Bolt: Sub-RTT Congestion Control for Ultra-Low Latency

Serhat Arslan* *Stanford University*
Yuliang Li *Google LLC*
Gautam Kumar *Google LLC*
Nandita Dukkipati *Google LLC*

Abstract

Data center networks are inclined towards increasing line rates to 200Gbps and beyond to satisfy the performance requirements of applications such as NVMe and distributed ML. With larger Bandwidth Delay Products (BDPs), an increasing number of transfers fit within a few BDPs. These transfers are not only more performance-sensitive to congestion, but also bring more challenges to congestion control (CC) as they leave little time for CC to make the right decisions. Therefore, CC is under more pressure than ever before to achieve minimal queuing and high link utilization, leaving no room for imperfect control decisions.

We identify that for CC to make quick and accurate decisions, the use of precise congestion signals and minimization of the control loop delay are vital. We address these issues by designing Bolt, an attempt to push congestion control to its theoretical limits by harnessing the power of programmable data planes. Bolt is founded on three core ideas, (i) Sub-RTT Control (SRC) reacts to congestion *faster* than RTT control loop delay, (ii) Proactive Ramp-up (PRU) *foresees* flow completions in the future to promptly occupy released bandwidth, and (iii) Supply matching (SM) explicitly matches bandwidth demand with supply to maximize utilization. Our experiments in testbed and simulations demonstrate that Bolt reduces 99^{th}-p latency by 80% and improves 99^{th}-p flow completion time by up to 3× compared to Swift and HPCC while maintaining near line-rate utilization even at 400Gbps.

1 Introduction

Data center workloads are evolving towards highly parallel, lightweight applications that perform well when the network can provide low tail latency with high bandwidth [5]. Accordingly, the Service Level Objectives (SLOs) of applications are becoming more stringent, putting increasing responsibility on network performance. To support this trend, the industry is inclined towards increasing line rates. 100Gbps links are already abundant, 200Gbps is gaining adoption, and industry standardization of 400Gbps ethernet is underway [24].

*Work done as a student researcher at Google

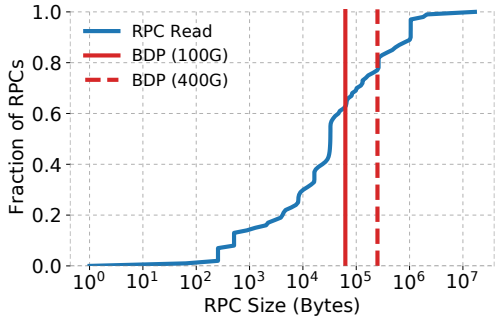


Figure 1: RPC size distribution for READ operations

With the increasing line rates, CC needs to make decisions with *higher quality and timeliness* over *a burstier workload*. We illustrate this based on a recent analysis of RPC sizes in our data centers with respect to BDP sizes at 100Gbps and 400Gbps (calculated using a typical base delay/RTT in data centers). Our findings are presented in Figure 1.

The fraction of RPCs that fit within 1 and 4 BDP increases from 62% and 80% at 100Gbps to 80% and 89% at 400Gbps. These RPCs are performance-sensitive to queuing and under-utilization. Ultimately, even a single incorrect or slow CC decision may end up creating tens of microseconds of tail queuing [12], or cause under-utilization [53] which prolongs the flow completion time by a few RTTs. Therefore, an increasing fraction of such RPCs raises the bar for the quality and timeliness of CC.

Concomitantly, at higher bandwidth, the workload becomes burstier and thus harder to control. Figure 1 also reveals that a 400Gbps link with just 40% load sees an RPC arrival or completion roughly every RTT! Hence, it becomes more difficult to control queuing and under-utilization as they arrive and finish quickly at RTT timescales. We expect these numbers to be even more challenging for upcoming workloads such as disaggregated memory and ML.

We identify two key aspects of CC that are important to address the challenges of achieving higher CC quality and timeliness on burstier workloads:

First, **granular feedback** about the location and severity of

congestion allows avoiding over/under-reaction [3]. A precise CC algorithm would receive the exact state of the bottleneck to correctly ramp down during congestion and ramp up during under-utilization. This congestion information would intuitively involve telemetry such as the current queue occupancy and a measure of link utilization [35]. Then, end-hosts would be able to calculate the exact number of packets they can inject into the network without creating congestion.

Second, the **control loop delay** is a determinant of how sensitive a control algorithm can be. It is defined as the delay between a congestion event and the reaction from the senders arriving at the bottleneck. Smaller the control loop delay, the more accurate and simpler decisions a control system can make [41]. The state-of-the-art CC algorithms in production are reported to work well to the extent their control loop delay allows [30, 35, 60]. However, even a delay of one RTT will be too long to tolerate for future networks because of the increasing BDPs [58]. We conjecture that the inevitable next step is to reduce the control loop delay to sub-RTT levels.

Fortunately, the flexibility and precision provided by programmable switches [7, 11, 22] allow designing new mechanisms to reduce the control loop delay and increase the granularity of control algorithms. These state-of-the-art switches can generate custom control signals to report fine-grained telemetry so that flows don't need to rely on end-to-end measurements for detecting congestion at the bottleneck link.

In this work, we present Bolt, our effort of harnessing the power of programmable data planes to design an extremely precise CC for ultra-low latency at very high line rates. Bolt collects congestion feedback with absolute minimum (sub-RTT) delay and ramps up flows proactively to occupy available bandwidth promptly. To achieve this, it applies the "packet conservation" principle [25] onto the traffic with accurate per-packet decisions in P4 [9]. Small per-packet `cwnd` changes, combined with the fine-grained in-network telemetry, help limit the effects of noise in the instantaneous congestion signal. With Bolt, end-hosts do not make implicit estimations about the severity and exact location of the congestion or the number of competing flows, freeing them from manually tuned hard coded parameters and inaccurate reactions.

The main contributions of Bolt are:

1. A discussion for the fundamental limits of an optimal CC algorithm with minimal control loop delay.
2. Description of 3 mechanisms that collectively form the design of Bolt – an extremely precise CC algorithm with the shortest control loop possible.
3. Implementation and evaluation of Bolt on P4 switches in our lab which achieves 86% and 81% lower RTTs compared to Swift [30] for median and tail respectively.
4. NS-3 [48] implementation for large scale scenarios where Bolt achieves up to 3× better 99^{th}-p flow completion times compared to Swift and HPCC [35].

The remainder of the paper describes the rationale behind the design of Bolt in §2, design details in §3, and implementation insights in §4. Further evaluations and benchmarks are provided in §5 followed by practical considerations in §6. Finally, a survey of related work is presented in §7.

2 Towards Minimal Control Loop Delay

Timely feedback and reaction to congestion are well understood to be valuable for CC [42]. With Bolt, we aim to push the limits on minimizing the control loop delay that is composed of two elements: (1) *Feedback Delay* (§2.1) is the time to receive any feedback for a packet sent, and (2) *Observation Period* (§2.2) is the time interval over which feedback is collected before `cwnd` is adjusted. Most CC algorithms send a window of packets, observe the feedback reflected by the receiver over another window, and finally adjust the `cwnd`, having a total control loop delay that is even longer than an RTT [1, 10, 19, 30, 35, 60]. In this section, we describe both *Feedback Delay* and *Observation Period* in detail and discuss how these elements can be reduced to their absolute minimum motivating Bolt's design in §3.

2.1 Feedback Delay

There are two main types of feedback to collect for congestion control purposes: (i) *Congestion Notification* and (ii) *Under-utilization Feedback*.

2.1.1 Congestion Notification

The earliest time a CC algorithm can react to drain a queue is when it first receives the notification about it. Traditionally, congestion notifications are reflected by the receivers with acknowledgments [1, 8, 30, 35, 42, 47, 60]. We call this the RTT-based feedback loop since the delay is exactly one RTT.

To demonstrate how notification delay affects performance, we run an experiment where the congestion notification is delivered to the sender after a constant configured delay (and not via acknowledgments). Setting this delay to queuing delay plus the propagation time in the experiment is equivalent to RTT-based control loops described above. The experiment runs two flows with Swift CC [30] on a dumbbell topology[1] where the second flow joins while the first one is at a steady state. The congestion signal is the RTT the packet will observe with current congestion. Figure 2 (left) shows the time to drain the congested queue for different notification delays. Clearly, smaller notification delay helps mitigate congestion faster as senders react sooner to it.

More importantly, in addition to traveling unnecessary links, traditional RTT-based feedback loops suffer from the congestion itself because the notification waits in the congested queue before it is emitted. Adding the queuing delay

[1] RTT is 8 μs and all the links are 100Gbps.

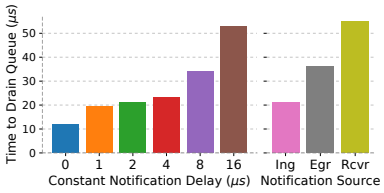
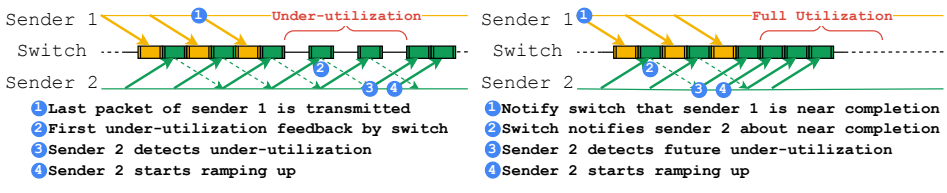


Figure 2: Effect of notification delay on queue draining time



(a) Reactive Ramp-Up **(b)** Proactive Ramp-Up

Figure 3: Under-utilization feedback

to the notification delay hinders tackling congestion even more. During severe congestion events, this extra delay can add multiples of the base RTT to the feedback delay [30].

To understand this more, we also measure the congestion mitigation time of scenarios where the notification is generated at different locations in the network in Figure 2 (right). "Rcvr" represents the RTT-based feedback loop where the congestion notification is piggybacked by the receiver. "Egr" is when the switch sends a notification directly to the sender from the egress pipeline, after the packet waits in the congested queue. "Ing" is when the notification is generated at the ingress pipeline, as soon as a packet arrives at the switch. As expected, generating the congestion notification as soon as possible improves performance by more than 2×.

Correspondingly, we stress that in order to reduce the notification delay to its absolute minimum, the congestion notification should travel directly from the bottleneck back to the sender without waiting in the congested queue.

2.1.2 Under-utilization Feedback

While flow arrival events add to congestion in the network, flow completion events open up capacity to be used by other flows. When a flow completes on a fully utilized link with zero queuing, the packets of the completing flow leave the network and the link will suddenly become under-utilized until the remaining flows ramp up (Figure 3a). As traffic gets more dynamic, such under-utilization events become more frequent, reducing the total network utilization. Therefore, in addition to detecting congestion, a good control algorithm should also be able to detect any under-utilization in order to capture the available bandwidth quickly and efficiently [44].

In practice, CC schemes deliberately maintain a standing queue under a steady state, so that when a flow completes, the packets in the queue can occupy the bandwidth released by the finished flow until the remaining flows ramp up [34, 40]. For example, while HPCC was designed to keep near-zero standing queue, the authors followed up that in practice, HPCC target utilization should be set to 150% to improve network utilization [36], which implies half a BDP worth of standing queue. Other CC schemes used in practice also maintain standing queues by filling up the buffers to a certain level before generating any congestion signal [1, 30, 60].

Figure 4 demonstrates how Swift behaves upon a flow completion when a long enough standing queue is not maintained. There are two flows in the network[2] and one of them completes at $t = 200\mu$s. The remaining flow's `cwnd` takes about 25 RTTs to occupy the released bandwidth as per the additive increase mechanism in Swift. During this time interval, under-utilization happens despite the non-zero queuing at a steady state. This under-utilization can also be observed when there are a larger number of flows if the standing queue size is not adjusted appropriately [53].

Ideally, any remaining flow should immediately capture the `cwnd` of the completing flow without under-utilizing the link. Therefore we conclude that an optimal congestion control algorithm would detect flow completions early enough, **proactively**, to ramp up as soon as the spare capacity becomes available (Figure 3b).

2.2 Observation Period

In addition to the feedback delay, the total control loop delay is usually one RTT longer for window-based data center CC schemes. Namely, once the sender adjusts its `cwnd`, the next adjustment happens only after an RTT to prevent reacting to the same congestion event multiple times. We call this extra delay the *observation period* and illustrate it in Figure 5.

Once-per-window semantics is very common among CC schemes where the per-packet feedback is aggregated into per-window observation. For example, DCTCP [1] counts the number of ECN markings over a window and adjusts `cwnd` based on this statistics once every RTT. Swift compares RTT against the target every time it receives an ACK but decreases `cwnd` only if it has not done so in the last RTT. Finally, HPCC picks the link utilization observed by the first packet of a window to calculate the reference `cwnd` which is updated once per window. As a consequence, flows stick to their `cwnd` decision for an RTT even if the feedback for a higher degree of congestion arrives immediately after the decision.

Updating `cwnd` only once per window removes information about how dynamic the instantaneous load was at any time within the window. This effect, naturally, results in late and/or incorrect congestion control decisions, causing oscillations between under and over-utilized (or congested) links when flows arrive and depart. Consider the scenario[2] in Figure 6

[2]The dumbbell topology from Figure 2 (RTT: 8 μs, 100Gbps links).

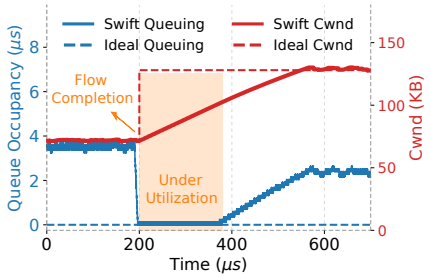


Figure 4: `cwnd` of the remaining Swift flow and queue occupancy after a flow completion.

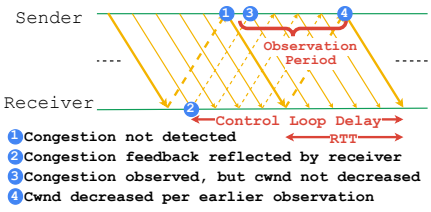


Figure 5: Observation period adds up to an RTT to the control loop delay.

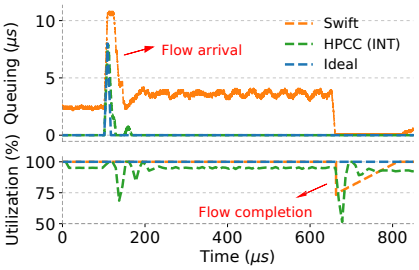


Figure 6: HPCC and Swift's reaction to flow arrival and completion.

where a new flow joins the network at $t = 100\mu$s while another flow is at its steady state. HPCC drains the initial queue built up in a couple of RTTs, but immediately oscillates between under-utilization and queuing for a few iterations. Moreover, the completion of a flow at $t = 650\mu$s again causes oscillations. Under highly dynamic traffic, such oscillations may increase tail latency and reduce network utilization.

An alternative way to avoid oscillations would be to react conservatively similar to Swift. It also reduces `cwnd` only once in an RTT during congestion but uses manually tuned parameters (i.e. *ai* and β) to make sure reactions are not impulsive. Although oscillations are prevented this way, Figure 6 shows that Swift takes a relatively long time to stabilize.

We conclude that once per RTT decisions can lead to either non-robust oscillations or relatively slow convergence. This is especially problematic in high-speed networks where flow arrivals and completions are extremely frequent. Ideally, the shortest observation period would be a packet's serialization time because it is the most granular decision unit for packet-switched networks. Yet, the per-packet CC decisions should only be incremental to deal with the noise from observations over such a short time interval.

3 Design

Bolt is designed for ultra-low-latency even at very high line rates by striving to achieve the ideal behavior shown in Figures 4 and 6. The design aims to reduce the control loop delay to its absolute minimum as described in §2. *First*, the congestion notification delay is minimized by generating notifications at the switches and reflecting them directly to the senders (§3.1). *Second*, the flow completion events are signaled by the senders in advance to hide the latency of ramp-up and avoid under-utilization (§3.2). *Third*, `cwnd` is updated after each feedback for quick stabilization where the update is at most one per packet to be resilient to noise. Together, these three ideas allow for a precise CC that operates on a per-packet basis minimizing incorrect CC decisions.

Prior works have separately proposed sub-RTT feedback [17, 50, 57], flow completion signaling [18], and per-packet cwnd adjustments [16, 27] which are discussed in §7. Bolt's main innovation is weaving these pieces into a harmonious and precise sub-RTT congestion control that is feasible for modern high-performance data centers. The key is to address congestion based on the *packet conservation principle* [25] visualized in Figure 7 where a network path is modeled as a pipe with a certain capacity of packets in-flight at a time. When the total `cwnd` is larger than the capacity by 1, there is an excess packet in the pipe which is queued. If the total `cwnd` is smaller than the capacity by 1, the bottleneck link will be under-utilized by 1 packet per RTT. Therefore, as soon as a packet worth queuing or under-utilization is observed, one of the senders should *immediately* decrement or increment the `cwnd`, without a long observation period.

Bolt's fundamental way of minimizing feedback delay and the observation period while generating precise feedback for per-packet decisions is materialized with 3 main mechanisms:

1. **SRC (Sub-RTT Control)** reduces congestion notification delay to its absolute minimum. (§3.1)
2. **PRU (Proactive Ramp Up)** hides any feedback delay for foreseen under-utilization events. (§3.2)
3. **SM (Supply Matching)** quickly recovers from unavoidable under-utilization events. (§3.3)

To realize these 3 mechanisms, Bolt uses 9 bytes of transport-layer header detailed in listing 1. We explain the purpose of each field as we describe the design of Bolt whose switching logic is summarized in Algorithm 1.

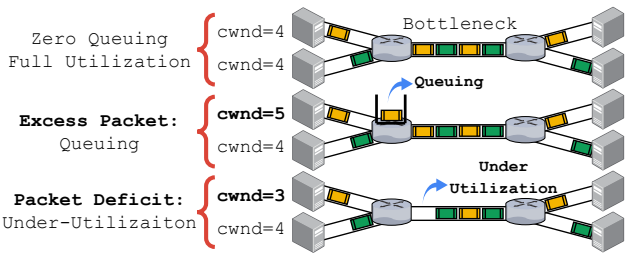


Figure 7: Pipe model of Packet Conservation Principle

Algorithm 1: BOLT LOGIC AT THE SWITCH

```
1  BoltIngress (pkt):
2      if !pkt.data then ForwardAndReturn(pkt)
3      CalculateSupplyToken(pkt)                    ▷ see Algorithm 3
4      if cur_q_size ≥ CC_THRESH then               ▷ Congested
5          if !pkt.dec then
6              pkt_src.queue_size ← switch.q_size
7              pkt_src.link_rate ← switch.link_rate
8              pkt_src.t_data_tx ← pkt.tx_time
9              SendSRC(pkt_src)
10         pkt.dec, pkt.inc ← 1,0
11     else if pkt.last then                        ▷ Near flow completion
12         if !pkt.first then pru_token++
13     else if pkt.inc then                         ▷ Pkt demands a token
14         if pru_token > 0 then
15             pru_token ← pru_token − 1
16         else if sm_token ≥ MTU then
17             sm_token ← sm_token − MTU
18         else
19             pkt.inc ← 0                          ▷ No token for cwnd inc.
20     ForwardAndReturn(pkt);
```

Listing 1: Bolt header structure

```
header bolt_h:
  bit<24> q_size;     // Occupancy at the switch
  bit<8>  link_rate;  // Rate of congested link
  bit<1>  data;       // Flags data packets
  bit<1>  ack;        // Flags acknowledgements
  bit<1>  src;        // Flags switch feedback
  bit<1>  last;       // Flags last wnd of flow
  bit<1>  first;      // Flags first wnd of flow
  bit<1>  inc;        // Signals cwnd increment
  bit<1>  dec;        // Signals cwnd decrement
  bit<1>  reserved;   // Reserved
  bit<32> t_data_tx; // TX timestamp for data pkt
```

3.1 SRC - Sub-RTT Control

As discussed in §2.1.1, a smaller feedback delay improves the performance of CC. Therefore, Bolt minimizes the delay of the feedback by generating control packets at the *ingress* pipeline of the switches and sending them directly *back to the sender*, a mechanism available in programmable switches such as Intel-Tofino2 [32]. While in spirit, this is similar to ICMP Source Quench messages [45] that have been deprecated due to feasibility issues in the Internet [33], Bolt's SRC mechanism exploits precise telemetry in a highly controlled data center environment.

Figure 8 depicts the difference in the paths traversed by the traditional ACK-based feedback versus the SRC-based feedback mechanism. As SRC packets are generated at ingress, they establish the absolute minimum feedback loop possible by traveling through the shortest path between a congested switch and the sender. Moreover, to further minimize the

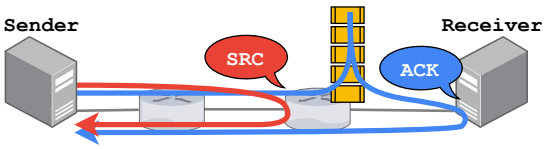


Figure 8: Path of ACK-based vs. SRC-based feedback

feedback delay, Bolt prioritizes ACK and SRC packets over data packets at the switches.

Bolt generates SRC packets for every data packet that arrives when the queue occupancy is greater than or equal to the CC_{THRESH} which is trivially set to a single MTU for minimal queuing. Yet, if there are multiple congested switches along the path of a flow, generating an SRC at each one of them for the same data would flood the network with an excessive amount of control packets. To prevent flooding switches mark the *DEC* flag of the original data packet upon generation of an SRC packet, such that no further SRC packets at other hops can be generated due to this packet (lines 5 and 10 in Algorithm 1). This implies that the number of SRC packets is bounded by the number of data packets in the network at any given time. In practice, however, we find that the actual load of SRC packets is extremely lower (§5.2.1) and present an approximation for the additional load of SRC packets in Appendix A.

When there are multiple congested hops, and the flow receives SRC packets only from the first one, the `cwnd` decrement still helps mitigate congestion at all of them. Consequently, even if congestion at the first hop is not as severe as the others, Bolt would drain the queue at the first hop and quickly start working towards the subsequent hops.

Bolt stamps two vital pieces of information on the SRC packets – the current queue occupancy and the capacity of the link. In addition, it reflects the TX timestamp of the original data packet (lines 6-8 in Algorithm 1). As the sender receives this packet, it runs the decision logic shown in Algorithm 2. First, rtt_{src} is calculated as the time between transmitting the corresponding data packet and receiving an SRC packet for it. This is the congestion notification delay for Bolt which is always shorter than RTT and enables sub-RTT control. The reflection of the TX timestamp enables this computation without any state at the sender. Next, $reaction\_factor$ is calculated as a measure of this flow's contribution to congestion. Multiplying this value with the reported queue occupancy gives the amount of queuing this flow should aim to drain. All the flows aiming to drain only what they are responsible for organically help for a fair allocation.

Finally, $\frac{rtt_{src}}{target_q}$ gives the shortest time interval between two consecutive `cwnd` decrements. This interval prevents overreaction because switches keep sending congestion notifications until the effect of the sender's `cwnd` change propagates to them. For example, if the target queue has a single packet, the sender decrements its `cwnd` only if rtt_{src} has elapsed since the last decrement. However, if the queue is larger, Bolt allows

Algorithm 2: BOLT LOGIC AT THE SENDER HOST

```
1  HandleSrc (pkt_src):
2      rtt_src ← now − pkt.t_tx_data
3      reaction_factor ← flow.rate / pkt_src.link_rate
4      target_q ←                                  ▷ in number of packets
5          pkt_src.queue_size × reaction_factor
6      if rtt_src / target_q ≤ now − last_dec_time then
7          cwnd ← cwnd − 1
8          last_dec_time ← now
9  HandleAck (pkt_ack):
10     if pkt_ack.inc then                         ▷ Capacity available
11         cwnd ← cwnd + 1
12     if pkt_ack.seq_no ≥ seq_no_at_last_ai then
13         cwnd ← cwnd + 1                         ▷ per-RTT add. inc.
14         seq_no_at_last_ai ← snd_next
```

more frequent decrements to equalize the total `cwnd` change to the target queue size in exactly one rtt_{src}. As the required `cwnd` adjustments are scattered over rtt_{src}, Bolt becomes more resilient to noise from any single congestion notification.

Events such as losses and timeouts do not happen in Bolt as it starts reacting to congestion way in advance. However, due to the possibility of such events occurring, say due to misconfiguration or packet corruption, handling retransmission timeouts, selective acknowledgments, and loss recovery are kept the same as in Swift [30] for completeness.

3.2 PRU - Proactive Ramp Up

Bolt explicitly tracks flow completions to facilitate Proactive Ramp Up (PRU). When a flow is nearing completion, it marks outgoing packets to notify switches, which plan ahead on distributing the bandwidth freed up by the flow to the remaining ones competing on the link. This helps remaining Bolt flows to *proactively ramp up* and eliminate the under-utilization period after a flow completion (see Figure 3b).

When flows larger than one BDP are sending their last `cwnd` worth of data, they set the *LAST* flag on packets to mark that they will not have packets in the next RTT. Note that this does not require knowing the application-level flow size. In a typical transport like TCP, the application injects a known amount of data to the connection at each `send` API call, denoted by the `len` argument [29]. Therefore, the amount of data waiting to be sent is calculable. *LAST* is marked only when the remaining amount of data in the connection is within `cwnd` size. Our detailed implementation is described in §4.2.

A switch receiving the *LAST* flag, if it is not congested, increments the *PRU token* value for the associated egress port. This value represents the amount of bandwidth that will be freed in the next RTT. The switch distributes these tokens to packets without the *LAST* flag, i.e. flows that have packets to send in the next RTT, so that senders can ramp up proactively.

However, only flows that are not bottlenecked at other hops should ramp up. To identify such flows, Bolt uses a greedy approach. When transmitting a packet, senders mark the *INC* flag on the packet. If a switch has PRU tokens (line 14 in Algorithm 1) or has free bandwidth (line 16 in Algorithm 1, explained in §3.3), it keeps the flag on the packet and consumes a token (line 15 and 17, respectively). Else, the switch resets the *INC* flag (line 19), preventing future switches on the path to consume a token for this packet. Then, if no switch resets the *INC* flag along the path, it is guaranteed that all the links on the flow's path have enough bandwidth to accommodate an extra packet. The receiver reflects this flag in the ACK so that the sender simply increments the `cwnd` upon receiving it (lines 10-11 in Algorithm 2). There are cases where the greedy approach can result in wasted tokens and we discuss the fallback mechanisms in §3.3.

Flows shorter than one BDP are not accounted for in PRU calculations. When a new flow starts, its first `cwnd` worth of packets are not expected by the network and contribute to the extra load. Therefore, the switch shouldn't replace these with packets from other flows once they leave the network. Bolt prevents this by setting the *FIRST* flag on packets that are in the first `cwnd` of the flow. Switches check against the *FIRST* flag on packets before they increment the *PRU token* value (line 12 of Algorithm 1).

Note that PRU doesn't need reduced feedback delay via SRC packets, because it accounts for a flow completion in the *next* RTT by design. A sender shouldn't start ramping up earlier as it can cause extra congestion before the flow completes. Therefore, the traditional RTT-based feedback loop is the right choice for correct PRU accounting.

3.3 SM - Supply Matching

Events like link and device failures or route changes can result in under-utilized links without proactive signaling. In addition, *PRU tokens* may be wasted if assigned to a flow that can not ramp up due to being already at line rate, or bottlenecked by downstream switches. For such events, conventional CC approaches rely on gradual *additive* increase to slowly probe for the available bandwidth which can take several tens of RTTs [1, 30, 42, 60]. Instead, Bolt is able to probe *multiplicatively* by explicitly matching utilization demand to supply through *Supply Matching* (SM) described below.

Bolt leverages stateful operations in programmable switches to measure the instantaneous utilization of a link. Each switch keeps track of the mismatch between the supply and demand for the link capacity for each port, where the number of bytes the switch can serialize in unit time is the supply amount for the link; and the number of bytes that arrive in the same time interval is the demand for the link. Naturally, the link is under-utilized when the supply is larger than the demand, otherwise, the link is congested. Note the similarity to HPCC [35] that also calculates link utilization, albeit from

Algorithm 3: Supply Token calculation at the ingress pipeline for each egress port of the switch

1 **CalculateSupplyToken** (pkt) **:**
2 $inter\_arrival\_time \leftarrow now - last\_sm\_time$
3 $last\_sm\_time \leftarrow now$
4 $supply \leftarrow BW \times inter\_arrival\_time$
5 $demand \leftarrow pkt.size$
6 $sm\_token \leftarrow sm\_token + supply - demand$
7 $sm\_token \leftarrow \min(sm\_token, MTU)$

an end-to-end point of view which restricts it to make once per RTT calculations. Bolt offloads this calculation to the switch data plane so that it can capture the precise instantaneous utilization instead of a coarse-grained measurement.

When a data packet arrives, the switch runs the logic in Algorithm 3 to calculate the *supply token* value (sm_token in the algorithms) associated with the egress port. The token accumulates the mismatch between the supply and demand in bytes on every packet arrival for a port. A negative value of the token indicates queuing whereas a positive value means under-utilization. When the token value exceeds one MTU, Bolt keeps the *INC* flag on the packet and permits the sender to inject an additional packet into the network (lines 16-17 in Algorithm 1). The *supply token* value is then decremented by an MTU to account for the inflicted future demand.

If a switch port doesn't receive a packet for a long time, the *supply token* value can get arbitrarily large, which prohibits capturing the instantaneous utilization if a burst of packets arrive after an idle period. To account for this, Bolt caps the *supply token* value at a maximum of one MTU. Details on how this feature is implemented in P4 are provided in §4.

As noted earlier, there are cases where there can be wasted tokens, i.e. a switch consumes a token (either PRU or SM) to keep *INC* bit, but is reset by downstream switches. In such cases, SM will find the available bandwidth in the next RTT. In the worst case, this happens for consecutive RTTs and Bolt falls back to additive increase similar to Swift [30] (lines 12-14 in Algorithm 2). Namely, `cwnd` is incremented once every RTT to allow flows to probe for more bandwidth and achieve fairness even if they do not receive any precise feedback as a fail-safe mechanism.

4 Implementation

We implemented Bolt through Host (transport layer and NIC) and Switch modifications in our lab. We used Snap [38] as our user-space transport layer and added Bolt in 1340 LOC in addition to the existing Swift implementation. Plus, the switch-side implementation consists of a P4 program – `bolt.p4` – in 1120 LOC. Figure 9 shows the overview of our lab prototype as a whole and we provide details below.

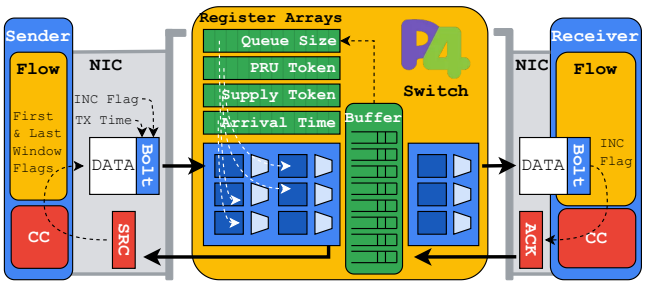


Figure 9: Bolt system overview

4.1 Switch Prototype

We based our implementation on the programmable data plane of Intel Tofino2 [11] switches in our lab as they can provide the queue occupancy of the egress ports in the ingress pipelines and generate SRC packets [32]. This is crucial for Bolt to minimize the feedback delay incurred by SRC packets as they are not subject to queuing delay at congested hops.

When congestion is detected in the ingress pipeline, the switch mirrors this packet to the input port while forwarding the original one along its path. The mirroring configuration is determined with a lookup table that matches the ingress port of the packet and selects the associated mirroring session.

The mirrored packet is then trimmed to remove the payload and the flow identifiers (i.e. source/destination addresses and ports) are swapped. Finally, *SRC* flag is set on this packet to complete its conversion into an SRC packet.

The entire `bolt.p4` consists mainly of register array declarations and simple if-else logic as shown in Algorithm 1. There are 4 register arrays for storing queue occupancy, token values, and the last packet arrival time. All of the register arrays are as large as the number of queues on the switch because the state is maintained per queue. In total, only 3.6% and 0.6% of available SRAM and TCAM, respectively, are used for the register arrays, tables, and counters.

The switch keeps the last packet arrival time for every egress port to calculate the supply for the link. On each data packet arrival, the difference between the current timestamp and the last packet arrival time is calculated as the inter-arrival time. This value should ideally be multiplied with the link capacity (line 4 of Algorithm 3) to find the supply amount. However, since floating point arithmetic is not available in PISA pipelines, we use a lookup table indexed on inter-arrival times to determine the supply amount. We set the size of this lookup table as 65536 where each entry is for a different inter-arrival time with a granularity of a nanosecond. Consequently, if the inter-arrival time is larger than 65 microseconds, the *supply token* value is directly set to its maximum value of 1 MTU which triggers *INC* flag to be set. We find that, at a reasonably high load, 65 microseconds of inter-arrival time is rare enough for links greater than 100Gbps such that any longer value can be safely interpreted as under-utilization.

Our prototype is based on a single HW pipeline. Therefore,

we implemented Bolt entirely at the ingress pipeline to make it easier to understand and debug its logic. However, since PRU and SM maintain state per egress port, they could also be implemented at the egress pipeline with minor modifications. This way, the state for packets from multiple ingress pipelines would naturally be aggregated.

4.2 Host Prototype

Our transport layer uses the NIC hardware timestamps to calculate rtt_{src} as described in Algorithm 2. When a sender is emitting data, the TX timestamp is stamped onto the packet. The switch reflects this value back to the sender, so that rtt_{src} is the difference between the NIC time when the SRC packet is received (RX timestamp) and the reflected TX timestamp. This precisely measures the network delay to the bottleneck without any non-deterministic software processing delays.

The transport layer also multiplexes RPCs meant for the same server onto the same network connection. Then, the first `cwnd` bytes of a new RPC isn't necessarily detected as the *first* window of the connection. To mitigate this issue, our prototype keeps track of idle periods of connections and resets the *bytes-sent counter* when a new RPC is sent after such a period. Therefore the *FIRST* flag is set on a packet when the counter value is smaller than `cwnd`.

Finally, the *last* window marking for PRU requires determining the size of the remaining data for each connection. In our prototype, the connection increments *pending bytes counter* by the size of data in each `send` API call from the application. Every time the connection transmits a packet into the network, the counter value is decremented by the size of the packet. Therefore the *LAST* flag is set on a packet when this counter value is smaller than `cwnd`.

4.3 Security and Authentication

Getting Bolt to work for encrypted and authenticated connections was a key challenge in our lab. Our prototype uses a custom version of IPsec ESP [23, 28] for encryption atop the IP Layer. However, switches need to read and modify CC information at the transport header without breaking end-to-end security. The *crypt_offset* of the protocol allows packets to be encrypted only beyond this offset. We set it such that the transport header is not encrypted, but is still authenticated.

In addition, switches cannot generate encrypted packets due to the lack of encryption and decryption capabilities. To remedy this, we generate SRC packets on switches as unreliable datagrams per RoCEv2 standard by adding IB BTH and DETH headers while removing the encryption header.

The RoCEv2 packets have the invariant CRC calculated over the packet and appended as a trailer. Fortunately, Tofino2 provides a CRC extern that is capable of this calculation over small, constant-size packets [31]. As a result, NICs are able to forward the SRC packets correctly to the upper layers based on the queue pair numbers (QPN) on the datagrams.

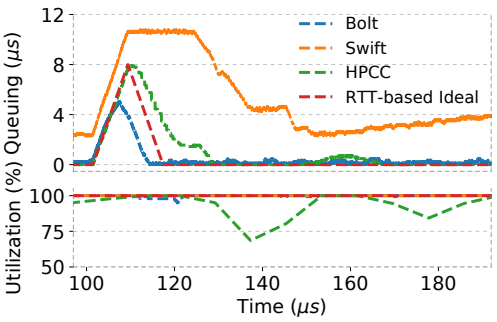


Figure 10: Bolt's reaction to flow arrival versus the ideal behavior.

5 Evaluation

We evaluate Bolt on NS3 [48] micro-benchmarks to demonstrate its fundamental capabilities in §5.1 followed by sensitivity and fairness analysis in §5.2 and §5.3. Then, in §5.4, we run large-scale experiments to measure the end-to-end performance of the algorithm, i.e. flow completion time slowdowns. Finally, we evaluate our lab prototype in §5.5.

5.1 Micro-Benchmarks

5.1.1 Significance of SRC

The only way for Bolt to decrease `cwnd` is through SRC whose effectiveness is best observed during congestion. Therefore, we repeat the same flow arrival scenario described in Figure 6 with Bolt.[3] Typically, with conventional RTT-based congestion control algorithms, a new flow starting at line rate emits BDP worth of packets until it receives the first congestion feedback after an RTT. If the network is already fully utilized before this flow, all emitted packets end up creating a BDP worth of queuing even for an RTT-based ideal scheme. Then, the ideal scheme would stop sending any new packets to allow draining the queue quickly which would take another RTT. This behavior is depicted as red in Figure 10 where a new flow joins at 100μs.

HPCC's behavior in Figure 10 is close to the ideal given that it is an RTT-based scheme with high precision congestion signal. As the new flow arrives, the queue occupancy rises to 1 BDP. However, the queue is drained at a rate slower than the link capacity because flows continue to occasionally send new packets while the queue is not completely drained.

Bolt, on the other hand, detects congestion earlier than an RTT. Therefore it starts decrementing `cwnd` before the queue occupancy reaches BDP and completely drains it in less than 2 RTTs, even shorter than the RTT-based ideal scheme.

In addition, HPCC's link utilization drops to as low as 75% after draining the queue and oscillates for some time, which is due to the RTT-long observation period (§2.2). Bolt's per-packet decision avoids this under-utilization.

[3] The dumbbell topology with two flows (8 μs RTT at 100Gbps).

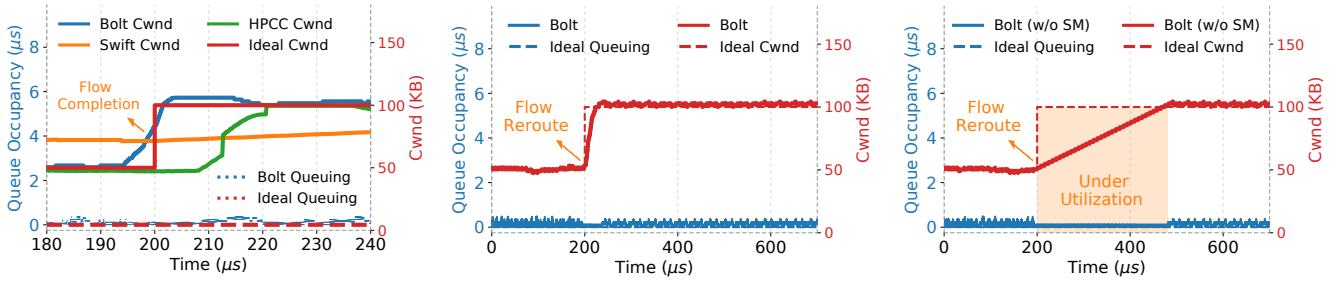


Figure 11: `cwnd` of the remaining flow and queue occupancy after a flow completion.

(a) Bolt **(b)** Bolt (without SM)

Figure 12: `cwnd` of the remaining flow and queue occupancy after a flow is rerouted.

Utilization (%)		PRU OFF	PRU ON
SM	**OFF**	90.46	97.38
	ON	92.41	98.54

Table 1: Effectiveness of PRU and SM on the bottleneck utilization.

5.1.2 Significance of PRU

Flow completions cause under-utilization without proactive ramp-up or standing queues because conventional congestion control algorithms take at least an RTT to react to them (§2.1.2). Moreover, as shown in Figure 4 for Swift, a standing queue might not be enough to keep the link busy if the `cwnd` of the completing flow is larger than the queue size

We repeat the same scenario with Bolt to test how effective proactive ramp-up can be upon flow completions against Swift and HPCC. Figure 11 shows the `cwnd` of the remaining flow and the queue occupancy at the bottleneck link. When a Bolt flow completes at t=200μs, the remaining one is able to capture the available bandwidth in 1μs because it starts increasing `cwnd` (by collecting PRU tokens) one RTT earlier than the flow completion. Moreover, neither queuing nor under-utilization is observed. HPCC, on the other hand, takes 20μs ($>$ 2$\times$RTT) to ramp up for full utilization because it needs one RTT to detect under-utilization and another RTT of observation period before ramping up. Finally, Swift takes more than 370μs to reach the stable value due to the slow additive increase approach which doesn't fit into Figure 11. The complete ramp-up of Swift is shown in Figure 4.

Although PRU and SM seem to overlap in the way they quickly capture available bandwidth, PRU is a faster mechanism compared to SM because it detects under-utilization proactively. To demonstrate that, we create a star topology with 100Gbps links and a base RTT of 5μs, where 5 senders send 500KB to the same receiver. Flows start 15μs apart from each other to complete at different times so that PRU and SM can kick in. We repeat while disabling PRU or SM and measure the bottleneck utilization to observe how each mechanism is effective at achieving high throughput.

Table 1 shows the link utilization between the first flow completion and the last one. When only PRU is disabled, the utilization drops by 6% despite having SM. On the other hand, disabling SM alone causes only a 1% decrease. This indicates that PRU is a more powerful mechanism compared to SM when under-utilization is mainly due to flow completions in the network. Together, they increase utilization by 8%.

5.1.3 Significance of SM

Unlike flow completions, events such as link failure or rerouting are not hinted in advance. Then, PRU doesn't kick in, making Bolt completely reliant on SM for high utilization. To demonstrate how SM quickly captures available bandwidth, we use the same setup from Figures 4 and 11, but reroute the second flow instead of letting it complete.

Figure 12 shows the `cwnd` of the remaining flow after the other one leaves the bottleneck. Thanks to SM, `cwnd` quickly ramps up to utilize the link in 23μs (12a). When SM is disabled, the only way for Bolt to ramp up is through traditional additive increase which increases `cwnd` by 1 every RTT (12b). Therefore it takes more than 33 RTTs to fully utilize the link.

5.2 Sensitivity Analysis

5.2.1 Overhead of SRC

To mitigate congestion, Bolt generates SRC packets in an already loaded network. In order to understand the extra load created by SRC, we measure the bandwidth occupied by SRC packets at different burstiness levels. For this purpose, we use the same star topology from §5.1.2. The number of senders changes between 1 and 63 to emulate different levels of burstiness towards a single receiver at 80% load. The traffic is based on the READ RPC workload from Figure 1.

Figure 13 shows the bandwidth occupied by the SRC packets (top) and the 99^{th}-p queue occupancy at the bottleneck (bottom) with a different number of senders. When there are multiple senders, the SRC bandwidth is stable at 0.33Gbps (0.33% of the capacity). Similarly, the tail queuing is also bounded below 6.4μs for all the experiments. Therefore, we conclude that Bolt is able to bound congestion with a negligible amount of extra load in the network. In §5.5, we show that the overhead is negligible for the lab prototype as well.

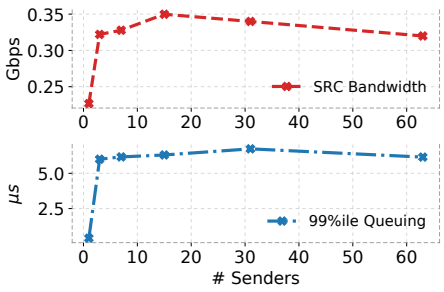


Figure 13: SRC overhead and sensitivity for different levels of burstiness

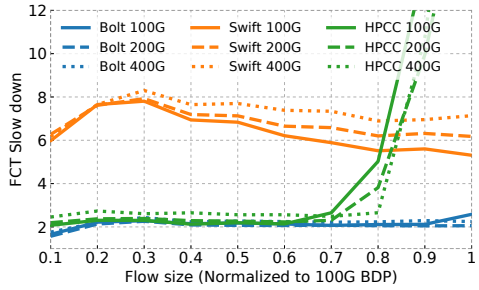


Figure 14: 99^{th}-p Slowdown for messages smaller than BDP

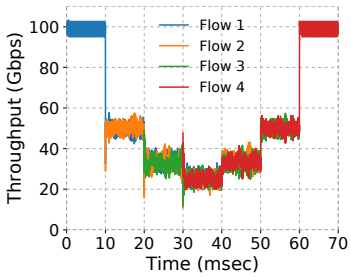


Figure 15: Fair allocation by Bolt

Metric	Swift	HPCC	Bolt
99^{th}-p Queuing (msec)	23.543	23.066	13.720
99^{th}-p FCT Slowdown	7017	5037	5000

Table 2: Tail queuing, and FCT slowdown for 5000-to-1 incast.

5.2.2 Robustness Against Higher Line Rates

One of the goals of Bolt is to be robust against ever-increasing line rates in data centers. To evaluate the performance at different line rates, we repeat the simulations from §5.2.1 with 63 senders where we increase the link capacity from 100Gbps to 200Gbps and 400Gbps. This way, the burstiness of the senders increases, making it difficult to maintain small queuing at the switches. Therefore, flow completion time (FCT) slowdown [14][4] of small flows are affected the most, whereas throughput oriented large flows would trivially be better off with higher line rates.

Accordingly, we plot the 99^{th}-p FCT slowdown for flows that are smaller than BDP (at 100 Gbps) in Figure 14. Swift's performance monotonically decays with higher link rates due to the increasing burstiness. Similarly, HPCC at 400Gbps achieves 25% worse performance compared to the 100Gbps scenario for flow sizes up to 0.7 BDP. For the rest of the workload, HPCC makes a leap such that it performs worse than other algorithms irrespective of the line rates. Bolt on the other hand is able to maintain small and steady tail slowdowns for all the small flows despite the increasing line rates.

5.3 Fairness Analysis

To test the fairness of Bolt, we run an experiment on a dumbbell topology with 100Gbps links. We add or remove a new flow every 10 milliseconds and measure the throughput of each flow which is shown in Figure 15. Our results indicate that Bolt flows converge to the new fair share quickly when the state of the network changes.

[4]FCT slowdown is flow's actual FCT normalized by its ideal FCT when the flow sends at line-rate (e.g., when it was the only flow in the network).

5.4 Large Scale Simulations

One of the most challenging cases for CC is a large-scale incast. To evaluate Bolt's performance in such a scenario, we set up a 5000-to-1 incast on the star topology described earlier where each one of 50 senders starts 100 same size flows at the same time. Table 2 presents the 99^{th}-p queue occupancy and FCT slowdown for the incast. Since Bolt detects congestion as early as possible, it bounds tail queuing to a 41% lower level compared to Swift and HPCC. In addition, the tail FCT slowdown for Bolt is 5000, indicating full link utilization. Moreover, the bandwidth occupied by the SRC packets is as low as 0.77Gbps throughout the incast. This is only twice the overhead for 80% load in §5.2.1, despite the extreme bursty arrival pattern of the incast.

We also evaluate the performance of Bolt on a cluster-scale network where 64 servers are connected with 100Gbps links to a fully subscribed fat-tree topology with 8 ToR switches. All the other links are 400Gbps and the maximum unloaded RTT is 5μs. We run traffic between servers based on two workloads at 80% load: (i) the READ RPC workload described in Figure 1 represents traffic from our data center, (ii) the Facebook Hadoop workload [49]. Figure 16 and 17 show the median and 99^{th}-p FCT slowdown for the workloads. Note that the Hadoop workload is relatively more bursty where 82% of the flows/RPCs fit within a BDP in the given topology. Hence a large fraction of the curves in Figure 17 is flat where all the RPCs in this region are extremely small (i.e. single packet).

For both of the workloads, Bolt performs well across all flow sizes. Specifically, Bolt and HPCC achieve very low FCT for short flows (<7KB) because of a few design choices: First, they maintain zero standing queues. Plus, Bolt's SRC reduces the height of queue spikes after flow arrivals. HPCC, on the other hand, tends to under-utilize the network upon flow completions (§2.1.2), statistically reducing queue sizes.

FCT of median-size flows (a few BDPs) starts to degrade for HPCC due to under-utilization described in §2.1.2 and §2.2. Bolt performs up to 3× better in this regime by avoiding under-utilization thanks to PRU and SM. Swift's standing queues prevent under-utilization, but FCTs are high because

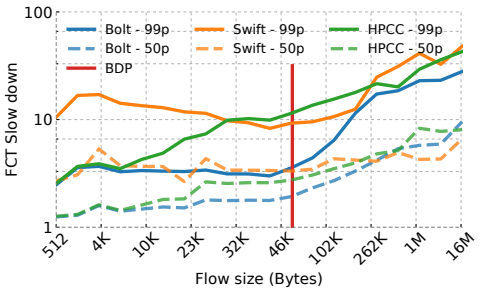


Figure 16: FCT slowdown for READ RPC Workload from Figure 1

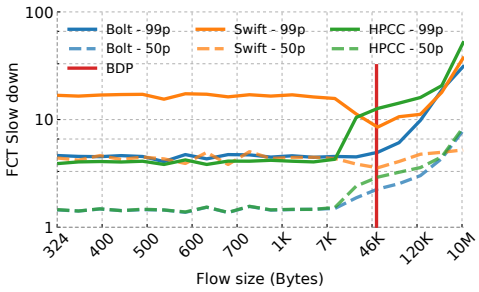


Figure 17: FCT slowdown for Facebook Hadoop Workload

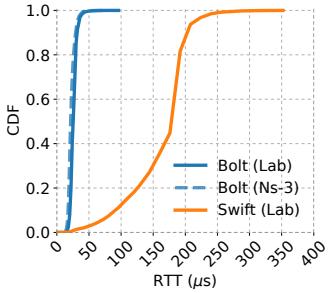


Figure 18: Bolt's lab prototype matches its simulator

median-size flows are also affected by the queuing delay.

The impact of queuing diminishes and utilization becomes the dominant factor for long flows. Therefore Bolt and Swift perform better than HPCC. In addition, Bolt is slightly better at the tail compared to Swift, while Swift is slightly better at the median, suggesting that Bolt is fairer.

5.5 Bolt in the Lab

Our lab testbed consists of 2 servers and 2 Intel Tofino2 [11] switches. Each server runs 4 packet processing engines running Snap [38] that provide the transport layer with the Bolt algorithm. Each engine is scheduled on a CPU core that independently processes packets so that we are able to create a large number of connections between the servers. Links from the servers to the switches are 100Gbps and the switches are connected to each other with a 25Gbps link to guarantee that congestion takes place within the network. The base-RTT in this network is 14μs and we generate flows between the servers based on the READ RPC workload.

We evaluate Bolt on two scenarios. First, we run 100% (of 25Gbps) load to see if our prototype can saturate the bottleneck. Then, we run 80% load to compare the congestion mitigation performance of Bolt against Swift in a more realistic scenario. Finally, we verify that our results from the lab and the simulations match to verify our implementations.

The median and the 99^{th}-p RTT at 100% load for Swift are 189μs and 208μs respectively. These numbers are high because Swift maintains a standing queue based on the configured base delay to fully utilize the link even after flow completions. Bolt on the other hand, attains 27μs and 40μs of median and tail RTT, 86% and 81% shorter than Swift. In the meantime, it achieves 24.7Gbps which is only 0.8% lower compared to Swift despite the lack of a standing queue.

We repeat the same experiment with 80% load and observe that both Swift and Bolt can sustain 80% (20Gbps) average link utilization. Figure 18 shows the CDF of measured RTTs throughout the experiment. Similar to the 100% load case, the median and tail RTTs for Bolt are 25μs and 40μs, 86% and 83% lower compared to Swift respectively[5].

Moreover, we measure that the bandwidth occupied by the SRC packets in our lab is 0.13Gbps, 0.536% of the bottleneck capacity. This is consistent with our observation in §5.2 despite the larger SRC packets with custom encapsulations.

Finally, we simulate the 80% load experiment in NS3 [48] with the same settings to verify that our simulator matches our observations in the lab. Figure 18 also shows the CDF of RTTs measured throughout the simulation. The median and tail RTTs from our simulations are 21μs and 39μs, within 15% and 0.025% of the lab results respectively.

6 Practical Considerations

Typically, new products are deployed incrementally in data centers due to availability, security, or financial concerns. As a consequence, the new product (i.e. the CC algorithm) lives together with the old one for some time called brownfield deployment. We identify three potential issues that Bolt could face during this phase and address them below.

First, some switches in the network may not be capable of generating SRC packets while new programmable switches are being deployed. Unfortunately, the vanilla Bolt design can not control the congestion at these switches. This can be addressed by running an end-to-end algorithm on top of Bolt. For example, imagine the Swift algorithm calculates a fabric `cwnd` as usual in parallel with Bolt's calculation of `cwnd` using SRC packets. Then, the minimum of the two is selected as the effective `cwnd` for the flow. When an older generation switch is congested, SRC packets are not generated, but Swift adjusts the `cwnd`. Consequently, flows benefit from ultra-low queuing at the compatible switches while falling back to Swift when a non-programmable switch becomes the bottleneck.

Second, hosts would also be migrated to Bolt incrementally. Therefore, Bolt would need to coexist with the prior algorithm. Studying the friendliness of algorithms with Bolt through frameworks such as [26] and [56] remains a future work. For example, TCP CUBIC would not coexist well with Bolt as it tries to fill the queues until a packet is dropped while Bolt would continuously decrement its `cwnd` due to

[5]For Swift we set 50μs base target delay as specified in the paper [30] and 200μs as flow scaling range. Swift's average RTT in Figure 18 is higher than Swift paper's value (~50μs), because of two reasons. First, this workload is burstier than the ones in Swift paper. Second, the 25Gbps bottleneck implies a higher level of flow scaling than with 100Gbps links.

queuing. Instead, we propose the use of QoS (Quality of Service) queues to isolate Bolt traffic from the rest. Appendix B describes a baseline approach for such deployment.

Finally, scenarios where packet transmissions are batched (say by the NIC) even when the `cwnd` is smaller than BDP can still trigger SRC generation, inhibiting flows to increase `cwnd` to the right value. We find that transport offloading on modern smart NICs uses batching to sustain high line rates. Bolt alleviates such bursts with a higher CC_{THRESH} that tolerates batch size worth of queuing at the switches.

7 Related Work

In addition to HPCC [35] and Swift [30] that serve as our primary comparison points, several other schemes have similar ideas or goals.

FastTune [59] uses programmable switches for precise congestion signal. Similar to HPCC, it calculates link utilization over an RTT to multiplicatively increase or decrease `cwnd`. For shorter feedback delay, it pads the INT header onto ACK packets in the reverse direction instead of data packets. ExpressPass [10] utilizes the control packets in the reverse direction as well. Nonetheless, forward and reverse paths for a flow are not always symmetrical due to ECMP-like load balancing or flow-reroutes. Therefore, Bolt chooses to explicitly generate SRC packets with little overhead (§5.2).

FastLane [57] is one of the early proposals to send notifications from switches directly to the senders. However, notifications are generated only for buffer overflows which is late for low latency CC in data centers. Annulus [50], on the other hand, uses standard QCN [21] packets from switches with queue occupancy information. Yet these packets are not L3 routable, so Annulus limits its scope only to detecting bottlenecks one hop away from senders. Bolt brings the best of both worlds and controls congestion at every hop while knowing the precise state of congestion.

XCP [27] and RCP [13] also propose congestion feedback generated by the switch. Switches wait for an average RTT before calculating CC responses that are piggybacked on the data packet and reflected on the ACK. As discussed in §2.2, this implies a control loop delay of two RTTs in total.

FCP [18] uses budgets and prices to balance the load and the traffic demand. FCP switches calculate the price of the link based on the demand while senders signal flow arrivals or completions similar to SM and PRU in Bolt. However, the required time series averaging and floating-point arithmetic make the calculation infeasible for programmable switches while consuming bytes on the header. In contrast, Bolt is based on the packet conservation principle with a simple, yet precise logic implementable in P4 and requires only 3 bits on the header (*FIRST*, *LAST*, and *INC*) for SM and PRU.

Switch feedback has also been studied for wireless settings. For instance, ABC [16] marks packets for `cwnd` increments or decrements with an RTT-based control loop for congestion mitigation. On the other hand, Zhuge [39] modifies the wireless AP to help senders detect congestion quicker. However, since it is challenging to modify schemes in WAN, Zhuge relies on the capabilities of existing schemes for the precision of the congestion signals, i.e. delayed ACKs for TCP.

Receiver-driven approaches such as NDP [19], pHost [15], and Homa [43] require receivers to allocate/schedule credits based on the demand from senders. They work well for congestion at the last hop because receivers have good visibility into this link. For example, when an RPC is fully granted, the Homa receiver starts sending grants for the next one without the current RPC being finished to proactively utilize the link. This is similar to PRU in Bolt despite being limited to the last hop. Unfortunately, the last hop is not always the bottleneck for a flow especially when the fabric is over-subscribed [52].

Schemes that use priority queues [2, 4, 20, 43] are proposed to improve the scheduling performance of the network to approximate SRPT [51] like behavior. We find deploying such schemes to be rather difficult because, typically, QoS queues in data centers are reserved to separate different services.

On-Ramp [37] is an extension for CC which proposes to pause flows at the senders when the one-way delay is high. Bolt can also benefit from its flow control mechanism. We leave evaluating Bolt with this extension as future work.

There are also per-hop flow control mechanisms such as BFC [17] and PFFC [55] that pause queues at the upstream switches via early notifications from the bottleneck. The deadlock-like issues of PFC [54] are resolved by keeping the per-flow state on switches, which we find challenging in our data centers as switches have to implement other memory or queue-intensive protocols, e.g., routing tables or QoS. Therefore, we scope Bolt to be an end-to-end algorithm with a fixed state similar to other algorithms in production [30, 35, 60].

8 Conclusion

Increasing line rates in data centers is inevitable due to the stringent SLOs of applications. Yet, higher line rates increase burstiness, putting more pressure on CC to minimize queuing delays for short flows along with high link utilization for long flows. We find that two key aspects of CC need to be pushed to their boundaries to work well in such highly dynamic regimes based on experience with our data centers.

Bolt addresses these aspects thanks to the flexibility and precision provided by programmable switches. First, it uses the most granular congestion signal, i.e. precise queue occupancy, for a per-packet decision logic. Second, it minimizes the control loop delay to its absolute minimum by generating feedback at the congested switches and sending them directly back to the senders. Third, it hides the control loop delay by making proactive decisions about foreseeable flow completions. As a result, accurate `cwnd` is calculated as quickly as possible, achieving more than 80% reduction in tail latency and 3× improvement in tail FCT.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.*, 40(4):63–74, August 2010. doi:10.1145/1851275.1851192.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. PFabric: Minimal near-Optimal Datacenter Transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, August 2013. doi:10.1145/2534169.2486031.

[3] Serhat Arslan and Nick McKeown. Switches Know the Exact Amount of Congestion. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3375235.3375245.

[4] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, 2015.

[5] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Commun. ACM*, 60(4):48–54, March 2017. doi:10.1145/3015146.

[6] Luiz Andrè Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, San Rafael, CA, USA, 3rd edition, October 2018. URL: https://doi-org.stanford.idm.oclc.org/10.2200/S00874ED3V01Y201809CAC046.

[7] Tanya Bhatia. UADP - The Powerhouse of Catalyst 9000 Family. Cisco Systems Inc., December 2018. URL: https://community.cisco.com/t5/networking-blogs/uadp-the-powerhouse-of-catalyst-9000-family/ba-p/3764605.

[8] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. RFC 5681, September 2009. URL: https://rfc-editor.org/rfc/rfc5681.txt, doi:10.17487/RFC5681.

[9] Mihai Budiu and Chris Dodd. The P4-16 Programming Language. *SIGOPS Oper. Syst. Rev.*, 51(1):5–14, September 2017. URL: https://doi-org.stanford.idm.oclc.org/10.1145/3139645.3139648, doi:10.1145/3139645.3139648.

[10] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 239–252, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3098822.3098840.

[11] Intel Corporation. Tofino 2: Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise, May 2021. URL: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[12] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013. URL: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[13] Nandita Dukkipati. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3292347. URL: https://dl-acm-org.stanford.idm.oclc.org/doi/10.5555/1368746.

[14] Nandita Dukkipati and Nick McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, January 2006. doi:10.1145/1111322.1111336.

[15] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2716281.2836086.

[16] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A Simple Explicit Congestion Controller for Wireless Networks . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/goyal.

[17] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, Renton, WA, April 2022. USENIX Associa-

tion. URL: https://www.usenix.org/conference/nsdi22/presentation/goyal.

[18] Dongsu Han, Robert Grandl, Aditya Akella, and Srinivasan Seshan. FCP: A Flexible Transport Framework for Accommodating Diversity. *SIGCOMM Comput. Commun. Rev.*, 43(4):135–146, August 2013. doi:10.1145/2534169.2486004.

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3098822.3098825.

[20] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi 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*, SIGCOMM '20, page 422–434, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405878.

[21] IEEE Standard for Local and Metropolitan Area Networks– Virtual Bridged Local Area Networks Amendment 13: Congestion Notification. *IEEE Std 802.1Qau-2010 (Amendment to IEEE Std 802.1Q-2005)*, pages 1–135, 2010. doi:10.1109/IEEESTD.2010.5454063.

[22] Broadcom Inc. High-Capacity StrataXGS Trident4 Ethernet Switch Series, May 2021. URL: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[23] Google Inc. PSP, March 2022. URL: https://github.com/google/psp.

[24] Versa Technology Inc. 400G Ethernet: It's Here, and It's Huge, December 2021. URL: www.versatek.com/400g-ethernet-its-here-and-its-huge/.

[25] Van Jacobson. Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 314–329, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/52324.52356.

[26] Raj Jain, Dah-Ming Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR*, cs.NI/9809099, January 1998. URL: https://arxiv.org/abs/cs/9809099.

[27] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 89–102, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/633025.633035.

[28] Stephen Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005. URL: https://www.rfc-editor.org/info/rfc4303, doi:10.17487/RFC4303.

[29] Michael Kerrisk. send(2) — Linux manual page, March 2021. URL: https://man7.org/linux/man-pages/man2/send.2.html.

[30] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3406591.

[31] Shiv Kumar, Pravein Govindan Kannan, Ran Ben Basat, Rachel Everman, Amedeo Sapio, Tom Barbette, and Joeri de Ruiter. Open Tofino, July 2021. URL: https://github.com/barefootnetworks/Open-Tofino.

[32] Jeongkeun Lee, Jeremias Blendin, Yanfang Le, Grzegorz Jereczek, Ashutosh Agrawal, and Rong Pan. Source Priority Flow Control (SPFC) towards Source Flow Control (SFC), November 2021. URL: https://datatracker.ietf.org/meeting/112/materials/slides-112-iccrg-source-priority-flow-control-in-data-centers-00.

[33] Konstantin Lepikhov. Source Quench. Atlassian Corporation Pty Ltd., April 2018. URL: https://wiki.geant.org/display/public/EK/Source+Quench.

[34] Yuliang Li. *Hardware-Software Codesign for High-Performance Cloud Networks*. PhD thesis, Harvard University, 2020. URL: https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37368976.

[35] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341302.3342085.

[36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control, December 2021 [Online]. URL: https://hpcc-group.github.io/results.html.

[37] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 47–63, Berkeley, CA, USA, April 2021. USENIX Association. URL: https://www.usenix.org/conference/nsdi21/presentation/liu.

[38] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359657.

[39] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving Consistent Low Latency for Wireless Real-Time Communications with the Shortest Control Loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 193–206, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544216.3544225.

[40] Rui Miao, Li Bo, Hongqiang Harry Liu, and Ming Zhang. Buffer sizing with HPCC. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, pages 1–2, New York, NY, USA, 2019. Association for Computing Machinery. URL: http://buffer-workshop.stanford.edu/papers/paper5.pdf.

[41] Leonid Mirkin and Zalman J. Palmor. Control Issues in Systems with Loop Delays. In Dimitrios Hristu-Varsakelis and William S. Levine, editors, *Handbook of Networked and Embedded Control Systems*, pages 627–648. Birkhäuser Boston, Boston, MA, 2005. doi:10.1007/0-8176-4404-0_27.

[42] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787510.

[43] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John 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*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230543.3230564.

[44] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting TCP for Reconfigurable Datacenter Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/mukerjee.

[45] John Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984. URL: https://www.rfc-editor.org/info/rfc896, doi:10.17487/RFC0896.

[46] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993. doi:10.1109/90.234856.

[47] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, February 2018. URL: https://rfc-editor.org/rfc/rfc8312.txt, doi:10.17487/RFC8312.

[48] George F. Riley and Thomas R. Henderson. The ns-3 Network Simulator. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-12331-3_2.

[49] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 123–137, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787472.

[50] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates. 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*, SIGCOMM '20, page 735–749, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405899.

[51] Linus E. Schrage and Louis W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research*, 14(4):670–684, 1966. doi:10.1287/opre.14.4.670.

[52] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787508.

[53] Bruce Spang, Serhat Arslan, and Nick McKeown. Updating the theory of buffer sizing. *Performance Evaluation*, 151:102232, 2021. doi:https://doi.org/10.1016/j.peva.2021.102232.

[54] IEEE Standard. Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011*, (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011):1–40, 2011. doi:10.1109/IEEESTD.2011.6032693.

[55] Shie-Yuan Wang, Yo-Ru Chen, Hsien-Chueh Hsieh, Ruei-Syun Lai, and Yi-Bing Lin. A Flow Control Scheme Based on Per Hop and Per Flow in Commodity Switches for Lossless Networks. *IEEE Access*, 9:156013–156029, 2021. doi:10.1109/ACCESS.2021.3129595.

[56] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 17–24, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3365609.3365855.

[57] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. FastLane: Making Short Flows Shorter with Agile Drop Notification. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 84–96, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2806777.2806852.

[58] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 78–85, New York, NY, USA, 2017. Association for Computing Machinery. URL: https://doi-org.stanford.idm.oclc.org/10.1145/3131365.3131375, doi:10.1145/3131365.3131375.

[59] Renjie Zhou, Dezun Dong, Shan Huang, and Yang Bai. FastTune: Timely and Precise Congestion Control in Data Center Network. In *2021 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 238–245, New York City, NY, USA, 2021. IEEE. doi:10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00043.

[60] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787484.

Appendix

A Approximating SRC Overhead

Bolt switches generate SRC packets for every data packet they receive as long as there is queuing, given that the data packet is not marked with the *DEC* flag. Then the number of

SRC packets depend on how long queuing persisted and how many packet are received in this time interval.

At steady state where no new RPCs join the network, we can estimate the fraction of time queuing persist on a bottleneck. Congestion at this regime happens only due to the once-per-RTT additive increase of 1 by each flow.

As described in §3.1, senders pace `cwnd` decrements such that the total number of decrements equals the queue occupancy after 1 rtt_{src}. This implies that any queuing will persist for 1 rtt_{src}, but will be completely drained after. Since a new congestion is not inflicted until the next RTT, we conjecture that the fraction of time that the switch has non-zero queuing is governed by the following golden ratio:

$$\text{fraction of time switch is congested} = \frac{rtt_{src}}{rtt} \tag{1}$$

which is always less then 1.

Note that equation 1 is an approximation for congestion interval since it doesn't incorporate traffic load, new RPC arrivals or multi bottleneck scenarios. Nonetheless, we can calculate the number of SRC packets generated at a bottleneck with it.

$$\text{\# of SRC pkts} = \text{\# of DATA pkts} \times \frac{rtt_{src}}{rtt} \tag{2}$$

Finally, we map equation 2 to the bandwidth occupied by the SRC packets by incorporating the link capacity and the packet sizes:

$$\text{SRC Bandwidth} = C \times \frac{p_{src}}{p_{data}} \times \frac{rtt_{src}}{rtt} \tag{3}$$

Where C is the rate at which the traffic is flowing through the bottleneck link, p_{src} is the size of SRC packets and p_{data} is the size of data packets, i.e. MTU.

When we calculate the bandwidth of SRC packets according to equation 3 for the simulation in §5.2.1, we find 0.37Gbps which is within 12% of the simulation result of 0.33Gbps. Moreover, equation 3 gives 0.10Gbps for our lab setup in §5.5 which is within 23% of the measured value of 0.13Gbps.

B Bolt with QoS

The relationship between congestion control algorithms and QoS has always been contradictory. An ideal congestion control algorithm aims to mitigate any queuing at the switch, whereas a QoS mechanism always needs enough queuing to be able to differentiate packet priorities and serve one before the other. Put another way, QoS only takes effect when the arrival rate at a link is greater than the capacity such that it causes queue build-up. Yet, QoS is vital for commercial networks in order to be able to differentiate applications or tenants for business related reasons [6]. This is particularly true for unavoidable transient congestion events, i.e. incast.

Algorithm 4: Supply Token calculated for QoS queue i at the switch with n QoS levels serving the same egress port

```
Function CalculateSupplyToken(pkt):
    inter_arrival_time ← now − last_sm_time
    last_sm_time ← now
    w_effective ← 0
    for j ← 0 to n do
        if i = j || q_size_j ≠ 0 then
            w_effective ← w_effective + w_j
    supply ← BW × inter_arrival_time × (w_i / w_effective)
    demand ← pkt.size
    sm_token ← sm_token + supply − demand
    sm_token ← min(sm_token, MTU)
```

Fortunately, the way Bolt reports queue occupancy is QoS-agnostic such that it can generate SRC packets with the occupancy of the queue assigned by the QoS mechanism. Consequently, it would try to minimize queuing at that particular queue. Similarly, the way *PRU token* are calculated would be queue specific instead of being egress port specific. For example, if there are P ports on a switch and n QoS levels per port, the size of the register array that maintains the token values would be of $P \times n$ and flows would only be able to proactively ramp-up if another flow with the same QoS level is about to finish.

On the other hand, accounting for the *supply token* requires the service rate for the associated queue (§3.3) which would be a dynamic value depending on the current demand for different QoS levels. We identify two approaches for maintaining *supply tokens* correctly and implementing a QoS aware version of Bolt on programmable switches.

B.1 Ideal Approach

Imagine a scenario where weighted fair queuing [46] is applied for QoS purposes. Then, Bolt would need to be able to increment the *supply token* value based on the weight associated to the QoS level (w_i) and the link capacity (C) as well as the demand for each QoS level. For example, when all QoS levels have at least 1 packet in their queue, a packet arriving at QoS level i should increment the token value by $C \times w_i \times t_{inter-arr}$.

If a QoS queue is empty, its weight is distributed to other QoS levels in proportion to each level's own weight. Therefore, Bolt should adjust the *supply token* value of QoS level i based on the logic presented in Algorithm 4.

Note that in order to be able to determine the service rate of each queue, queue occupancy of other queues would be required. This requirement creates a challenge for P4 switches since only one queue's occupancy can be read at a time. A workaround to this would be to create shadow register arrays

for each priority queue where they get updated whenever a value is not being read from them. Moreover the calculation at the line 8 of Algorithm 4 requires floating point arithmetic which could be address via lookup tables.

B.2 Heuristic Approach

A simpler mechanism to enable QoS on Bolt switches would be to introduce probabilistic SRC generation where higher priority traffic has lower probability to generate a SRC packet. This would naturally keep the rates of high priority flows high while throttling others. Yet, an extensive empirical study would be required to determine the probabilities such that the queuing for all the QoS levels are bounded to some extent.