay always (orange, bottom figure). This example suggests that instantaneous measurements cannot be

[3]Receiver participation will improve accuracy by avoiding the need to estimate $R(t)$ from ACKs at the sender, but would be a little harder to deploy.




(a) Elastic cross-traffic

(b) Inelastic cross-traffic

Figure 4-3: **Cross-traffic's reaction to pulses** — The pulses change the inter packet spacing for cross-traffic. Elastic traffic reacts to these changes after a RTT, while inelastic traffic does not.

used to distinguish between elastic and inelastic cross-traffic.

**To detect elasticity, tickle the cross-traffic!** Our method detects elasticity by monitoring how the cross-traffic responds to induced traffic variations at the bottleneck link over a period of time. The key observation is that elastic flows react in a predictable way to rate fluctuations at the bottleneck. Consider, for example, long-running Cubic or NewReno flows, which are ACK-clocked. For these flows, if an ACK is delayed by a time duration $\delta$, then the next packet transmission will also be delayed by $\delta$. Therefore changes in the rate of packet arrivals at the receiver cause similar changes in the sending rate after one RTT via the ACKs. By contrast, the sending rate of inelastic flows does not depend on the receive rate.

We induce changes in the inter-packet spacing of cross-traffic at the bottleneck link by sending packets in *pulses*. We take the desired sending rate, $S(t)$, and alternate between sending at rates higher and rates lower than $S(t)$, ensuring that the mean rate is $S(t)$. Sending in such pulses (e.g., modulated on a sinusoid) changes the inter-packet spacing of the cross-traffic departing the bottleneck link in a controlled manner. If the cross-traffic contains elastic flows, then because of the induced changes in the ACK clocks of those flows, their rates will react to our pulses. When we increase our rate, the elastic cross-traffic will reduce its rate in the next RTT, and conversely. If enough of the cross-traffic is elastic, then our sender can measure and detect these fluctuations in the cross-traffic rate.

Fig. 4-3a and Fig. 4-3b compare the responses of elastic (Cubic) and inelastic (constant bit rate) cross-traffic when the sender transmits packets in sinusoisal pulses at frequency $f_p = 5$ Hz. $S(t)$ is the sender's rate and $z(t)$ is the estimated cross-traffic rate computed using Equation (4.1). The path has a minimum RTT of 50 ms and a buffer size of 100 ms ($2\times$ the bandwidth-delay product). The elastic flow's sending rate after one RTT is inversely correlated with the pulses in the sending rate, while the inelastic flow's sending rate is unaffected.

### 4.3.3 Elasticity Detection: Practice

To produce a practical method to detect cross-traffic using this idea, we must address three challenges:

1. Pulses in the sending rate must induce a measurable change in $z$, but not congest the bottleneck link.
2. Because there is natural variation in cross-traffic, and noise in $\hat{z}$, it is not easy to perform a robust comparison between the predicted change in $z$ and the measured $z$.
3. Because the sender does not know the RTTs of cross-traffic flows, it does not know when to look for the predicted response in the cross-traffic rate.

The first method we developed to solve these problems measured the *cross-correlation* between $S(t)$ and $z(t)$. A cross-correlation near zero would be considered inelastic cross-traffic, whereas a significant non-zero value would indicate elastic cross-traffic. We found that this approach works well (with square-wave pulses) if the cross-traffic is substantially elastic and has a similar RTT to the flow trying to detect elasticity, but not otherwise. The trouble is that because elastic cross-traffic will react after *its* RTT, $S(t)$ and $z(t)$ must be aligned using the cross-traffic's RTT, which is not easy to infer. Moreover, the elastic flows in the cross-traffic may have different RTTs, making the alignment even more challenging.

**From time to frequency domain:** We have developed a method, Nimbus, that overcomes the challenges stated above. It uses two ideas. First, the sender modu-

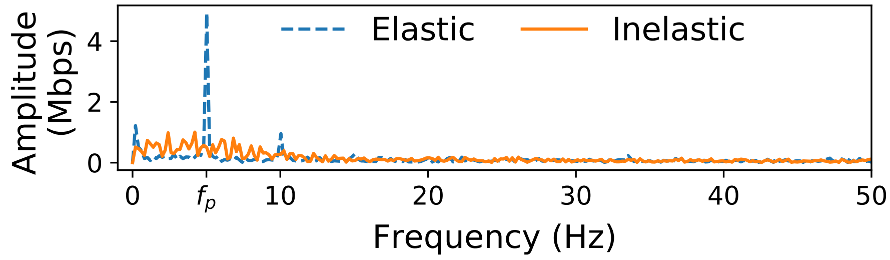



Figure 4-4: **Cross-traffic FFT for elastic and inelastic traffic** — Only the FFT for elastic traffic has a pronounced peak at $f_p$ (5 Hz).

lates its packet transmissions using *sinusoidal pulses* at a known frequency $f_p$, with amplitude equal to a modest fraction (e.g., 25%) of the bottleneck link rate. These pulses induce a noticeable change in inter-packet times at the link without causing congestion, because the queues created in one part of the pulse are drained in the subsequent part, and the period of the pulses is short (e.g., $f_p = 5$ Hz). By using short pulses, we ensure that the total burst of data sent in a pulse is a small fraction of the typical bottleneck queue size.

Second, the sender looks for periodicity in the cross-traffic rate at frequency $f_p$, using a frequency domain representation of the cross-traffic rates. We use the Fast Fourier Transform (FFT) of the time series of the cross-traffic estimate $\hat{z}(t)$ over a short time interval (e.g., 5 seconds). Detecting periodicity in the frequency domain is more robust than the time-domain, for the same reason that frequency modulation provides better signal-to-noise ratio than amplitude modulation [119]: it is less affected by variations in the cross-traffic rate and measurement noise. Further, observing the cross-traffic's response at a known frequency, $f_p$, yields a method that is robust to the presence of multiple elastic flows with different RTTs, and even, different congestion control protocols, because all elastic flows (irrespective of RTT and protocol) will exhibit rate oscillations at the frequency $f_p$. As a result, there will be an overall response at frequency $f_p$ in the cross-traffic, equal to superposition of the responses of the individual elastic flows at frequency $f_p$.[4]

[4]In theory, the response of flows with different RTTs may cancel each other out, but this is very unlikely since it requires specific combinations of RTTs. We have not seen this problem occur in our experiments (§4.7.2).

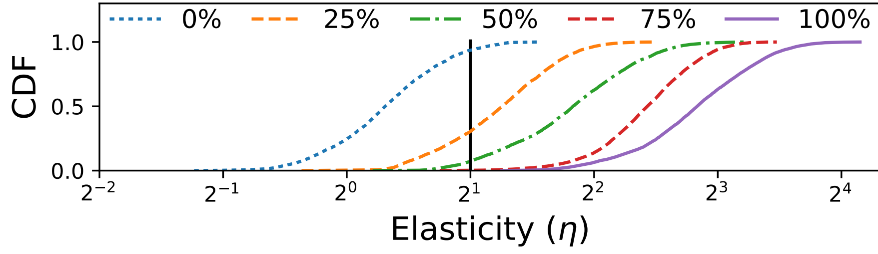



Figure 4-5: **Distribution of elasticity with varying elastic fraction of cross-traffic** — The cross-traffic consists of an elastic Cubic flow and inelastic Poisson-distributed traffic with different rates. Completely inelastic cross-traffic has $\eta$ close to zero, while completely elastic cross-traffic exhibits a high $\eta$. Cross-traffic with some elastic fraction also exhibits high elasticity ($\eta > 2$).

Fig. 4-4 shows the FFT of the $\hat{z}(t)$ time-series produced using Equation (4.1) for examples of elastic and inelastic cross-traffic, respectively. Elastic cross-traffic exhibits a pronounced peak at $f_p$ compared to the neighboring frequencies, while for inelastic traffic the FFT magnitude is spread across many frequencies. The magnitude of the peak depends on how much of the cross-traffic is elastic; the more elastic the cross-traffic, the sharper the peak at $f_p$. Therefore, rather than compare the peak at $f_p$ to a pre-determined threshold, we compare it to the magnitude of the nearby frequencies.

We define the *elasticity metric*, $\eta$, as follows:

$$\eta = \frac{|FFT_z(f_p)|}{\max_{f \in (f_p+\epsilon, 2f_p-\epsilon)} |FFT_z(f)|} \tag{4.3}$$

Equation (4.3) compares the magnitude of the FFT at frequency $f_p$ to the peak magnitude in the range from just above $f_p$ to just below $2f_p$. We use $\epsilon = 0.5$ Hz (with $f_p = 5$ Hz) in our implementation. If $\eta$ is less than a threshold $\eta_{thresh}(\geq 1)$, then the cross-traffic is deemed inelastic; otherwise, it is elastic.

### 4.3.4 Setting Parameters for Elasticity Detection

**Detection threshold:** In practice, cross-traffic can be a mix of elastic and inelastic flows. In such scenarios, we want our detector to be sensitive to the presence of any

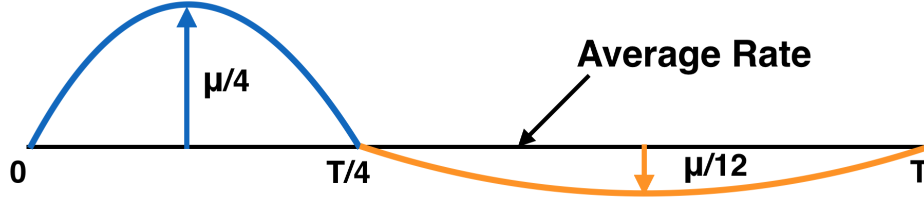



Figure 4-6: **Asymmetric sinusoidal pulse** — The pulse has period $T = 1/f_p$. The positive half-sine lasts for $T/4$ with amplitude $\mu/4$, and the negative half-sine lasts for the remaining duration, with amplitude $\mu/12$. The two half-sines cancel out each other over one period.

elastic flows, since even one elastic flow can eventually grab all the link bandwidth from a delay-controlling flow. Fig. 4-5 shows the CDF of elasticity ($\eta$) as the fraction of bytes belonging to elastic flows in the cross-traffic varies. $\eta$ varies due to variations in the cross-traffic but its value generally increases as more of the cross-traffic becomes elastic: the median values range from $\eta = 1$ for purely inelastic traffic to $\eta = 10$ for purely elastic traffic.

The value of $\eta_{thresh}$ dictates which type of traffic is detected more reliably. A large $\eta_{thresh}$ will ensure that inelastic traffic is always classified correctly, but it increases the chance that cross-traffic with a small elastic component is misclassified as inelastic (potentially hurting throughput). With a small $\eta_{thresh}$, on the other hand, elastic traffic will be classified correctly, but we may occasionally classify inelastic traffic as elastic, losing an opportunity to reduce delays. To balance these concerns, we choose a small fixed threshold $\eta_{thresh} = 2$, which in Fig. 4-5 corresponds to classifying cross-traffic with a 25% elastic component correctly 75% of the time. We evalutate the impact of $\eta_{thresh}$ on congestion control performance in §4.7.2.

**Pulse shaping:** Rather than a pure sinusoid, we use an *asymmetric* sinusoidal pulse, as shown in Fig. 4-6. In the first one-quarter of the pulse cycle, the sender adds a half-sine of a certain amplitude (e.g., $\mu/4$) to $S(t)$; in the remaining three-quarters of the cycle, it subtracts a half-sine with one-third of the amplitude used in the first quarter of the cycle (e.g., $\mu/12$). The reason for this asymmetric pulse is that it enables senders with low sending rates, $S(t)$, to generate pulses. For example, for a peak amplitude of $\mu/4$, a sender with $S(t)$ as low as $\mu/12$ can generate the asymmetric

pulse shown in Fig. 4-6; a symmetric pulse with the same peak rate would require $S(t) > \mu/4$.

Our pulses produce an observable pattern in the FFT when the cross-traffic is elastic. Using asymmetric sinusoidal pulses creates harmonics at multiples of the pulse frequency $f_p$. However, these harmonics do not affect $\eta$ (see Equation (4.3)), which only uses the FFT in the frequency band $[f_p, 2f_p - \epsilon)$.

**Pulse duration:** What should the duration, $T$, of the pulse be? The answer depends on two factors: first, the interval over which $S$ and $R$ are measured (with which the sender computes $\hat{z}$), and second, the amount of data we are able to send in excess of the mean rate without causing congestion. If $T$ were smaller than the measurement interval of $S$ and $R$, the perturbation to the cross-traffic rate during one part of the pulse will be averaged out during the rest of the pulse, resulting in no impact on $\hat{z}(t)$. But $T$ cannot be too large because the sender transmits in excess of the mean rate $S(t)$ for $T/4$. In particular, the size of the burst sent in a pulse is $\frac{2}{\pi}\frac{\mu}{4}\frac{T}{4} = \frac{T\mu}{8\pi} \approx 0.04\mu T$. If $T$ is equal to the RTT, this is 4% of the bandwidth-delay product (BDP) at the peak. Moreover, since pulsing doesn't increase the average sending rate, there is no increase in the average queuing delay (§4.5).

We set $T$ to a large RTT value observed on the Internet, for example $T = 200$ ms, with the rationale that router buffers are typically provisioned to avoid packet losses for one such RTT, and because our implementation measures $S$ and $R$ over one RTT. We measure rates over one RTT because sub-RTT measurements are confounded by burstiness in packet transmissions (e.g., caused by ACK compression [79]).

If the cross-traffic reacts slower than the pulse duration, Nimbus might misclassify those flows. Using longer pulses could improve detection accuracy in such scenarios but it might cause congestion. We evaluate this alternative for detecting PCC-Vivace, a rate-based scheme (not ACK-clocked), in Appendix G.2.3.

**FFT duration:** Computing FFTs over a small duration allows quick responses to changes in cross-traffic, but it increases errors due to noise. For example, natural variations in inelastic cross-traffic over small periods can cause false peaks at $f_p$ in the FFT, resulting in a misclassification. The FFT duration also impacts the

frequency resolution of the FFT; in particular, $\epsilon$ in Equation (4.3) must be larger than 1/`FFT_Duration`. We choose an FFT duration of 5 seconds (corresponding to 25 pulses) and $\epsilon = 0.5$ Hz to balance these concerns. In §4.7.1, we show that even when the cross-traffic is a highly dynamic mix of inelastic and elastic flows of different sizes, Nimbus achieves high detection accuracy and congestion control performance.

## 4.4 NimbusCC

NimbusCC is a congestion control system that uses mode switching. It has a TCP-competitive mode in which the sender transmits using a TCP-competitive congestion control algorithm (e.g., Cubic), and a delay-control mode that uses a delay-controlling algorithm (e.g., Copa). NimbusCC switches between the two modes using our elasticity detector, Nimbus.

### 4.4.1 Mode Switching

At any given time, NimbusCC transmits data at the time-varying rate dictated by the congestion control algorithm running at that time. It modulates this rate with asymmetric sinusoidal pulses (Fig. 4-6). NimbusCC uses the pulsing parameters described in §4.3.4, calculating $S$ and $R$ over one window's worth of packets. It computes the FFT for the $z$ measurements reported in the last 5 seconds to calculate elasticity ($\eta$) using Equation (4.3), and it picks the mode by comparing $\eta$ to $\eta_{thresh} = 2$ (§4.3.4).

We support Cubic and NewReno for the TCP-competitive mode and Copa's default mode and Vegas for the delay-control mode. We also implemented a simple delay-controlling algorithm, called BasicDelay, which relies on our cross-traffic rate estimator to calculate the spare capacity at the sender.

BasicDelay uses a typical control loop inspired by prior explicit control protocols [81, 132, 57]. Let $S$ be the sending rate and $\hat{z}$ be the estimated cross-traffic rate, both measured over the last window of packets. Also, let $x$ be the current RTT, and $x_{min}$ be the minimum observed RTT. Upon receiving an ACK, BasicDelay sets its current rate to:

$$\text{Rate} \leftarrow S + \alpha(\mu - S - \hat{z}) + \beta\frac{\mu}{x}(x_{\min} + d_t - x), \tag{4.4}$$

where $\alpha$ and $\beta$ are constants smaller than 1, and $d_t$ is a target queuing delay. The term $(\mu - S - z)$ is the sender's estimate of the spare capacity in the last RTT. By adding an $\alpha$-fraction of the spare capacity to $S(t)$, BasicDelay tries to get closer to the ideal rate. The second term in the above rule seeks to maintain a specified queuing delay, $d_t$, to prevent the queue from both growing too large or going empty. Recall that our cross-traffic estimator, Equation (4.1), requires a non-empty queue to estimate $z$.

NimbusCC takes special care in initializing the rate when switching to TCP-competitive mode. NimbusCC sets the rate (and equivalent window) to the rate that was used 5 seconds ago because the elasticity detector takes 5 seconds (FFT Duration) to detect elastic cross-traffic. During this time, the elastic traffic could cause a reduction in the delay-control mode's rate. Hence, NimbusCC resets its rate to the rate at the beginning of the 5-second detection period.

### 4.4.2 Multiple NimbusCC Flows

What happens when a bottleneck is shared by multiple NimbusCC flows? If all the NimbusCC flows pulse at the same frequency ($f_p$), then they will all detect a peak in the FFT at that frequency and stay in the TCP-competitive mode (regardless of the other cross-traffic). Thus they will achieve the same throughput as the TCP-competitive protocol and compete fairly with each other, but will not maintain low delays when there is no elastic cross-traffic.

Ideally, we want all the NimbusCC flows to remain in delay-control mode when there is no elastic cross-traffic, and use TCP competitive mode otherwise. One approach is for different NimbusCC flows to pulse at different frequencies. But this approach cannot scale to more than a few flows, because the set of distinguishable frequencies is limited (recall that the pulse period $T$ cannot be too small).

**The pulser and the watchers:** We propose a different approach. One of the NimbusCC flows assumes the role of the *pulser*, while the others are *watchers*. They

coordinate without explicit communication; in fact, each NimbusCC flow is unaware of the identities, or even existence, of the others.

The pulser sends data by modulating its rate with asymmetric sinusoids. The pulser uses two different frequencies, $f_{pc}$ in TCP-competitive mode, and $f_{pd}$ in delay-control mode. The values of these frequencies are fixed and agreed upon beforehand; we use $f_{pc} = 5$ Hz and $f_{pd} = 6$ Hz in our experiments.[5]

A watcher infers whether the pulser is pulsing at frequency $f_{pc}$ or frequency $f_{pd}$ by computing the FFT of its receive rate, $R$, at these two frequencies. It then picks the mode corresponding to the larger peak to match the pulser's mode. Note that since a watcher is not pulsing, it can detect the pulser's pulses in its own receive rate, $R$; i.e., it does not even need to estimate $z$. The pulser, on the other hand, cannot look at its own $R$ to detect pulses in the cross-traffic, since it will end up detecting its own pulses.

For multiple NimbusCC flows to maintain low delays during times when there is no elastic cross-traffic on the link, the pulser must classify watcher traffic as inelastic. Note that from the pulser's perspective, the watcher flows are part of the cross-traffic; thus, to avoid confusing the pulser, the rate of watchers must not react to the pulses of the pulser. To achieve this goal, a watcher applies an exponentially weighted moving average (EWMA) filter to its transmission rate before sending data. The EWMA filter cuts off all frequencies in the sending rate that exceed $\min(f_{pc}, f_{pd})$.

**Pulser election:** A distributed and randomized election decides which flow is the pulser and which are watchers. If a NimbusCC flow determines that there is no pulser (by seeing that there is no peak in the FFT at the two potential pulsing frequencies), then it decides to become a pulser with a probability proportional to its transmission rate:

$$p_i = \frac{\kappa\tau}{\text{FFT Duration}} \times \frac{R_i}{\mu}. \tag{4.5}$$

Each flow makes decisions periodically, e.g., every $\tau = 10$ ms, $\kappa$ is a constant, and $R_i$ is the receive rate of the $i^{th}$ flow. This rule ensures that the expected number of flows that become pulsers over the FFT duration is at most $\kappa$. To see why, note that the

[5]These values are in accordance with bounds on $T$ and $f$ described in §4.4.

expected number of pulsers is equal to the sum of the probabilities in Equation (4.5) over all the decisions made by all flows in the FFT duration. Since $\sum_i R_i \leq \mu$ and each flow makes (FFT Duration/$\tau$) decisions, these probabilities sum up to at most $\kappa$.

It is also not difficult to show that the number of pulsers within an FFT duration has approximately a Poisson distribution with a mean of $\kappa$ [47]. Thus the probability that after one flow becomes a pulser, a second flow also becomes a pulser before it can detect the pulses of the first flow in its FFT measurements is $1 - e^{-\kappa}$. Therefore, $\kappa$ involves a tradeoff: a smaller $\kappa$ will lead to fewer conflicts but will take longer to elect a pulser. In our experiments, we use $\kappa = 1$.

For any value of $\kappa$, there is a non-zero probability of more than one concurrent pulser. In such cases, all the NimbusCC flows will stay in TCP-competitive mode: they could miss opportunities to reduce delay but will *not* lose throughput relative to the status quo. As a further optimization, if there are multiple pulsers, then each pulser will observe that the cross-traffic has more variation than the variations it creates with its pulses. This can be detected by comparing the magnitude of the FFT of the cross-traffic $z(t)$ at $f_p$ with the FFT of the pulser's receive rate $R(t)$ at $f_p$. If the cross-traffic's FFT has a larger magnitude at $f_p$, the NimbusCC pulser concludes that there must be multiple pulsers and switches to a watcher with a fixed probability.

**Remark:** This scheme for coordinating pulsers is similar to receiver-driven layered multicast (RLM) congestion control [102]. In RLM, a sender announces to the multicast group that it is conducting a probe experiment at a higher rate, so any losses incurred during the experiment should not be heeded by the other senders. In contrast, in Nimbus, there is no explicit coordination channel, and the pulsers and watchers coordinate via their independent observations of cross-traffic patterns. The pulser election also shares similarities with carrier sense multiple access (CSMA) protocols. Similar to a CSMA sender, a watcher looks for the absence of any pulser (free channel) on the shared bottleneck, and switches probabilistically to a pulser to try to avoid multiple pulsers (collisions). However, unlike CSMA/CD or CSMA/CA protocols,

collisions are harder to detect (and consequently rectify) in Nimbus.

## 4.5 Visualizing NimbusCC

We illustrate NimbusCC on a synthetic workload with time-varying cross-traffic. We emulate a bottleneck link in Mahimahi [110], a link emulator. The network has a bottleneck rate of 96 Mbit/s, a minimum RTT of 50 ms, and 100 ms (2 BDP) of buffering. We compare two mode-switching protocols, NimbusCC (Cubic+BasicDelay) and NimbusCC (Cubic+Copa), with Cubic, BBR, Vegas, and PCC-Vivace (all from Linux), Copa (from Copa's authors), and Compound atop CCP (written by us).

The cross-traffic varies over time between elastic, inelastic, and a mix of the two. We generate inelastic cross-traffic using Poisson packet arrivals at the specified mean rate. Elastic cross-traffic uses Cubic, via `iperf` [134].

Fig. 4-7 shows the throughput and queuing delays for the various protocols, as well as the correct fair-share rate. Table 4.1 summarizes the deviation from fair-share throughput in the elastic (20–120 s) and inelastic (0–20 and 120–180 s) regions, and the mean queuing delay in the inelastic region. The delay in the elastic region is similar for all schemes.

Throughout the experiment, both NimbusCC variants achieve throughput close to the fair-share rate and low ($\leq$15 ms) queuing delays in the presence of inelastic cross-traffic. With elastic cross-traffic, both variants switch to TCP-competitive mode within 5 seconds and achieve close to their fair share. The delays during this period approach the buffer size because the competing traffic is buffer-filling; the delays return to their previous low value (15 ms) within 5 seconds after the elastic flows complete. NimbusCC stays in the correct mode throughout the experiment, except for one interval in the elastic period. The deviation from fair-share in the elastic region is because Cubic is not perfectly fair to itself over short time periods.

Cubic achieves it's fair-share rate but experiences high delays (80 ms) throughout. BBR's throughput is often much higher than its fair share with high delays even against inelastic cross-traffic, which prior work has also observed [22, 65].

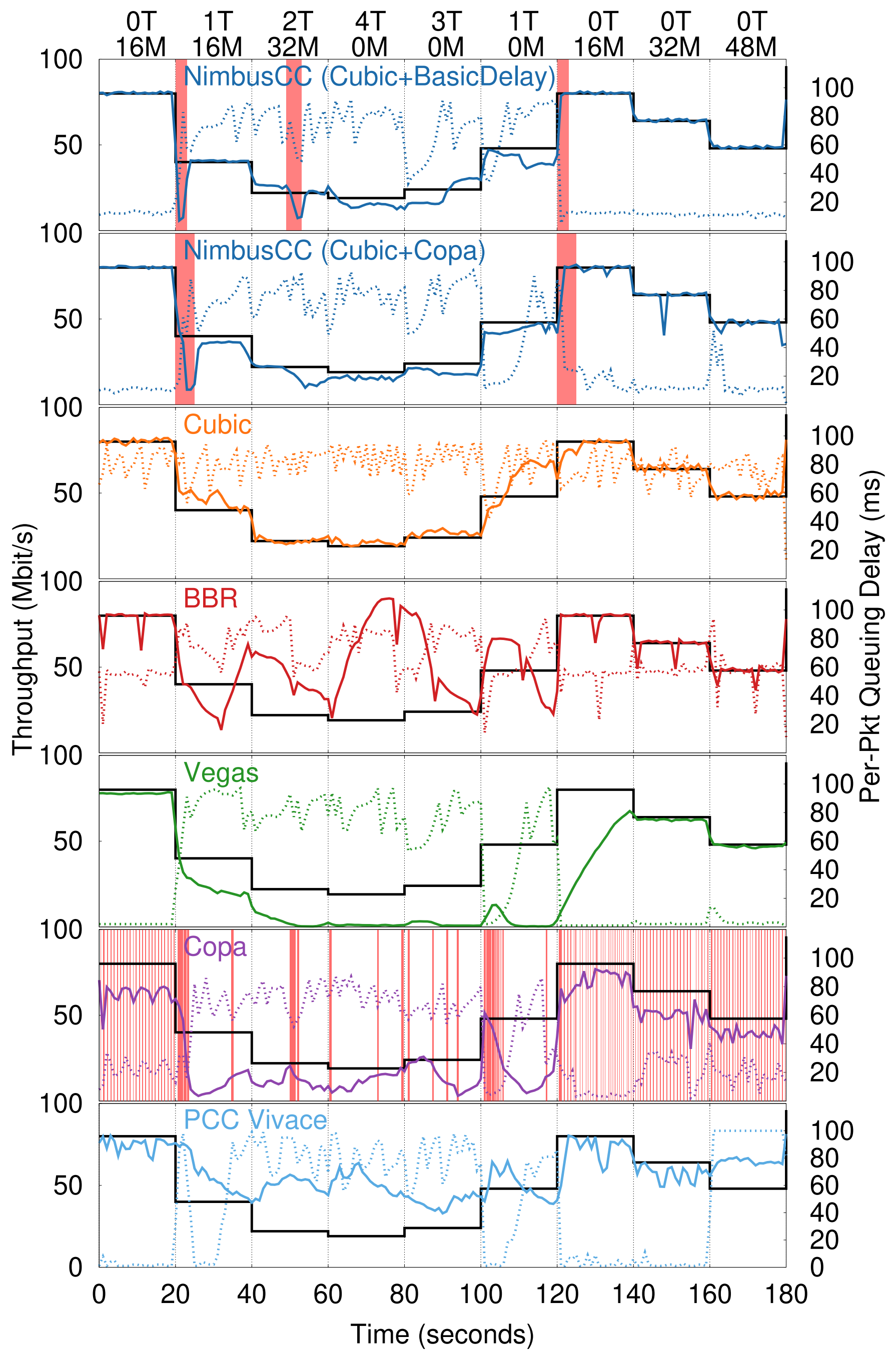



Figure 4-7: Performance on a 96 Mbit/s Mahimahi link with 50 ms delay and 2 BDP of buffering while varying the rate and type of cross-traffic as denoted at the top of the graph. $x$M denotes $x$ Mbit/s of inelastic Poisson cross-traffic. $y$T denotes $y$ long-running Cubic cross-flows. The solid black line indicates the correct time-varying fair-share rate that the protocol should achieve given the cross-traffic. For each scheme, the solid line shows throughput and the dotted line shows queuing delay. The cross-traffic contains elastic flows from 20–120 s. For Nimbus and Copa, the red shaded regions indicate times spent in the wrong mode (e.g., delay-controlling with elastic cross-traffic).

| Scheme | Throughput Δ Elastic | Throughput Δ Inelastic | QDelay Inelastic |
|---|---|---|---|
| NimbusCC Cubic+BasicDelay | −10% | 0% | 12 ms |
| NimbusCC Cubic+Copa | −15% | −1% | 14 ms |
| Cubic | +12% | 0% | 78 ms |
| BBR | +61% | −2% | 56 ms |
| Vegas | −79% | −15% | 3 ms |
| Copa | −54% | −19% | 18 ms |
| PCC-Vivace | +61% | −2% | 27 ms |

Table 4.1: Average queuing delay (in ms) in the inelastic region, and deviation from fairshare throughput in elastic and inelastic regions from Fig. 4-7. NimbusCC is the only scheme to achieve close to fair-share throughput and low delays.

Vegas suffers from low throughput in the presence of elastic cross-traffic as it reacts to packet delays.

While Copa generally uses the correct mode it frequently switches mode unnecessarily; Copa makes 28 switching errors in the elastic region, while NimbusCC only switches once. In the elastic period, Copa's frequent mode switches lower its throughput (14 Mbit/s) compared to NimbusCC (27.5 Mbit/s) and fair-share rate (e.g., see 100–120 s). Further, by draining queues periodically, Copa incurs minor underutilization against inelastic traffic (e.g., 140–160 s).

Vivace competes unfairly with elastic traffic. At times, Vivace fails to maintain low delays against inelastic cross-traffic and incurs heavy packet loss (e.g., 160–180s).

## 4.6 Discussion and Limitations

**Rate-based protocols:** Table 4.2 summarizes how Nimbus classifies different types of cross traffic. Recall that our method relies on the cross traffic responding to variations induced by pulses on an RTT timescale. This is true of all ACK-clocked protocols, which are classifed as elastic.

For BBR, recent work has showed that it is ACK-clocked when competing with other flows [138]; Nimbus thus classifies it as elastic. We therefore find that NimbusCC

| Cross Traffic | Elastic | ACK-Clocked | Classification |
|---|---|---|---|
| Cubic | Yes | Yes | Elastic |
| NewReno | Yes | Yes | Elastic |
| Copa | Yes | Yes | Elastic |
| Vegas | Yes | Yes | Elastic |
| BBR | Yes | If CWND-limited | Elastic* |
| PCC-Vivace | Yes | No | Inelastic* |
| Fixed window | Yes | Yes | Elastic |
| App. limited | No | No | Inelastic |
| Const. stream | No | No | Inelastic |

Table 4.2: Classification by Nimbus.

(with Cubic as the TCP-competitive protocol) achieves similar throughput to Cubic when competing against BBR (Appendix §G.2.2).

Some rate-based protocols may not react on RTT timescales. For example, Nimbus in its default configuration classifies PCC-Vivace as inelastic because it does not react quickly enough to Nimbus's pulses. Increasing the pulse duration helps Nimbus to correctly classify such flows as elastic (Appendix G.2.3). Increasing the pulse duration might of course also increase queuing delays. Since most elastic traffic today is ACK-clocked, we use a small pulse duration by default. In the future, if rate-based protocols become widely deployed, the pulse duration could be adjusted accordingly.

**When assumptions do not hold:** The elasticity detector assumes a single bottleneck link with fixed capacity and requires an estimate of the bottleneck link rate. We now analyze the detection algorithm in simple scenarios when either of these assumptions are not met. We show that when assumptions break, the algorithm classifies all traffic as elastic (with high likelihood) regardless of it's true elasticity. We also evaluate the performance of NimbusCC in such scenarios (see §4.7.3). As expected, NimbusCC primarily operates in the TCP-competitive mode, achieving similar throughput and delay as the status quo.

*Error in link rate estimation:* If the link rate estimate has too much error, the detector classifies traffic as elastic even if it is inelastic. To understand why, define $\hat{z}(t)$ as the estimate of the cross traffic rate, $z^*(t)$ as the actual cross traffic rate, $\hat{\mu}$ as the estimate

of the link rate, and $\mu^*$ as the actual link rate. Then, from Equation (4.1):

$$\hat{z}(t) = \hat{\mu}\frac{S(t)}{R(t)} - S(t), \qquad z^*(t) = \mu^*\frac{S(t)}{R(t)} - S(t) \tag{4.6}$$

Combining the equations above, we get

$$\hat{z}(t) = \frac{\hat{\mu}}{\mu^*}z^*(t) + \big(\frac{\hat{\mu}}{\mu^*} - 1\big)S(t) \tag{4.7}$$

When the link rate estimate is inaccurate, $\hat{z}(t)$ is a linear combination of the cross traffic rate and the sending rate. As the error in the link rate estimate increases, the contribution of the sending rate to $\hat{z}(t)$ increases. Since the sending rate oscillates at the pulse frequency, $\hat{z}(t)$ also oscillates, and all cross traffic (regardless of its nature) is classified as elastic.

*Time-varying links:* On time-varying links (e.g., wireless links), the elasticity detector cannot obtain an accurate estimate of the link rate and will therefore tend to classify traffic as elastic, for the same reason described above.

*Multiple bottlenecks:* In scenarios with multiple bottleneck links,[6] the elasticity detector breaks, but again in a predictable way. Similar to the incorrect $\mu$ case, the cross traffic estimate is a combination of the actual cross traffic rate and the sending rate in such scenarios, and Nimbus will tend to classify traffic as elastic. To understand why, consider a scenario where a NimbusCC flow is going through two bottleneck links, with rates $\mu_1^*$ and $\mu_2^*$, in series. Let $z_1^*(t)$ and $z_2^*(t)$ be the cross traffic rate on the two links respectively. Let $\hat{z}(t)$ be the cross traffic estimate, and $\hat{\mu}$ be the bottleneck link rate estimate provided to the elasticity detection algorithm. We define $R_1(t)$ and $R_2(t)$ as the rate at which Link 1 and Link 2 dequeue packets from the NimbusCC

[6] We expect such scenarios to be rare, since congestion in the Internet typically occurs at the network edge [99, 53].

flow respectively. Assuming that both links are fully utilized, we have:

$$R_1(t) = \frac{\mu_1^* \cdot S(t)}{z_1^*(t) + S(t)},$$
$$R_2(t) = \frac{\mu_2^* \cdot R_1(t)}{z_2^*(t) + R_1(t)} = \frac{\mu_1^* \cdot \mu_2^* \cdot S(t)}{z_1^*(t) \cdot z_2^*(t) + z_2^*(t) \cdot S(t) + \mu_1^* \cdot S(t)}. \tag{4.8}$$

Since the receive rate of the NimbusCC flow is $R_2(t)$, the cross traffic estimate given by Equation (4.1) is:

$$\hat{z}(t) = \hat{\mu} \cdot \frac{S(t)}{R_2(t)} - S(t) = \hat{\mu} \cdot \frac{z_1^*(t) \cdot z_2^*(t)}{\mu_1^* \cdot \mu_2^*} + S(t) \cdot \left( \frac{\hat{\mu} \cdot z_2^*(t)}{\mu_1^* \cdot \mu_2^*} + \frac{\hat{\mu}}{\mu_2^*} - 1 \right). \tag{4.9}$$

The cross traffic estimate is thus a combination of the real cross traffic rate and the sending rate. Since the sending rate component oscillates at the pulse frequency, Nimbus will detect oscillations at the pulsing frequency in $\hat{z}(t)$ and classify cross traffic as elastic.

**Insufficient share of the bottleneck:** To generate pulses, the detector must control a fraction of the traffic at the bottleneck link ($\geq \mu/12$). If the sender's rate is not high enough, NimbusCC switches to the TCP-competitive mode to safeguard against losing throughput from misclassification. Therefore when there are a large number of flows competing at the bottleneck, each with a tiny share of the link bandwidth, NimbusCC is similar to the status quo. Note that in such scenarios, the cross traffic is more likely to be elastic.

**Prevalence of inelastic cross-traffic.** Nimbus's effectiveness in improving delay on Internet paths depends on how often cross traffic is inelastic. Our experiments on paths between different cloud regions and a small number of residential hosts suggest that scenarios where cross traffic is inelastic might be common (§4.7.5). But quantifying how often they occur would require a large-scale measurement study that is beyond the scope of this paper. Our goal here is not to provide a verdict on this question but to show plausible situations where Nimbus is useful and to characterize its limitations.

## 4.7 Evaluation

We have implemented NimbusCC using CCP [109], which provides a convenient way to express the signal processing operations in user-space code. It uses estimates of $S$, $R$, the RTT, and packet losses from the Linux kernel every 10 ms.

We evaluate our elasticity detection method and NimbusCC. We use the Mahimahi emulator and investigate the performance benefits (§4.7.1) of elasticity detection with realistic workloads, its robustness (§4.7.2), and its behavior in scenarios where the assumptions underlying the method are not met (§4.7.3) or when there are competing flows using the detector (§4.7.4). Unless specified otherwise, the topology in these experiments consists of a single bottleneck link with a stable link rate. Finally, we evaluate the performance of NimbusCC on real Internet paths (§4.7.5).

### 4.7.1 Performance Benefits from Elasticity Detection

We evaluate the delay and throughput benefits of mode switching via elasticity detection using trace-driven emulation. We generate cross traffic from an empirical distribution of flow sizes derived from a wide-area packet trace from CAIDA [34]. This packet trace was collected at an Internet backbone router on January 21, 2016 and contains over 30 million packets recorded over 60 seconds. We generate Cubic cross traffic flows with flow sizes drawn from this data, with flow arrival times generated by a Poisson process to offer a fixed average load to fill 50% of the link (48 Mbit/s). The experiment duration is 360 s and consists of 100,000 cross traffic flows.

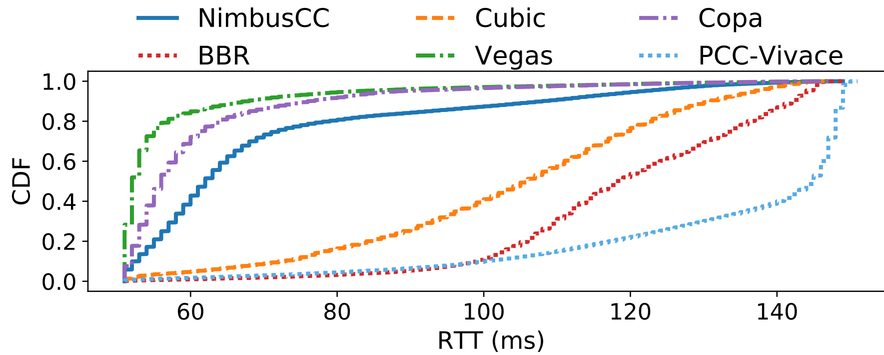
The cross traffic consists of a highly dynamic mix of short and long flows, with a heavy-tailed distribution of flow sizes, ranging from 10 KB to 100 MB (average flow size is 22 KB). Very short flows with size less than the initial congestion window ($<$ 15 KB) are inelastic as they transmit all data at once and don't react to the fluctuations in the available bandwidth, whereas long (backlogged) flows are elastic. The traffic trace consists of periods with *a mix of elastic and inelastic cross traffic,* along with periods with only inelastic cross traffic flows. There is high churn in the number of flows, and the cross traffic exhibits periods of high load that span from a

few RTTs to several minutes.

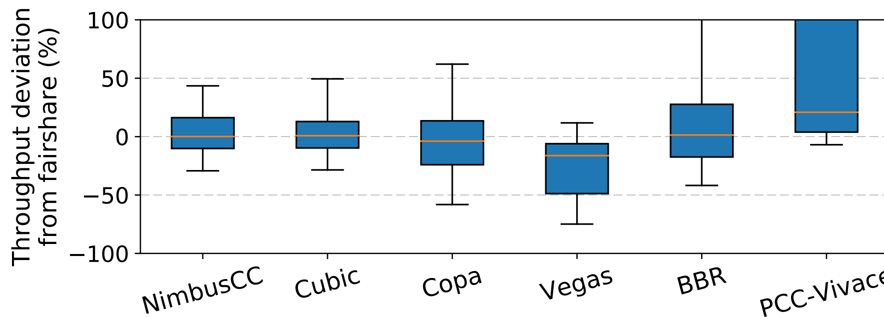
We start one backlogged flow using different congestion control algorithms (NimbusCC, Cubic, Copa, Vegas, PCC-Vivace or BBR) and sharing a 96 Mbit/s bottleneck link with the cross traffic flows. The propagation RTT is 50 ms and the buffer size is 1.2 Mbytes (2 bandwidth-delay products). NimbusCC uses Cubic in the TCP competitive mode and BasicDelay (§4.4.1) in the delay-controlling mode. For BasicDelay we used $\alpha = 0.8$, $\beta = 0.5$ and $d_t = 12.5$ ms.

**NimbusCC reduces delays while achieving fair-share throughput:** Fig. 4-8a shows the distribution of per-packet RTT and Fig. 4-8b shows the deviation from fair-share throughput (over 5-second intervals) for various schemes. NimbusCC and Cubic achieve the lowest deviation from fair share among these schemes. NimbusCC's deviation profile is comparable to Cubic (note that both NimbusCC and Cubic deviate from the fair share since Cubic is not perfectly fair to itself over short time periods). The reason is that NimbusCC correctly switches to Cubic mode in the presence of elastic flows. Additionally, by switching to delay-controlling mode in the absence of elastic flows, NimbusCC achieves lower RTTs, with a median delay only 10 ms higher than Vegas and >50 ms lower than Cubic and BBR.

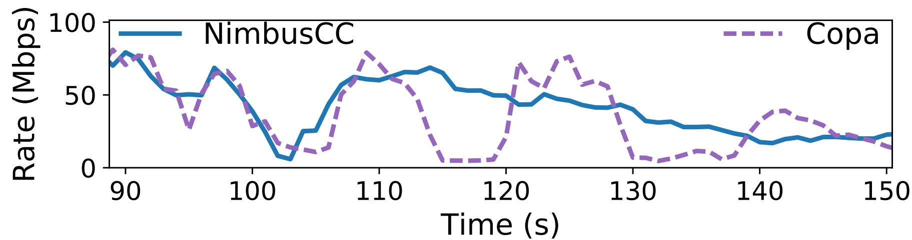
**Cost of incorrect mode-switching:** Copa has a slightly lower median delay than Nimbus, but at a high cost: its throughput is significantly lower than the fair-share at the $10^{\text{th}}$ and $25^{\text{th}}$ percentiles (corresponding to times with significant elastic traffic). Fig. 4-8c shows this more clearly, comparing the rates of NimbusCC and Copa during a 60-second interval. Because of the high variations in the cross traffic rate, the queuing delay can drop below Copa's detection threshold even in the presence of elastic flows (e.g., due to departure of other cross flows). Copa often incorrectly operates in its default delay-control mode against elastic cross traffic (e.g., 115–120, 130–140 s). These incorrect switches cause Copa to nearly stop sending. Since the elastic flows competing with Copa in such periods achieve a higher throughput than the same flows against NimbusCC (which attains the fairshare rate), they complete more quickly, freeing up bandwidth that Copa grabs subsequently. This is why in periods like 120–130 s, which immediately follow a low-rate period for Copa, it achieves a




(a) NimbusCC reduces delay relative to Cubic, BBR and PCC-Vivace. It has higher delays than Copa and Vegas, but those two schemes have lower throughput than the fair share against elastic cross traffic (see figure below).




(b) Deviation in throughput from fair-share. NimbusCC and Cubic achieve highest fairness. Vegas and Copa deviate from fair-share at lower percentiles and lose throughput; BBR and PCC-Vivace are significantly higher than fair share. Midline is the median, the box edges are the 25%ile and 75%ile, the whisker notches are 10%ile and 90%ile.




(c) Copa incorrectly switches to its default delay-control mode even when competing against elastic traffic, unlike NimbusCC.

Figure 4-8: Performance of NimbusCC on a cross traffic workload derived from a packet trace collected at a WAN router.

higher rate than NimbusCC. Also, since NimbusCC competes fairly with elastic flows rather than yielding bandwidth, it has a slightly higher delay than Copa at the tail.

While both schemes achieve the same overall average throughput[7], NimbusCC is better suited to applications that value stable bandwidth, e.g., video streaming, interactive web browsing, online gaming, etc. In such applications, sending at a very low rate for several seconds when cross traffic has elastic flows is unacceptable. Note that elastic traffic was present for only about 25% of the duration of this experiment with Copa, which is why we see Copa under-performing only at the lower percentiles.

**NimbusCC helps cross traffic.** The 95$^{th}$ percentile flow completion time (FCT) of cross traffic flows reduces by 3-4× compared to BBR, and 1.3× compared to Cubic for short ($\leq$ 15 KB) flows (Appendix G.1). In contrast, PCC-Vivace is unfair to the background flows (positive deviation from fairshare). It grabs significantly more bandwidth than all the other schemes and keeps the buffer near-full more than half the time. The result is that many background flows do not complete, and their completion times are over 100× worse than with other schemes. PCC-Vivace also shows higher delays that any other scheme; the median delay is 90 ms higher than NimbusCC.

**Elasticity detection is accurate:** To define ground truth, we note that short flows ($<$ 10 packets) transmit all data at once, without any rate adjustments. We thus classify a cross traffic flow as elastic if it is larger than the initial congestion window of 10 packets, finishing in greater than a RTT.

The top chart in Fig. 4-9 shows the fraction of bytes belonging to elastic flows as a function of time. The bottom chart shows the output of the elasticity detector with the dashed threshold line at $\eta = 2$. The shading corresponds to periods when NimbusCC is in delay-control mode. Shaded regions correlate well with the periods when the true fraction of elastic traffic is low (e.g., $< 0.3$), while white regions correlate well with periods when the elastic fraction is high. Unlike Copa, our elasticity detector observes fluctuations in cross traffic *over a period of time in the frequency domain*,

[7] Any work-conserving scheme will achieve the same throughput in this experiment because the cross traffic sends a fixed number of bytes.

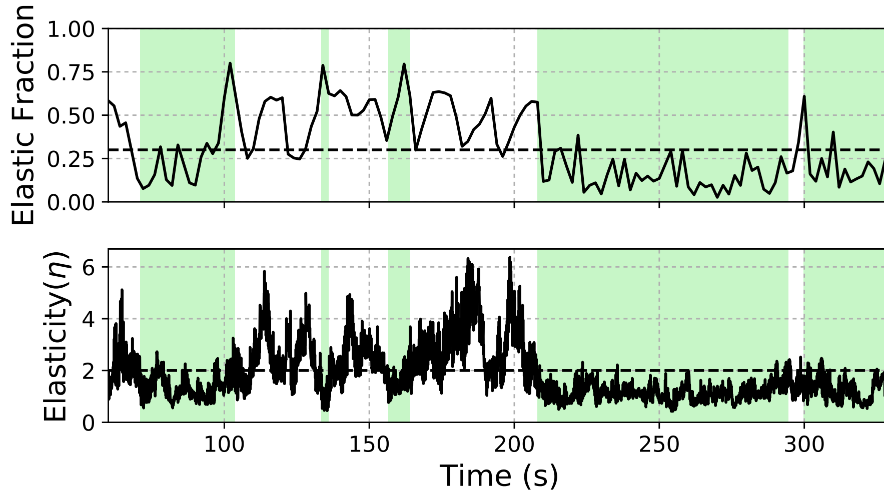



Figure 4-9: The elasticity metric closely tracks elastic cross traffic (ground truth measured independently from the rate of ACK-clocked flows). Green-shaded regions indicate inelastic periods.

and the accuracy is less susceptible to variations in the cross traffic rate. Despite the churn in cross traffic flows, the overall accuracy of our elasticity detector is over 90% when the fraction of the elastic traffic is high ($> 30\%$). When the fraction of elastic traffic is low, NimbusCC operates primarily in the delay-controlling mode. In this case, the elastic flows in the cross traffic are relatively short. Such short elastic flows do not last long enough to grab bandwidth from the NimbusCC flow.

**Performance against different congestion control protocols:** We repeat the experiment in Fig. 4-8 but with cross traffic consisting of an equal (on average) mix of Cubic, NewReno and BBR flows. The results are similar to the previous experiment: NimbusCC achieves lower delays than Cubic for a similar throughput profile, while Copa and Vegas lose throughput when cross traffic is elastic. The reason is that, regardless of the congestion control protocol, the elastic cross traffic flows react to Nimbus's pulses, and can therefore be classified correctly (Appendix G.2.1).

**Performance with video cross traffic:** Video streams can be application-limited (e.g., when the client playback buffer is full) or network-limited (e.g., when the client is downloading a high-bitrate chunk) at different points in time. Therefore video traffic can exhibit both inelastic and elastic behavior. We compare the performance of congestion control algorithms running against cross traffic consisting of a 4k DASH [95] video stream using Cubic on a 48 Mbit/s link with 50 ms RTT for 80 seconds. Fig. 4-

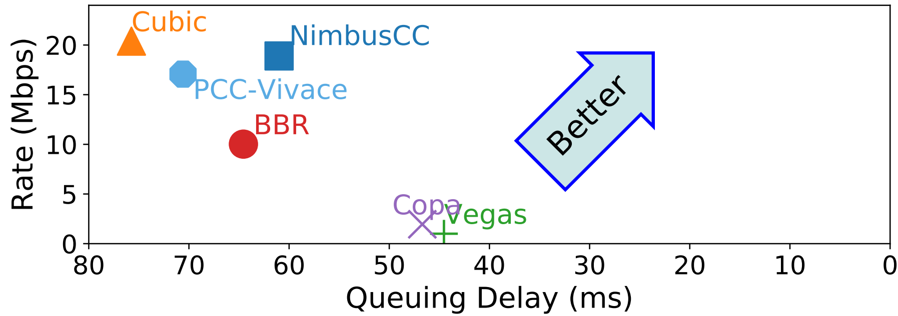



Figure 4-10: Mean throughput and queuing delay (lower delay on the right) with video cross traffic. NimbusCC achieves similar throughput as Cubic but reduces delays and performs better than the other schemes. Copa and Vegas achieve low throughput.

10 shows the throughout and delay of the various schemes. Because of effective mode switching, NimbusCC achieves similar throughput as Cubic at 15 ms lower delay. Nimbus recognizes application-limited video traffic as inelastic, allowing the sender to control delays in those cases; it rarely recognizes network-limited elastic traffic as inelastic, so does not wrongly reduce its rate as Copa does. Note that the figure shows the rate of a backlogged flow competing against video cross traffic; the total link utilization with all the schemes was at least 90%.

### 4.7.2 Robustness of Elasticity Detection

We evaluate the robustness of Nimbus under a variety of network and traffic conditions. Unless specified otherwise, we run NimbusCC as a backlogged flow on a 96 Mbit/s bottleneck link with a 50 ms propagation RTT and a 100 ms drop-tail buffer (2 BDP). We consider three categories of synthetic cross traffic sharing the link with NimbusCC: (i) inelastic Poisson-distributed traffic; (ii) fully elastic traffic (backlogged NewReno flows); and (iii) an equal mix of inelastic and elastic traffic. The duration of each experiment is 120 seconds. We evaluate *accuracy*: the fraction of time Nimbus correctly detects the presence of elastic cross traffic. For each experiment, we report the mean accuracy of the detector across 5 runs.

**Impact of cross traffic RTT:** We vary the cross traffic's minimum RTT from 10 ms to 200 ms ($0.2 - 4\times$ NimbusCC's RTT). We find that varying cross traffic RTT does not reduce accuracy. For purely inelastic and purely elastic traffic, the accuracy

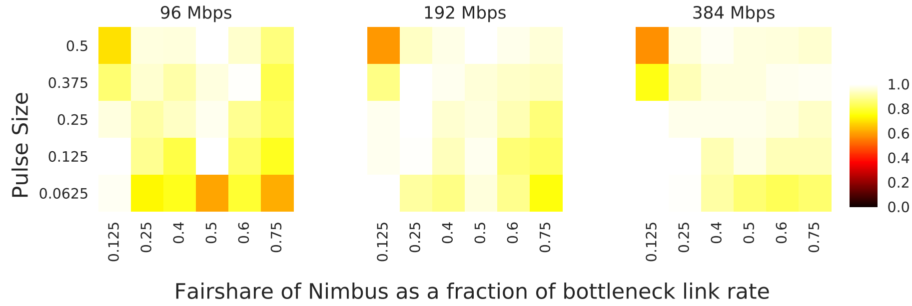



Figure 4-11: Nimbus is robust to variations in link bandwidth and fraction of traffic controlled by it. The accuracy is high even when the fraction of traffic under control is small. Increasing pulse size increases robustness.

is more than 98% in all cases, while for mixed traffic, the accuracy is more than 85% in all cases (a random guess would have only achieved 50%). Regardless of the cross traffic RTT, the elastic flows respond to fluctuations created by Nimbus, generating a peak in the cross traffic FFT at the oscillation frequency. The cross traffic's RTT *affects the phase, but not the amplitude* of the peak in the FFT.

**A mix of RTTs in the cross traffic:** We vary the number of elastic cross traffic flows from 1 to 5, where the RTT of $n^{\text{th}}$ flow in $20 \cdot n$ ms. In case the cross traffic contains elastic flows, all the elastic flows oscillate at Nimbus's pulse frequency. As a result, the sum of the rates of these elastic flows also oscillates,[8] and the traffic is correctly classified as elastic. For purely elastic and inelastic traffic, Nimbus achieves an average accuracy of 98% across 5 runs, while for mixed traffic, the mean accuracy is greater than 90% in all cases. In other words, heterogeneity in RTTs of cross-flows does not degrade the accuracy of elasticity detection.

**Pulse size, link rate, and offered cross traffic load:** We perform a multi-factor experiment varying Nimbus's pulse size from 1/16 to 1/2 the link rate, the fair share of the bottleneck link rate from 12.5%—75% (by varying the cross traffic load), bottleneck link rates set to 96, 192, and 384 Mbit/s. The accuracy for purely elastic cross traffic is always higher than 95%. while the average accuracy over all the points for the other two traffic mixes is more than 90%. Fig. 4-11 shows the

[8]Since the RTTs are different, the elastic flows' oscillations will differ in phase and the oscillations could in theory cancel each other out leading to mis-classification, but it requires specific combinations of RTT and is unlikely.

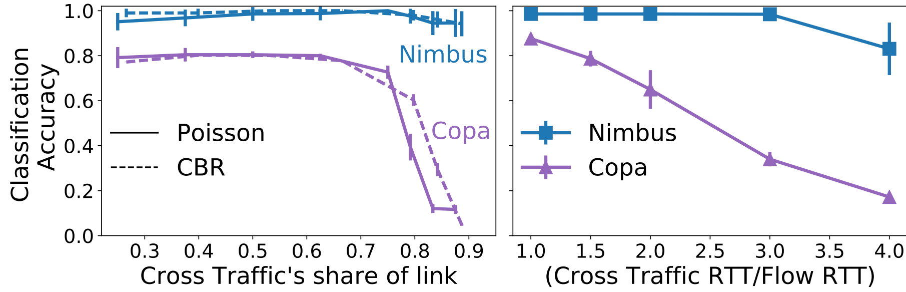



Figure 4-12: Nimbus is more accurate than Copa when (i) inelastic cross traffic occupies a large fraction of the link (left); (ii) elastic cross traffic has higher RTT than the flow's RTT (right).

average detection accuracy over the other two categories of cross traffic (mix + purely inelastic). The classification accuracy is not sensitive to cross traffic load. Nimbus's *use of asymmetric pulses* enables a sender to create fluctuations in the cross traffic even when the sending rate is low. As a result, the detection accuracy remains high under high cross traffic load.

In general, increasing the pulse sizes improves accuracy because the elasticity detector can create a more easily observable change in the cross traffic sending rates. An increase in the link rate results in higher accuracy for a given pulse size and Nimbus link share because the variance in the rates of inelastic Poisson cross traffic reduces with increasing cross traffic sending rate, reducing the number of false peaks in the cross traffic FFT. However, the elasticity detector has low accuracy ($\sim$60%) when it uses high pulse sizes and controls a low fraction of the link rate. We believe that this is due to a quirk in the way the Linux networking stack reports round-trip time measurements under sudden sending rate changes.

**Comparison with Copa:** We now compare the classification accuracy of Nimbus with Copa. First, we generate inelastic cross traffic at different rates and measure the accuracy. We consider both constant-bit-rate (CBR) and Poisson cross traffic.

Fig. 4-12 (left) shows that Nimbus has high accuracy in all cases, but Copa's accuracy drops sharply when the cross traffic occupies over 80% of the link. This result highlights a pitfall of Copa's approach: setting an operating mode based on the absolute value of queueing delays is problematic. With a high inelastic cross traffic load, Copa is unable to drain the queue quickly enough (i.e., every 5 RTTs),

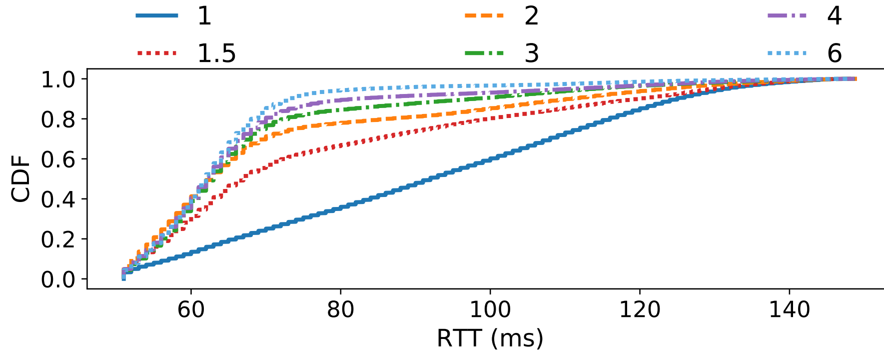
which throws off its detector. In contrast, the elasticity detector estimates elasticity through delay variations caused by its pulses, and is more robust.

Next, we ran a backlogged NimbusCC or Copa flow competing against a backlogged NewReno flow. We vary the RTT of the NewReno flow between $1-4\times$ the RTT of the NimbusCC/Copa flow. Fig. 4-12 (right) shows that Copa's accuracy degrades as the RTT of the cross traffic increases; Nimbus's accuracy is much higher, dropping only slightly when the cross traffic RTT is $4\times$ larger than NimbusCC.

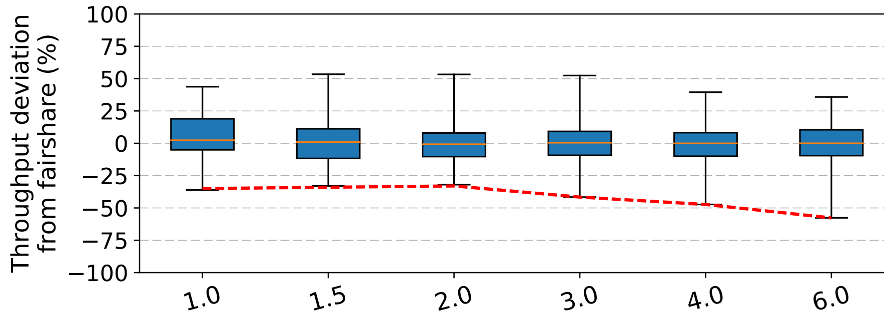
An elastic cross-flow with a large RTT increases its rate slowly enough to evade detection by Copa. Therefore, Copa drains the queue as it expects and concludes the absence of non-Copa cross traffic. This behavior continues until the cross-flow has grown to offer a load close to the link rate, when it starts interfering with Copa's queue draining. By contrast, Nimbus is more robust since it is based on the time series of variations of the cross traffic rate. Moreover, even when the classification accuracy for Copa is higher, it makes frequent mode-switches and is suspectible to lose throughput against elastic traffic. Appendix G.3 shows the throughput and queueing delay dynamics of Copa and NimbusCC.

**Impact of $\eta_{thresh}$:** We evaluate the sensitivity of Nimbus's performance to the detection threshold parameter ($\eta_{thresh}$). We repeat the experiment in Fig. 4-8 but vary the detection threshold from 1 to 6. Fig. 4-13 shows the performance as a function of $\eta_{thresh}$. $\eta_{thresh}$ presents a performance trade-off. With a high $\eta_{thresh}$, Nimbus classifies traffic as inelastic more frequently. NimbusCC operates in delay-controlling mode a higher fraction of the time reducing delays. However, at times NimbusCC operates in the delay-controlling mode incorrectly against elastic cross traffic, losing throughput. This affect can be seen prominently at the lowest percentiles in the throughput profile. Similarly, a small $\eta_{thresh}$ causes NimbusCC to miss opportunities for controlling delays against inelastic traffic, but NimbusCC doesn't lose throughput against elastic traffic.

**Further robustness results:** In Appendix G.4, we explore variations in buffer size, RTT of the Nimbus flow, and presence of active queue management schemes, and we show that Nimbus is robust to these settings. In Appendix G.5, we demonstrate




(a) Per-packet RTT



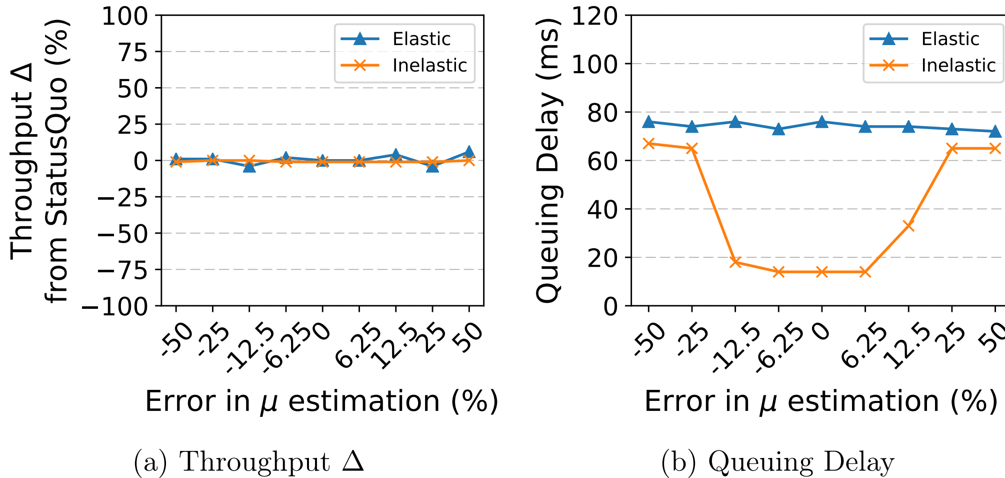

(b) Deviation from fairshare throughput

Figure 4-13: **Impact of $\eta_{thresh}$** — With a high $\eta_{thresh}$, NimbusCC operates in delay-controlling mode more often, reducing delays but losing throughput against elastic cross traffic (see the $10^{\text{th}}$ percentile in the throughput profile, shown in red)

the versatility of NimbusCC in supporting different combinations of algorithms for its delay-controlling and TCP-competitive modes.

### 4.7.3 Performance When Assumptions Do Not Hold

The elasticity detector is designed for a single bottleneck link with stable rate and requires the knowledge about the bottleneck link rate. What happens when these assumptions do not hold? We compare the throughput of the NimbusCC to the baseline TCP-competitive protocol (referred to as "status quo" in this section) and also report the per-packet queuing delay. We evaluate the performance for two cross traffic scenarios: (i) inelastic Poisson-distributed traffic, and (ii) fully elastic traffic (backlogged NewReno flows). As explained in §4.6, when the assumptions break, NimbusCC mostly classifies the cross traffic as elastic and uses the TCP-competitive




(a) Throughput Δ (b) Queuing Delay

Figure 4-14: **Impact of incorrect $\mu$** — When the error is high, all the traffic is classified as elastic and NimbusCC operates in TCP-competitive mode.

protocol, therefore achieving similar throughput to the status quo.

**Error in link rate estimation:** We explicitly supply an incorrect link rate estimate to NimbusCC. We vary the error in the link rate estimate from −50% to +50% of the real link rate value (96 Mbit/s). Fig. 4-14 reports the average results across 5 runs (120 s each). NimbusCC achieves throughput similar to status quo in all the scenarios. The classification accuracy is high ($>95\%$) for elastic cross traffic in all the cases. When the error rate is high ($>12.5\%$), inelastic cross traffic is also classified as elastic and NimbusCC fails to control delays.

**Performance on time-varying links:** The experiment consists of a single bottleneck with a time-varying link rate. We model the rate of the bottleneck link as a random walk; the rate can change by ± 20 Mbit/s every second. Table 4.3 summarizes the results across 5 such traces. NimbusCC achieves throughput within 1% of the status quo. However, the classification accuracy is low against inelastic traffic and the queuing delay is high (though no higher than the status quo). On time-varying links, inferring the bottleneck link rate is hard in an end-to-end manner [58, 57]. When NimbusCC's estimate of the link rate[9] differs substantially from the bottleneck link rate, the cross traffic estimator fails, and NimbusCC simply operates in the TCP-competitive mode regardless of the cross traffic (§4.6). Thus, on time-varying

[9] We use the average bandwidth as the link rate estimate for these experiments.

| Cross Traffic | Throughout Δ | Q Delay | Classification Accuracy |
|---|---|---|---|
| Elastic | -1% | 103 ms | 95% |
| Inelastic | -1% | 70 ms | 39% |

Table 4.3: Performance on time-varying links.

| Cross Traffic (Link 1) | Cross Traffic (Link 2) | Throughout Δ | Q Delay |
|---|---|---|---|
| Elastic | Elastic | -3% | 146 ms |
| Inelastic | Elastic | 5% | 76 ms |
| Elastic | Inelastic | -14% | 91 ms |
| Inelastic | Inelastic | 0% | 14 ms |

Table 4.4: Performance on a topology with multiple bottlenecks.

bottleneck links NimbusCC is safe to run and will not lose throughput relative to the status quo, but it might lose opportunities to control delays.

**Multiple bottleneck links:** We evaluate NimbusCC on a topology with multiple bottleneck links. The topology consists of two links. Link 1's bandwidth is 192 Mbit/s and link 2's bandwidth is 96 Mbit/s. The experiment consists of a single NimbusCC flow going through the two links. The propagation RTT is 50ms. Each link has either elastic or inelastic cross traffic (a cross traffic flow only traverses one of the two links). Depending on the instantaneous rate of the cross traffic at each link, the bottleneck could either be both links or one of the links (the bottleneck can change within an experiment). Table 4.4 shows the throughput delta relative to status quo and the total queuing delay across both links, averaged over 5 runs of each scenario. NimbusCC achieves throughput comparable to the status quo (within 15%) in all the cases. In scenarios where either of the links contained elastic cross traffic NimbusCC stayed in the TCP-competitive mode majority of the time ($>$ 85%). When both links had inelastic cross traffic, NimbusCC uses the delay-controlling mode and is able to reduce delays; note that in this case the slower link (link 2) is the bottleneck.

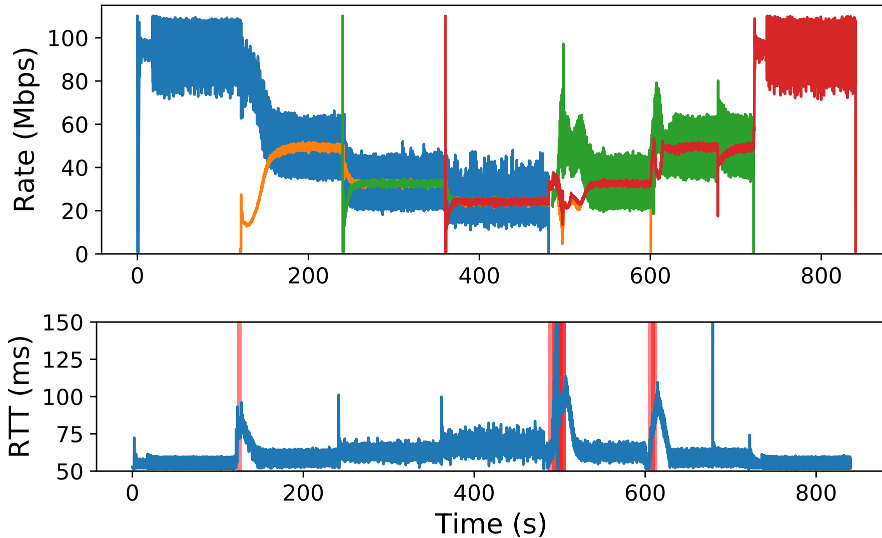



Figure 4-15: **Multiple competing NimbusCC flows** — Multiple NimbusCC flows achieve fair sharing of a bottleneck link (top graph). There is at most one pulser flow at any time; identified by its rate variations. Together, the flows achieve low delays by staying in delay mode for most of the duration (bottom graph). The red background shading shows when a NimbusCC flow was (incorrectly) in competitive mode

### 4.7.4 Elasticity Detection with Multiple NimbusCC Flows

Can multiple flows run elasticity detection and share a bottleneck link fairly with each other and with cross traffic?

We run NimbusCC with Vegas as its delay-control algorithm. Fig. 4-15 demonstrates how NimbusCC flows react as other NimbusCC flows arrive and leave (there is no other cross traffic). Four flows arrive at a link with rate 96 Mbit/s and round-trip time 50 ms. Each flow begins 120 s after the last one began, and lasts for 480 s. The top half shows the rates achieved by the four flows over time. Each new flow begins as a watcher. If the new flow detects a pulser ($t = 120, 240, 360$ s), it remains a watcher. If the pulser goes away or a new flow fails to detect a pulser, one of the watchers becomes a pulser ($t = 480, 720$ s). The pulser can be identified visually by its rate variations.

The flows share the link rate equally. The bottom half of the figure shows the achieved delays with red background shading to indicate when one of the flows is (incorrectly) in competitive-mode. The flows maintain low RTTs and stay in delay-mode for most of the time.

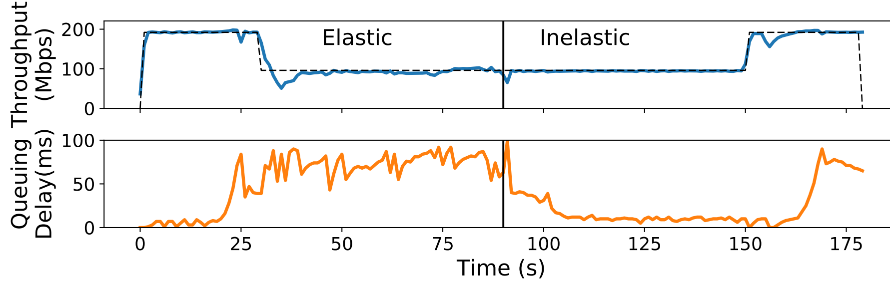


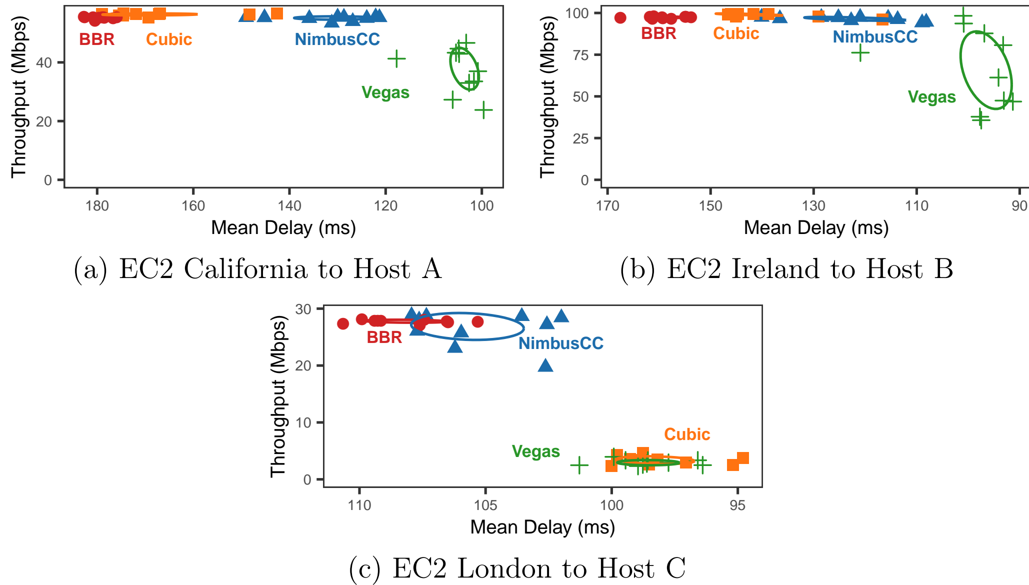

Figure 4-16: **Multiple NimbusCC flows and other cross traffic** — There are 3 NimbusCC flows throughout. Cross traffic in 30-90 s is elastic and made up of 3 Cubic flows. Cross traffic in 90-150 s is inelastic and made up of a 96 Mbit/s constant bit-rate stream. NimbusCC flows achieve their fair share (top) while achieving low delays in the absence of elastic cross traffic (bottom).

| $\kappa$ | Time to elect a pulser | Fraction of time with multiple pulsers |
|---|---|---|
| 0.5 | 19.8 s | 0% |
| 0.75 | 15.3 s | 4.6% |
| 1 | 8.0 s | 9.3% |
| 1.5 | 5.4 s | 15.4% |
| 2 | 2.7 s | 29.3% |

Table 4.5: Impact of $\kappa$.

Fig. 4-16 demonstrates multiple NimbusCC flows switching in the presence of cross traffic. We run three NimbusCC flows on an emulated 192 Mbit/s link with a propagation delay of 50 ms. In the first 90 s, the cross traffic is elastic (three Cubic flows), and for the rest of the experiment, the cross traffic is inelastic (96 Mbit/s constant bit-rate). The top graph shows the total rate of the three NimbusCC flows, along with a reference line for the fair-share rate of the aggregate. The graph at the bottom shows the measured queuing delays. NimbusCC shares the link fairly with other cross traffic, and achieves low delays by staying in the delay-controlling mode in the absence of elastic cross traffic for most of the experiment.

**Impact of $\kappa$.** We evaluate the impact of $\kappa$ on pulser election. In the experiment eight NimbusCC flows start simultaneously, the bottleneck link is 96 Mbits/s and the base RTT is 50ms. We report the time it takes to elect a pulser and the fraction of time there were multiple pulsers. Table 4.5 summarizes the average values across 20 runs (45 s each). As expected, increasing $\kappa$ reduces the time to elect a pulser, but also increases the chances of multiple pulsers being elected.




(a) EC2 California to Host A

(b) EC2 Ireland to Host B

(c) EC2 London to Host C

Figure 4-17: **Performance on three example Internet paths** — The $x$ axis is inverted; better performance is up and to the right. On paths with buffering and no drops, ((a) and (b)), NimbusCC achieves the same throughput as BBR and Cubic but reduces delays significantly. On paths with significant packet drops (c), Cubic suffers but NimbusCC achieves high throughput.

### 4.7.5 Testing on Internet Paths

We ran NimbusCC on 25 paths between five senders and five receivers. The servers were EC2 instances located in California, London, Frankfurt, Ireland, and Paris, all with 10 Gbit/s links.[10] The receivers were residential hosts, connected directly to the Internet router via 1 Gbit/s ethernet links. We verified that the bottleneck in each case was not the server's Internet link or the ethernet access link. While we cannot be certain, we believe that the bottleneck link on these paths was at last-hop internet service provider.

We initiated bulk data transfers using NimbusCC, Cubic, BBR, and Vegas. We ran one-minute experiments over five hours on each path, and measured the achieved mean throughput and mean delay. Fig. 4-17 shows throughput and delays over three of the paths. The $x$ (delay) axis is inverted; better performance is up and to the right. NimbusCC achieves high throughput comparable to BBR in all cases, at significantly lower delays. Cubic attains high throughput on paths with deep buffers (Fig. 4-17a

[10]We also ran experiments between pairs of cloud servers but we observed no congestion on any such path.

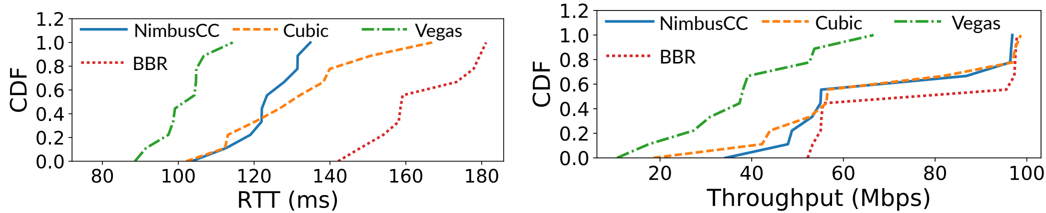



Figure 4-18: **Paths with queuing** — NimbusCC reduces the RTT compared to Cubic and BBR (upto 50ms), at similar throughput.

and Fig. 4-17b), but not on paths with packet drops or policers (Fig. 4-17c). [11] Vegas attains poor throughput on these paths because it does not keep the bottleneck link busy and is unable to compete with elastic cross traffic. These trends show the utility of elasticity detection on Internet paths: it is possible to achieve high throughput and low delays over the Internet using delay-control algorithms with the ability to switch to a different competitive mode when required.

Fig. 4-18 summarizes the results on paths with larger buffers. NimbusCC's throughput is similar to Cubic's and 10% lower than BBR's but with much lower delays (40–50 ms lower than BBR). NimbusCC's lower mean delay indicates that the cross traffic at the bottleneck link often did not contain long backlogged elastic flows. We believe that during these periods, the cross traffic was application-limited (e.g., video streams where the available bandwidth exceeded the maximum video bitrate). It is in such cross-traffic scenarios that NimbusCC provides the most benefits in terms of delay reduction while still achieving high throughput.

## 4.8 Conclusion

In this chapter, we showed that characterizing the *elasticity* of cross traffic is a useful building block for improving congestion control. We introduced a method for detecting and quantifying the elasticity of cross traffic. Our detection technique, Nimbus,

[11] For each path, we ran experiments in the night (when the cross traffic load was likely close to 0) and compared the throughput of Cubic and BBR. On the paths where the Cubic throughput was lower consistently across runs, we observed frequent packet drops without much variation in RTT. We inferred that the drops either occur at a shallow-buffered bottleneck link or a policer, both of which are known to hurt the throughput of Cubic [48, 36].

uses asymmetric sinusoidal pulses to modulate the sending rate and observes the frequency response of the cross traffic rate, taking advantage of the property that elastic cross traffic can be made to oscillate at a pulsing frequency set by sender. Nimbus relies only on end-to-end rate and delay measurements and requires no changes to the routers. We presented several experiments to demonstrate the robustness and accuracy of our proposed method. We also showed that elasticity detection enables transport protocols to combine the best aspects of delay-control methods while being competitive with elastic flows when necessary. We found that our proposed methods are beneficial not only on a variety of emulated conditions that model realistic workloads, but also on a collection of 25 real-world Internet paths.

# Chapter 5

# Conclusion

In this dissertation, we present high performance, practical congestion control solutions for three highly variable network environments: (1) Wireless Networks; (2) Datacenter Networks; (3) Wide-area Internet. The key philosophy behind these solutions is to design custom feedback mechanisms in each environment, that provide accurate and timely information on how to adapt to variations in the network conditions. To this end, we employ two principles in our design:

**Leverage in-network support practically:** By enabling routers/switches in the network to generate or react to high granularity feedback, we can better adapt to variations in the network conditions. In past, researchers have proposed many congestion control solutions that leverage such in-network support. Despite the advantages, most of these solutions haven't seen much adoption as they are hard to deploy. We make the following contributions to overcome the deployment challenges associated with in-network solutions.

*Single-bit explicit feedback:* ABC proposes a simple technique that uses only single-bit of per-packet feedback to signal both increases and decreases to the sender's congestion window. Single-bit feedback can be implemented on top of the existing ECN infrastrucutre. In contrast, existing explicit schemes use multi-bit per-packet feedback and thus require major changes to packet headers, routers, and endpoints for deployment.

*Practical per-hop per-flow flow control:* BFC is the first practical congestion control architecture for modern datacenter networks that achieves an approximation of per-hop per-flow flow control. BFC overcomes the programmability limitations of modern datacenter switches by using a limited amount of switch state, modest number of switch queues, and only simple constant-time per-packet switch operations.

**Compute feedback accurately:** The exact specifics of how the feedback is calculated can make a significant impact on performance. We make the following contributions to compute feedback more accurately.

*Use dequeue rate to compute explicit feedback:* Existing explicit schemes use the enqueue rate at the router to determine the feedback. In contrast, an ABC router compares the dequeue rate to the link capacity to compute the accel-brake feedback. This change is rooted in the observation that, for an ACK-clocked protocol like ABC, the current dequeue rate provides an accurate prediction of the future enqueue rate, one RTT in advance. This enables the ABC sender to better adapt its rate to the link capacity.

*Elasticity detection:* Nimbus is a robust end-to-end technique to rigorously characterize whether the competing cross-traffic is elastic (i.e., competing to grab more bandwidth) or not. To detect elasticity, Nimbus modulates the sending rate at a given frequency and observes whether the cross-traffic rate responds to variations at the same frequency. In contrast to observing cross-traffic response in the time domain, using frequency domain makes our mechanism robust to a variety of traffic and network conditions. Elasticity detection enables a sender to reduce delays while maintaining high throughput against elastic cross-traffic.

## 5.1 Future Work

So far, we did not consider feedback to or from applications in our design. The feedback mechanisms we propose are agnostic to the needs of the individual applications in the network and only focus on the generic goal of high throughput and low latency for all flows. For example, ABC does not take into account the fact that low packet

latency is more crucial for video conferencing compared to a file transfer. We can improve performance by incorporating applications in the feedback mechanism.

Feedback from applications on their communication requirements to the network can guide congestion control or even scheduling decisions at the sender and bottleneck links. Similarly, exposing congestion feedback from the network to an application can help it better adapt to variations in the network conditions. For example, typically, video conferencing applications try to infer the network conditions indirectly to adjust the video bitrate. With explicit feedback from the network, they can better adjust their bitrates to avoid video stalls while maintaining high utilization. We believe that network environments with a single authority of control offer enticing opportunities for exploring such solutions.

**SLO aware scheduling for datacenters:** Applications running in datacenters often express communication requirements in the form of Service Level Objectives (SLOs). Achieving these SLOs is one of the primary goals of the network operator. Whether an application's SLO can be met or not depends on a variety of factors such as the link capacities, scheduling at the switches, traffic patterns in the network, the SLO itself, etc. It is possible to achieve better performance for applications by taking into account these factors.

We are currently working on such a system [148]. The key idea is to dynamically adapt scheduling policy at the switches based on the observed traffic patterns and SLO of the applications running in the network. In our initial evaluation, we find that such a system with dynamic weighted fair-queueing among traffic classes can provide performance gains over the status quo scheduling policies. Some of the questions we are thinking about are: What are the right statistics to characterize the traffic pattern: is it traffic load, flow size distribution, flow inter-arrival pattern, or some combination of these? How fast can such a system adapt to the changes in the traffic pattern? In a datacenter, what is the rate at which traffic patterns change, is the timescale seconds or hours?

**Edge computing in cellular networks:** Cellular operators are increasingly deploying edge datacenters near cellular base stations to host applications such as online

gaming. Further, to increase computation efficiency and programmability, cellular operators are also trying to move various base station operations such as scheduling decisions to the edge datacenters. Control over both the application servers and the base station functionality implies that we can redesign both: (1) Scheduling at the base station to take into account the needs of the application; (2) Applications to take into account the available bandwidth. Such a design can potentially enable guaranteed performance for mission critical applications such as online machine learning inference for self-driving cars.

# Appendix A

# ABC: Stability Analysis

This appendix establishes the stability bounds for ABC's control algorithm (Theorem 1).

**Model:** Consider a single ABC link, traversed by $N$ ABC flows. Let $\mu(t)$ be the link capacity at time $t$. As $\mu(t)$ can be time-varying, we define stability as follows. Suppose that at some time $t_0$, $\mu(t)$ stops changing, i.e., for $t > t_0$ $\mu(t) = \mu$ for some constant $\mu$. We aim to derive conditions on ABC's parameters which guarantee that the aggregate rate of the senders and the queue size at the routers will converge to certain fixed-point values (to be determined) as $t \to \infty$.

Let $\tau$ be the common round-trip propagation delay on the path for all users. For additive increase (§2.3.1.3), assume that each sender increases its congestion window by 1 every $l$ seconds. Let $f(t)$ be the fraction of packets marked accelerate, and, $cr(t)$ be the dequeue rate at the ABC router at time $t$. Let $\tau_r$ be time it takes accel-brake marks leaving the ABC router to reach the sender. Assuming that there are no queues other than at the ABC router, $\tau_r$ will be the sum of the propagation delay between ABC router and the receiver and the propagation delay between receiver and the senders. The aggregate incoming rate of ACKs across all the senders at time $t$, $R(t)$, will be equal to the dequeue rate at the router at time $t - \tau_r$:

$$R(t) \; = \; cr(t - \tau_r). \tag{A.1}$$

In response to an accelerate, a sender will send 2 packets, and, for a brake, a sender won't send anything. In addition to responding to accel-brakes, each sender will also send an additional packet every $l$ seconds (because of AI). Therefore, the aggregate sending rate for all senders at time $t$, $S(t)$, will be

$$\begin{aligned} S(t) &= R(t) \cdot 2 \cdot f(t-\tau_r) + \frac{N}{l} \\ &= 2cr(t-\tau_r)f(t-\tau_r) + \frac{N}{l}. \end{aligned} \tag{A.2}$$

Substituting $f(t-\tau_r)$ from Equation (2.2), we get

$$S(t) = tr(t-\tau_r) + \frac{N}{l}. \tag{A.3}$$

Let $\tau_f$ be the propagation delay between a sender and the ABC router, and $eq(t)$ be the enqueue rate at the router at time $t$. Then $eq(t)$ is given by

$$\begin{aligned} eq(t) &= S(t-\tau_f) \\ &= tr(t-(\tau_r+\tau_f)) + \frac{N}{l} \\ &= tr(t-\tau) + \frac{N}{l}. \end{aligned} \tag{A.4}$$

Here, $\tau = \tau_r + \tau_f$ is the round-trip propagation delay.

Let $q(t)$ be the queue size, and, $x(t)$ be the queuing delay at time $t$:

$$x(t) = \frac{q(t)}{\mu}.$$

Ignoring the boundary conditions for simplicity ($q(t)$ must be $\geq 0$), the queue length

has the following dynamics:

$$\begin{aligned}\dot{q}(t) &= eq(t) - \mu \\ &= tr(t-\tau) + \frac{N}{l} - \mu \\ &= \left((\eta - 1)\cdot\mu + \frac{N}{l}\right) - \frac{\mu}{\delta}(x(t-\tau) - d_t)^+,\end{aligned}$$

where in the last step we have used Equation (2.1). Therefore the dynamics of $x(t)$ can be described by:

$$\begin{aligned}\dot{x}(t) &= \left((\eta - 1) + \frac{N}{\mu\cdot l}\right) - \frac{1}{\delta}(x(t-\tau) - d_t)^+ \\ &= A - \frac{1}{\delta}(x(t-\tau) - d_t)^+, \end{aligned} \quad \text{(A.5)}$$

where $A = \left((\eta - 1) + \frac{N}{\mu\cdot l}\right)$, and, A is a constant given a fixed number of flows $N$. The delay-differential equation in Equation (A.5) captures the behavior of the entire system. We use it to analyze the behavior of the queuing delay, $x(t)$, which in turn informs the dynamics of the target rate, $tr(t)$, and enqueue rate, $eq(t)$, using Equations (2.1) and (A.4) respectively.

**Stability:** For stability, we consider two possible scenarios 1) $A < 0$, and 2) $A \geq 0$. We argue the stability in each case.

**Case 1:** $A < 0$. In this case, the stability analysis is straightforward. The fixed point for queuing delay, $x^*$, is 0. From Equation (A.5), we get

$$\dot{x}(t) = A - \frac{1}{\delta}(x(t-\tau) - d_t)^+ \leq A < 0. \quad \text{(A.6)}$$

The above equation implies that the queue delay will decrease at least as fast as $A$. Thus, the queue will go empty in a bounded amount of time. Once the queue is empty, it will remain empty forever, and the enqueue rate will converge to a fixed

value. Using Equation (A.4), the enqueue rate can will converge to

$$\begin{aligned} eq(t) &= tr(t-\tau) + \frac{N}{l} \\ &= \eta\mu + \frac{N}{l} - \frac{\mu}{\delta}(x(t-\tau) - d_t)^+ \\ &= \eta\mu + \frac{N}{l} \\ &= (1+A)\mu. \end{aligned} \tag{A.7}$$

Note that $\eta\mu < (1+A)\mu < \mu$. Since both the enqueue rate and the queuing delay converge to fixed values, the system is stable for any value of $\delta$.

**Case 2:** $A > 0$: The fixed point for the queuing delay in this case is $x^* = A \cdot \delta + d_t$. Let $\tilde{x}(t) = x(t) - x^*$ be the deviation of the queuing delay from its fixed point. Substituting in Equation (A.5), we get

$$\begin{aligned} \dot{\tilde{x}}(t) &= A - \frac{1}{\delta}(\tilde{x}(t-\tau) + A \cdot \delta)^+ \\ &= -max(-A, \frac{1}{\delta}\tilde{x}(t-\tau)) \\ &= -g(\tilde{x}(t-\tau)), \end{aligned} \tag{A.8}$$

where $g(u) = \max(-A, \frac{1}{\delta}u)$ and $A > 0$.

In [145] (Corollary 3.1), Yorke established that delay-differential equations of this type are globally asymptotically stable (i.e., $\tilde{x}(t) \to 0$ as $t \to \infty$ irrespective of the initial condition), if the following conditions are met:

1. $\mathbf{H}_1$**:** g is continuous.
2. $\mathbf{H}_2$**:** There exists some $\alpha$, s.t. $\alpha \cdot u^2 > ug(u) > 0$ for all $u \neq 0$.
3. $\mathbf{H}_3$**:** $\alpha \cdot \tau < \frac{3}{2}$.

The function $g(\cdot)$ trivially satisfies $\mathbf{H}_1$. $\mathbf{H}_2$ holds for any $\alpha \in (\frac{1}{\delta}, \infty)$. Therefore, there exists an $\alpha \in (\frac{1}{\delta}, \infty)$ that satisfies both $\mathbf{H}_2$ and $\mathbf{H}_3$ if

$$\frac{1}{\delta} \cdot \tau < \frac{3}{2} \implies \delta > \frac{2}{3} \cdot \tau. \tag{A.9}$$

This proves that ABC's control rule is asymptotically stable if Equation (A.9) holds. Having established that $x(t)$ converges to $x^* = A \cdot \delta + d_t$, we can again use Equation (A.4) to derive the fixed point for the enqueue rate:

$$eq(t) = \eta\mu + \frac{N}{l} - \frac{\mu}{\delta}(x(t-\tau) - d_t)^+ \rightarrow \mu, \tag{A.10}$$

as $t \rightarrow \infty$.

Note while, we proved stability assuming that the feedback delay $\tau$ is a constant and the same value for all the senders, the proof works even if the senders have different time-varying feedback delays (see Corollary 3.2 in [145]). The modified stability criterion in this case is $\delta > \frac{2}{3} \cdot \tau^*$, where $\tau^*$ is the maximum feedback delay across all senders.

# Appendix B

# ABC: Miscellaneous Results

## B.1 BBR Overestimates the Sending Rate

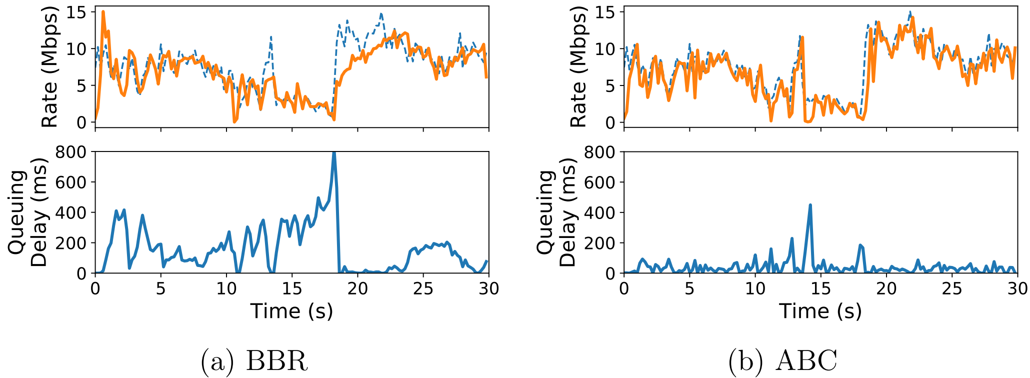
Fig. B-1 shows the throughput and queuing delay of BBR on a Verizon cellular trace. BBR periodically increases its rate in short pulses, and frequently overshoots the link capacity with variable-bandwidth links, causing excessive queuing.

## B.2 Wi-Fi Evaluation

In this experiment we use the setup from Fig. 2-9a. To emulate movement of the receiver, we model changes in MCS index as brownian motion, with values changing every 2 seconds. Fig. B-2 shows throughput and $95^{th}$ percentile per packet delay for a number of schemes. Again, ABC outperforms all other schemes achieving better throughput and latency trade off.

## B.3 Low Delays and High Throughput

Fig. B-3 shows the mean per packet delay achieved by various schemes in the experiment from Fig. 2-8. We observe the trend in mean delay is similar to that of $95^{th}$ percentile delay ( Fig. 2-8b). ABC achieves delays comparable to Cubic+Codel, Cubic+PIE and Copa. BBR, PCC Vivace-latency and Cubic incur 70-240% higher




(a) BBR

(b) ABC

Figure B-1: **Comparison with BBR** — BBR overshoots the link capacity, causing excessive queuing. Same setup as Fig. 2-1.

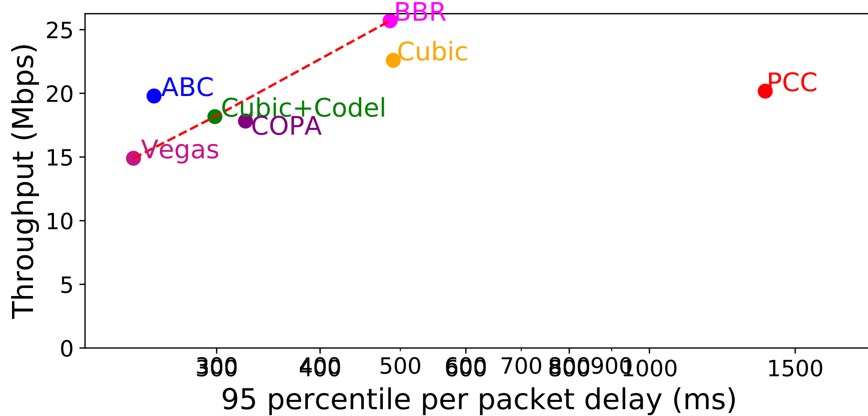



Figure B-2: **Throughput and $95^{th}$ percentile delay for a single user in WiFi** — We model changes in MCS index as bownian motion, with values changing every 2 seconds. We limit the MCS index values to be between 3 and 7. ABC outperforms all other schemes.

mean delay than ABC.

## B.4 ABC vs Explicit Control Schemes

In this section we compare ABC's performance with explicit congestion control schemes. We consider XCP, VCP, RCP and our modified implementation of XCP ($\text{XCP}_w$). For XCP and $\text{XCP}_w$, we used constant values of $\alpha = 0.55$ and $\beta = 0.4$, which the authors note are the highest permissible stable values that achieve the fastest possible link rate convergence. For RCP and VCP, we used the author-specified parameter values of $\alpha = 0.5$ and $\beta = 0.25$, and $\alpha = 1$, $\beta = 0.875$ and $\kappa = 0.25$, respectively. Fig. B-4

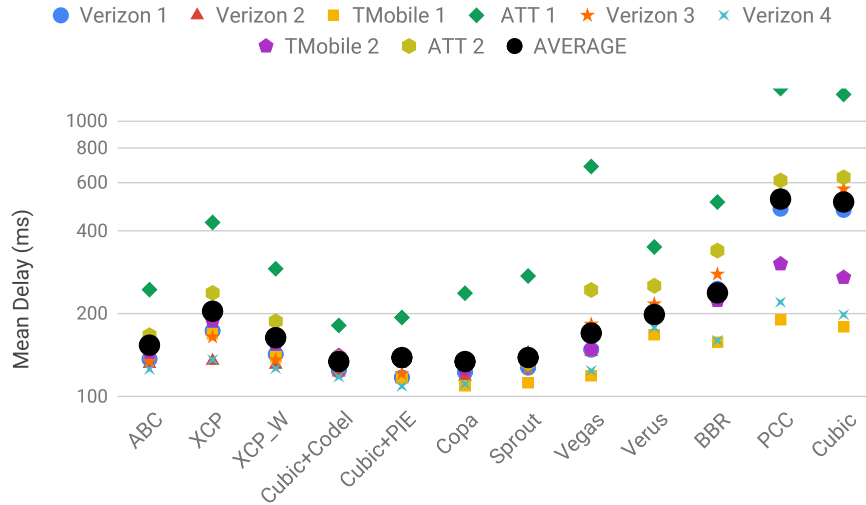
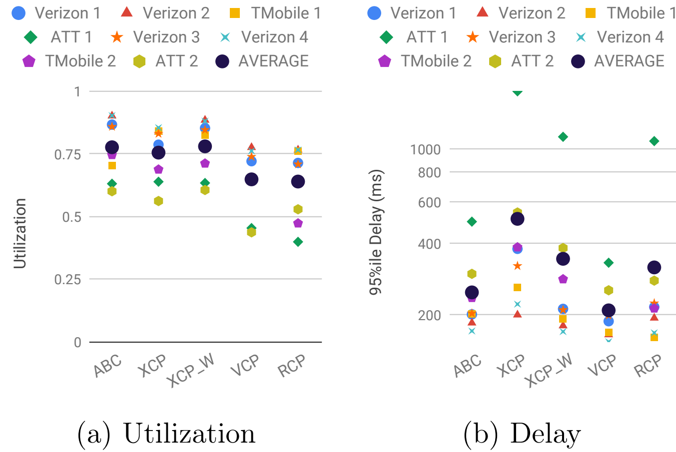



Figure B-3: **Utilization and mean per-packet delay across 8 different cellular network traces** — On average, ABC achieves similar delays and 50% higher utilization than Copa and Cubic+Codel. BBR, PCC, and Cubic achieve slightly higher throughput than ABC, but incur 70-240% higher mean per-packet delays.

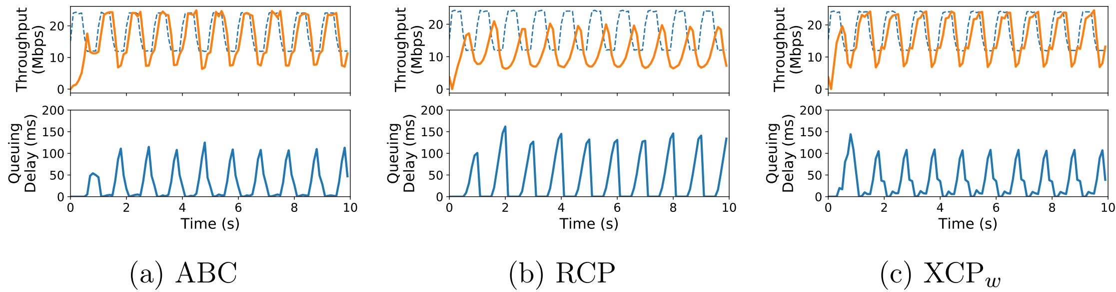
shows utilizations and mean per packet delays achieved by each of these schemes over eight different cellular link traces. As shown, ABC is able to achieve similar throughput as the best performing explicit flow control scheme, XCP$_w$, without using multibit per-packet feedback. We note that XCP$_w$'s 95$^{th}$ percentile per-packet delays are 40% higher than ABC's. ABC is also able to outperform RCP and VCP. Specifically, ABC achieves 20% higher utilization than RCP. This improvement stems from the fact that RCP is a rate based protocol (not a window based protocol)—by signaling rates, RCP is slower to react to link rate fluctuations (Figure B-5 illustrates this behavior). ABC also achieves 20% higher throughput than VCP, while incurring slightly higher delays. VCP also signals multiplicative-increase/multiplicative-decrease to the sender. But unlike ABC, the multiplicative increase/decrease constants are fixed. This coarse grained feedback limits VCP's performance on time varying links.

Fig. B-5 shows performance of ABC, RCP and XCP$_w$ on a simple time varying link. The capacity alternated between 12 Mbit/sec and 24 Mbit/sec every 500 milliseconds. ABC and XCP$_w$ adapt quickly and accurately to the variations in bottleneck rate, achieving close to 100% utilization. RCP is a rate base protocol and is inherently




(a) Utilization (b) Delay

Figure B-4: **ABC vs explicit flow control** — ABC achieves similar utilization and $95^{th}$ percentile per-packet delay as XCP and XCP$_w$ across all traces. Compared to RCP and VCP, ABC achieves 20% more utilization.




(a) ABC (b) RCP (c) XCP$_w$

Figure B-5: **Time series for explicit schemes** — We vary the link capacity every 500ms between two rates 12 Mbit/sec and 24 Mbit/sec.The dashed blue in the top graph represents bottleneck link capacity. ABC and XCP$_w$ adapt quickly and accurately to the variations in bottleneck rate, achieving close to 100% utilization. RCP is a rate base protocol and is inherently slower in reacting to congestion. When the link capacity drops, RCP takes time to drain queues and over reduces its rates, leading to under-utilization.

slower in reacting to congestion. When the link capacity drops, RCP takes time to drain queues and over reduces its rates, leading to under-utilization.

# Appendix C

# BFC: Impact of Pause Threshold

A consequence of the simplicity of BFC's backpressure mechanism is that a flow can temporarily run out of packets at a bottleneck switch while the flow still has packets to send. The pause threshold ($Th$) governs the frequency of such events. Using a simple model, we quantify the impact of $Th$.

Consider a long flow $f$ bottlenecked at a switch $S$. To isolate the impact of the delay in resuming, we assume that $f$ is not sharing a queue with other flows at $S$ or the upstream switch. Let $\mu_f$ be the dequeue rate of $f$ at $S$, i.e., when $f$ has packets in $S$, the packets are drained at a steady rate of $\mu_f$. Similarly, let $\mu_f \cdot x$ be the enqueue rate of $f$ at the switch, i.e., if $f$ is not paused at the upstream, $S$ receives packets from $f$ at a steady rate of $\mu_f \cdot x$. Here, $x$ denotes the ratio of enqueue to dequeue rate at $S$. Since $f$ is bottlenecked at $S$, $x > 1$.

We now derive the fraction of time in steady state that $f$ will not have packets in $S$. We show that this fraction depends only on $x$ and $Th$, and is thereby referred as $E_f(x, Th)$.

The queue occupancy for $f$ will be cyclic with three phases.

- Phase 1: $S$ is receiving packets from $f$ and the queue occupancy in increasing.
- Phase 2: $S$ is *not* receiving packets from $f$ and the queue is draining.
- Phase 3: $S$ is not receiving packets from $f$ while the queue is empty.

The time period for phase 1 ($t_{p1}$) can be calculated as follows. The queue occupancy at start of the phase is 0 and $S$ is receiving packets from $f$. $f$ gets paused when the queue occupancy exceeds $Th$. The queue builds at the rate $\mu_f \cdot x - \mu_f$ (enqueue rate - dequeue rate). The pause is triggered after $\frac{Th}{\mu_f \cdot (x-1)}$ time from the start of the phase. Since the pause takes an $HRTT$ to take effect, the queue grows for an additional $HRTT$. $t_{p1}$ is therefore given by:

$$t_{p1} = \frac{Th}{\mu_f \cdot (x-1)} + HRTT. \tag{C.1}$$

The queue occupancy at the end of phase 1 is $Th + HRTT \cdot \mu_f \cdot (x-1)$. The time period for phase 2 ($t_{p2}$) corresponds to the time to drain the queue. $t_{p2}$ is given by:

$$t_{p2} = \frac{Th + HRTT \cdot \mu_f \cdot (x-1)}{\mu_f}. \tag{C.2}$$

At the end of phase 2, there are no packets from $f$ in $S$. As a result, $S$ resumes $f$ at the upstream. Since the resume takes an $HRTT$ to take effect, the queue is empty for an $HRTT$. Time period for phase 3 ($t_{p3}$) is given by:

$$t_{p3} = HRTT \tag{C.3}$$

Combining the equations, $E_f(x, Th)$ is given by:

$$\begin{aligned} E_f(x, Th) &= \frac{t_{p3}}{t_{p1} + t_{p2} + t_{p3}} \\ &= \frac{x-1}{\frac{Th}{HRTT \cdot \mu_f} \cdot x + (x^2 - 1)}. \end{aligned} \tag{C.4}$$

Notice that for a given $x$, $E_f(x, Th)$ reduces as we increase $Th$. Increasing $Th$, increases the time period for phase 1 and phase 2, and the fraction of time $f$ runs out of packets reduces as a result.

We now quantify the impact of pause threshold on the worst case (maximum) value of $E_f(x, Th)$. Given a $Th$, $E_f(x, Th)$ varies with $x$. When $x \to 1$, $(E_f(x, Th) \to 0$, and when $x \to \infty$, $(E_f(x, Th) \to 0$. The maxima occurs somewhere in between. More

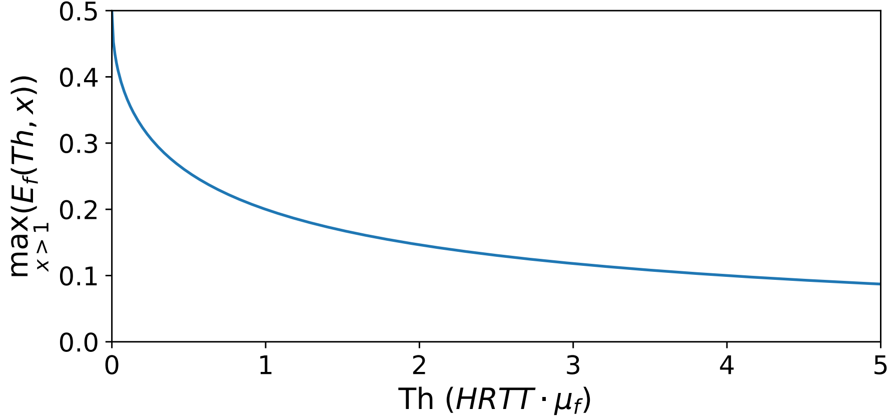



Figure C-1: Impact of pause threshold ($Th$) on the metric of worst case inefficiency. Increasing $Th$ reduces the maximum value for the fraction of time $f$ can run out of packets at the bottleneck.

concretely, for a given value of $Th$, the maxima occurs at $x = \sqrt{\frac{Th}{HRTT \cdot \mu_f}} + 1$. The maximum value ($\max_{x>1}(E_f(x, Th))$) is given by:

$$\max_{x>1}(E_f(x, Th)) = \frac{1}{\left(\sqrt{\frac{Th}{HRTT \cdot \mu_f}} + 1\right)^2 + 1}. \tag{C.5}$$

Fig. C-1 shows how $\max_{x>1}(E_f(x, Th))$ changes as we increase the pause threshold. As expected, increasing the pause threshold reduces $\max_{x>1}(E_f(x, Th))$. However, increasing the pause threshold has diminishing returns. Additionally, increasing $Th$ increases the buffering for $f$ (linearly).

In BFC, we set $Th$ to 1-Hop BDP at the queue drain rate, i.e., $Th = HRTT \cdot \mu_f$. Thereofore, the maximum value of $E_f(x, Th)$ is 0.2 (at $x = 2$). This implies, under our assumptions, that a flow runs out of packets at most 20% of the time due to the delay in resuming a flow.

Note that 20% is the maximum value for $E_f(x, Th)$. When $x \neq 2$, $E_f(x, Th)$ is lower. For example, when $x = 1.1$ (i.e., the enqueue rate is 10% higher than the dequeue rate), $E_f(x, Th)$ is only 7.6%.

The above analysis suggests that the worst-case under-utilization caused by delay in resuming is 20%. Note that in practice, when an egress port is congested, there are typically multiple flows concurrently active at that egress. In such scenarios, the under-utilization is much less than this worst-case bound, because it is unlikely that

all flows run out of packets at the same time. As our evaluation shows, with BFC, flows achieve close to ideal throughput in realistic traffic scenarios (§3.6).

# Appendix D

# BFC: Deadlock Prevention

We formally prove that BFC is deadlock-free in absence of cyclic buffer dependency. Inspired by Tagger [71], we define a backpressure graph ($G(V, E)$) as follows:

1. Node in the graph ($V$): A node is an egress port in a switch and can thus be represented by the pair <switchID, egressPort>.

2. Edge in the graph ($E$): There is a directed edge from $B \rightarrow A$, if a packet can go from $A$ to $B$ in a single hop (i.e., without traversing any other nodes) and trigger backpressure from $B \rightarrow A$. Edges represent how backpressure can propagate in the topology.

We define deadlock as a situation when a node (egress port) contains a queue that has been paused indefinitely. Cyclic buffer dependency is formally defined as the situation when $G$ contains a cycle.

**Theorem 2.** *BFC is deadlock-free if $G(V, E)$ does not contain any cycles.*

*Proof:* We prove the theorem by using contradiction.

Consider a node $A$ that is deadlocked. $A$ must contain a queue ($A_q$) that has been indefinitely paused as a result of backpressure from the downstream switch. If all the packets sent by $A_q$ were drained from the downstream switch, then $A_q$ will get unpaused (§3.3.3.2). There must be at least one node ($B$) in the downstream switch that triggered backpressure to $A_q$ but hasn't been able to drain packets from $A_q$, i.e.,

$B$ is deadlocked. This implies, in $G$, there must be an edge from $B \to A$. Applying induction, for $B$ there must exist another node $C$ (at the downstream switch of $B$) that is also deadlocked (again there must be an edge from $C \to B$). Therefore, there will be an infinite chain of nodes which are paused indefinitely, the nodes of the chain must form a path in $G$. Since $G$ doesn't have any cycles, the paths in $G$ can only be of finite length, and therefore, the chain cannot be infinitely long. A contradiction, hence proved.

**Preventing deadlocks:** To prevent deadlocks, given a topology, we calculate the backpressure graph, and pre-compute the edges that should be removed so that the backpressure graph doesn't contain any cycles. Removing these edges thus guarantees that there will be no deadlocks even under link failures or routing errors. To identify the set of edges that should be removed we can leverage existing work [71].

To remove a backpressure edge $B \to A$, we use the simple strategy of skipping the backpressure operation for packets coming from $A$ going to $B$ at the switch corresponding to $B$.[1] Note that, a switch can identify such packets *locally* using the ingress and egress port of the packet. This information can be stored as a match-action-table (indexed by the ingress and egress port) to check whether we should execute the backpressure operations for the packet.

For Clos topologies, this just includes backpressure edges corresponding to packets that are coming from a higher layer and going back to a higher layer (this can happen due to rerouting in case of link failures). Note that, usually the fraction of such packets is small ($< 0.002\%$ [71]), so forgoing backpressure for a small fraction of such packets should hurt performance marginally (if at all).

[1] To remove backpressure edges in PFC, Tagger uses a more complex approach that invloves creating new cycle free backpressure edges corresponding to the backpressure edges that should be removed. To ensure losslessness, Tagger generates backpressure using these new cycle free edges instead of the original backpressure edge. In our proposed solution, we forgo such requirement for simplicity.

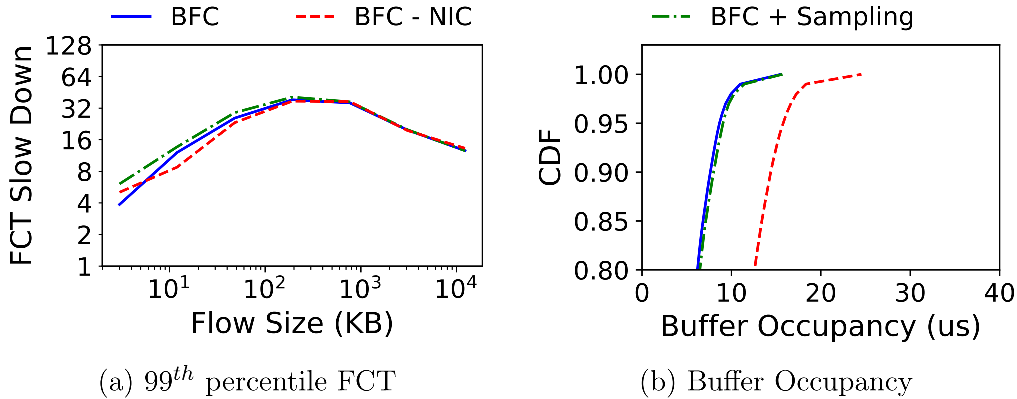
# Appendix E

# BFC: Incremental Deployment

We repeated the experiment in Fig. 3-11a in the scenario where i) BFC is deployed in part of the network; ii) The switch doesn't have enough capacity to handle all the recirculations. Fig. E-1 reports the tail FCT and buffer occupancy for these settings.

**Partial deployment in the network:** We first evaluate the situation when BFC is only deployed at the switches and the sender NICs don't respond to backpressure signal (shown as BFC - NIC). To prevent sender NIC traffic from filling up the buffers at the ToR, we assume a simple end-to-end congestion control strategy where the sender NIC caps the in-flight packets for a flow to 1 end-to-end bandwidth delay product (BDP). As expected, BFC - NIC experiences increased buffering at the ToR (Fig. E-1b). However, the tail buffer occupancy is still below the buffer size and there are no drops. Since all the switches are BFC enabled and following dynamic queue assignment, the frequency of collisions and hence the FCTs are similar to the orignal BFC.

**Sampling packets to reduce recirculations:** A BFC switch with an RMT architecture [30] recirculates packets to execute the dequeue operations at the ingress port. Depending on the packet size distribution of the workload, a switch might not have enough packet processing (pps) capacity or recirculation bandwidth to process these recirculated packets. In such scenarios, we can reduce recirculations by sampling packets. Sampling works as follows.




(a) $99^{th}$ percentile FCT (b) Buffer Occupancy

Figure E-1: FCT slowdown ($99^{th}$ percentile) and buffer occupancy distribution for two BFC variants. When NICs don't respond to backpressure (BFC - NIC), BFC experiences moderate increased buffering. Using sampling to reduce recirculation (BFC + sampling) has marginal impact on performance.

On a packet arrival (enqueue), sample to decide whether a packet should be recirculated or not. Only increment the pause counter and `size` in the flow table for packets that should be recirculated. The dequeue operations remain as is and are only executed on the recirculated packets. The `size` now counts the packets sampled for recirculation and residing in the switch. While sampling reduces recirculations, it can cause packet reordering. Recall, BFC uses `size` to decide when to reassign a queue. With sampling, `size` can be zero even when a flow has packets in the switch. This means a flow's queue assignment can change when it already has packets in the switch, causing reordering. However, sticky queue assignment should reduce the frequency of these events (§3.3.3.2).

We now evaluate the impact of sampling on the performance of BFC (shown as BFC + Sampling). In the experiment, the sampling frequency is set to 50%, i.e., only 50% of the packets are recirculated. BFC + Sampling achieves nearly identical tail latency FCT slowdowns and switch buffer occupancy as the orginal BFC. With sampling, fewer than 0.04% of the packets were retransmitted due to packet reordering.

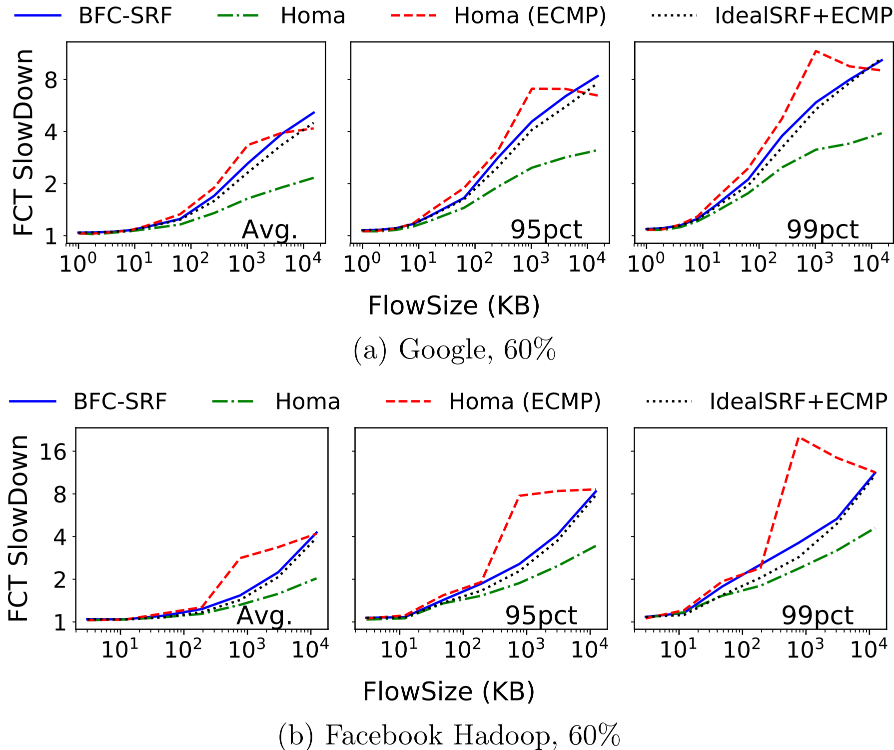
# Appendix F

# BFC: Miscellaneous Results

## F.1 Comparison with Homa

Homa is a receiver driven data center transport that uses network priorities to achieve an approximation of shortest-remaining-flow-first (SRF) scheduling. Homa divides a flow's data into unscheduled (first BDP of traffic) and scheduled categories. The sender assigns a fixed priority level to a flow's unscheduled bytes based on its size and the flow size distribution of the workload. The unscheduled bytes are transmitted at line rate. The receiver assigns priority levels to the scheduled bytes and issues grants (credits) for them. Homa assumes per-packet spraying to ensure load balancing across core links, and sufficient core capacity to guarantee minimal congestion in the core.

While we focus on fair queuing in this dissertation, BFC's design is applicable to other scheduling policies. In this section, we evaluate a variant of BFC, BFC-SRF, that aims to approximate SRF. Flows insert their remaining size into a header field in each packet transmitted, and the switch schedules queues in order of remaining size of the packet at the head of the queue. Similarly to Homa, NICs also follow SRF scheduling. We ran Homa using its OMNet++simulator [3]. The Homa simulator assumes unbounded buffers at the switch. For BFC, we use a 12 MB shared buffer. We use 32 queues for both Homa and BFC. For Homa, the 32 priority levels are divided between unscheduled and scheduled priorities based on the ratio of unscheduled and scheduled traffic; the overcommitment level is equal to the number of scheduled pri-




(a) Google, 60%

(b) Facebook Hadoop, 60%

Figure F-1: FCT slowdown on an oversubscribed clos topology. With packet spraying, Homa encounters minimal congestion in the core and outperforms other schemes.

orities [107]. We use our default topology with 128 servers and 2:1 oversubscription at the ToR uplinks (§3.6.2.1).

Two differences between Homa and BFC-SRF are worth highlighting. First, BFC-SRF uses flow-level ECMP rather than packet spraying for enforcing per-flow back-pressure. Second, BFC-SRF uses dynamic queue assignment and performs SRF scheduling directly on the switch, as opposed to Homa's priority assignment from the end-points. To understand the impact of these aspects separately, we also evaluate a variant of Homa with ECMP, and report results for IdealSRF+ECMP, an idealized SRF scheme with unlimited queues and unbounded buffers at each switch with ECMP load balancing.

We repeat the experiments in Fig. 3-10 and Fig. 3-11b for the Google and Facebook workloads at 60% load (log-normal flow arrivals without incast). Fig. F-1 reports the FCTs. Homa performs the best out of all schemes, achieving up to 2× better FCTs for long flows. With packet spraying, flows encounter minimal congestion in the core, and compete for bandwidth primarily at the last-hop. In contrast, ECMP is prone

| Scheme | Link | 95% Delay ($\mu$s) | 99% Delay ($\mu$s) |
|---|---|---|---|
| Homa | Agg-ToR | 2.4 | 6.7 |
| Homa | ToR-Agg | 2.1 | 6.0 |
| Homa ECMP | Agg-ToR | 40.8 | 87.2 |
| Homa ECMP | ToR-Agg | 43.7 | 93.3 |

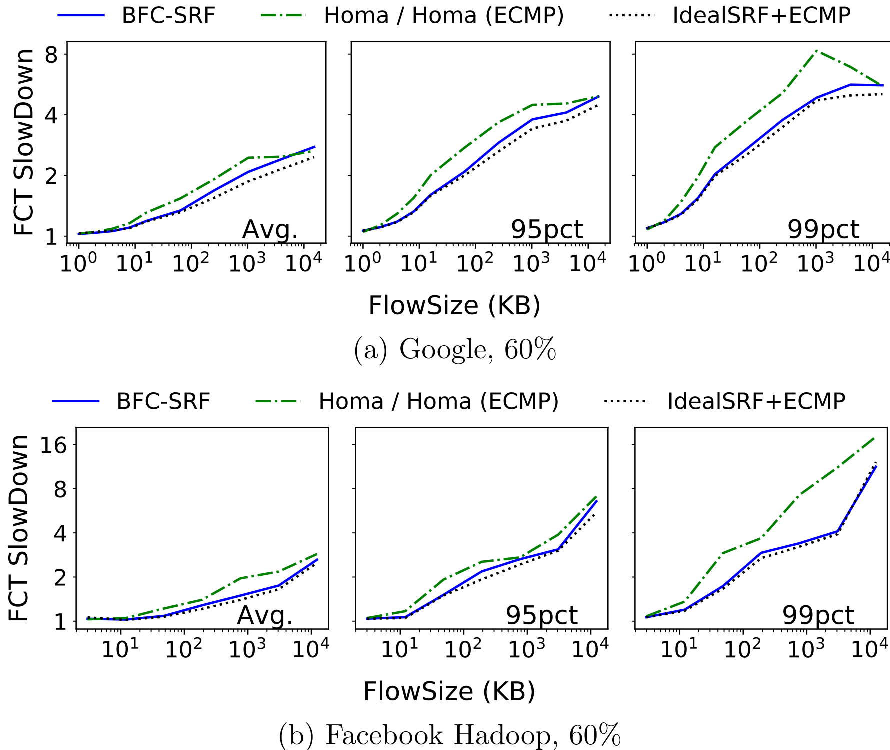
Table F.1: Per-packet queuing delay for scheduled traffic in the core.

to path collisions [13] and flows encounter congestion in the core. Notice that a last-hop link carries half the load of a core link (30% vs 60%) in this experiment on average (§3.6.2.1). Since packet spraying essentially eliminates congestion on the core links, with Homa flows experience congestion only on the last-hop links. But with the ECMP-based schemes, flows contend at the core links (with 2× the load). As a result, Homa even outperforms IdealSRF+ECMP. This result illustrates the benefits of packet spraying; nevertheless, packet spraying is rarely deployed in practice because it can cause packet reordering, increasing CPU overhead at endpoints[1], and it can hurt performance in asymmetric topologies (e.g., caused by rolling upgrades or link failures) [136].

Among the ECMP approaches, BFC-SRF is close to IdealSRF+ECMP and Homa is worse. In Homa, receivers have no visibility into congestion in the core and don't react to queue buildup in the core (though each flow limits its total in-flight data to 1 BDP). Also, Homa's receiver-set priorities are only based on contending flows at the last hop, and can violate SRF scheduling when congestion occurs in the core. Table F.1 shows that with ECMP, the scheduled traffic encounters significantly higher queuing in the core.

**Benefits of BFC's dynamic queue assignment over Homa.** BFC makes queue assignment and scheduling decisions at the switch, based on an instantaneous view of competing flows. In principle, this should allow BFC to more accurately approximate SRF compared to Homa. To understand if this is actually the case, we conduct an experiment with the same Google and Facebook workloads but with all flows destined to a single receiver, and the senders located within the same rack as the receiver. Since there is no traffic in the core, load balancing (ECMP vs. packet spraying) does not

[1]Packet reordering makes hardware offloads such as Large Receiver Offload (LRO) ineffective [52].




(a) Google, 60%

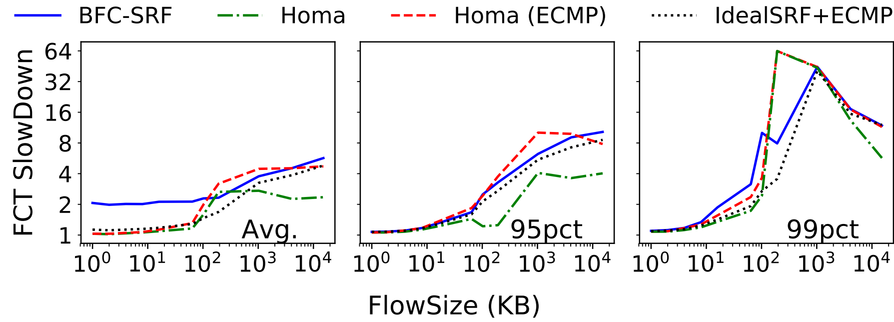
(b) Facebook Hadoop, 60%

Figure F-2: BFC's dynamic queueue assignment achieves a better approximation of the SRF scheduling policy. BFC-SRF achieves close to optimal FCTs.

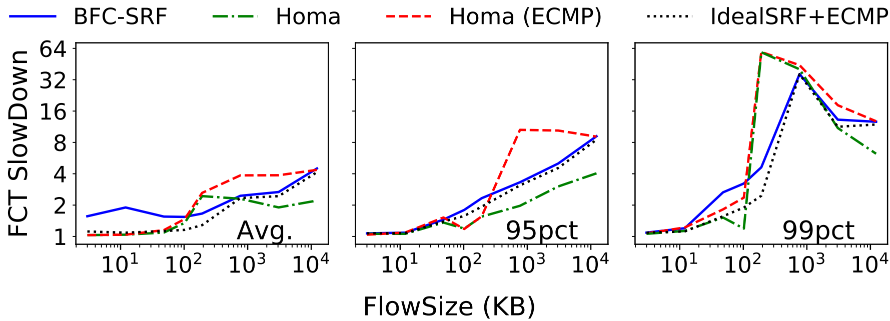
matter in this case. Flow arrivals are log-normal and the load on the receiver's link is 60%. Fig. F-2 shows the results. BFC-SRF achieves better FCTs primarily at the tail.

We give two examples of priority inversions in Homa which BFC avoids. First, the Homa sender assigns priorities to unscheduled traffic based on flow size distributions rather than using the current set of flows competing at the switch due to lack of visibility for the first RTT. As a result with Homa, short flows ($<1$ BDP) with similar flow sizes can end up sharing unscheduled priority queues unnecessarily, even when there are sufficient queues at the switch to assign each flow a unique queue. Second, in Homa the unscheduled bytes of a flow are always scheduled ahead of the scheduled bytes of competing flows. This implies that the unscheduled bytes of a new long flow will be *incorrectly* scheduled ahead of the scheduled bytes of a shorter flow. This also violates SRF and increases FCT for flows larger than a BDP.

**Impact of collisions on BFC-SRF.** Recall that with large incast, BFC can experience collisions. For BFC-SRF, such collisions can cause priority inversions that




(a) Google, 55% + 5% 100-1 incast




(b) Facebook Hadoop, 55% + 5% 100-1 incast

Figure F-3: FCT slowdown with 100-1 incast. Collisions in BFC-SRF can cause priority inversions hurting FCTs

hurt FCTs. To illustrate this, we repeat the experiments in Fig. 3-9 and Fig. 3-11a (55% load plus 5% 100-1 incast traffic). Fig. F-3 shows that the average FCT for short flows is higher with BFC-SRF. This is because of high completion times for a (small) fraction of short flows sharing queues with longer flows. To understand why, consider the following situation. An incoming short flow arrives when there are no free queues, and ends up sharing the queue with a long flow. Let's say the remaining size of the long flow is greater than the incast flow size (200 KB in this experiment). In case there are competing incast flows present in other queues, the incast flows will be scheduled ahead of this long flow. Therefore, the short flow will have to wait for *all* the traffic from the incast flows to finish to make any progress. This can severely degrade its completion time. The core of this problem is that when a port runs out of queues, the BFC switch assigns the new flow to a queue randomly. This is fine for fair queuing but with SRF, a more sophisticated strategy may improve performance (e.g., assign the new flow to a queue with similar remaining flow sizes).

As explained earlier, Homa is not immune to priority inversions. Fig. F-3 shows

that with Homa, flows with size greater than 1 BDP but less than 2 BDP have high FCTs at the tail. This is because unscheduled bytes of the the incast flows are incorrectly scheduled ahead of the scheduled bytes of such flows.

These experiments suggest an interesting possibility to try to get the best of both schemes: we could combine BFC's dynamic queue assignment for unscheduled traffic with Homa's grant mechanism for controlling scheduled traffic. We leave exploration of such a design to future work.

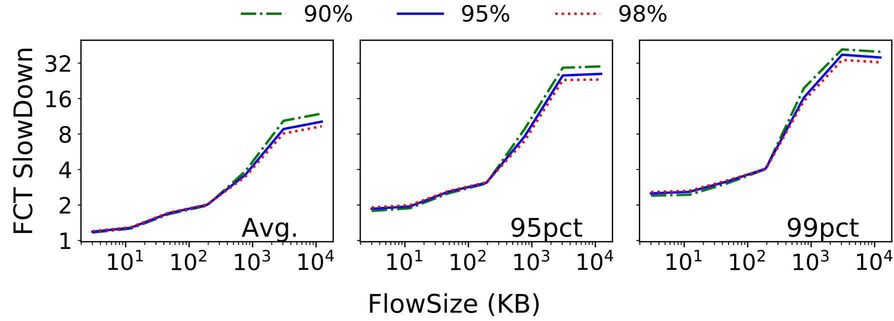
## F.2 Parameter Sensitivity for Comparison Schemes

In this section, we perform sensitivity analysis to understand the impact of parameters on performance of HPCC, DCTCP and ExpressPass. We repeat the experiment in Fig. 3-11b (Facebook distribution with 60% load). Fig. F-4 reports the average, $95^{th}$ and $99^{th}$ percentile flow completion times as we vary the parameters. In general, we observe that parameters present a trade-off between the latency of short flows (queuing) and the throughput of long flows (link utilization).

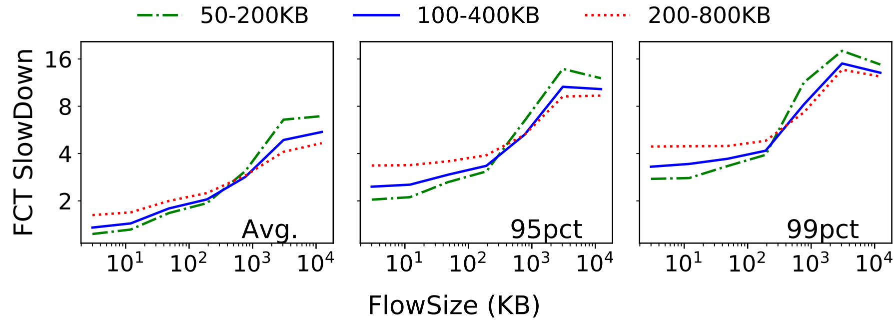
*HPCC:* We vary the target utilization ($\eta$) from 90 to 98%. As expected, increasing $\eta$ worsens the FCT of short flows but improves the FCT for long flows (marginally for both), see Fig. F-4a.

*DCTCP:* We vary the ECN marking threshold governed by parameters $K_{min}$ and $K_{max}$. Increasing the threshold increases the queuing at the switch, which increases FCT of short flows but improves link utilization (Fig. F-4b).

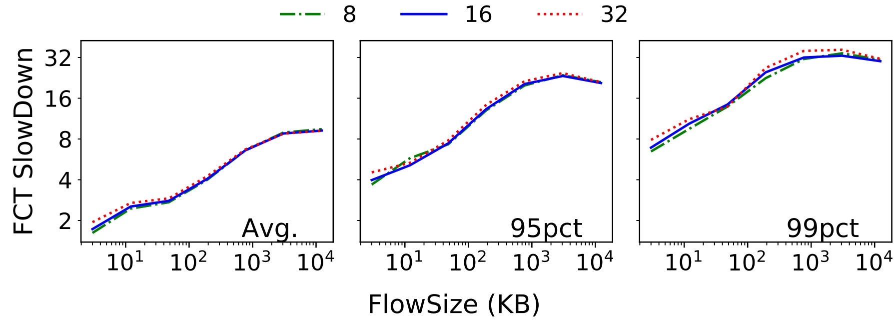
*ExpressPass:* Varying the credit buffer size has little impact on performance (Fig. F-4c). We vary $\alpha$, which controls how the receiver credits are generated. Reducing $\alpha$ reduces "credit waste", improving the FCT of long flows. However, it also increases the FCT of short flows (Fig. F-4d).



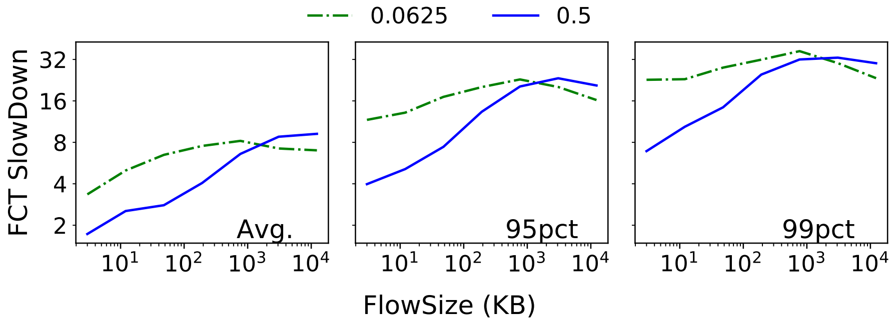

(a) HPCC ($\eta$)




(b) DCTCP (ECN marking threshold: $K_{min}$-$K_{max}$)

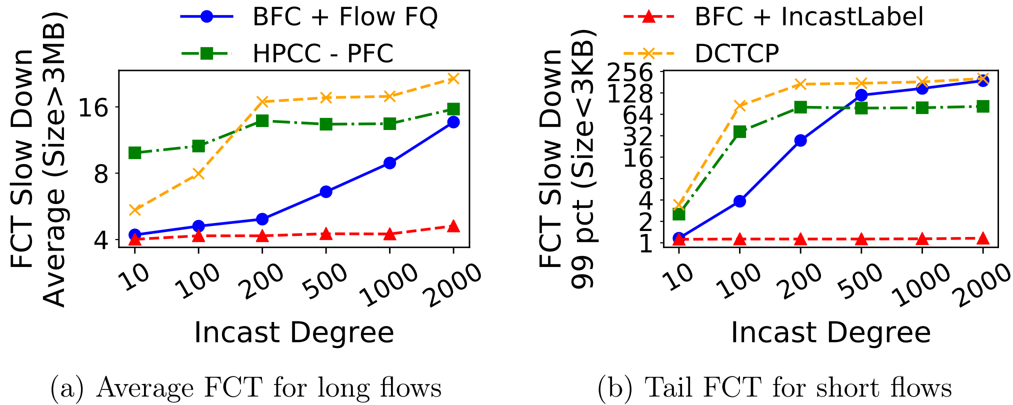



(c) ExpressPass (Credit Buffer Size)




(d) ExpressPass ($\alpha$)

Figure F-4: **Parameter sensitivity for comparison schemes** — $99^{th}$ percentile FCT slowdown for the Facebook workload, 60% load without incast. Sensitivity to the choice of parameters in HPCC, DCTCP, and ExpressPass.




(a) Average FCT for long flows (b) Tail FCT for short flows

Figure F-5: FCT slowdown for short and long flows as a function of incast degree. The x axis is not to scale. By isolating incast flows, BFC + IncastLabel reduces collisions and achieves the best performance.

## F.3 Reducing Contention for Queues

To reduce contention for queues under incast, we tried a variant of BFC where the sender labels incast flows explicitly (similar to the potential optimization in [107]). BFC + IncastLabel assigns all the incast flows at an egress port to the same queue. This frees up queues for non-incast traffic, reducing collisions and allowing the scheduler to share the link between incast and non-incast traffic more fairly.

Fig. F-5 shows the performance of BFC + IncastLabel in the same setup as Fig. 3-13. The original BFC is shown as BFC + Flow FQ for per-flow fair queuing. BFC + IncastLabel achieves the best performance across all the scenarios. However, the FCTs for incast flows is higher compared to BFC + Flow FQ (numbers not shown here). When there are multiple incast flows at an ingress port, the incast flows are allocated less bandwidth in aggregate compared to per-flow fair queuing.

While BFC + IncastLabel achieves great performance, it assumes the application is able to label incast flows, and so we use a more conservative design for the main body of our evaluation.

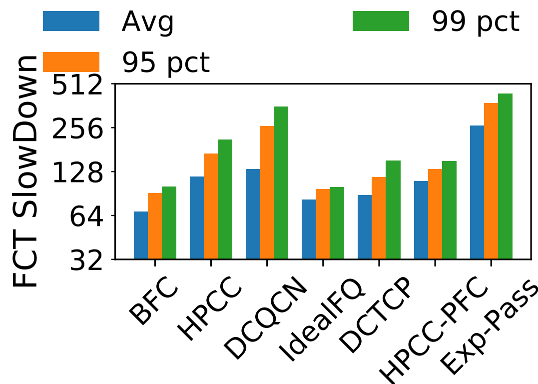



Figure F-6: FCT slowdown for incast traffic. Slowdown is defined per flow. BFC reduces the FCT for incast flows compared to other feasible schemes. Setup from Fig. 3-9.

## F.4 Incast Flow Performance

Fig. F-6 shows the slowdown for incast flows for the Google workload used in Fig. 3-9. The benefits of BFC for non-incast traffic do not come at the expense of worse incast performance. Indeed, BFC improves the performance of incast flows relative to end-to-end congestion control, because it reacts faster when capacity becomes available at the bottleneck, reducing the percentage of time the bottleneck is unused while the incast is active.

# Appendix G

# Nimbus: Miscellaneous Results

## G.1 Nimbus Helps Cross Traffic

In the setup from §4.7.1, we measure the flow completion time (FCT) of cross traffic flows. Fig. G-1 compares the 95th percentile (p95) FCT for flows of different sizes. The FCTs are normalized by the corresponding value for NimbusCC at each flow size (i.e., NimbusCC is always 1).

BBR and PCC-Vivace exhibits much higher FCT at all cross traffic flow sizes compared to the other protocols, consistent with the unfairness seen in the experiment in §4.5.

For small flows ($\leq$15 KB), the p95 FCT with NimbusCC and Copa are comparable to Vegas and lower than Cubic. With NimbusCC, p95 FCT of cross traffic at higher flow sizes are slightly lower than Cubic because of small delays in switching to TCP-competitive mode. At all flow sizes, Vegas provides the best cross traffic flow FCTs, but its own flow rate is dismal; Copa is more aggressive than Vegas but less than NimbusCC, but at the expense of its own throughput (§4.7.1).

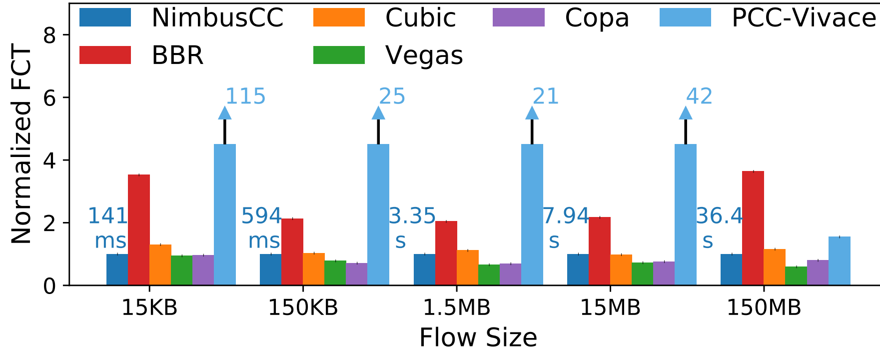



Figure G-1: Using NimbusCC reduces the p95 FCT of cross-flows relative to BBR at all flow sizes, and relative to Cubic for short flows. Vegas provides low cross-flow FCT, but its own rate is low.

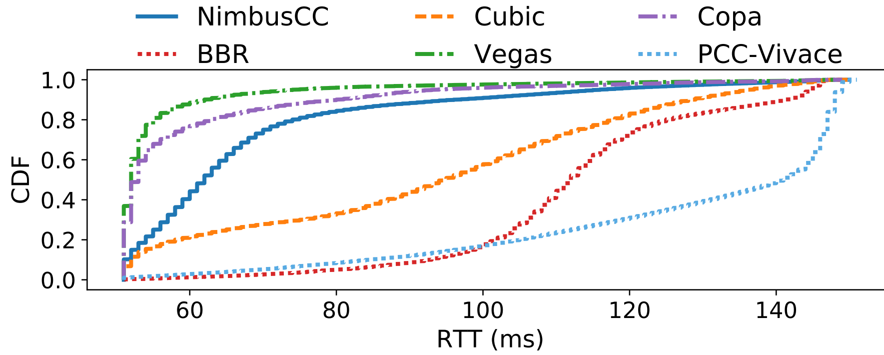
## G.2 Cross-traffic Congestion Control Protocols

### G.2.1 Multiple Elastic Flows using Different Congestion Control Protocols.

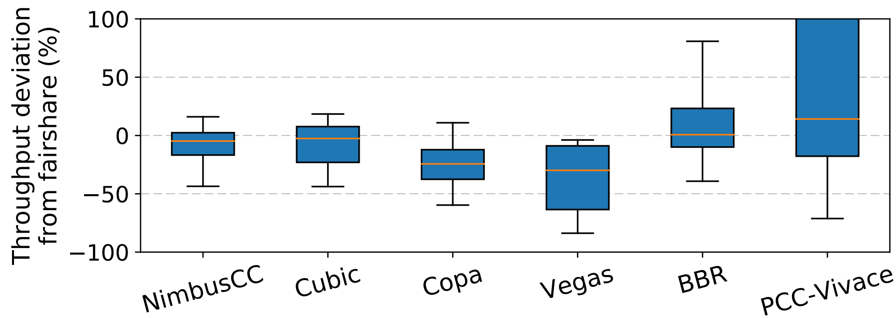
We repeat the experiment in Fig. 4-8 but with cross traffic consisting of an equal (on average) mix of Cubic, NewReno and BBR flows. Whenever a new cross traffic flow starts, with an equal probability it chooses one of the three congestion control protocols. Fig. G-2 shows performance of various schemes. The results are similar to the experiment in Fig. 4-8: NimbusCC achieves lower delays than Cubic for a similar throughput profile, while Copa and Vegas lose throughput when cross traffic is elastic. The reason is that, regardless of the congestion control protocol, the elastic cross traffic flows react to Nimbus's pulses, and can therefore be classified correctly.

### G.2.2 NimbusCC & Cubic v. BBR

We now evaluate how well a NimbusCC (Cubic + BasicDelay) flow competes with a BBR flow. In this experiment, the cross traffic is 1 BBR flow and the bottleneck link bandwidth is 96 Mbit/s. We vary the buffer size from 0.5 BDP to 4 BDP. Fig. G-3 shows the mean throughput of NimbusCC and Cubic flows while competing with BBR over a 2-minute experiment. NimbusCC achieves same throughput as Cubic for all buffer sizes.




(a) Per-packet RTT




(b) Deviation from fairshare throughput

Figure G-2: Performance with WAN cross traffic consisting of an equal mix of Cubic, NewReno and BBR flows. The deviation profile of NimbusCC is similar to that of Cubic, however, NimbusCC reduces delays.

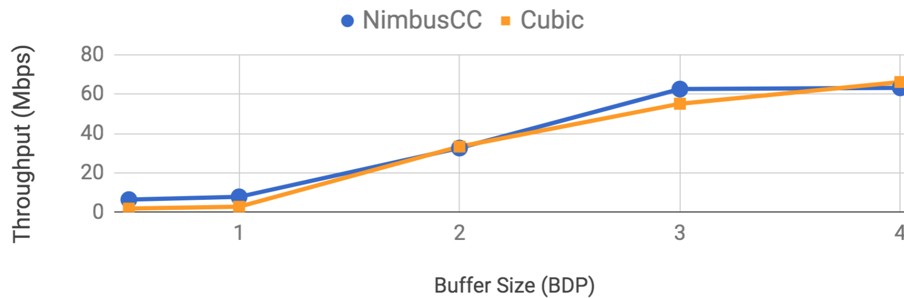



Figure G-3: **NimbusCC's performance against BBR is similar to that of Cubic—** Both NimbusCC and Cubic compete against 1 BBR flow on a 96 Mbit/s link. For various buffer sizes, NimbusCC achieves the same throughput as Cubic.

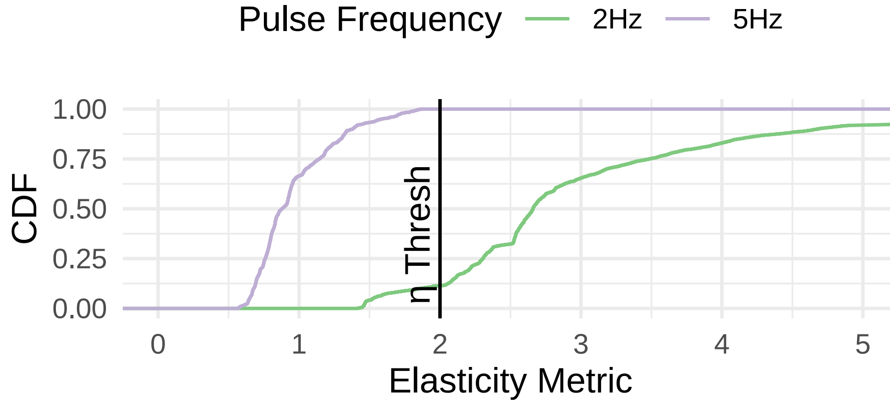



Figure G-4: By modifying the pulse frequency, Nimbus correctly classifies PCC-Vivace, a rate-based elastic protocol, as elastic.

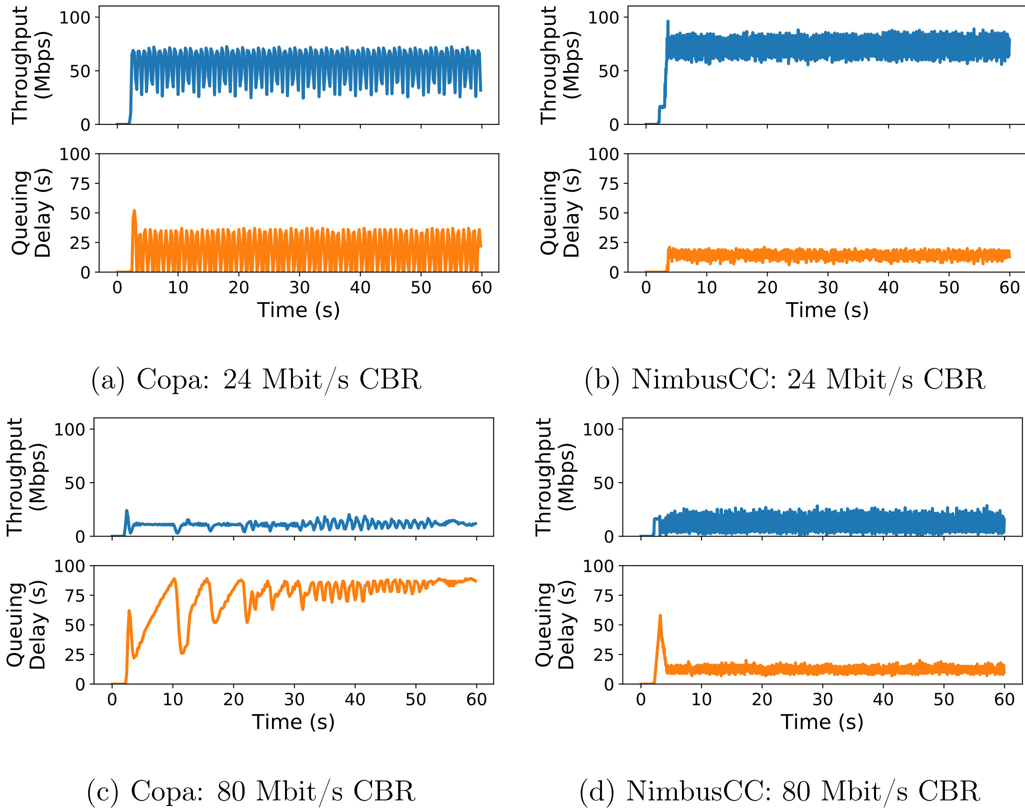
### G.2.3 Elastic Flows, No ACK Clocking

Nimbus aims to detect ACK-clocked elastic flows that react quickly to changes in available bandwidth on RTT timescales. This experiment demonstrates Nimbus's ability to also detect slow-reacting elastic cross traffic by tuning the pulse frequency. We ran a NimbusCC flow against a PCC-Vivace flow on a 96 Mbit/s link with 100ms of buffering. Fig. G-4 shows the CDF of the elasticity metric, $\eta$, for two different pulse frequencies, $f_p$. PCC-Vivace is not ACK-clocked and does not react to Nimbus's pulses at $f_p = 5$ Hz. As a result $\eta$ is below the threshold most of the time. Reducing the pulse frequency to 2 Hz creates pulses with a longer duration. PCC-Vivace reacts to these slower variations in available bandwidth, and is correctly classified as elastic ($\eta > \eta_{thresh}$).

Changing the pulse frequency involves a trade-off. Increasing the pulse duration will increase queuing delays and congestion. But if slowly-reacting elastic protocols become widely deployed, competing with them using Nimbus for delay-control opportunities will require an increase in pulse duration.

## G.3 Copa Mode Switching Errors

We explore the dynamics of NimbusCC and Copa's mode switching in experiments from the scenarios in §4.7.2.




(a) Copa: 24 Mbit/s CBR

(b) NimbusCC: 24 Mbit/s CBR

(c) Copa: 80 Mbit/s CBR

(d) NimbusCC: 80 Mbit/s CBR

Figure G-5: When the CBR traffic is low (a), Copa classifies the traffic as non buffer-filling and is able to achieve low queuing delays. But when the CBR traffic occupies a high fraction (c), Copa incorrectly classifies the traffic as buffer-filling, resulting in higher queuing delays. In both the situations (b and d), the elasticity detector correctly classifies the traffic as inelastic and NimbusCC achieves low queuing delays.

### G.3.1 CBR Cross Traffic

Fig. G-5 shows throughput and delay profile for Copa and NimbusCC while competing against inelastic CBR traffic. We consider two scenarios: (i) CBR occupies a small fraction of the link (24 Mbits/s, 25%) and (ii) CBR occupies majority of the link (80 Mbit/s, 83%). When the CBR traffic is low (Fig. G-5a and Fig. G-5b), both Copa and Nimbus identify it as non-buffer-filling and inelastic, respectively, and achieve low queuing delays.

When the CBR's share of the link is high (Fig. G-5c), Copa incorrectly classifies the cross traffic as buffer-filling and stays in competitive mode, leading to high queuing delays. Copa relies on a pattern of emptying queues to detect whether the cross traffic is buffer-filling or not. However, when the rate of cross traffic is $z$, the fastest possible

rate at which the queue can drain is $\mu - z$, even if Copa reduces its rate to zero. If the cross traffic occupies $x$ fraction of the link (i.e., $z = x\mu$), then

$$\max(-\frac{dQ}{dt}) = \mu - z = (1-x)\mu = (1-x)\frac{BDP}{RTT}. \tag{G.1}$$

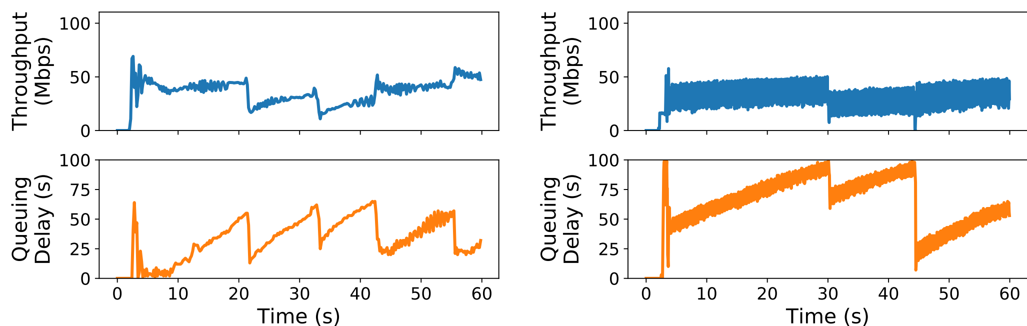
Hence, if the queue size exceeds $5\times(1-x)BDP$, Copa won't be able to drain the queue in 5 RTTs, and it will mis-classify the cross traffic as buffer-filling. The queue size can grow large due to a transient burst or if Copa incorrectly switches to competitive mode. Once Copa is in competitive mode, it will drive the queues higher, and may get stuck in that mode.

Nimbus doesn't rely on emptying queues and correctly classifies cross traffic as inelastic, achieving low delays (Fig. G-5d).

### G.3.2 Elastic Cross Traffic

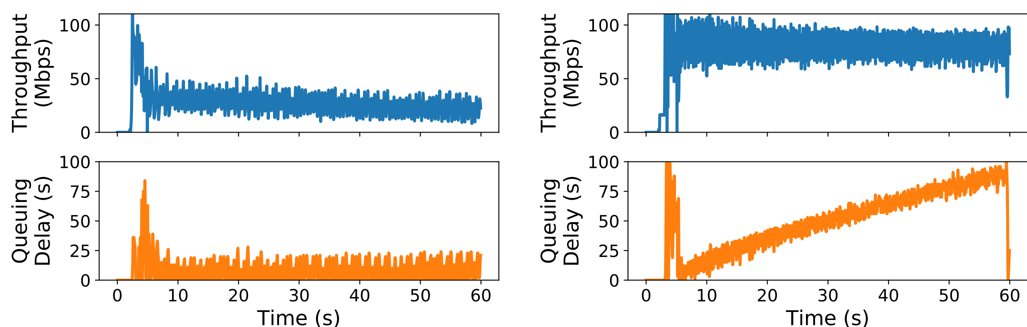
Fig. G-6 shows throughput and delay over time for Copa and NimbusCC while competing against an elastic NewReno flow. We consider two scenarios: (1) both flows have the same propagation RTT, and (2) the cross traffic's propagation RTT is 4× higher than the Copa or NimbusCC flow. When the RTTs are the same (Fig. G-6a and Fig. G-6b), both Copa and Nimbus correctly classify the cross traffic, achieving their fair share.

When the cross traffic RTT is higher (Fig. G-6c), NewReno ramps up its rate slowly, causing Copa to mis-classify the traffic and achieve less than its fair share. Here, Copa achieves 27 Mbit/s but its fair share is at least 48 Mbit/s (in fact, 77 Mbit/s considering the RTT bias). In contrast, (Fig. G-6d), Nimbus correctly classifies the cross traffic as elastic, and NimbusCC achieves its RTT-biased share of throughput.




(a) Copa: Cross Traffic RTT = 1 × Flow RTT

(b) NimbusCC: Cross Traffic RTT = 1 × Flow RTT




(c) Copa: Cross Traffic RTT = 4 × Flow RTT

(d) NimbusCC: Cross Traffic RTT = 4 × Flow RTT

Figure G-6: **Queuing delay and throughput dynamics for elastic cross traffic** — When the elastic cross traffic increases fast enough (a), Copa classifies it as buffer-filling and is able to achieve its fair share. But when the elastic cross traffic increases slowly (c), Copa incorrectly classifies the traffic as non-buffer-filling, achieving less than its fair share. In both the situations (b and d), Nimbus correctly classifies the traffic as elastic and NimbusCC achieve its fair share.

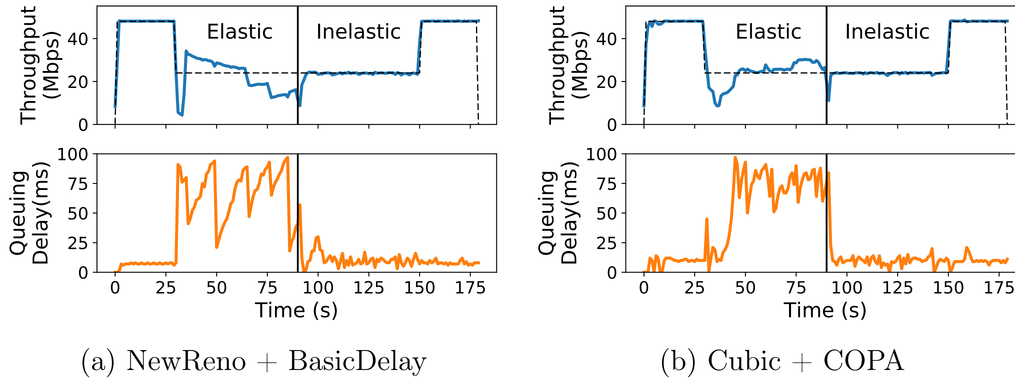
## G.4 Buffer size, RTT, and AQM

We vary the bottleneck drop-tail buffer size from 0.25 BDP to 4 BDP for three categories of cross traffic as in the earlier experiments, with propagation delays of 25 ms, 50 ms, and 75 ms. We also measured classification accuracy when the bottleneck link implements PIE [112] at two target delays (0.25 BDP and 1 BDP) with a propagation delay of 50 ms. With purely elastic or inelastic traffic, Nimbus has a mean accuracy (across five runs) of 98% or more in all cases but two, while with mixed traffic, the accuracy is always 85% or more. In all cases (including low accuracy ones), NimbusCC achieves its fair-share throughput and low delays.

Now we discuss the cases with low classification accuracy. First, with shallow buffers of size less than the product of the delay threshold $x_t$ and the bottleneck link rate (e.g., 0.25 BDP when the round-trip time is 50 ms), Nimbus classifies all traffic as elastic. Second, with the bottleneck link implementing PIE with small target delay (e.g., corresponding to 0.25 BDP), Nimbus classifies all traffic as elastic. In both cases, NimbusCC can incur heavy losses in delay-control mode as NimbusCC's target queuing delay of 0.25 BDP is comparable to the drop-tail buffer size or target delay of PIE. These losses interfere with the cross traffic estimator leading to classification errors (in delay-control mode). However, low accuracy does not impact the performance of NimbusCC as it achieves its fair-share throughput and low delays (bounded by the small buffer size for a drop-tail queue and the delay control threshold of PIE). Further, classification accuracy decreases when Nimbus's RTT exceeds its pulse period. Since Nimbus's measurements of rates are over one RTT, any oscillations over a smaller period cannot be observed.

## G.5 Using Different CC Algorithms with NimbusCC

NimbusCC can employ a variety of congestion control algorithms for its delay-controlling and TCP-competitive modes. We have implemented Cubic, NewReno, and MulTCP [40] as competitive-mode algorithms, and BasicDelay, Vegas, FAST [140], and COPA [22]




(a) NewReno + BasicDelay (b) Cubic + COPA

Figure G-7: **NimbusCC's versatility**— NimbusCC with different combinations of delay-controlling and TCP-competitive algorithms.

as delay-controlling algorithms. In Fig. G-7, we illustrate two combinations of delay and competitive mode algorithms sharing a bottleneck link with synthetic elastic and inelastic cross traffic active at different periods during the experiment. The fair-share rate over time is shown as a reference. Both NewReno+BasicDelay (Fig. G-7a) and Cubic+COPA (Fig. G-7b) achieve their fair-share rate while keeping the delays low in the absence of elastic cross traffic.

# Bibliography


[1] 3gpp technical specification for lte. https://www.etsi.org/deliver/etsi_ts/132400_132499/132450/09.01.00_60/ts_132450v090100p.pdf.

[2] Express pass simulation. https://github.com/kaist-ina/ns2-xpass.

[3] Homa simulation. https://github.com/PlatformLab/HomaSimulation/tree/omnet_simulations/RpcTransportDesign.

[4] Hpcc simulation. https://github.com/alibaba-edu/High-Precision-Congestion-Control.

[5] Network simulator 3. https://www.nsnam.org.

[6] Ns-3 simulator for rdma. https://github.com/bobzhuyb/ns3-rdma.

[7] sfqCoDel. http://www.pollere.net/Txtdocs/sfqcodel.cc.

[8] Slither.io interactive multiplayer game. http://slither.io.

[9] 802.11Qau. Congestion notification, 2010.

[10] Filipe Abrantes and Manuel Ricardo. XCP for shared-access multi-rate media. *Computer Communication Review*, 36(3):27–38, 2006.

[11] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 113–119. ACM, 2019.

[12] Aditya Akella, Srinivasan Seshan, Scott Shenker, and Ion Stoica. Exploring congestion control. Technical report, CMU School of Computer Science, 2002.

[13] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 503–514. ACM, 2014.

[14] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In Shivkumar Kalyanaraman, Venkata N. Padmanabhan, K. K. Ramakrishnan, Rajeev Shorey, and Geoffrey M. Voelker, editors, *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010*, pages 63–74. ACM, 2010.

[15] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 253–266. USENIX Association, 2012.

[16] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. In Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan, editors, *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, pages 435–446. ACM, 2013.

[17] Mark Allman. Tcp congestion control with appropriate byte counting (abc)", rfc 3465. 2003.

[18] Amazon. Amazon Web Services. `https://aws.amazon.com/s3/`.

[19] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High speed switch scheduling for local area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, 1993.

[20] Lachlan L. H. Andrew, Stephen V. Hanly, Sammy Chan, and Tony Cui. Adaptive deterministic packet marking. *IEEE Communications Letters*, 10(11):790–792, 2006.

[21] Arista. Arista 7170 Multi-function Programmable Networking. `https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf`.

[22] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 329–342. USENIX Association, 2018.

[23] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 455–468, 2015.

[24] Deepak Bansal, Hari Balakrishnan, Sally Floyd, and Scott Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. In Rene L. Cruz and George Varghese, editors, *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, pages 263–274. ACM, 2001.

[25] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostic, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 667–683. USENIX Association, 2020.

[26] Barefoot. Tofino: World's Fastest P4-Compatible Ethernet Switch ASICs. `https://www.barefootnetworks.com/products/brief-tofino/`.

[27] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *Comput. Commun. Rev.*, 40(1):92–99, 2010.

[28] John Charles Bicket. *Bit-rate selection in wireless networks*. PhD thesis, Massachusetts Institute of Technology, 2005.

[29] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.

[30] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[31] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In Jon Crowcroft, editor, *Proceedings of the ACM SIGCOMM 1994 Conference on Communications Architectures, Protocols and Applications, London, UK, August 31 - September 2, 1994*, pages 24–35. ACM, 1994.

[32] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, pages 858–867. IEEE Computer Society, 1994.

[33] Broadcom. StrataXGS. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs`.

[34] CAIDA. The CAIDA Anonymized Internet Traces 2016 Dataset - 2016-01-21. `http://www.caida.org/data/passive/passive_2016_dataset.xml`, 2016.

[35] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Site-to-site internet traffic control. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 574–589. ACM, 2021.

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

[37] D-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[38] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 239–252. ACM, 2017.

[39] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 1–14. ACM, 2017.

[40] Jon Crowcroft and Philippe Oechslin. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM CCR*, 28(3):53–69, July 1998.

[41] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[42] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, Renton, WA, 2018. USENIX Association.

[43] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. What do packet dispersion techniques measure? In *INFOCOM*. IEEE, 2001.

[44] Allen B Downey. Using pathchar to estimate internet link characteristics. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 241–250. ACM, 1999.

[45] David Ely, Neil Spring, David Wetherall, Stefan Savage, and Tom Anderson. Robust congestion signaling. In *Network Protocols, 2001. Ninth International Conference on*, pages 332–341. IEEE, 2001.

[46] Florian Fainelli. The openwrt embedded development framework. In *Proceedings of the Free and Open Source Software Developers European Meeting*, 2008.

[47] Willliam Feller. *An introduction to probability theory and its applications*, volume 2. John Wiley & Sons, 2008.

[48] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 468–482. ACM, 2016.

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

[50] Rodrigo Fonseca, George Porter, R Katz, Scott Shenker, and Ion Stoica. Ip options are not an option. *University of California at Berkeley, Technical Report UCB/EECS-2005-24*, 2005.

[51] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: distributed near-optimal datacenter transport over commodity network fabric. In Felipe Huici and Giuseppe Bianchi, editors, *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT 2015, Heidelberg, Germany, December 1-4, 2015*, pages 1:1–1:12. ACM, 2015.

[52] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. Juggler: a practical reordering resilient network stack for datacenters. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 20:1–20:16. ACM, 2016.

[53] Daniel Genin and Jolene Splett. Where in the internet is congestion? *arXiv preprint arXiv:1307.3696*, 2013.

[54] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, 9(11):40, 2011.

[55] Boris Ginzburg and Alex Kesselman. Performance analysis of a-mpdu and a-msdu aggregation in ieee 802.11 n. In *Sarnoff symposium, 2007 IEEE*, pages 1–5. IEEE, 2007.

[56] Google. Google Cloud Platform. https://cloud.google.com.

[57] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A simple explicit congestion controller for wireless networks. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 353–372. USENIX Association, 2020.

[58] Prateesh Goyal, Mohammad Alizadeh, and Hari Balakrishnan. Rethinking congestion control for cellular networks. In Sujata Banerjee, Brad Karp, and Michael Walfish, editors, *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 29–35. ACM, 2017.

[59] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: A building block for internet congestion control. *CoRR*, abs/1802.08730, 2020.

[60] Prateesh Goyal, Preey Shah, Kevin Zhao, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*. USENIX Association, 2022.

[61] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 1–14. USENIX Association, 2015.

[62] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 202–215. ACM, 2016.

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

[64] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 29–42. ACM, 2017.

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

[66] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*, 1996.

[67] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In Patrick Thiran and Walter Willinger, editors, *Proceedings of the 11th ACM SIGCOMM Internet Measurement Conference, IMC '11, Berlin, Germany, November 2-, 2011*, pages 181–194. ACM, 2011.

[68] Ningning Hu and Peter Steenkiste. Estimating available bandwidth using packet pair probing. Technical report, DTIC Document, 2002.

[69] Ningning Hu and Peter Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE JSAC*, 21(6):879–894, 2003.

[70] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In Bryan Ford, Alex C. Snoeren, and Ellen W. Zegura, editors, *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016*, pages 92–98. ACM, 2016.

[71] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical PFC deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*, pages 451–463. ACM, 2017.

[72] Intel. Tofino2. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html`. 2020.

[73] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[74] Van Jacobson. Pathchar: A tool to infer characteristics of internet paths, 1997.

[75] A Jain, A Terzis, H Flinck, N Sprecher, S Arunachalam, and K Smith. Mobile throughput guidance inband signaling protocol. *IETF, work in progress*, 2015.

[76] Manish Jain and Constantinos Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. In *Passive and Active Measurements (PAM) Workshop*, 2002.

[77] Raj Jain. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN Systems*, 28(13):1723–1738, 1996.

[78] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput fairness index: An explanation. In *ATM Forum contribution*, volume 99, 1999.

[79] Hao Jiang and Constantinos Dovrolis. Source-level IP packet bursts: causes and effects. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 301–306. ACM, 2003.

[80] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. A distributed algorithm to calculate max-min fair rates without per-flow state. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2):21:1–21:42, 2019.

[81] Dina Katabi, Mark Handley, and Chalrie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[82] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. TAS: TCP acceleration as an OS service. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 24:1–24:16. ACM, 2019.

[83] Stephen T. Kent and Karen Seo. Security architecture for the internet protocol. *RFC*, 4301:1–101, 2005.

[84] Linux Kernel. Tcp bbr implementation. `https://elixir.bootlin.com/linux/v4.14/source/net/ipv4/tcp_bbr.c`.

[85] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. 2015.

[86] Leonard Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[87] Smaragda Konstantinidou and Lawrence Snyder. Chaos router: Architecture and performance. In Zvonko G. Vranesic, editor, *Proceedings of the 18th Annual International Symposium on Computer Architecture. Toronto, Canada, May, 27-30 1991*, pages 212–221. ACM, 1991.

[88] Abdesselem Kortebi, Luca Muscariello, Sara Oueslati, and James W. Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. In Derek L. Eager, Carey L. Williamson, Sem C. Borst, and John C. S. Lui, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2005, June 6-10, 2005, Banff, Alberta, Canada*, pages 217–228. ACM, 2005.

[89] Mirja Kühlewind and Richard Scheffenegger. Design and evaluation of schemes for more accurate ECN feedback. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012*, pages 6937–6941. IEEE, 2012.

[90] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 514–528. ACM, 2020.

[91] NT Kung and Robert Morris. Credit-based flow control for ATM networks. *IEEE network*, 9(2):40–48, 1995.

[92] Srisankar S. Kunniyur and Rayadurgam Srikant. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. In Rene L. Cruz and George Varghese, editors, *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, pages 123–134. ACM, 2001.

[93] Kevin Lai and Mary Baker. Measuring link bandwidths using a deterministic model of packet delay. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 283–294. ACM, 2000.

[94] Kevin Lai and Mary Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *USITS*, volume 1, pages 11–11, 2001.

[95] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[96] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In Jianping Wu and Wendy Hall, editors, *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 44–58. ACM, 2019.

[97] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E. Anderson. F10: A fault-tolerant engineered network. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*, pages 399–412. USENIX Association, 2013.

[98] Feng Lu, Hao Du, Ankur Jain, Geoffrey M. Voelker, Alex C. Snoeren, and Andreas Terzis. CQIC: revisiting cross-layer congestion control for cellular networks. In Justin Manweiler and Romit Roy Choudhury, editors, *Proceedings*

*of the 16th International Workshop on Mobile Computing Systems and Applications, HotMobile 2015, Santa Fe, NM, USA, February 12-13, 2015*, pages 45–50. ACM, 2015.

[99] Jacob B Malone, Aviv Nevo, Jonathan W Williams, et al. The tragedy of the last mile: Congestion externalities in broadband networks. Technical report, 2016.

[100] BA Mar. pchar: A tool for measuring internet path characteristics. *http://www.employees. org/˜ bmah/Software/pchar/*, 2000.

[101] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 399–413. ACM, 2019.

[102] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven Layered Multicast. In *SIGCOMM*, 1996.

[103] Paul E. McKenney. Stochastic fairness queueing. In *Proceedings IEEE INFOCOM '90, The Conference on Computer Communications, Ninth Annual Joint Conference of the IEEE Computer and Communications Societies, The Multiple Facets of Integration, San Francisco, CA, USA, June 3-7, 1990*, pages 733–740. IEEE Computer Society, 1990.

[104] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[105] Microsoft. Microsoft Azure. `https://azure.microsoft.com/`.

[106] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: rtt-based congestion control for the datacenter. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 537–550. ACM, 2015.

[107] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 221–235. ACM, 2018.

[108] Akshay Narayan, Frank Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. The Case for Moving Congestion Control Out of the Datapath. In *HotNets*, 2017.

[109] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *SIGCOMM*, 2018.

[110] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX Annual Technical Conference*, 2015.

[111] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.

[112] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *IEEE 14th International Conference on High Performance Switching and Routing, HPSR 2013, Taipei, Taiwan, July 8-11, 2013*, pages 148–155. IEEE, 2013.

[113] The Next Platform. Flattening networks - and budgets - with 400G ethernet. `https://www.nextplatform.com/2018/01/20/flattening-networks-budgets-400g-ethernet/`. January 20, 2018.

[114] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, and Felipe Huici. Flowblaze: Stateful packet processing in hardware. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 531–548. USENIX Association, 2019.

[115] Ihsan Ayyub Qazi, Lachlan L. H. Andrew, and Taieb Znati. Congestion control with multipacket feedback. *IEEE/ACM Trans. Netw.*, 20(6):1721–1733, 2012.

[116] Ihsan Ayyub Qazi, LLH Andrew, and Taieb Znati. Incremental deployment of new ecn-compatible congestion control. In *Proc. PFLDNeT*, 2009.

[117] Ihsan Ayyub Qazi, Taieb Znati, and Lachlan L. H. Andrew. Congestion control using efficient explicit feedback. In *INFOCOM 2009. 28th IEEE International*

*Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pages 10–18. IEEE, 2009.

[118] K. K. Ramakrishnan, Sally Floyd, and David L. Black. The addition of explicit congestion notification (ECN) to IP. *RFC*, 3168:1–63, 2001.

[119] Theodore S Rappaport et al. *Wireless communications: principles and practice*, volume 2. prentice hall PTR New Jersey, 1996.

[120] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: controlling user-perceived delays in server-based mobile applications. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 85–100. ACM, 2013.

[121] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. Ledbat: The new bittorrent congestion control protocol. In *ICCCN*, pages 1–6, 2010.

[122] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa H. Ammar, Ellen W. Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and WAN traffic aggregates. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 735–749. ACM, 2020.

[123] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 1–16. USENIX Association, 2018.

[124] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 183–197. ACM, 2015.

[125] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In

Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 15–28. ACM, 2016.

[126] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 44–57. ACM, 2016.

[127] M. Sridharan, K. Tan, D. Bansal, and D. Thaler. Compound TCP: A New TCP congestion control for high-speed and long distance networks. Technical report, Internet-draft draft-sridharan-tcpm-ctcp-02, 2008.

[128] Brent Stephens and Alan L. Cox. Deadlock-free local fast failover for arbitrary data center networks. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9. IEEE, 2016.

[129] Brent Stephens, Alan L. Cox, Ankit Singla, John B. Carter, Colin Dixon, and Wes Felter. Practical DCB for improved data center networks. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*, pages 1824–1832. IEEE, 2014.

[130] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 118–130, 1998.

[131] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *Internet Measurement Conf.*, 2003.

[132] Chia-Hui Tai, Jiang Zhu, and Nandita Dukkipati. Making large scale deployment of rcp practical for real networks. In *INFOCOM 2008. 27th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 13-18 April 2008, Phoenix, AZ, USA*, pages 2180–2188. IEEE, 2008.

[133] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

[134] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects`, 2005.

[135] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible

assumption? In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 565–580, 2019.

[136] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 407–420. USENIX Association, 2017.

[137] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Morley Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. In Srinivasan Keshav, Jörg Liebeherr, John W. Byers, and Jeffrey C. Mogul, editors, *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 374–385. ACM, 2011.

[138] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling BBR's Interactions with Loss-Based Congestion Control. IMC, 2019.

[139] Jim Warner. Switch buffer size. `https://people.ucsc.edu/~warner/buffer.html`. 2020.

[140] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, 2006.

[141] Robert Williams and Bahadir Erimli. Method and apparatus for performing priority-based flow control, October 18 2005. US Patent 6,957,269.

[142] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 459–471. USENIX Association, 2013.

[143] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One More Bit is Enough. *IEEE/ACM Trans. on Networking*, 16(6):1281–1294, 2008.

[144] Xiufeng Xie, Xinyu Zhang, Swarun Kumar, and Li Erran Li. pistream: Physical layer informed adaptive video streaming over LTE. In Serge Fdida, Giovanni Pau, Sneha Kumar Kasera, and Heather Zheng, editors, *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom 2015, Paris, France, September 7-11, 2015*, pages 413–425. ACM, 2015.

[145] James A Yorke. Asymptotic stability for one dimensional differential-delay equations. *Journal of Differential equations*, 7(1):189–202, 1970.

[146] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 509–522. ACM, 2015.

[147] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy H. Katz. Detail: reducing the flow completion time tail in datacenter networks. In Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese, editors, *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, pages 139–150. ACM, 2012.

[148] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E. Anderson. SWP: microsecond network slos without priorities. *CoRR*, abs/2103.01314, 2021.

[149] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 221–234. USENIX Association, 2019.

[150] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 523–536. ACM, 2015.

[151] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas E. Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 362–375. ACM, 2017.