# Fast Convergence to Fairness for Reduced Long Flow Tail Latency in Datacenter Networks

John Snyder and Alvin R. Lebeck
Department of Computer Science
Duke University
Email: {jsnyder, alvy}@cs.duke.edu

***Abstract*—Many data-intensive applications, such as distributed deep learning and data analytics, require moving vast amounts of data between compute servers in a distributed system. To meet the demands of these applications, datacenters are adopting Remote Direct Memory Access (RDMA), which has higher bandwidth and lower latency than traditional kernel-based networking. To ensure high performance of RDMA networks, congestion control manages queue depth on switches, and historically focused on moderating queue depth to ensure small flows complete quickly. Unfortunately, one side-effect of many common decisions is that large flows are starved of bandwidth. This negatively impacts the flow completion time (FCT) of large, bandwidth-bound flows, which are integral to the performance of data-intensive applications. The FCT is particularly impacted at the tail, which is increasingly critical for predictable application performance. We identify the root causes of the poor performance for long flows and measure the impact. We then design mechanisms that improve long flow FCT without compromising small flow performance. Our evaluations show that these improvements reduce 99.9% tail FCT of long flows by over 2x.**

## I. Introduction

In 2018, users collectively created 2.5 quintillion bytes a day, and the amount of data created was growing rapidly [22]. To store and synthesize this data, data owners use frameworks like Apache Spark [28] and Tensorflow [2]. These frameworks utilise distributed systems to meet the application's computational and storage demands. Running an application in a distributed system enables the application to scale but may create new performance bottlenecks.

Application performance relies on network performance in distributed systems. If servers spend excessive time communicating over the network, then application performance suffers because servers wait for messages instead of computing. The key metric for network performance is Flow Completion Time (FCT) [10]. A flow is a sequence of data packets from a source to a destination, and the FCT measures how long it takes for the network to deliver those packets. We particularly want to manage the tail FCT in a distributed application [9]. In a set of flows, the tail is a high percentile of the flow completion times.

Historically, distributed systems optimize FCT with two metrics: low packet latency and high network throughput. Low packet latency allows small flows to complete quickly and high throughput enables large flows to send packets as fast as possible without causing congestion. Recently, datacenters adopted Remote Direct Memory Access (RDMA) hardware to remove system software from the critical path of network operations, which improves packet latency and network throughput.

Congestion control protocols manage packet queues on switches, which enables low packet latency and high throughput in RDMA networks. Lower queueing delay on switches reduce packet latency because packets spend less time sitting in queues and more time traversing the network. RDMA networks are generally lossless, which means they can suffer from Head-of-Line (HoL) blocking. HoL blocking occurs when switch packet buffers fill and upstream switches cannot send packets due to insufficient space. Congestion control reduces the likelyhood of HoL blocking because packet buffers are less likely to fill with packets and block the network. Numerous congestion control algorithms exist for RDMA networks that achieve these two goals [18], [20], [23], [24], [31].

We identify a third metric that influences FCT, which is largely ignored by previous work: *convergence to fairness*. Most protocols are provably fair [18], [20], [30], but few optimize to converge quickly to fair rates. Convergence to fairness dictates how quickly an unfair allocation of bandwidth between end-hosts becomes fair. 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.

We improve convergence to fair rate allocations by identifying three reasons why many existing congestion control protocols converge to fair rates slowly: 1) conservative additive increase 2) reducing rates once per RTT and 3) deterministic feedback. We then identify that the bandwidth allocation often becomes unfair when a new flow joins the network because new flows in RDMA networks often start sending packets at line rate [20], [23], [31]. In addition to creating unfairness, new flows joining the network create congestion, so we infer that network congestion indicates the bandwidth allocation may be unfair. Based on these two observations, we introduce two new mechanisms: Variable Additive Increase and Sampling Frequency, which significantly improve long flow tail FCT.

We augment Swift [18] and HPCC [20] with our new mechanisms in ns-3 [25] and show how they perform in micro-benchmarks and datacenter simulations. While using our mechanisms, we achieve near zero extra queueing delay, improve convergence to fairness, and maintain high throughput. When we improve the convergence to fair rate allocations

in HPCC and Swift, the tail FCT of long flows decreases by 2x without affecting small flow performance.

We provide background information on congestion control in Section II. In Section III, we identify the sources of slow convergence to fairness in HPCC and Swift and do a case study measuring slow convergence. Next we introduce our mechanisms and describe their implementation in Sections IV and V, respectively. We evaluate our new mechanisms in Section VI, and we conclude in Section VII.

## II. Background

There are numerous methods to calculate rates in a congestion control protocol. Network switches in some protocols (RCP [10], XCP [17], s-PERC [16], RoCC [26]) calculate packet injection rates for flows and explicitly add that information to packet headers. While these protocols are effective, they require complex switches and are not common. Other protocols schedule flows at the receiver [13], [15], [24], but they are uncommon because they rely on a non-oversubscribed network. The most common type of congestion control is sender-side reaction protocols [18], [20], [23], [31]. These protocols observe network state through mechanisms like Round Trip Time (RTT) measurements, Explicit Congestion Notification (ECN), and In-band Network Telemetry (INT). If the sender observes congestion, it reduces its injection of packets into the network. These protocols generally use Additive-Increase Multiplicative-Decrease (AIMD) to converge to fair rates [8], [20].

Our work focuses on providing additional mechanisms to sender-side protocols to improve convergence to fairness. We augment two protocols: Swift [18] and HPCC [20]. Both protocols effectively minimise packet latency for small flows but fail to allocate bandwidth fairly to large flows, which extends the tail FCT of long flows.

Swift uses RTT measurements to detect congestion severity and then reduces packet injection based on the degree of network congestion [18]. If Swift detects no congestion, Swift increases the injection rate of packets additively. Swift also pioneered Flow-based Scaling (FBS), which improves fairness and is therefore relevant to our work. Swift calculates the multiplicative decrease factor (mdf) with the equation

$$\text{mdf} = \max(1 - \beta * \frac{\text{Delay} - \text{Target Delay}}{\text{Delay}}, \text{max_mdf}) \quad (1)$$

where $0 < \beta, \text{max_mdf} < 1$. The larger the difference between the delay and target delay, the more severe the decrease. Target delay is not fixed; Swift changes target delay based on the injection rate (or congestion window) of the end host. The process of changing the target delay is called Flow-based Scaling (FBS). The lower the congestion window, the higher FBS sets the target delay. If there is congestion, FBS reduces the difference between delay and target delay and therefore reduces the mdf.

Li et al. [20] observed that a cloud storage system caused large packet queues on switches, which negatively impacted the latency sensitive flows in a deep learning framework running in the same cluster. They wanted a congestion control algorithm that achieved near zero queues to avoid this issue. To minimize queueing delay, they utilize In-band Network Telemetry (INT), a new technology in some P4 [7] switches. Switches with INT can add telemetry data to packets as the packets pass through the switch. HPCC uses three pieces of telemetry from network switches to determine the extent of congestion (queue depth) and available bandwidth (transmitted bytes and timestamps). With these metrics, HPCC achieves near zero queueing delay and high bandwidth utilization.

While Swift and HPCC are effective congestion control protocols, unfortunately they fail to fairly allocate bandwidth to flows of all sizes. This negatively impacts the FCT of long flows.

DCQCN [31] is a data center congestion control protocol for RDMA networks that uses Random Early Detection (RED) marking to indicate congestion using ECN. RED marks packets as experiencing congestion randomly if the packet queue on the switch exceeds a threshold. If a receiver receives an ECN marked packet, it sends a Congestion Notification Packet (CNP) to the sender and the sender reduces its injection of packets into the network. DCQCN does not suffer from unfairness like Swift and HPCC because RED marking makes it more likely for flows with more bandwidth to reduce their rates.

## III. 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. Identifying these sources of unfairness is a critical step toward mitigating their impact.

### A. Conservative Additive Increase

AIMD converges to fairness by generating congestion events that trigger multiplicative decrease [8], [11]. In the abscence of congestion, all flows increase their rates. The sum of all flow rates eventually exceeds the bandwidth of the bottleneck link because each flow is increasing 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 [5], [18], [20]. If congestion 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 slower [8]. If a protocol 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.*

### B. One Reaction Per RTT

HPCC and Swift fully react to at most one congestion signal per RTT [18], [20]. This enables the protocols to not react 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, one of which is allocated 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 any packets received in that RTT were marked ECN, both flows divide their rates by half. However, if the protocol reacted to every ECN marked packet instead of every RTT, then the rate 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.*

### C. Deterministic Feedback

Probabilistic feedback is more fair than deterministic feedback. To improve fairness in DCQCN, Zhu, et al. [31] use probabilistic feedback and 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. [12] showed that infrequent congestion signals can lead to poor performance during Incast traffic.

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. Flow-based Scaling in Swift addresses this, but it is insufficient to improve tail FCT of long flows, which we demonstrate with experiments in Section III-E [18].

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

### D. Methodology

We demonstrate unfairness in an unmodified version of HPCC and a slightly modified version of Swift. We use ns-3 [25] code provided as an artifact from HPCC [1] to evaluate HPCC and Swift. We run a 16-1 incast trafic pattern, which is important to datacenter networks [27]. We expand to datacenter simulations and larger incast patterns in Section VI. 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. Two flows start every 20 microseconds and each flow sends 1MB. Each link has $1\mu$s of propagation delay.

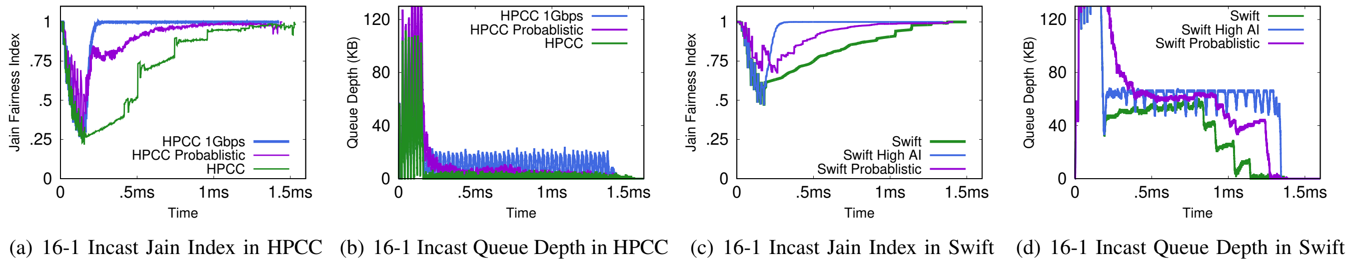
We use the parameters suggested by Li et al. [20] when evaluating HPCC. We set Additive Increase to 50 Mb/s, utilization rate to 0.95, and increment stage to 5. We refer to this set of parameters as ”default HPCC”. In the figures, these settings are labelled as ”HPCC”. We also run HPCC with a higher 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 probablistic feedback, where feedback from the network is sometimes ignored. Feedback is disregarded based on the following equation:

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

This creates a linear equation for the likelihood a packet is ignored as a function of current window size. [1] 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 Probablistic”.

Swift does not suggest settings for multiple parameters, so we set reasonable parameters. Like HPCC, we set additive increase to 50 Mb/s. We set $\beta$ (see Equation 1) to 0.8 and the maximum mdf to 0.5. For FBS, we use the parameters in Kumar et al. [18], except when we are running on the smaller topology because the window is smaller, so we lower the max target scaling window from 100 to 50 packets. For Topology-based Scaling, we set the base RTT to $5\mu$s, and add $2\mu$s of

[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

Fig. 1. Jain Fairness Index (closer to 1 is better) and Queue depth during Incast Traffic in HPCC and Swift

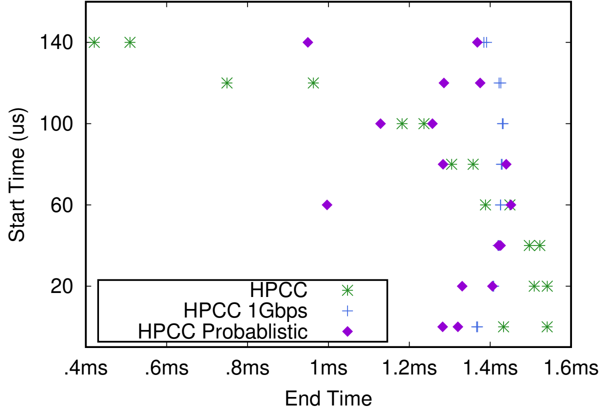



Fig. 2. Start Time vs. Finish Time 16-1 Staggered Incast HPCC

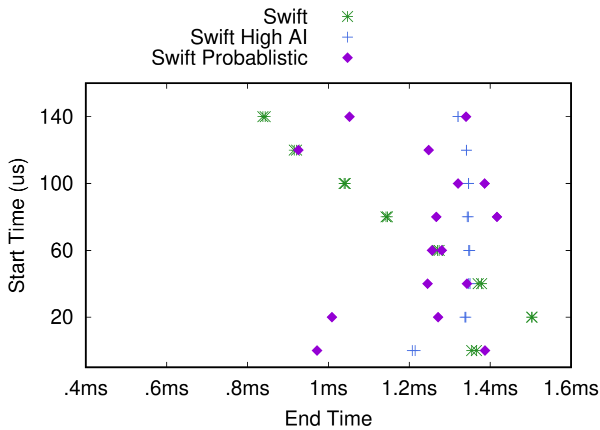



Fig. 3. Start Time vs. Finish Time 16-1 Staggered Incast in Swift

delay per hop. We also run Swift with probablistic feedback and a 1Gbps AI to show the effect on fairness. Swift does not define flow start behavior, so we start flows at line rate in Swift to fit with other RDMA congestion control protocols.

### *E. Unfairness in HPCC and Swift*

HPCC and Swift exhibit the previously discussed characteristics that create unfairness. Figures 2 and 3 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 probablistic feedback eliminates this trend and the flows finish at generally the same time.

Figures 1(a) and 1(c) plot the commonly used Jain Fairness Index over time [14]. When the Jain Fairness Index is closer to one, the rates are more fair. A lower fairness index indicates an unfair allocation. Using the default parameter settings, both Swift and HPCC take several hundred microseconds to get close to an index of one. Using probablistic feedback and a higher AI improves convergence to fair rates in both protocols. However, this has negative effects. Figures 1(b) and 1(d) show both congestion control protocols have larger queue oscillations with a higher additive increase.

## IV. Mechanisms for Improved Fairness

The previous results demonstrate how state-of-the-art congestion control techniques trade convergence to fairness in favor of latency and throughput. In this section we introduce Variable Additive Increase and Sampling Frequency: two new mechanisms that raise fairness to be on-par with latency and throughput and can be added to many sender-side congestion control protocols. These new mechanisms are based on two key observations from our analysis of 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 the network allocates significantly more bandwidth to new flows joining the network than existing flows because new flows generally start sending packets at line rate in RDMA networks [20], [31]. As a very simple example of this principle, assume 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 $< B$, so no queue builds up. Now 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 $B$. 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 they are allocated 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.

We exploit these two observations to create two new mechanisms to mitigate the sources of unfairness outlined in Section III. The first mechanism is a Variable Additive Increase, which increases the additive increase when the protocol believes the bandwidth allocation is unfair. The second mechanism is Sampling Frequency, which updates a flow's rate after a certain number of ACKs instead of each RTT. 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 [6] to include fairness.*

### A. Variable AI

Variable AI increases the AI value when the protocol detects the rate allocation maybe 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.

**Algorithm 1** Generating Tokens and Setting Dampener

```
1:  if RTT Finished then
2:      if Measured Congestion > Token_Thresh then
3:          AI_Bank=min(Meas. Cong. / AI_DIV + AI_Bank, Bank_Cap)
4:      end if
5:      if Measured Congestion>Token_Thresh then
6:          dampener += Meas. Cong. / Token_Thresh
7:      else if AI_Bank == 0 then
8:          if No Congestion then
9:              dampener = 0
10:         else if Measured Congestion < Token_Thresh then
11:             dampener = max(dampener-1, 0)
12:         end if
13:     end if
14:     Measured Congestion = 0
15: end if
```

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

**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: end if
6: divisor = (dampener / Dampener_Constant) + 1
7: tokens = max(tokens / divisor, 1)
8: Additive Increase = tokens * base_AI
```

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 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.

Feedback is one potential issue with Variable AI. Additive increase causes queue oscillations, and those oscillations are larger if additive increase is larger. 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 only resets 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 [4], [21]. 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 is only trying to force small congestion events to improve fairness.

### *B. Sampling Frequency*

Sampling Frequency tunes how often a protocol reacts to congestion signals. Section III-B detailed how reacting only once per round trip time 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.

This parameter allows users of the protocol to choose between improving fairness and tail latency at the expense of bandwidth. 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 affect 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 [31] and Zhu et al [30]. We model Sampling Frequency with the equation

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

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 $\beta$, 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

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

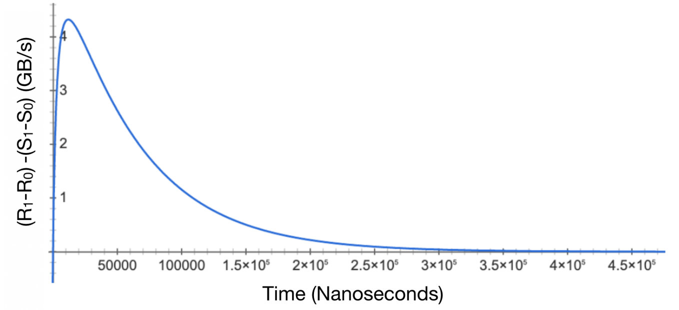



Fig. 4. Plotting difference in fairness between to MD methods. r = 30000, MTU=1000, s = 30, $\beta$=.5

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

$$(R_1(t) - R_0(t)) - (S_1(t) - S_0(t)) > 0 \tag{2}$$

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 constrains 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.

To demonstrate, we graph equation 2 with reasonable parameters and show how the rates converge during congestion in Figure 4. We use bytes per nanosecond as our units for rates, 30,000 for the observed network RTT (r), 30 for the Sampling Frequency (s), 1,000 bytes for packet MTU, and .5 for $\beta$. The initial rates for the flows are 100Gbps and 50Gbps. Sampling Frequency converges much faster to fair rates than using a per-RTT decrease, which leads to a positive fairness difference. Over time the fairness difference diminishes, but the goal is to converge to nearly fair rates quickly, which the mechanism achieves.

## V. 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 Section III.

### A. 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.

### B. Sampling Frequency

Normally, Swift and HPCC wait one-RTT between rate updates. Sampling Frequency instead sets how many acknowledgements between multiplicative decreases. We amend Swift and HPCC to decrease their rates after a predetermined number of acknowledgements. 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 change 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 rate adjustments occur from, which is updated once per-RTT. Using Sampling Frequency the ”reference rate” is updated per-sampling period, which is a configurable number of acknowledgements. The protocol updates the actual injection rate after every acknowledgement. The per-ACK rate changes based on the ”reference rate” and not on other per-ACK updates. 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.

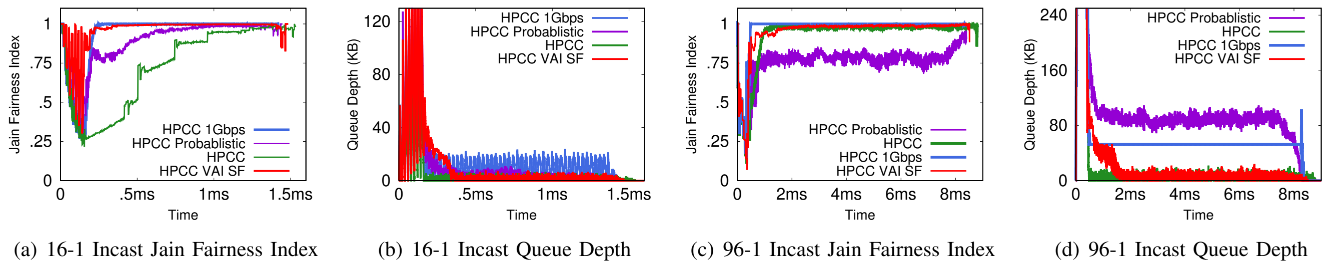
The second change is to always perform an additive increase regardless of congestion like in HPCC. This improves the functionality of Variable AI since the tokens are always spent. This can have a small latency penalty, but it should only incur an additional latency equal to when latency causes the MD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

## VI. Evaluation

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

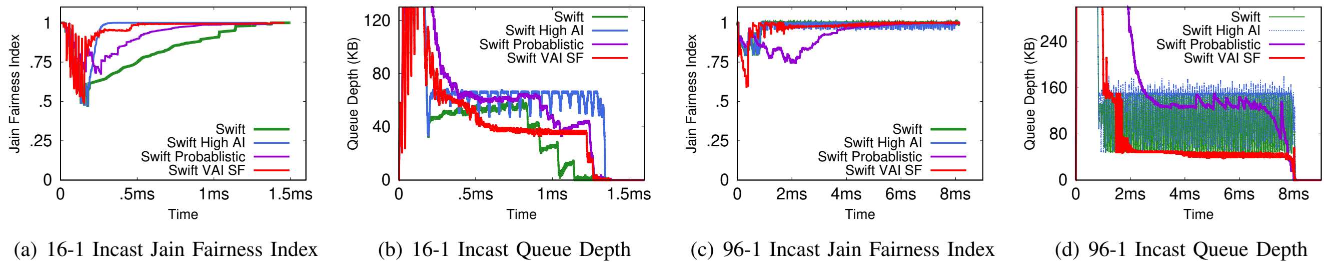
### A. Methodology

We use the same parameters as Section III-D. 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 [29]. The hadoop traffic contains mostly small flows (95% $<$300KB) and a small number of large flows (2.5% $>$ 1MB). The second datacenter benchmark is flow size distributions from two applications mixed together to simulate a shared




(a) 16-1 Incast Jain Fairness Index (b) 16-1 Incast Queue Depth (c) 96-1 Incast Jain Fairness Index (d) 96-1 Incast Queue Depth

Fig. 5. Jain Fairness Index (closer to 1 is better) and Queue depth during Incast Traffic with HPCC




(a) 16-1 Incast Jain Fairness Index (b) 16-1 Incast Queue Depth (c) 96-1 Incast Jain Fairness Index (d) 96-1 Incast Queue Depth

Fig. 6. Jain Fairness Index (closer to 1 is better) and Queue depth during Incast Traffic with Swift

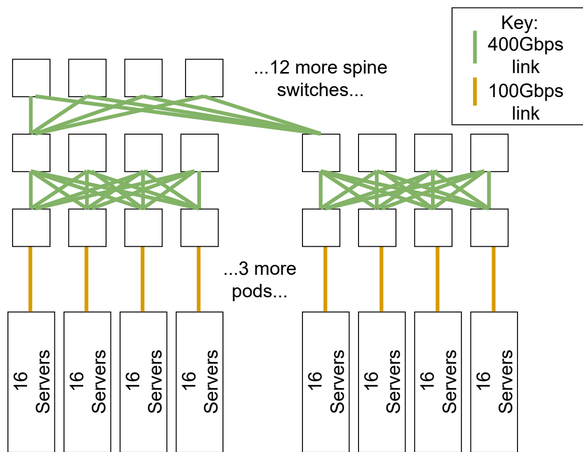



Fig. 7. Fat-tree topology used in simulations

environment. 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 (96% $<$128KB and 100% $<$ 2MB). The full flow size distributions for each benchmark are publicly available [1]. The datacenter benchmarks run the network at 50% load for 50ms.

For the datacenter simulation, we use the same topology as Li et al. [20]. The simulation has 320 end-hosts connected in a 3 layer fat-tree [3], [19]. Figure 7 shows the topology we use in simulations. Each host has a 100Gbps link to its ToR switch. There are five 2-layer pods; each pod has 4 Agg switches and 4 ToR switches, so there are a 20 Agg switches and 20 ToR switches. There are 16 spine switches. The ToR to Agg and Agg to Spine connections have 400Gbps links. The maximum number of hops between two hosts is 5, and each link's propagation delay is $1\mu$s.

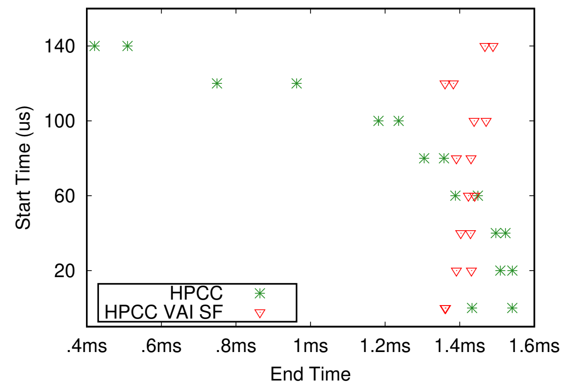



Fig. 8. 16-1 Incast Traffic Start Time vs Finish Time with HPCC

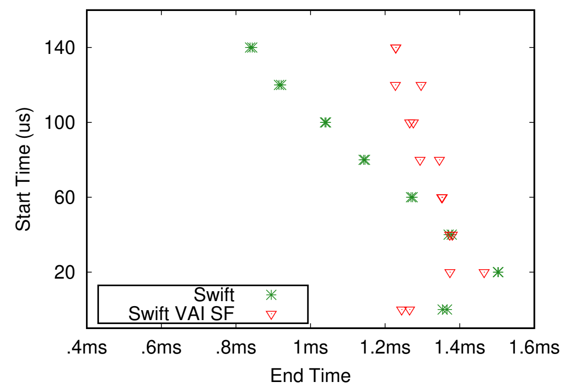



Fig. 9. 16-1 Incast Traffic Start Time vs Finish Time with Swift

For Sampling Frequency, 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. 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.

### B. Experimental Results

*1) Incast:* First, we run the 16-1 incast congestion and compare our VAI SF variants with the existing baselines. Figures 8 and 9 show the start versus finish time of HPCC and Swift with default settings versus their VAI SF variants. Omitting the other baselines avoids clutter. The finish time of the flows is much closer together when using our mechanisms. Figures 5(a) and 6(a) 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 probablistic variants. Figure 5(b) shows that when using VAI and SF, HPCC still maintains near 0 queues. Figure 6(b) 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 queueing delay. Swift VAI SF also has small queue oscillations.

The same trends continue when we scale the incast to 96-1. Figures 5(c) and 6(c) show that when using VAI and SF, the system becomes fair quickly. Figure 5(d) 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 6(d), Swift maintains the smallest queue because it does not use FBS, and smaller oscillations because it has a small AI in the steady state.

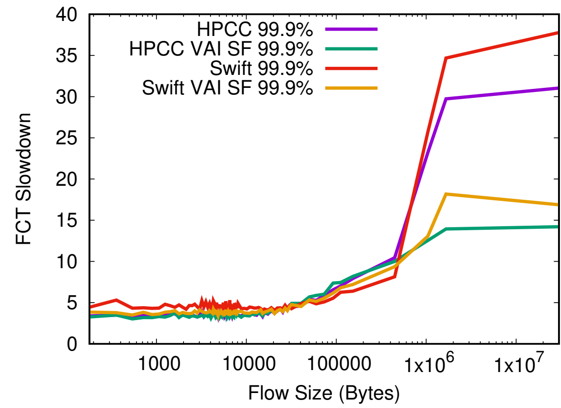



Fig. 11. 99.9% FCT for various flow sizes in WebSearch and Storage Traffic

*2) Datacenter Simulations:* We found that a slow convergence to fair rates significantly impacted the performance of long flows, particularly at the tail. Figure 10 shows how the unfairness affects long flows in a Hadoop traffic pattern. We plot the FCT slowdown as a function of flow size and each data point represents 1% of flows. 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.

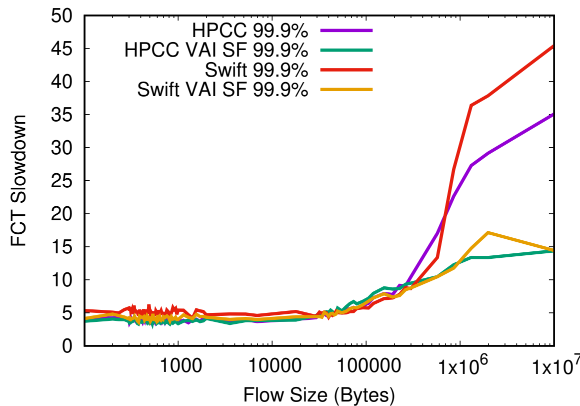



Fig. 10. 99.9% FCT for various flow sizes in Hadoop Traffic

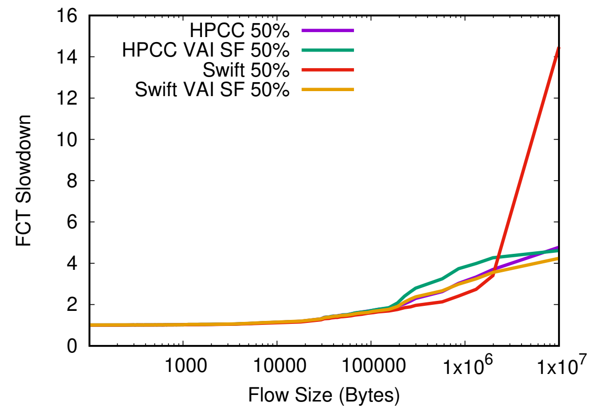



Fig. 12. Median FCT for various flow sizes in Hadoop Traffic

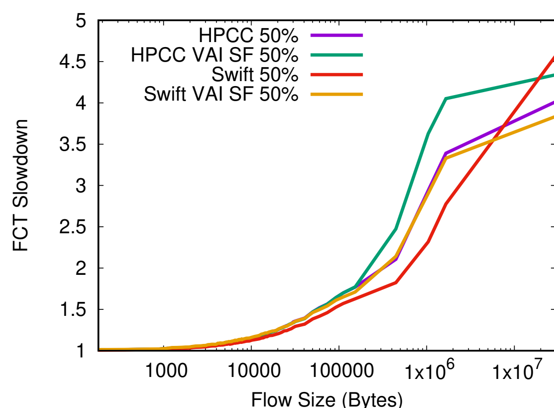



Fig. 13. Median FCT for various flow sizes in WebSearch and Storage Traffic

Figure 11 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.

VAI and SF improve the tail FCT with no significant repercussions on median FCT. Figures 12 and 13 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 [23], which can help grab available bandwidth. Raising the target delay improves long flow FCT, but increases short flow FCT. Our target delay covers the network's serialization and propagation delay, so it is reasonable.

## VII. Conclusion

Data-intensive applications require large amounts of bandwidth to move data within a distributed system. While the physical infrastructure enables this, the congestion control algorithms deployed in datacenters favor small flows, which harms large flow FCT. We identify that 1) conservative additive increase 2) one reaction per-RTT and 3) deterministic feedback cause some protocols to converge slowly to fair rates. We observe that unfairness occurs when a new flow joins the network, and we can infer unfairness at the end-host by an increase in congestion. To exploit these observations and improve convergence to fairness, we create two generic mechanisms that can improve fairness in sender-side reaction based protocols: 1) Variable Additive Increase and 2) Sampling Frequency. Adding these mechanisms to HPCC and Swift improves convergence to fairness while still maintaining small queues and high throughput. Variable AI and Sampling Frequency could be used with a multitude of congestion control algorithms and require minimal changes on end hosts. Our mechanisms significantly improve convergence to fairness, almost no impact on queue size during incast or datacenter traffic, and improve large flow tail FCT by 2x.

## VIII. Acknowledgements

We thank Danyang Zhuo and the anonymous reviewers for their helpful feedback on this work. This work was supported in part by the National Science Foundation (CNS-1616947).

## References

[1] "https://github.com/alibaba-edu/High-Precision-Congestion-Control," Jul. 2020.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[3] 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

[4] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data center transport mechanisms: Congestion control theory and ieee standardization," in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, 2008, pp. 1270–1277.

[5] 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

[6] 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.

[7] 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

[8] 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

[9] 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

[10] N. Dukkipati 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

[11] 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

[12] Y. Gao, Y. Yang, T. Chen, J. Zheng, B. Mao, and G. Chen, "Dcqcn+: Taming large-scale incast congestion in rdma over ethernet networks," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, pp. 110–120.

[13] 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

[14] 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.

[15] 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

[16] L. Jose, S. Ibanez, M. Alizadeh, and N. McKeown, "A distributed algorithm to calculate max-min fair rates without per-flow state," in *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 57–58. [Online]. Available: https://doi.org/10.1145/3309697.3331472

[17] 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*, ser. SIGCOMM '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 89–102. [Online]. Available: https://doi.org/10.1145/633025.633035

[18] G. Kumar, N. Dukkipati, 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

[19] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, 1985.

[20] 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

[21] 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.

[22] B. Marr, "How much data do we create every day? the mind-blowing stats everyone should read," *Forbes*, May 2018.

[23] R. Mittal, T. Lam, N. Dukkipati, 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.

[24] 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

[25] 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

[26] 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

[27] 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

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

[29] 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.

[30] 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

[31] 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.