

Improving Congestion Control Convergence in RDMA Networks

by

John Snyder

Department of Computer Science
Duke University

Date: _____

Approved:

Alvin R. Lebeck, Supervisor

Jeffrey S. Chase

Bruce M. Maggs

Xiaowei Yang

Danyang Zhuo

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2022

ABSTRACT

Improving Congestion Control Convergence in RDMA
Networks

by

John Snyder

Department of Computer Science
Duke University

Date: _____

Approved:

Alvin R. Lebeck, Supervisor

Jeffrey S. Chase

Bruce M. Maggs

Xiaowei Yang

Danyang Zhuo

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2022

Copyright © 2022 by John Snyder
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

Remote Direct Memory Access (RDMA) networks are becoming a popular interconnect technology to enable high performance communication in distributed systems. While RDMA hardware enables high bandwidth and low latency networking, networks require congestion control algorithms to ensure they operate efficiently. Ideally, a congestion control algorithm allows computers in the network to inject enough traffic to keep the network fully utilized, stops computers from causing congestion, and allocates bandwidth fairly to all computers in the network. This enables the network to operate at peak performance and be fair to all users. While many protocols eventually converge to this ideal network state over time, they often take too long, reducing performance. We develop mechanisms and protocols that improve convergence time.

In this thesis, we identify several ways in which slow convergence to the ideal network state harms performance and leaves RDMA networks susceptible to performance isolation attacks. We show that slow convergence to fair injection rates on end-hosts in RDMA networks greatly increases the communication time for long flows, which leads to sub-optimal application performance. We identify why unfairness occurs and measure unfairness' impact on application performance. We then show that because RDMA networks are loss-less and start sending packet at line-rate, users can unfairly gain more bandwidth and sometimes ignore congestion control altogether. This allows misbehaving users to harm the performance of other users

sharing the network.

To improve long flow performance in RDMA networks, we propose two new mechanisms for Additive-Increase Multiplicative-Decrease protocols: 1) Variable Additive Increase and 2) Sampling Frequency. These mechanisms reduce the time it takes for the network to allocate bandwidth fairly between end-hosts, so long flows are not starved of bandwidth. To create these mechanisms, we determine when unfairness occurs and how end-hosts can infer unfairness without any additional information from switches.

We then introduce One Round Trip Time Convergence (1RC) and a new method of setting flow weights, which improve performance and isolation in RDMA networks. 1RC enables a network to converge to fair rates during the first RTT by dropping packets when a flow uses too much bandwidth. We do this while maintaining packet ordering and fairness between flows that start sending packets at the same time. We then use a new weighting scheme, which decreases the bandwidth allocation to users opening too many connections. A lower weight for misbehaving users mitigates the impact of a user trying to gain more bandwidth than is fair.

Finally, we introduce the Collective Congestion Control Protocol (3CPO), which improves convergence in multicast networks designed for collective communication. Multicast operations send a single packet to several destinations at once and cause severe congestion in the network quickly. 3CPO manages multicast congestion by inferring global congestion state through multicast operations and then tunes each end-host injection rate to be near optimal. 3CPO requires no extra information from switches and works entirely on end-hosts.

To my parents.

Contents

Abstract	iv
List of Tables	xii
List of Figures	xiii
Acknowledgements	xvi
1 Introduction	1
1.1 Congestion Control Overview and Issues	2
1.2 Congestion Control Performance and Isolation Issues	4
1.3 Improving Long Flow Tail Latency	5
1.4 One RTT Convergence and Weighting Scheme	6
1.5 Fast Convergence in Multicast Networks	7
1.6 Organization of Dissertation	8
2 Congestion Control Background	9
2.1 Network Congestion	9
2.1.1 Congestion in Lossy Networks	10
2.1.2 Lossless Networks	11
2.2 Congestion Control Protocols	13
2.2.1 Sender Side Congestion Control	14
2.2.2 Receiver-based Congestion Control	17
2.2.3 Switch-based Congestion Control	18

2.3	Additive-Increase Multiplicative-Decrease	20
2.4	Remote Direct Memory Access	22
2.5	Congestion Control Evaluation	23
2.5.1	Simulations	24
2.5.2	Congestion Control Metrics	24
2.6	Summary	26
3	Performance and Isolation Issues in RDMA Networks	27
3.1	Long Flow Tail Latency	28
3.2	Background	29
3.2.1	HPCC	29
3.2.2	Swift	30
3.3	Sources of Unfairness	32
3.3.1	Conservative Additive Increase	32
3.3.2	One Reaction Per RTT	33
3.3.3	Deterministic Feedback	34
3.3.4	Methodology	34
3.3.5	Unfairness in HPCC and Swift	40
3.4	Performance Isolation Attacks	41
3.4.1	IB and Isolation Background	43
3.5	RDMA Congestion Control Attacks	44
3.5.1	Parallel QP Attack	45
3.5.2	Staggered QP Attack	45
3.5.3	Shuffled Overlay Attack	46
3.6	Attack Evaluation	47
3.6.1	Testbed Experiments	48

3.6.2	Datacenter Simulations	51
3.7	Potential Solutions and Future Work	52
3.8	Summary	54
4	Improving Long Flow Tail Latency	56
4.1	Variable Additive Increase and Sampling Frequency	58
4.1.1	Variable AI	59
4.1.2	Sampling Frequency	61
4.2	Implementation	64
4.2.1	Variable AI	65
4.2.2	Sampling Frequency	66
4.3	Evaluation	67
4.3.1	Methodology	67
4.3.2	Experimental Results	70
4.4	Conclusion	74
5	Towards RDMA Performance Isolation and Effective Congestion Control in High BDP Networks	76
5.1	Background	79
5.1.1	s-PERC	79
5.1.2	Speculative Packets	82
5.1.3	Synchronized Clocks	83
5.1.4	Diminishing Weight Scheduling	84
5.2	1RC Design	84
5.3	Setting Weights	88
5.3.1	1RC, Weights, and Fairness	92
5.4	Implementing 1RC and Weights in s-PERC	95
5.5	Evaluations	98

5.5.1	1RC Performance Evaluations	98
5.5.2	Performance Isolation Evaluation	101
5.6	Conclusion	107
6	Multicast Congestion Control	108
6.1	Background and Motivation	110
6.1.1	Collective Communication	111
6.1.2	Swift	112
6.1.3	Injection Throttling	112
6.1.4	Existing Multicast CC	114
6.2	3CPO Design	118
6.2.1	Intuition of Multicast CC in 3CPO	118
6.2.2	Multicast CC in 3CPO	119
6.2.3	Unicast CC	123
6.2.4	Conversion from Multicast CC to Unicast CC	124
6.2.5	Virtual Channels and Arbitration	129
6.3	Evaluation	130
6.3.1	Methodology	130
6.3.2	Varied Initiators	131
6.3.3	Symmetric Multicast Workloads	133
6.3.4	Dynamic Traffic Patterns	134
6.3.5	Shared Network Workloads	136
6.4	Conclusion	137
7	Conclusion	138
7.1	Key Contributions	139
7.2	Future Work	141

7.2.1	Hardware Implementations	141
7.2.2	Sharing In-Network Collective Resources	142
7.2.3	Adaptive Routing and Congestion Control	142
7.2.4	Improving and Implementing 3CPO	143
7.3	Conclusion	144
	Bibliography	145
	Biography	157

List of Tables

3.1	HPCC and Swift Parameter Settings	36
3.2	Various Environments Susceptibility to Attack.	53
4.1	HPCC and Swift VAI SF Parameters	68
6.1	Tuned Window Size for Different Number of Initiators	115

List of Figures

2.1	Topology and network flows that cause Head-of-Line Blocking	11
2.2	Congestion Causing Head-of-Line Blocking	12
2.3	Saw-tooth Behavior of TCP and DCTCP	20
3.1	Microbenchmark Topology	35
3.2	Experimental Datacenter Topology	37
3.3	Start Time vs. Finish Time 16-1 Staggered Incast HPCC	37
3.4	Start Time vs. Finish Time 16-1 Staggered Incast in Swift	38
3.5	Jain Fairness Index and Queue depth during Incast Traffic in HPCC and Swift	39
3.6	Tail Latency 99.9% for Swift and HPCC	39
3.7	Changing Ring to create more src/dst pairs.	46
3.8	Experiment Dumbbell Topology	48
3.9	Parallel QP Attack Testbed Results	48
3.10	Parallel and Staggered Attack BW 1.125MB Transfer	50
3.11	Victim Traffic DC Simulation	51
3.12	99% and 99.9% tail latency of HPCC flows with different CC enforcement. The vertical line shows Bandwidth Delay Product.	53
4.1	Unfairness occurring when starting at line rate	57
4.2	Plotting difference in fairness between to MD methods. $r = 30000$, $MTU=1000$, $s = 30$, $\beta=.5$	64
4.3	16-1 Incast Traffic Start Time vs Finish Time with HPCC	69

4.4	16-1 Incast Traffic Start Time vs Finish Time with Swift	69
4.5	16-1 Incast in HPCC and Swift	70
4.6	Jain Fairness Index and Queue depth during Incast Traffic with 96-1 Incast in Swift and HPCC	71
4.7	99.9% FCT for various flow sizes in Hadoop Traffic	72
4.8	99.9% FCT for various flow sizes in WebSearch and Storage Traffic .	73
4.9	Median FCT for various flow sizes in Hadoop Traffic	73
4.10	Median FCT for various flow sizes in WebSearch and Storage Traffic .	74
5.1	1RC example demonstrating how 1RC generates window table entries, indexes the window table, and drops packets.	86
5.2	Flows 1 and 2 belong to the same end host. Starting two flows in the same Epoch enables an end host to send more speculative packets. . .	92
5.3	Flows 1 and 2 belong to the same end host. The flows from the same end host share speculative sequence numbers.	93
5.4	Flows 1 and 2 belong to the same end host. Flow 2's weight increases because flow 1 already existed when it started.	94
5.5	Alibaba Storage 99.9% tail latency using 1RC alongside s-PERC. . .	99
5.6	Alibaba Storage 50% tail latency using 1RC alongside s-PERC. . . .	100
5.7	Impact of staggered attack on network performance in 16-1 Incast benchmark average victim BW with and without 1RC and Weights .	101
5.8	Attacking Flow FCT in 16-1 Incast with and without 1RC and Weights (lower is better)	102
5.9	s-PERC with and without 1RC and weights simulating Hadoop traffic in a datacenter with and without staggered attacks	104
5.10	Packet latency during staggered attacks using s-PERC variants	104
5.11	Hadoop 99.9% FCT slowdown multiplied by weight	106
6.1	Watermark throttling reducing injection rate of 5-flit packets	113
6.2	Effective bandwidth of All-Reduce	116
6.3	Average and 100% Tail Packet Latency of All-Reduce	116

6.4	Example of injection rate throttling using 3CPO for 5-flit multicast packets	118
6.5	Process of matching local and remote operations and consuming IRL Tokens.	120
6.6	Tracking local requests reduces the chance of matches between requests that do not contend for bandwidth.	121
6.7	Incrementing <i>unmatched</i> if outstanding requests is greater. <i>Outstanding</i> operations increments as items in the remote request queue expire.	122
6.8	Congestion resolving with exclusively multicast CC. Reducing the <i>watermark</i> after the first RTT would result in a loss of throughput. . . .	126
6.9	Demonstrates when unicast CC takes over if congestion persists and the multicast CC does not increase, the 3CPO will throttle the rate. .	126
6.10	Effective Bandwidth of in All-Reduce Workload 64 GPUs	131
6.11	Average and Tail Packet Latency in All-Reduce Workload 64 GPUs .	131
6.12	Effective Bandwidth of in All-Reduce Workload 64 GPUs	132
6.13	Effective Bandwidth of Symmetric Workloads 128 GPUs	133
6.14	Average Packet latency in Symmetric Workloads with 128 GPUs . . .	133
6.15	Effective Bandwidth in Dynamic Workloads 128 GPUs	135
6.16	Average Packet Latency of Dynamic Workloads 128 GPUs	136
6.17	Effective Bandwidth of Shared Network Workloads 128 GPUs	136

Acknowledgements

First and foremost I must thank my advisor, Alvy, for his support over the past four years. He allowed me to pursue research that excited me while ensuring I kept my eyes on the big picture of the research's overarching goals. His insightful comments about our research enabled many of our projects to succeed. This thesis would not have been possible without him.

I would also like to thank my committee members, Jeff, Bruce, Xiaowei, and Danyang, who have all given great feedback about my work and great ideas about how to improve it. Danyang was heavily involved in parts of this thesis and made countless important contributions to improve it, for which I am extremely grateful.

I was also fortunate enough to work with several fantastic folks at Nvidia (and formerly Mellanox) that helped shape how I view problems. Ted Jiang, Al Davis, Larry Dennison at Nvidia Research were integral in the development of 3CPO and improving congestion control in multicast networks. Sylvain Jeaugey, Sreeram Potluri, and Jim Dinan taught me valuable lessons about communication patterns that are important to modern distributed systems, which helped motivate much of this thesis. Rich Graham introduced me to many topics in High Performance Computing.

I also want to thank my lab mates, Sahba Tashakkori, Mike Zhang, Ramin Bashizade, Chris Kjellqvist, and Pulkit Misra, with whom I shared many good conversations and laughs.

Another incredibly impactful person on my decision to attend graduate school

and pursue my Ph.D. was Brian Larkins at Rhodes College. He invited me to do research with him, was my earliest mentor, and taught me what research was and how to do it effectively. Without working with Brian, I would never have known that I wanted to pursue a career in research. I cannot thank him enough for providing me with direction.

My family was also incredibly supportive throughout my Ph.D. My parents in particular helped me with the emotional support and sage advice. They are my closest friends and most trusted confidants. I also must thank my siblings, Megan, Dylan, and Gabe for their help. Dylan in particular is always a good friend and sounding board. Many of my aunts and uncles also provided support, Eric, Lynn, Anne, Rob, and Marie, thank you. I also need to thank my closest friends Nick and Ryan for often providing a welcome distraction. My friends and roommates in Durham, Sam and Michael, also provided support and good times.

1

Introduction

A myriad of workloads rely on tightly-coupled distributed systems to meet application computation and storage demands. Numerous scientific fields such as physics [12, 14, 13], fluid dynamics [6, 93], computational chemistry [16], and biology [38] utilise high-performance computing (HPC) systems to enable large simulations, which model physical phenomenon. Additionally, datacenters are distributed systems that support distributed storage, big-data, deep learning, and website hosting workloads. While the workloads that utilise distributed systems vary wildly, they share two common characteristics; 1) vast amounts of storage and 2) computation.

Distributed systems provide the horizontal scaling required to store and compute upon vast datasets. However, scaling software beyond a single machine opens additional opportunities for performance bottlenecks. All distributed workloads require the various computational devices to communicate because the system stores data in physically distributed places throughout the system. Distributed systems utilize a network that allows the computational devices to exchange data.

In order to prevent the network from bottlenecking application performance, tightly-coupled distributed systems generally use high-speed, low latency intercon-

nection networks. This enables applications to move large amounts of data quickly. Common open standard networks are Ethernet [79] and Infiniband [45], which provide bandwidth up to 400Gb/s per port [87, 86]. Several high performance proprietary networks also exist, such as Nvidia’s NVSwitch [85], AMD’s Infinity Fabric [71], Google’s TPU supercomputer network [56], and Cray’s Slingshot [24]. A variety of systems use these different networks to provide applications with high performance communication.

While the underlying hardware supports fast networking, congestion can severely limit performance. Network congestion occurs when the bandwidth requested by the ingress ports on a switch exceeds the bandwidth of the egress port. During congestion, egress ports store packets in queues, which can lead to performance issues. As queue depth increases, packet latencies increase because packets sit in queues instead of traversing the network. If congestion becomes severe enough that the queue is full, packet drops and queue saturation will occur and degrade throughput [107, 90]. To manage congestion in the network and therefore improve latency and throughput, networks often deploy congestion control algorithms, which limit the injection of packets into the network. Distributed applications often require networks with congestion control to achieve high performance.

1.1 Congestion Control Overview and Issues

Despite being a research topic since 1988 [46], network congestion still negatively impacts network performance. This is largely because the solution space for network congestion varies widely depending on system characteristics. For example, TCP works well in wide area networks like the internet but performs poorly in local area networks [4]. Even within local area networks the potential solutions are numerous. For example, Google developed Swift [66] because Google’s datacenters have simple

switches with no programmability, so Swift works with only end-to-end metrics. Using a completely programmable network, Jose et al. developed s-PERC [55], which requires switches to compute injection rates for each end host and modify packets as the packets flow through the switch.

Ideally, a congestion protocol enables end hosts to inject enough traffic to saturate the network without exceeding the bandwidth of the most in-demand network link. This enables high throughput and no network congestion. Additionally, the congestion control protocol should have a fairness policy that ensures end hosts share bandwidth fairly. Common examples of fairness policies are max-min fairness [32] and proportional fairness [62]. An optimal allocation of bandwidth between end hosts saturates the network, does not cause congestion, and is fair.

A congestion control algorithm cannot converge to the optimal allocation instantly because the demands of each end host are not globally known. Therefore, congestion control protocols iteratively probe for the optimal injection rate. We define convergence as the time between end hosts demanding bandwidth and converging to the optimal bandwidth allocation. While protocols rarely perfectly converge to the optimal rate, an allocation close to optimal is often sufficient.

This dissertation identifies several issues with convergence in congestion control protocols for tightly-coupled distributed systems. We find that slow convergence causes poor performance in end hosts that have long flows. Slow convergence also enables several performance isolation attacks in Infiniband (IB) [45] and RDMA over Converged Ethernet (RoCE) [44], common open standard networks. We also find that multicast networks, which support a single end host sending to several destinations, converges to near-optimal injection rates much faster when exploiting the symmetry of certain traffic patterns. We now summarize the contributions of this dissertation.

1.2 Congestion Control Performance and Isolation Issues

We identify how slow convergence in certain networks causes performance and isolation issues. The problems identified occur in networks that are loss-less (packets cannot be dropped) and allow end hosts to start sending packets at line rate. These characteristics are common in high-performance networks, most notably in Infiniband [45] and RoCE [44]. This section focuses solely on systems that calculate injection rates on end hosts instead of on network switches.

Initially sending packets at line rate is a performance optimization that allows short flows to complete quickly. A flow is a sequence of packets that must be delivered in order, and the flow completion time (FCT) (how long it takes a flow to send all its packets) is an important congestion control metric [28]. End hosts in most RDMA networks assume the network is not congested and do not reduce their injection rates until the end hosts detect congestion in the network. We summarize congestion detection methods in Chapter 2.

Starting flows at line rate can cause fairness issues because a new flow has more bandwidth than existing flows. If the network is saturated, the new flow causes all flows to reduce their injection rates because the new flow causes network congestion. The congestion control algorithm then calculates new, reduced rates for each flow, which hopefully alleviates congestion. However, since most congestion control algorithms use Additive-Increase Multiplicative-Decrease (AIMD) because of its fairness guarantees [19], the new flow with more bandwidth still has a higher allocation. Over time the bandwidth allocation becomes fair, but, if the convergence to fair rates is slow, the performance of flows allocated too little bandwidth suffers.

Favoring new network flows over existing flows causes a long tail for large flows. Large flows are bandwidth bound, take several network round trips to complete, and therefore exist for a long period of time. While long flows are active, new flows

continuously join the network, which lowers the injection rate of long-lived flows. A long flow is then unfairly allocated too little bandwidth. Not all long flows experience this unfairness, but a small percentage do, and even a small percentage of flows taking a long time to complete can negatively impact application performance [25]. We find that unfair bandwidth allocations often cause long flows to take several times longer to complete than is necessary.

Misbehaving users can exploit slow convergence in IB and RoCE networks to gain an unfair amount of bandwidth. IB and RoCE follow the same specification; RoCE simply allows IB in an ethernet network. IB and RoCE enable RDMA, which removes host networking software from the communication critical path. We identify several vulnerabilities in the congestion control specification in IB and RoCE that allow a misbehaving user to gain extra bandwidth or completely ignore congestion control. We demonstrate the attacks on current hardware when emulating the IB specification and show how nefarious users benefit from the attack and well-behaved users suffer. In order to remove the attacks, flows can no longer start sending packets at line rate, which negatively impacts the performance of short, latency bound flows. We demonstrate this trade-off with Datacenter simulations.

1.3 Improving Long Flow Tail Latency

In the preceding section, we described how some congestion control protocols favor new flows over existing flows, which harms the performance of large flows, particularly at the tail. We create end host based mechanisms that substantially improve long flow tail latencies. Network switches have limited processing power and an in-network congestion control solution is complex and expensive. Therefore, modifying the end host congestion control protocol is the most practical solution of slow convergence. We improve long flow tail latency and do not require any additional information from

network switches.

We improve convergence to fair rate allocations by identifying when unfairness occurs and how to infer unfairness at the end host. We introduce our two new mechanisms: Variable Additive Increase and Sampling Frequency, which significantly improve the tail FCT of long flows. We augment Swift [66] and HPCC [75] with our new mechanisms in ns-3 [95] and show their performance in micro-benchmarks and datacenter simulations. When we improve the convergence to fair rate allocations in HPCC and Swift, the tail FCT of long flows dropped by half without affecting small flow performance.

1.4 One RTT Convergence and Weighting Scheme

Two issues challenge existing RDMA congestion control methods: 1) as bandwidth delay product (BDP) existing methods of congestion control become ineffective and 2) RDMA is becoming more common in shared environments. Starting flows at line rate is a performance optimization that allows short flows to complete quickly. However, it leaves the network susceptible to performance isolation attacks and causes congestion quickly. To maintain starting at line rates performance benefits, mitigate performance isolation attacks, and minimize congestion, we introduce One RTT Convergence (1RC). 1RC lets flows start at line rate and drops packets when letting the packet through would be unfair. It also does not require detecting packet drops, maintaining per-flow state on the switch, or reordering packets. Switches can easily support 1RC with small modifications.

We also show how we can weight certain network flows to mitigate a misbehaving user’s ability to unfairly gain bandwidth. We lower a flow’s weight as the end host opens more connections. This discourages a user from opening more connections to gain more bandwidth.

We demonstrate with datacenter workloads that 1RC enables congestion control

to effectively manage congestion as BDP grows when existing methods fail. We then demonstrate with datacenter workloads and microbenchmarks that 1RC and our new weighting scheme effectively isolate performance between users, even when one is misbehaving.

1.5 Fast Convergence in Multicast Networks

Point-to-point communication is the most common communication paradigm; one computer sends a sequence of packets to another computer. However, many distributed applications benefit from collective communication, where a group of servers all communicate with one another. Collective communication is common in scientific computing [64] and deep learning [8] workloads, so research efforts accelerated collective communication with in-network computing [65, 67, 36, 97, 73]. These networks can increase the throughput of large collective operations by up to 2x and improve latency sensitive small collectives even more drastically.

Our network performs in-network collectives with an architecture similar to Klenk et al.’s [65] where users implement collective operations in a memory fabric with multicast writes and read reductions. This network architecture is effective at improving collective operations, but a single packet expands in the network and can cause congestion at several points in the network simultaneously. We develop the Collective Congestion Control PrOtocol (3CPO) for Klenk et al.’s architecture that exploits symmetry of collective operations to converge to near-optimal rates faster. When performing an all-reduce collective operation with 64 GPUs, our protocol dynamically changes the injection rate of packets and achieves throughput within 98% and has average packet latency 200 cycles less than that of a hand-tuned static congestion window. We also design our protocol to work with unicast traffic in addition to in-network collective traffic. 3CPO works only on the end hosts and requires no switch modifications.

1.6 Organization of Dissertation

The rest of this dissertation is as follows. Chapter 2 provides background information on congestion control and RDMA networks. Chapter 3 details and demonstrates how slow convergence affects long flow completion time and leaves IB and RoCE networks susceptible to performance isolation attacks. Chapter 4 explains how we solve the issue of long flow completion times using an end host only solution. Chapter 5 shows how we can use in-network computing and synchronized system clocks to enable faster convergence. Chapter 6 introduces 3CPO, a congestion control protocol that improves network performance when using multicast operations. Finally, we conclude in Chapter 7.

2

Congestion Control Background

Congestion dramatically reduces a network's effectiveness. This Chapter gives specific details and examples how congestion effects throughput and latency in different types of networks. Section 2.2 explains how congestion control protocols reduce congestion, and the different places we can implement congestion control. Section 2.3 details the Additive-Increase Multiplicative-Decrease paradigm, and its desirable properties. Section 2.4 briefly explains RDMA and why we use it for many of our experiments. Finally, we discuss how we evaluate our protocols and mechanisms and the key metrics we evaluate congestion controls protocols with in Section 2.5.

2.1 Network Congestion

Network congestion occurs when end hosts' bandwidth demands exceed a switch's physical bandwidth capabilities. Congestion occurs at the bottleneck link, which is the most in-demand link along a flow's path. Network end hosts generally demand bandwidth with a network flow, which is a sequence of packets that must be delivered in-order to the destination. When a network link cannot meet end hosts' bandwidth demands, the switch must queue packets. A small amount of queueing is necessary

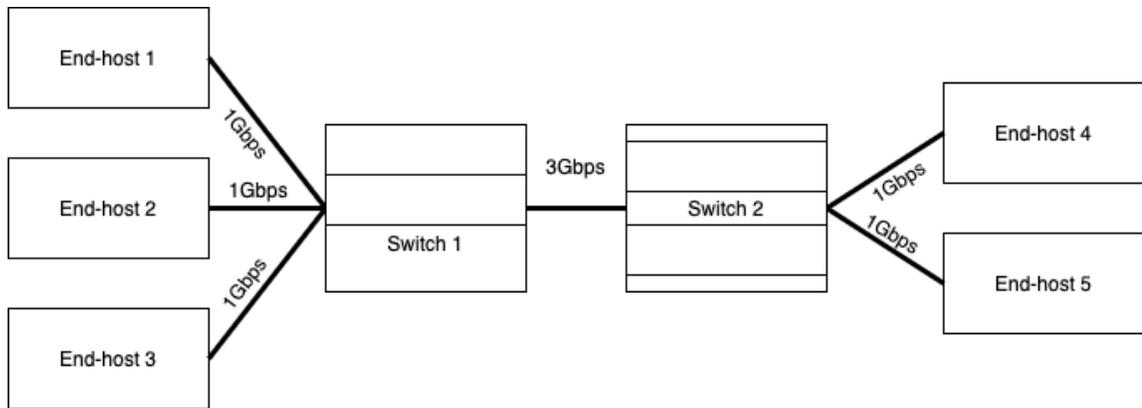
and does not harm performance, but large queues can increase packet latencies and reduce network throughput. We now explain exactly how network congestion harms throughput and latency in lossy and lossless networks.

2.1.1 Congestion in Lossy Networks

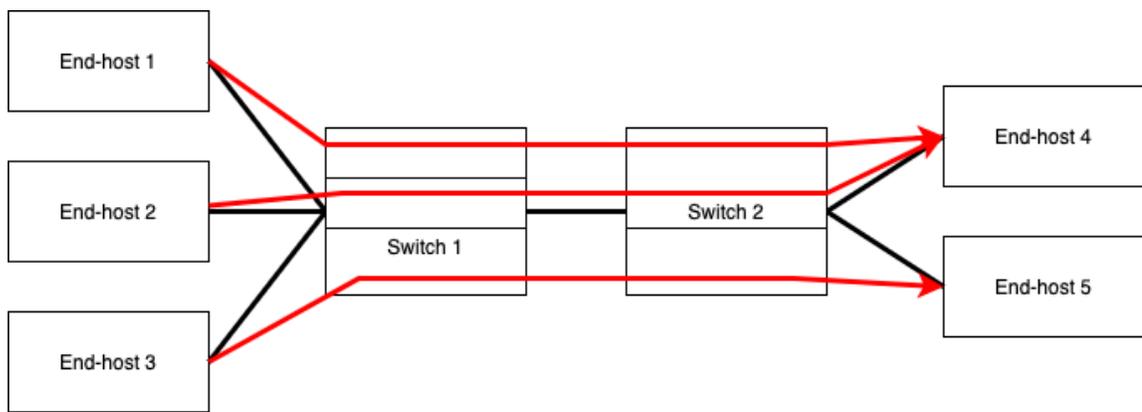
When a packet arrives at a switch port with a full packet queue in a lossy network, the congested port drops the packet because the switch has no available storage. While necessary, dropping packets severely damages a network's throughput. In transfer protocols such as TCP, the network has to deliver all packets, so an end host must detect packet drops and resend dropped packets. Further, TCP requires packets arrive in-order, so when the network drops a single packet, TCP must resend any subsequent packets as well.

TCP detects packet drops with two mechanisms. First, TCP sets a timer each time it sends a packet. If the network does not acknowledge the packet's delivery before the timer expires, the end host assumes the packet is lost. TCP also detects loss through the destination end host notifying the sending end host that a packet is missing. When TCP detects loss, it resends the missing packet. Resending packets reduces throughput because the end host cannot use valuable bandwidth sending new packets. Instead the end host uses bandwidth resending lost packets and any subsequent packet, so the end hosts wastes cycles redoing tasks.

In interactive applications such as ssh connections, online gaming, and some websites, low packet latency ensures a satisfactory user experience. In any network, congestion directly increases packet latency because packets spend time sitting in switch queues instead of traversing the network. Packet loss further degrades latency because detecting packet loss takes a significant amount of time. Loss detection timers must be long enough to not retransmit packets that were not actually dropped [46], so TCP usually sets the timeout to be several times longer than the network's minimum



(a) Example Dumbbell Topology



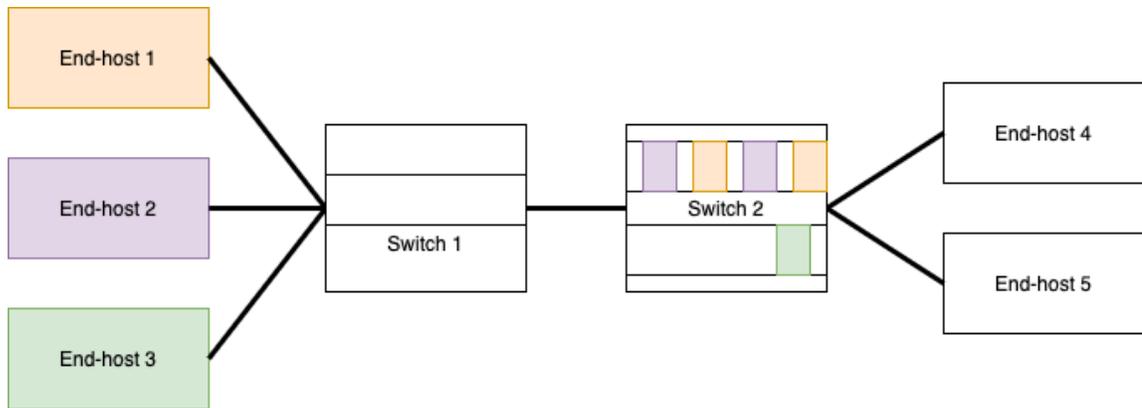
(b) Source Destination Pairs

FIGURE 2.1: Topology and network flows that cause Head-of-Line Blocking

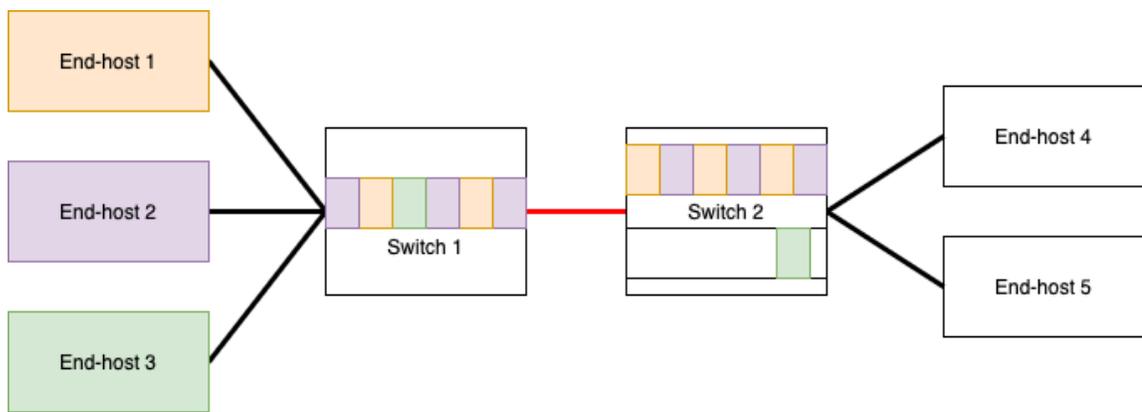
round trip time (RTT). Therefore, the end host must wait before retransmitting the packet because the end host cannot determine if the packet was lost. This long wait significantly increases time-sensitive data transfers latency.

2.1.2 Lossless Networks

Packet loss detection and retransmission hinders lossy network performance. To remove packet loss issues, some networks do not drop packets during congestion. Lossless networks utilise link-level flow control mechanisms like Priority Flow Control (PFC) Pause Frames in Ethernet and Credit-based flow control, both of which stop upstream ports and switches from sending packets when packet queues fill up.



(a) Congestion Begins at Switch 2 to End host 4



(b) Congestion propagates backwards through the network

FIGURE 2.2: Congestion Causing Head-of-Line Blocking

Congestion still increases packet latencies and reduces throughput in lossless networks despite not detecting loss and retransmitting packets.

Like a lossy network, packets in lossless networks suffer from queueing delay, which increases latency. Congestion causes severe queueing delay in lossless networks because congestion propagates through the network, often called tree-saturation [90]. Figures 2.1 and 2.2 demonstrate congestion propagating in a lossless network. Figure 2.1(a) shows the example dumbbell topology, with three end hosts on the left side of the topology and two on the right side. The link between Switches 1 and 2 can support End hosts 1-3 sending at line rate, so it should never bottleneck transfers. The smaller boxes within the switch boxes are packet queues. Figure 2.1(b) shows

each end host's path with red lines. End hosts 1 and 2 send packets to End host 4, and End host 3 sends packets to End host 5. The link between Switch 2 and End host 4 bottleneck End hosts 1 and 2 packet transmission to End host 4. Figure 2.2(a) shows that because End host 1 and 2 send at 2Gbps and the link between Switch 2 and End host 4 is only 1Gbps, packets queue on End host 4's port on Switch 2. This harms latency for End host 1 and 2 but is inevitable given the bandwidth demands. A larger problem occurs when Switch 1 becomes congested. Figure 2.2(b) shows what happens when Switch 2's queue for End host 4 fills with packets. Because Switch 2 cannot drop packets, Switch 2 stops Switch 1 from sending packets along the link connecting them, denoted with the link turning red. This causes a queue to form on Switch 1 despite Switch 1 having plenty of bandwidth! Now, packets from End host 3 queue on Switch 1 and latency increases even though End host 3 does not share a bottleneck with End host's 1 and 2. This would not occur in a lossy network because Switch 2 would simply drop packets from End host 1 and End host 2, and packets from End host 3 would move freely through the network.

The same example demonstrates a loss of throughput in lossless networks. The example should have 2Gbps of throughput, but now the entire network is limited by the link between End host 4 and Switch 2. Throughput would drop to around 1Gbps because Switch 1 can now only send packets as quickly as Switch 2. We call this phenomenon Head-of-Line (HoL) blocking, and it annihilates network performance [60].

To prevent increased packet latencies and reduced throughput, networks employ congestion control protocols, which limit congestion in the network.

2.2 Congestion Control Protocols

Congestion occurs when end hosts send too many packets. Networks limit the injection of packets with congestion control protocols. There are three network locations

where a congestion control protocol can allocate bandwidth: 1) sender 2) receiver and 3) switch. In sender-based congestion control protocols, the end host sending data adjusts packet injection based on feedback received from the network. Sender-based protocols are common because they require little involvement from network switches and destinations.

In receiver-based congestion control, a packet flow's receiver explicitly schedules when the sender should send data. Receiver-based congestion control is effective but requires that congestion only occurs at the last-hop switch. In many networks, like the internet, that is a bad assumption. For congestion to only happen at the last-hop switch, the network cannot be over-subscribed, which makes the network expensive.

In switch-based congestion control, switches calculate each flow's injection rate and deliver rate information to senders. Switch-based protocols work well but require complex switches that are often expensive and difficult to program.

Subsequent sections provide more details on each type of congestion control protocol. In this dissertation, we prefer sender-side solutions because they are the simplest but also use switch-based solutions when necessary.

2.2.1 Sender Side Congestion Control

TCP [17] was the first sender-side congestion control protocol. In TCP, a congestion window limits a flow's in-flight bytes, which limits the amount of congestion. If the network drops or delivers a packet out-of-order, TCP re-sends the affected packets and all subsequent packets. TCP detects a drop if a packet is not acknowledged within a certain amount of time or the receiving end host notifies the sending end host that a packet is missing.

While effective for over a decade, in 1986, researchers at Lawrence Berkeley Lab (LBL) observed that throughput between LBL and the University of California Berkeley, despite being 400 yards apart, dropped from 40Kb/s to 40b/s [46]. In-

terestingly, Jacobson found that end hosts assumed switches dropped packets, but that was not the case. Instead, the switch queued packets for so long that the sender assumed the switch dropped the packets. Because the switch did not drop the packets, packets that TCP assumed were lost filled the switch queues and TCP resent packets that the switch had not dropped!

Jacobson fixed the observed issues with several changes to the congestion window in TCP [46]. Instead of a fixed window, TCP now dynamically changes the number of inflight bytes based on network conditions. If the network reliably delivered packets, TCP increased the congestion window additively. If the network drops a packet, TCP reduces the window multiplicatively. We refer to this as an Additive-Increase Multiplicative-Decrease (AIMD) protocol, which Chiu et al. later proved has desirable properties [19]. Section 2.3 has more details on AIMD principles. Jacobson proposed increasing the congestion window by 1 packet if the network delivered the previous window properly and cutting the window in half if the network lost a packet.

While the exact algorithm described by Jacobson is scarcely used today, TCP still uses AIMD algorithms and dynamically changes a congestion window based on congestion. Microsoft, Google, and Alibaba applied these principles to congestion control algorithms for their datacenters [4, 66, 75]. Timely [81], DCQCN [110], and congestion control in Infiniband [45] decided against a congestion window and instead limit packet injection by adding time between the injection of each packet. This allowed end hosts to send many packets quickly but significantly increased congestion [75, 82] and packet latency. More recent protocols reintroduce congestion windows to improve packet latency [75, 66] because Li et al. and Mittal et al. found that a congestion window equal to bandwidth delay product (BDP) only limits throughput if the network is congested.

Jacobson's protocol detects congestion when a packet is dropped, which leads to lower throughput because it requires an extremely congested network before end

hosts reduce their injection rates. To reduce the level of congestion required to drop packet, Floyd and Jacobson [30] proposed Random Early Detection (RED), which drops packets probabilistically based on queue size. As the queue becomes larger, the switch is more likely to drop each admitted packet. This is an improvement on waiting for the queue to become full to drop a packet because end hosts reduce their rates earlier.

While an effective mechanism to detect congestion, packet loss severely harms throughput because end-points need to detect and retransmit lost packets. To detect congestion before the network has to drop packets, Brakmo et al. proposed using packet round trip time (RTT) as a congestion signal [15] in TCP. The length of time between a packet entering the network and its acknowledgement arriving at the sending end host is the RTT. RTT increases as congestion increases because a larger RTT indicates the packet spent significant time in a switch packet queue. Many TCP variants use RTT [54].

Unlike packet drops, RTT measurements can effectively measure congestion in lossless networks [66, 81]. Some networks prevent packet loss due to packet buffer overflow with link-level flow control mechanisms, which stop upstream end-points from sending packets before buffers fill up. Lossless networks provide higher performance than lossy networks and are often used in local area networks. Wide-area networks need to be lossy because the network is not tightly controlled, and even one lossy switch makes a lossless network into a lossy network. However, in tightly controlled setting like a datacenter or supercomputer, lossless networks are feasible.

As network switches became more sophisticated, congestion information from switches became possible, and many congestion control algorithms rely on switches for congestion information. Along with introducing RED, Floyd and Jacobson proposed marking a bit in a packet header to indicate congestion instead of dropping the packet [30]. This idea later became Explicit Congestion Notification (ECN), an

extension to the Internet Protocol (IP) header [92]. DCQCN [110], a widely used datacenter congestion control algorithm, still use ECN and RED to detect network congestion. While ECN is useful, it provides no information about the extent of congestion or network under-utilization. Recently, Alibaba developed HPCC [75], which utilizes In-Band Network Telemetry (INT). INT is an extension to P4 [11] and adds switch telemetry to packets as they move through the switch. HPCC utilizes 3 pieces of telemetry, 1) queue depth 2) transmitted (tx) bytes and 3) timestamps. Queue depth tells HPCC the extent of congestion, and HPCC calculates throughput using tx bytes and timestamps. INT and HPCC manage queues well and maintain high throughput.

Protocols like HPCC, DCQCN, and DCTCP require specific parameter settings and functionality from switches and end host hardware. Protocols with these requirements are not suitable for a wide-area network like the internet, and are only deployed in tightly controlled settings like datacenters and HPC systems. This thesis focuses solely on protocols for tightly controlled settings.

Chapter 3 explains performance issues in some sender-side congestion control protocols. The identified issues show that many congestion control protocols optimize for certain metrics, which harms performance in other key metrics. Chapter 4 proposes new mechanisms for sender-side reaction protocols that improves performance across negatively impacted metrics without harming performance in any other area. We implement our new mechanisms in HPCC [75] and Swift [66].

2.2.2 Receiver-based Congestion Control

In highly controlled settings like a datacenter or HPC system, receiver-based protocols are feasible because both end hosts belong to the same user. Receiver-based protocols assume that congestion only happens at a flow's last hop. Therefore, the receiver knows all connections and demands along the bottleneck link (the last hop),

and can perfectly schedule traffic. pHost [33], NDP [42], LHRP [51], and Homa [83] are receiver-based congestion control protocols.

There are two issues with a receiver-based congestion control algorithm: 1) it requires a non-oversubscribed network and 2) scheduling overhead can harm packet latency. If a network is oversubscribed, places beyond the last-hop are susceptible to congestion and the receiver no longer has a global view of congestion. An over-subscribed network requires that each level of the topology has as much bandwidth going into the network as it does coming out of the network. For example, a Top-of-Rack (ToR) switch, which connects end hosts to the network, must have as much bandwidth going to end hosts as it does into the network. Over-subscribed networks are cheaper because they require fewer switches to construct and have less bandwidth.

Receiver-based protocols require a flow to schedule itself at the receiver by sending connection establishment packets. The connection establishment overhead increases latency because flows require a minimum of two RTTs to complete (scheduling RTT and data RTT) instead of one (data RTT). Speculative packets [51, 42, 43] remove the scheduling overhead when the network is not congested. Speculative packets are lower priority than scheduled packets, and the network drops speculative packets if the network is congested, so in the worst case, scheduling overhead still harms performance.

2.2.3 Switch-based Congestion Control

Switches have the most knowledge about congestion state in the network, so some protocols rely on the switches to calculate each end hosts injection rate. XCP [61], s-PERC [55], RoCC [103], ExpressPass [20], and RCP [28] all receive explicit or implicit rate information from switches, which end hosts use for injection rates. Switch based protocols are effective and work in any type of network. However, all switches in the network must be programmable. For example, s-PERC [55], a distributed max-min

fair congestion control algorithm that converges in bounded time, requires an FPGA on the network switch. This makes the network more expensive and complicated. In Chapter 5, we modify s-PERC to mitigate performance isolation attacks in certain networks.

The previously described congestion control protocols require end hosts comply with their assigned rates. To enforce perfectly fair and efficient congestion control without end host compliance, a switch can dedicate a switch queue to each user and service each users queue fairly. While ensuring perfect congestion control, dedicating a queue to each user is unimplementable as the system scales. A network could have thousands of tenants, which would make switch architecture impossible. Stochastic Fair Queueing [78] solves this problem by hashing flows to different queues. However, performance degrades when the number of users far exceeds the number of queues. Core-Stateless Fair Queueing (CSFQ) [102] drops packets of flows that use too much bandwidth and approximates fair queueing without per-flow state or multiple queues. Approximate Fair Queueing (AFQ) [98] also drops packets of bandwidth hungry flows using commodity switches with P4 programmability. CSFQ and AFQ both achieve fair bandwidth sharing but end hosts must still handle packet reordering and detect drops, which can harm throughput.

Switch-based protocols are effective in any network. However, since they require specific switch programmability, they are often reserved for local area networks.

As stated before, sender-side protocols are the most practical. Sender-side protocols probe for bandwidth and mitigate congestion using AIMD algorithms. We now provide more details on how AIMD algorithms work.

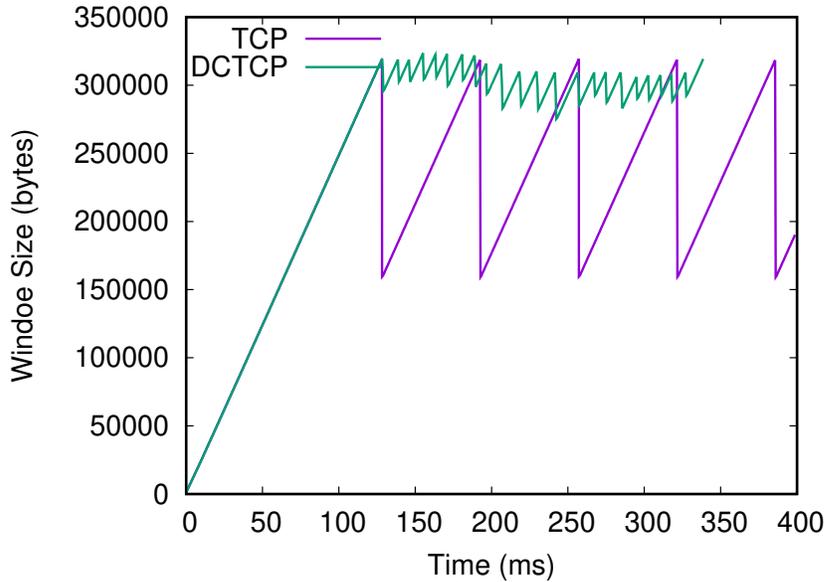


FIGURE 2.3: Saw-tooth Behavior of TCP and DCTCP

2.3 Additive-Increase Multiplicative-Decrease

Many sender-side congestion control protocols use the AIMD paradigm [110, 109, 75, 66] because it converges to max-min fair, uses available bandwidth, and clears congestion [19]. If there is congestion in an AIMD protocol, all the flows multiply their injection rate by a number < 1 , which reduces the flows injection rate or congestion window, which alleviates congestion. In the absence of congestion, each flow increases their injection rate by a constant amount, which grabs available bandwidth. AIMD protocols converge to fair rates by creating congestion through additive increase, and then multiplicative decrease forces flows with more bandwidth to decrease their rate by more than flows with less bandwidth. This creates a saw-tooth behavior, which Figure 2.3 demonstrates

TCP is an AIMD protocol and limits the injection of packets with a congestion window. A larger congestion window indicates more bandwidth. To show how AIMD protocols work, we run a simulation in an ns-3 simulator provided as an artifact by Li et al. [1], which we modify slightly to support traditional TCP. The simulated

network is loss-less, and TCP reduces its window by half if any packet in a congestion window has its ECN bits marked, so end hosts know the packet experienced congestion. Switches mark packets ECN if the packet queue exceeds 40KB, which is 40 packets. The network has 100Gbps links between 17 hosts on a single switch and the delay along each link is 100us. 16 hosts all send 200MB of data simultaneously to the 17th host, which produces an incast traffic pattern. Historically, congestion control protocols handle incast poorly [107]. To simplify the graph, we plot the congestion window over time for only one flow since the behavior of each flow is roughly the same. The maximum window size is BDP, which is about 5MB.

TCP slowly increases its window because the flows start conservatively¹. Eventually, the network saturates and congestion occurs, so TCP cuts its window in half. TCP continues this process. Several times throughout the experiment, TCP increases its window to grab available bandwidth and then alleviates congestion by reducing its window. However, because TCP reduces the rate of flows so aggressively, the network becomes under-utilized after multiplicative decreases and performance suffers.

To demonstrate this inefficiency, we compare TCP with DCTCP, which adjusts the multiplicative decrease based on the extent of congestion [4]. Instead of cutting the congestion window in half if there is any congestion, DCTCP measures congestion by dividing the number of ECN marked packets by the total number of packets sent in a congestion window, which improves stability because the window decreases less aggressively if congestion is minimal. The smaller saw-teeth for DCTCP in Figure 2.3 show the reduced multiplicative decrease. Because TCP underutilizes the network after multiplicative decrease, flows in our experiment using TCP complete sending their data 17% slower than when using DCTCP.

A higher ECN marking threshold increases throughput of TCP because there is

¹ We do not use slow start because we only want to see the saw-tooth behavior

a longer queue built up when flows back off, and the queue can keep the network saturated while flow rates recover. However, this harms network latency because the network maintains long queues. DCTCP’s improved stability allows DCTCP to lower the average queue depth, without losing throughput. RED also improves TCPs throughput, but Alizadeh et al. show that RED creates large queue oscillations, which harms application tail performance [3].

The improvements DCTCP makes over TCP demonstrates how new congestion control protocols can expand a network’s pareto frontier for performance. DCTCP improves throughput without increasing latency. Newer protocols, like HPCC [75] and Swift [66], expand on DCTCP’s performance benefits for datacenter networks. In Chapter 3, we go into further detail about HPCC [75] and Swift [66] and then improve sender-side protocol performance in Chapter 4.

2.4 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) removes system software from the critical path of network operations. Traditionally, computers communicate through OS level libraries like Sockets. RDMA allows applications to communicate without invoking the system software on the data path by offloading the network protocol stack processing, congestion control, and packet retransmission to the network interface card (NIC). Today, many technology companies such as Alibaba [75] and Microsoft [41] deploy RDMA networks in their datacenters, and exploring how to use RDMA in application design has become a major topic in the networking community [58, 59, 57, 40, 53].

The most common RDMA open standard is Infiniband (IB) and its extension RoCE². IB is a lossless network, and credit-based flow control ensures IB switches do not drop packets. RoCE enables IB in an ethernet network. However, ethernet does

² Pronounced Rocky, like the mountains

not have credit-based flow control, so RoCE uses Priority flow control, which stops upstream ports from sending packets to a congested port. Because IB and RoCE are lossless, they can suffer from tree saturation and head-of-line blocking, which can severely limit network throughput and harm packet latency. IB and RoCE avoid tree saturation and HoL blocking with congestion control.

IB uses sender-side congestion control. The IB specification lays out specific details about congestion detection and reaction. While Zhu et al. [110] designed DCQCN to fit into the specific RoCE constraints, more recent IB and RoCE congestion control protocols eschewed the requirements for better performance [75, 81]. One requirement for IB/RoCE congestion control that remains popular is starting flows at line rate. TCP starts sending packets conservatively and slowly increases its window over time. However, IB assumes the network is not congested when starting a flow and sends packets as quickly as possible. This is a performance optimization because it allows short flows to send all packets quickly. IB and RoCE also do not specify the use of a window, and instead pace packets using inter-packet delay, so the NIC delays sending a packet for a certain amount of cycles. The inter-packet delay changes based on network congestion. HPCC adds a congestion window in addition to pacing packets [75].

This dissertation simulates a RoCE network unless otherwise specified. We use this as our sample network because it is widely used and therefore topical. While we only evaluate our mechanisms in a RoCE network, many of the lessons apply to lossless networks that start sending packets at line rate.

2.5 Congestion Control Evaluation

We evaluate our congestion control mechanisms in a simulator and optimize for key application level metrics. This section explains how and what we simulate, as well as the metrics that we evaluate the protocols and mechanisms on.

2.5.1 Simulations

Simulators allow fast development of new network protocols. We design our congestion control protocols and mechanisms for networks that implement congestion control in hardware. This dissertation uses two simulators to simulate 1) a RoCE network and 2) an NVLink network. To simulate a RoCE network, we use an ns-3 [95] simulator released as an artifact by Li et al. [75], which models RoCE networks, so we refer to this simulator as the RoCE simulator. The second simulator is based on booksim [50] and simulates an NVLink network. We use a second simulator because it is the most accurate model of an NVLink network and implements many hardware features that our new protocol uses. Because we use the NVLink simulator exclusively in Chapter 6, we provide more details about NVLink and the simulated network in that chapter. We provide more details about simulations in subsequent chapters.

2.5.2 Congestion Control Metrics

The three classic network performance metrics are throughput, latency, and fairness. Throughput measures how many packets get through the network over a period of time, and latency evaluates how quickly the packets move through the network. Different applications optimize for different metrics. For example, a web-server workload constructs webpages for users as quickly as possible. Web-server workloads generally move small pieces of data, but the data must move quickly, so latency is an important metric. Other workloads, such as big-data and deep learning, require moving vast amounts of data quickly between servers, so the application requires high throughput. To measure bandwidth fairness, we use the Jain fairness index [19]. The Jain fairness index assumes all flows share the same bottleneck and have equal demands, so we only use it in microbenchmarks because in larger systems, flows are bottlenecked at different points in the network. If the bandwidth allocation is fair, the

Jain fairness index equals 1, and if the bandwidth allocation is completely unfair, the index is $\frac{1}{N}$, where N is the number of flows.

A common metric that unites both latency and throughput for application level performance is Flow Completion Time (FCT), which measures how long a flow takes to complete. Generally, this measurement starts when a flow sends its first packet and concludes when the last packet acknowledgement confirms the network delivered all the data to the destination. Latency dictates if a small flow has a low FCT and high throughput lowers long flow FCT. An extension of FCT is FCT slowdown, where we divide the observed FCT by the theoretical minimum FCT of the flow based on the network characteristics. FCT slowdown tells us how congestion and contention limited the performance of a flow. Congestion increases the FCT of short flows because the switches queued the packets for a long period of time. Contention increases the FCT of long flows because the long flows require bandwidth, and the network shared the finite amount of bandwidth between many competing flows.

After running or simulating a workload, we create a distribution of FCTs from the numerous flows that complete. Median FCT slowdown is a useful statistic for determining the overall network performance. However, the median case is not useful to many applications because applications require predictable performance. Therefore, many congestion control protocols design for the tail FCT, so performance is sufficient even in the worst case. Tail FCT also informs about fairness. If the network unfairly allocates one flow too little bandwidth, the tail FCT increases. By improving fairness, we dramatically improved the tail FCT of long flows in Chapter 4. In this dissertation, we often report the median and 99.9% tail FCTs for flows when evaluating our protocols or mechanisms.

2.6 Summary

This chapter gives an overview of congestion control. First, we explain what congestion is and why it harms network performance. We then discuss where we can implement congestion control and the pros and cons of putting congestion control in various points in the network. We then give a description of RDMA networks and why we simulate them. We provide a more in-depth detail about AIMD congestion control algorithms and how and why they work. We conclude by explaining how this dissertation evaluates congestion control protocols.

Performance and Isolation Issues in RDMA Networks

A fair congestion control protocol ensures that the network treats all flows fairly. If all bandwidth allocations are fair, all flows perform equally well. However, if the protocol takes too long to converge to the fair rates, performance and isolation issues can occur. If the network allocates some flows too little bandwidth for too long, the flow's performance suffers. Further, if users can identify when protocols unfairly allocate flows too much bandwidth, users can abuse the unfairness to gain extra bandwidth. This section describes why congestion control protocols converge slowly and measures how slow convergence affects performance and performance isolation.

Slow convergence to fair rates causes a long tail for large flow FCT. Because the bandwidth allocation becomes fair slowly, the network sometimes allocates bandwidth bound flows (long) too little bandwidth. This has little effect on overall system throughput, and most flows perform well, but the small number of flows allocated too little bandwidth have long FCTs, which negatively impacts applications that rely on tail latency [25]. First, we identify why popular protocols converge slowly and demonstrate with microbenchmarks how the slow convergence affects FCTs. We

then show how slow convergence greatly increases the FCT of long flows in datacenter simulations.

Slow convergence also leaves a network susceptible to performance isolation attacks. Some protocols, particularly those for RDMA networks [110, 75] start sending flows at line rate and only reduce rates when end hosts detect congestion. Starting flows at line rate is a performance optimization that improves small flow completion time and throughput. However, we find starting at line rate leaves the network susceptible to performance isolation attacks because users can exploit the networks slow convergence to unfairly increase their packet injection rate. Using the attacks, a misbehaving user unfairly receives too much bandwidth and causes severe congestion in the network. We also show that stopping some of the performance isolation attacks increases the FCT of small flows, and we discuss the tradeoff. We demonstrate the attacks in a hardware testbed and then show the attacks degrade system performance in datacenter benchmarks.

3.1 Long Flow Tail Latency

Congestion control protocols manage packet queues on switches, which enables low packet latency and high throughput in RDMA networks. Numerous congestion control algorithms exist for RDMA networks that achieve these two goals [110, 81, 66, 75, 83]. Using these protocols, flows often complete quickly and applications perform well.

While good network performance requires low latency and high throughput, we identify a third metric that influences FCT, which is largely ignored by previous work: *convergence to fairness*. Most protocols are provably fair [109, 75, 66], but few optimize to converge quickly to fair rates. Faster convergence can dramatically impact the FCT of long flows that are bound by their bandwidth allocation. Anytime a protocol allocates bandwidth unfairly, a flow’s performance suffers. This chapter

identifies why some protocols converge slowly and demonstrates how convergence impacts performance.

3.2 Background

We provide details about Swift [66] and HPCC [75], the two congestion control protocols we use for our case study. Swift and HPCC have many similarities. Both ensure high throughput and low latency but converge slowly with reasonable parameters. Both also only perform a multiplicative decrease at most once per-RTT and converge to fair rates with AIMD.

HPCC and Swift also have many differences. Li et al. [75] and Kumar et al. [66] designed their protocols for different environments and different network state information. Li et al. had fairly advanced switches with INT, which gives them fine-grained data about network state. Kumar et al. assume a network with extremely simple switches that provide no feedback about network state. Swift assumes the network cannot even mark ECN bits in the packet header. In this section, we provide more details about how the protocols work.

3.2.1 HPCC

Alibaba developed HPCC [75] for their datacenters. Alibaba originally used DCQCN [110] but noticed significant performance issues when running a storage workload alongside a machine-learning workload. Alibaba improved application isolation by creating HPCC.

HPCC bases rate adjustments on 3 pieces of telemetry data from INT, 1) queue depth 2) tx bytes and 3) timestamps. Queue depth informs HPCC about congestion, and HPCC calculates throughput with tx bytes and timestamps. Most congestion control protocols are AIMD, but HPCC is both Multiplicative-Increase Multiplicative-Decrease (MIMD) and AIMD. HPCC converges to pareto optimal us-

ing MIMD and ensures fairness with AIMD.

Additive-increase slowly grabs available bandwidth in many congestion control protocols, but HPCC can be more aggressive because HPCC knows the amount of available bandwidth. Therefore, each flow can multiply their injection rate by the percentage of available bandwidth. This grabs all available bandwidth in one RTT but is not fair. To ensure fairness, HPCC also uses additive increase, which increases rates in addition to the multiplicative increase. This gives HPCC higher bandwidth utilization and AIMD’s fairness benefits.

Like many protocols, HPCC decrease flow’s rates when it detects congestion. HPCC bases the multiplicative decrease on the queue depth information from INT. If a queue is large, HPCC decreases injection rates by more than if the queue is small. HPCC also uses a congestion window, which limits the number of packets in the network, and packet pacing, which adds delay between each packet’s injection into the network.

3.2.2 *Swift*

Unlike Alibaba, Google’s congestion control must rely only on end-to-end metrics, and protocols cannot rely on ECN, RED, P4, or INT for network state. Swift measures network congestion with packet RTT; if congestion increases, RTTs increase, and Swift reduces the congestion window size. Swift reduces the congestion window when the observed packet latency exceeds *target delay*. *Target delay* is the maximum amount of tolerable delay in the network. A user can tune *target delay*, but it also changes dynamically, which we further explore later. Swift adjusts the congestion window (cwnd) with the following equation:

$$\text{cwnd} = \max(1 - \beta * (\text{delay} - \text{target_delay}, 1 - \text{max_mdf}) * \text{cwnd} \quad (3.1)$$

where β and *max_mdf* (maximum multiplicative decrease factor) are adjustable

parameters that make multiplicative decrease more or less severe. Delay is how long the acknowledgement took to arrive after the end host sent the data packet. Swift increases its congestion window slightly each time a packet acknowledgement arrives and the packet delay is below *target delay*.

Three components make up *target delay*: 1) base delay 2) topology-based scaling and 3) flow-based scaling. Base delay determines the minimum tolerable delay in the network, and *target delay* never drops below base delay. Topology-based scaling adjusts *target delay* based on a flow's path. In a datacenter, the length of different paths can vary wildly, particularly as the datacenter becomes large. If two servers are far apart, topology-based scaling increases *target delay*. For example, if two servers in the same rack communicate, *target delay* is lower than two servers with 10 switches between them.

$$fbs = \max(0, \min(\frac{\alpha}{\sqrt{cwnd}} + \beta)) \quad (3.2)$$

$$\alpha = \frac{fs_range}{\frac{1}{\sqrt{fs_min_cwnd}} - \frac{1}{\sqrt{fs_max_cwnd}}}, \beta = -\frac{\alpha}{\sqrt{fs_max_cwnd}} \quad (3.3)$$

The third and most complicated component, flow-based scaling, adjusts *target delay* based on window size. If a flow's congestion window is small, flow-based scaling increases *target delay*. This has two effects 1) if a flow has a small window, it is less likely the flow reduces its rate because it has more delay tolerance and 2) if there is a multiplicative decrease, it is less severe because the difference between delay and *target delay* becomes smaller, and so does the multiplicative decrease factor. Equations 3.2 and 3.3 layout the exact equations for FBS. FBS changes *target delay* on the range *fs_min_cwnd* to *fs_max_cwnd* and varies from 0 to *fs_range*. Flow-based scaling improves fairness because a flow with a smaller congestion window reduces its

rate less than a flow with a larger congestion window. Kumar et al. found flow-based scaling improves fairness particularly well when each flow’s congestion window is less than 1 packet.

This section provided more details about HPCC and Swift, which we analyse in this chapter. We now detail the exact reasons they converge slowly and measure the slow convergence.

3.3 Sources of Unfairness

State-of-the-art datacenter congestion control techniques minimize latency (queueing delay) and maximize throughput. However, in efforts to achieve these goals, several design decisions eschew fast convergence to fairness in favor of low latency and high throughput. We outline these decisions and how they lead to slow convergence.

3.3.1 *Conservative Additive Increase*

AIMD converges to fairness by generating congestion events that trigger multiplicative decrease [31, 19]. In the absence of congestion, all flows increase their rates. The sum of all flow rates eventually exceeds the bottleneck link’s bandwidth because each flow decreases its rate. Multiplicative decrease reduces a flow’s rate by a multiple of the rate, therefore the larger a flow’s rate, the more its send rate decreases. This converges to fairness *eventually* because the flows with more bandwidth reduce their rate by more than the flows with less bandwidth, and all flows increase their rates by a constant amount.

AIMD relies on introducing queueing delays to converge to fair rates. To minimize queueing delay while still converging to fair rates, protocols set additive increase parameters conservatively to introduce only modest congestion. Protocols also scale the multiplicative decrease factor with the extent of congestion[4, 75, 66]. If con-

gestion is modest, so are rate reductions. This enables provably fair protocols while ensuring high throughput and low latency. However, small multiplicative decreases and small additive increases make the protocol converge to fair rates more slowly [19]. If the network allocates a bandwidth bound flow too little bandwidth, its FCT increases, which harms application performance.

***Insight:** Using a conservative AI value favors low latency over fairness.*

3.3.2 One Reaction Per RTT

HPCC and Swift fully react to at most one congestion signal per RTT [75, 66]. This prevents the protocols from reacting twice to the same observed congestion. Consider a flow that sends 10 packets in an RTT, and each packet reports a queue of about 100KB on the same switch. The queue depth or queueing delay each packet reports is likely the same congestion event. If the protocol reacted to each packet, it would react to the same congestion event (a queue of about 100KB) multiple times.

Reacting only once per-RTT removes a natural fairness effect. When reacting to every acknowledgement, the flows that have more acknowledgements are those with a larger congestion window and therefore a higher rate, and flows with higher rates react more often than those with lower rates. As an example, suppose a congestion control algorithm uses ECN as its congestion signal, and there are two flows, and one uses twice as much bandwidth as the other. The flow with twice as much bandwidth likely receives twice as many ECN marked packets in an RTT. If the protocol only reacts once per RTT and divides the rate by two if the switch marked any packet's ECN bits in the RTT, both flows divide their rates by half. However, if the protocol reacted to every ECN marked packet instead of every RTT, then the flow with twice as much bandwidth and twice as many ECN marked packets would decrease its rate twice as many times as the flow with less bandwidth. When reacting once per RTT, less progress is made toward achieving a fair allocation of bandwidth.

***Insight:** Not reacting to every congestion signal eschews convergence to fairness for the sake of throughput.*

3.3.3 Deterministic Feedback

Probabilistic feedback is more fair than deterministic feedback. To improve fairness in DCQCN, Zhu, et al. [110] suggest the maximum probability a switch marks a packet as experiencing congestion under moderate congestion is 1%. Under these conditions, a flow without many packets in the queue (low rate) has a low probability of a packet getting marked even when the link is congested. Therefore end hosts sending more packets are more likely to reduce their injection rate of packets. While this leads to better fairness, Gao, et al. [34] showed that infrequent congestion signals can lead to poor performance during Incast traffic. We call this probabilistic feedback.

In contrast, INT and RTT use deterministic feedback because no matter how many packets the flow has in the queue, it receives generally the same feedback as a flow with many packets, since both flows experienced the same queueing delay (RTT) or have nearly the same queue depth upon egress from the switch (INT). Because all flows see almost exactly the same congestion, the competing flows react the same even if they have different bandwidth allocations. Meanwhile when using probabilistic feedback, a flow with a higher rate is more likely to receive feedback that indicates congestion. Swift’s flow-based scaling [66] addresses this, but it is insufficient to improve tail FCT of long flows, which we demonstrate with experiments .

***Insight:** All flows receiving the same feedback leads to unfairness.*

3.3.4 Methodology

We demonstrate unfairness in an unmodified version of HPCC and a slightly modified version of Swift. We use ns-3 [95] code provided as an artifact from HPCC [1]

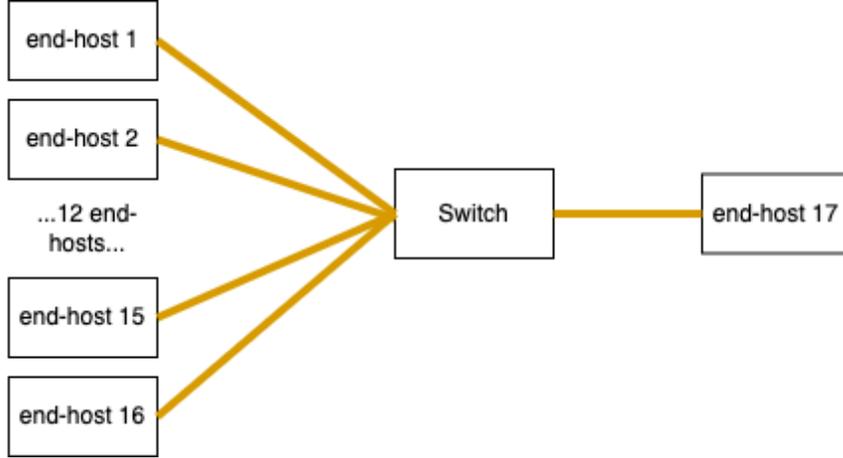


FIGURE 3.1: Microbenchmark Topology

to evaluate HPCC and Swift. The simulator already implemented HPCC, and we implement Swift to the best of our ability. We run a 16-1 incast traffic pattern, which is important to datacenter networks [107]. Incast occurs when multiple end hosts send to a single destination simultaneously. To evaluate each protocol’s handling of incast traffic, we use a single switch topology with 17 hosts and each host has a 100Gbps link to the switch, and 16 of the hosts have one flow to the 17th host. Figure 3.1 shows the topology. Two flows start every 20 microseconds and each flow sends 1MB. Each link’s propagation delay is $1\mu s$. Staggering each flow’s start demonstrates how new network unfairly allocates new flows too much bandwidth.

Table 3.1 provides a concise summary of the experiment’s parameter settings. We use the parameters suggested by Li et al. [75] when evaluating HPCC. We set Additive Increase to 50 Mb/s, utilization rate to 0.95, and increment stage to 5. We refer to this set of parameters as ”default HPCC”. In the figures, these settings are labelled as ”HPCC”. We also run HPCC with a higher AI value of 1Gbps to demonstrate the effect of AI on fairness. This variant is labelled as ”HPCC 1Gbps”. Additionally, we modified HPCC to use randomized feedback, where feedback from the network is sometimes ignored. Feedback is disregarded based on the following

Table 3.1: HPCC and Swift Parameter Settings

HPCC	AI	50Mb/s
	Utilization Rate	.95
	Increment Stage	5
Swift	AI.	50Mb/s.
	β	.8
	max_mdf	.5
	base delay	$5\mu\text{s}$
	per-hop delay scaling	$2\mu\text{s}$
	fs_range	5 * base_delay
	fs_max_win	50 (100 in DC sims)
	fs_min_win	.1

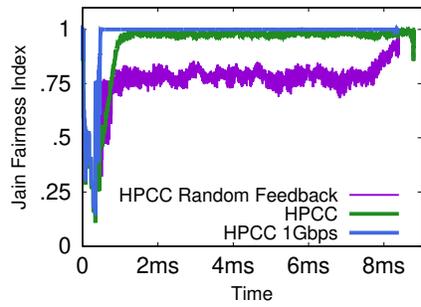
equation:

$$\text{Current Window} < (\text{rand()} \% \text{Max Window})$$

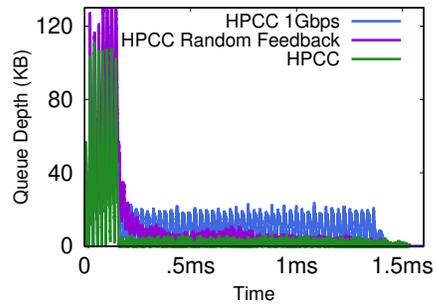
This creates a linear equation for the likelihood a packet is dropped as a function of current window size¹. If a window size is at its max size, the information is always used. If the window is 0, the feedback is never used. If the current window is half the maximum size, there is a 50% chance the packet feedback is used. This process only occurs if there is a multiplicative decrease and the reaction would update the reference rate, so it does not affect rate increases. This HPCC variant is labelled "HPCC Random Feedback".

Swift does not suggest settings for multiple parameters, so we set reasonable parameters. Like HPCC, we set additive increase to 50Mb/s. We set β (see Equation 3.1) to 0.8 and the maximum mdf to 0.5. For FBS, we use the parameters in Kumar et al. [66], except when we are running on the smaller topology because the window is smaller, so we lower the *fs_max_win* from 100 to 50 packets. For topology-based scaling, we set the base RTT to 5us, and add $2\mu\text{s}$ of delay per hop. We also run Swift with randomized feedback and a 1Gbps AI to show the effect on fairness. Because

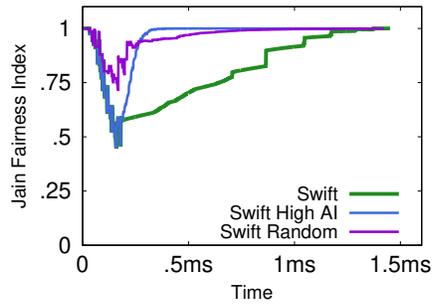
¹ Current Window size refers to the window based on the per-hop, per-RTT rate not the per-ACK rate.



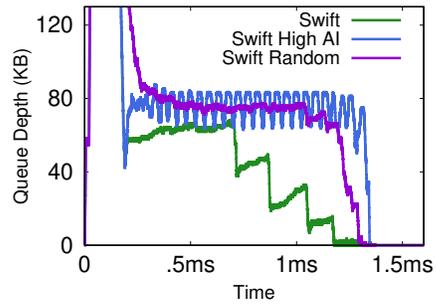
(a) 16-1 Incast Jain Index in HPCC



(b) 16-1 Incast Queue Depth in HPCC



(c) 16-1 Incast Jain Index in Swift



(d) 16-1 Incast Queue Depth in Swift

FIGURE 3.5: Jain Fairness Index and Queue depth during Incast Traffic in HPCC and Swift

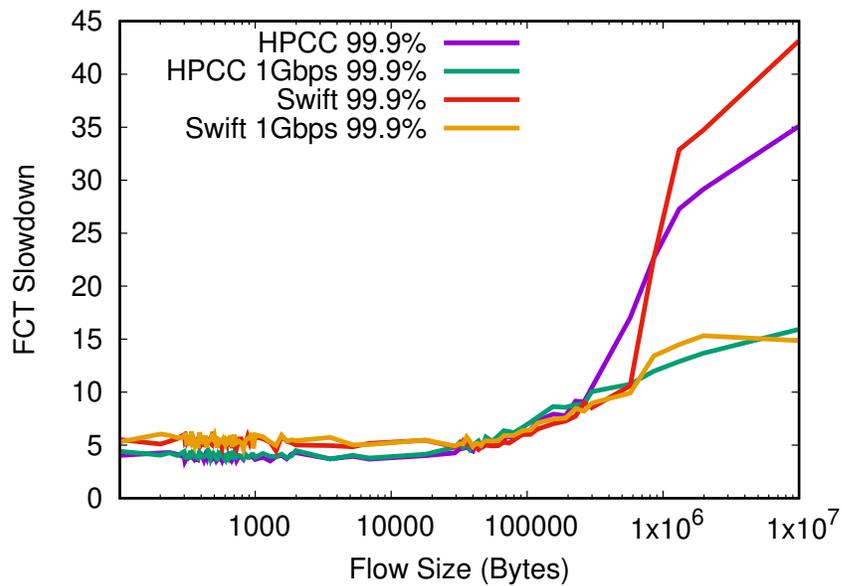


FIGURE 3.6: Tail Latency 99.9% for Swift and HPCC

to HPCC.

3.3.5 Unfairness in HPCC and Swift

HPCC and Swift exhibit the previously discussed characteristics that create unfairness. Figures 3.3 and 3.4 plot the start time versus the finish time of flows in the 16-1 incast in HPCC and Swift respectively. The trend for both default Swift and HPCC is the same. Flows that begin last finish first because existing flows reduce their rates several more times than the most recent flows to start. Increasing AI and randomizing the feedback eliminates this trend and the flows finish at generally the same time.

Figures 3.5(a) and 3.5(c) plot the Jain fairness index over time [47]. When the Jain fairness index is closer to one, the rates are more fair. Using the default parameter settings, both Swift and HPCC take several hundred microseconds to get close to an index of one. Using randomization and a higher AI improves convergence to fair rates in both protocols. However, this has negative effects. Figures 3.5(b) and 3.5(d) show both congestion control protocols have larger queue oscillations with a higher additive increase.

Figure 3.6 shows the 99.9% of FCTs based on flow size in the datacenter simulations. With the default AI, the FCT slowdown is 35-45x, but with a larger AI, the slowdown drops to between 10-15x. Any application that relies on long flow completion time would benefit from this increased performance.

We show that increasing AI improves performance of long flows but has negative consequences during incast traffic. In Chapter 4, we introduce mechanisms that improve long flow tail FCTs while maintaining small queues during incast traffic. In the next section, we explain how slow convergence also creates security issues in RDMA networks.

3.4 Performance Isolation Attacks

Datacenter applications demand high throughput, low latency, and low CPU overheads. This leads to the growing adoption of Remote Direct Memory Access (RDMA). The natural next step is thus to bring the benefits of RDMA to cloud users, and this requires designing for multiple mutually untrusted users sharing an RDMA network. Researchers identified security issues with RDMA [96, 100], but the CC aspect has received little attention. TCP/UDP networks cannot assume that a user follows the CC protocol, but since RDMA NICs (RNICs) enforce CC in hardware, which is outside the bounds of user control, RDMA users must follow the CC protocol. In shared environments, we must consider *can malicious users gain more bandwidth than their fair share by exploiting the current design of RDMA CC mechanisms?*

There are several production variants of RDMA [44, 9, 89], so answering this question broadly is impossible. However, the most common standards are Infiniband (IB) and RoCE. RoCE is an appendix on the Infiniband specification and implements the IB protocol layer in an Ethernet network. We use IB and RoCE CC as a proxy for RDMA networks.

Malicious users trying to gain more bandwidth is not new in a traditional kernel-based datacenter networking setting. For example, users can open multiple TCP sockets to gain extra bandwidth [91]. Furthermore, a user can simply use UDP to avoid CC. To disincentivize CC avoidance, researchers developed mechanisms that drop users' packets via fair-queuing-based approaches [98, 102] if end hosts use too much bandwidth.

RDMA networks differ from TCP networks in two unique ways that render existing solutions useless. First, applications offload communication to an RNIC, and thus we cannot have any software-based indirection on the data path. The second and more important property is that RDMA requires a lossless network, so we can

no longer drop packets actively in the network as in the existing fair-queuing-based approaches.

We analyze two major RDMA CC mechanisms, DCQCN [110] and HPCC [75]. DCQCN is the default CC algorithm in RoCE NICs from NVIDIA Networking, a major RDMA NIC vendor worldwide. Alibaba deploys HPCC and is currently in the process of being standardized by the Internet Engineering Task Force (IETF). We uncover several performance attacks that allow a user to take substantially more bandwidth than is fair. The attacks are feasible because of how IB/RoCE allocates bandwidth and flows take several RTTs to converge to their fair rates. The attacks include creating more than one queue pair (parallel QP attack), sending data through a set of queue pairs in a round-robin fashion to completely circumvent CC (staggered QP attack), and using multiple overlay topologies for collective communication (shuffled overlay attack). The key property that our attacks exploit is that RDMA CC algorithms fundamentally favor short flows, i.e., CC is enforced on a per-connection basis and each queue pair starts at line rate. Users can gain more bandwidth by abusing flows start sending at line rate and protocols converging over several RTTs.

These attacks allow a malicious user to harm the network performance of other well-behaved users. In our testbed experiments, an attacker can obtain 72% of the available bandwidth with a victim flow on a RoCE NIC using the DCQCN CC algorithm. Furthermore, ignoring CC causes switch buffers to fill with packets. This dramatically extends packet queuing delay. In a simulated RoCE datacenter using DCQCN, the 99% tail of small flow completion times increases by 7 times when a user attacks the network. Since RDMA networks are lossless and can suffer from tree saturation [90], these attacks could theoretically render an entire network unusable as congestion spreads through the network.

In addition to identifying the above attacks, we uncover a fundamental tradeoff between short flow completion times and the ability to mitigate these attacks. RDMA

CC protocols start sending packets at line rate for several reasons, but primarily to allow flows smaller than the network Bandwidth Delay Product (BDP) to send all packets in one RTT. However, when a protocol allows a short flow to start at line rate, a user could exploit slow convergence and imitate short flow behavior by breaking a long flow into several short flows and continuously send packets at line rate. Therefore, anytime a CC protocol provides exceptions for a certain flow type, it creates a vulnerability. CC is an attack vector in RDMA networks, and performance isolation must be considered when designing and evaluating CC algorithms.

3.4.1 IB and Isolation Background

Remote Direct Memory Access allows users to directly interface with hardware resources on an RDMA NIC (RNIC). RoCE and IB are the most popular RDMA standards and both follow the IB specification. In RoCE, users send messages along Queue Pairs (QPs), which are similar to sockets in TCP. There are many types of QPs, but we only consider the Reliable Connection (RC) QP type in our experiments because it is the only QP type that enables all operation types² and only has one destination, which works well with CC.

When implementing RoCE/IB in hardware, there are three choices to enforce CC. First, CC is optional in IB/RoCE, so hardware is not required to support it. The second option is for the hardware to enforce CC per-QP, and the third option is to enforce per-service level (per-SL) [45]. A service level is analogous to a priority and if one QP on a given SL experiences congestion, all QPs on that service level throttle their rates. Per-SL makes sense in certain topologies, like a ring, where all QPs may share a common path. However, in high radix topologies like a Clos [21] or a fat-tree [70, 2], per-QP CC is more intuitive because not all QPs on the same

² The Dynamically Connected Transport available in Nvidia Networking NICs supports all types of operations, but it is not part of the IB standard.

SL are throttled when one QP experiences congestion.

Prior work shows the dangers of per-flow fairness [91, 99]. FairCloud [91] explored this space extensively. They proposed several different methods of enforcing rate-limiting to ensure that users could not simply open more connections to increase their bandwidth allocations. However, only one of our proposed attacks involves using multiple QPs simultaneously to gain an advantage; the others exploit QPs starting at line rate and only require one QP to send packets at a time. Further, two of the proposed methods in FairCloud require fair queueing on switches [91]. This either requires a virtual lane per-tenant, which is unimplementable, or approximating fair queueing with mechanisms like Core-Stateless Fair Queueing (CSFQ)[102] or Approximate Fair Queueing (AFQ) [98]. However, AFQ and CSFQ require dropping packets, so they are not suitable for RoCE/IB. Seawall [99] enforces per-src CC, which eliminates the improvements of opening multiple connections but causes asymmetric bandwidth allocation.

3.5 RDMA Congestion Control Attacks

We introduce three performance attacks that work against the current IB/RoCE specification. The first attack involves opening and sending data on several QPs simultaneously. The second attack also sends data on several QPs but does so sequentially, continuously changing which QP sends data. This allows a user to ignore CC. The final attack involves changing between multiple equal-cost communication overlays. By constantly changing the source-destination pairs for communication, an application can again ignore CC completely. All attacks cause congestion and allow a malicious user to gain extra bandwidth. These issues exist due to 1) how connections are defined in the IB/RoCE specification and 2) slow convergence. If flows converged to their fair rates instantly and used a simple fairness policy, the attacks would be useless.

3.5.1 *Parallel QP Attack*

The Parallel QP attack requires a user to open several QPs to the same destination instead of a single QP per destination. Because IB/RoCE enforces CC on a per-QP basis, the share of a bottleneck link is distributed based on the number of QPs each host sends data along. The parallel QP attack is analogous to opening multiple TCP sockets [29]. However, since the RDMA network is lossless, switches cannot drop packets of misbehaving users, which is a solution to the issue in lossy networks [102, 98]. Simply using per-src/dst fairness instead of per-QP fairness solves this issue.

3.5.2 *Staggered QP Attack*

Using the staggered attack, a user ignores CC completely. Unlike TCP sockets, each new QP initially sends packets at line rate. If a user continuously sends packets on new QPs, the NIC never triggers CC. An RNIC waits for at least one RTT before it receives feedback from the network to reduce a QP's rate. This is because destinations generate negative feedback, either in the form of Backward Explicit Congestion Notifications (BECNs) or Congestion Notification Packets (CNPs). The packet arriving at the destination and then the congestion information arriving back at the sender takes at least one RTT. Assuming that the QP does not compete for RNIC resources, a QP can send at least BDP packets before it throttles its rate due to congestion.

Several factors affect the utility of a staggered attack. In a 3-tier fat tree with 100Gbps links and 1us of propagation delay, the expected RTT of the network is 12us. This means a QP sending at line rate can send 153KB before it throttles its rate. As network characteristics change, such as longer delays or more hops, a QP can send more packets. Another concern is the size of the message. If the message size is 100MB and the BDP is 153KB, a user would need 650+ QPs to the same destination.

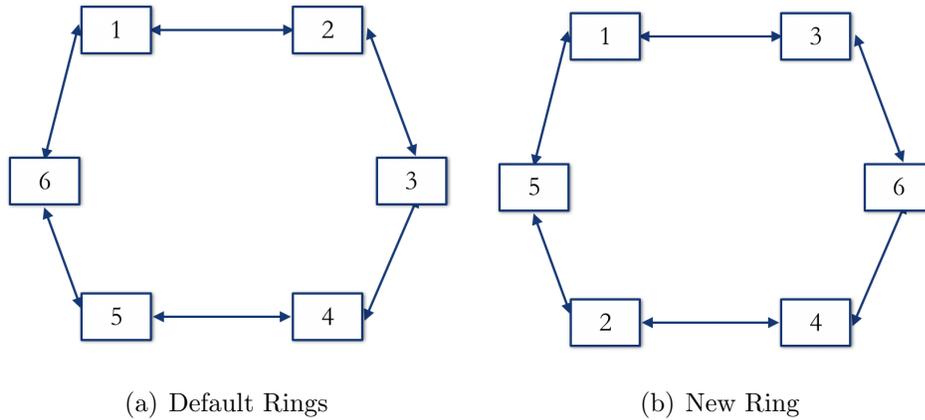


FIGURE 3.7: Changing Ring to create more src/dst pairs.

While this is not impossible, using too many QPs can cause performance degradation due to cache misses of QPs' metadata [27, 58]. There may be several ways to reduce the number of QPs used. First, rates in IB/RoCE recover over time, so when a QP is left idle while other QPs send their data, the QPs rate eventually increases back to line rate. DCQCN has parameters that determine how quickly QP rates recover over time. Second, a user could destroy old QPs and set up new QPs while other QPs send data.

This attack is unique to RDMA networks. TCP starts a connection by only sending a single packet and slowly increasing the window over time, so staggering connections to have a new connection always starting harms performance. RDMA network endpoints aggressively inject traffic and only reduce their injection rate if they detect congestion. By continuously starting new connections, a user can send at line rate regardless of congestion.

3.5.3 Shuffled Overlay Attack

The shuffled overlay attack exploits common communication patterns to mitigate the effect of CC. For example, distributed data-parallel deep learning requires an all-reduce, which is often implemented with a ring communication pattern to maximize

bandwidth utilization. Avoiding all-to-all communication allows a user to shuffle communication overlays and thus circumvent CC.

Figure 3.7 shows several unique rings a user can create with just six servers. In these rings, each server sends to a new destination, either by reversing the original communication direction, changing the overlay ring, or doing both. Figure 3.7(a) displays two rings, each going in a different direction. Figure 3.7(b) changes the overlay, so all the neighbors in the ring are new. As the system scales, there are more opportunities to create new rings. The number of possible overlays is proportional to the number of servers. Assuming all servers are connected in a clique, there are $n-1$ equal-cost overlays if $n > 8$ where n is the number of servers [105].

New servers with multiple RNICs further exacerbate this issue [85]. In this case, a user can create even more source-destination pairs.

Shuffling overlays enable a user to perform the staggered and parallel attacks even if a system enforces per-src/dst CC opposed to a per-QP basis. A user can perform the staggered attack by sending across a new src/dst pair each RTT. This lets the attacker send at line rate and ignore CC. A user can perform the parallel attack by sending across several overlays simultaneously.

3.6 Attack Evaluation

We demonstrate the parallel and staggered attacks in a small cluster testbed and ns-3 simulations. We focus on the parallel and staggered attacks since the shuffled overlay attack is comprised of these primitive attacks. All evaluations take place on a cluster with 6 servers each with a 100Gbps single port ConnectX5 RoCE NIC connected to a 100Gbps Mellanox SN2100 Ethernet switch. There is a single switch, but we emulate a dumbbell topology by connecting the switch to itself and forcing all traffic through that link. Mellanox NICs use DCQCN as their CC algorithm. The NICs and the switch use the vendors default settings unless otherwise specified.

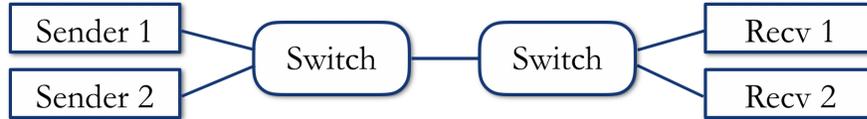


FIGURE 3.8: Experiment Dumbbell Topology

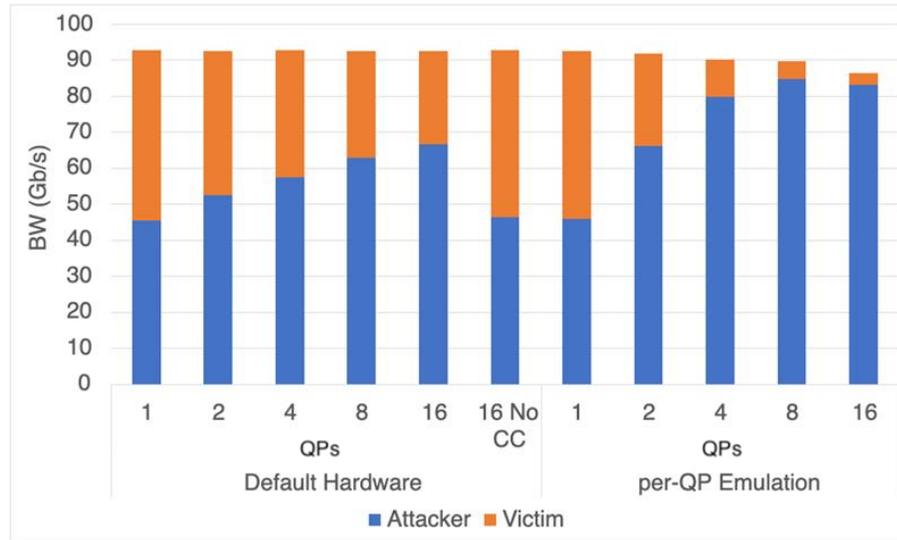


FIGURE 3.9: Parallel QP Attack Testbed Results

We experiment further in ns-3 to show the impact these attacks have in a larger setting. The simulations demonstrate the effects of the attacks in a datacenter setting. We use the same datacenter setup described in Section 3.3.4 and used earlier in this chapter. We also use the same Facebook Hadoop workload [108].

3.6.1 Testbed Experiments

Parallel Experiment: First, we demonstrate the parallel attack on the testbed. We run the *ib_write_bw* test on 4 servers. Figure 3.8 illustrates the experimental topology. Both senders and receivers are on the same side of the dumbbell, so the source-destination pairs share the bottleneck link. On one sender and receiver, we open several extra QPs and run the write bandwidth test. The victim sender and receiver only open one QP. Figure 3.9 shows how the attacker gains more bandwidth

as it opens more QPs in the default hardware configuration. We omit error bars because all standard deviations are within 1% of the mean.

Congestion Control causes this misallocation. The bar labeled "16 No CC" shows the results when we disable CC in our RDMA NICs. Fair arbitration on the switch shares the bandwidth fairly between the two input ports.

Mellanox hardware does not allocate bandwidth on a per-QP basis. If CC enforces per-QP allocations, we expect that with an extra QP the attacker would get two-thirds of the bandwidth. However, the attacker receives far less than that. After further investigation, we discovered that our NICs did not follow the IB specification³ and instead enforce CC per destination. On our NICs, all QPs to the same destination use the same send rate.

QPs on the same RNIC share CC information and send rate but do not split the rate between the QPs. For example, consider two sources sending to a shared destination on a 100Gbps link. Source 1 opens two QPs and source 2 opens 1 QP. Source 1 calculates that it should send to the destination at a rate of 30Gbps. However, instead of splitting this rate between the two QPs, both QPs send at a rate of 30Gbps. Source 1 then sends at 60Gbps to the destination. Source 2 only sends at 40Gbps. Source 1 calculates a lower rate than source 2 (30Gbps vs 40Gbps) because Source 1 receives more negative feedback from the network due to its overall higher rate (60Gbps). This results in neither per-QP fair nor per-src/dst fair.

To trick the hardware into doing per-QP CC, we create multiple IP addresses on the destination and open a new QP on each IP address. This allows us to emulate the IB/RoCE specification and per-QP CC. Figure 3.9 shows that in implementations that adhere to the IB/RoCE spec, the attack can get a far larger percentage of the bottleneck link.

³ Mellanox (now Nvidia) owns a patent on destination based CC [10]. Some QPs (UD and Mellanox's DCT [23]) can send packets to multiple destinations, so per-QP CC can throttle the rate of a QP even if the destination and bottleneck changed.

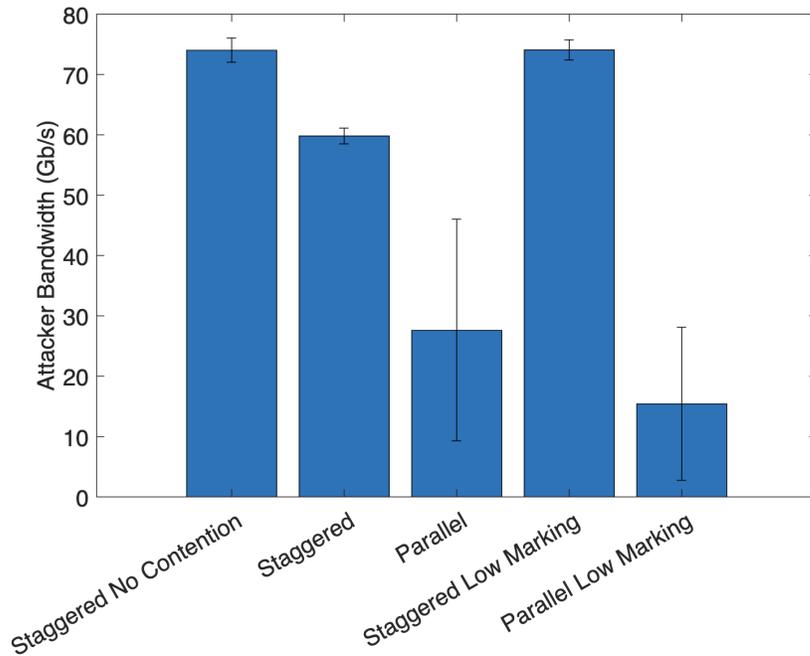


FIGURE 3.10: Parallel and Staggered Attack BW 1.125MB Transfer

Staggered Experiment: Next, we demonstrate the staggered attack’s effectiveness in hardware. We run the staggered attack with and without a competing flow to show the upper bound on performance and also rerun the parallel attack to show the superiority of the staggered attack. We use 30 QPs, and the bandwidth delay product of our testbed’s network is 37.5KB. This enables us to do a 1.125MB transfer for staggered attacks. Figure 3.10 shows the staggered attack results. Running the staggered attack without a competing flow achieves a throughput of about 74Gb/s; the maximum bandwidth we achieve on our 100Gb/s RNICs is 92Gb/s. When we run the staggered attack with a competing flow, the attacker receives 60Gb/s. Since the link is 100Gb/s, the maximum bandwidth the competing flow could receive is 40Gb/s. However, since we only achieve just over 90Gb/s, the competing flow receives less than 40Gb/s.

We can’t achieve the same performance as no contention because of switch parameters and per-port fair sharing on the switch. If the victim’s injection rate is at

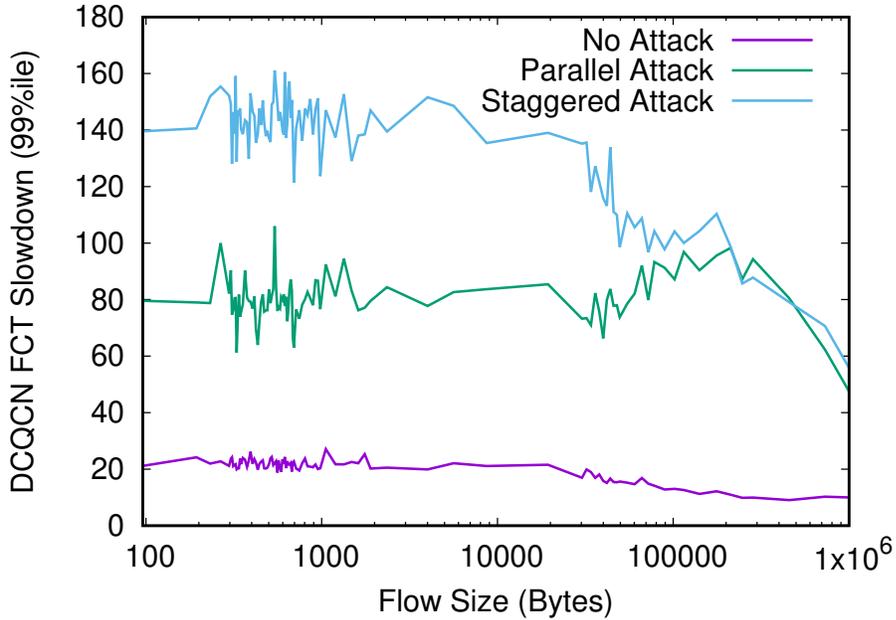


FIGURE 3.11: Victim Traffic DC Simulation

least half of line rate, then the victim receives its fair share because the switch only allocates more bandwidth to the attacker’s port if the victim’s port does not send enough traffic to saturate the link. Because the attacker only sends 1.125MB and the switch does not mark packets until the switch queue depth exceeds 200KB, the victim does not reduce its rate enough for the attacker to achieve line rate in our system. To demonstrate that a longer-lived attack would be more detrimental to the victim flow, we lower the ECN marking threshold to 8KB, so the victim backs off earlier. Figure 3.10 shows that ”Staggered Low Marking” matches the theoretical limit of the staggered attack.

3.6.2 Datacenter Simulations

To show the drastic effect these attacks have on overall network health, we run 10ms of random traffic with flow sizes based on Hadoop traffic from Facebook [108]. We perform three experiments. First, we run DCQCN with per-QP fairness. We rerun the same traffic except we break every flow >1MB into smaller flows of 150KB

each, so a 1MB flow becomes six 150KB flows and one 100KB flow. We start all these flows at the same time. In our third experiment, we simulate the staggered attack and break up the large flows but wait for 13us (RTT of the network) before starting each new flow. Figure 3.11 plots the 99% FCT slowdown of victim flows from the three experiments. The FCT slowdown is the relative slowdown to the flow’s theoretical completion time without congestion (propagation delay plus serialization delay). Victim flows are smaller than 1MB because they did not break into smaller flows to get more bandwidth. The parallel and staggered attacks devastate the performance of small flows. The 99% slowdown of flows less than BDP goes from about $\sim 20x$ without the attacks to over $\sim 80x$ with the parallel attack and $\sim 140x$ with the staggered attack. We observe similar trends when we performed the same experiments with HPCC [75]. No matter the CC algorithm, these attacks create congestion because they allow a user to ignore the CC.

3.7 Potential Solutions and Future Work

The current IB/RoCE specification leaves the network susceptible to several performance attacks, and solutions to the hacks expose a fundamental tradeoff between starting flows at line rate and performance isolation. These attacks exist because CC is enforced per-QP and QPs start sending packets at line rate. Changing CC enforcement to per-src/dst easily renders the staggered and parallel attacks useless. However, enforcing CC on a per-QP granularity and allowing QPs to start at line rate is a performance optimization. It allows short flows to send all their bytes as quickly as possible. By enforcing CC on a per-src/dst granularity and potentially throttling the rate of flows when they start, short flows do not complete as quickly. This trade-off between isolation and short flow completion time is summarized in Table 3.2. We also include the susceptibility of our Mellanox hardware. If CC allows short flows to start at line rate, a long flow could pretend to be short and hack the

Table 3.2: Various Environments Susceptibility to Attack.

	IB Spec.	CTX-5	Per-src/dst CC
Parallel	Vulnerable	Vulnerable	Secure
Staggered	Vulnerable	Secure	Secure
Shuffled Overlay	Vulnerable	Vulnerable	Vulnerable
Short Flow Penalty?	No	Yes	Yes

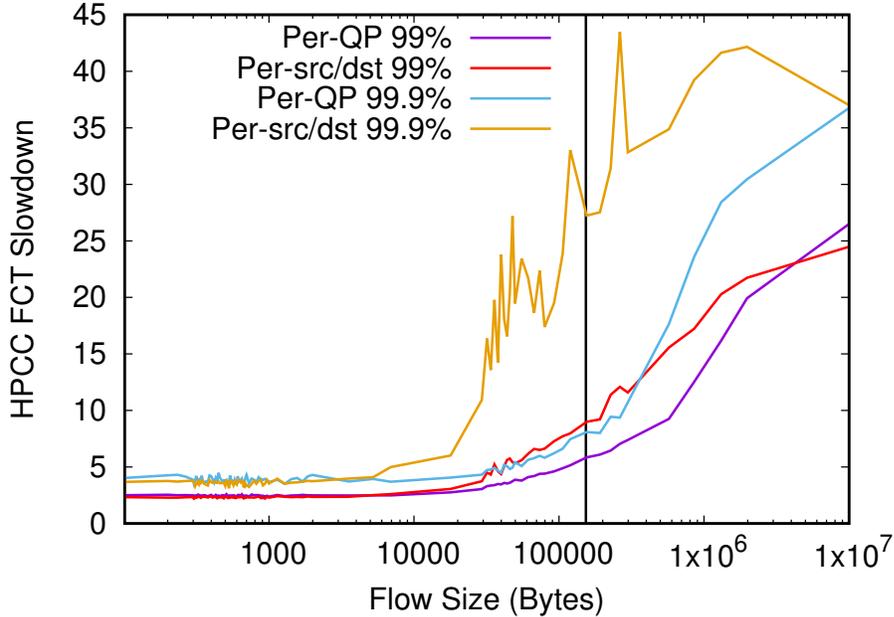


FIGURE 3.12: 99% and 99.9% tail latency of HPCC flows with different CC enforcement. The vertical line shows Bandwidth Delay Product.

system.

To demonstrate the trade-off between small flow tail latency and performance isolation, we run datacenter simulations. However, instead of showing this trade-off in DCQCN, we run simulations with HPCC. This is because the trade-off is more apparent in protocols with a congestion window and that do not define idle behavior. DCQCN allows a flow’s rate to recover over time. Per-src/dst still causes performance issues in DCQCN, but they were not as pronounced in our experiments.

Figure 3.12 shows the results of 50ms of Hadoop traffic in a network using HPCC

with different CC enforcement. The 99.9% FCT slowdown of flows between the size of 20KB-110KB goes from 5-10x to 6-34x. This is a dramatic increase for flows that are smaller than the BDP (denoted with a vertical line in the Figure) of the network. The 99% of flows also take longer to complete, but the difference is less pronounced.

We garner two insights from this result. First, that improving performance isolation in CC protocols can have an impact on the performance of the application. Second, researchers should design and evaluate CC algorithms with performance isolation in mind. We demonstrated that CC enforcement dramatically impacts performance and that it is vital to improving performance isolation.

Further, HPCC and other CC algorithms, like Timely and Swift, omit characteristics that should be defined. For example, designers should consider the idle behavior of a CC algorithm. In DCQCN, the rate of a flow increases over time, while in HPCC an idle flow's rate does not change. Recovering the rate over time in HPCC may improve performance.

Additionally, per-src/dst fairness only solves the parallel and staggered attacks; solving the shuffled overlay attack requires a more complex solution. Because the src-dst pairs change completely, the solution is unlikely to be implementable on a NIC. Chapter 5 shows how we allow flows to start sending at line rate, but do not leave the network susceptible to attacks. We have to relax the network's lossless property, but we maintain packet ordering and do not detect loss, which are the main reasons to keep a network lossless. Existing solutions like CSFQ and AFQ [102, 98] do not preserve these properties.

3.8 Summary

This Chapter introduces two issues with RDMA congestion control that are rooted in convergence. The first issue shows that some existing RDMA congestion control protocols converge to fair bandwidth allocations slowly. This causes the network to

unfairly allocate some flows too little bandwidth, which greatly extends their completion time. This harms overall application performance by increasing large flow tail FCT. Next, we demonstrate that allocating bandwidth at a per-QP granularity and starting flows at line rate leaves a network susceptible to performance isolation attacks. These attacks give an attacking user too much bandwidth and cause congestion in the network.

We address the issue of long flow completion time in sender-side protocols in Chapter 4. In Chapter 5, we make the network safe from performance isolation attacks using speculative packets and in-network computing.

Improving Long Flow Tail Latency

Tail latencies are critical for many distributed applications [25]. In order to minimize tail latencies, the network must ensure high, predictable performance. Network congestion delaying small data transfers is a common example of the network harming application performance and congestion control protocols optimize small flow FCT. Many applications also rely on long flow tail latencies. For example, deep-learning often requires an All-Reduce to aggregate gradients. All-Reduces can move large amounts of data to all servers within a computational group. Since an All-Reduce cannot complete until all end hosts receive data from all other end hosts it is tail bound; the operation cannot complete until the last flow finishes.

Chapter 3 demonstrates how state-of-the-art congestion control techniques trade convergence to fairness in favor of latency and throughput. Because protocols favor throughput and latency above fairness, most flows complete quickly, but an unfortunate few take several times longer to complete than necessary. To reign in long flow tail FCTs, we introduce Variable Additive Increase and Sampling Frequency: two new mechanisms that raise fairness to be on-par with latency and throughput

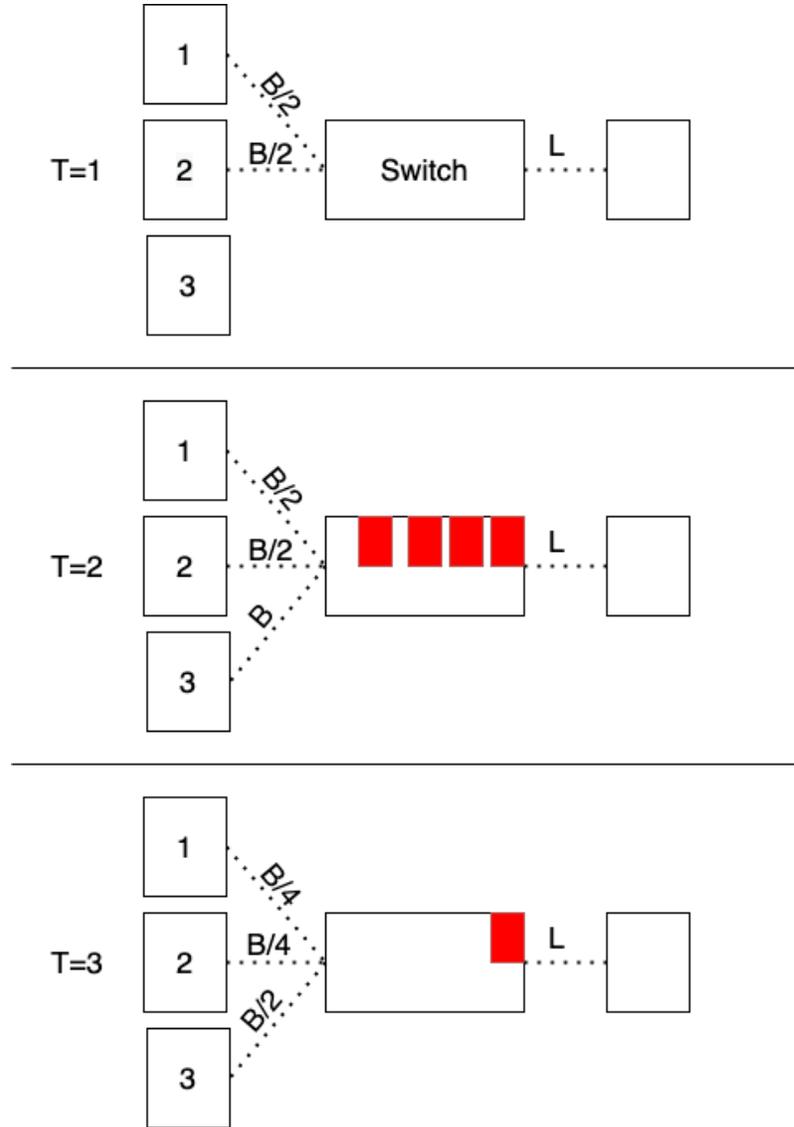


FIGURE 4.1: Unfairness occurring when starting at line rate

and can be added to many sender-side congestion control protocols. Our mechanisms improve long flow tail FCT without causing congestion or leaving the network underutilized. Two key observations motivate the new mechanisms for existing protocols: 1) Bandwidth allocations are unfair when new flows join, and 2) Queues on a bottleneck link increase dramatically when new flows join.

We observe congestion control protocols allocate new flows significantly more bandwidth than existing flows because RDMA networks generally start sending new

flow packets at line rate [75, 110]. We use a simple example in Figure 4.1 to demonstrate this principle. In timestep 1 ($T=1$), there are two flows, 1 and 2, sharing a link L with bandwidth B . The bandwidth allocation is perfectly fair, so flows 1 and 2 have the same rate and the sum of their rates is $\leq B$, so no queue builds up. In timestep 2 ($T=2$), flow 3 joins and starts sending at line rate. In this situation a queue builds up on L because the sum of the rates of all flows exceeds B . This causes a congestion event and all the flows reduce their rate with a multiplicative decrease, which in this example is 2. Before the multiplicative decrease flows 1 and 2 rates were $B/2$ and flow 3's rate was B . In timestep 3 ($T=3$) after the decrease, Flow 1 and 2 have a rate of $B/4$ and flow 3 has a rate of $B/2$. This example shows how a new flow joining is the primary source of unfairness since the protocol allocates more bandwidth to the new flow than the existing flows. Given a reasonable protocol, the rates should eventually converge to fair rates, but while the protocol converges, flows 1 and 2's performance suffers because the network allocates them too little bandwidth.

The second observation is that a new flow joining the network leads to a large increase of the queue on the bottleneck link. This occurs because the bottleneck link is fully or nearly saturated before the new flow joins. The switches queue any additional packets arriving at the link, which creates congestion. Using this observation, we infer that a large increase in congestion corresponds to a new flow joining the network.

4.1 Variable Additive Increase and Sampling Frequency

Using these two observations, we create two new mechanisms that improve fairness without significantly increasing queueing delay or losing throughput. Chapter 3 outlines how many congestion control protocols improve latency and throughput by using conservative additive increase parameters and reacting only once per-RTT.

Our mechanisms maintain low latency and high throughput while also improving fairness. The first mechanism is a Variable Additive Increase (VAI), which adjusts the additive increase parameter when the end host believes the bandwidth allocation is unfair. Increasing the additive increase only temporarily, forcing fairer bandwidth allocations without incurring queueing delay in the common case when allocations are fair. The second mechanism is Sampling Frequency, which updates a flow’s rate after a certain number of ACKs instead of each RTT. Reducing rates after a certain number of acknowledges restores a natural fairness affect where flows with more bandwidth reduce their rate more often than those with less bandwidth. Variable AI allows a user to trade off latency for the sake of fairness with a smaller latency penalty than simply increasing a standard additive increase parameter. Sampling Frequency allows users to trade off bandwidth for slightly lower latency and improved fairness. *Our mechanisms extend the design space beyond the well documented trade off between latency and bandwidth [5] to include fairness.*

4.1.1 Variable AI

Variable AI increases the AI value when the protocol detects the rate allocation may be unfair and decreases the AI value when the allocation is fair to keep latency low. HPCC and Swift use Additive Increase to enforce fairness; however, as discussed previously the AI value is set conservatively to keep queue oscillations small.

Exploiting the observation that bandwidth allocations are generally unfair right after a new flow joins, we make additive increase a function of congestion. However, we make careful choices to ensure this does not lead to further congestion during large congestion events, such as incast.

When congestion occurs, Variable AI creates AI tokens. Algorithm 1 includes the entire protocol. The protocol produces AI tokens by dividing the difference between "Measured Congestion" (Queue depth in HPCC and RTT in Swift) by a

Algorithm 1 Generating Tokens and Setting Dampener

```
1: if RTT Finished then
2:   if Measured Congestion > Token_Thresh then
3:     AI_Bank = min( $\frac{\text{Meas. Cong.}}{\text{AI\_DIV}} + \text{AI\_Bank}$ , Bank_Cap)
4:   if Measured Congestion > Token_Thresh then
5:     dampener+ =  $\frac{\text{Meas. Cong.}}{\text{Token\_Thresh}}$ 
6:   else if AI_Bank == 0 then
7:     if No Congestion then
8:       dampener = 0
9:     else if Measured Congestion < Token_Thresh then
10:      dampener = max(dampener-1, 0)
11:   Measured Congestion = 0
```

Algorithm 2 Calculate Additive Increase

```
1: tokens = 1
2: tokens = min(AI_Cap, AI_Bank)
3: if Rate Adjustment then
4:   AI_Bank = max(AI_Bank - tokens, 0)
5: divisor = (dampener / Dampener_Constant) + 1
6: tokens = max( $\frac{\text{tokens}}{\text{divisor}}$ , 1)
7: Additive Increase = tokens * base_AI
```

configurable constant (AI_DIV) when "Measured Congestion" exceeds a threshold (Token_Thresh). The protocol adds these tokens to the AI_Bank, which cannot exceed Bank_Cap.

Since Variable AI is a function of congestion and additive increases can cause congestion, Variable AI can enter a feedback loop. To prevent feedback, we add a dampener that reduces the effect of Variable AI if queues persist for a significant amount of time. Dampener is only reset to zero when there are no more tokens in the AI Bank, and there is no congestion over the entire RTT. No feedback can occur because there is no more input from Variable AI (no AI tokens), and there is no congestion to produce new tokens. The protocol increases dampener as a function of congestion, which improves performance during large congestion events like incast. If there are numerous new competing senders, like in a 100-1 incast pattern, then

Variable AI creates many AI tokens, which causes an elevated AI for a large period of time, which can increase congestion. In the case with many concurrent senders, dampener increases quickly, so the elevated AI creates less congestion.

Variable AI creates new tokens every RTT; we now detail how Variable AI spends those tokens. Algorithm 2 shows how many tokens Variable AI uses when increasing the Additive Increase. When calculating the additive increase, we multiply the default AI by the minimum of two values, the AI Cap and the number of available AI tokens. The AI Cap is the largest number of tokens the protocol can expend in a rate update period. If the rate decreases overall, Variable AI removes tokens every decrease period, which is set by the Sampling Frequency. If the rate increases, we remove tokens each RTT. A larger AI Cap leads to higher latencies but better fairness. Variable AI spreads the increased AI over time, which avoids large queues.

QCN and TCP BIC have a similar mechanism to Variable AI called Fast Recovery [3, 76]. In the face of a congestion event, Fast Recovery assumes that transient queues cause congestion signals. Therefore flows enter Fast Recovery, which quickly tries to grab the bandwidth lost during congestion. However, since Fast Recovery is designed to improve throughput and not fairness it is designed differently. Variable AI is more conservative because it assumes the link is fully utilized already and does not want to cause congestion, only improve fairness.

4.1.2 Sampling Frequency

Sampling Frequency tunes how often a protocol reacts to congestion signals. Section 3.3.2 detailed how reacting only once per-RTT leads to unfairness. Sampling Frequency determines how many acknowledgements (ACKs) a flow receives before reducing its rate. If Sampling Frequency is set low, the protocol reacts to more congestion signals because it reacts after fewer packets, so the flow decreases its rate more often.

A low Sampling Frequency (reacts often) reduces bandwidth utilization, but improves fairness and reduces packet latency. If a protocol reacts more often, the rate decreases more often and therefore is more likely to leave unused bandwidth. However, this reduces the chances of queueing delay because the rates are lower, so tail latency is reduced for latency bound flows. Most importantly, flows with more bandwidth, which receive more ACKs, reduce their rates more often than flows with less bandwidth.

Sampling Frequency is only invoked if the rate is decreasing; it has no effect if the rate increases. If we use Sampling Frequency to also increase rates, flows with more bandwidth would increase their rate more often, which goes against our goal of fairness. We recommend flows increase their rates once per RTT.

We provide proof for when Sampling Frequency improves convergence to fair rates. We use a fluid model similar to the ones used by Zhu et al [110, 109]. We model Sampling Frequency with the equation

$$f = \frac{s * MTU}{S_i(t)}$$

where f is the frequency of the multiplicative decrease, s is the number of ACKs between rate decreases, and $S_i(t)$ is the injection rate for flow i using Sampling Frequency opposed to per RTT. MTU is the maximum packet size. We then model a generic multiplicative decrease function with

$$S'_i(t) = -\frac{\beta * S_i(t)}{f}$$

where $0 < \beta < 1$. If we integrate this over a decrease interval (f), then the rate decreases by a factor of β , which is the desired behavior. When we substitute f , we get the full equation

$$S'_i(t) = -\frac{\beta * S_i^2(t)}{s * \text{MTU}}$$

To model a protocol that decreases once per RTT, we use the equation

$$R'_i(t) = -\frac{\beta * R_i(t)}{r}$$

where r is the measured RTT of the packets, and $R_i(t)$ is the injection rate for flow i when performing decreases each RTT. For simplicity, we use a fixed RTT to show the behavior while the network is congested. Since multiplicative decrease only occurs when the network is congested, this is a fair simplification.

We now show that rates using $S_i(t)$ converge faster under reasonable and desirable constraints. We use two flows and measure fairness as $S_1(t) - S_0(t)$ or $R_1(t) - R_0(t)$ and initially

$$C_1 = S_1(0) = R_1(0) > S_0(0) = R_0(0) = C_0$$

. Sampling Frequency is more fair when

$$(R_1(t) - R_0(t)) - (S_1(t) - S_0(t)) > 0 \tag{4.1}$$

at $t = 0$ the fairness metric is the same, but we show that over time given certain constraints, S_i becomes fairer faster. We show that the gap between the two protocols fairness metrics increases faster by showing

$$(R'_1(0) - R'_0(0)) - (S'_1(0) - S'_0(0)) > 0$$

when we simplify, we get the constraints that this is true when

$$\frac{1}{r} < \frac{C_1 + C_0}{s * \text{MTU}}$$

When initial injection rates are high and sampling happens frequently and round trip times are long, the rates using Sampling Frequency converge faster. This is the

desired behavior. When a new flow joins, its rate is high and it causes congestion, so RTTs are high. Sampling Frequency converges quickly when we size it appropriately.

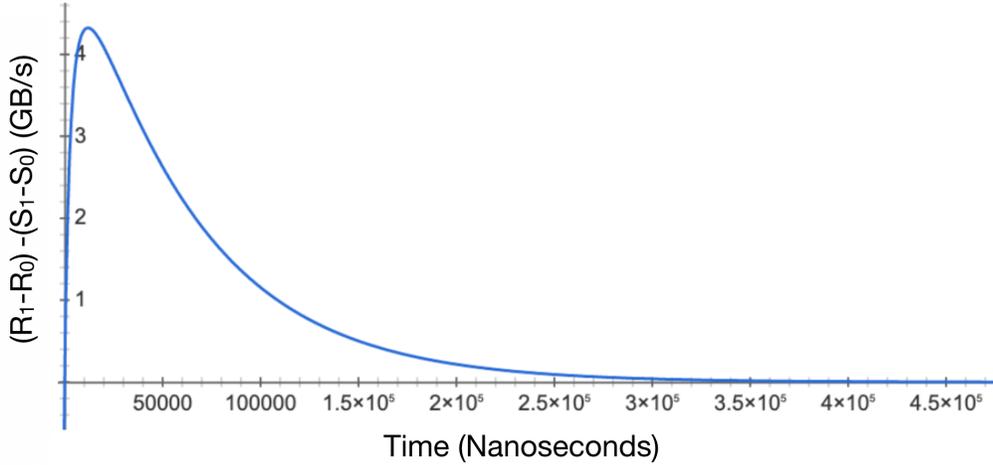


FIGURE 4.2: Plotting difference in fairness between to MD methods. $r = 30000$, $MTU=1000$, $s = 30$, $\beta=.5$

To demonstrate, we graph equation 4.1 with reasonable parameters and show how the rates converge during congestion in Figure 4.2. We use bytes per nanosecond as our units for rates, 30,000ns for the observed network RTT (r), 30 for the Sampling Frequency (s), 1,000 bytes for packet MTU, and .5 for β . The initial rates for the flows are 100Gbps and 50Gbps. The function becomes positive quickly, which indicates that Sampling Frequency is more in the short term. Eventually, per-RTT sampling converges to fair rates as well, so they become equally fair. However, the goal is to converge to nearly fair rates quickly, which the mechanism achieves and is indicated by the a positive function value with a small input.

4.2 Implementation

This section details how we implement Variable AI and Sampling Frequency in HPCC and Swift using the ns-3 network simulator. We constrain this work to HPCC and

Swift, two representative state-of-the-art congestion control protocols. Nonetheless, we believe our mechanisms are broadly applicable to other sender reaction-based protocols because they address the fundamental issues introduced in Chapter 3.

4.2.1 Variable AI

We implement Variable AI in HPCC and Swift, which require slightly different implementations due to different design methods of rate increases and congestion measurements.

Variable AI in HPCC creates new tokens using queue depth, which HPCC already requires. Based on the generic protocol in Algorithm 1, we must determine how to generate AI tokens, increase dampener, and reset dampener. Variable AI creates AI tokens only if the maximum observed queue depth during the RTT exceeds the minimum Bandwidth Delay Product (BDP) of the network. We use minimum BDP as the `Token_Thresh` because unfairness occurs when a new flow joins the network. Assuming a new flow exists for several RTTs, the new flow creates a queue buildup equal to the BDP of the flows path. This BDP is at least the minimum BDP of the network. HPCC converges to pareto optimal in terms of latency and bandwidth using MIMD. Additive increase ensures fairness. Since HPCC is MIMD, HPCC always multiplies the existing rate by C , a value calculated using network feedback. If C is > 1 , the end hosts injection rate decreases. If C is < 1 , the end hosts injection rate increases. We track the maximum C observed over a round trip time. If $C < 1$ for an entire RTT, we determine there is no congestion and Variable AI can reset the dampener.

Variable AI creates AI tokens in Swift based on RTT measurements. If an RTT measurement exceeds "target delay" in Swift, Swift performs a multiplicative decrease. Similar to HPCC, we set `Token_Thresh` to the sum of the target delay and the delay incurred by the minimum BDP of the network, so the protocol likely only

generates new tokens when a new flow joins. If no packets delay exceeds "delay target" over the RTT and the AI_Bank is empty, we reset dampener because congestion alleviated and there cannot be any feedback since there is no input into the system.

Variable AI has one issue that could increase unfairness. An existing flow could experience congestion for a long period of time and therefore have a high dampener value, which would lead to a low AI. A new flow, however, starts with a dampener value of 0. In this case, a new flow could have a higher AI than an existing flow. We found no way around this issue because it is impossible to distinguish whether feedback from the increased AI or new flows joining the network caused the observed congestion. We ran experiments with this exact pattern and Variable AI still improved fairness.

4.2.2 Sampling Frequency

Normally, Swift and HPCC wait one-RTT between rate updates. To implement Sampling Frequency, we amend Swift and HPCC to decrease their rates after a predetermined number of acknowledgements opposed to waiting an RTT. Sampling Frequency only affects when HPCC and Swift perform rate reductions. Rate increases still happen once per-RTT. If increases happened on the Sampling Frequency schedule, flows with a higher rate increase their rate more often and worsen fairness.

HPCC inspired two additional changes in Swift to make Sampling Frequency more effective. HPCC has a per-ACK and a per-RTT update schedule. HPCC maintains a "reference rate" that is updated once per-RTT. HPCC then updates the per-ACK rate after every acknowledgement, but the per-ACK updates are relative to the "reference rate" not previous per-ACK rates. Using Sampling Frequency the reference rate is updated per-sampling period, which is a configurable number of acknowledgements. As an example, suppose a flow has an injection rate of R . It receives an ACK that indicates congestion, so it updates its rate to $\frac{R}{2}$, however the

reference rate remains unchanged from R . The flow receives another packet that indicates congestion and again reduces its rate. However, it reduces its rate based on the reference rate, so the injection rate remains $\frac{R}{2}$. After an RTT, if congestion persists, HPCC updates the reference rate to $\frac{R}{2}$. If the next ACK after the reference update indicates congestion, the rate reduces to $\frac{R}{4}$. We add this same functionality to Swift. It improves performance with Sampling Frequency because when rates get low, Sampling Frequency does not update the injection rate often. The per-ACK adjustments allow Swift to react without causing long lived unfairness because flows with a higher injection rate update their per-sampling period rate more often.

The second change is to always perform an additive increase regardless of congestion like in HPCC. This improves the functionality of Variable AI since the tokens are always spent. This can have a small latency penalty, but it should only incur an additional latency equal to when latency causes the MD to equal the AI, which is a small amount of delay.

4.3 Evaluation

We implement our mechanisms and evaluate them as additional parameters in HPCC and Swift in an ns-3 simulator. Results show that these mechanisms improve fairness and throughput.

4.3.1 Methodology

We use the same parameters as Section 3.3.4. In addition to the 16-1 incast traffic, we run three new benchmarks. First, we run a 96-1 incast pattern to show our mechanisms work with a higher incast degree. Then we run two datacenter simulations. The first is based on a hadoop traffic trace from Facebook [108]. The hadoop traffic contains mostly small flows (95% <300KB). The second datacenter benchmark is flow size distributions from two applications mixed together to simulate a shared en-

Table 4.1: HPCC and Swift VAI SF Parameters

Parameter	HPCC	Swift
Sampling Frequency	30 ACKs	30 ACKs
Token_Thresh	50KB	$4\mu s$
AI_DIV	1KB	30ns
AI_Cap	100	100
Bank_Cap	1000	1000

vironment. The first application is a Microsoft WebSearch traffic pattern with many long flows (30% > 1MB), and the second application is a Alibaba storage workload with almost exclusively small flows (97.5% <256KB and 100% < 2MB). The datacenter benchmarks run the network at 50% load for 50ms. For the datacenter simulation, we use the same experimental setup for the datacenter workloads as in Chapter 3.

We summarize our VAI and SF parameters in Table 4.1. We decrease the injection rate of packets every 30 ACKs in both Swift and HPCC when they use Sampling Frequency. For Variable AI, we set Token_Thresh to the minimum BDP of the network, which is about 50KB. For Swift, we use $4\mu s$ plus target delay, which is a base target delay of $5\mu s$ and $2\mu s$ are added per-hop for topology based scaling. $4\mu s$ is the delay incurred when queue depth is 50KB. In HPCC, Variable AI produces 1 AI token for every KByte of queue depth (AI_DIV) and set bank cap to 1000 tokens. In Swift, we produce an AI token for every 30ns of queueing delay, and cap the number of tokens at 1000 as well. Variable AI for both HPCC and Swift can only use 100 tokens at a time (AI_Cap). We set dampener constant to 8 in Swift and HPCC.

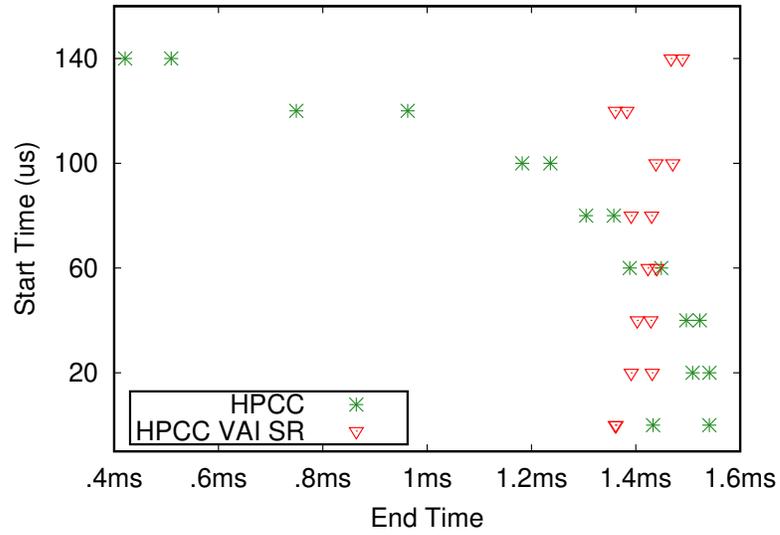


FIGURE 4.3: 16-1 Incast Traffic Start Time vs Finish Time with HPCC

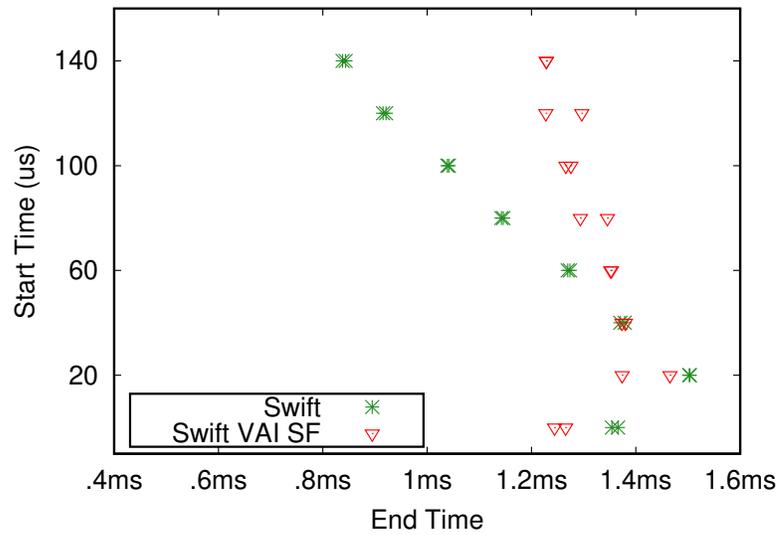
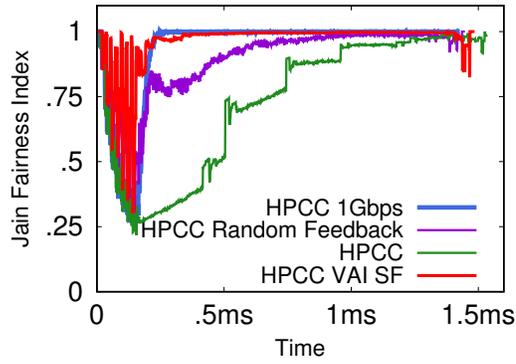
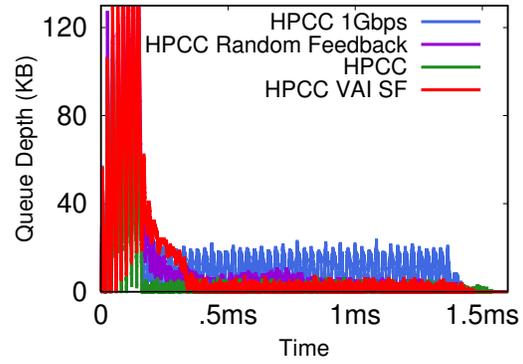


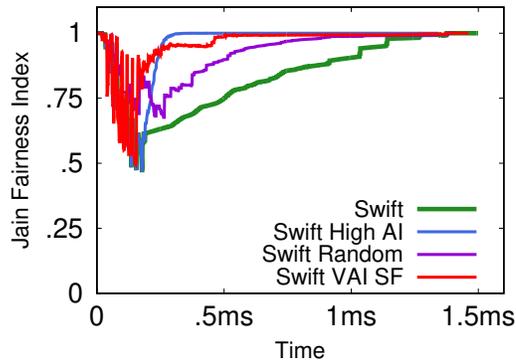
FIGURE 4.4: 16-1 Incast Traffic Start Time vs Finish Time with Swift



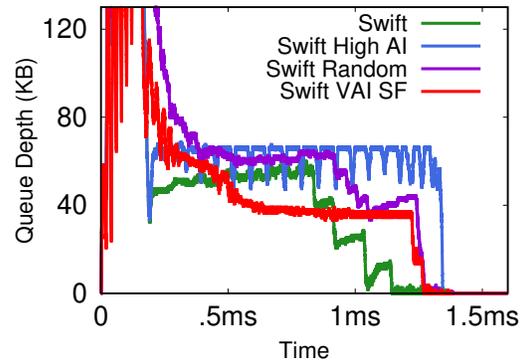
(a) HPCC 16-1 Incast Jain Fairness Index



(b) HPCC 16-1 Incast Queue Depth



(c) Swift 16-1 Incast Jain Fairness Index



(d) Swift 16-1 Incast Queue Depth

FIGURE 4.5: 16-1 Incast in HPCC and Swift

4.3.2 Experimental Results

Incast

First, we rerun the 16-1 incast congestion from Chapter 3 and compare our VAI SF variants with the existing baselines. Figures 4.3 and 4.4 show the start versus finish time of HPCC and Swift with default settings versus their VAI SF variants. Omitting the HPCC and Swift with a higher AI and random feedback avoids clutter. The finish time of the flows is much closer together when using our mechanisms. Figures 4.5(a) and 4.5(c) further demonstrate the improved fairness. Our mechanisms converge to a Jain Index of nearly 1 much quicker than with default settings and about as quickly as the high AI and random variants. Figure 4.5(b) shows that when using

VAI and SF, HPCC still maintains near 0 queues. Figure 4.5(d) shows that Swift with VAI and SF sustains smaller queues than all other variants likely because we do not use FBS, which increases the tolerated queuing delay. Swift VAI SF also has small queue oscillations.

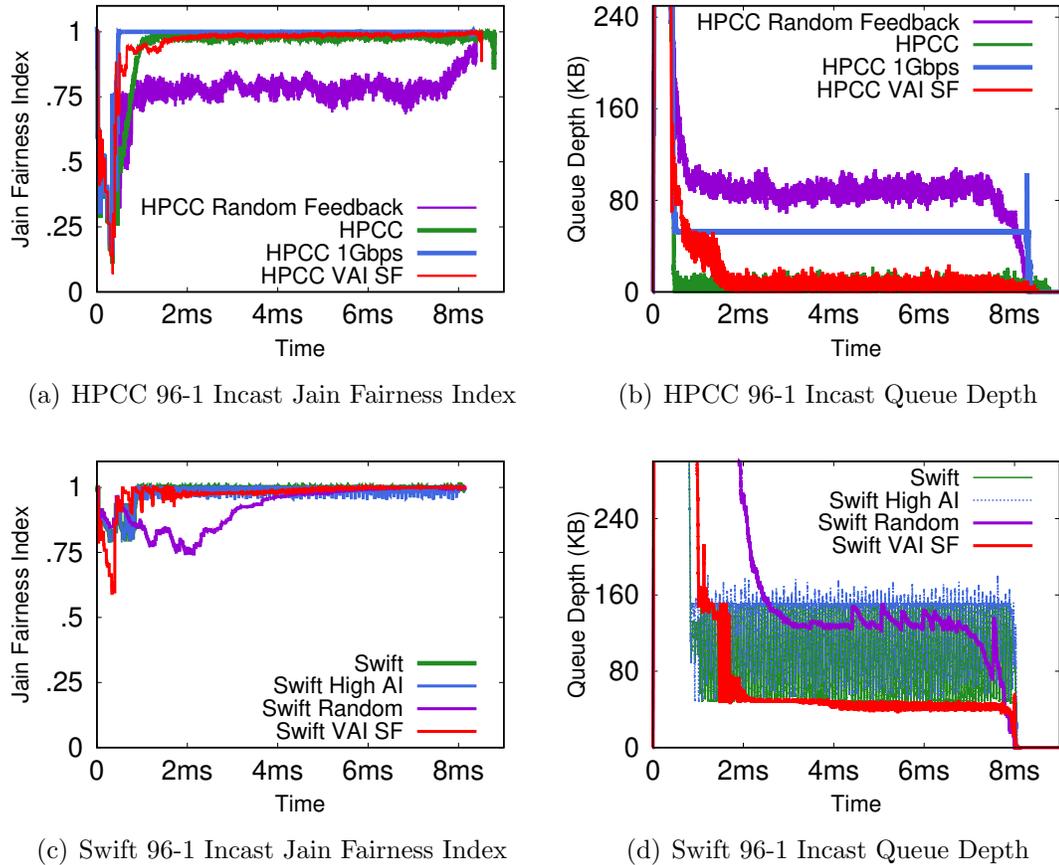


FIGURE 4.6: Jain Fairness Index and Queue depth during Incast Traffic with 96-1 Incast in Swift and HPCC

The same trends continue when we scale the incast to 96-1. Figures 4.6(a) and 4.6(c) show that when using VAI and SF, the system becomes fair quickly. Figure 4.6(b) shows that HPCC VAI SF again maintains near zero queues like the default HPCC configuration. Meanwhile the other variants sustain queues throughout the experiment. In Figure 4.6(d), Swift maintains the smallest queue because it does not

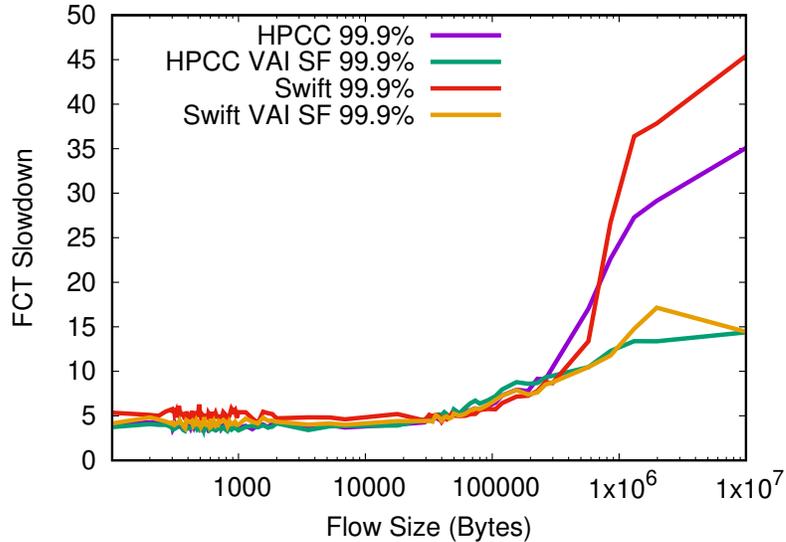


FIGURE 4.7: 99.9% FCT for various flow sizes in Hadoop Traffic

use FBS, and smaller oscillations because it has a small AI in the steady state.

Datacenter Simulations

We found that a slow convergence to fair rates significantly impacted the performance of long flows, particularly at the tail. Figure 4.7 shows how the unfairness affects long flows in a Hadoop traffic pattern. We plot the FCT slowdown as a function of flow size. We take the 99.9% from each flow size. The FCT Slowdown divides the achieved FCT by the theoretical minimum FCT (propagation delay + serialization delay). Because Swift and HPCC keep small queues and enable low packet queueing delay, small flows complete quickly. However, as the flow sizes increase and FCT becomes a function of bandwidth allocation, the FCT slowdowns start to increase. For flow sizes greater than 1MB, HPCC and Swift without our mechanisms perform poorly. Flows take 20-40x longer to complete than the theoretical minimum. When we add our mechanisms, long flow performance improves substantially. Our mechanisms halve the tail FCT of long flows; the FCT slowdown goes from 30-40x without our mechanisms to 10-15x with our mechanisms.

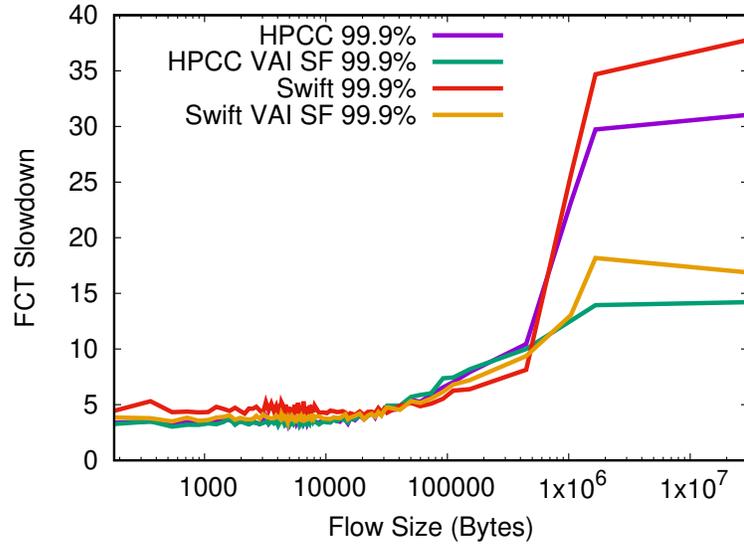


FIGURE 4.8: 99.9% FCT for various flow sizes in WebSearch and Storage Traffic

Figure 4.8 shows the same trend with the Websearch and Storage benchmark. The FCT slowdown of flows greater than 1MB grows to several times compared to smaller flows. Meanwhile with our mechanisms the FCT stays several times lower. This improves the performance of any workload that relies on long flows and needs small tail latencies like big-data and deep-learning.

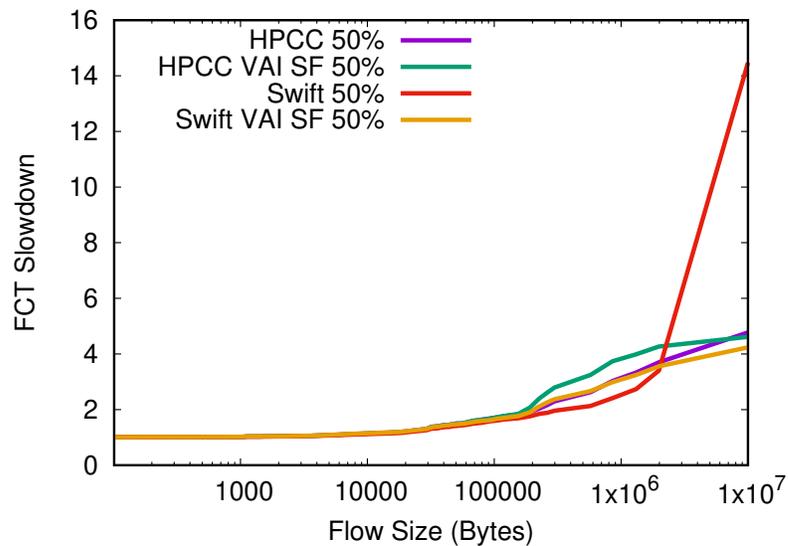


FIGURE 4.9: Median FCT for various flow sizes in Hadoop Traffic

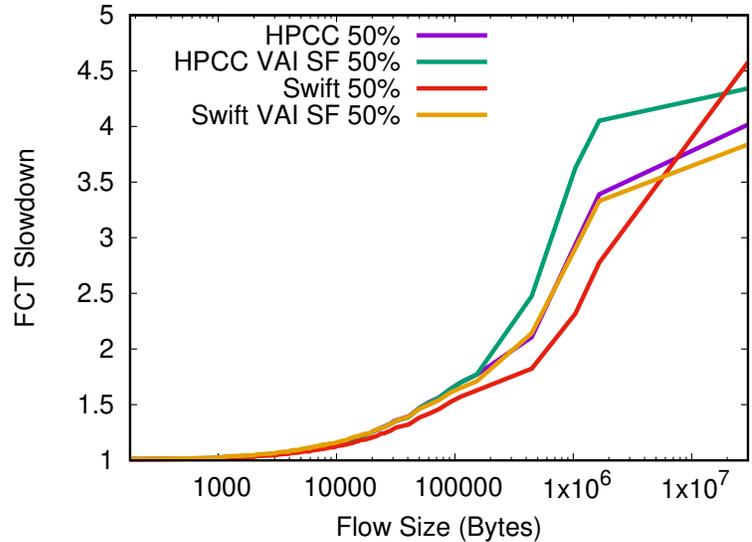


FIGURE 4.10: Median FCT for various flow sizes in WebSearch and Storage Traffic

VAI and SF improve the tail FCT with no significant repercussions on median FCT. Figures 4.9 and 4.10 show the median FCT slowdown in the Hadoop and Websearch/Storage workloads respectively. This shows that VAI and SF do not incur any extra queueing delay in the common case. The median FCT slowdown in Hadoop for Swift is significant. Swift only uses a single, constant additive increase, which may cause rates to recover slowly even if there is significant available bandwidth. The slowdown is not present in the Websearch/Storage workload. Swift may benefit from a hyper additive increase setting like in Timely [81], which can help grab available bandwidth.

4.4 Conclusion

Slow convergence to fairness significantly raises the FCT of long flows. This chapter identifies that unfairness occurs when new flows join the network and that a new flow joining causes large packet queue depth on network switches. Based on these observations, we develop Variable Additive Increase and Sampling Frequency, two sender-side mechanisms that greatly improve fairness without harming throughput

or increasing queue size. We evaluate our mechanisms in HPCC and Swift and show a dramatic improvement in fairness during micro-benchmarks and datacenter simulations. In a datacenter simulation, our mechanisms improved 99.9% by more than 2x. While a significant performance improvement, convergence still takes several round trips. Further, the flows still start sending packets at line rate, which leaves the network to the attacks described in Chapter 3. In the next chapter, we describe how flows can start at line rate and converge during the first RTT, so the network is free from performance isolation attacks.

Towards RDMA Performance Isolation and Effective Congestion Control in High BDP Networks

Effective and secure congestion control is imperative for networks. Without an effective congestion control protocol, applications can suffer high latencies [110, 75] and degraded throughput [46, 90], which harms application performance. As shown in Chapter 3, if users can abuse a network’s congestion control, a nefarious user can cause severe congestion in the network and unfairly gain bandwidth. This Chapter explores mechanisms to ensure that congestion control is effective in shared environments and future networks.

Two trends challenge existing congestion control methods in RDMA networks: 1) RDMA is becoming more common in shared environments and 2) network bandwidth is growing rapidly. Originally, sharing an RDMA network was not an issue. RDMA was reserved for supercomputers, which are either not shared or partitioned statically, and there was little chance for performance interference. However, now RDMA is commonly available in cloud platforms such as Microsoft Azure, Google Cloud, and Amazon Web Services. Numerous users share these systems at a fine granularity,

and isolating their performance is essential. Chapter 3 demonstrates how users in RDMA networks can unfairly gain extra bandwidth. *This chapter introduces new mechanisms that mitigate a user’s ability to unfairly gain bandwidth.*

The second issue facing RDMA congestion control is that common congestion techniques in RDMA networks become ineffective as network BDP increases. RDMA uses reactive/sender-side congestion control, and end hosts only reduce their injection rates when they receive congestion signals from the network, like ECN marked packets [110], RTT increases [66, 81], or In-band Network Telemetry [75]. Many protocols [110, 81, 75] and the IB specification [45] start sending packets at line rate. Since it takes at least one RTT for congestion protocols to throttle injection rates, an end host can send BDP bytes before the NIC reduces its injection rate. Currently this is effective, but as network bandwidth scales. current congestion control mechanisms are ineffective because most flows are smaller than network BDP.

A naive solution to the issues posed above is starting flows pessimistically. A flow sends a single packet and ramps up its injection rate over time, similar to TCP. However, this has negative performance implications. Starting flows at line rate allows small flows to complete quickly and ensures end hosts use all available bandwidth. We want flows to start at line rate without the congestion and isolation issues that occur with misbehaving users and networks with large BDPs.

We introduce two mechanisms that mitigate a nefarious users ability to unfairly gain extra bandwidth: 1) One-RTT Convergence (1RC) and 2) a new flow weighting scheme. Using 1RC, flows converge to their fair rates during the first RTT, so flows can start sending packets at line rate, but the network is not susceptible to the staggered attack or shuffled overlay attack because the network drops packets from flow’s that exceed their bandwidth allocation. Our new weighting scheme reduces a flows bandwidth allocation if the flow’s end host has numerous open connections. We set a flows weight equal to the inverse of the number of concurrent connections

on the end host, so the network allocates less bandwidth to end hosts with many flows. This aims to reduce the effectiveness of the parallel attack and parallel shuffled overlay attack.

Both the issues of performance isolation attacks and congestion control becoming ineffective stem from flows starting at line rate and taking at least one-RTT to converge to their fair rates. This chapter shows how we can converge to fair rates during the first RTT and start sending packets at line rate. Our mechanism, One RTT Convergence (1RC) uses speculative packets and system-wide synchronized end host clocks [68, 74, 69] to drop certain packets. 1RC only drops a flow's packets if letting the packets through would unfairly allocate the flow too much bandwidth. Unlike existing speculative packet methods, 1RC drops packets in order and detects loss implicitly, so end hosts do not need to detect packet loss or reorder packets. We show that 1RC makes a network less susceptible to performance isolation attacks and enables congestion control to work even in high BDP networks.

1RC requires a switch-based protocol because the switch must know the flow's fair rate. If the switch did not know the fair rate, the switch would not know how many packets to drop. While there are many effective switch-based protocols [61, 28, 103], we choose s-PERC because Jose et al. [55] designed it for datacenter networks and it converges to max-min fair rates in bounded time. Note that 1RC does not guarantee that a flow converges to its max-min fair rate in one RTT, only to the fair rate computed by the switch, which may take several RTTs to converge. 1RC requires programmable switches to calculate fair injection rates, store data about how many packets to drop, and selectively drop packets when a packet meets certain criteria.

Using 1RC, flows converge to their allocated rates during their first RTT, do not have to reorder packets or detect drops, and receive only their fair share of bandwidth, even with a misbehaving user. 1RC is not a complete congestion control protocol and therefore does not calculate injection rates for end hosts, it is simply

a method to drop speculative packets in-order. 1RC works with any switch-based protocol and allows flows to start sending packets at line rate without allocating the new flow an unfair amount of bandwidth.

The second method to improve performance isolation is a new weighting scheme. In traditional fair queueing, increasing a flow’s weight increases the flow’s bandwidth allocation. For example, a flow with a weight of 2 receives twice the bandwidth as a flow with a weight of 1. While we do not perform true fair queueing on the switch because that requires a virtual channel for each flow, one can also weight network flows at end hosts [22], so a flow converges to a higher bandwidth allocation as its weight increases. We instead reduce a flow’s weight as the flow’s source opens more connections. This reduces the effectiveness of the parallel QP and shuffled overlay attacks shown in Chapter 3.

5.1 Background

1RC augments several existing congestion control methods. While 1RC works in any switch-based congestion control protocol, we use s-PERC [55] because it enables low latency and high throughput networking and converges to max-min fair rates in bounded time. 1RC also uses speculative packets [49, 51, 42, 43], which are lower priority packets that the network can drop if the network becomes congested. Finally, 1RC relies on datacenter-wide synchronized system clocks [68, 74, 69]. This ensures 1RC drops the correct number of speculative packets. We also introduce work related to setting flow weights to discourage misbehaving users. This section provides an overview of these foundational techniques.

5.1.1 *s-PERC*

s-PERC converges to max-min fair rates in bounded time without storing per-flow state. s-PERC tracks several summary statistics about the switches usage, such as

Algorithm 3 s-PERC

```
1: b, x, s: vector of bottleneck, allocated rates, and bottleneck states in packet
   (initially  $\infty$ , 0, E, respectively).
2: i: vector ignore bits in a packet (initially, 1)
3: SumE, NumB: sum of limit rates of E flows, and number of B flows at link
4: MaxE, MaxE': max. allocated rate of flows classified into E since last round
   (and in this round, respectively) at link.
5: if  $\mathbf{s}[l] = E$  then
6:    $\mathbf{s}[l] \leftarrow B$ 
7:    $SumE \leftarrow SumE - \mathbf{x}[l]$ 
8:    $NumB \leftarrow NumB + 1$ 
9:    $b \leftarrow (\mathbf{c} - SumE) / NumB$ 
10: for each link j do
11:   if  $\mathbf{i}[j] = 0$  then  $\mathbf{p}[j] \leftarrow \mathbf{b}[j]$  else  $\mathbf{p}[j] \leftarrow \infty$ 
12:  $\mathbf{p}[l] \leftarrow \infty$ 
13:  $e \leftarrow \min \mathbf{p}$ 
14:  $x \leftarrow \min(b, e)$ 
15:  $\mathbf{b}[l] \leftarrow b, \mathbf{x}[l] \leftarrow x$ 
16: if  $b < MaxE$  then  $\mathbf{i}[l] \leftarrow 1$  else  $\mathbf{i}[l] \leftarrow 0$ 
17: if flow is leaving then
18:    $NumB \leftarrow NumB - 1$ 
19: else if  $e < b$  then
20:    $\mathbf{s}[l] \leftarrow E$ 
21:    $SumE \leftarrow SumE + x$ 
22:    $NumB \leftarrow NumB - 1$ 
23:    $MaxE \leftarrow \max(x, MaxE)$ ;  $MaxE' \leftarrow \max(x, MaxE')$ 
```

the number of flows using the switch, the number of flows bottlenecked at the current switch, and the rate sum of flows bottlenecked at other switches.

To explain the s-PERC algorithm, we show the original algorithm from Jose et al. [55] in Algorithm 3. In s-PERC, each end host periodically sends a control packet that contains four pieces of data for the switches: 1) a vector of the bottleneck rates at each switch, 2) a vector of the rates allocated by each switch, 3) the flow's current state at each switch, and 4) ignore bits.

The switches use the bottleneck rates (**b**) on the control packet to determine where the flow is bottlenecked (lines 10 and 11). Each switch calculates the bottle-

neck rate (line 9) adds the bottleneck rate to the control packet (line 15).

If a flow is bottlenecked elsewhere, s-PERC does not want to allocate the flow bandwidth that it will not use. To redistribute the bandwidth not used for flows bottlenecked elsewhere and determine the bottleneck rate for flows that are bottlenecked at the current switch, each switch keeps a sum of the rates for all flows bottlenecked elsewhere ($SumE$). Line 19 determines if the flow is bottlenecked at the current switch or another switch by comparing the bottleneck rate on the current switch (b) with the lowest assigned rate at another switch or on the end host (e). If the flow is bottlenecked elsewhere, the switch marks that the packets belongs to E (line 20), adds the minimum allocation for the flow (e) to $SumE$ (line 21), removes the flow from B (line 22), and checks if the current flow's allocation is higher than any other flow in E (line 23).

To calculate the bottleneck for flows bottlenecked at the current switch, line 15 subtracts $SumE$ from the link's bandwidth (c) and divides that result by the number of flows currently bottlenecked at that switch ($NumB$). Line 13 determines the flow's previous bottleneck (e), and line 14 determines the minimum rate allocated to a flow including the current switch (x).

Line 16 sets the ignore bit for the control packet. $MaxE$ is the maximum rate of a flow bottlenecked at another switch. If $MaxE$ is greater than b , some flow (not necessarily the flow associated with the current control packet) was bottlenecked at another switch, but now should be bottlenecked at the current switch. This unfairly allocates flows currently bottlenecked at the current switch too little bandwidth because $SumE$ is larger than it should be. s-PERC does not want to propagate this unfair rate to other switches, so on line 16, if the bottleneck rate is less than $MaxE$, the switch sets the ignore bit for that hop on the current flow. This tells subsequent switches to ignore that allocation because it is unfairly low (line 11). We augment s-PERC to use 1RC and weights, which make it more secure and robust as BDP

increases. The core s-PERC algorithm does not change, but we make slight changes that we explain in Section 5.2.

5.1.2 *Speculative Packets*

Switch and receiver-based congestion control methods incur scheduling overhead. For example, in s-PERC if a flow is larger than BDP, the flow must send and receive a control packet before sending more packets. This ensures the switch allocates the flow bandwidth before the flow starts sending data packets. s-PERC exempts short flows ($< \text{BDP}$), so they can complete quickly. This problem also occurs in other switch and receiver-based protocols such as ExpressPass [20].

To alleviate scheduling overhead, some receiver and switch-based protocols use speculative packets. Speculative packets are lower priority, and switches can drop them if the switch is sufficiently congested. In the best case, the network is not congested, and switches deliver all speculative packets, so the flow completes quickly. In the worst case, the network drops some packets and schedules the flow, so the flow completes no slower than without using speculative packets.

Several congestion control protocols [49, 51, 42, 43] use speculative packets to reduce scheduling overhead. However, no speculative packet method maintains packet ordering. SRP [49], LHRP [51], and NDP [42] all assume infinite reordering resources at the destination. Aeolus [43] requires packets arrive in order, so if the network drops a single speculative packet, the source resends all speculative packets once the destination schedules the flow. This wastes bandwidth because the network delivers packets that the source resends. Homa [83] sends unscheduled packets (similar to speculative packets) on a higher priority than scheduled traffic. This can cause large flows to have long completion times.

Bai et al. [7] noticed that short flows tail latencies were still high even with speculative packets because the network dropped speculative packets for short flows.

They introduced SSP [7], which prioritizes short flow’s speculative packets over all other packets. This enables short flows to complete quickly because their speculative packets are rarely, if ever, dropped. While 1RC does not prioritize short flows, it does reduce the chance the network drops short flow speculative packets because 1RC lets through a fair number during the first RTT, instead of dropping all speculative packets during congestion like previous methods.

Further, to our knowledge no existing speculative packet mechanisms consider speculative packet fairness; if the port queue contains too many packets, the port drops the speculative packet. In a worst case scenario, if two speculative flows compete for bandwidth, the switch may let all of one flow’s speculative packets through and no speculative packets through for the other flow.

1RC provides a speculative packet mechanism that maintains packet ordering and fairness between flows that start around the same time. This enables simpler end hosts that are not required to handle reordering.

5.1.3 *Synchronized Clocks*

Many applications rely on systems with synchronized clocks, such as transactional key-value stores [80], ensuring consistency during network operations [72], and congestion control [88]. Systems synchronize clocks with mechanisms like PTP [68], DTP [69], HUYGENS [37], and Sundial [74], which exchange messages between end hosts and can achieve clock skews <100ns.

We use synchronized clocks to ensure flows that start around the same time receive a similar amount of bandwidth. Without synchronized clocks, we could include timestamps on packets, but the times would be meaningless on a global scale because different timestamps could refer to the same physical time due to clock skew. Synchronized clocks give us a global reference point for when speculative packets entered the network.

5.1.4 *Diminishing Weight Scheduling*

Garg et al. [35] proposed Diminishing Weight Schedulers (DWS) that reduces the weight of misbehaving users. Garg et al. design DWS for networks where there is no control over end hosts. To discourage users from sending too many packets, DWS reduces a flow’s weight if the flow sends too many packets. DWS then drops packets from a misbehaving flow if the flow’s rate exceeds its fair share. Similar to DWS, we decrease a flow’s weight if it might be misbehaving. DWS does not fit our model because it relies on a single end host sending along a path and does not account for a user trying to gain bandwidth by sending along multiple paths simultaneously.

5.2 1RC Design

1RC ensures that whenever a new flow joins, it converges to its assigned rate during its first RTT. 1RC’s goal is solving the staggered attacks described in Section 3. There are simpler ways to prevent the staggered attacks, such as slow start or existing speculative packet methods, but those solutions have drawbacks. For example, slow start harms short flow performance because a flow that could complete in one RTT now takes several to ramp up its rate, and existing speculative packet methods require end hosts to reorder packets if the network drops speculative packets out-of-order. 1RC allows flows to start sending packets at line rate, drops packets in-order, and ensures no flow gets more bandwidth than the protocol allocated to other flows that sent packets around the same time.

1RC does not determine each end hosts injection rate, it is simply enforces rates that an existing switch-based congestion control protocol calculates. It runs on a switch alongside an existing switch-based congestion control method. 1RC enforces fairness between flows starting in the same *epoch*, which is a network wide time period that determines how many speculative packets should be allowed through

the network for a new flow. Each *epoch*, 1RC takes one fair rate calculated by the switch’s existing congestion control protocol and stores that rate in terms of packets in the *window table*. When a speculative packet arrives at the switch, the switch checks the *epoch* that the flow started in and discards speculative packets based on the *window table* entry for that epoch. If the sequence number is greater than or equal to the *window table* entry for the flow’s epoch, the switch drops the packet.

A network administrator can configure the length of an epoch, but we choose the maximum network RTT when the network is not congested. In other terms, the longest propagation delay between any two hosts. *Epoch* length is the same across the entire network.

The *window table* determines how many speculative packets the switch lets through from each flow. When the first control packet in a new *epoch* arrives, 1RC stores the rate assigned to that flow in the *window table*. For example, if the *window table* entry for an *epoch* is 10 packets, 1RC lets the first 10 speculative packets from each flow that started in that *epoch* through the switch and drops all subsequent packets. End hosts send non-speculative packets once they receive their fair injection rate from switches on control packets, and switches never drop scheduled packets unless there are hardware issues.

When a speculative packet arrives at a switch, it indexes the *window table* based on the *epoch* that the flow started in. When a flow starts, the end host assigns the flow a *Speculative Start Time Stamp* (SSTS), which determines a flow’s *epoch*. The SSTS for a new flow is the system time when the end host sent the first control packet for the new flow. 1RC relies on tightly synchronized system clocks between end hosts, so flows that start at the same time have nearly the same SSTS. The switch calculates the *epoch* by integer dividing the SSTS by the *epoch* length. If division is too computationally intensive for the switch, end hosts can also calculate the *epoch* since *epoch* length is a network wide parameter. Because each speculative

packet in a new flow has the same SSTS, they all index to the same *epoch* in the *window table*. If the speculative packet's sequence number is larger than or equal to the window table's entry for the speculative packets epoch, then the switch drops the packet.

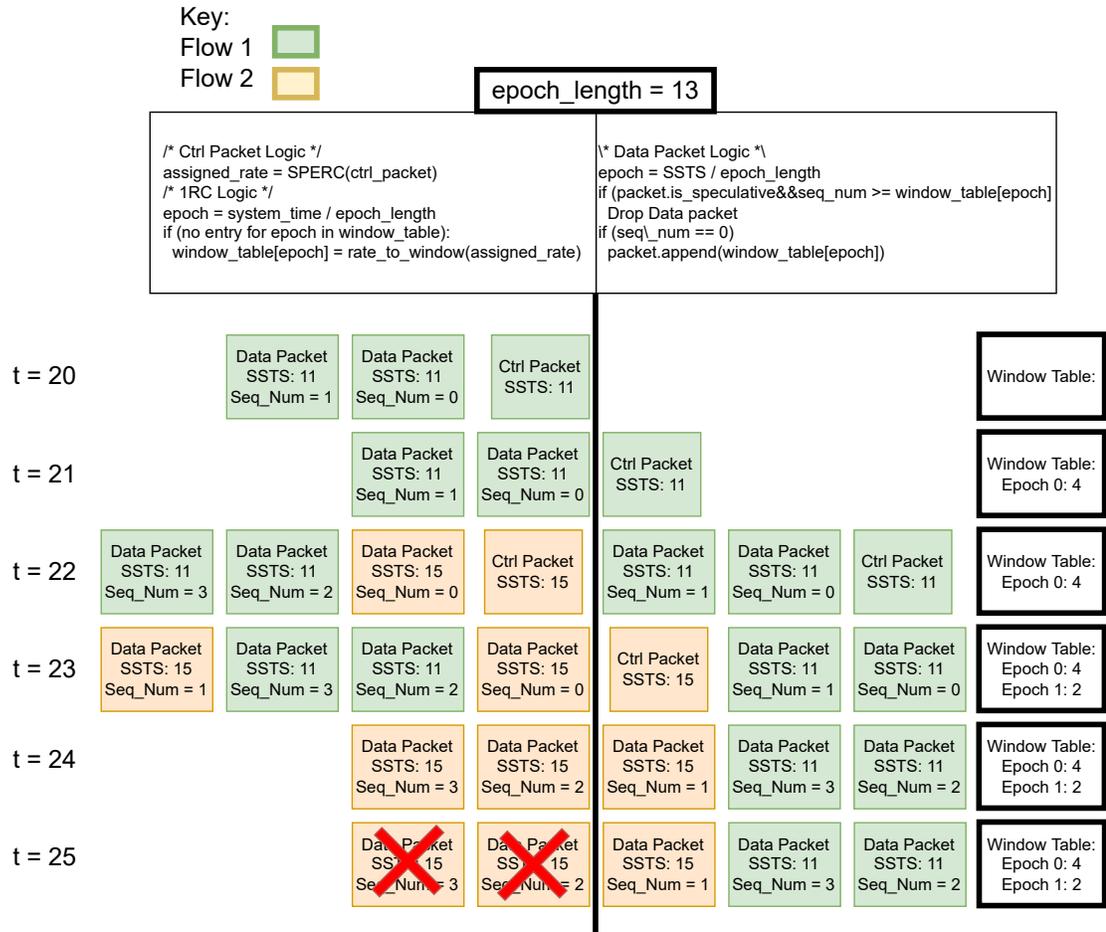


FIGURE 5.1: 1RC example demonstrating how 1RC generates window table entries, indexes the window table, and drops packets.

Figure 5.1 demonstrates how 1RC works. In this example there are two flows, flow 1 and flow 2, that start in timestep 11 and 15 respectively (not shown). In timestep 20 ($t=20$), no flow uses the switch and there are no *window table* entries. The vertical line delineates when 1RC runs in the switch pipeline. At timestep 21

($t=21$), the control packet from flow 1 arrives, and the switch runs the control packet logic and, because there is no entry in the *window table* for the control packet's SSTS, 1RC creates a new entry in the window table. Since there are no other flows, the rate assigned to that control packet is the maximum rate and, in this example, corresponds to a 4 packet window. In the next timestep ($t=22$), two speculative data packets with sequence numbers 0 and 1 and a SSTS of 11 trigger the switch logic. They index the *window table* at *epoch 0* ($\text{window table entry} = 11 \text{ div } 13$) and because their sequence number is less than the allowed packets through, the switch drops neither packet. In timestep 23 ($t=23$), flow 2's first control packet arrives at the switch and triggers the control packet logic. The flow 2's SSTS is 15 and because the *epoch* length is 13, the switch creates a new *window table* entry because flow 2's first packet belongs to a new *epoch*. Now 2 flows use the switch, so SPERC assigns flow 2 a rate twice as small as flow 1. Because the rate is twice as small, the switch only allows 2 speculative packets through if the flow started in *epoch 1*. In timestep 24 ($t=24$), the switch allows more data packets from flows 1 and 2 through the switch because their sequence numbers are less than the *window table* entries for their respective epochs. However, in timestep 25 ($t=25$), packets from flow 2 arrive with sequence numbers 2 and 3 arrive at the switch logic, and the switch drops them (denoted with the red X's) because their sequence number is greater than or equal to the *window table* entry for their SSTS.

Notice that 1RC did not require the switch to maintain per-flow state, only compute each speculative packets *epoch* based on the SSTS. All speculative packets from the same flow have the same SSTS by definition because the SSTS indicates when the flow started. Because all speculative packets from the same flow index into the same entry into the *window table* and the switch drops based on sequence number, which denotes the packet order, the switch maintains packet ordering when dropping packets.

1RC reduces the effectiveness of the staggered QP and shuffled overlay performance isolation attacks identified in Chapter 3 because the network drops a flow's packet if letting that packet through gives the new flow too much bandwidth. Flows start at line rate, so small flows complete quickly in the common case that the network is not congested. However, if letting all the flow's speculative packets through allocates the flow too much bandwidth, the switch drops the extra packets, so the flow receives its fair share. 1RC allows flow's to start sending packets at line rate but prevents a switch from allocating a new flow an unfair amount of bandwidth. We demonstrate 1RC's effectiveness in Section 5.5.1.

5.3 Setting Weights

In Chapter 3, we found that opening multiple connections simultaneously in IB and RoCE gives a user more bandwidth. In the simplest case where a user opens multiple connections between a single source-destination pair, simply enforcing congestion control on a per-src/dst granularity removes any advantage. However, some application communication patterns enable multiple equal cost communication paths, and if a user sends along all the equal cost paths simultaneously, they can get more bandwidth than they otherwise would.

To prevent users improving their bandwidth allocation by opening more connections, NICs set a flow's weight so its allocation reduces as the source opens more connections. This does not render opening extra QPs useless, but it does reduce a user's bandwidth allocation. Traditionally, weights are set to give more bandwidth to certain flows. A higher priority flow's weight may be ten, while all other flows' weight is one. This gives the higher priority flow with a higher weight ten times as much bandwidth as the competing flows.

Flow's are allocated bandwidth with the following equation

$$A_i = w_i * \frac{L}{\sum_{j=1}^N w_j}$$

where flow i 's allocation (A_i) is a function of flow i 's weight (w_i) among N competing flows for a link with bandwidth L .

We set a flow's weight to $\frac{1}{\text{open connections}}$. Using this method, increasing an end hosts open connections decreases the bandwidth allocation to each flow on the end host. This discourages a user from opening more connections to gain more bandwidth because each flow receives less bandwidth.

Ideally, weights would completely disincentive someone from sending along multiple overlays simultaneously (parallel shuffled overlay attack). Unfortunately that is not the case. We prove that it does not solve the issue with a counter example. Suppose an end host 1 opens one flow along link i that has bandwidth L ; end host 1's allocation along link i is

$$A_{i,1} = \frac{L}{W_i + 1} \tag{5.1}$$

where W_i is the weight sum of all other flows using link i . Now suppose end host 1 opens a second connection along link j that is equally in demand so $W_j = W_i$. The total bandwidth allocated to both flows is both their allocations summed when both flow's weights are $\frac{1}{2}$. We represent the sum of their allocation with the equation

$$A_{i,1} + A_{j,1} = \frac{1}{2} * \frac{L}{W_i + \frac{1}{2}} + \frac{1}{2} * \frac{L}{W_j + \frac{1}{2}} \tag{5.2}$$

where $A_{i,j}$ is end host j 's allocation along path i . We want to show that a users total bandwidth allocation between all their flows improves when they open multiple connections, which we represent with the inequality

$$A_{i,1} + A_{j,1} > A_{i,1} \tag{5.3}$$

Because W_j and W_i are equal, we can substitute. If we substitute Equation 5.1 and 5.2 into Equation 5.3, we get

$$\frac{L}{W_i + \frac{1}{2}} > \frac{L}{W_i + 1} \quad (5.4)$$

which reduces to

$$1 > \frac{1}{2} \quad (5.5)$$

which proves that when a users opens a second connection on a different path that is equally in demand, the user receives a greater total bandwidth allocation. Even though this weighting method does not completely mitigate the benefits of the parallel shuffled overlay attack, it does reduces its effectiveness because each flow receives less bandwidth.

Our weighting scheme does not solve the parallel shuffled overlay attack because a weight of $\frac{1}{2}$ allocates a flow half the bandwidth of competing flows, not half the bandwidth that the flow was originally allocated. For example suppose flow 1 and 2 use link i . Both allocations are

$$A = \frac{L}{2} \quad (5.6)$$

To solve the parallel shuffled overlay attack, opening a second connection must halve a flow's bandwidth allocation, but if we open a second connection on flow 1's end host, flow 1's weight becomes $\frac{1}{2}$, and, flow 1's allocation becomes

$$A_i = \frac{1}{2} \frac{L}{\frac{3}{2}} = \frac{L}{3} \quad (5.7)$$

but we want the allocation to be

$$A_i = \frac{L}{4} \quad (5.8)$$

We explored assigning weights to get the desired behavior. We want flow 1's allocation with c flows to be $\frac{1}{c}$ times its allocation with 1 flow. Formally,

$$A_{1,c} = \frac{1}{c} * A_{1,1} \quad (5.9)$$

where $A_{i,j}$ is end host i 's allocation with c connections. To determine the proper weight, we must solve the equation

$$w_i * \frac{L}{W + w_i} = \frac{1}{c} * \frac{L}{W + 1} \quad (5.10)$$

for w_i . W is the weight sum excluding i . Solving for w_i , we get

$$w_i = \frac{W}{c(W + 1) - 1} \quad (5.11)$$

so the weight for any flow w_i depends on the number of connections (c) on the source and the weight sum W .

On the surface, this seems easy to calculate. A control packet for flow i could carry c and the switch knows the weight sum, W . However, anytime we update the weight for one flow, the weight sum changes for another flow. To solve this for all flows, the switch would need to know the c_i for each flow i and solve a large system of equations. This would require per-flow state on the switch (storing c for all flows) and the switch to perform a vast amount of computation (solving the large system of equations). For these reasons, we choose the sub-optimal solution of setting the weight equal to the number of open connections. While this does not solve the parallel shuffled overlay attack, it does limit its effectiveness.

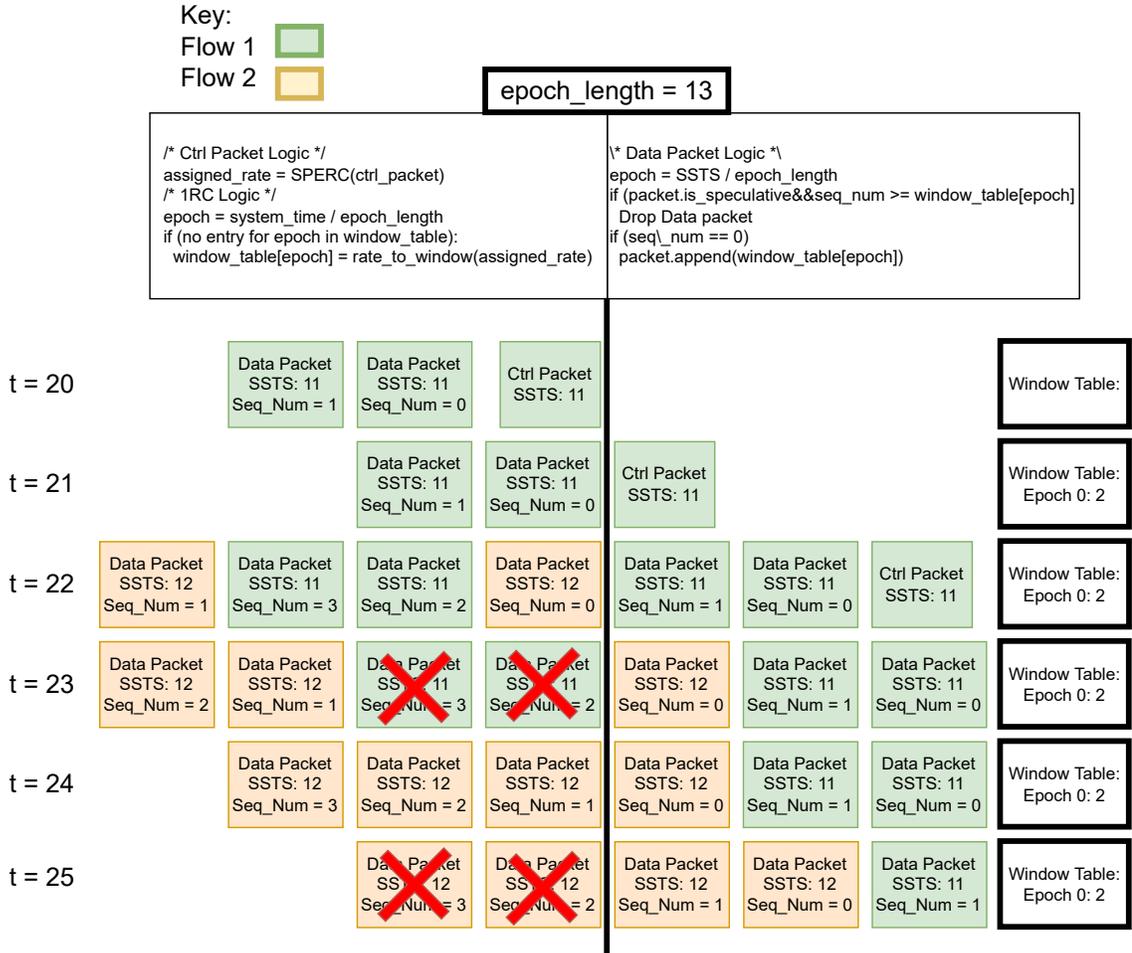


FIGURE 5.2: Flows 1 and 2 belong to the same end host. Starting two flows in the same Epoch enables an end host to send more speculative packets.

5.3.1 1RC, Weights, and Fairness

1RC's original goal was solving the staggered attack as described in Chapter 3, where a user sends along a new QP each BDP. This allows a user to send at line rate and minimizes the QP's used during the attack. 1RC mitigates the staggered attack in this context because the network drops a misbehaving flow's packets. However, if we use 1RC as described in Section 5.2, a malicious user can still gain extra bandwidth. Figure 5.2 demonstrates how a malicious user can gain more bandwidth by exploiting speculative packets. In Figure 5.2, we have a network with a BDP of 4 packets and

the fair rate for a new flow is 2 packets. When flow 1 in Figure 5.2 starts, the network allows 2 packets for the new flow through. However, user 1 is malicious and starts a second flow in the same *epoch*, but slightly after the first flow. Since 1RC drops packets based on sequence numbers and a new flow has different sequence numbers, 1RC allows 2 packets from both flow 1 and flow 2 onto the switch. This allocates the misbehaving user too much bandwidth.

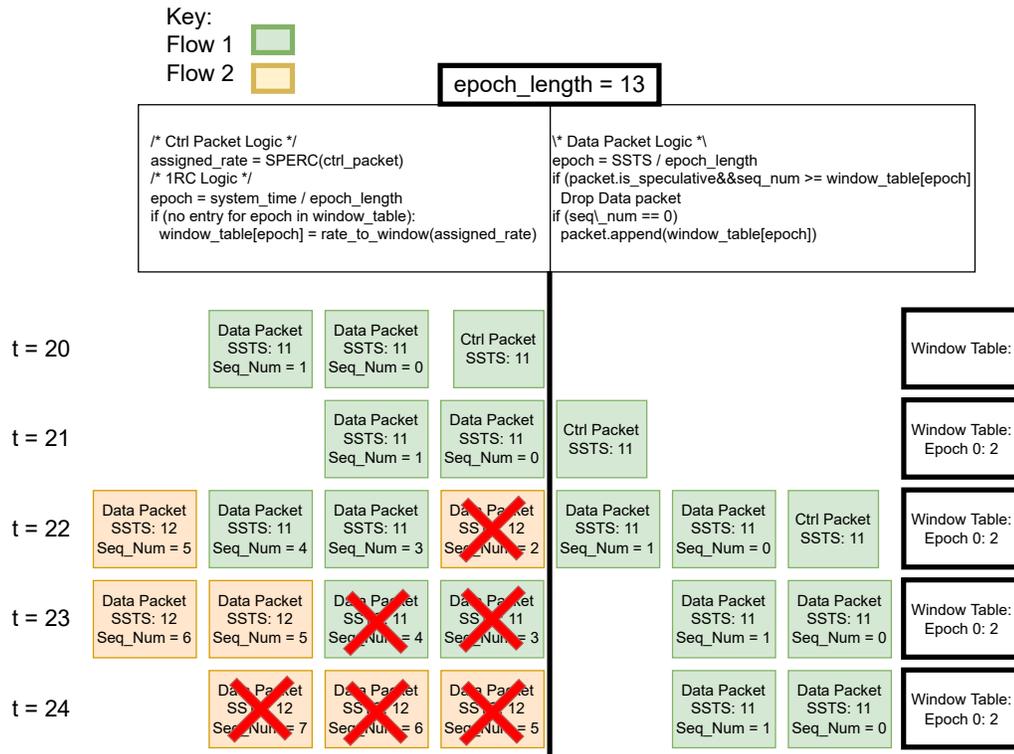


FIGURE 5.3: Flows 1 and 2 belong to the same end host. The flows from the same end host share speculative sequence numbers.

Using flow weights helps because we can multiply the number of speculative packets allowed through by the flows weight, which is the inverse of the number of concurrent connections. While this reduces the user's bandwidth, 1RC still lets too many packets through. When the first flow started, there is only one open connection, 1RC allows 2 of the first flow through. The second flow's weight is two, so 1RC allows

1 packets through. Together, 1RC allows 3 speculative packets through for the new flows in that epoch, which is still too many (2 is the goal).

To further mitigate a malicious users ability to gain more bandwidth, flows starting in the same *epoch* share speculative packet sequence numbers, which Figure 5.3 demonstrates. Using this method, either flow sending in that *epoch* increments the sequence number and together they can only get 2 speculative packets through, which is that end hosts fair rate.

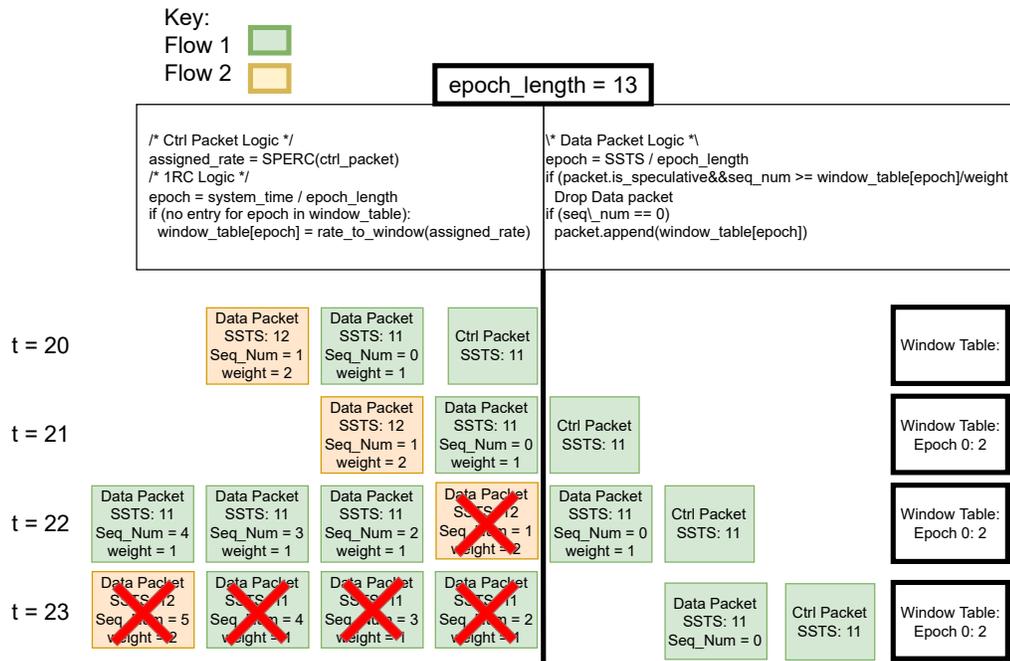


FIGURE 5.4: Flows 1 and 2 belong to the same end host. Flow 2’s weight increases because flow 1 already existed when it started.

The weight 1RC uses to determine the number of speculative packets let through is the number of concurrent connections at the start of the *epoch*. This prevents shared speculative packets and flow weights from both throttling speculative packets at the same time. Figure 5.4 shows how using shared sequence numbers and incorrect weights at the same time leads to suboptimal results. Flow 2’s weight is 2 because flow 1 already existed when flow 2 started. If this weight reduces the number of

speculative packets allowed through, 1RC only allows packets with a sequence number less than one through for flow 2. In this example, 1RC only allowed one packet through for both flow 1 and flow 2 combined when it should only allow two packets through because of flow 2's increased weight. By using a weight equal to the number of concurrent flows at the start of the epoch, flow 1 and 2's weight are both 1, and 1RC allows the correct number of packets through.

5.4 Implementing 1RC and Weights in s-PERC

In the previous sections, we explain how 1RC and our weights converge to fair rates during the first RTT and limit a malicious users ability to unfairly get more bandwidth. Now we explain how we use 1RC and weights in s-PERC.

Algorithm 4 shows the complete s-PERC algorithm with our changes marked in red. 1RC and adding weights requires four new fields in the s-PERC control header 1) *let_through*, 2) *old_weight*, 3) *new_weight*, and 4) when the end host sent the control packet (timestamp). *let_through* notifies the end host how many of the new flow's speculative packets the switch allows through. This enable an end host to determine which packet is sent next. *let_through* notifies the end host which packets the switch dropped, so the end hosts do not require NACKs or timeouts.

Packets carry a flow's weight from the previous RTT (*old_weight*) and the current RTT (*new_weight*). The weights in the packet header correspond to the number of open connections on the end host, and the switch calculates the actual weight taking the inverse of the packet's weight field. Lines 12 and 13 update the switch state when the flow's weight changes. We subtract out the old weight from the total weight and add in the new weight. Lines 10 and 31 reduce a flows allocation based on the flow's weight $x[l]$ on line 10 and x on line 33. When setting the injection rate at the end host, the end host divides the injection rate by the flows *new_weight* if a switch bottlenecks the flow. A flow's weight is the number of active flows on the

Algorithm 4 s-PERC with IRC and Weights

1: \mathbf{b} , \mathbf{x} , \mathbf{s} : vector of bottleneck, allocated rates, and bottleneck states in packet (initially ∞ , 0, E, respectively).
2: *let_through*: minimum number of speculative packets let through any switch
3: *old_weight*, *new_weight*: The flows weight in the previous RTT and current RTT.

4: *timestamp*: Time when end host sent control packet
5: \mathbf{i} : vector ignore bits in a packet (initially, 1)
6: *SumE*, *NumB*: sum of limit rates of E flows, and number of B flows at link
7: *MaxE*, *MaxE'*: max. allocated rate of flows classified into E since last round (and in this round, respectively) at link.
8: **if** $\mathbf{s}[l] = E$ **then**
9: $\mathbf{s}[l] \leftarrow B$
10: $SumE = SumE - (\mathbf{x}[l] / \text{old_weight})$
11: $NumB \leftarrow NumB + 1 / \text{new_weight}$
12: **else if** $\text{old_weight} \neq \text{new_weight} \ \&\& \ \mathbf{s}[l] \neq \text{NEW_FLOW}$ **then**
13: $NumB \leftarrow NumB - 1 / \text{old_weight} + 1 / \text{new_weight}$
14: $\mathbf{b} \leftarrow (\mathbf{c} - SumE) / NumB$
15: **if** no entry for $\text{window_table}[\text{timestamp} / \text{epoch_length}]$ **then**
16: $\text{window_table}[\text{timestamp} / \text{epoch_length}] = \text{rate_to_window}(\mathbf{b})$
17: **if** $\mathbf{s}[l] = \text{NEW_FLOW}$ **then**
18: $\text{epoch_entry} = \text{window_table}[\text{timestamp} / \text{epoch_length}] / \mathbf{p} \rightarrow \text{new_weight}$
19: $\text{let_through} = \max(1, \min(\text{epoch_entry}, \text{let_through}))$
20: **for** each link j **do**
21: **if** $\mathbf{i}[j] = 0$ **then** $\mathbf{p}[j] \leftarrow \mathbf{b}[j]$ **else** $\mathbf{p}[j] \leftarrow \infty$
22: $\mathbf{p}[l] \leftarrow \infty$
23: $e \leftarrow \min \mathbf{p}$
24: $x \leftarrow \min(b, e)$
25: $\mathbf{b}[l] \leftarrow b, \mathbf{x}[l] \leftarrow x$
26: **if** $b < MaxE$ **then** $\mathbf{i}[l] \leftarrow 1$ **else** $\mathbf{i}[l] \leftarrow 0$
27: **if** flow is leaving **then**
28: $NumB \leftarrow NumB - 1 / \text{flow_weight}$
29: **else if** $e < b$ **then**
30: $\mathbf{s}[l] \leftarrow E$
31: $SumE \leftarrow SumE + x / \text{flow_weight}$
32: $NumB \leftarrow NumB - 1 / \text{flow_weight}$
33: $MaxE \leftarrow \max(x, MaxE)$; $MaxE' \leftarrow \max(x, MaxE')$

flow’s source end host. Each end host simply tracks its number of open flows and increments the counter when a new flow is created and decrements the counter when a flow finishes.

We do not enforce the increase weight and the shared sequence numbers at the same time. This would prevent the correct number of speculative packets from making it through the switch because a speculative flow would have inflated sequence numbers and a reduced number of speculative packets allowed through.

Algorithm 5 s-PERC and 1RC Speculative Packet Logic

```

1: epoch = p.SSTS / epoch_length
2: let_through_thresh = max(1, window_table[epoch] / p.weight)
3: if p.is_speculative && p.seq_num ≥ let_through_thresh then
4:   Drop p

```

To implement 1RC, s-PERC simply checks if the packet’s SSTS entered a new *epoch* (line 15) and if there is a new epoch, it adds a new entry into the *window_table* that maps the new *epoch* to a certain number of speculative packets. Algorithm 4 only shows the changes to control packets. Algorithm 5 shows how 1RC handles data packets. Line 1 calculates the *epoch*. Line 2 checks the sequence number, and if that is the first speculative data packet for a new flow, the switch records the minimum speculative packets that flow got through any switch so far. On line 4, the switch checks if the packet is speculative and if the sequence number is greater than the *window table* entry for that flow’s epoch. 1RC also divides the *window table* entry by the flow’s weight because we want to enforce the weight for the new flow. If the packet is speculative and the sequence number is too high, the switch drops the packet.

One potential issue arises if there is no *window table* entry for a speculative packet’s epoch. This can never happen because a flow sends its first control packet before the flow sends any speculative packets and the timestamp on the first control packet is equal to the speculative packet’s SSTS, so the control packet creates a

window_table entry for the speculative packets. We are currently investigating how many *window_table* entries are necessary. We will need some sort of garbage collection mechanism to remove entries and a policy to handle the case where a flow from a garbage collected *epoch* arrives at the switch.

To support 1RC, the switch needs all the capabilities required for s-PERC and some additional modifications. The switch must determine if packets are speculative by reading the packet header. The switch also requires storage for the window table, which should only be a few kilobytes.

5.5 Evaluations

We evaluate 1RC and our weighting scheme in two contexts. First, we show that as network BDP grows 1RC enables a congestion control protocol to manage network congestion. Next, we show that using speculative packets in conjunction with how we set weights prevents a user from opening extra connections to gain an advantage over other users. We evaluate 1RC using a simulated datacenter workload and demonstrate that 1RC effectively manages congestion while still ensuring congestion does not impact short flows. We evaluate performance isolation with microbenchmarks and show how a misbehaving user does not receive more bandwidth than a well-behaved user. Finally, we show that a misbehaving user does not cause congestion in a datacenter workload.

5.5.1 1RC Performance Evaluations

When evaluating the performance improvements with 1RC, we use the same datacenter topology used in Chapter 3 and 4. However, we increase link speed by $16\times$ to simulate future networks. The end host NIC speeds increase from 100Gb/s to 1.6Tb/s and inter-switch connections increase from 400Gb/s to 6.4Tb/s. We evaluate 1RC's performance benefits with an Alibaba storage workload because all flows

are smaller than BDP in our faster network. This demonstrates how workloads with small flows evade congestion control as BDP grows. We run the workload in ns-3 at 70% load for 10ms.

We evaluate three different congestion control methods. First, we run the original s-PERC without any modification. s-PERC manages congestion poorly because all flows less than BDP are exempt from congestion control, so they can complete quickly. However, in this benchmark, all flows are less than BDP, so s-PERC does not manage congestion. We then run s-PERC again and lower the scheduling threshold to 153KB, so any flow longer than 153KB cannot start sending packets until its first control packet returns. We chose 153KB because that is the threshold (BDP) in the smaller 100Gb/s network. This incurs scheduling overhead for flows larger than 153KB. Finally, we run a modified version of s-PERC that includes 1RC and show that it maintains high throughput for long flows, while still maintaining small queues, so short flows complete quickly.

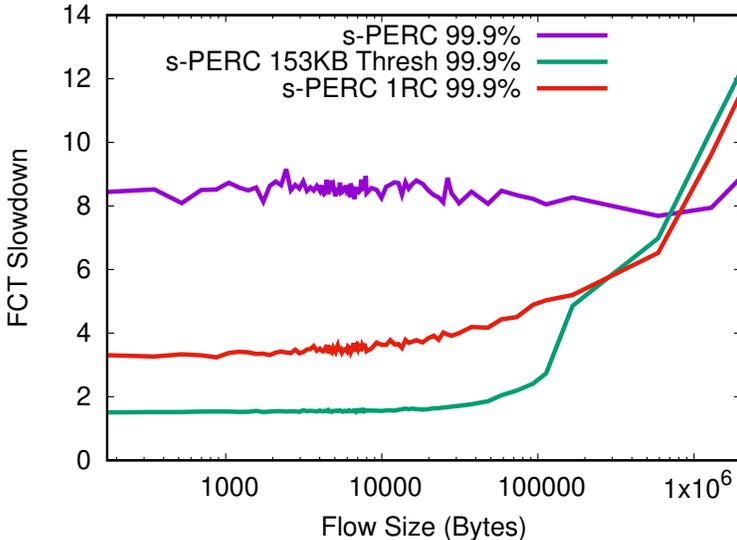


FIGURE 5.5: Alibaba Storage 99.9% tail latency using 1RC alongside s-PERC.

Figure 5.5 reports the 99.9% FCT slowdown from the three experiments. As

expected, using default s-PERC leads to severe congestion, which causes short flows to complete eight time longer than they would in a non-congested network. When using 1RC alongside s-PERC, congestion is minimal, and small flow FCT slowdown reduces from about eight to around three and a half. This is a significant reduction in tail latency. The best performing in terms of short flow tail latency is s-PERC with a 153KB threshold. This is likely because 1RC allows some packets through for long flows during the first RTT, while s-PERC with a threshold schedules the flow before sending any packets. The speculative packets let through the network cause a small amount of congestion. Using 1RC and the threshold have similar performance for flows longer than 153KB.

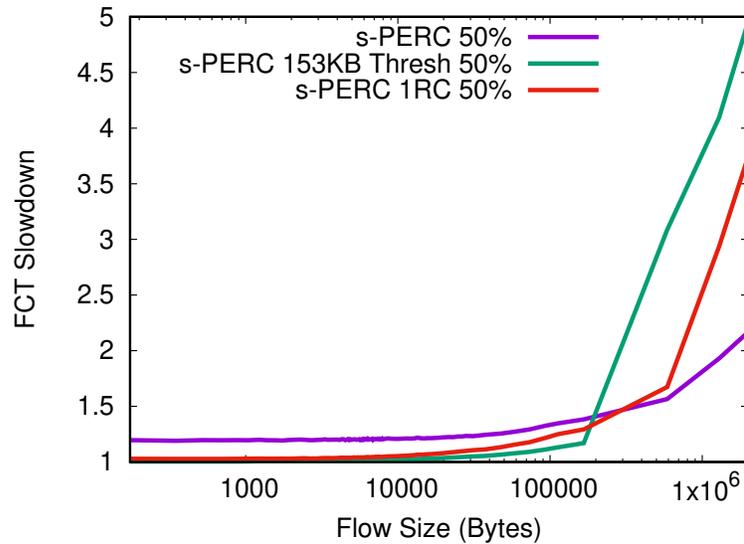


FIGURE 5.6: Alibaba Storage 50% tail latency using 1RC alongside s-PERC.

The advantage of 1RC over a threshold becomes apparent when we look at the median FCT slowdown, which Figure 5.6. For small flows, 1RC and threshold perform similarly. However, as flow size exceeds 153KB, 1RC significantly outperforms threshold. 1RC increases throughput for flows larger than 153KB by 38% over the 153KB threshold version. These experiments show that 1RC allows short flows to

complete quickly in high BDP networks while still maintaining high throughput.

5.5.2 Performance Isolation Evaluation

We now evaluate how well 1RC and our weight assignment method isolates the performance. First, we show with a microbenchmark that a misbehaving user reduces the throughput of a well behaved user, but that our mechanisms remove this negative impact. Next we demonstrate how a misbehaving user increases its performance, but that our mechanisms remove the misbehaving user’s performance gains. Finally, we show that, in a datacenter workload, a misbehaving user running a large workload does not cause congestion in the network, so tenants sharing that network would not experience congestion due to the performance isolation attacks.

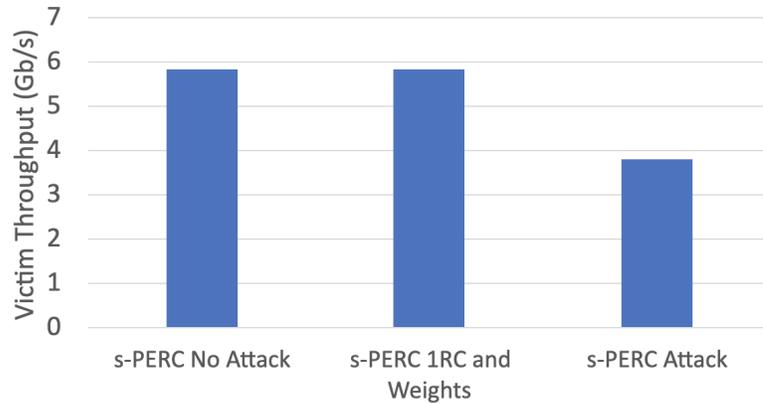


FIGURE 5.7: Impact of staggered attack on network performance in 16-1 Incast benchmark average victim BW with and without 1RC and Weights

For the microbenchmarks, we use the same microbenchmark topology as in Chapters 3 and 4, and a similar incast benchmark. 16 senders all send packets to a single destination at the same time. We then choose one flow to be the “attacker”, which performs the staggered attack by starting a new flow every RTT and sends BDP bytes along each new flow. The “attacker” sends 10MB and all other flows send 1MB. We show that an attacker reduces the victim flows’ bandwidth when using

default s-PERC. We then show that when we use 1RC and weights, the offending flow no longer negatively impacts the victim flows. Figure 5.7 shows that without an attack and using s-PERC, flows average throughput around 6Gb/s. When one flow attacks the network, the victim throughput drops to below 4Gb/s. Adding 1RC and weights isolates the victims' and attacker's performance, and restores the victims' throughput. Our simulated switches do FIFO queueing opposed to round robin. In this exact experiment, round robin arbitration between input ports would remove the degraded performance because each flow has its own ingress queue. However, in a more complex topology where flow's share queues, round robin arbitration would not solve the problem.

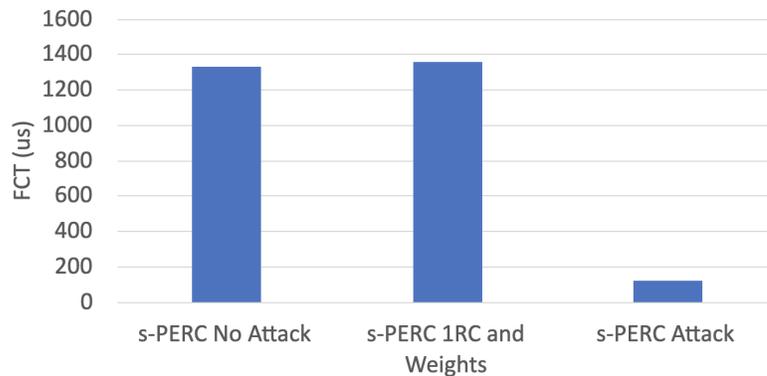


FIGURE 5.8: Attacking Flow FCT in 16-1 Incast with and without 1RC and Weights (lower is better)

In the next microbenchmark, we show that a misbehaving user gains throughput by attacking the network and that 1RC and weights mitigate the performance gains. In this experiment, we show that the attacking flow completes far more quickly than competing flows when using default s-PERC, and that using 1RC and weights removes the attacker's performance benefits. This experiment has 16 flows, and they all send 1MB to the same destination. One user attacks the network and performs the staggered attack by breaking its 1MB into several smaller flows equal to BDP

spaced one RTT apart. Figure 5.8 demonstrates that when using default s-PERC, the attacker’s performance greatly improves. When we run the attack using 1RC and weights, the attack loses all performance benefit and performs nearly the same as when it does not attack the network. Therefore, there is no incentive for the user to attack the network. In summary, the microbenchmarks show that an attacking user limits victim flow’s throughput and improves the attacker’s throughput. We then demonstrate how adding our performance isolation mechanisms, 1RC and weights, removes the performance gain and restores victim flow throughput.

Next, we show that 1RC and weights mitigate congestion caused by misbehaving users in a simulated datacenter. We run the same Facebook Hadoop datacenter workload [84] for 50ms on the 100Gb/s network with a fat-tree topology from Chapters 3 and 4. We run the experiment with s-PERC with and without our mechanisms. s-PERC is the baseline and demonstrates how s-PERC handles traffic without our mechanisms. s-PERC with 1RC and weights shows how our mechanisms affect performance when no one attacks the network. Ideally, our mechanisms would not degrade performance relative to the default s-PERC baseline. We then run simulations where a misbehaving user performs the staggered attack. To simulate the staggered attack, we take any flow longer than 1MB and divide it into smaller flows that each send BDP each and space the new, smaller flows an RTT apart. We simulate an attack in both default s-PERC and s-PERC with our mechanisms.

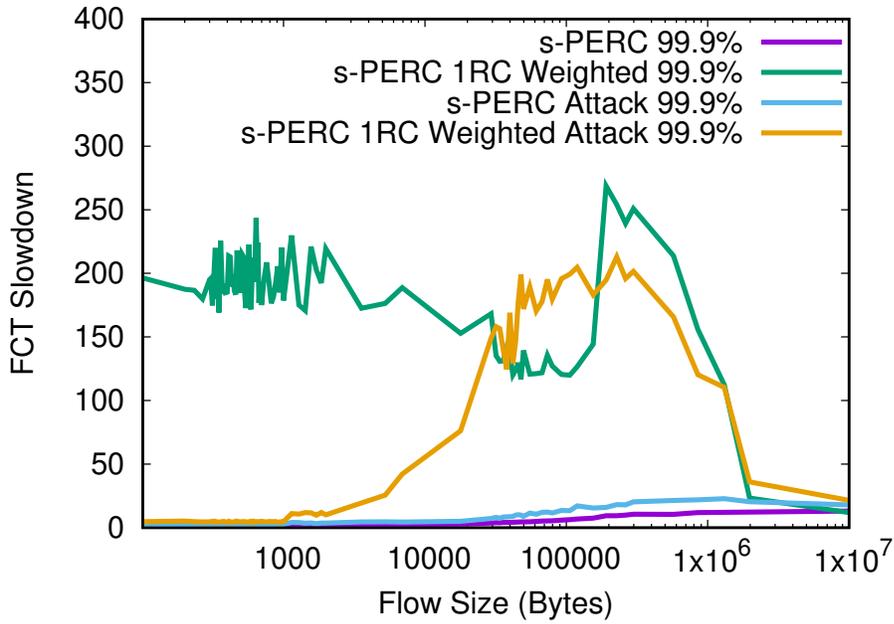


FIGURE 5.9: s-PERC with and without 1RC and weights simulating Hadoop traffic in a datacenter with and without staggered attacks

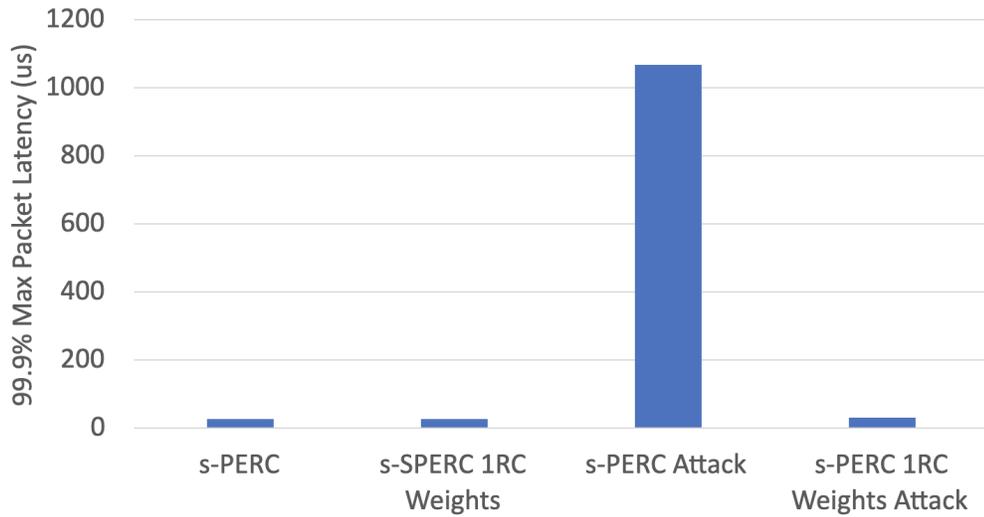


FIGURE 5.10: Packet latency during staggered attacks using s-PERC variants

Figure 5.9 demonstrates that in the absence of attacks s-PERC and s-PERC with 1RC and weights perform well. s-PERC with 1RC and weights performs slightly worse for longer flows, likely due to flow's co-located on the same end host reducing

each others weight. When long flows perform the staggered attack in default s-PERC, short tail FCT increases dramatically due to increased congestion. To support this claim, we measured packet latency in addition to FCT. To measure packet latency, we record each flow's max packet latency for all packets in the flow. Of these max packet latencies, we then took the 99.9% percentile latency. Figure 5.10 shows that s-PERC's packet latencies during the attack exceeds 1ms. Meanwhile, when using 1RC and weights, latencies are about $26\mu\text{s}$ during the attack. When a user performs the staggered attack, latencies do increase by about $5\mu\text{s}$ when using our mechanisms, but this is a significant improvement over default s-PERC. Congestion is one way that applications interfere with one another. By significantly reducing congestion during an attack, 1RC and weights improve performance isolation.

The attack also sees less of an increase in throughput when using our mechanisms. When using default s-PERC, the attack increases throughput of flows longer 1MB by 37%. However, when using 1RC and weights, the throughput of flows longer than 1MB only increases by 20%. We are currently investigating why there is any increase during the attack when using 1RC and our weighting scheme.

When using our mechanisms, flows from about 10KB to 1MB see a dramatic increase in FCT slowdown. We hypothesize that this is not due to a poor bandwidth allocation by switches, but caused by the a flow being co-located with one or several attacking flows that decrease the weight of all flows on the end host. To support this claim, we record each flow's average weight over its lifetime and multiply it by its FCT slowdown. If a flow has many co-located flows on the end host, it has a low weight, and multiplying the weight by the FCT slowdown should yield a much smaller value.

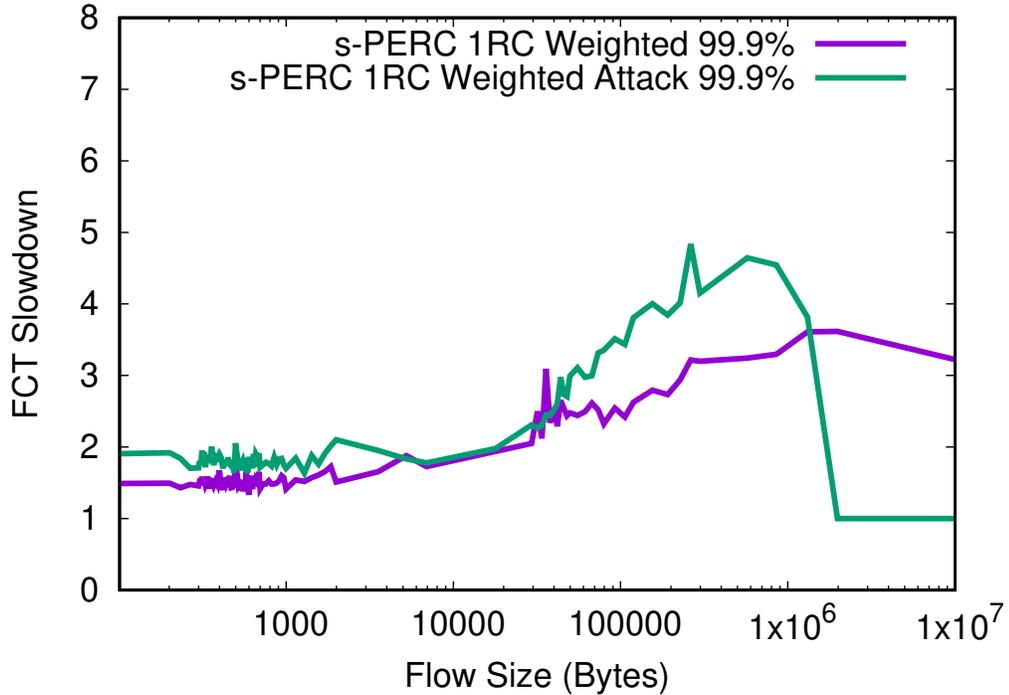


FIGURE 5.11: Hadoop 99.9% FCT slowdown multiplied by weight

Figure 5.11 shows the 99.9% FCT slowdown when multiplied by a flows weight. Without multiplying by the flow’s weight, the FCT slowdown for flows between 10KB and 1MB is around 100x. However, when we multiply by the flows weight, the 99.9% FCT slowdown reduces to less than 5x. This indicates that the flows have many co-located flows on the end host, likely due to the user attacking the network. Therefore the weights are working correctly because the goal of the weights is throttling rates of end hosts that open many connection.

In this section, we show that in default s-PERC a user gets significantly more bandwidth when performing the staggered attack in microbenchmarks, which also reduces the bandwidth allocated to well-behaved users. We then demonstrate that when using 1RC and weights, a misbehaving flow does not take bandwidth away from other users and performing the attack yields no performance gains. In a simulated datacenter workload, we show that without our mechanisms the staggered attack

creates significant congestion, which would interfere with other applications. Using our mechanisms, the increased congestion during the attack is marginal and would isolate application performance well.

There are currently some issues with using weights in the datacenter workloads. Each new connection decreases existing an flow’s weight on the same end host, even if the new connection needs little bandwidth. Therefore workloads with a large number of small flows can decrease overall system throughput even when the user is well-behaved. We are currently examining how we can mitigate this issue and maintain high performance in all workloads.

5.6 Conclusion

As BDP increases and sharing becomes more common in RDMA networks, we must reexamine congestion control. In this Chapter, we introduce 1RC and a new weighting scheme that improve performance in high BDP networks and mitigate performance isolation attacks. We show in datacenter simulations that existing congestion control in high BDP networks fails to manage congestion and that using 1RC enables high throughput while maintaining low congestion levels. Next, we show that performance isolation attacks give a misbehaving user more bandwidth than well-behaved users. Using our mechanisms restores fairness to the network and gives all users, well-behaved and mis-behaved, equal bandwidth allocations. Further, we demonstrate that in a datacenter simulation a misbehaving user can cause severe congestion and interfere with other applications. 1RC and our weight scheme significantly decrease the congestion caused by misbehaving users, which helps ensure isolation between users.

6

Multicast Congestion Control

Traditionally, datacenter networks are designed for unicast traffic where each packet has a single source and destination. However, many applications from HPC [64] and machine learning (ML) [8] use collective operations to transfer data. During a collective operation, a group of end hosts all communicate together. While unicast traffic can implement collective operations, multicast traffic can accelerate collective communication. Multicast packets allow a single end host to send packets to several destinations simultaneously. Networks can accelerate multicast operations by replicating and reducing multicast packets in the network to decrease load and increase effective bandwidth. To accelerate collective communication, researchers created multiple implementations of in-network collectives, such as SHArP[39], Klenk et al. [65], ATP [67], SwitchML [97], and Panama [36]. These networks can increase the throughput of large all-reduce operation by up to 2x and improve latency sensitive small collectives even more drastically.

While in-network collectives can provide large performance gains, this potential is only realized when network congestion does not severely limit achieved bandwidth and increase packet latency. Congestion control (CC) mechanisms *ideally* mitigate

the congestion problem before the level of congestion creates a noticeable bottleneck. Sources throttling packet injection rates can help the network avoid congestion, however numerous problems must be solved to achieve this *ideal* network.

Existing mechanisms like round-trip time (RTT) measurements and ECN markings simplify congestion detection. The challenge is incorporating *local* congestion observations into a global understanding of how to converge to optimal injection rates. Existing end-point based CC mechanisms [110, 66, 75] use additive-increase multiplicative-decrease (AIMD) algorithms to probe for optimal injection rates, which is proven to be fair [19]. This poses two issues. First, congestion control information is inherently delayed because congestion information takes time to propagate through the network. While the information propagates, the congestion may cause performance degradation [104]. Second, AIMD algorithms converge slowly to optimal injection rates. While servers determine their optimal injection rates, the network may still be congested or under-utilized, leading to sub-optimal performance.

In-network collectives exacerbate both of these issues because multicast operations expand in the network and can cause congestion in several places quickly. To manage congestion in multicast networks, we introduce a new multicast congestion control protocol. Coupling an existing in-network collectives with a novel collective aware CC mechanism significantly reduces congestion issues in multicast networks by converging quickly to near-optimal rates and achieving near ideal network performance.

Our network performs in-network collectives with an architecture similar to Klenk et al.'s [65] where users implement collective operations in a memory fabric that supports multicast writes and read reductions. To avoid congestion, Klenk et al. statically limit the number of outstanding multicast operations in the network at any given time. Unfortunately, this approach can underutilize the network when the window does not match the dynamic network state because a static congestion

window does not adjust as network traffic characteristics change.

Ideally, we want a congestion control protocol that converges quickly, avoids multiple round-trips to probe network state, and provides high network utilization as network state changes. We observe that multicast operations exhibit unique properties that we exploit to meet this challenge. Specifically, when an end host receives a multicast packet, the receiving end host knows that same packet is replicated on several links in the network. The receiving end host can therefore infer global congestion state based on local information.

We propose the Collective Congestion Control PrOtocol (3CPO), which mitigates congestion in networks that use multicast writes and read reductions to implement collective operations. 3CPO exploits the symmetry of multicast operations to converge to effective injection rates quickly and mitigate congestion while maintaining high throughput. 3CPO works on each end host independently and observes the multicast operations that the end host receives, which informs 3CPO about potential congestion at other points in the network. Since multicast operations do not exist alone in the network, we also develop a method to allow 3CPO to interoperate both multicast CC and unicast CC for networks where both traffic types are present.

When compared to a hand-tuned CC window during an all-reduce operation with 64 end hosts, 3CPO reduces bandwidth by less than 2% and decreases median tail latency by about 200 cycles. We further scale 3CPO to a system with 128 end hosts and observe similar performance. We also evaluate 3CPO in a situation with a dynamic multicast traffic and unicast traffic flows to show that it adjusts injection rates quickly to meet application demands.

6.1 Background and Motivation

3CPO builds on several prior works; this section provides an overview of these foundational techniques. We first present background on collective communication and

the in-network collective architecture that 3CPO builds upon. Swift [66] inspires the unicast congestion control in 3CPO. We give an overview of Swift in Chapter 3. We then explain the injection throttling mechanism in our network. Finally, we demonstrate why existing CC for in-network collectives is insufficient for dynamic workloads or is ineffective.

6.1.1 Collective Communication

Collective communication involves a group of servers that communicate with one another. Collective operations include: 1) barrier, which synchronizes servers in the group; 2) broadcast, where one server sends data to all others in the group; and 3) reduce, where a user applies an operator (Relational, bitwise, arithmetic operations) to a dataset and each server in the group provides a subset of the data. Machine Learning [8] and HPC [64] applications often use collective communication, which makes collectives a common target for acceleration [65, 67, 36, 97, 73].

Klenk et al. proposed a collective communication architecture for shared-memory multiprocessor collectives (SMMC) [65]. Klenk et al. designed SMMC for clusters of accelerators like the Nvidia DGX-2 systems, which use NVLink to communicate between GPUs in a globally shared address space. NVLink allows any GPU to access the memory of any other GPU through writes, reads, and atomic operations. To fit within the communication model of the NVLink network, SMMC added multicast regions to the global address space to create multicast groups. GPUs can register existing memory allocations to the multicast regions and map the multicast regions into their virtual address space to join the multicast group. Network switches use tables to store multicast regions to GPU address mappings that indicate how switches should forward and replicate multicast packets. When any GPU issues a memory access to a multicast region, switches trigger a collective operation to all GPUs registered to the region. A write operation to a multicast region causes the write to

be replicated to all GPUs participating in that multicast. A pull operation (a read with a reduction operator) from a multicast region causes a read request to be sent to all GPUs participating in that multicast, and the reduction operator is applied to all read responses before a single read response is returned to the requesting GPU. 3CPO builds upon SMMC.

6.1.2 *Swift*

Numerous congestion control algorithms exist that work in a variety of networks. Our work focuses on sender-side protocols, where servers throttle the injection of packets based on congestion feedback like ECN marked packets [110], In-band Network Telemetry (INT) [75], and RTT measurements [66]. We use a protocol similar to Swift [66] to limit unicast congestion. Chapter 3 provides background information on Swift.

6.1.3 *Injection Throttling*

Network interfaces use various mechanisms to limit packet injection. For example, Swift [66] limits the number of inflight bytes from each end host, and Infiniband [45] waits a certain number of cycles before injecting a new packet. The existing shared-memory network uses a leaky bucket [94, 26, 106, 18] that contains injection rate limiting (IRL) tokens. Memory request packets consume IRL tokens when they enter the network. End hosts do not send a request packet if there are insufficient tokens. The network interface maintains two types of IRL tokens: response tokens and request tokens. Request packets consume both types of tokens when the requests enter the network. For example, if a write operation has nine request flits and one response flit, the packet consumes nine IRL request tokens and one IRL response token. Response packets do not consume any IRL tokens and by limiting the injection rate of the request packets we indirectly limit the responses.

Without Rate Throttling					With Rate Throttling				
Cycle	Action	Tokens	WM/ WM_Max	New Token?	Cycle	Action	Tokens	WM/ WM_Max	New Token?
0	send packet	1	10/10	Yes	0	send packet	1	5/10	No
1	wait	-3	10/10	Yes	1	wait	-4	5/10	Yes
2	wait	-2	10/10	Yes	2	wait	-3	5/10	No
3	wait	-1	10/10	Yes	3	wait	-3	5/10	Yes
4	wait	0	10/10	Yes	4	wait	-2	5/10	No
5	send	1	10/10	Yes	5	wait	-2	5/10	Yes
6	wait	-3	10/10	Yes	6	wait	-1	5/10	No
7	wait	-2	10/10	Yes	7	wait	-1	5/10	Yes
8	wait	-1	10/10	Yes	8	wait	0	5/10	No
9	wait	0	10/10	Yes	9	wait	0	5/10	Yes
10	send	1	10/10	Yes	10	send	1	5/10	No

FIGURE 6.1: Watermark throttling reducing injection rate of 5-flit packets

GPUs generate IRL tokens over time based on a *watermark*, which corresponds to the injection rate of packets. A high/low *watermark* indicates a high/low injection rate. Each GPU adjusts its *watermark* by observing network conditions. Each cycle, each GPU generates a random number between 0 and WM_MAX, the maximum possible *watermark* value. The GPU generates a new token only if the random number is less than the *watermark*.

Figure 6.1 demonstrates how a lower *watermark* makes a GPU generate fewer tokens. In the table on the left, the *watermark* is 10 and WM_MAX is 10, so the GPU always generates a new token. However, on the right the GPU lowers the *watermark* to 5, so the GPU on average only generates a token every other cycle. This makes the GPU send packets half as often, which lowers the level of congestion in the network. Since the network uses both request and response tokens, the network has request and response *watermarks* that adjust separately. In general request tokens throttle

write operations, which have large request packets and small response packets, and the response tokens throttle read operations, which have small request and large response packets.

The network interface adjusts a GPU’s *watermark* once per RTT based on the observed packet delay in the network. If the network RTT exceeds a certain threshold, the *watermark* reduces multiplicatively. If the network RTT is below that threshold, the *watermark* increases additively. The *watermark* cannot be less than WM_MIN, which must be greater than zero. *3CPO extends this leaky bucket model by tailoring token consumption for collective operations.*

6.1.4 Existing Multicast CC

SMMC statically limits the number of outstanding multicast operations each GPU can have in the network at any given time. For example, the network may limit each GPU to five outstanding write multicasts and five outstanding read reductions each, which limits network congestion.

To demonstrate that static windows are inadequate for SMMC, we simulate a 64 GPU network, and each GPU is connected with a 25Gbps link and are organized in a 2-level fat-tree topology where each ToR switch connects to 8 GPUs and there is no over-subscription. The simulated switch has a latency of 150 cycles and a bandwidth of 25GB/s per port. The switch can sustain all-to-all uniform random traffic at greater than 95% throughput. The switch’s internal architecture is output-queued with two virtual channels to segregate request and response traffic, with the exception of GPU injection channels which have separate VCs for read and write requests. The output queues are sized to handle the round-trip latency between switches. The maximum packet payload is 128B and each packet has a 16B (1 flit) header, so the maximum packet size is 144B. Read requests and write response are single-flit packets. The GPU’s memory system is given a static latency of 180 cycles.

Table 6.1: Tuned Window Size for Different Number of Initiators

Initiators	8	16	32	64
Window Size	24	12	5	3

All 64 GPUs belong to the same multicast group.

We evaluate static window size (Hand-Tuned Window) compared to a baseline configuration without CC (Baseline) and to a simple additive increase, multiplicative decrease protocol (AIMD). AIMD sends a probe packet each RTT and adjusts the *watermark* based on the probe packet latency. If the latency exceeds 5,000 cycles, AIMD reduces the watermark by half. If the packet latency is below 2,000 cycles, AIMD increases the watermark by 50. The multiplicative decrease factor linearly scales from 1 to 0.5 as the packet latency goes from 2,000 to 5,000 cycles. The WM_MAX is 2,468 and the WM_MIN is 100.

We run several experiments, each with a different number of initiators. An initiator is a GPU that issues multicast operations. If there are only 8 initiators, only 8 GPUs send multicast operations, but all 64 GPUs receive the multicast operations. In this experiment, each initiator performs 128 read reductions and 128 write multicasts operations to simulate an all-reduce implemented as a reduce scatter and followed by an all-gather. If there are 64 initiators, all 64 GPUs send multicast operations. We empirically determine each static window size by attempting to maximize throughput while minimizing packet latency. Table 6.1 shows how the window size changes with the number of initiators. As expected, the window size reduces dramatically as we increase the number of initiators. In a real environment, a GPU may not know the number of initiators, so it could not adjust to the correct window size dynamically.

We measure latency in number of cycles and measure network throughput using “effective bandwidth”, which is the “effective” number of flits transferred divided by the number of cycles required to transfer all data. With in-network collectives, a

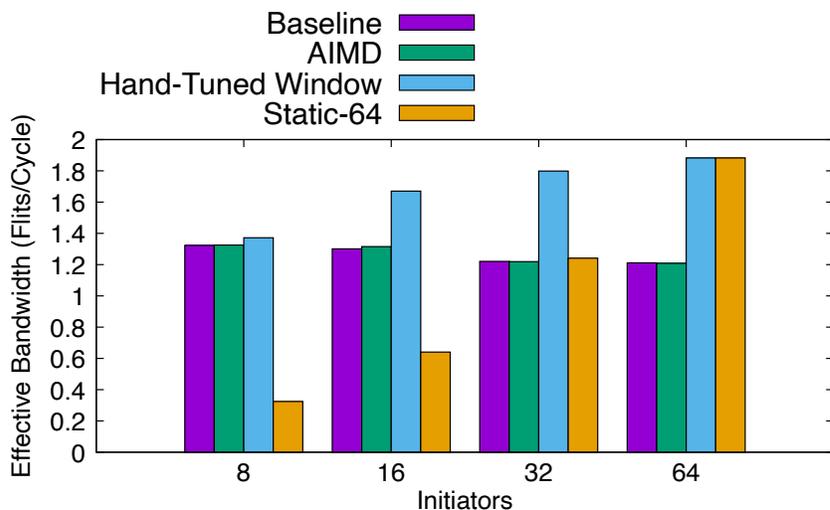


FIGURE 6.2: Effective bandwidth of All-Reduce

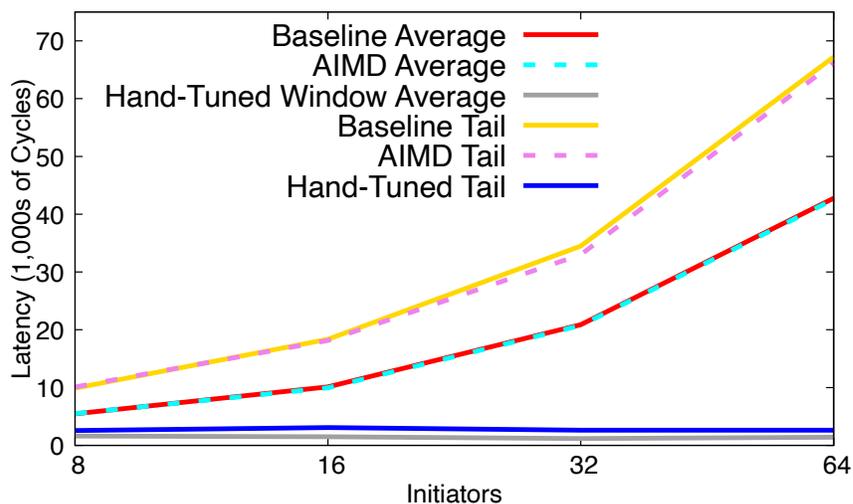


FIGURE 6.3: Average and 100% Tail Packet Latency of All-Reduce

network’s effective bandwidth can be greater than one. For example, if a 64-way read reduction operation completes, only a single read response is sent from the network to the GPU, but the GPU effectively received results of 64 read responses. In the system we model, the maximum effective bandwidth of an all-reduce operation is two and the zero-load latency is just below 1,000 cycles.

Figures 6.2 and 6.3 demonstrates the drastic effect congestion management has on

packet latencies and bandwidth. With a tuned congestion window and 64 initiators, effective bandwidth is nearly two, tail packet latencies are below 5,000 cycles, and average packet latencies are below 3,100 cycles. Meanwhile with no CC, the effective bandwidth drops to about one, and the tail packet latency is close to 70,000 cycles. Effective bandwidth degrades because there is tree saturation when the network is congested, which limits the throughput [60]. AIMD is not effective because it reacts too slowly, taking several RTTs to find the correct injection rate, but the experiment ends before that occurs.

A congestion window is not a one-size-fits-all solution to congestion in multicast networks because it is static and traffic in the network can be dynamic. Limiting the number outstanding multicast operations works when used on known, static workloads, but performance suffers if the workload changes. Figure 6.2 demonstrates the loss of throughput when a static congestion window is misapplied. *Static-64* applies the window used for 64 concurrent senders to all initiator sizes. Performance is great for 64 initiators, but for fewer initiators, this window size is too small and throughput drops precipitously.

For some applications the number of senders is known beforehand [48, 101] and a tuned congestion window works well if there is no skew between each end host starting to send their data. However, skew between end hosts initiating their data transfers is inevitable, and as demonstrated above, if the congestion window is too small for the number of initiators, the network is underutilized. For other types of applications the number of concurrent senders may not be known a priori by the hardware. Therefore, *an ideal CC protocol must handle dynamic traffic conditions where the packet sources can change rapidly.* The next section describes our dynamic CC protocol for multicast networks.

Without Rate Throttling					4 GPUs with 3CPO				
Cycle	Action	Tokens	WM/ WM_Max	New Token?	Cycle	Action	Tokens	WM/ WM_Max	New Token?
0	send packet	1	10/10	Yes	0	send packet	1	10/10	Yes
1	wait	-3	10/10	Yes	1	wait	-18	10/10	Yes
2-4	wait	0	10/10	Yes	2-4	wait	-15	10/10	Yes
5	send	1	10/10	Yes	5	wait	-14	10/10	Yes
6-9	wait	0	10/10	Yes	6-9	wait	-10	10/10	Yes
10	send	1	10/10	Yes	10	wait	-9	10/10	Yes
11-14	wait	0	10/10	Yes	11-14	wait	-5	10/10	Yes
15	send	1	10/10	Yes	15	wait	-4	10/10	Yes
16-19	wait	0	10/10	Yes	16-19	wait	0	10/10	Yes
20	send	1	10/10	Yes	20	send	1	10/10	Yes

FIGURE 6.4: Example of injection rate throttling using 3CPO for 5-flit multicast packets

6.2 3CPO Design

6.2.1 Intuition of Multicast CC in 3CPO

Network multicasts and reductions exhibit unique properties that we exploit to create an effective congestion control (CC) protocol. A collective operation generates traffic at every GPU in a multicast group. Therefore, if a GPU receives a collective operation on a multicast group, and it is sending collective operations on that same multicast group, the GPU can infer that it has contention on that multicast group and should inject fewer packets, either by producing fewer IRL tokens or consuming extra tokens.

We use this observation to create a CC protocol that quickly converges to an effective injection rate for collective operations. The multicast CC in 3CPO scales the number of IRL tokens required to send a packet with the number of concurrent senders on that multicast group. Figure 6.4 demonstrates how increasing the cost

of sending a packet in proportion to the number of concurrent senders converges quickly to a rate that does not cause congestion and has high throughput. With no rate throttling, the protocol sends a packet roughly every 5 cycles. Since these are 5 flit packets and the switch is capable of sending 1 flit per cycle, the table on the left sends at line rate. When using 3CPO, the throttled GPU injects a packet every 20 cycles. Generally, 3CPO wants to wait $N \times flits$ cycles between sending each packet, where N is the number of concurrent senders. This keeps links fully utilized, but latency low.

6.2.2 Multicast CC in 3CPO

Normally, a request packet only consumes IRL tokens at the source GPU. To increase the IRL token cost of a multicast request, a multicast request consumes IRL tokens at the destination GPU when the destination GPU receives the request. This allows GPUs to consume each other’s IRL tokens. However, we must ensure that the way GPUs consume each other’s IRL tokens is fair. For example, if GPU A starts sending requests 10 microseconds before all other GPUs in the multicast group, then GPU A’s requests arrive first and consume all the IRL tokens of the GPUs in the multicast group. This prevents any other GPU in the group from sending a request. To avoid this scenario, GPUs only take each other’s IRL tokens if the destination GPU is also sending multicast requests.

3CPO should only throttle packet injection when there is active contention in the network. To achieve this, we introduce data structures to 1) match sent and received multicast traffic and 2) to avoid matching old and new traffic and therefore unnecessarily throttling injection rates. For each data structure, we maintain state for both request and response traffic, but we only detail one set for simplicity because both share the same behavior. The *unmatched* array tracks the amount of unmatched received or sent traffic in the granularity of flits. Whenever a packet is sent or

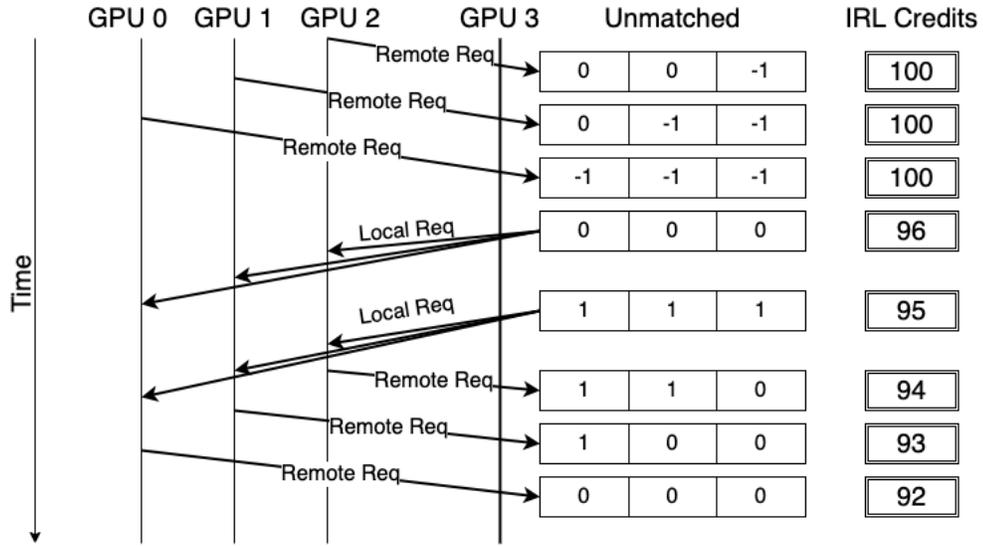


FIGURE 6.5: Process of matching local and remote operations and consuming IRL Tokens.

received, it indexes the *unmatched* array to determine whether the packet should consume extra tokens. 3CPO increments the *unmatched* array when it sends flits and decrements *unmatched* when it receives flits. The rest of this section details how sending and receiving packets changes the data structures and influences IRL token consumption.

Figure 6.5 shows the process of matching and tracking the state of the *unmatched* array. In the example there are four GPUs, GPU[0-3], and we observe the state on GPU 3 as it receives remote request from the other 3 GPUs and sends its own requests into the network. GPU 3 receives a 1 flit remote request from each of the other three GPUs, and during this time GPU 3 sends no requests. The *unmatched* array now has -1 for each index in the array, and GPU 3 consumes no IRL tokens because the requests are unmatched. GPU 3 then sends a 1 flit multicast request into the network, which matches with the 3 previously received remote requests and consumes 4 IRL tokens (1 token for the 1 flit request and 3 tokens for the 3 remote requests that matched). Immediately after, GPU 3 sends a second multicast request, and

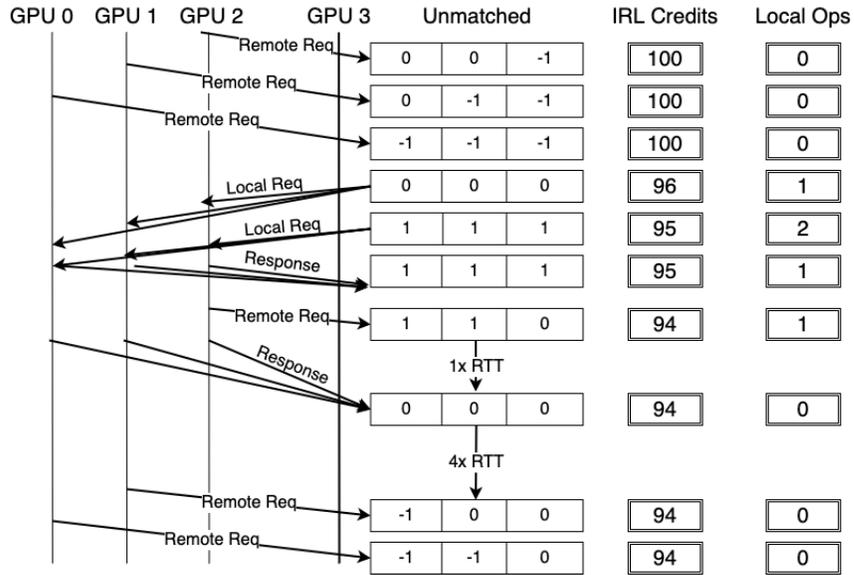


FIGURE 6.6: Tracking local requests reduces the chance of matches between requests that do not contend for bandwidth.

because there are no unmatched remote requests, sending the packet only consumes 1 IRL token, but the unmatched local request increments the *unmatched* array. Later, remote requests arrive on GPU 3, which match with the previously sent local multicast request and consumes IRL tokens. This method of matching works well if all GPUs send requests around the same time. However, after a certain amount of time, the array may contain stale information because enough time passed that the local and remote traffic no longer contend for bandwidth.

Avoiding Stale Matches When Receiving To avoid keeping stale information in the *unmatched* array, we introduce the *outstanding* array and the *Local Ops* counter, which track the maximum possible matchable flits. Flits are unmatchable when the packet the flits correspond to is no longer in the network. The *outstanding* array decrements when remote requests arrive and increments over time based on network congestion. The *Local Ops* increments when the GPU sends a request and decrements when the GPU receives a response.

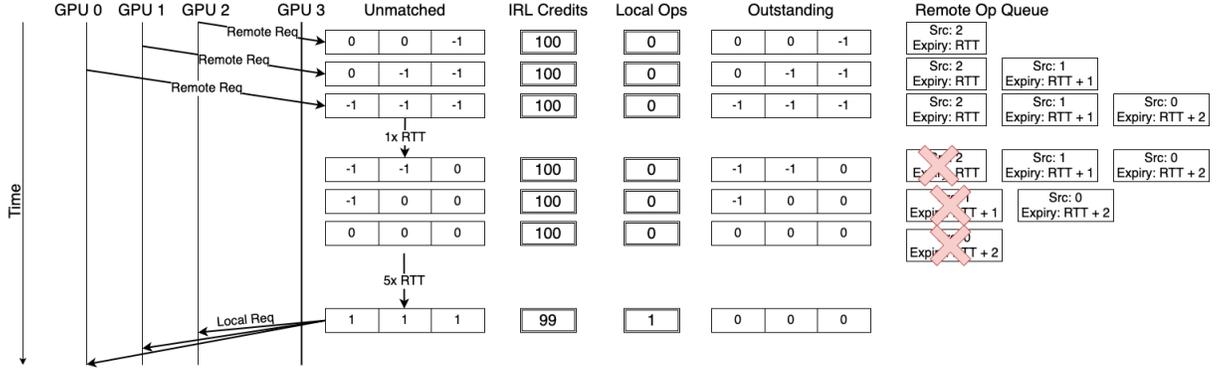


FIGURE 6.7: Incrementing *unmatched* if outstanding requests is greater. *Outstanding* operations increments as items in the remote request queue expire.

Suppose there are five RTTs between the last local request and the last set of the remote requests arriving, yet they still match even though they do not contend for bandwidth. To avoid spurious matches, we track the number of outstanding local operations (*Local Ops*) and elements in the *unmatched* array cannot exceed the number of outstanding operations. In Figure 6.6, *Local Ops* increments when a local request sends and decrements when the response arrives. When the first response arrives, the *unmatched* array remains unchanged because no index in the *unmatched* array exceeds *Local Ops*. However, once the second response arrives, *Local Ops* becomes 0. Since indices 0 and 1 in *unmatched* exceed *Local Ops*, 3CPO sets the indices equal to *Local Ops*, which is now zero. When the new remote requests arrive four RTTs later, they no longer match with the previous local requests and therefore do not consume IRL tokens. This is the desired behavior because we do not want remote requests matching with old local requests that are no longer in the network.

Avoiding Stale Matches When Sending The opposite issue can also occur; old remote requests may match with new local requests. Figure 6.7 shows the process of adding and removing items from the queue and changing the state of the *outstanding* array. We maintain the *outstanding* array to track the number of received requests from each

remote host. The value of an entry in the *unmatched* array must be greater than the corresponding entry in the *outstanding* operations array. We then store some meta data about received requests in the *Remote Op Queue*. We remove entries from the queue over time, and then increment the *outstanding* array, which reduces the chance of a spurious match. Each entry stays in the queue for about an RTT. We use local RTT measurements to determine how long a request is held in the queue. However, only incorporating RTT is insufficient, so we also include how long a packet waits at the head of the queue. Therefore, meta data remains in the queue for an RTT plus the delay between packets sent.

By tracking information about remote requests, we avoid matching two requests that do not contend for bandwidth. 3CPO removes remote requests when the current clock hits the arrival time of the packet plus the sum of the currently observed network RTT and the delay between sending multicast packets. As 3CPO removes entries from the remote request queue, we increment *outstanding* operations for the corresponding index and increment *unmatched* if *unmatched*[source] is less than *outstanding*[source].

This protocol works well at managing congestion but leaves the network slightly underutilized. To increase throughput, we added multiplicative increase to 3CPO, which reduces the number of IRL tokens taken by remote request operations. If the measured latency of a collective operation is less than 1,000 cycles, remote operations only take eight tenths the number of tokens that they normally do. In the range from 1,000 cycles to 2,000 cycles, the cost increases linearly from .8 to 1. If the RTT exceeds 2,000 cycles, the cost is 1. This improves the throughput of the protocol.

6.2.3 Unicast CC

Swift [66] inspired the unicast CC in 3CPO. Like Swift, 3CPO uses RTT as a congestion signal. If the network is congested, RTTs increase because packets spend more

time in switch queues. Like Swift, 3CPO scales the multiplicative decrease factor based on congestion; 3CPO decreases the watermark more as congestion increases.

3CPO uses a mechanism similar to Flow-Based Scaling (FBS) in Swift. Instead of using FBS to adjust *target delay* (Section 6.1.2 describes in detail) like in Swift, we use a FBS inspired function to adjust the multiplicative decrease factor. This makes the multiplicative decrease a function of injection rate and improves fairness.

3CPO measures congestion by always having an outstanding *probe packet*. If the probe packet's RTT exceeds the congestion threshold, then 3CPO lowers the *watermark* using a multiplicative decrease factor. If the RTT is below the congestion threshold, then 3CPO increases the *watermark* by a constant amount. Since the unicast portion of 3CPO is AIMD and converges to a fixed RTT latency, it is fair [19, 109].

6.2.4 Conversion from Multicast CC to Unicast CC

3CPO throttles multicast traffic by increasing the cost in terms of IRL tokens (multicast CC), and unicast CC works by lowering the *watermark*. We want to support both types of traffic but need to avoid both methods from triggering simultaneously, which can leave the network under-utilized, unless strictly necessary.

During the startup of a collective, congestion is inevitable because the multicast portion of 3CPO takes time to reduce congestion, and 3CPO should not lower the watermark. 3CPO only lowers the *watermark* if the multicast CC throttles packets less than lowering the *watermark* would. To determine which method of congestion control throttles more, we develop a method to convert between the unicast CC and multicast CC present in 3CPO, which we explain in this section.

3CPO takes a sample from the network each RTT to determine the extent of congestion. Over this time period, 3CPO records how many extra tokens it consumed when sending multicast packets, which measures multicast packet throttling.

3CPO then determines what change in the *watermark* leads to the same reduction of packets sent over the packet's RTT. We first calculate the number of tokens that the GPU likely generated over the last RTT, which we can determine using the current *watermark*.

$$L * (\textit{watermark}/\text{WM_MAX})$$

where L (for latency) is the RTT of the packet. We then create a second expression that determines how many tokens would be produced over L if we lowered the *watermark* by X .

$$L * ((\textit{watermark} - X)/\text{WM_MAX})$$

We then subtract these two expressions and set them equal to the tokens consumed in the current RTT (N).

$$N = L * \frac{\textit{watermark}}{\text{WM_MAX}} - L * \frac{(\textit{watermark} - X)}{\text{WM_MAX}}$$

which simplifies to

$$X = (\text{WM_MAX} * N)/L$$

This gives us a conversion from reducing the rate with multicast CC to reducing the rate with unicast CC. 3CPO then evaluates how much the *watermark equivalent* changed over the last RTT, which informs 3CPO's changes to the *watermark*.

Figures 6.8 and 6.9 demonstrates that 3CPO only lowers the watermark if multicast CC is insufficient to manage the network congestion. In the first RTT, there is congestion and the multicast CC starts throttling aggressively and calculates a *watermark equivalent* of 73.33. Since $73.33 > 50$ (*watermark equivalent* compared to $\text{MD} * \text{WM}$), 3CPO does not change the *watermark* because 3CPO waits to see

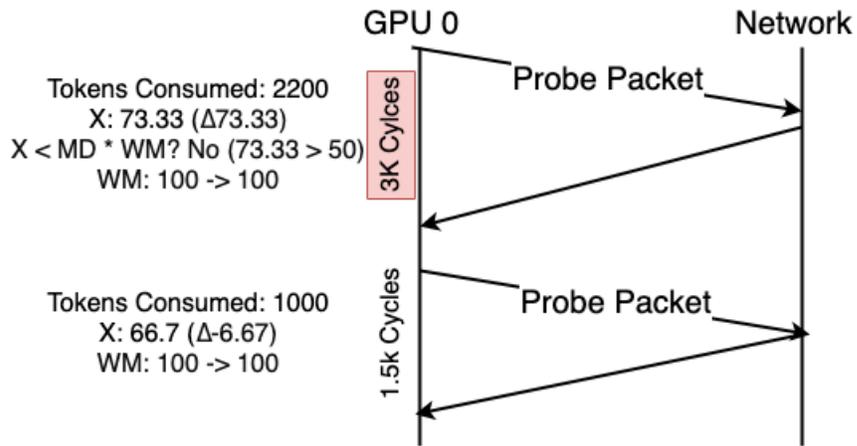


FIGURE 6.8: Congestion resolving with exclusively multicast CC. Reducing the *watermark* after the first RTT would result in a loss of throughput.

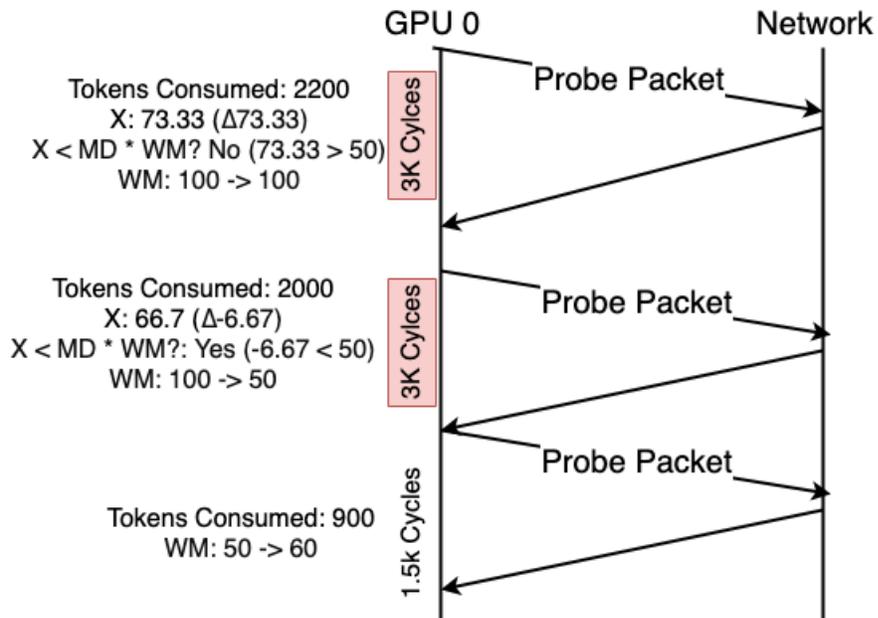


FIGURE 6.9: Demonstrates when unicast CC takes over if congestion persists and the multicast CC does not increase, the 3CPO will throttle the rate.

if the congestion alleviates with just the multicast CC. In Figure 6.8 the network congestion stops, but in Figure 6.9, the congestion persists, and the change in *watermark equivalent* (-6.67) is smaller than the watermark reduction ($100 * 0.5$), so 3CPO lowers the *watermark*, which resolves the congestion.

Algorithm 6 Protocol when Probe Packet Arrives

```

1: md_scaling =  $\frac{\text{packet\_lat} - \text{lat\_thresh}}{\text{max\_lat\_thresh} - \text{lat\_thresh}} * \text{base\_md}$ 
2: md = 1 - max(1 - md_scaling, base_md)
3: if RTT Finished then
4:   percent_mcast_writes =  $\frac{\text{sent\_mcast\_writes}}{\text{sent\_mcast\_total}}$ 
5:   if percent_writes  $\leq .5$  then
6:     size_diff = write_size - write_ack
7:     scaling_fact = (2 * (1 - percent_writes) - 1)
8:     global_dec+ = size_diff * global_dec * scaling_fact
9:   global_dec += carry_over
10:  wm_eq =  $\frac{\text{WM\_MAX} * \text{global\_dec}}{\text{packet\_lat}}$ 
11:  max_possible = wm / WM_MAX * packet_lat
12:  if wm_eq > last_wm_eq then
13:    wm_Δ = wm_eq - last_wm_eq
14:    wm = min(wm + wm_Δ, WM_MAX)
15:    if (packet_lat > lat_thresh && wm_Δ < wm * md) then
16:      fbs = max((1 - ( $\alpha / \sqrt{\text{wm} - \text{wm\_eq}} - \beta$ )), .05)
17:      wm = wm - fbs * md * (wm - wm_eq)
18:    else if packet_lat > lat_thresh then
19:      fbs = max((1 - ( $\alpha / \sqrt{\text{wm} - \text{wm\_eq}} - \beta$ )), .05)
20:      wm = wm - fbs * md * (wm - wm_eq)
21:  last_wm_eq = wm_eq
22:  carry_over = calculate_carry_over()
23:  global_dec = 0
24:  sent_mcast_writes = 0
25:  sent_total = 0
26:  if packet_lat < lat_thresh then
27:    wm = min(wm + WM_AI, WM_MAX)
28:  wm = min(max(wm, WM_MIN), WM_MAX)

```

3CPO adjusts the *watermark* based on probe packet delay and the multicast CC as shown in Algorithm 6. Line 1 determines the multiplicative decrease, which is a linear function from 1 to base_md based on the latency. The larger the latency, the

larger the decrease. Note, a lower *md* indicates a larger *watermark* decrease. Line 3 ensures that this is a probe packet.

Lines 4-9 scale the request tokens decremented (*global_dec*) based on the percentage of multicast operations that were writes. Without scaling the tokens, 3CPO may incorrectly reduce the *watermark* for the minority type of traffic. For example, suppose 3CPO handles a read dominate workload, so 3CPO mostly consumes response IRL tokens, and 3CPO consumes few request tokens. When 3CPO does the conversion from consumed request tokens to *watermark equivalent*, 3CPO calculates a low watermark, which indicates the protocol did not throttle packets, and 3CPO should lower the request watermark. However, this is not necessarily true. The request *watermark equivalent* is only small because the user sent few writes. To prevent 3CPO from unnecessarily lowering the request *watermark*, we increase the *watermark equivalent* based on how skewed the workload is.

Line 11 calculates the *watermark equivalent* and Line 12 calculates the max possible tokens produced over the last RTT. Line 13 checks to see if the *watermark equivalent* grew over the last RTT. If it did grow, Line 14 checks the change in *watermark equivalent*. If the network is congested and the change in *watermark equivalent* is less than the potential multiplicative decrease, we scale the multiplicative decrease based on the current *watermark* and *watermark equivalent*. We then decrease the *watermark*. Like in Swift, our version of FBS has two parameters, α and β , which are set with the following equations from Swift [66].

$$\alpha = \frac{1}{\frac{1}{\sqrt{\text{WM_MIN}}} - \frac{1}{\sqrt{\text{WM_MAX}}}}, \beta = \frac{\alpha}{\sqrt{\text{WM_MAX}}}$$

On lines 20-22, 3CPO decreases the watermark if the watermark equivalent shrunk over the last RTT and the network is congested. Lines 24-28 updates state for the next RTT. On lines 29-31, we perform an additive increase if there is no

congestion. Line 32 ensures the watermark is in its bounds.

6.2.5 *Virtual Channels and Arbitration*

To make 3CPO effective, we modify how the network interface assigns packets to virtual channels (VCs) and how the network arbitrates between those VCs. Originally, the network used 2 VCs: one for request traffic and one for response traffic. The network uses additional VCs when a topology introduced potential circular dependencies. The switch used a round robin arbiter between VCs.

Sending reads and writes on the same VC and using round robin arbitration between the VCs led to issues with 3CPO during all-reduce traffic patterns. To execute the all-reduce, each GPU reduces a subset of the data, and then the GPU broadcasts the reduction result. During a large all-reduce the network interface throttles the injection of packets to a small fraction of its maximum injection rate, so there is significant queueing between the GPU and the network. Then once each read completes, the application issues the corresponding write to broadcast the partial result, but the reads block the writes from entering the network. Since the network is most effective when reads and writes are interleaved, reads blocking the writes leads to poor performance.

To fix this issue, we use separate VCs for reads and writes on the GPU injection channel. The network moves the reads and writes to the same VC after the first hop. We also change from round robin arbitration to least recently used arbitration. Round robin arbitration worked poorly with 3CPO because the arbiter skips a VC if there is insufficient IRL tokens. However, when enough IRL tokens existed for the skipped VC, the round robin arbiter often did not point to the starved VC. Using least-recently used arbitration fixed this issue, and we use this new arbiter and VC allocation policy in all experiments.

6.3 Evaluation

We evaluate 3CPO on several synthetic congestion benchmarks to show its effectiveness. We demonstrate that 3CPO manages congestion in common communication patterns: all-reduce, write multicast, and read reductions. We evaluate using an all-reduce because its performance is critical to deep-learning workloads [73]. We then run workloads to demonstrate that 3CPO works in dynamic traffic situations. In one workload, we run unicast traffic intermixed with multicast traffic, which shows 3CPO manages congestion with both types of traffic. We also run a random multicast workload, where the number of GPUs initiating multicasts changes rapidly, which shows how 3CPO adjusts when network conditions change. Finally, to simulate a shared network, we run two disjoint workloads in the network simultaneously, which demonstrates 3CPO provides reasonable performance isolation in shared environments.

6.3.1 Methodology

We test 3CPO in a heavily modified version of booksim [50]. For all simulations, we use a two-level fat-tree topology with 64 or 128 GPUs. Each ToR switch connects to 8 GPUs, and the network is not oversubscribed. In the 128 GPU network, there are 16 ToR and 16 Core switches, and there are 8 ToR and 4 core switches for the 64 GPU network. The switches are the same as the ones we used in Section 6.1.4. Scaling 3CPO beyond 128 GPUs is future work. We compare 3CPO to a baseline network with no CC, and a hand-tuned congestion window, which we size to maximize bandwidth and manage congestion. For all experiments, we do not change any 3CPO parameters, which demonstrates how well 3CPO adjusts to traffic without network administrator intervention.

3CPO has several parameters that we set empirically for our evaluation. Multicast

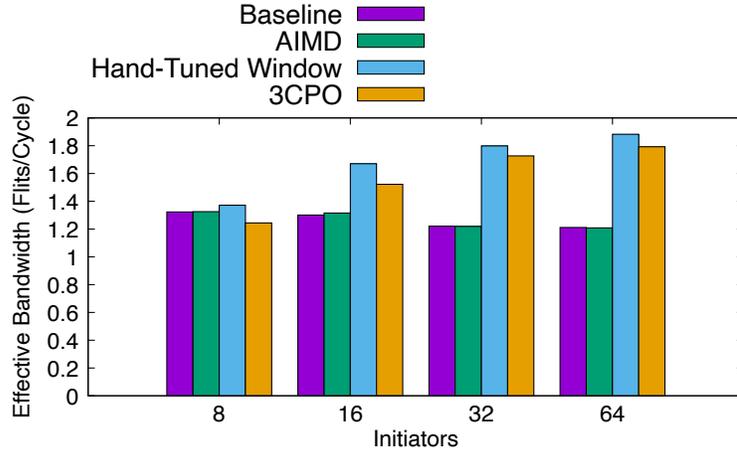


FIGURE 6.10: Effective Bandwidth of in All-Reduce Workload 64 GPUs

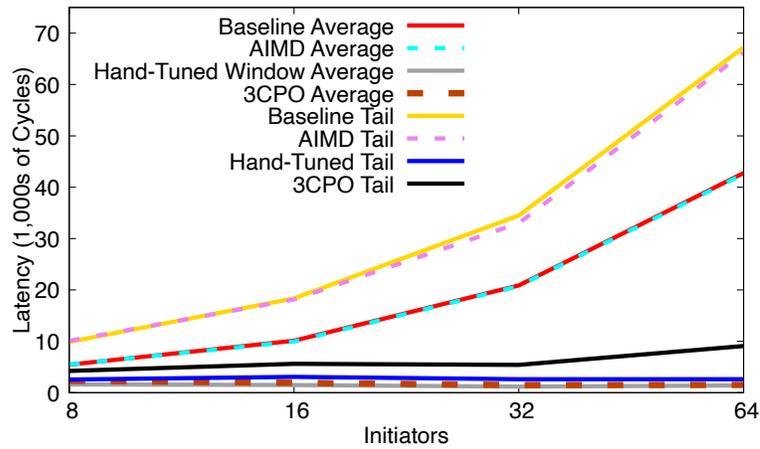


FIGURE 6.11: Average and Tail Packet Latency in All-Reduce Workload 64 GPUs

CC has three parameters that determine the multiplicative increase and are described in Section 6.2.2. 3CPOs unicast CC uses the same parameters as the AIMD protocol in Section 6.1.4.

6.3.2 Varied Initiators

We perform the same experiments from Section 2 using 3CPO. In these experiments, we perform an all-reduce where each *initiator* performs 128 read reductions, each of which is followed by a corresponding write multicast. We call a read-write operation pair a *transaction*. There are 64 GPUs in the network. We then vary the number of

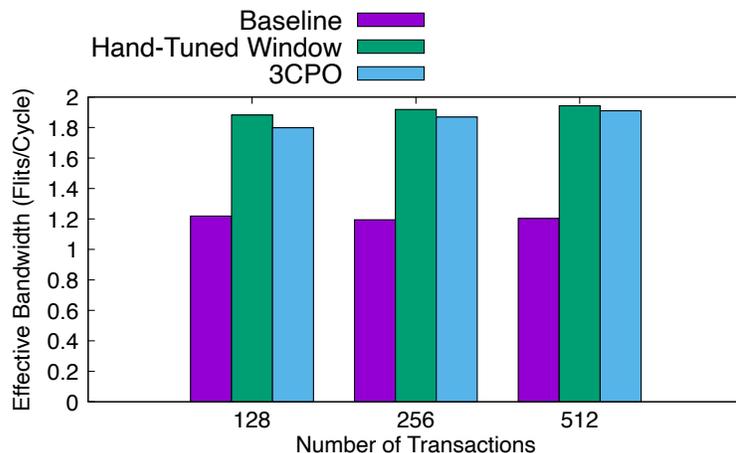


FIGURE 6.12: Effective Bandwidth of in All-Reduce Workload 64 GPUs

initiators across experiments. Figures 6.10 and 6.11 shows the effective bandwidth and packet latencies. 3CPO and the hand-tuned window have similar average packet latencies, but 3CPO has a slightly higher 100% tail latency because 3CPO takes time to adjust the injection rate, so there is transient start-up congestion. However, recall that the hand-tuned window size changes for each unique number of initiators. In contrast, 3CPO adjusts dynamically to the observed traffic conditions, making 3CPO a practical congestion management method.

3CPO dynamically achieves performance comparable to a congestion window that is tuned specifically for each experiment. Like the hand-tuned window, 3CPO maintains high throughput for all experiments. 3CPO performs slightly worse than the hand-tuned window because 3CPO slightly over-throttles during startup. Figure 6.12 shows effective bandwidth as we increase the number of transactions (perform a larger all-reduce). As we issue more transactions during the experiment the difference between 3CPO and the hand-tuned window decreases. For the rest of the experiments, we omit AIMD since it is clearly an ineffective method of CC in our network.

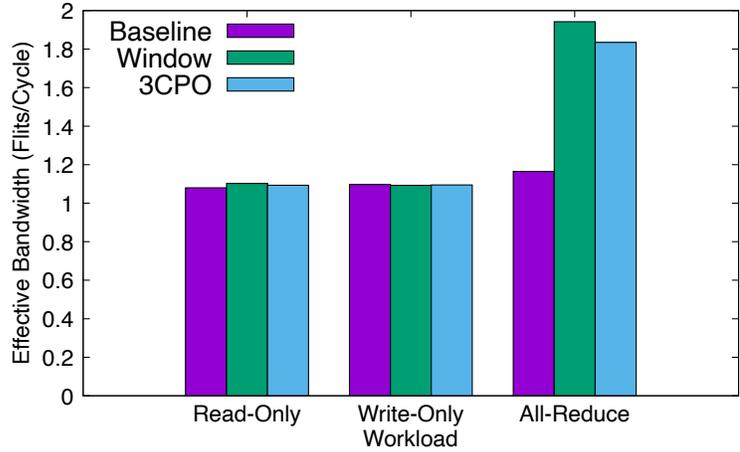


FIGURE 6.13: Effective Bandwidth of Symmetric Workloads 128 GPUs

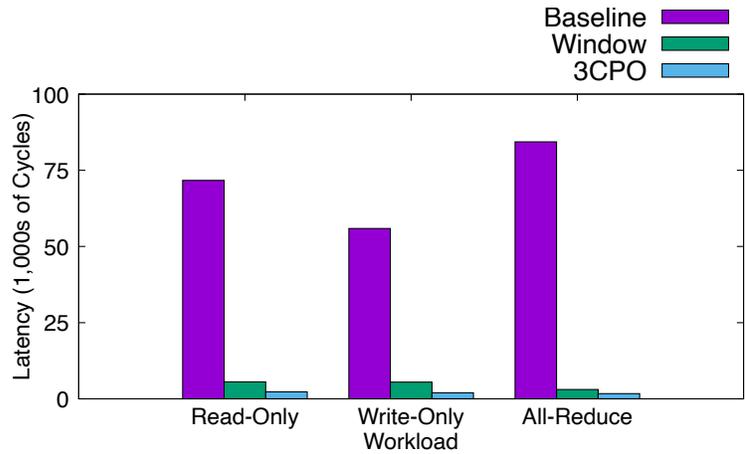


FIGURE 6.14: Average Packet latency in Symmetric Workloads with 128 GPUs

6.3.3 Symmetric Multicast Workloads

We run three workloads that each involve all 128 GPUs performing the same operations at the same time and show that 3CPO handles these cases well. We run one workload that is 128 read reductions, one workload issues 128 write multicasts, and the third is an all-reduce that issues 128 read reductions and 128 write multicasts. Figures 6.13 and 6.14 show the effective bandwidth and average packet latencies of the three workloads. We compare 3CPO with a hand-tuned window based approach and a baseline network with no CC. In the read-only workload all three variants

achieve nearly the same effective bandwidth. However, the packet latencies differ greatly. With no CC, the average packet latency is nearly 140,000. Meanwhile, the average packet latency with the window is 5,600 cycles and 1,800 cycles with 3CPO.

3CPO maintains high performance during the write and all-reduce workloads. In the write-only workload, all three variants have the same effective bandwidth, but again the packet latencies in 3CPO and static window are small compared to baseline. In the all-reduce workload, effective bandwidth in baseline suffers due to reads blocking writes in the network. Window and 3CPO have high effective bandwidths and maintain low packet latencies. 3CPO has slightly lower effective bandwidth due to over throttling the injection of packets during application start-up. We already showed this discrepancy reduces as we perform longer experiments.

6.3.4 Dynamic Traffic Patterns

To demonstrate that 3CPO works in dynamic traffic patterns, we create two new benchmarks that change the traffic characteristics during the simulation. The first workload shows that 3CPO adjusts quickly when GPUs randomly start and stop sending packets. The second workload shows 3CPO is effective when both unicast and multicast packets exist in the network. First, we create a random multicast benchmark (Random-Multicast) where each GPU injects 10 sets of multicast operations. Each operation has an equal chance to be a read or write multicast. Each set is a random number of multicast operations on the range 64 to 256. Between each set of multicast operation, the GPU sits idle for a random number of cycles on the range 5,000 to 10,000 cycles. This randomizes the number of GPUs injecting traffic at any given time and shows how 3CPO adjusts to dynamic traffic situations.

The second workload mixes unicast traffic with multicast traffic (Multicast-Uni). Like Random-Multicast, Multicast-Uni hosts generate a random number of operations to perform. However, in Multicast-Uni the sets of operations alternate between

being unicast packets and multicast packets. All operations in the same set are the same type, either unicast or multicast; no set of operations has both unicast and multicast packets. If a set is unicast packets, then the random number of operations in the set is multiplied by 5. All unicast packets in a set share a destination. Because each GPU sends a random number of operations in the workload, instead of reporting the average packet latency across all GPUs, we take the average latency for each GPU and then find the mean of those averages.

Figures 6.15 and 6.16 show the effective bandwidth and average packet latency during these experiments. 3CPO maintains high throughput and low latencies during both workloads. Baseline suffers from high packet latencies and the window based CC cannot adjust its static window in Random-Multicast, so it has lower throughput. We could make the difference in performance larger with sparser communication, but Section 2 already shows how sparse communication reduces throughput with a static window. 3CPO has the highest effective bandwidth because it correctly adjusts the injection rate of packets to meet the current demand. 3CPO also keeps packet latencies low. 3CPO has higher packet latencies in the Multicast-Uni benchmark because the unicast packet throttling reacts slower than the multicast packet throttling.

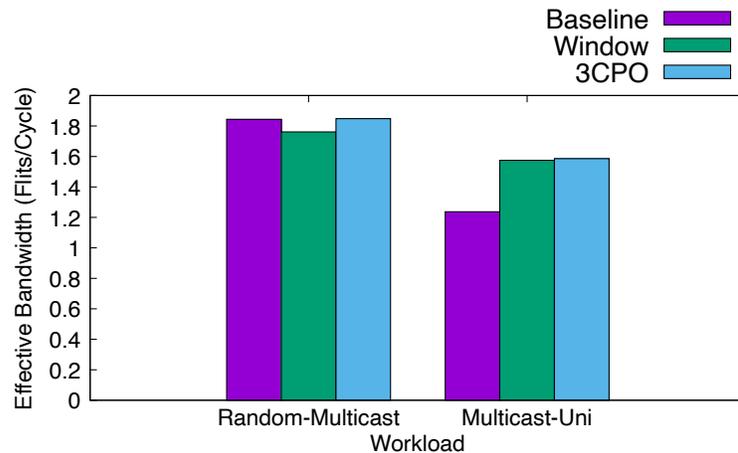


FIGURE 6.15: Effective Bandwidth in Dynamic Workloads 128 GPUs

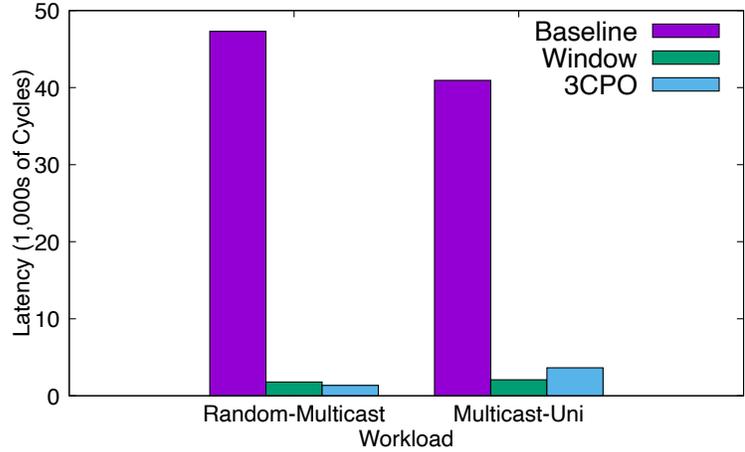


FIGURE 6.16: Average Packet Latency of Dynamic Workloads 128 GPUs

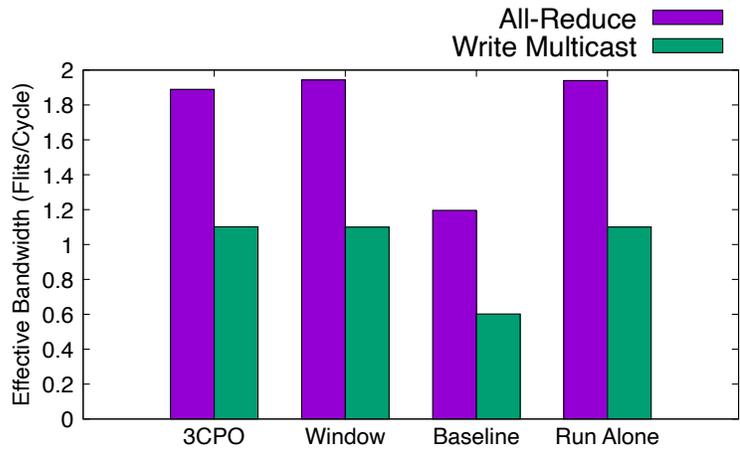


FIGURE 6.17: Effective Bandwidth of Shared Network Workloads 128 GPUs

6.3.5 Shared Network Workloads

Another motivation for managing congestion in our network is performance isolation. For example, two users may be sharing the NVLink network and congestion can cause the applications to interfere with one another. To demonstrate application interference and how congestion management can improve performance, we run several simulations where two applications run in the network simultaneously. Each workload runs on a randomly selected group of 64 GPUs in the 128 GPU network. Figure 6.17 shows the effective bandwidth of an all-reduce workload, which performs 512 trans-

actions, run alongside a write multicast workload, which performs 512 writes. The bars labeled “Run Alone” demonstrate the workload’s throughput when each workload is run in the network alone on 64 GPUs with a perfectly configured window, so there is no interference. Baseline shows that without congestion management, both workloads performance suffers. 3CPO performs only slightly worse than the hand-tuned congestion window. All the workloads have average packet latencies in the range 1,300 to 1,600 cycles, except in the experiment without congestion management, which has packet latencies around 75,000 cycles. Expanding beyond two applications is left for future work.

6.4 Conclusion

Many data-intensive applications now rely on distributed accelerator systems to meet computational and energy-efficiency demands. While distributed systems meet these demands, the network interconnecting the devices can bottleneck application performance. To improve application throughput, networks integrated in-network collectives, which increase the effective bandwidth of the network for certain traffic patterns common to HPC applications and Machine Learning. These networks increase performance but perform poorly when congested. We greatly improve existing congestion control for multicast networks by exploiting the symmetry in multicast operations to converge quickly to effective injection rates. We develop 3CPO, a congestion control protocol for multicast networks that integrates with existing unicast congestion control. We show that 3CPO adjusts quickly to dynamic traffic situations, maintains high throughput, and keeps packet latencies low. The performance of 3CPO is comparable to application specific hand-tuned CC and maintains flexibility to adjust to dynamic workloads.

Conclusion

Distributed systems underpin many critical workloads like Artificial Intelligence, Scientific Computing, and web-services. Although distributed applications can scale to massive size, large and complex distributed systems create the potential for performance bottlenecks. One possible bottleneck present in distributed systems is communication delay. Different processing elements within the system often must communicate with one another to exchange data or synchronize. If communication costs are high, overall application performance suffers because the application spends excessive time performing network operations instead of computation.

RDMA networks provide extremely low network latency and high bandwidth, which increase network performance. To exploit this performance, many applications tightly integrate RDMA into the application. While RDMA network hardware enables low communications costs, it is not sufficient to guarantee good network performance. To ensure RDMA networks perform up to their potential, networks often use congestion control protocols, which limit the packet injection if too many packets occupy network queues.

A congestion control protocol aims to fully utilize network bandwidth without

causing switch queues to fill with packets while also ensuring the network allocates bandwidth fairly to competing entities. If a congestion control protocol does not utilize network bandwidth, it wastes valuable resources and harms application performance. If there is excessive queueing delay in the network, small, latency bound flows complete slowly and harm application performance. Further, since most RDMA networks are lossless, they suffer from tree saturation when packet queues become full. Further, if the network does not allocate bandwidth fairly, flows with too little bandwidth perform poorly.

These three criteria: 1) low latency, 2) high bandwidth utilization, 3) and fairness are difficult to achieve on a short time scale. Because network latencies are so low and network bandwidth is so high, flows complete quickly and the congestion control needs to make fast decisions to ensure good performance. Often times bandwidth allocations to end hosts are not optimal, so they do not meet one of the three criteria. Over time, the protocol ideally finds the optimal injection rates, but while the protocol searches for the correct injection rates, application performance suffers. This dissertation introduces mechanisms that improve convergence to optimal injection rates that meet the three criteria.

7.1 Key Contributions

This thesis makes contributions in four key areas of congestion control convergence: 1) identifying performance and security issues related to slow convergence, 2) improving convergence in sender-side congestion control protocols, 3) converging to fair rates in the first RTT in switch-based protocols, and 4) improving convergence in multicast networks.

We found that when protocols converge slowly to fair rate allocations long flow tail latency becomes extremely high and the network becomes susceptible to performance isolation attacks. Anytime the network's bandwidth allocation is unfair, a

flow’s performance suffers because the network allocates at least one flow too little bandwidth. This flow then takes longer to send its data, so it takes longer to complete. Our experiments show that improving convergence to fair rates can reduce long flow tail FCT by $2\times$. This poor performance is then passed along to the application, which requires the flow to complete. Further, we found that bandwidth unfairness in common RDMA networks is predictable, which allows a misbehaving user to gain an unfair bandwidth allocation. We demonstrate that a user can completely ignore RDMA congestion control and in a small testbed experiment, a misbehaving was able to get 75% of the available network bandwidth. We also find a fundamental tradeoff between performance isolation and starting flows at line rate.

Most RDMA networks rely on sender-side protocols, where an end host receives congestion feedback from the network and then adjusts its injection rate based on that feedback. We find that existing congestion control protocols converge to fair rates slowly, extending long flow tail FCT. To improve convergence to fair rates, we design two mechanisms, Variable Additive Increase and Sampling Frequency, that augment existing sender-side protocols to improve convergence to fair rate allocations. These new mechanisms require no changes to switches or network telemetry. Variable Additive Increase infers unfairness on the end host by observing network state and adjusting the additive increase temporarily to improve fairness without increasing latency in the long term. Sampling frequency adjusts the multiplicative decrease schedule to be per-ACK instead of per-RTT. This exploits a natural fairness affect where a flow with more bandwidth receives more ACKs. Using sampling frequency, a flow with more bandwidth decreases its rate more often. When we augment Swift and HPCC with Variable Additive Increase and Sampling Frequency, the 99.9% tail long flow completion time is $2x$ lower than without our mechanisms. Variable Additive Increase and Sampling Frequency also maintain high throughput and low latency, and do not harm performance during incast traffic.

The next contribution moves into switch-based congestion control protocols. To mitigate performance isolation attacks and ensure congestion control is effective in high BDP networks, we introduce 1RC and a new weighting scheme. 1RC enables flows to converge to their near fair rate during the flow’s first RTT, does not require storing per-flow state on switches, and maintains packet ordering, which similar mechanisms do not. 1RC allows flows to start sending packets at line rate, but the switch drops packets from new flows if letting the packet through the switch allocates the new flow too much bandwidth. This prevents performance isolation attacks because a new flow is no longer allocated more bandwidth than an existing flow. Further, it enables congestion control protocols to manage congestion well even when BDP product is larger the number of bytes most flows send. Our new weighting scheme decrease a flows bandwidth allocation as the flows weight increases. We then set a flows weight to the number of concurrent flows the end host has, which discourages a user from opening more flows to gain more bandwidth.

The final contribution is 3CPO, a congestion control protocol specifically for multicast networks that exploits the symmetry of multicast operations and converges to near optimal rates almost instantly. Our protocol is particularly effective during collective operations, which are important for HPC and AI workloads. 3CPO also integrates seamlessly with unicast congestion control, so it can manage both traffic types simultaneously. Our new congestion control protocol achieves throughput within 2% of a hand-tuned congestion control window and has similar packet latencies.

7.2 Future Work

7.2.1 *Hardware Implementations*

This thesis exclusively evaluated new congestion control designs with simulators. While simulators effectively model algorithm behavior and enable evaluating new

features quickly, they do not model all of a network’s complexities. Implementing our congestion control methods in switches and NICs would provide a more robust evaluation of our features.

7.2.2 Sharing In-Network Collective Resources

Congestion control is a distributed scheduling problem, where we want to allocate bandwidth between several competing entities (flows, users, applications, etc). However, Chapter 6 introduces in-network collective functionality, which is tied closely to bandwidth, but has separate resources. Now the network must share in-network collective resources between users in addition to network bandwidth. Further complicating in-network collective resource sharing is how bandwidth allocations and in-network collective allocations can affect each other. In Klenk et al. [64], if a multicast packet cannot reserve in-network collective resources, packets wait for the switch to free resources as other multicast operations complete. Because the multicast packets wait in network queues, this can cause head-of-line blocking for unicast packets stuck behind the waiting multicast packet and reduce overall network throughput. Therefore, a poor in-network collective resource allocation can reduce an applications performance even if the packets do not rely on multicast resources. Congestion control protocols for networks with in-network collectives must consider collective resources in addition to bandwidth when making scheduling decisions.

7.2.3 Adaptive Routing and Congestion Control

Adaptive routing complicates congestion control because packets from the same flow may take different paths, making accurate congestion detection more difficult and introducing a new potential avenue for unfairness between competing flows. Network congestion decisions depend on a flows path. In HPCC [75], end hosts adjust injection rates based on fine-grained feedback from network switches. However, if the network

adaptively routes the packets, the end host does not know which path the packets take and therefore does not know exactly how much to reduce or increase its injection rate. Adaptive routing also complicates Swift [66] because Swift relies on RTT measurements and adaptive routing may use non-minimal adaptive routing, which changes the packet’s path’s length. Non-minimal adaptive routing is common in Dragonfly networks [63, 52], which makes packets from the same flow have different base RTTs.

Adaptive routing also has new fairness implications. Since an end host does not know the path its packets take, it cannot converge to fair rates because the fair rate may be different along each path. Investigating how adaptive routing affects fairness remains an interesting, open problem.

7.2.4 Improving and Implementing 3CPO

Currently, 3CPO effectively manages congestion in Klenk et al.’s multicast architecture [64]. However, 3CPO may have some features that are difficult to implement in a network interface. For example, the remote op queue described in Section 6.2.2 can become large and potentially exhaust network interface resources. We want to investigate simpler designs for 3CPO that may be more amenable to simple network interfaces.

We also stopped scaling 3CPO at 128 GPUs because this more than covers the largest existing NVLink systems. However, future NVLink systems may be larger, and we must study how 3CPO scales. Further, we only investigated how 3CPO handles 2 applications running simultaneously. We plan to see how 3CPO performs when more applications share the network. More applications will cause more network congestion in the network core due to how packets are replicated and reduced. 3CPO therefore needs to quickly handle congestion deep within the network topology.

7.3 Conclusion

Distributed systems rely on congestion control to ensure network performance meets application demands. We identify several ways in which modern RDMA networks perform poorly, including susceptibility to performance isolation attacks and poor convergence to fair rates. We measure the effectiveness of performance isolation attacks and showed they can devastate network performance. We then show that slow converge to fair rates significantly increases long flow tail FCT.

To improve network performance, we introduce several novel congestion control methods 1) Variable Additive Increase and Sampling Frequency, which improve convergence to fair rates in sender-side protocols, 2) 1RC and a new weighting scheme to prevent or discourage users from trying to unfairly gain bandwidth, and 3) 3CPO, a congestion control protocol specifically designed for multicast networks. All these systems improve convergence to effective congestion rates in different contexts. These mechanisms improve congestion control performance and security, so distributed applications perform well.

Bibliography

- [1] “<https://github.com/alibaba-edu/High-Precision-Congestion-Control>,” Jul. 2020.
- [2] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, p. 63–74, Aug. 2008. [Online]. Available: <https://doi.org/10.1145/1402946.1402967>
- [3] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshminantha, R. Pan, B. Prabhakar, and M. Seaman, “Data center transport mechanisms: Congestion control theory and iee standardization,” in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, 2008, pp. 1270–1277.
- [4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center TCP (DCTCP),” in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM ’10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: <http://doi.acm.org/10.1145/1851182.1851192>
- [5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, “Less is more: Trading a little bandwidth for ultra-low latency in the data center,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’12. USA: USENIX Association, 2012, p. 19.
- [6] J. Ames, D. F. Puleri, P. Balogh, J. Gounley, E. W. Draeger, and A. Randles, “Multi-gpu immersed boundary method hemodynamics simulations,” *Journal of Computational Science*, vol. 44, p. 101153, 2020.
- [7] Y. Bai, D. Dong, S. Huang, Z. Zhou, and X. Liao, “Ssp: Speeding up small flows for proactive transport in datacenters,” in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 153–161.
- [8] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019. [Online]. Available: <https://doi.org/10.1145/3320060>

- [9] M. S. Birrittella *et al.*, “Intel® omni-path architecture: Enabling scalable, high performance fabrics,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 1–9.
- [10] N. Bloch, B. Shlomo, E. Zahavi, and Z. Yaakov, “Destination-based congestion control,” Apr 2014.
- [11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: <https://doi.org/10.1145/2656877.2656890>
- [12] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, “Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation,” *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008.
- [13] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, “0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner,” in *SC’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–11.
- [14] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, “Advances in petascale kinetic plasma simulation with vplic and roadrunner,” in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012055.
- [15] L. S. Brakmo, S. W. O’Malley, and L. L. Peterson, “Tcp vegas: New techniques for congestion detection and avoidance,” in *Proceedings of the conference on Communications architectures, protocols and applications*, 1994, pp. 24–35.
- [16] e. a. Bylaska, EJ, W. De Jong, N. Govind, K. Kowalski, T. Straatsma, M. Valiev, D. Wang, E. Apra, T. Windus, J. Hammond *et al.*, “Nwchem, a computational chemistry package for parallel computers, version 5.1,” *Pacific Northwest National Laboratory, Richland, Washington*, vol. 99352, p. 0999, 2007.
- [17] V. Cerf and R. Kahn, “A protocol for packet network intercommunication,” *IEEE Transactions on communications*, vol. 22, no. 5, pp. 637–648, 1974.
- [18] H. J. Chao, “Design of leaky bucket access control schemes in atm networks,” in *ICC 91 International Conference on Communications Conference Record*. IEEE, 1991, pp. 180–187.
- [19] D.-M. Chiu and R. Jain, “Analysis of the increase and decrease algorithms for congestion avoidance in computer networks,” *Comput. Netw. ISDN Syst.*, vol. 17, no. 1, p. 1–14, Jun. 1989. [Online]. Available: [https://doi.org/10.1016/0169-7552\(89\)90019-6](https://doi.org/10.1016/0169-7552(89)90019-6)

- [20] I. Cho, K. Jang, and D. Han, “Credit-scheduled delay-bounded congestion control for datacenters,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 239–252. [Online]. Available: <https://doi.org/10.1145/3098822.3098840>
- [21] C. Clos, “A study of non-blocking switching networks,” *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.
- [22] J. Crowcroft and P. Oechslin, “Differentiated end-to-end internet services using a weighted proportional fair sharing tcp,” *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 3, pp. 53–69, 1998.
- [23] D. Crupnicoff, M. Kagan, A. Shahar, N. Bloch, and H. Chapman, “Dynamically-connected transport service,” Apr 2014.
- [24] D. De Sensi, S. Di Girolamo, K. H. McMahan, D. Roweth, and T. Hoefer, “An in-depth analysis of the slingshot interconnect,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.
- [25] J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: <http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext>
- [26] G. Dearth, N. Jiang, J. Wortman, A. Ishii, M. Hummel, and R. Reeves, “Techniques for reducing congestion in a computer network,” Aug. 3 2021, uS Patent 11,082,347.
- [27] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, “Farm: Fast remote memory,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414. [Online]. Available: <https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%7B%7D%40tempbox%7B%7D%40global%7B%7D%40mathchardef%7B%7D%40spacefactor%7B%7D%40let%7B%7D%40begingroup%7B%7D%40relax%7B%7D%40ignorespaces%7B%7D%40relax%7B%7D%40accent19c%7B%7D%40egroup%7B%7D%40spacefactor%7B%7D%40accent%7B%7D%40spacefactor%7B%7D%7D>
- [28] N. Dukkupati and N. McKeown, “Why flow-completion time is the right metric for congestion control,” *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, p. 59–62, Jan. 2006. [Online]. Available: <https://doi.org/10.1145/1111322.1111336>
- [29] L. Eggert, J. Heidemann, and J. Touch, “Effects of ensemble-tcp,” *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 1, pp. 15–29, 2000.

- [30] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.
- [31] S. B. Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts, “Statistical bandwidth sharing: A study of congestion at flow level,” in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM ’01. New York, NY, USA: Association for Computing Machinery, 2001, p. 111–122. [Online]. Available: <https://doi.org/10.1145/383059.383068>
- [32] E. Gafni and D. Bertsekas, “Dynamic control of session input rates in communication networks,” *IEEE Transactions on Automatic Control*, vol. 29, no. 11, pp. 1009–1016, 1984.
- [33] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, “phost: Distributed near-optimal datacenter transport over commodity network fabric,” in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015, pp. 1–12.
- [34] Y. Gao, Y. Yang, T. Chen, J. Zheng, B. Mao, and G. Chen, “Dcqn+: Taming large-scale incast congestion in rdma over ethernet networks,” in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, pp. 110–120.
- [35] R. Garg, A. Kamra, and V. Khurana, “A game-theoretic approach towards congestion control in communication networks,” *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 3, pp. 47–61, 2002.
- [36] N. Gebara, M. Ghobadi, and C. Paolo, “In-network aggregation for shared machine learning clusters,” *Proceedings of Machine Learning and Systems*, vol. 3, 2021.
- [37] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, “Exploiting a natural network effect for scalable, fine-grained clock synchronization,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 81–94.
- [38] E. Georganas, R. Egan, S. Hofmeyr, E. Goltsman, B. Arndt, A. Tritt, A. Buluç, L. Olikar, and K. Yelick, “Extreme scale de novo metagenome assembly,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 122–134.
- [39] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir *et al.*, “Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient

- data reduction,” in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 2016, pp. 1–10.
- [40] J. Gu *et al.*, “Tiresias: A {GPU} cluster manager for distributed deep learning,” in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 485–500.
- [41] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, “Rdma over commodity ethernet at scale,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 202–215. [Online]. Available: <https://doi.org/10.1145/2934872.2934908>
- [42] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. 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*, ser. SIGCOMM ’17. New York, NY, USA: ACM, 2017, pp. 29–42. [Online]. Available: <http://doi.acm.org/10.1145/3098822.3098825>
- [43] S. Hu, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang, “Aeolus: A building block for proactive transport in datacenters,” in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 422–434.
- [44] *Supplement to InfiniBandTM Architecture Specification*, InfinibandTM Trade Association, 9 2014, volume 1 Release 1.2.1.
- [45] *InfiniBandTM Architecture Specification*, InfinibandTM Trade Association, 4 2020, volume 1 Release 1.4.
- [46] V. Jacobson, “Congestion avoidance and control,” *ACM SIGCOMM computer communication review*, vol. 18, no. 4, pp. 314–329, 1988.
- [47] R. Jain, D.-M. Chiu, and W. Hawe, “A quantitative measure of fairness and discrimination for resource allocation in shared computer systems,” *ArXiv*, vol. cs.NI/9809099, 1998.
- [48] S. Jeagey, “Nccl 2.0,” in *GPU Technology Conference (GTC)*, 2017.
- [49] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally, “Network congestion avoidance through speculative reservation,” in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

- [50] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, “A detailed and flexible cycle-accurate network-on-chip simulator,” in *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2013, pp. 86–96.
- [51] N. Jiang, L. Dennison, and W. J. Dally, “Network endpoint congestion control for fine-grained communication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2807591.2807600>
- [52] N. Jiang, J. Kim, and W. J. Dally, “Indirect adaptive routing on large scale interconnection networks,” in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 220–231.
- [53] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, “A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters,” in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 463–479.
- [54] C. Jin, D. X. Wei, and S. H. Low, “Fast tcp: motivation, architecture, algorithms, performance,” in *IEEE INFOCOM 2004*, vol. 4. IEEE, 2004, pp. 2490–2501.
- [55] L. Jose, S. Ibanez, M. Alizadeh, and N. McKeown, “A distributed algorithm to calculate max-min fair rates without per-flow state,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 2, pp. 1–42, 2019.
- [56] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, “A domain-specific supercomputer for training deep neural networks,” *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.
- [57] A. Kalia, M. Kaminsky, and D. Andersen, “Datacenter rpcs can be general and fast,” in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 1–16.
- [58] A. Kalia, M. Kaminsky, and D. G. Andersen, “Design guidelines for high performance {RDMA} systems,” in *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, 2016, pp. 437–450.
- [59] —, “Fasst: Fast, scalable and simple distributed transactions with two-sided ({RDMA}) datagram rpcs,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 185–201.
- [60] M. J. Karol, “Input vs. output queuing on a space-division packet switch,” *IEEE Trans. Commun.*, vol. 35, no. 12, pp. 110–115, 1987.

- [61] D. Katabi, M. Handley, and C. Rohrs, “Congestion control for high bandwidth-delay product networks,” in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, 2002, pp. 89–102.
- [62] F. Kelly, “Charging and rate control for elastic traffic,” *European transactions on Telecommunications*, vol. 8, no. 1, pp. 33–37, 1997.
- [63] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-driven, highly-scalable dragonfly topology,” in *2008 International Symposium on Computer Architecture*. IEEE, 2008, pp. 77–88.
- [64] B. Klenk and H. Fröning, “An overview of mpi characteristics of exascale proxy applications,” in *International Supercomputing Conference*. Springer, 2017, pp. 217–236.
- [65] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, “An in-network architecture for accelerating shared-memory multiprocessor collectives,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 996–1009.
- [66] G. Kumar, N. Dukkupati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat, “Swift: Delay is simple and effective for congestion control in the datacenter,” in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 514–528. [Online]. Available: <https://doi.org/10.1145/3387514.3406591>
- [67] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, “Atp: In-network aggregation for multi-tenant learning,” in *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, 2021, pp. 741–761.
- [68] K. Lee and J. Eidson, “IEEE-1588 standard for a precision clock synchronization protocol for networked measurement and control systems,” in *In 34 th Annual Precise Time and Time Interval (PTTI) Meeting*, 2002, pp. 98–105.
- [69] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon, “Globally synchronized time via datacenter networks,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 454–467.
- [70] C. E. Leiserson, “Fat-trees: Universal networks for hardware-efficient supercomputing,” *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, 1985.

- [71] K. Lepak, G. Talbot, S. White, N. Beck, S. Naffziger, S. FELLOW *et al.*, “The next generation amd enterprise server product architecture,” *IEEE hot chips*, vol. 29, 2017.
- [72] B. Li, G. Zuo, W. Bai, and L. Zhang, “1pipe: scalable total order communication in data center networks,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 78–92.
- [73] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, “Accelerating distributed reinforcement learning with in-switch computing,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 279–291.
- [74] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkupati, P. Chandra *et al.*, “Sundial: Fault-tolerant clock synchronization for datacenters,” in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 1171–1186.
- [75] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, “Hpcc: High precision congestion control,” in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 44–58. [Online]. Available: <https://doi.org/10.1145/3341302.3342085>
- [76] Lisong Xu, K. Harfoush, and Injong Rhee, “Binary increase congestion control (bic) for fast long-distance networks,” in *IEEE INFOCOM 2004*, vol. 4, 2004, pp. 2514–2524 vol.4.
- [77] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkupati, W. C. Evans, S. Gribble *et al.*, “Snap: A microkernel approach to host networking,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 399–413.
- [78] P. E. McKenney, “Stochastic fairness queueing,” in *Proceedings. IEEE INFOCOM ’90: Ninth Annual Joint Conference of the IEEE Computer and Communications Societies The Multiple Facets of Integration*, 1990, pp. 733–740 vol.2.
- [79] R. M. Metcalfe and D. R. Boggs, “Ethernet: Distributed packet switching for local computer networks,” *Communications of the ACM*, vol. 19, no. 7, pp. 395–404, 1976.
- [80] P. A. Misra, J. S. Chase, J. Gehrke, and A. R. Lebeck, “Enabling lightweight transactions with precision time,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 779–794, 2017.

- [81] R. Mittal, T. Lam, N. Dukkupati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, “Timely: Rtt-based congestion control for the datacenter,” in *Sigcomm ’15*, 2015.
- [82] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, “Revisiting network support for rdma,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 313–326.
- [83] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, “Homa: A receiver-driven low-latency transport protocol using network priorities,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: ACM, 2018, pp. 221–235. [Online]. Available: <http://doi.acm.org/10.1145/3230543.3230564>
- [84] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. M. Tung, and V. Venkataramani, “Scaling memcache at facebook,” in *NSDI*, 2013.
- [85] Nvidia, “Nvidia dgx a100 the universal system for ai infrastructure,” Nvidia, Tech. Rep., 2020. [Online]. [Online]. Available: <https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-infographic.pdf>
- [86] Nvidia, “Nvidia quantum-2 qm9700 series scaling out data centers with 400g infiniband smart switches,” *nvidia.com*, 2021. [Online]. Available: <https://nvdam.widen.net/s/k8sqcr6gzb/infiniband-quantum-2-qm9700-series-datasheet-us-nvidia-1751454-r8-web>
- [87] —, “Nvidia spectrum sn4000 open ethernet switches,” *nvidia.com*, 2021. [Online]. Available: <https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/>
- [88] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, “Fastpass: A centralized ”zero-queue” datacenter network,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14. New York, NY, USA: ACM, 2014, pp. 307–318. [Online]. Available: <http://doi.acm.org/10.1145/2619239.2626309>
- [89] C. Peterson, J. Sutton, and P. Wiley, “iwarp: a 100-mops, liw microprocessor for multicomputers,” *IEEE Micro*, vol. 11, no. 3, pp. 26–29, 1991.
- [90] G. F. Pfister and V. A. Norton, ““hot spot” contention and combining in multistage interconnection networks,” *IEEE Transactions on Computers*, vol. 100, no. 10, pp. 943–948, 1985.

- [91] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, “Faircloud: Sharing the network in cloud computing,” *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, p. 187–198, Aug. 2012. [Online]. Available: <https://doi.org/10.1145/2377677.2377717>
- [92] K. Ramakrishnan, S. Floyd, and D. Black, “Rfc3168: The addition of explicit congestion notification (ecn) to ip,” 2001.
- [93] A. Randles, E. W. Draeger, T. Ooppelstrup, L. Krauss, and J. A. Gunnels, “Massively parallel models of the human circulatory system,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–11.
- [94] J. Rexford, F. Bonomi, A. Greenberg, and A. Wong, “A scalable architecture for fair leaky-bucket shaping,” in *Proceedings of INFOCOM’97*, vol. 3. IEEE, 1997, pp. 1054–1062.
- [95] G. F. Riley and T. R. Henderson, “The ns-3 network simulator.” in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Günes, and J. Gross, Eds. Springer, 2010, pp. 15–34. [Online]. Available: <http://dblp.uni-trier.de/db/books/collections/Wehrle2010.htmlRileyH10>
- [96] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler, “Redmark: Bypassing RDMA security mechanisms,” in *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger>
- [97] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, “Scaling distributed machine learning with in-network aggregation,” *arXiv preprint arXiv:1903.06701*, 2019.
- [98] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, “Approximating fair queueing on reconfigurable switches,” in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’18. USA: USENIX Association, 2018, p. 1–16.
- [99] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, “Sharing the data center network,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11. USA: USENIX Association, 2011, p. 309–322.
- [100] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang, “Securing RDMA for high-performance datacenter storage systems,” in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, Jul.

2020. [Online]. Available: <https://www.usenix.org/conference/hotcloud20/presentation/kornfeld-simpson>
- [101] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI—the Complete Reference: the MPI core*. MIT press, 1998, vol. 1.
- [102] I. Stoica, S. Shenker, and H. 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*, 1998, pp. 118–130.
- [103] P. Taheri, D. Menikkumbura, E. Vanini, S. Fahmy, P. Eugster, and T. Edsall, “Rocc: Robust congestion control for rdma,” in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 17–30. [Online]. Available: <https://doi.org/10.1145/3386367.3431316>
- [104] M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee, “Self-tuned congestion control for multiprocessor networks,” in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 107–118.
- [105] T. W. Tillson, “A hamiltonian decomposition of k_{2m}^* , $2m \geq 8$,” *Journal of Combinatorial Theory, Series B*, vol. 29, no. 1, pp. 68–74, 1980.
- [106] J. S. Turner, “New directions in communications (or which way to the information age?),” *IEEE Communications Magazine*, vol. 40, no. 5, pp. 50–57, 2002.
- [107] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, “Safe and effective fine-grained TCP retransmissions for datacenter communication,” in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, ser. SIGCOMM ’09. New York, NY, USA: ACM, 2009, pp. 303–314. [Online]. Available: <http://doi.acm.org/10.1145/1592568.1592604>
- [108] H. Zeng, J. Bagga, G. Porter, and A. Snoeren, “Inside the social network’s (datacenter) network,” *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 123–137, 08 2015.
- [109] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye, “Ecn or delay: Lessons learnt from analysis of dcqcn and timely,” in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’16. New York, NY, USA: Association for

Computing Machinery, 2016, p. 313–327. [Online]. Available: <https://doi.org/10.1145/2999572.2999593>

- [110] Y. Zhu, M. Zhang, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, and M. Yahia, “Congestion control for large-scale rdma deployments,” *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 523–536, 08 2015.

Biography

John Snyder earned his B.S. from Rhodes College in 2018 and a Ph.D. in Computer Science from Duke University in 2022.

During his third year at Rhodes College, he won the Computer Science Award, given to the top computer science student at Rhodes College each year. At Duke University, the Computer Science Department awarded him the Outstanding Teaching Award for his work as a TA for an undergraduate Operating Systems course.