Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT

Weitao Wang[*†], Masoud Moshref[*], Yuliang Li[*], Gautam Kumar[*],
T. S. Eugene Ng[†], Neal Cardwell[*], and Nandita Dukkipati[*]

[*]Google LLC, [†]Rice University

Abstract

The difficulty in gaining visibility into the fine-timescale hop-level congestion state of networks has been a key challenge faced by congestion control (CC) protocols for decades. However, the emergence of commodity switches supporting in-network telemetry (INT) enables more advanced CC. In this paper, we present *Poseidon*, a novel CC protocol that exploits INT to address blind spots of CC algorithms and realize several fundamentally advantageous properties. First, Poseidon is *efficient*: it achieves low queuing delay, high throughput, and fast convergence. Furthermore, Poseidon decouples bandwidth fairness from the traditional AIMD control law, using a novel adaptive update scheme that converges quickly and smooths out oscillations. Second, Poseidon is *robust*: it realizes CC for the actual *bottleneck hop*, and achieves max-min fairness across traffic patterns, including multi-hop and reverse-path congestion. Third, Poseidon is *practical*: it is amenable to incremental brownfield deployment in networks that mix INT and non-INT switches. We show, via testbed and simulation experiments, that Poseidon provides significant improvements over the state-of-the-art Swift CC algorithm across key metrics – RTT, throughput, fairness, and convergence – resulting in end-to-end application performance gains. Evaluated across several scenarios, Poseidon lowers fabric RTT by up to 50%, reduces time to converge up to 12×, and decreases throughput variation across flows by up to 70%. Collectively, these improvements reduce message transfer time by more than 61% on average and 14.5× at 99.9p.

1 Introduction

Effective datacenter congestion control (CC) needs to provide high throughput, low latency, fairness, and fast convergence across varied workloads. CC is becoming more and more critical as applications increasingly demand low-latency operations at datacenter scale. Examples of such applications include memory and storage disaggregation [9, 21, 25, 31], which require latencies as low as O(10μs) at 1M+ IOPs per server [14], and ML applications that require high network utilization to keep expensive accelerators busy [37, 45]. Large scale incasts with O(5000) flows [35] caused by shuffle operations [1] and partition-aggregate workflows continue to be prevalent and need CC to be fair across flows in order to avoid starvation and control the tail latency, which is critical for the performance of such applications [20]. Simultaneously, CC is becoming more challenging because link bandwidths are growing faster than buffers at switches [5], and high-packet-rate servers [3, 24] benefit from simple CC algorithms offloaded to NICs to save CPU for applications.

Datacenter CC algorithms in deployment today rely on either end-to-end signals (e.g., delay [35]) or quantized in-network feedback (e.g., ECN [8]), owing to their simplicity. An underlying problem with these signals is that they are aggregated *end-to-end* across all hops on a flow's path. Thus, these CC algorithms react to collective congestion along the path (for delay) or congestion at *any* hop on the path at different times (for ECN), leading to reducing a flow's rate before reaching its fair share in the network. This leads to under-utilization, slow ramp-up, and/or unfairness in multiple scenarios shown in §2.1 and §5.

However, with the emergence of commodity switches that support in-network telemetry (INT), a new opportunity has emerged. INT-enabled switches can modify or append to packet headers to convey information local to the switch, such as the time the packet spent in the queue. Some state-of-the-art CC algorithms [7, 40], use INT to gather telemetry information for every hop to gain more visibility into the network and control the outstanding packets at each hop. Still, such solutions react to congestion at *any* hop, which leads to the unfairness and ramp-up problems mentioned above.

In the last few decades, several schemes have been introduced that leverage help from network switches for better CC [13, 22, 27, 34, 40] but almost none have been deployed widely in datacenters. Based on the successful deployment of ECN-based solutions [8] and no deployments of XCP [34], RCP [22], and similar AQM solutions, we believe a *deployable* CC scheme using INT should also have the following properties: 1) works seamlessly in heterogeneous brownfield deployments where new switches and old switches co-exist and provides benefit even if a subset of switches support INT. 2) uses a simple, low-overhead, non-intrusive INT scheme that requires minimal coordination among applications, networking stacks, NICs, and switches.

Therefore, in this paper, we ask the question: *How can we harness the power of INT to design a datacenter CC algorithm that is* ***efficient*** *(high throughput, low latency, and fast convergence),* ***robust*** *(max-min fairness across traffic patterns including multi-hop and reverse-path congestion), and* ***practical*** *(simple and deployable)?*

We find that learning the congestion state of every hop of a flow is unnecessary. Instead, an efficient and practical

CC can be realized based on the congestion state of *only the bottleneck hop* of a flow – the hop that limits the rate of the flow as per the max-min fair allocation. It's worth noting that a *congested hop* is not the *bottleneck hop* for all flows passing through it, but only for flows that send more than their max-min fair-share rate, and thus, CC should ideally decrease the rate of only those flows that send above their fair-share.

Armed with this key insight, we develop a novel INT-based CC protocol called *Poseidon*. Poseidon grounds itself in Swift [35], the state-of-the-art CC that's deployed in production at scale for TCP [17] and kernel-bypass stacks [41]. But Poseidon advances beyond Swift by leveraging INT instead of purely E2E measurements, and formalizes an adaptive congestion window update function that compares the *max* per-hop delay (obtained via INT) against a *rate-adjusted target* bottleneck hop delay.

This paper makes contributions in two main areas:

First, Poseidon utilizes the power of INT to provide unique properties, like network-wide max-min fairness, monotonic fast convergence, and stable rate under high concurrency.

- By comparing the max per-hop delay against a dynamic target, Poseidon converges to the network-wide max-min fair allocation, where flows only react to congestion on their bottleneck hop. A corollary to this is that flows in Poseidon do not decelerate before reaching their fair-share rate, resulting in fast convergence.
- Poseidon provides a characterization for the spectrum of *cwnd update* and *target max per-hop delay* functions that guarantee both fairness and high utilization. This allows us to explicitly decouple the fairness objective from the rate increase-decrease function (e.g., AIMD); Poseidon leverages this to use an *adaptive* increase-decrease function (without an AI component) that accelerates arriving at the fair-share allocation and smooths oscillations around it in the presence of many flows. Poseidon uses a novel target function, which achieves low queuing delay and high utilization for both sparse workloads (a few fast flows) and high-concurrency workloads (many slow flows).
- Poseidon is amenable to incremental deployment, including seamless coexistence in brownfield scenarios.

Second, Poseidon provides a simple, practical, and deployable design for enabling INT in datacenters for CC.

- We detail an efficient INT mechanism where switches signal the maximum per-hop queuing delay on a packet's path, using only a small and fixed amount of packet header space, at line rate.
- We analyze requirements for deployable INT for CC and compare proposed formats against those requirements.

We implemented Poseidon in a production networking stack (similar to Pony Express [41]) and a testbed with commodity programmable switches, with no changes to the NIC or applications. Our testbed evaluation shows that Poseidon is robust to reverse-path and multi-hop congestion scenarios explained in §2.1. In addition, we have evaluated Poseidon

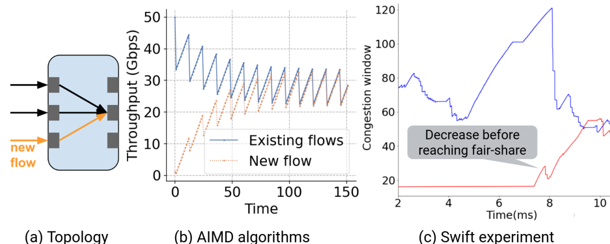


Figure 1: MD on ramping-up flows delays convergence.

extensively in packet-level simulations (§5) and show that, compared to Swift and HPCC [40], it is robust to the above scenarios. Relative to Swift, Poseidon improves application-level message transfer latency by 61% at median and 14.5× at 99.9p. This is achieved by lowering fabric RTT by more than 50%, reducing congestion window ramp-up time up to 12×, and decreasing throughput variation for flows with small windows by up to 70%. In brownfield, Poseidon achieved at least 50% of the op latency gain of full deployment.

2 Motivation

In this section, we first show how congestion control (CC) algorithms are inefficient if they cannot distinguish the bottleneck hop of a flow from a merely congested hop. Then, we motivate the importance of brownfield deployment to support incremental roll-out and highlight why the format of INT is important for deployment.

2.1 CC Challenges in Datacenters

We use several scenarios in datacenter networks to highlight how two classes of issues – reacting to signals from hops other than the bottleneck hop, and increasing with a fixed value – cause unfairness, low link utilization, and slow ramp-up.

2.1.1 Decelerating Before Reaching Fair-share

Traditionally, when a hop is congested, a flow with a lower rate (e.g., a new flow) does not increase its rate monotonically to the fair share; instead, with every congestion signal, its rate decreases. Figure 1(a) draws an example where a new flow competes with two existing flows, Figure 1(b) shows the typical behavior for AIMD algorithms, and Figure 1(c) shows the data from that experiment in production using Swift. This behavior prolongs the time for the lower-rate flow to ramp up and leads to a longer tail flow completion time. The root cause is that in current CC algorithms, all flows must react *the same way* to the congested hop (either increase/decrease) regardless of their rate. This mechanism is designed to achieve fairness and stability given an end-to-end signal (e.g., delay, loss, ECN) without coordination across flows [19]. Poseidon leverages INT to get a richer signal and allows flows to increase their rates monotonically until reaching the fair-share rate.

2.1.2 Multi-hop Congestion

Datacenter networks are usually oversubscribed at ToR and Spine layers [46], thus it is common for a flow to see multiple

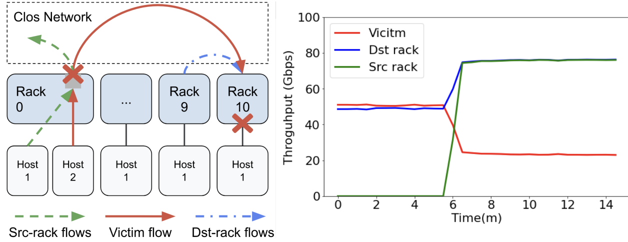


Figure 2: A Swift flow facing congestion at multiple hops (red) cannot compete at congested hops

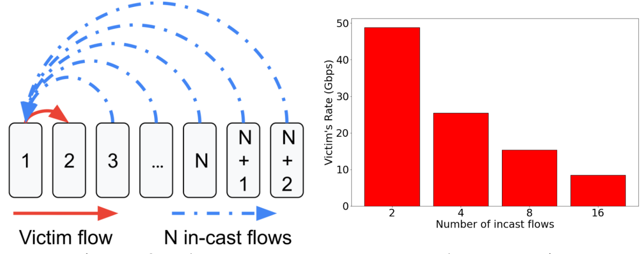


Figure 3: Flows react to reverse-path congestion.

congested hops in its path, especially in an incast. However, when a flow faces congestion in more than one hop, it gets lower throughput than other competing flows that traverse a single congested hop. The reason is that flows reacting to loss and ECN [8, 28] from multiple hops see more losses or marked packets on average, as the drops or markings happen asynchronously across different congested hops. In Swift, the fabric delay of such flows is higher, since every congested hop introduces more delay to the sum. HPCC [40], even though it uses INT, will also react to congestion at *any* hop with high in-flight bytes even if the flow is not contributing much to it.

Figure 2 shows this scenario in an experiment in production settings using Swift. The red flow (victim), that competes with the blue flow at the destination, gets much lower throughput once the green flow starts at the source ToR. The root cause is that the victim flow reacts to the congestion on Rack 0 uplink or Rack 10 downlink even when it didn't get the fair-share. This is because Swift looks at the end-to-end fabric delay and the victim's fabric delay includes both the delay at Rack 0 and Rack 10.[1] We observe the same problem even if the flow reacts to the max hop delay [8, 40] as shown in §5.6. Ideally, the victim flow should always only react to the congestion at the hop where it got more than the fair-share.

2.1.3 Reverse-path Congestion

In Figure 3, as we increase the number of flows on the reverse-path (blue), the forward traffic (red) gets lower throughput and cannot utilize the bandwidth. The root cause is that *the end-to-end delay* used in Swift includes the delay of ACKs in the reverse-path. Thus, Swift decreases the congestion window as if it is competing for the forward *and* reverse path bandwidth. This issue can happen because of congestion on any hop in the reverse-path, and can also cause unfairness if only a subset of flows on a bottleneck see reverse-path congestion, but is special for CC algorithms that use the end-to-end delay. A solution is to use synchronized timestamps at hosts (at μs level) in order to break fabric delay into forward and backward delays [39], but we show that CC can use INT to separate congestion signals of forward and reverse path and avoid the overhead of maintaining a synchronized clock.

Summary of the above three scenarios: many existing CC algorithms – when using loss, ECN, delay, or INT signals – react to every congested hop along the path, rather than only the congestion on the bottleneck hop. To put it another way, all flows going through a congested hop react the same way, either increase or decrease their rate, regardless of whether they have achieved their fair share or not. In §3.1, we show how Poseidon solves this problem by reacting to congested hops only for flows that reached their fair share.

2.1.4 Slow Convergence and Throughput Oscillation

An efficient CC algorithm should converge quickly to the right rate when the flow's rate is far from it and stay near it in a stable fashion. However, because many existing CC algorithms [8, 32, 35, 40] do not know the fair-share rate or how far they are from that rate, they rely on an AIMD, a well-understood algorithm that converges to fairness.

However, AIMD causes slow convergence for large windows and an unstable rate for small windows because AIMD increases the congestion window (*cwnd*) by a *fixed* amount every RTT. On the one hand, as *cwnd* becomes larger, the increase ratio compared to the window size becomes smaller: An increase of 1, takes 5 RTTs to double a window of 5, but 50 RTTs to double a window of 50. Slow *cwnd* growth can be particularly detrimental in workloads that desire high throughput from a few flows per host (e.g., ring topology in ML applications). On the other hand, as we increase the number of flows and get smaller *cwnd*, the effect of the increase amplifies for windows close to the additive factor. (Each one of 500 flows with a window of 1 may double its rate.) This causes oscillating *cwnd* in high-degree incast applications (e.g., shuffle [1]). A CC algorithm may use a combination of a multiplicative factor and additive factor [7, 40] for faster ramp-up, but still, the disproportionate effect of the additive increase component will manifest for a small *cwnd*.

The root cause is that AIMD was designed to provide fairness regardless of the quality of the signal (e.g., a binary loss signal in TCP Reno). Yet, it is used in many modern datacenter CC [8, 35], including the ones based on INT [40]. If we knew the fair-share from switches, we could converge faster [22, 34], but such solutions are hard to deploy. Instead, in Poseidon, flows can estimate if they are close or far from the fair-share and adjust the step size accordingly to converge faster and have a more stable throughput around the fair-share rate (§3.3), similar to some previous CC algorithms designed to facilitate large WAN BDPs [18, 28].

[1] Although the victim flow always faces a higher delay than the other two flows, its throughput didn't reach 0. The reason is that flow-scaling, designed for windows<10 [35], rises victim's target delay.

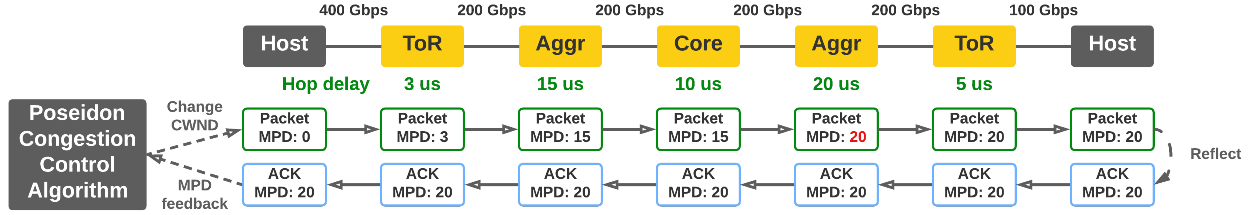


Figure 4: An example of Poseidon MPD signal propagation.

2.2 Deployment

Brownfield deployment. An important requirement for deploying INT in production is to support brownfield deployments. Hardware may be replaced gradually, from the ToR level to higher levels, or from one pod to another [26]. This transition phase can last for years [47]. INT may not be enabled on some switches, and at any point, we may want to roll back to disable INT without coordinating hosts and switches. Therefore, even if we use a separate queue for the new traffic [35], we still have to address the following requirements:

1. Being able to route the traffic regardless of whether a switch has INT enabled or not: While two hosts can coordinate their capabilities during a connection's initial handshake, we don't want any coordination between hosts and switches or switches with each other in order to forward packets and use ECMP. This places a tight requirement on the format of INT packets, as discussed in §4.2.
2. Getting some gains on an incremental INT deployment: Even though only a subset of hops supports INT, the CC algorithm should still benefit from that partial information.
3. Fair interaction between flows that have INT support on every hop and those that have it only on a subset of hops.

In §4.1, we explain why adjusting the target helps deploy Poseidon in brownfield. We also present our solution to combine end-to-end delay and max-hop delay to keep fairness while providing some incremental benefit in brownfield.

Low-overhead non-intrusive INT. For easy deployment, we prefer coordinating the least number of components and sustaining minimum overhead. Above, we mentioned that the traffic must go through the brownfield without any coordination between hosts and switches. At the end-host, we also want minimum coordination between applications, networking stack, and NIC. For example, a fixed INT length is preferred as it doesn't change MTU.

We want INT on all packets, so its overhead regarding bandwidth and packet processing in the hosts, NIC, and switches is important. Small INT length is preferred for low bandwidth overhead and easy deployment in offloaded NICs [10, 11]. Finally, INT information cannot be encrypted, require complex functions, or rely on the per-flow state in the switch.

There are multiple formats for supporting INT, two of which are IFA [36] and P4-INT [2]. These formats differ in multiple aspects. Instead of proposing yet another format, we describe the features required for an INT format to be deployable in a production datacenter for CC. §4.2 covers these requirements and how the formats satisfy them.

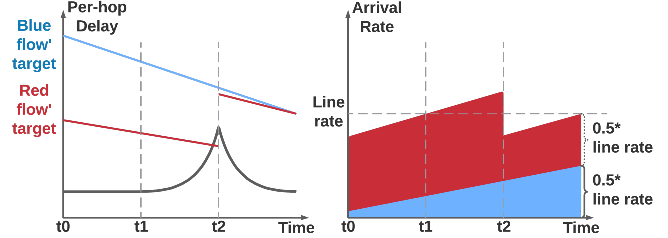


Figure 5: Delay is bounded by the faster flow's target.

3 Design

Poseidon achieves high link utilization, low queuing delay, network-wide max-min fairness, with fast convergence and stable per-flow throughput. In this section, we describe the design of Poseidon: First, we introduce a key idea that allows Poseidon to only react to the bottleneck hop (§3.1). Next, we demonstrate how Poseidon guarantees fairness on a single hop (§3.2) and how decoupling the fairness from the *fixed* increase in AIMD allows us to introduce an adaptive increase/decrease algorithm that achieves faster convergence and more stable throughput than AIMD (§3.3). Finally, we show that Poseidon achieves network-wide max-min fairness (§3.4).

3.1 Key-idea: Only React to Bottleneck Hop

Poseidon only reacts to the bottleneck hop by decreasing the congestion window only if the flow got the fair-share on congested hops over its path. We explain how to do that without knowing the fair-share. Poseidon compares a delay signal with a target delay to increase or decrease the window. The key idea is in the definition of the delay signal and target:

1. It applies the target to the **maximum per-hop delay (MPD)** to allow flows to react to the most congested hop.
2. It adjusts the **target** based on the **throughput** of the flow to make sure only the flows that get the highest rate on the hop reduce their windows.

Figure 4 illustrates an example of how max per-hop delay is propagated. Each packet carries the MPD and each hop updates it. The ACK packet will reflect MPD back to the source. Note that Poseidon could naturally support heterogeneous link bandwidth in the network.

Now we describe each point in more detail. Every flow tries to maintain the **maximum per-hop delay (MPD)** close to a **maximum per-hop delay target (MPT)**, namely, increasing the congestion window when $MPD \leq MPT$ to keep the link busy and decreasing the window when $MPD > MPT$ to limit the congestion. MPD adds small and fixed overhead to packets and is one of the important designs to find the bottleneck hop:

In the max-min fair state, the hop with maximum latency is the bottleneck hop of the flow for Poseidon; otherwise, the flow has not reached its fair-share along its path (§3.4). The former case must decrease the congestion window, and the latter must ignore the congestion and increase the window. We achieve that by adjusting the target.

Poseidon calculates MPT for each flow based on its rate: **the larger the rate is, the smaller MPT will be** (§3.3 defines the function). This means that flows with higher rates have lower targets, thus decreasing their window earlier and more aggressively[2]. This became possible using INT, as now all flows competing in the same queue tend to observe the same congestion signal (per-hop delay). Figure 5 shows an example: As the arrival rate on the link goes over the line rate at time $t1$, a queue builds up. The hop delay grows over the target of faster flow, red, and forces it to reduce its window at $t2$. However, the slower flow, blue, can still increase its window (solves §2.1.1). Interestingly, this means that given the same congestion signal from the network, some flows increase and some decrease their rate. In the next section, we demonstrate how Poseidon achieves fairness given this flexibility without relying on an additive increase.

Algorithms 1 shows how Poseidon updates the congestion window (*cwnd*). The pacing only happens when the *cwnd* is less than 1, similar to Swift [35]. Note that the multiplicative-increase (MI) happens per packet, thus Line 5 in Algorithm 1 has to approximate the ratio for each packet, while *cwnd* decreases happen only once per RTT, thus it is a simple multiplication. The retransmit and recovery functions are included in Appendix A.

Algorithm 1: Poseidon's Main Algorithm

Input: *mpd*: maximum per-hop delay,
cwnd: flow's congestion window size,
rtt: round-trip time,
now: current timestamp
Parameter : *min_md*: minimum MD ratio,
max_mi: maximum MI ratio,
min_cwnd: minimum cwnd,
max_cwnd: maximum cwnd

```
1  Function ReceiveACK():
2      mpt ← T(cwnd/rtt)
3      update_ratio ← U(mpt, mpd)
4      if mpd ≤ mpt then
5          cwnd ← cwnd * (1 + (update_ratio − 1)/cwnd * num_acked)
6      else
7          if now − t_last_decrease > rtt then
8              cwnd ← cwnd * update_ratio
9      return cwnd
10 Function Poseidon():
11     cwnd_prev ← cwnd
12     if is_ack then
13         cwnd ← ReceiveACK()
14     else if is_retransmit then
15         cwnd ← RetransmitTimeout()
16     else if is_fast_recovery then
17         cwnd ← FastRecovery()
18     cwnd ← clamp(cwnd, min_cwnd, max_cwnd)
19     if cwnd < cwnd_prev then
20         t_last_decrease ← now
21     pacing_delay ← 0
22     if cwnd < 1 then
23         pacing_delay ← rtt/cwnd
24     return cwnd, pacing_delay
```

3.2 Single-hop Fairness

We show that with the right increase/decrease functions, Poseidon can achieve fairness on a single hop. The AIMD algorithm benefits from the fact that *all* flows either increase rate with the same amount or decrease rate with the same ratio [19]. However, because of Poseidon's rate-adjusted target delay and delay-based increase/decrease function, Poseidon has a new case, where *faster flows decrease rate* while *slower flows increase rate*. This happens if the queuing delay is higher than the faster flow's target, but lower than the slower flow's target.

To prove that Poseidon can achieve fairness, we show that fairness improves in all possible cases:

1. MPD is low, and all flows increase rate.
2. MPD is high, and all flows decrease rate.
3. MPD is high, some faster flows decrease, other slower flows increase their rate.

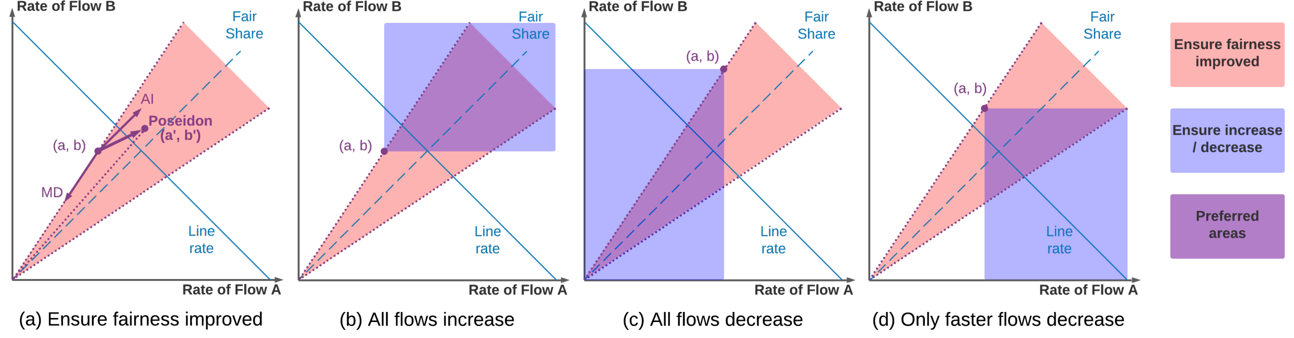
Assume a queue with two flows A and B with rates a and b where $b > a$. As a result, the target of A is larger than the target of B ($T(a) > T(b)$). In Figure 6, the fairness is graphically defined as the angle between the actual bandwidth share and fair-share line. We define the update function $U(T(rate), delay)$ as the multiplicative factor (where $new_cwnd = cwnd \times U()$) with a specific flow rate and network delay. In order to converge to the line rate, it is ≥ 1 if the delay is less than or equal to the target and < 1 if the delay is more than the target[3].

$$U(T(rate), delay) = \begin{cases} \geq 1, delay \leq T(rate) \\ < 1, delay > T(rate) \end{cases} \quad (1)$$

In all three cases, if we want to guarantee that the fairness improves, the updated rates should stay in the red triangle

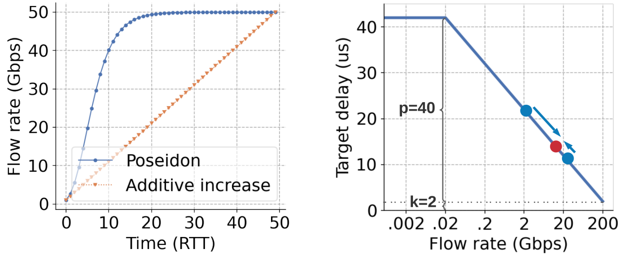
[2] In rare cases, the queuing delay of a port may jump over the target of both fast and slow flows because of synchronized packet arrival. We make sure that faster flows with smaller targets decrease more aggressively (§3.3)

[3] We assumed, in average, if arrival rate < line rate, delay is low, and if arrival rate > line rate, delay increases.



(a) Ensure fairness improved (b) All flows increase (c) All flows decrease (d) Only faster flows decrease

Figure 6: Poseidon updates the rate (in the purple area) such that it increases fairness (red) toward the line rate (blue). b) the queue is under-utilized and both flows increase rates; c) the queue is overloaded, and both flows decrease rates; d) the faster flow decreases, and the slower flow increases its rate.



(a) AI takes 50 RTTs from 1G to 50G, Poseidon takes around 15 RTTs and is stable at the fair-share rate.

(b) Poseidon target function has high resolution over all spectrum of rate (min-rate=0.02G, max-rate=200G).

Figure 7: The ramp-up using adaptive step sizes is fast and slows down near the target for stability.

(Figure 6(a)). One side of the triangle is defined by the current ratio of rates, and the other side is symmetric across the fair-share line. If we assume $a < b$ and the delay is D, this requirement can be written as:

$$\frac{a}{b} < \frac{b \cdot U(T(b), D)}{a \cdot U(T(a), D)} < \frac{b}{a}, \forall a < b, \forall D > 0$$
$$\frac{a^2}{b^2} < \frac{U(T(b), D)}{U(T(a), D)} < 1, \forall a < b, \forall D > 0 \tag{2}$$

In summary, Poseidon achieves high link utilization and fairness if the functions $T()$ and $U()$ satisfy Eq. 1 and Eq. 2. Figure 6 illustrates Eq. 1, updates that allow full link utilization, in blue color, and Eq. 2, updates that converge toward fairness, in red. The desirable overlapped area is marked in purple. The additive increase will be in parallel to the fair-share line, and the multiplicative increase/decrease with the same ratio stays on the same edge of the red triangle where the node (a,b) is (Figure 6(a)). For case 1 in Figure 6(b), the red area ensures the fairness is improved, and the blue area ensures all flows increase their rate; for case 2 in Figure 6(c), the blue area ensures all flows decrease their rate; for case 3 in Figure 6(d), the blue area ensures the slower flow increases rate while the faster flow decreases rate. Next, we introduce a target function $T()$ and the update function $U()$ which satisfy the above requirements and have more desirable properties.

3.3 Adaptive Update Steps

Based on §3.2, Poseidon can use any function that satisfies Eq. 1 and Eq. 2. But we designed the following functions to leverage the *distance* between the target and max-hop delay to not only decide whether to increase or decrease, but also adjust *the update ratio adaptively* to reach a better trade-off between stability and fast convergence. Appendix B proves that they satisfy Eq. 1 and Eq. 2:

$$T(rate) = p \cdot \frac{ln(max_rate) - ln(rate)}{ln(max_rate) - ln(min_rate)} + k$$
$$min_rate < rate < max_rate, p > 0, k > 0 \tag{3}$$

$$U(T(rate), delay) = \exp\left[\frac{T(rate) - delay}{p} \cdot \alpha \cdot m\right]$$
$$\text{where } \alpha = ln(max_rate) - ln(min_rate) \tag{4}$$

$rate$ is $cwnd * MTU/RTT$. k defines the minimum target delay; p tunes the maximum target when the rate is equal to min_rate and decides how far-apart the target of two flows with close rate can be. In practice, the target cannot be lower than a limit without decreasing the throughput because synchronized arrivals can cause premature window decrease. The target cannot be very large too because a) it can cause packet drops in switches when the target delay exceeds the queue capacity; b) as long as we achieve high utilization, we prefer to back-pressure hosts to leverage other mechanisms such as load-balancing and admission control for isolation. We use min_range and max_range to not waste the target range for differentiating rates that only happen rarely [35]. m defines the "step" when updating the rate. The larger m is, the slower the rate of update will be (sensitivity analysis is in §5.6.2).

When $|T(rate) - delay| \to 0$, then $U(rate, delay) \to 1$. This means when the delay is far away from the target, flows increase/decrease more drastically for faster convergence, and when the delay approaches the target delay, the steps will be more gentle to achieve stable flow rates (solves §2.1.4). Figure 7(a) shows how the flow can quickly increase its rate to reach 50 Gbps using the adaptive solution. We explain the intuition behind the update function with an example. Assume

the rate of a flow is x, and the target delay is D. We define the target rate r, such that $T(r) = D$ thus $U(r,D) = 1$. We can rewrite the update function for the flow as follows (calculation is in Eq. 11):

$$U(x,D) = \frac{U(x,D)}{U(r,D)} = \left(\frac{r}{x}\right)^m \tag{5}$$

Thus, the update function is only related to the ratio of r and x; when x is far-away from r, the change will be larger. Poseidon updates the rate of flow from x to r in one RTT, because $x \cdot U(x,D) = r$ if $m = 1$, and for $m < 1$, it will take more RTTs because $x \cdot U(x,D)^m = r$. In this way, the parameter m controls how fast Poseidon converges to the fair-share rate.

A legitimate alternative for $T(rate)$ is $\frac{\alpha}{\sqrt{rate}} + \beta$ which is an extension of the Swift flow-scaling (Appendix §C). However, we designed Eq. 3 because it gives a meaningful difference between the target of flows over *all* rates: The target of a flow with rate a and $c \cdot a$ have a fixed difference $T(a) - T(c \cdot a) = ln(c)/p$, providing uniform resolution across all ranges of rates (Figure 7(b)). This generalizes Swift's use of $1/\sqrt{cwnd}$ for target flow scaling (§3.5 of [35]), which only provides high resolution for small windows. Similarly, an option for the update function is to use the ratio of target over delay, similar to Swift. Appendix B.3 shows why distance provides a better result in high concurrency scenarios.

3.4 Network-wide Max-min Fairness

The key designs of Poseidon to achieve network-wide max-min fairness are: 1) only react to the max-hop delay; 2) the target delay of a flow increases when the flow rate decreases. We will start from the definition of max-min fair and then show how the above two designs achieve max-min fairness.

Definition 1 (Max-min Fairness [16, 38])**.** A feasible allocation of rate $\vec{x}$ is "max-min fair" if and only if an increase of any rate within the domain of feasible allocations must be at the cost of a decrease of some already smaller rate. Formally, for any other feasible allocation $\vec{y}$, if $y_s > x_s$ (s is a flow), then there must exist another flow s' such that $x_{s'} \leq x_s$ and $y_{s'} < x_{s'}$.

For a certain network and workload, the max-min fair allocation is unique [38]. In the max-min fair allocation, for each flow, there is a unique queue (switch port), which restricts the rate for that flow. We denote this queue as a flow's **bottleneck**, and the flow's rate should be the fair-share rate of that queue. (As a special case, a flow's bottleneck can also be the source or destination host, if either of them restricts the rate of the flow.) Specifically, we can conclude the following Lemma from the above definition (proved in Appendix D):

Lemma 1. *When achieving network-wide max-min fairness, each flow will have the largest rate among all flows on its bottleneck hop and not on any other saturated hop.*

Formally, for the "max-min fair" allocation $\vec{x}$, for any flow s, denote the flows that traverse s' bottleneck as $\{b_1, b_2, ..., b_k\}$,

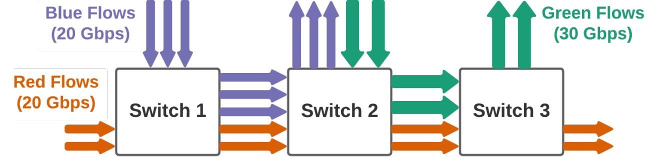


Figure 8: The stable state of max-min fairness among 3 switches with 100 Gbps links.

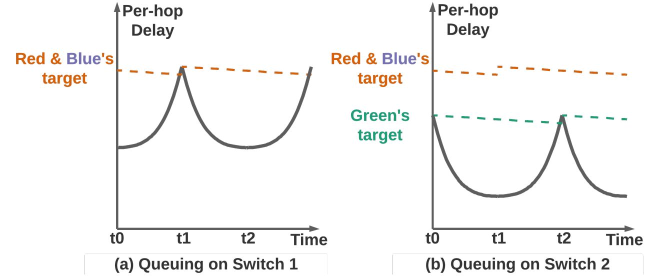


Figure 9: Only the queuing delay on red flows' bottleneck (switch 1) can reach red flows' target.

then for any flow b_i, $x_s \geq x_{b_i}$. Denote the flows that traverse one of the saturated non-bottleneck hops of s as $\{c_1, c_2, ..., c_k\}$, then there must exist some c_j such that $x_{c_j} > x_s$.

With the above definition and Lemma, we first give an *intuition* about why Poseidon could converge to the max-min fair state from any initial state.

With other CC algorithms, the hop with max queuing delay for a flow may not be the bottleneck hop based on the max-min fairness. Thus, using INT naively and reacting to the max delay cannot lead to max-min fairness. However, Poseidon uses a monotonically decreasing target function, which lets faster flows have lower target delay. With this design and Lemma 1, a flow should have the smallest target among all other flows on its bottleneck, and its target is never the smallest on other congested hops. Moreover, the delay on a queue will generally remain close to the minimum target among all flows on that queue. So gradually, the delay may reach a flow's target on its bottleneck; but on other congested non-bottleneck hops, the delay is not able to reach its target. Thus, **in Poseidon's final stable state, the max hop delay must come from flow's bottleneck**. And because each flow only reacts to its bottleneck, it achieves fairness on the bottleneck with other flows that have the same bottleneck (§3.2). Then, the network-wide max-min fairness is achieved by Poseidon.

We provide an example in Figure 8 where green flows have higher rates than red and blue flows in max-min fair state, $r_{green} > r_{red} = r_{blue}$, so green flows also have smaller targets, namely, $T(r_{green}) < T(r_{red}) = T(r_{blue})$. Switch 1 is the bottleneck of red and blue flows, and switch 2 is the bottleneck of green flows. On switch 2, the delay is similar to the target of green flows, $d_{sw2} \approx T(r_{green})$, because the moment the delay passes the target, green flows reduce their rate. Meanwhile, red and blue flows have higher targets than the delay d_{sw2}, as shown in Figure 9(b). This prevents red flows from reacting to the queuing on switch 2, which means every flow only

reacts to its bottleneck and maintains max-min fairness. This property of Poseidon solves the problem mentioned in §2.1.2.

Theorem 1. *Poseidon converges to the max-min fairness.*

To formally prove that the network converges to the max-min fair state, we use induction to prove that each queue achieves max-min fair. Denote the max-min fair rate allocation as $\vec{x}$, and for each queue, we denote the fastest flow's rate on that queue as R_q^x. Then we sort all the k queues in the network according to R_q^x from smallest to largest: $R_{q_1}^x \leq R_{q_2}^x \leq ... \leq R_{q_k}^x$. For any other flow rate allocation $\vec{y}$, with induction: **(1)** we prove that the queue q_1 will converge to the max-min fair allocation; **(2)** assuming the queue q_1 to queue q_m have already converged to the max-min fair allocation $\vec{x}$, we prove that the queue q_{m+1} will also converge to $\vec{x}$. A detailed proof is provided in Appendix §E.

4 Deployment

Here, we discuss the design decisions that facilitate the deployment of Poseidon in a large-scale datacenter network. Firstly, Poseidon provides benefits even if only part of the network supports INT (incremental deployment), and bounds the unfairness between flows that see INT vs those that do not. Secondly, Poseidon allows old switches to transparently route INT traffic, adds minimum overhead to packets and switches, and requires no changes in applications or NICs.

4.1 Brownfield Deployment

For a network where a subset of switches can provide hop delay information, Poseidon splits the fabric delay into two parts: the MPD from switches equipped with INT; and the delay from the rest of the path. This is calculated based on the end-to-end delay, using the NIC timestamp similar to Swift [35], minus the max-hop delay (both forward and backward). Then we apply Poseidon based on the maximum of the two. Note that this solution is not robust to reverse-path or multi-hop congestion happening in the hops that do not have INT, but still provides incremental benefits (§5.5).

The fairness issue is only relevant if the bottleneck hop of the two flows is the same. Consider two flows A and B and three switches, X, Y, and Z. A goes through switch X to Z, and B goes through Y to Z. The common switch, Z, is the bottleneck, X supports INT and Y doesn't. If Z has INT, both flows get the right feedback about Z in max-hop delay. Therefore, we get partial benefits. If Z doesn't have INT, the fabric delay of flow A doesn't include the delay of X, but for flow B it will include the delay of Y. Therefore, flow B observes a high delay and may decrease its window sooner. However, we argue that this decrease will be minimal and bounded because of target scaling. As the rate of flow B goes down, its target will go higher. The moment the target increases by the delay at hop Y, the rate of flow B will stabilize.

Interestingly, the above argument suggests that in order to get most of the benefit, we should prioritize deploying INT in the usual congestion points (ToRs with oversubscribed uplinks or incast in downlinks). We evaluate this in §5.5.

4.2 A Deployable INT Format for CC

In this section, we describe the requirements for deploying INT in datacenters for CC and compare existing INT formats.

4.2.1 Requirements

We consider both the INT metadata that we ask from each hop and how/where we put it inside the packet.

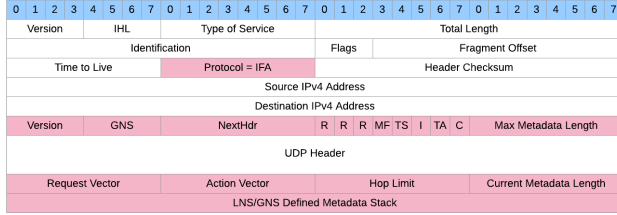
Make INT information available to the sender for CC: INT metadata on the forward path should be reflected in the reverse path ACKs for CC signaling. Ideally, ACKs could reflect opaque information that could be carried in the INT header but not be replaced by switches. Or, similar to ECN, INT could be marked by switches in the forward direction and echoed back to the sender in L4 headers.

Low-overhead INT metadata: For simplicity and precision, we want INT on all packets, thus its bandwidth and processing overhead must be low. Having many metadata fields per packet adds bandwidth overhead [15] and is costly for switches, NICs, and offloaded transports to process [11].

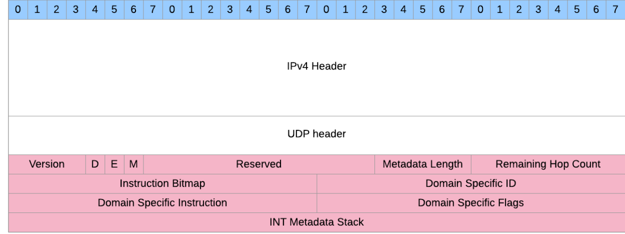
Fixed-sized INT metadata: Per-hop INT metadata makes the number of INT fields not only large but also variable. This is bad for two reasons: a) It is wasteful to reduce MTU for the worst case because link failures may add more hops to packets transiently, so that the number of hops is long-tailed. A smaller MTU means more packets to be processed by hosts and switches. b) Variable-sized INT metadata is more complex to parse at switches, offloaded transports, and middleboxes if they access bytes after the INT header. Therefore, for the CC use case, it is essential for an INT format and its implementation in the switch to use fixed-size INT metadata, and support *aggregation* functions (e.g., max/min/sum) that can overwrite the information from the previous hop.

Implementable in dataplane at line rate: The aggregation function must require minimal state and computation in the switch. This means that it must be simple (e.g., max/min/sum) and not require per-flow state.

Transparent to routing: Many datacenters use a hash-based scheme (ECMP [30], WCMP [48]) to balance the load over multiple ports. Such schemes may use the 5-tuple and/or IPv6 flow label. A brownfield deployment requires a scheme that balances load efficiently for packets with/without INT metadata in switches with/without INT support. For switches without INT support to balance INT traffic, they must be able to find and parse L4 headers. Thus, we either need to a) put INT metadata after L4, b) enable switches to pass over the INT metadata by adding it as a sub-header in headers that support extensions, such as VLAN-tag, MPLS-tag, IP option, GRE shim layer, or VXLAN shim layer. We believe option (a) is easier to deploy as it is transparent to the network, and thus works with different L2/L3 protocols, with virtualized and



(a) IFA packet format



(b) P4-INT packet format

Figure 10: INT packet format in a) In-band Flow Analyzer (IFA) b) P4-INT

non-virtualized traffic, and with other boxes (except special middleboxes, which are usually implemented in extensible software anyway [23, 43], and don't need to parse INT).

Compatible with encrypted packets: Many cloud providers encrypt network traffic inside datacenters [4]. But switches must be able to change INT data. Fortunately, recent NIC encryption modules such as PSP [6] allow passing an offset in the packet descriptor to only encrypt the bytes after. PSP also only authenticates the bytes after its header. For UDP checksum, PSP requires its implementation to support zero values (thus there's no need to rely on switches, even though programmable switches can update that). We also verified that we can change where the NIC expects the PSP header.

4.2.2 INT formats

Figure 10 shows two predominant INT formats in the context of IPv4; the IPv6 format is similar. Poseidon is possible on both formats, but a few improvements help its deployability.

IFA [36] indicates the presence of INT with a special protocol value in the IP header and adds part of the header between the IP and L4 header. P4-INT [2] indicates INT using a DSCP (traffic class) value/bit and puts all metadata after the L4 header. In order to use ECMP on switches without changing their configuration, we prefer to not change the location of L4. Still, IFA can be used on most switches by first changing the expected location of the UDP header for IFA packets (using User Defined Fields, UDF) and then rolling out INT. IFA also supports a format that puts INT metadata at the tail of the packet to avoid changing the location of L4.

Neither of the formats has a place to reflect the forward path INT. But given that the switches in the reverse path don't need to read the forward path INT, the receiver can just reflect the INT metadata in L4 headers, and the sender networking stack will consume it along with the INT metadata.

Neither of the formats defines a max calculation action, however, they allow extending the action vector.

Finally, we believe an overhead of 12B for sending a 2B metadata is excessive for small packets and look forward to working with the community to reduce overhead while maintaining protocol flexibility.

5 Evaluation

First, §5.1 explains our prototype implementation on a testbed with a production networking stack and NIC to highlight the ease of implementation and show the robustness of Poseidon to multi-hop and reverse-path congestion. Then we use simulations to explain how and why Poseidon is robust in those scenarios (§5.2). §5.3 shows that the adaptive window update enables faster convergence and better stability. Next, we present the aggregate benefit of the above techniques on op latency (flow completion) in multiple scenarios (§5.4). Finally, we wrap up with brownfield results (§5.5) and a parameter sensitivity analysis (§5.6). We use **Swift**, a practical CC deployed at scale, and **HPCC**, the state-of-the-art in INT-based CC, as our main points of comparison.

Simulation setup: We implemented Poseidon along with Swift and HPCC in the OMNeT++ packet simulator and simulated a Clos network of 200 Gbps links, with 245 ns link delay (including 230 ns FEC delay), 600 ns switch delay, 64 MB buffer size, and 4096 Bytes MTU size. For Poseidon, we set the parameters in Eq. 3 and Eq. 4 as $p = 40$, $k = 2$, $m = 0.25$ based on §5.6.2. For Swift, we follow the best parameters in [35] and set the base delay to 25 μs, the max flow scaling to 100 μs, and the hop-based scaling to 1 μs per hop. We verified the fidelity of the simulator by comparing it to the result of the testbed. Note that RTT here is calculated based on NIC timestamp and doesn't include the delay in the networking stack at the host. For HPCC parameters, we use the values from the paper [40]. To be fair in our comparisons, we enable pacing only when $cwnd < 1$, similar to Swift[4].

5.1 Implementation in Testbed

For the host networking stack, we change Swift implementation in a transport stack similar to Pony Express [41] to 1) at the sender, add a 2-bytes INT header for max-hop-delay right after L4; 2) at the receiver, reflect back the max-hop-delay in another 2-bytes in the payload; 3) at the sender, update the congestion window based on Poseidon algorithm in §3.3.

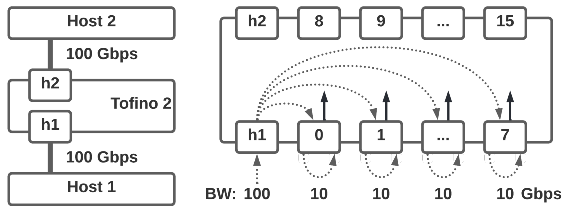
For network switches, we extract the queuing delay at the egress pipeline and update the max-hop-delay in the INT header. We implemented the P4 program with only 2 lines of P4 code (Listing 1) and 16 lines of parser/deparser code in a Tofino switch. Moreover, we verified that packets with or without INT headers can both be routed.

Our testbed only has two hosts and one switch (Figure 11(a)). To simulate congestion from multiple hosts, we create 8 virtual interfaces in hosts with 100G links and route the

[4]HPCC always paces packets, but that is costly in software and hardware.

```
bit<16> queuing_delay =
    (bit<16>) (eg_intr_md.deq_timedelta >> 8);
hdr.telemetry.max_hop_delay =
    max(hdr.telemetry.max_hop_delay, queuing_delay);
```

Listing 1: Core P4 code for telemetry in Poseidon.



(a) Testbed topology (b) Create virtual hosts

Figure 11: Testbed with 8 virtual sender/receiver ports

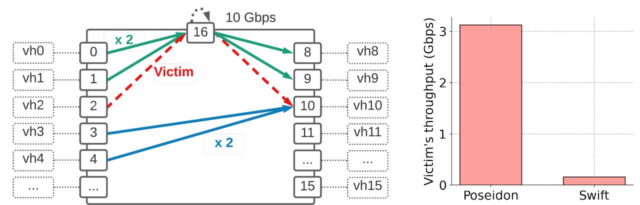
traffic insides the switch based on the virtual IPs into separate loopback ports. Each port, loopback at MAC layer, is configured to 10Gbps and plays the role of a virtual sender/receiver (Figure 11(b)). Ports 0-7 receive traffic from host 1, and ports 8-15 pass the traffic to host 2.

Testbed Results: To create multi-hop and reverse-path congestions from Figure 2 and Figure 3, we route the flows between virtual senders/receivers as Figure 12(a) and Figure 13(a) show. For the multi-hop congestions, Poseidon could fairly share the bandwidth between background flows and the victim flow, while Swift only spares 0.16 Gbps for the victim flow in Figure 12(b). For the reverse-path congestion, Poseidon could achieve line rate for the victim flow, while Swift could only achieve 1.91 Gbps (similar throughput as the flows on the reverse-path) as shown in Figure 13(b).

5.2 Robustness From Max-min Fairness

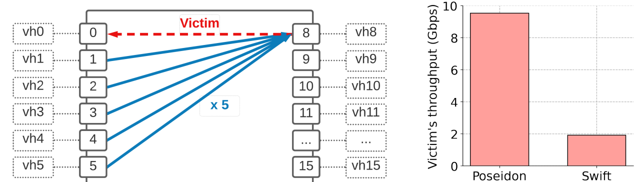
Poseidon achieves fairness in multi-hop congestion. Consider the scenario in Figure 14(a) where we have M green flows at Rack 0 and N blue flows at Rack 10. We add more flows to change M and N to create different multi-hop congestion scenarios. With Swift, the moment we have congestion at multiple hops ($M > 0$ and $N > 0$), the victim flow, red, cannot compete with other flows (Figure 14(b)). The reason is that Swift reacts to the inflated sum of delays (Figure 14(c)). Therefore, the victim reduces its congestion window until its scaled (because of flow-scaling) target delay matches this larger end-to-end delay. HPCC and DCTCP also react to the congestion at *any* hop, thus when $M = N$, the victim does an MD when *either* of the hops gives a congestion signal (ex: ECN) and cannot achieve the fair rate.

Poseidon, however, allows the victim flow to achieve its max-min fair share ($200Gbps/max(M+1, N+1)$) by only reacting to the bottleneck hop where it gets the fair-share. One reflection is that Poseidon's congestion signal, max-hop delay, and target only changes when the bottleneck hop or its congestion changes. For example, they stay the same when



(a) Victim from virtual host 2 (vh2) contends with 2 flows on port 16 and 2 flows on port 10. (b) Victim achieves fair-share rate in Poseidon.

Figure 12: Multi-hop congestion. (Linerate: 10 Gbps)



(a) 4 flows create a congestion on the victim's reverse-path. (b) Victim achieves linerate (10 Gbps) in Poseidon.

Figure 13: Reverse-path congestion. (Linerate: 10 Gbps)

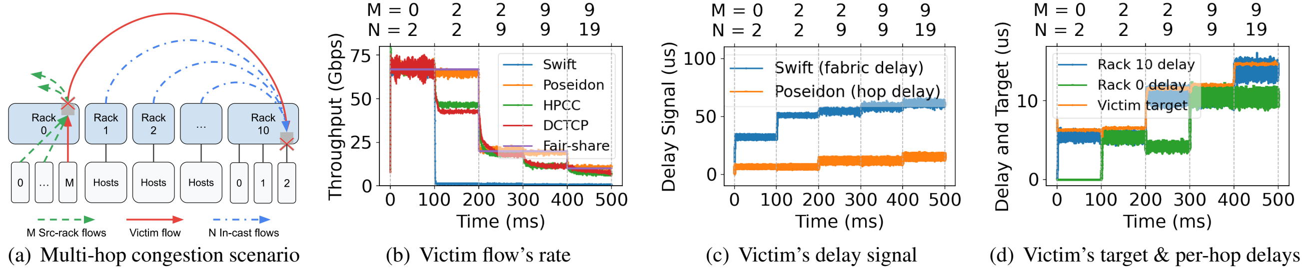
M changes from 0 to 2, but change when N increases from 2 to 9 in Figure 14(c) and Figure 14(d). Although the victim flow experiences higher RTT than other flows, Poseidon uses a higher congestion window to achieve the fair rate. Another interesting point happens when both hops have the same fair-share rate ($M = N$). Although the delay of both hops is close to the target (Figure 14(d)), with a rate-adjusted target, the moment the victim reduces its window, Poseidon raises its target and will not react to the max-hop delay until the rate increases again. §5.6 shows that both max-hop latency and scaling the target are necessary to achieve fairness.

Poseidon utilizes forward path regardless of reverse-path congestion. Reproducing the scenario in Figure 3, Figure 15(a) shows that with Swift, as the number of flows on the reverse-path, N, increases, victim's throughput decreases to the fair-share rate in reverse-path. However, with Poseidon, the victim could maintain 200 Gbps (line rate). The reason is that Poseidon only uses the max-hop delay from the forward path, which is not affected by the reverse traffic (Figure 15(b)). HPCC doesn't have this problem since it only uses INT information on the forward path.

5.3 Fast Convergence and Stability

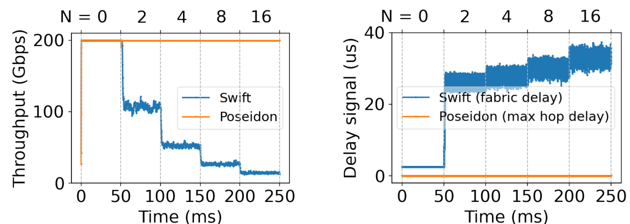
Figure 16 shows the rate of flows in Swift and Poseidon as we add competing flows one by one and then remove them. At a *single* hop, not only does Poseidon achieve the fair-share, similar to Swift, but also lower throughput variation, hence better stability. Next, we evaluate Poseidon's convergence time and throughput stability.

Poseidon converges fast for flows with large windows. Figure 17(a) shows the ramp-up phase of a flow, growing its window to a large value. This flow is competing with another one on a 200G link. First, the ramp-up shows that Poseidon does fewer rate reductions than Swift and HPCC. Second, it shows that Poseidon achieves a super-linear ramp-up at



(a) Multi-hop congestion scenario (b) Victim flow's rate (c) Victim's delay signal (d) Victim's target & per-hop delays

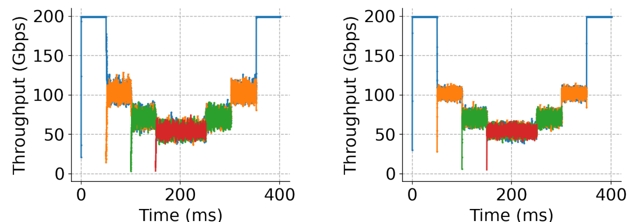
Figure 14: Multi-hop congestion with the same or different fair-share rate on different hops (linerate: 200 Gbps).



(a) Victim's rate is protected by Poseidon (linerate: 200 Gbps) from the congestion on the reverse-path.

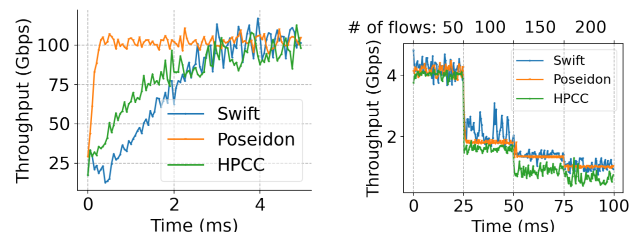
(b) Victim's congestion signal never changes for Poseidon, despite increasing delay on the reverse-path.

Figure 15: Reverse-path congestion: N reverse flows



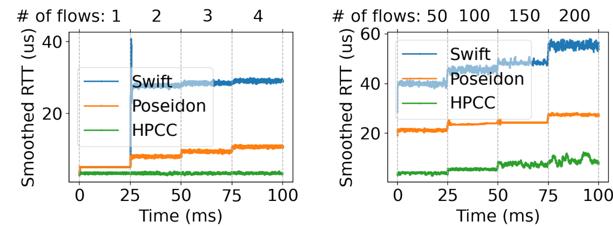
(a) Single hop fairness in Swift (b) Single hop fairness in Poseidon

Figure 16: Fairness on a single hop with step-in&out flows, throughput is measured every 50 μs.



(a) Fast convergence for big windows (b) Stability under high concurrency

Figure 17: Poseidon achieves fast convergence for flows with large windows & stable rate for flows with small windows.



(a) Large window: +1 flow per 25ms (b) Small wnd: +50 flows per 25ms

Figure 18: Poseidon achieves lower RTT than Swift by keeping queues short and stable.

the beginning, and the rate of growth decreases as it reaches the fair-share rate as expected in Figure 7(a). As a result, Poseidon converges around 12× faster than Swift and HPCC.

Poseidon achieves stable throughput for flows with small windows. Figure 17(b) shows the throughput for a flow competing with $N-1$ others on a hop. Poseidon reduces the standard deviation of throughput by 24× for $N = 200$ (70% on average over the four cases). As mentioned in §3.3, the reason is that AIMD in Swift and HPCC becomes more aggressive for smaller congestion windows. By contrast, Poseidon uses an adaptive update ratio to adjust the congestion window.

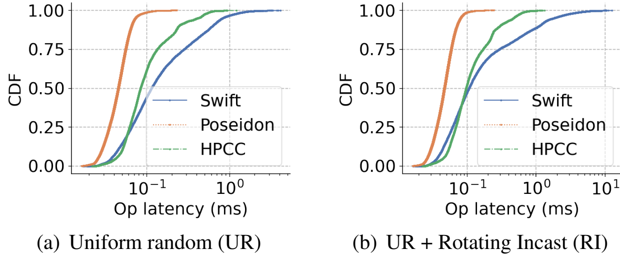
Poseidon keeps link utilization high with low RTT. Poseidon achieves a smaller RTT than Swift for flows across different rates, which means lower latency for small messages. For example, Figure 18 compares the RTT over different numbers of flows in two cases: large window and small window. Swift cannot use a very low target delay because, for high link utilization, it has to accommodate the summation of delays on multiple hops and the variation of delay in a high-degree incast caused by AI (Figure 17(b)). Since the adaptive update ratio can stabilize the rate, Poseidon could afford to use a tighter target and achieve high link utilization and small queues at the same time. HPCC, however, achieves lower RTT than Poseidon as it targets near-zero in-network queues at the cost of op latency (§5.4).

5.4 Application-level Improvements

A key application-level performance metric is op latency, namely, the time from a message was enqueued for sending to its completion [22]. We create two scenarios on two racks (A and B) with 3:1 oversubscription and compare op latency for 128 KB messages:

1) Uniform Random (UR): Rack A sends 960 Gbps to Rack B (60% uplink load), while Rack B sends 480 Gbps to Rack A (30% uplink load). The source and destination hosts are randomly chosen. Poseidon has a 61% lower median and 14.5× lower 99.9p op latency than Swift (PLB [44] enabled), 44% lower median and 5.49× lower 99.9p op latency than HPCC in Figure 19(a). This mostly comes from robustness to reverse-path and multi-hop congestion.

2) Uniform Random with Rotating Incast (UR+RI): A and B communicate similar to UR scenario, but Rack A also suffers from rotating incast from 100 hosts in other racks (not A or B). The incast traffic has 100 flows with 0.5 Gbps load



(a) Uniform random (UR)

(b) UR + Rotating Incast (RI)

Figure 19: Poseidon improves op latency (FCT)

(50G in aggregate) and changes its target after sending a message from each host. Poseidon achieves 56% faster median and 41× lower 99.9th op latency for UR traffic than Swift (PLB [44] enabled), 51% faster median and 6.25× lower 99.9p op latency than HPCC in Figure 19(b). Besides being robust to reverse-path congestion, Poseidon allows UR flows to ramp up faster than Swift when the rotating incast targets another victim.

5.5 Brownfield Evaluation

With brownfield deployments, Poseidon achieves partial performance gains over Swift. We repeat the UR scenario explained in §5.4 over 4 racks. There are 24 hosts in each rack connected through 6 hops to hosts in other racks. Only ToRs have 3:1 oversubscription. Figure 20(a) compares the op latency of Poseidon with INT at all switches vs. Poseidon in the brownfield where only 2 or 4 ToRs support INT. It shows that Poseidon can achieve most of the gains compared to Swift in the brownfield.

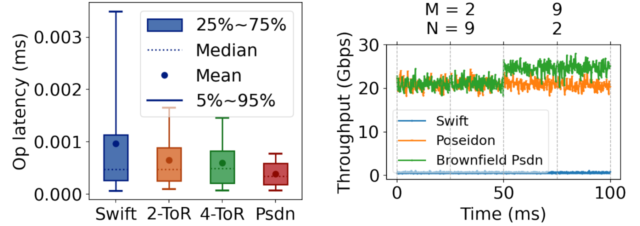
To evaluate the fairness scenarios in §4.1, we use the topology in Figure 14(a) and enable INT only on Rack 10. We make sure all blue flows send to the same host and all green flows send to another to create congestion on multiple hops for them. In Figure 20(b), when $M = 2, N = 9$, the bottleneck of red and blue flows is on Rack 10 that has INT, thus victim and other blue flows reach the fair rate (20 Gbps). When $M = 9, N = 2$, the bottleneck of red and green flows is on Rack 0, which doesn't have INT, and brownfield Poseidon gives a little more bandwidth to the red flow because the green flows have to react to the end-to-end delay. The unfairness is bounded because green flows increase their target until it covers the sum of delay on their congested hops.

5.6 Sensitivity Analysis

5.6.1 Ablation Study

To show the importance of each major aspect of the Poseidon design, we use the same algorithm and parameters as Poseidon, but remove one design aspect each time: 1) **maximum per-hop delay (MPD)** information instead of RTT; 2) **rate-adjust** target for per-hop delay; and 3) **adaptive increase ratio** algorithm instead of AIMD.

Figure 21(a) compares the throughput in the multi-hop scenario (Figure 14(a)). It shows that network-wide fairness is only achieved when using *both* rate-based target scaling



(a) Message op latency shrinks when more switches are equipped with INT capability.

(b) Unfairness is bounded when the bottleneck is not on the INT-capable switches.

Figure 20: Partial gains and fairness in brownfield Poseidon

and max hop delay. However, removing the adaptive update ratio will not harm the max-min fairness, and Poseidon can achieve fairness even using AIMD. Figure 17 has already shown that AIMD slows down ramp-up and causes wider throughput variations in the presence of many flows.

5.6.2 Robustness of Parameters

Though Poseidon could achieve the design targets with a wide range of functions and parameters, it is worthwhile to understand the trade-off of each parameter. In this section, we vary the three parameters in Eq. 3 and Eq. 4 and show why we choose: $p = 40$, $k = 2$, and $m = 0.25$.

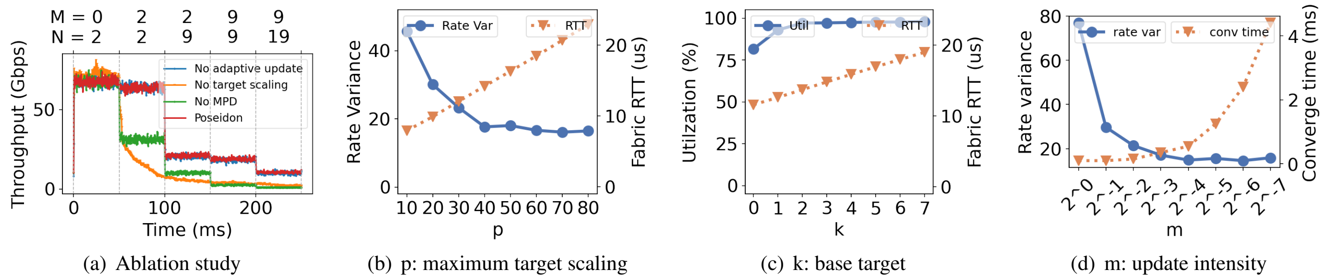
p controls the range of target scaling, affecting round-trip time and rate variation. Figure 21(b) shows that when we have congestion from hundreds of flows, a higher p allows reacting to rate unfairness faster and reduces rate variations. However, that means enduring larger RTT in the network.

k avoids under-utilizing the link bandwidth. Figure 21(c) shows the utilization % when we have a few flows on the bottleneck (where the rate is close to max_rate thus the target is close to k) vs. the fabric RTT when we have hundreds of flows. For small k values, the fluctuation of the queuing delay may lead to link under-utilization as the target is low and flows reduce the congestion window conservatively. However, if the value of k is too large, the RTT will increase.

m determines the trade-off between the variance of flow rates and the convergence speed. Figure 21(d) compares the convergence time in Figure 17(a) experiment and rate variation in Figure 17(b) for different values of m. Larger m values improve stability as they dampen the effect of the target in Eq. 4 but also slow down convergence.

6 Related Work

Delay-based: Swift [35], the basis of Poseidon, is a state-of-the-art delay-based algorithm that relies on hardware timestamps from NICs. Swift has some elements of Poseidon, although for different purposes. 1) It separates fabric delay from engine delay and tracks a separate congestion window for each. However, this separation was because of fundamental differences in congestion at fabric hops vs. hosts, and doesn't address fabric issues explained in §2.1. 2) Swift uses a larger target delay for flows with smaller congestion windows to address synchronized packet arrival from many flows on a



(a) Ablation study (b) p: maximum target scaling (c) k: base target (d) m: update intensity

Figure 21: Ablation study in multi-hop congestion scenario and sensitivity analysis over three parameters.

bottleneck link (flow scaling for windows <10), which forces flows to converge to the same window. Appendix §C shows even if we combine flow-scaling with max-hop latency, Swift still faces unfairness. 3) For multiplicative decrease, Swift decrease depending on how far the delay is from the (almost fixed) target, but still uses a fixed step for additive increase. At a higher level, Swift only looks at the end-to-end delay while Poseidon uses max-per hop latency and rate-based target scaling to respond to the bottleneck hop. Copa [12] adjusts the target rate based on *end-to-end delay* to achieve short queues and fair allocation, but Poseidon compares the max-hop delay to a rate-adjust target to reach *network-wide max-min fairness*.

ECN-based: ECN can be seen as a one-bit INT signal from switches that is set based on a configurable threshold inside switches. It is successfully deployed in datacenters and used by end-to-end CC algorithms such as DCTCP [8] because ECN was non-intrusive (two bits in the IP header) and those algorithms were deployable in brownfield environments. Still, ECN-based algorithms do not recognize the bottleneck hop and *all* flows react to any hop in their path that marks packets.

Richer signals from switches: XCP [34] and RCP [22] get help from switches to enable flows to react to congestion and achieve the fair-share allocation. In particular, XCP introduced the idea of decoupling utilization from fairness. However, both proved difficult to deploy in datacenters because of the lack of a brownfield solution and the overhead in high-speed switches. Poseidon achieves fairness using target scaling and introduces adaptive update ratios to reach better stability. It is deployable in brownfield and requires minimal changes in hardware to support max-hop delay in INT.

HPCC [40] uses queue length, timestamp, and tx-bytes of each hop to estimate in-flight bytes on each link and update a congestion window in an AIMD fashion in order to achieve very low queuing in the network. However, HPCC doesn't recognize the bottleneck hop: high utilization on *any* hop along the path should not force a flow that didn't get the link's *fair-share* to reduce rate. In addition, HPCC assumes all flows experience the same base RTT, relies on the additive increase to achieve fairness, doesn't address brownfield deployment, and requires bandwidth and CPU overhead from three INT metadata *per hop*. In contrast, by using a novel target-scaling solution, Poseidon achieves fairness without relying on AIMD, supports brownfield deployment, and only needs *a single max hop delay per packet*. PowerTCP [7] argues that CC should react to both absolute CC signal and its change rate to avoid slow reaction or overreaction to queue build-up. Poseidon's adaptive update ratio in Figure 7(a) addresses this issue. PowerTCP similar to HPCC still looks at congestion at *any* hop and uses per-hop INT metadata.

Receiver-driven: NDP [29] and HOMA [42] face challenges in oversubscribed networks where may have congestion in the core. However, Poseidon is insensitive to over-subscription, and we expect similar gains on op latency by applying Poseidon's idea to receiver-driven schemes.

Combined with schedulers: HOMA [42] combines a CC algorithm with a scheduling policy that prioritizes the shortest remaining flows to achieve shorter flow completion time. While Poseidon is currently a pure CC algorithm, we believe it has the potential to be integrated with similar scheduling policies and preserve the benefits of fast convergence and robust performance.

7 Conclusion

We proposed *Poseidon*, a congestion control algorithm that reduces op latency through fast convergence and lower latency and is robust in multi-hop and reverse-path congestion by leveraging in-band network telemetry (INT) in a novel way. Poseidon only needs a single max-hop delay per packet from INT, which makes it easily deployable with low overhead. We showed how INT packets can be deployed in brownfield and how Poseidon can still gain from an incremental deployment. In the future, we plan to implement Poseidon in NIC offloading protocols (e.g., RDMA), leverage INT to break down the delay at end-host networking stacks, use INT to hint path changes to avoid hash collisions [33], and apply the target scaling idea to other congestion signals, such as in-flight bytes [40], to achieve lower in-network delay.

This work doesn't raise any ethical issues.

Acknowledgment

We would like to thank our shepherd Paolo Costa and the anonymous NSDI reviewers for providing valuable feedback. We thank the production, serving, and support teams at Google for their contributions to the work and the platform. T. S. Eugene Ng is partially supported by the NSF under CNS-2214272 and CNS-1815525.

References

[1] How Distributed Shuffle improves scalability and performance in Cloud Dataflow pipelines, 2018. https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines.

[2] In-band Network Telemetry (INT) Dataplane Specification, 2020. https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[3] Amazon EC2: Linux accelerated computing instances: Networking performance, 2021. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/accelerated-computing-instances.html#gpu-network-performance.

[4] Encryption in Transit in Google Cloud , 2021. https://cloud.google.com/security/encryption-in-transit.

[5] Tomahawk4 / bcm56990 series, 2021. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series.

[6] PSP Architecture Specification, 2022. https://github.com/google/psp.

[7] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *NSDI*, 2022.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[9] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[10] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *NSDI*, 2020.

[11] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. NanoTransport: A low-latency, programmable transport layer for NICs. In *SOSR*, 2021.

[12] Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based congestion control for the internet. In *NSDI*, 2018.

[13] Sanjeewa Athuraliya, Victor H Li, Steven H Low, and Qinghe Yin. REM: Active queue management. In *Teletraffic Science and Engineering*, volume 4, pages 817–828. 2001.

[14] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[15] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *SIGCOMM*, 2020.

[16] Dimitri Bertsekas and Robert Gallager. *Data networks*. Athena Scientific, 2021.

[17] Neal Cardwell, Yuchung Cheng, et al. BBR Update:1: BBR.Swift; 2: Scalable Loss Handling. IETF 109. https://datatracker.ietf.org/meeting/109/materials/slides-109-iccrg-update-on-bbrv2-00, Nov 2020.

[18] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[19] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[20] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.

[22] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, 2006.

[23] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI*, 2016.

[24] Vishal Fadia and Philip Wells. Turbo boost your compute engine workloads with new 100 gbps networking, 2021. https://cloud.google.com/blog/products/networking/increasing-bandwidth-to-c2-and-n2-vms.

[25] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[26] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *NSDI*, 2022.

[27] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. Backpressure flow control. In *NSDI*, 2022.

[28] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[29] 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 *SIGCOMM*, 2017.

[30] Christian Hopps. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November, 2000.

[31] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient remote storage access with i10. In *NSDI*, 2020.

[32] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329, 1988.

[33] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. FlowBender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *CoNEXT*, 2014.

[34] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, 2002.

[35] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Mike Ryan, David J. Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[36] J. Kumar, S. Anubolu, J. Lemon, R. Manur, H. Holbrook, A. Ghanwani, D. Cai, H. ou, and Y. Li X. Wang. Inband flow analyzer, 2021. https://datatracker.ietf.org/doc/html/draft-kumar-ippm-ifa.

[37] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *NSDI*, 2021.

[38] Jean-Yves Le Boudec. Rate adaptation, congestion control and fairness: A tutorial. *on line*, 2000.

[39] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *OSDI*, 2020.

[40] 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 *SIGCOMM*. ACM, 2019.

[41] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[43] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[44] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. PLB: Congestion signals are simple and effective for network load balancing. In *SIGCOMM*, 2022.

[45] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *NSDI*, 2021.

[46] 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 *SIGCOMM*, 2015.

[47] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *NSDI*, 2019.

[48] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *EuroSys*, 2014.

A Poseidon Algorithm

Algorithms 2 shows the *RetransmitTimeout* and *FastRecovery* functions called in Algorithm 1 for completeness.

Algorithm 2: Poseidon's CWND Update Algorithms

```
1  Function RetransmitTimeout():
2      retransmit_count ← retransmit_count + 1
3      if retransmit_count ≥
        RETX_RESET_THRESHOLD then
4          cwnd′ ← min_cwnd
5      else
6          if now − t_last_decrease > rtt then
7              cwnd′ ← cwnd ∗ min_md
8      return cwnd′
9  Function FastRecovery():
10     retransmit_count ← 0
11     if now − t_last_decrease > rtt then
12         cwnd′ ← cwnd ∗ min_md
13     return cwnd′
```

B A Valid Cluster of Functions

We prove that the cluster of functions in Eq. 3 and Eq. 4 satisfy Eq. 1 and Eq. 2.

The target functions are:

$$T(x) = p * \frac{ln(max_rate) - ln(x)}{ln(max_rate) - ln(min_rate)} + k \quad (6)$$
$$min_rate < x < max_rate, p > 0, k > 0$$

Then we give a cluster of update functions, which is specifically designed for the above target functions:

$$U(x,D) = \exp\left[\frac{T(x) - D}{p} \cdot \alpha \cdot m\right] \quad (7)$$
$$\text{where } \alpha = ln(max_rate) - ln(min_rate)$$

B.1 Proof for Target Functions

When delay $D \le T(x)$:

$$Update(x,D) = \exp\left[\frac{T(x) - D}{p} \cdot \alpha \cdot m\right] \ge 1 \quad (8)$$

When delay $D > T(x)$:

$$Update(x,D) = \exp\left[\frac{T(x) - D}{p} \cdot \alpha \cdot m\right] < 1 \quad (9)$$

Thus, Eq. 3 satisfies Eq. 1.

B.2 Proof for Update Functions

Without loss of generality, assume two flows' rates $a < b$, delay is D.

For the rhs, since T(a) > T(b):

$$\begin{aligned}\frac{U(b,D)}{U(a,D)} &= \frac{\exp\left[\frac{T(b)-D}{p} \cdot \alpha \cdot m\right]}{\exp\left[\frac{T(a)-D}{p} \cdot \alpha \cdot m\right]} \\ &= \exp\left[\frac{T(b) - T(a)}{p} \cdot \alpha \cdot m\right] \\ &< 1\end{aligned} \quad (10)$$

For the lhs:

$$\begin{aligned}\frac{U(b,D)}{U(a,D)} &= \frac{\exp\left[\frac{T(b)-D}{p} \cdot \alpha \cdot m\right]}{\exp\left[\frac{T(a)-D}{p} \cdot \alpha \cdot m\right]} \\ &= \exp\left[\frac{T(b) - T(a)}{p} \cdot \alpha \cdot m\right] \\ &= \exp\left[p \cdot \frac{ln(a) - ln(b)}{\alpha} \cdot \frac{1}{p} \cdot \alpha \cdot m\right] \\ &= \exp\left[m \cdot (ln(a) - ln(b))\right] \\ &= \exp\left[m \cdot ln(\frac{a}{b})\right] \\ &= \left(\frac{a}{b}\right)^m\end{aligned} \quad (11)$$

So as long as $m < 2$, we can have

$$\frac{U(b,D)}{U(a,D)} = \left(\frac{a}{b}\right)^m > \frac{a^2}{b^2} \quad (12)$$

Thus, Eq. 4 satisfies Eq. 2.

B.3 Updating Based on Ratio vs. Distance

A valid update function with the same target function as in Eq. 3 is to use the ratio of target and max-hop delay. This can be seen as an extension of the Swift's MD function.

$$U(T(rate), delay) = \frac{T(rate) + m}{delay + m}, m \ge 0 \quad (13)$$

A problem with Eq. 13 is that it scales its update ratio depending on the value of delay. As an example, suppose that m is negligible, and we went 1 μs above the target. If the target is 4, the update ratio will be 0.8, but if the target is 30 (high concurrency scenario), the update ratio will be 0.968.

This means that for high concurrency scenarios where fair-share rate is low, and the target is high, the convergence will be slow. For example, Figure 22 compares the op latency in UR+RI scenario introduced in §5.4 for the update function based on the distance in Eq. 4 vs. the function based on the ratio in Eq. 13. The update function based on the distance clearly has an advantage at the tail.

C Flow Scaling in Swift

Swift uses flow scaling to inflate target delay to compensate for synchronized packet arrivals. The authors in [35] noticed

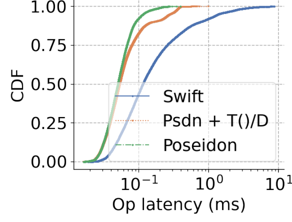
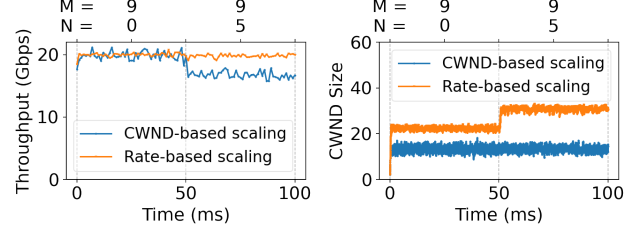


Figure 22: Poseidon can achieve lower op latency using the update function based on the distance of targe and max-hop-delay



(a) Rate of the victim flow (b) CWND of the victim flow

Figure 23: Compare scaling the target using the rate or congestion window in Poseidon.

that the average queue length grows as $O(\sqrt{N})$ where N is the number of flows on a link. Swift adjusts the target in proportional to $1/\sqrt{cwnd}$ because it argues that the *cwnd* trend is inversely proportional to the number of flows when Swift converged to its fair-share. The flow scaling in Swift also helps fairness as it speeds slow flows with a larger target, and slows fast flows with a smaller target. However, the flow scaling of Swift is not applicable to max-hop delay to find the bottleneck hop because of two reasons: 1) Its effect is nominal for windows > 10 (See Figure 5 in [35]) and more importantly 2) The formulation that reached $1/\sqrt{cwnd}$ assumes, at fair-share, flows see the same RTT (≈target delay) and pushes them to have the same *cwnd*. However, to solve the scenarios in §2.1 and get the fair-share, we only need to react to the congestion at the bottleneck hop. This means that flows get different RTTs and pushing flows to get the same *cwnd* cannot achieve the fair share rate ($rate = \frac{cwnd}{RTT}$). For example, in the fair-share allocation of multi-hop congestion scenario, the victim flow will have higher RTT and needs higher *cwnd* to achieve the fair-share.

We show this shortcoming in the following equations. Suppose that the link capacity is C, and the congestion window and RTT for flow i are $cwnd_i$ and RTT_i. The target delay is calculated as follows, where t_{base} is the base delay in Swift and A is just a constant.

$$t = t_{base} + A \cdot \frac{\sqrt{N}}{C} \tag{14}$$

$$\text{Fair share for flow i} = \frac{cwnd_i}{RTT_i} = \frac{C}{N} \tag{15}$$

If we **assume** that at the steady state RTT_i is equal to t (target delay) for all flows and thus *cwnds* are equal to w in order to get the same throughput, we can estimate $\sqrt{N}$ from Eq. 15 as follows

$$\sqrt{N} = \frac{A + \sqrt{A^2 + 4 \cdot C \cdot w \cdot t_{base}}}{2 \cdot w} \tag{16}$$

Therefore, $\sqrt{N}$ can be estimated by $\frac{\alpha}{\sqrt{w}} + \beta$.

However, as explained above, to achieve fair-share rate, flows get different RTTs thus converging to the same congestion window is not fair. Now, we show that if we follow the formulation of how Swift reached $1/\sqrt{cwnd}$ for cases that flows have different RTTs at fair share, we end up with a valid Poseidon rate-based scaling. We repeat Eq. 14 here as Eq. 17 after changing t to t_{hop} to emphasize that for Poseidon we have a target for per-hop delay.

$$t_{hop} = t_{hop_base} + A \cdot \frac{\sqrt{N}}{C} \tag{17}$$

If we combine Eq. 17 and Eq. 15 we get

$$t_{hop} = t_{hop_base} + \frac{A}{\sqrt{C}} \cdot \sqrt{\frac{RTT_i}{cwnd_i}} = \frac{\alpha}{\sqrt{\frac{cwnd_i}{RTT_i}}} + \beta \tag{18}$$

$\frac{cwnd_i}{RTT_i}$ in Eq. 18 is the rate of flow i. Therefore, for the flow scaling of Swift to work in a fair-share setting where flows can have different RTTs, the target should increase in reverse relation to rate not just *cwnd*. Figure 23 compares the throughput and congestion window of the victim flow in Poseidon if it uses the above target function using rate vs *cwnd* in the multi-hop congestion scenario (Figure 14(a)). The victim and $N = 9$ flows start at time 0. Then at 50 ms, $M = 5$ flows start to create congestion at the source rack. Figure 23(b) shows that after 50ms, target scaling based on the rate converged to a higher *cwnd* to keep the throughput the same for the victim flow.

Eq. 18 is a special case of $T(b) = p \cdot b^q + k$, a valid cluster of functions that satisfy Eq. 1 and Eq. 2 for $-2 \leq q < 0$, with $q = -0.5$. However, we believe Poseidon's function in Eq. 3 is a better function as explained in §3.3.

D Proof of Lemma 1

We first repeat the Lemma here: When achieving network-wide max-min fairness, each flow will have the largest rate among all flows on its bottleneck hop and not on any other saturated hop. Formally, for the "max-min fair" allocation $\vec{x}$, for any flow s, denote the flows shared the same bottleneck with s as $\{b_1, b_2, ..., b_k\}$. For any flow b_i, $x_s \geq x_{b_i}$. Denote the flow's share on the saturated non-bottleneck hop of s as $\{c_1, c_2, ..., c_k\}$, then there must exist some c_j such that $x_{c_j} > x_s$.

Proof: Assume there exists a flow s that has reached its fair-share rate r, and there is another flow s' on its bottleneck hop with an even larger rate r'. But this state is not max-min fair

because flow s could get some bandwidth from flow s' and let them have the same rate $\frac{r+r'}{2}$. By contradiction, the flow s has the largest rate on its bottleneck hop.

On the other hand, assume there exists a flow s, which is the fastest flow, with rate r, on one of the non-bottleneck hops. However, given that this link is congested, its fair-share flows with rate r' could obtain bandwidth from flow s and increase their fair-share rate to at least $\frac{r+n\cdot r'}{n+1}$, where n is the number of fair-share flows. By contradiction, the flow s cannot be the largest flow on its non-bottleneck hop.

E Proof of Convergence to Max-min Fairness

Problem description:

For any network topology, any traffic pattern (flows' source and destination, routing), given an initial flow rate allocation, Poseidon converges to the max-min fair allocation.

Notations:

Denote all the link bandwidth as B;

Denote the target for flow with rate r as $T(r)$;

Denote a flow rate allocation as $\vec{y}$;

In allocation $\vec{y}$, denote the rate of flow f as y_f;

In allocation $\vec{y}$, denote the maximum flow rate on a saturated queue q as R^y_q;

In allocation $\vec{y}$, denote the delay on a queue q as D^y_q;

Denote the max-min fair rate allocation as $\vec{x}$;

In max-min fair allocation $\vec{x}$, denote the maximum flow rate on a saturated queue q as R^x_q, which is also the fair-share rate of that port.

Designs of Poseidon and observations:

Design 1: Poseidon reacts to the maximum hop delay along the path.

Design 2: In Poseidon, the target of a flow increases when the flow rate decreases. And Poseidon decreases the flow rate when the delay is higher than the flow's target; increases the rate when the delay is lower than the target.

Observation 1: The queuing delay on a saturated port is no larger than the target of flows with the fastest rate on that port.

Observation 1 holds true because of design 2: if the delay exceeds the target of a flow, that flow will decrease its rate immediately. However, in Poseidon, the decrease operation only happens once per RTT, so the reaction of decreasing rate may happen at most one RTT later. But this will not affect the overall trend of queuing.

Observation 2: the queuing delay on an unsaturated port is always 0.

Observation 2 holds true when senders send packets without bursts. However, the synchronized arrival of many flows may create a transient queue. But the queue will disappear within 1 RTT, because the average data sent within one RTT is less than the line rate.

Proof:

We will use induction to prove any allocation $\vec{y}$ will converge to max-min fair allocation $\vec{x}$.

In allocation $\vec{x}$, if we sort the saturated queues based on their maximum flow rate (fair-share rate), we can get:

$$R^x_{q_1} \le R^x_{q_2} \le ... \le R^x_{q_k} \tag{19}$$

$$T(R^x_{q_1}) \le T(R^x_{q_2}) \le ... \le T(R^x_{q_k}) \tag{20}$$

(1) prove queue q_1 will converge to the max-min fair allocation:

For any allocation $\vec{y}$, for queue q_1, its fastest flow's rate is $R^y_{q_1}$. Note that this q_1 is still the same q_1 sorted by allocation $\vec{x}$.

Because the queue q_1 is saturated in $\vec{y}$, it has to satisfy:

$$\sum_{f\in Flows(q_1)} y_f \ge B \tag{21}$$

And we already know:

$$\sum_{f\in Flows(q_1)} x_f = B \tag{22}$$

Because in allocation $\vec{x}$, all the flows on queue q_1 has the same rate, which is $R^x_{q_1}$. Any other allocation $\vec{y}$'s largest rate cannot be as small as $R^x_{q_1}$ because their rates are not all equal, so we have:

$$R^y_{q_1} \ge R^x_{q_1} \tag{23}$$

$$D^y_{q_1} = T(R^y_{q_1}) \le T(R^x_{q_1}) = D^x_{q_1} \tag{24}$$

Because of the same reason, we also have:

$$D^y_{q_i} = T(R^y_{q_i}) \le T(R^x_{q_i}) = D^x_{q_i}, \forall i \in [2,k] \tag{25}$$

So for flows whose rates are smaller than $R^x_{q_1}$, their target is higher than delay on queue q_1 and also delay on any other queue q_i:

$$T(y_f) > T(R^x_{q_1}) \ge R^y_{q_1}, \forall y_f < R^x_{q_1} \tag{26}$$

$$T(y_f) > T(R^x_{q_1}) \ge T(R^x_{q_i}) \ge R^y_{q_i}, \forall y_f < R^x_{q_1}, \forall i \in [2,k] \tag{27}$$

Thus, those flows with smaller rate will keep increasing and flows with larger rate than $R^x_{q_1}$ will decrease because of the delay on queue q_1 or on other queues. Eventually, all of them will converge to the same target:

$$T(y_f) = T(R^x_{q_1}), \forall f \in q1 \tag{28}$$

So that:

$$T(R^y_{q_1}) = T(R^x_{q_1}) \tag{29}$$

Thus, we show that the queue q_1 will converge to the max-min fair allocation x

(2) Assume queue q_1 to q_m have already converged, prove queue q_{m+1} will converge:

Assume queue q_1 to q_m have already converged to max-min fair allocation $\vec{x}$, so we have:

$$T(R^y_{q_i}) = T(R^x_{q_i}), \forall i \in [1,m] \tag{30}$$

For queue q_{m+1}, the flows whose bottleneck is q_{m+1} will not travel queue q_1 to q_m. Because if they travel to one of those ports, those ports will have a higher fair-share rate, which contradicts the max-min fair allocation's conclusion.

Thus, with a similar analysis as the proof for step 1, we have:

$$T(y_f) > T(R^x_{q_{m+1}}) \geq R^y_{q_{m+1}}, \forall y_f < R^x_{q_{m+1}} \tag{31}$$

$$T(y_f) > T(R^x_{q_1}) \geq T(R^x_{q_i}) \geq R^y_{q_i}, \forall y_f < R^x_{q_1}, \forall i \in [m+2,k] \tag{32}$$

So that, the flows with smaller rate than $R^x_{q_{m+1}}$ will increase their rate, while flows with larger rate will decrease their rate, until:

$$T(y_f) = T(R^x_{q_{m+1}}), \forall f \in q1 \tag{33}$$

So that:

$$T(R^y_{q_{m+1}}) = T(R^x_{q_{m+1}}) \tag{34}$$

So we proved that queue q_{m+1} will also converge to max-min fair allocation $\vec{x}$.

In conclusion, all the ports in allocation $\vec{y}$ will eventually converge to the max-min fair allocation $\vec{x}$.