Network-Assisted Congestion Feedback

Seifeddine Fathalli, Emilia N. Weyulu, Danesh Zeynali, Balakrishnan Chandrasekaran, *Member, IEEE*, and Anja Feldmann

***Abstract*—We present Network Congestion Feedback (NCF), a novel congestion control framework that leverages programmable data planes for generating a rich congestion signal for use in the public Internet. `NCF` makes several contributions, including isolating 'mice' and 'elephant' flows using separate queues, detecting congestion in the elephants' queue and generating a rich sub-RTT signal for the concerned senders, and designing a congestion-control algorithm (CCA) that matches a flow's demands with supply (i.e., available bandwidth) for maximizing utilization and fairness. It extends two key ingredients from prior work on datacenter CCAs—a short control-loop delay and a precise congestion signal—that are crucial for designing an efficient, fair CCA, by adapting them for the more challenging Internet context. `NCF` isolates mice and elephant flows so that the former cannot unfairly degrade the throughput of the latter, and it guarantees that mice flows experience minimal round-trip times (RTTs) even when contending with elephant flows. `NCF` virtually eliminates slow-start spikes and achieves high fairness in both shallow and deep-buffer configurations, and even when the flows experience drastically different RTTs. Lastly, `NCF` offers low flow completion times (FCTs) to short flows even in challenging multiple-bottleneck scenarios.**

***Index Terms*—Congestion control, flow control, data plane programmability, P4.**

I. Introduction

A FUNDAMENTAL and long-standing challenge in networking is effective mitigation of congestion on a bottleneck link. There exists, unsurprisingly, an extensive body of prior work on congestion control algorithms (CCAs) spanning about four decades. The network landscape, however, evolves continually (examples include programmable data planes, low-Earth-orbit satellite networks, 5G and WiFi 7 for cellular and wireless networks, and a plethora of novel applications with varying performance demands), and this evolution, in turn, ushers in innovative CCA proposals from the networking community. The evolution in networking technologies and applications (or systems) also introduce new challenges for CCA design. Today, an "ideal" CCA must cater to a multitude of objectives including ensuring fair or equitable sharing of network resources (e.g., [1], [2], [3]), reducing delays or RTTs, maximizing throughput, minimizing FCTs, avoiding starvation, tackling TCP Incast (e.g., [4], [5], [6], [7], [8]), and adapting to diverse network settings (e.g., datacenter [9], [10], [11], [12] and wireless networks [13], [14]). In this work, we exploit the recent advancements in programmable data planes [15] to design a framework that facilitates designing a multi-objective CCA for the public Internet.

We review prior work on CCAs to identify four key challenges that any congestion control signaling or notification mechanism, specifically designed for the public Internet, should meet.

Responsiveness: The primary goal of a network congestion notification is *prompt* signaling. Quickly notifying a sender of *impending* congestion is crucial for reducing (TCP) timeouts and retransmissions (e.g., DCTCP [5]). Moreover, recent work show that a *sub-RTT* congestion feedback is essential for guaranteeing high utilization and minimizing FCTs [9].

Stability: Detailed telemetry data can curb oscillations in a sender's throughput and support consistent, low queue usage at the bottleneck. If we quickly deliver such rich telemetry data to a (traffic) sender, we can minimize the likelihood of packet drops from buffer overflows as well as network under-utilizations from queue underflows.

Protocol independence: The congestion notifications (e.g., ECN) are agnostic to the transport protocol. Any upgrade to this old scheme, incorporating rich feedback, should still strive to be independent of transport protocol. TCP and its variants, and even the modern QUIC implementations can then leverage the signaling, as required, and contribute to improvements in the performance of diverse applications using these transports.

Incremental deployment: Most prior work on innovative congestion notification mechanisms focused on data centers or enterprise networks. The public WAN or Internet, in contrast, is not within the control of a single provider or organization. The adoption of any congestion signaling mechanism must allow for incremental or selective (i.e., strategic) deployments of the solution.

We present a simple and robust congestion signaling or notification mechanism, called *network-assisted congestion feedback* (`NCF`), that addresses all these four objectives. At its core, `NCF` exploits the programmability in today's switches to distinguish long (i.e., *elephant*) flows from short (i.e., *mice*) flows (Fig. 1). We then isolate the traffic corresponding to the two flow types using the support for priority classes and queues that are available even in commodity networking

Received 20 February 2025; revised 2 September 2025 and 5 December 2025; accepted 12 December 2025. Date of publication 31 December 2025; date of current version 15 January 2026. The associate editor coordinating the review of this article and approving it for publication was G. Leduc. *(Corresponding author: Danesh Zeynali.)*

Seifeddine Fathalli, Emilia N. Weyulu, Danesh Zeynali, and Anja Feldmann are with the Internet Architecture Department, Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany, and also with the Computer Science Department, Saarland University, 66123 Saarbrücken, Germany (e-mail: fathalli@mpi-inf.mpg.de; eweyulu@mpi-inf.mpg.de; dzeynali@mpi-inf.mpg.de; anja@mpi-inf.mpg.de).

Balakrishnan Chandrasekaran is with the Computer Science Department, Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands (e-mail: b.chandrasekaran@vu.nl).

Digital Object Identifier 10.1109/TNSM.2025.3648180

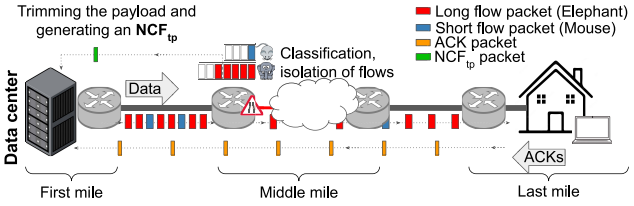


Fig. 1. Illustrating the use of NCF in congestion scenarios where a rich, sub-RTT feedback can be immensely beneficial.

hardware. When a bottleneck experiences congestion, NCF meticulously constructs a congestion feedback signal and sends it *only* to the elephant flow (senders) for throttling them. All of these tasks are performed in the data plane, at line rate, by exploiting the capabilities of a programmable switch; we refer to these data-plane implementations as NCF's data-plane logic ($\texttt{NCF}_{\texttt{dl}}$).

NCF provides a quick, i.e., sub-RTT, congestion feedback to the senders of an elephant flow. While prior work on generating explicit congestion signals focused on data centers, we exploit the availability of programmable data planes in the public WAN (e.g., in IXPs and access networks). Even if many end users experience congestion in the last mile, delivering the congestion notification from the last hop prior to the customer premise equipment (CPE) instead of the end-user devices themselves could substantially shorten the control loop delay; such a notification would, for instance, avoid the high delays that are typical in WiFi connections, the common mode of Internet connectivity for end-user devices. Furthermore when congestion occurs far away from users, closer to content providers and origin servers (e.g., [16], [17], [18]), a quick signaling could substantially shorten the otherwise high-delay round-trip-path over the public Internet.

NCF generates a rich and explicit congestion feedback in the form of NCF telemetry packets ($\texttt{NCF}_{\texttt{tp}}$). The data conveyed via $\texttt{NCF}_{\texttt{tp}}$ include link speed, queue size, and an estimate of the count of elephant flows. This rich telemetry data about the conditions at the bottleneck can be exploited by the sender to adapt the congestion window quickly and precisely, at *sub-RTT* reaction times (Fig. 1).

The design of NCF is transport-protocol agnostic. Unlike explicit congestion notifications and related work, NCF does not require receiver support; $\texttt{NCF}_{\texttt{tp}}$ packets are sent directly to the sender by the data plane. The sending mechanism can then adapt its response based on the telemetry data delivered by $\texttt{NCF}_{\texttt{tp}}$ as we demonstrate through a simple CCA scheme, which we label NCF's sender logic ($\texttt{NCF}_{\texttt{sl}}$). Our approach enables novel CCA designs for the public Internet that can offer substantial performance and fairness by using the precise, sub-RTT feedback offered by NCF, and $\texttt{NCF}_{\texttt{sl}}$ is simply one point in this solution space.

Our approach can be incrementally or strategically deployed, as required. A content provider serving a large customer base might experience congestion close to the edge of their serving infrastructure (e.g., at upstream links from the "edge" servers) [16]. To alleviate the issue, we can deploy NCF in the first mile, at data centers of content provider or in the upstream network. Similarly, an eyeball AS may deploy NCF to mitigate congestion in the last mile and facilitate fair sharing of network resources by various application traffic. We can also deploy NCF at strategic locations such as colocation facilities and Internet exchange points (IXPs), which host the points of presence (PoPs) where traffic from a content provider's infrastructure connect to various ISP networks [19], [20], [21], [22], [23], [24], [25]. Any network can deploy NCF without requiring support or coordination from another network. The deployments may also catalyze or motivate other networks to adopt NCF, as each deployment may essentially move the bottleneck elsewhere along the path traversed by application traffic to adopt NCF.

The core ideas in NCF may not be new given the extensive work on congestion control, but integrating these ideas into a cohesive end-to-end system is a non-trivial endeavor; we also tackle several challenges in optimizing the system for use in the public Internet. Our signaling mechanism may resemble the idea of source quench (SQ) [26] or choke messages [27], but it is *without* the issues (e.g., [28], [29]) that resulted in the deprecation of those old ideas [30]. SQ was weak and unfair, for instance, because it sent one (ICMP) message per congested packet without regard to flow type; besides, it carried little information about the bottleneck's characteristics. NCF instead focuses on elephant flows that can react to congestion and carries enough information concerning the bottleneck to the sender. The sender, hence, can back off immediately to its fair share; NCF, hence, obviates the need for heuristics to approximate the extent of congestion. SQ messages are also easy to filter or spoof. NCF, in contrast, generates valid, low-overhead ACK-like packets that middleboxes are unlikely to filter, and it preserves existing TCP sequence and acknowledgment numbers, which makes spoofing harder. We also allay security concerns, e.g., switch overload, by generating $\texttt{NCF}_{\texttt{tp}}$ purely in the data plane and mitigate DDoS vulnerabilities through safeguards such as nonces, verification, and damping. Likewise, in the data center context, prior work have investigated generating precise congestion feedback using programmable hardware (e.g., XCP [31], HPCC [32], FastTune [10], Bolt [9]) and quickly sending this feedback using ACKs traveling towards the sender (e.g., FastTune [10], ExpressPass [11]). While the idea of eliciting explicit network support for congestion control originated about two decades ago (e.g., [31], [33], [34]), we recently resurrected this idea, on account of recent advances in networking, in a preliminary version of this work at a workshop in 2019 [35]. Indeed the most recent work Bolt [9] followed up our call for the community to rethink explicit congestion signaling. The scope of the signaling or feedback in prior work has been, nevertheless, rather narrow (e.g., in data centers, or requiring support at every hop along the path, or with sweeping changes across the network stack), which renders the approach either insufficient or impractical for deployment at scale in the public Internet.

We summarize our key contributions as follows.

- We propose a new congestion control framework (NCF) for designing CCAs for the public Internet. NCF exploits a rich and explicit congestion signal, $\texttt{NCF}_{\texttt{tp}}$, that

we generate using widely available programmable data planes.

- We present a P4-prototype ($\texttt{NCF}_{\texttt{dl}}$) for generating $\texttt{NCF}_{\texttt{tp}}$ at line rate and a simple, robust sender logic ($\texttt{NCF}_{\texttt{sl}}$) to demonstrate how to exploit the telemetry data in $\texttt{NCF}_{\texttt{tp}}$.
- We show the benefits of separating short-lived (mice) flows from long-lived (elephant) flows using separate queues for each type, and demonstrate how this approach prevents a flood of mice from substantially degrading the throughput of elephants, while still ensuring that the mice experience minimal RTTs. We show, using different benchmarks and across diverse network conditions, that NCF virtually eliminates slow-start spikes (Section V-C), ensures high fairness while minimizing queue usage in various scenarios (Section V-D), and offers these benefits even when competing flows have dissimilar RTTs and regardless of bottleneck buffer sizes (Section V-E). Compared to three widely used CCAs, NCF offers the lowest FCTs to short flows even in challenging multi-bottleneck scenarios (Section V-F).

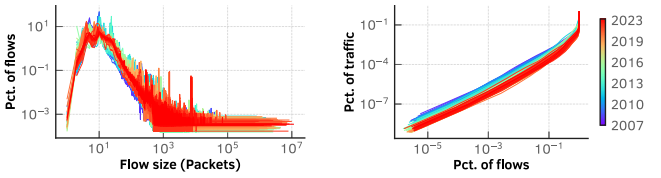
II. Motivation and Insights

Below, we discuss the key insights behind NCF's design.

A. Isolating Mice and Elephant Flows

A TCP flow starts with a "slow start" phase, during which the sender keeps doubling its congestion window (cwnd) for each RTT. This doubling of the volume of inflight data continues typically until the sender detects congestion. Slow start, historically, started with a cwnd of 1 maximum segment size (MSS); the doubling of cwnd for any flow, in other words, commenced with an initial cwnd of 1 MSS. Today, TCP implementations typically use a value of 10 MSS or higher [36], [37], and it has substantial implications for congestion control: Initial cwnd dictates the amount of data that a sender can send, when the flow starts, without waiting for any acknowledgment. Said differently, during this "ramp up" period of the slow start phase, the sending rate is not subject to any limitations, other than the initial cwnd. Efforts to congestion-control them are, hence, futile.

Most flows in the Internet are short-lived or mice flows. The sizes of flows in the MAWI data set, a collection of sampled Internet transit traffic data between WIDE and its upstream ISP [38], illustrates this observation (Fig. 2(a)). Indeed the dominance of mice flows (in terms of flow counts) in the Internet has been consistent since 2007. If the initial cwnd is 10 or higher, then per Fig. 2(a) most flows in the Internet will never exit the slow start phase. A substantial number of them will even finish within the initial ramp up period of the slow start phase: In 2023, for instance, about a third of the flows have fewer than 10 packets. A CCA scheme cannot, however, only focus on the mice flows. Most of the traffic volume is contributed by the small fraction of elephant flows (Fig. 2(b)): If we use a threshold of 12 packets to distinguish mice from elephant flows, the MAWI data set shows that 99.5% of the traffic constitutes elephant flows. Mice flows are not subject to congestion control, so they can rapidly



(a) *Flow size distribution* (b) *Flow share vs. traffic share*

Fig. 2. Per the MAWI dataset, (a) most flows in the Internet are rather short-lived, but (b) the long-lived flows contribute most towards traffic volume.

fill up a bottleneck buffer at any time [39]. Latency-sensitive mice flows are particularly vulnerable to queuing delay, which can substantially degrade end users' experiences. Besides, multiplexing elephant and mice flows in a single queue adversely affects the FCTs of both flow classes [40], [41]. A flood of mice flows—triggered, for instance, by DDoS attacks or TCP incast [7]—can occupy the shared buffer and degrade the performance of elephant flows as TCP's traditional congestion signals (queuing delay and packet loss) induce persistent backoff [42]. Conversely, mice flows can themselves suffer from head-of-line (HOL) blocking caused by elephant flows [5], [43].

Takeaways: A CCA scheme must offer low FCTs to mice and high throughput for elephant flows. Separating the two flow types using dedicated queues is crucial to satisfy these objectives, since it (a) offers short FCTs to latency-sensitive mice flows (e.g., RPCs), (b) alleviates head-of-line blocking, (c) can cope with TCP incast, and (d) mitigates the DDoS impact, which are induced typically by a "flood" of mice flows. It also crucially enables us to send congestion signal only to the elephant flows, which can respond appropriately to the signal.

B. Rich Telemetry and Sub-RTT Feedback

The idea of explicitly notifying a (TCP) sender of congestion dates back to *DECbit*, which eventually paved the way for random early detection (RED) and Explicit Congestion Notification (ECN) [44]. These approaches offered only a limited feedback: They did not convey to the senders, for instance, the degree of congestion [31]. Recent research in CCAs clearly demonstrate the need for a rich, explicit congestion signaling as well as their implications for both application performance and network utilization [9], [10], [32]. Virtually all of these prior work, however, are designed for specialized environments where we can explicitly dictate the CCAs and/or make sweeping modifications to the network stack at either endpoints; some approaches even require additional support at each hop along a network path. Such deployment-environment requirements and/or assumptions make these schemes impractical, if not infeasible, to deploy them in the public Internet.

Even if we generate a rich congestion feedback, its utility crucially depends on how quickly we can share it with a sender. Requiring support from the receiver (for instance, to reflect the congestion feedback generated by a switch or router along the path back to the sender) results in a control-loop

Algorithm 1 Flow Classification & Congestion Signaling

```
Require: Pkt on Ingress
1: ρ: Flow size threshold for separating mice and elephants
2: η: Queue congestion threshold
3: count ← GetFlowSize (Pkt)
4: if count > ρ then
5:     ElephantQueue.Enq (Pkt)
6: else
7:     MiceQueue.Enq (Pkt)
8: Process the Elephant Queue:
9: if ElephantQueue.Occupancy () > η then
10:    procedure GENERATENCF(Pkt)
11:        Mirror (Pkt)
12:        TrimPayload (Pkt)
13:        Switch.AddrPort (Pkt)
14:        AddTelemetryData (Bandwidth, Queue, Elephant-count, Pkt)
15:        Send (NCF)
16: Forward (Pkt)
```

delay. Unsurprisingly, there is prior work on reducing this control-loop delay (e.g., [9], [10], [11]). FastTune [10] and ExpressPass [11], for instance, send the congestion feedback via ACKs traveling towards the sender. In the Internet, this assumption of symmetric paths may not always hold true. We demonstrated, instead, a proof-of-concept of an approach that completely removes the dependency on receivers by exploiting programmable hardware [35]. Recent work showed the feasibility of sub-RTT feedback in a tightly controlled environment such as a datacenter [9].

Eliminating the need for support from receivers has substantial benefits especially in the public Internet. It would drastically reduce the control-loop delay, since the RTTs in the Internet are at least an order of magnitude larger than those in datacenters; even if the bottleneck is in close proximity to end users (e.g., in home WiFi networks), signaling the sender of this congestion from a switch or router deployed a hop prior to the CPE will avoid the large delays typical in the last hop (e.g., WiFi and cellular networks). Furthermore, instead of dictating the sender's behavior through the feedback signal (e.g., by specifying the delivery rates [31], [33]), allowing them to exploit the rich feedback, as appropriate, decouples the sender logic from the congestion signaling. This decoupling facilitates innovation in CCA designs to proceed without being impeded by what we can achieve with programmable data planes today.

Takeaways: An ideal congestion signaling mechanism should offer a flow sender (TCP or QUIC) insights into the congestion at the bottleneck, so that the sender can take the appropriate action. This congestion feedback should avoid receiver-side support, especially in the Internet, and be delivered as quickly as possible to the sender.

III. System Design

Below, we describe how NCF's data-plane logic ($\texttt{NCF}_{\texttt{dl}}$) distinguishes between mice and elephant flows, isolates the two flow types, and generates the telemetry packet ($\texttt{NCF}_{\texttt{tp}}$) that notifies senders of elephant flows of congestion at a bottleneck. We then briefly describe how we can rethink a CCA or retrofit the sender-side logic ($\texttt{NCF}_{\texttt{sl}}$) to benefit from the rich, sub-RTT congestion feedback offered by NCF.

A. NCF Programmable Data Plane

In NCF, we exploit programmable switches to continuously track flow sizes in the data plane for distinguishing between mice and elephant flows. After classifying the flows, we isolate them using a separate queue for each flow type. Lastly, when experiencing congestion, we generate a telemetry packet, $\texttt{NCF}_{\texttt{tp}}$, essentially a rich congestion signal, and send it directly to the senders of the elephant flows. All of these computations are performed strictly in the data plane, and we refer to them together as the NCF programmable data plane ($\texttt{NCF}_{\texttt{dl}}$).

① *Tracking flow sizes:* Recent advances in programmable data planes (e.g., P4 Tofino switches), allow us to track flow sizes at line rate in the data plane and use them for identifying mice and elephant flow types [45]. We classify all flows initially as mice, but they can evolve to be classified as elephants, if they continue to persist and contribute more traffic. To track the sizes of flows, we use "rolling" counters. Specifically, we program the switch to maintain a packet counter, $c_{f_i}^t$, per flow over n time bins of duration d. The n bins allow us to track the sizes of flows for a duration of $d \times n$. At time t_0, we use the 0^{th} bin for counting packets. The flow counter $c_{f_i}^0$ captures, hence, the packets arriving within the time interval $[\lfloor t/d \rfloor * d, (\lfloor t/d \rfloor + 1) * d]$. More generally, the flow counters $c_{f_i}^j$ capture packets arriving within the time window $[(\lfloor t/d \rfloor - j) * d, (\lfloor t/d \rfloor - j + 1) * d]$. When time t crosses an interval boundary, the current flow counters are moved from index 0 to 1, and the counting resumes on newly initialized counters.

One alternative for overcoming the limited memory scalability of per-flow counters is a *sketch*, a probabilistic data structure that allows us to trade off accuracy for memory footprint. The Count-Min sketch, for instance, requires only a small, fixed memory budget regardless of the number of flows tracked by the data structure [46]; the sizes of flows reported by the sketch is, however, an approximation of the true values. Prior work have shown the feasibility of implementing sketches in the data plane, although they are not strictly in-line, e.g., [47], [48], [49], [50], [51]: They use many P4 pipeline stages or even require control plane support. To simplify the implementation and keep assumptions about hardware capabilities minimal, we use rolling counters instead of sketches in NCF. We can easily update the $\texttt{NCF}_{\texttt{dl}}$ (for instance, replacing counters with a sketch optimized for the specific hardware), independent of other components, since the $\texttt{NCF}_{\texttt{dl}}$ and rich telemetry signal that we generate using it are decoupled from the sender-side logic of NCF.

② *Distinguishing between mice and elephants:* Upon receiving a packet, $\texttt{NCF}_{\texttt{dl}}$ identifies to which (currently tracked) flow, if any, the packet belongs and retrieves the associated flow counter from the registers (see line 3 of Algorithm 1). We compare the current counter value against a predefined threshold (ρ) to determine whether the flow associated with this packet can be classified as a mouse or elephant (lines 4-7 of Algorithm 1). NCF uses two queues to separate the packets belonging to the two different flow types (unlike our earlier work [35]), and both flow types have the same priority level for their respective queues in $\texttt{NCF}_{\texttt{dl}}$. This design with just two queues offers a baseline

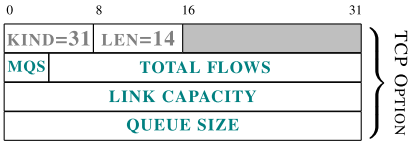


Fig. 3. NCF telemetry packet header.

configuration, and it allows a more complex implementation of packet-handling strategies commonly associated with multiple-queue setups in network switches. We ensure a fair distribution of traffic transmission from the two queues by utilizing the switch's round-robin (RR) traffic scheduler. This scheduler serves packets from both queues in a round-robin fashion, adhering to a predefined split ratio of β.

To detect congestion quickly and deliver timely notifications to senders, $\texttt{NCF}_{\texttt{dl}}$ adopts the three-color marking feature supported by the switch [52]. Color-aware queue management assigns packets a drop precedence—red, yellow, and green. Red denotes incipient congestion, yellow indicates that the switch will generate an NCF and forward the packet, and green denotes that the packet will be dropped. Each queue has per-color thresholds (based on queue size) that determine when packets are marked or dropped. Essentially, we assign two congestion thresholds (red), μ for mice, and η for elephants. We opt for stricter thresholds for elephant flows than for mice flows, since it allows us to trigger congestion notifications (to senders of elephant flows) as early as feasible. We also monitor the mice-queue occupancy to prompt a back-off of the elephant flows, if required, e.g., when the mice flows require some headroom to drain queued packets.

③ ***Generating telemetry packet:*** Before forwarding an elephant packet, $\texttt{NCF}_{\texttt{dl}}$ checks the occupancy (or congestion status) of the elephant queue. If the elephant queue's occupancy exceeds the predefined threshold, $\texttt{NCF}_{\texttt{dl}}$ gathers the relevant telemetry data and generates a $\texttt{NCF}_{\texttt{tp}}$ *prior* to forwarding the original packet (lines 9-15 of Algorithm 1). Concretely, to signal the sender of an elephant flow of congestion, $\texttt{NCF}_{\texttt{dl}}$ generates the $\texttt{NCF}_{\texttt{tp}}$ data by (a) replicating a packet of this flow, (b) perform payload trimming (which extends a similar idea used in a controlled datacenter environment [53], [54]), (c) append crucial details relevant to the congestion at the bottleneck, and (d) send it directly to the sender.

Fig. 3 shows the telemetry data that $\texttt{NCF}_{\texttt{tp}}$ shares with the senders (of elephant flows). $\texttt{NCF}_{\texttt{tp}}$ includes a 1-bit flag indicating the congestion status of the mice queue (MQS), the count of active elephant flows, the capacity of the egress link, and the elephant queue occupancy. The MQS field helps the (elephant-flow) senders to take the appropriate action when the mice queue is full, which may involve, for instance, backing off to help the FCTs of mice flows. Other telemetry data such as the link capacity and queue size enables the senders to estimate the bandwidth-delay product (BDP) precisely. The senders can then determine the optimal sending rates using the BDP estimate and count of flows ("Total Flows" in Fig. 3) at the bottleneck. $\texttt{NCF}_{\texttt{dl}}$ also detects when an elephant flow has ended to update the count of (active) elephant flows. $\texttt{NCF}_{\texttt{dl}}$ generates $\texttt{NCF}_{\texttt{tp}}$ entirely in the data plane and delivers it directly to the sender, eliminating the need for receiver-side support. We can ensure that the generation and delivery of $\texttt{NCF}_{\texttt{tp}}$ data to senders does not introduce any security vulnerability by rate limiting the congestion notifications using a static limit or based on the observed RTT between a sender and the $\texttt{NCF}_{\texttt{dl}}$ (since that determines how quickly a sender can respond and how many $\texttt{NCF}_{\texttt{tp}}$ it requires to converge to the available capacity at the bottleneck).

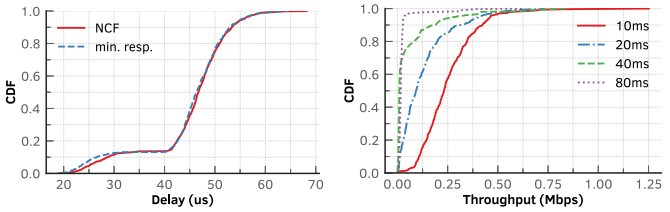
Algorithm 2 Sender Logic or CCA

1: *in* $\texttt{NCF}_{\texttt{tp}}$ *packet:*
2: *queue-size*: Switch queue size
3: *N*: Number of flows
4: *is-mice-congested*: Congestion state of mice queue
5: *BW*: Bottleneck bandwidth
6: *ssthresh*: Congestion slow start threshold
7: *Cwnd*: Congestion window
8: *BDP*: Bandwidth delay product
9: *sRTT*: Smoothed RTT
10:
11: $\texttt{NCF}_{\texttt{tp}}$ received:
12:
13: **if** *change-in-flow-count* $> \tau$ OR *is-expired(last-*$\texttt{NCF}_{\texttt{tp}}$*)* **then**
14: $BDP \leftarrow BW \cdot sRTT$
15: $bdp\text{-}factor \leftarrow BDP \ / \ queue\text{-}size$
16: **if** $bdp\text{-}factor > 1.2$ **then** ▷ To avoid bursts that queue cannot absorb
17: $bdp\text{-}factor \leftarrow bdp\text{-}factor \cdot \delta$
18: $ssthresh_{new} \leftarrow bdp\text{-}factor \cdot queue\text{-}size/N$
19: **if** *is-mice-congested* **then** ▷ Backoff if mice queue is congested
20: $ssthresh_{new} \leftarrow ssthresh_{new} \cdot \gamma$
21: **if** $ssthresh_{new} > ssthresh_{old}$ **then** ▷ Preventing drastic change of ssthresh
22: $ssthresh_{\text{new}} \leftarrow \lambda \cdot ssthresh_{\text{old}} + (1-\lambda) \cdot BDP/N$
23: **else**
24: $ssthresh_{\text{new}} \leftarrow (1-\lambda) \cdot ssthresh_{\text{old}} + \lambda \cdot BDP/N$
25: $CWND \leftarrow Average(\ ssthresh_{new}, CWND)$
26: **Drop** $\texttt{NCF}_{\texttt{tp}}$

B. NCF Sender Logic

CCAs can immensely benefit from the rich, sub-RTT congestion feedback provided by $\texttt{NCF}_{\texttt{tp}}$. Algorithm 2 demonstrates how the sender logic (or sender-side CCA) can benefit from exploiting $\texttt{NCF}_{\texttt{tp}}$ data. The CCA or NCF's sender-side logic ($\texttt{NCF}_{\texttt{sl}}$) can, for instance, quickly determine the optimal sending rate using the data in a $\texttt{NCF}_{\texttt{tp}}$, namely, link capacity, bottleneck queue size, and count of active elephant flows. The inclusion of the number of active (elephant) flows allows the sender to adjust its bandwidth demands quickly in accordance with its fair (or equitable) share. The rich telemetry data, hence, enables a sender to quickly converge to its "optimal" throughput share both when new elephant flows join or old flows leave the bottleneck link. By delivering this telemetry data quickly (typically, with sub-RTT delay) NCF ensures that sender can react fast before the conditions at the bottleneck link change. The sender logic ($\texttt{NCF}_{\texttt{sl}}$) we present in this work is rather simple (based on NewReno), although our evaluations show how even this simple scheme fares better than other widely used CCAs.

Our algorithm consists of a few knobs that can be tuned according to the network's traffic characteristics. First, we enable the sender to re-adjust the BDP if we receive a new $\texttt{NCF}_{\texttt{tp}}$ signal and the last such feedback has expired (line 13 in Algorithm 2). The sender can obtain the current count of



(a) *Latency overheads* (b) *Bandwidth overheads*

Fig. 4. Even with our prototype implementation, we incur only negligible (a) latency and (b) bandwidth overheads in generating NCF and delivering it directly to the sender.

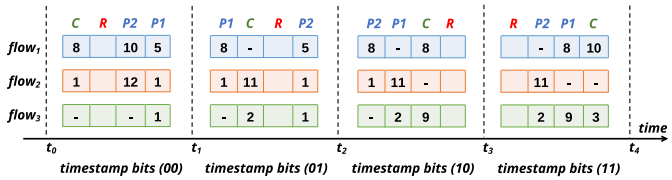


Fig. 5. The states of four flow-tracking counters $\{C, P_1, P_2, R\}$ at different points of time.

active flows in $\texttt{NCF}_{\texttt{tp}}$, and if that count changes beyond a threshold, τ, $\texttt{NCF}_{\texttt{sl}}$ re-computed the BDP. This mechanism allows the sender to cope with a burst of flow arrivals or departures. We further describe the parameter values and the choices behind these in Section IV.

IV. Prototype Implementation

We used the P4 programmable switch with Intel's Tofino1 chips [55] for building our prototype. This switch, with support for stateful packet processing, allows us to track flows at a bottleneck. It has multiple configurable priority RED queues that we use for flow-type isolation, and a mechanism to gather telemetry data. Lastly, its color-aware traffic policing feature enables congestion detection in the elephant flows' queue. Below, we discuss implementation details of the key functionalities of NCF.

Our prototype implementation tracks flows using rolling counters, as outlined in Section III-A, to monitor n elephant flows. Each packet triggers updates to at most four counters (see Section IV), resulting in constant-time complexity per packet. Although this approach is not as memory-efficient as, e.g., using a sketch-based solution—which are typically optimized for both time and space complexity [46], we believe it is sufficient to show the feasibility of the idea and offers predictable performance and low per-packet processing cost. Our design spans eight pipeline stages, with an average match-action SRAM utilization of 22.66% of the available capacity across all stages and an average map RAM (indirection/mapping resource) utilization of 32.55%. We use ten registers—seven with a width of 64 KB and three with a width of 16 B—along with the simple if-else logic outlined in Algorithm 1 for flow tracking and classification. The complete data-plane program comprises 951 lines of P4 code, including support for flow classification, flow separation, packet trimming, and packet mirroring.

To guarantee that our prototype implementation scales with flows, sketches (see Section III-A) would allow us to trade off accuracy for memory efficiency. Alternatively, a memory-efficient algorithm such as that described by Sivaraman et al. [45] could help in limiting flow tracking to only the "heavy-hitters," on busy links, which in turn will reduce memory usage, and subsequently only send congestion signals to the top flows filling up the link. Our implementation does not necessarily require all long-lived flows to be tracked, and we have empirically observed that reacting when there is a 30% of change in number of active flows already performs well on a range of scenarios (Section V).

ⓐ Tracking flow sizes: We use rolling counters using four P4 table registers $(wnd_0, wnd_1, wnd_2, wnd_3)$ for flow type classification: One register serves as the current time window (C), two store previous windows (P_2) and (P_1), and one supports re-initialization (R). We use bits 27 and 28 of the switches' hardware timestamp to trigger the rotation resulting in a period of $\approx$ 33.5ms. Re-initialization is delegated to the controller and takes about 15 ms, which is smaller than the time span we use.

At time t_0 (i.e., timestamp bits '00'), the first state in Fig. 5 shows the four registers, $wnd_0 - wnd_3$ serving as C, R, P_2, and P_1, respectively. At t_1, the registers "roll over" to the right: wnd_1 becomes current (C), wnd_0 denotes the previous time window (P_1), wnd_2 is re-initialized, and wnd_3 denotes P_2. Just before t_1, flows f_1 and f_2 contributed 8 and 1 packets, respectively, to the current interval. In earlier intervals (refer P_1 and P_2 corresponding to t_0), f_1 contributed 5 and 10 packets, and f_2 contributed 1 and 12 packets, making both f_1 and f_2 "elephant" flows. Flow f_3, in contrast, only contributed 1 packet during t_0; we label it as a short (or mouse) flow. At t_3, the counters indicate that f_2, an elephant flow, becomes a mice, while f_3 evolves from being a mouse to an elephant.

ⓑ Identifying mice and elephants: On receiving a packet, $\texttt{NCF}_{\texttt{dl}}$ computes a flow ID by hashing the packet's five-tuple using CRC_{16}. We then retrieve the counters corresponding to this flow ID from the registers and compare them to a predefined threshold (ρ) for determining whether the flow to which this packet belongs is an elephant or mouse. In our evaluations, we set ρ to 12; this threshold permits a flow, regardless of its size, to establish the connection (which typically involves one or two packets for the three-way handshake) as well as send an initial-cwnd worth of data (which is typically 10 packets [36], [37]). This assumption concerning the initial-cwnd is also informed by our empirical observation of flow sizes from sampled Internet transit-traffic data (Section II-A): A substantial number of flows constitute at most 10 packets in the MAWI data set (Fig. 2(a)).

ⓒ Isolating mice and elephants: We designate two RED queues for mice and elephants, with the same priority level from the eight queues available in the switch. Our design, therefore, enables other security or traffic-steering mechanisms, which might need these queues, to coexist alongside NCF on the switch. We then configured the RR scheduler to drain the two queues with equal priority and used a static allocation of memory for the two queues, $\alpha = 20/80$; mice

queue receives 20% of the available memory, and the elephant queue uses the rest. We set the congestion thresholds for the queues as follows: $\mu = 50\%$ (for the mice queue) and $\eta = 75\%$ (for the elephant queue). Our choice of α was guided by the characteristics of the traffic distribution. The network traffic traces from Facebook that many prior work used, for instance, reports that the share of traffic contributed by mice and elephant flows are 20% and 80%, respectively. For a given $\texttt{NCF}_{\texttt{dl}}$ deployment, a network provider can sample the traffic distribution and decide on this value, but we recommend allocating a smaller queue for mice than elephants. In our analyses, the choice of α was not sensitive to small variations in traffic distribution, and in particular if the delay between the traffic source and the $\texttt{NCF}_{\texttt{dl}}$ deployment was not substantially large. We defer further analyses of our design choices to Section V.

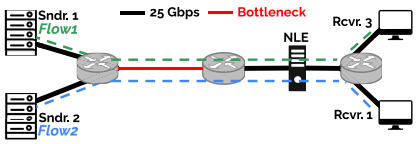
ⓓ ***Generating congestion feedback:*** Prior to forwarding a packet (P) of an elephant flow, we check the congestion status of the elephant queue. If the queue occupancy exceeds the predefined threshold, we generate a congestion notification ($\texttt{NCF}_{\texttt{tp}}$) *before* forwarding P. To generate a $\texttt{NCF}_{\texttt{tp}}$ we use the mirroring feature of the P4 switch, which is commonly used for multicasts or broadcasts. We mirror P and then *trim* the packet. We drop all but a subset of the original headers, including Ethernet (14 Bytes), IP (20 Bytes), and TCP (20 Bytes); unsurprisingly, we swap the address and port fields in these headers. $\texttt{NCF}_{\texttt{tp}}$, hence, retains the TCP sequence number and other pertinent data that enables the receiver of this signal (i.e., sender of the elephant flow) to identify precisely which packet triggered the congestion signal. Additionally, it helps the sender of the flow to distinguish between genuine and fake $\texttt{NCF}_{\texttt{tp}}$ signals. We also add a TCP option for the telemetry data (`congestion header` in Fig. 3) by using the `RES` TCP options field. In our prototype we statically configured the queues for mice and elephants. We encode the total queue capacity (rather than instantaneous queue occupancy) in $\texttt{NCF}_{\texttt{tp}}$ to expose the maximum available buffer to the sender. Notably, we also report (via $\texttt{NCF}_{\texttt{tp}}$) the number of flows competing for the buffer, which enables the sender to back off to the fair-share rate and release the buffer for other elephant flows. Shared-buffer configurations that dynamically apportion memory across queues are, however, becoming commonplace in production devices [56]. To accommodate such deployments, we propose extending $\texttt{NCF}_{\texttt{tp}}$ with the average queue occupancy, which then provides an estimate of the effective total queue capacity per ingress port. Lastly, we update the IP and TCP checksum fields. To maintain the count of active elephant flows (in `congestion header`) accurately, our prototype matches on the easily accessible TCP *RST* and *FIN* flags and decrements the appropriate flow counters. We can also easily extend the prototype to use timeouts for this task.

To estimate the overhead of generating NCFs (i.e., Algorithm 1), we measured the RTT of this feedback signal (i.e., time elapsed between when the sender sent the data packet and when it received the feedback), and compared it to the minimal achievable response time (i.e., time elapsed between when the sender sent a data packet and when it received the "echo" of that packet from the switch). In the former case, we perform all calculations to classify mice and elephant flows and generate $\texttt{NCF}_{\texttt{tp}}$ signals for all packets received from a sender, but in the latter, we simply strip the payload of the received packet, swap the source and destination addresses, and reflect it back to the sender. For both experiments, we configured a sender to transmit 1000 MTU-sized packets and measured the RTT of the response ($\texttt{NCF}_{\texttt{tp}}$ signals or "echo"s) from the switch. Per Fig. 4(a), the distribution of latencies incurred when generating $\texttt{NCF}_{\texttt{tp}}$ signals is quite similar to that incurred when simply generating echoes. In the worst case, for a few cases, generating $\texttt{NCF}_{\texttt{tp}}$ takes between 2 µs and 3 µs more than compared to sending an echo. We also measured the bandwidth consumption of $\texttt{NCF}_{\texttt{tp}}$ in a heavily loaded network by using a realistic traffic workload from Facebook (Section V-F) at various bottleneck-delay configurations. We observe (in Fig. 4(b)) that $\texttt{NCF}_{\texttt{tp}}$s consume little bandwidth (less than three orders of magnitude smaller than compared to the bottleneck bandwidth of 200Mbps) across all RTTs—even when the sender's sending rate is high.

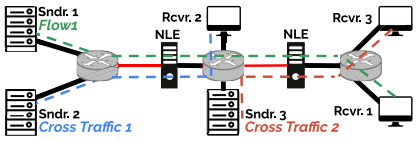
ⓔ ***NCF sender logic:*** We implemented $\texttt{NCF}_{\texttt{s1}}$ by extending NewReno in the FreeBSD TCP stack. We retained NewReno's state machine for managing flow progress, i.e., `SEQ` generation, `ACK` processing, RTT estimation, and loss recovery. We extended the state machine for parsing the data in $\texttt{NCF}_{\texttt{tp}}$ (Algorithm 2) and updating the relevant TCP parameters (i.e., `cwnd` and `ssThresh`). Though $\texttt{NCF}_{\texttt{dl}}$ sets the `ACK` bit in $\texttt{NCF}_{\texttt{tp}}$ to prevent the signal from being dropped (since no flags are set), the $\texttt{NCF}_{\texttt{s1}}$ can disambiguate it from a 'traditional' TCP `ACK`. $\texttt{NCF}_{\texttt{s1}}$ reacts to at most one $\texttt{NCF}_{\texttt{tp}}$ per RTT, independent of the source of the $\texttt{NCF}_{\texttt{tp}}$. Upon receiving a $\texttt{NCF}_{\texttt{tp}}$, the $\texttt{NCF}_{\texttt{s1}}$ checks when it received the last $\texttt{NCF}_{\texttt{tp}}$. If the last $\texttt{NCF}_{\texttt{tp}}$ has not expired, it discards this new $\texttt{NCF}_{\texttt{tp}}$. Otherwise, it inspects the $\texttt{NCF}_{\texttt{tp}}$ and if the count of active flows is non-zero, then $\texttt{NCF}_{\texttt{s1}}$ computes `ssThresh` and `cwnd` using the data in $\texttt{NCF}_{\texttt{tp}}$ and based on the connection's state machine.

In our experiments using diverse workloads, we empirically observed setting $\tau = 0.3$ (i.e., a threshold of 30% of change in number of active flows) performs well on a wide range of network scenarios and conditions. To enable the sender to back off and drain any queue build up, we set $\delta = 0.8$; it quickly lowers the `ssThresh`. In the event that a flood of mice flows congest the mice queue, γ allows the sender to throttle its rate and alleviate this transient congestion. The $\texttt{NCF}_{\texttt{s1}}$ performs such throttling at most *once* per RTT to avoid starvation or introduce DDoS vulnerabilities. Lastly, λ prevents drastic changes in `ssThresh`. Based on our empirical evaluations in a wide range of network settings, we set γ and λ to 0.95 and 0.7, respectively. $\texttt{NCF}_{\texttt{s1}}$ is only one of many feasible CCA designs enabled by `NCF`, but it demonstrates how we can rethink or retrofit a CCA to benefit from `NCF`.

The preliminary version of this work did *not* have a prototype implementation or evaluation sections [35].



(a) *Single bottleneck setup.*



(b) *Multiple bottleneck setup.*

Fig. 6. Network testbed for performing head-to-head comparisons of NCF against other CCAs.

V. Evaluation

In addition to our NCF realization using Intel's Tofino1 chipset [55], we also implemented NCF on NS-3 [57]. Below, we discuss how we used both to characterize NCF's performance and demonstrate its benefits.

A. Experimental Setup

Testbed: We built the testbed using three APS Networks BF6064X-T programmable switches with Intel's Tofino1 chip in a dumbbell topology as shown in Fig. 6. We used eight Dell R6515 blade servers each with 16 cores and 128GiB of RAM, running Linux kernel version 5.4 (Debian 10). We configured some as traffic senders, a few as network latency emulators (NLEs), and the rest as receivers. We installed one or two Broadcom NetXtreme BCM5720 25 GbE NICs, as required, and used 25Gbps DACs for interconnecting the servers and switches. We varied the capacity of the bottleneck link (marked in red in Fig. 6) as needed for the different experiments using the traffic shaping feature of the Tofino switches. The base RTT between sender-receiver pairs in the testbed, without any added delay, was less than 0.06ms. We, nevertheless, introduced different one-way delays using DEMU [59], a DPDK-based solution with low overhead and high accuracy, at the NLE(s). We evaluated various CCAs in both single bottleneck (Fig. 6) and multiple bottleneck (Fig. 6(b)) settings and using a range of bottleneck buffer configurations.

Traffic generators and workloads: We implemented $\texttt{NCF}_{\texttt{sl}}$ using F-Stack [60], a DPDK based framework to bypass the kernel overhead. We used hping (v3), which repeatedly opens a TCP connection, sends 50 B, and tears it down, to generate a flood (i.e., 200Mbps of traffic) of mice flows. We also used a simple client-server setup to open new connections (and tear them down as soon as they are setup) every 100ms to generate repetitive mice flows for latency-related tests. We generated elephant flows using iperf (v2). For large-scale evaluations, we generated workloads based on the data from Parsonson et al. [61] which follow the widely used traffic traces from Facebook [62]. In particular, we derived the flow size and inter-arrival time distributions (Weibull and Lognormal, respectively) from the traces, and then used Harpoon [63] to generate traffic with 400 concurrent senders using samples from these empirical distributions.

Choice of CCAs: We compare NCF against the two widely used CCAs, namely CUBIC [64] and NewReno, as well as the relatively modern BBRv1 [4] from Google. Our choice of CCAs is influenced by the availability of CCA implementations on F-stack. We supplement these testbed evaluations with experiments on the NS-3 simulator, where we compare NCF with Bolt [9] and HPCC [32].

B. Isolating Mice and Elephant Flows

We demonstrate the need for isolating mice and elephant flow types in the single bottleneck setting (Fig. 6(a)) as follows. We start two elephant flows, and provide them sufficient time (i.e., 35 s[1]) to saturate the bottleneck (200 Mbps) and obtain their fair shares. We then create a flood of mice flows (using hping) that generate up to 200 Mbps traffic to saturate the bottleneck. We used a tail-drop queue of size 1 BDP at the bottleneck link and added a delay of 10 ms through the NLE (refer Fig. 6(a)).

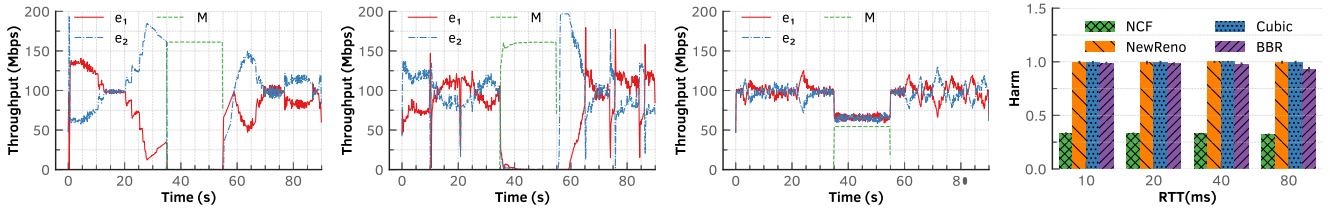
With both Cubic (in Fig. 7(a)) and BBR (in Fig. 7(b)), the elephant flows struggle to compete with the mice flows; the latter rapidly consume about 80% of the bottleneck bandwidth. We observe similar behavior for NewReno (Appendix A). The short lifetimes of mice flows makes it impractical for any congestion control mechanism to rein them in—the elephant flows being in steady state has no bearing on the outcome. Unlike these CCAs, NCF (in Fig. 7(c)) prevents the mice flood from impairing the performance of elephant flows by isolating the two flow types via dedicated queues. We quantified the flow-type isolation in terms of the "harm" inflicted by mice flows on the elephant flows. To this end, we computed the *harm factor* [58] by comparing the average throughput of the elephant flows before and after the introduction of the flood of mice flows. Per Fig. 7(d), mice flows harm elephant flows only about a third as much as they do when using Cubic, BBR, or NewReno.

The use of dedicated queues for the two flow types allows elephant flows competing on a bottleneck to fill up only their own queue; the mice queue and, hence, the RTTs of mice flows are unaffected by the elephant flows. To illustrate this benefit we repeat the last experiment with one change: Instead of creating a flood of mice flows, we introduced new mice flows repeatedly, one after another. With Cubic and BBR (Fig. 8), due to the lack of dedicated queues for the flow types, competing elephant flows fill up the queue and inflate the RTTs of the (periodic) mice flows.[2] The medians of mice flow RTTs (Fig. 8(b)) are at least 60% higher than the theoretical minimum (of 20 ms) in Cubic, BBR, and NewReno. High RTT inflations and fluctuations can be detrimental for the mice flows, which typically represent latency-sensitive or interactive applications, e.g., [9], [13].

The RTTs of mice flows in NCF are minimal, with virtually no inflation or fluctuations (Fig. 8(a) & Fig. 8(b)). Even if the elephant flows try to fill up their queue ("NCF(e)" in Fig. 8(c)), the mice queue (labelled "NCF(m)") remains virtually empty.

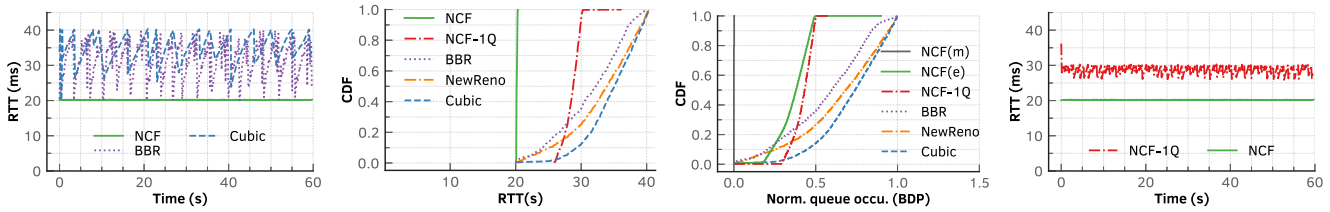
[1]We start the mice flows at 35s to avoid interfering with BBRv1's RTT probing phase [4].

[2]We omitted NewReno in Fig. 8 for readability.



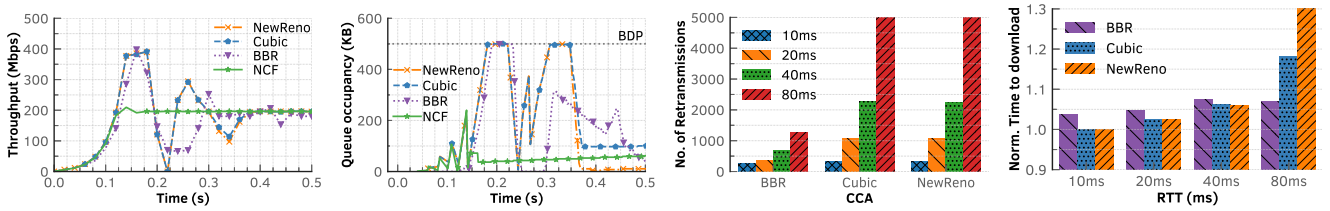
(a) *Cubic; RTT: 20 ms* (b) *BBR; RTT: 20 ms* (c) *NCF; RTT: 20 ms* (d) *Harm factor [58]*

Fig. 7. The mice flow (M) affects the throughput of the elephant flows (e_1,e_2) substantially when the flows use (a) Cubic and (b) BBR, even if all flows experience the same RTT. NCF, in contrast, achives (c) an equitable bandwidth sharing between two flow types, and ensures that flows experience (d) significantly lesser harm than that when using other CCAs.



(a) *Timeline of mice RTTs* (b) *CDF of Mice RTTs* (c) *Queue usage* (d) *NCF vs. NCF-1Q*

Fig. 8. When recurring mice flows compete with elephant flows on a bottleneck, NCF provides minimal RTTs for mice flows. Other CCAs, however, exhibit large (a) variations and (b) inflations. NCF also (c) lowers queue overall occupancy. Without the use of separate queues, NCF-1Q performs poorly compared to NCF.



(a) *tput. spikes; RTT: 20 ms* (b) *Queue usage; RTT: 20 ms* (c) *Retransmissions* (d) *Download times*

Fig. 9. NCF's quick feedback to a sender (a) eliminates slow-start (throughput) spikes and (b) prevents it from filling up the bottleneck queue. (c) NCF senders experience virtually no loss or retransmissions during slow start. (d) For a given volume of data, they achieve smaller download times than other CCAs.

In every other CCA, the comparatively high queue occupancy (Fig. 8(c)) coupled with the lack of dedicated queues for the two flow types results in mice flows experiencing high RTTs.

NCF fares well compared to other CCAs because of two factors: the use of (i) sub-RTT feedback and (ii) dedicated queues for the two different flow types. To analyze the benefit of classification of the flows and the subsequent flow isolation, we created a variant of NCF that uses a single queue (labeled "NCF-1Q" in Fig. 8). NCF-1Q clearly benefits from the sub-RTT feedback, showing relatively small RTT inflation and fluctuations (Fig. 8(b)) and low queue occupancy (Fig. 8(c)) compared to other CCAs. Without flow type isolation (in NCF-1Q), however, mice flows get impeded by elephant flow packets in the queue (Fig. 8(d)): RTTs of mice flows when using NCF-1Q are, hence, larger than those when using NCF.

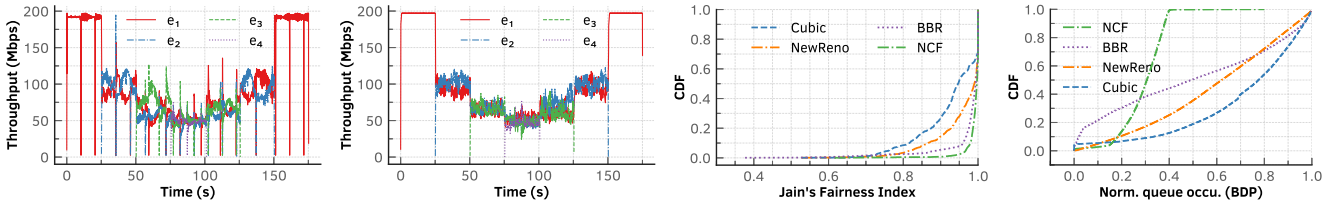
Takeaways. The use of dedicated queues to separate mice flows from elephant flows is fundamental to providing performance isolation between the flow types. Dedicated queues ensure that a flood of mice flows does not impair the throughputs of elephant flows sharing the bottleneck link. They also ensure that mice flows experience minimal RTTs, even in the presence of elephant flows.

C. Taming Slow Start

During *slow start*, a TCP sender increases its sending rate rapidly until it encounters a loss or reaches the slow-start threshold. At scale, the slow-start phase of multiple concurrent senders at a bottleneck can induce high losses due to the senders' rates (occasionally) exceeding the bottleneck link capacity; the losses, in turn, lower the link utilization. Below, we evaluate NCF's effectiveness in taming the throughput spikes during slow starts.

We show, in Fig. 9(a), the throughputs of different TCP flows, each run separately, in a single bottleneck setting (Fig. 6(a)). We set the bottleneck to 200 Mbps, the delay to 10 ms, and the bottleneck queue to 1 BDP. The slow-start phase of all TCP senders, except for NCF, substantially overshoot the bottleneck capacity. NCF, unlike the other CCAs, eliminates the slow-start spike: The sender receives a timely (NCF) signal to tame its sending rate. Even during the aggressive slow-start phase, NCF does not fill up the bottleneck queue or buffer like other CCAs and quickly drains the packets in the queue (Fig. 9(b)).

The ability to tame slow starts has significant implications for reducing retransmissions during this phase. To capture the



(a) *BBR* (b) *NCF* (c) *JFI* (d) *Queue usage*

Fig. 10. When we stagger the arrivals and departures of four elephant flows (e_1, e_2, e_3, e_4) at a bottleneck link of 200 Mbps by 25 s, (a) the flows struggle to achieve their fair shares in BBR, while (b) they converge quickly to their fair shares in NCF; NCF offers (c) higher fairness and (d) smaller bottleneck buffer occupancy than the other CCAs.

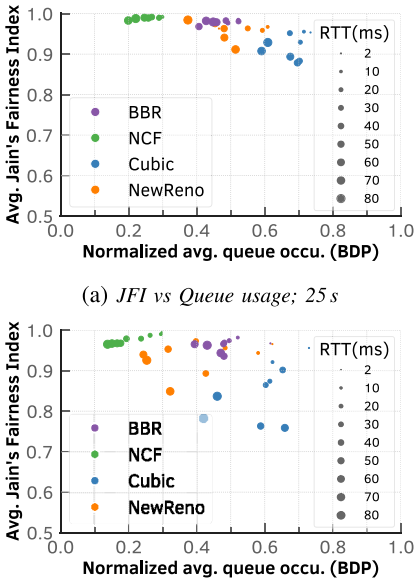
implications, we plot the number of retransmissions observed during the delivery of the first 20 K segments for each CCA as a function of RTT in Fig. 9(c). NewReno, Cubic, and BBR experience many retransmissions, particularly at large link latencies common in WAN settings, since their feedback signal (e.g., packet loss or delay) must traverse the round-trip between the sender and receiver. BBR fares well compared to Cubic and NewReno, since it closely monitors RTT to pace its sender and does not rely on losses. NCF, unlike the other CCAs, sustains *no* losses, and, hence, offers minimal download times at all tested RTTs. Fig. 9(d) shows the implications of retransmissions for download times: Cubic and NewReno, which suffer numerous retransmissions at 80ms RTT (Fig. 9(c)), are slower than NCF—by a factor of 1.18 and 1.3, respectively—in downloading the first 30MB, or about 20 K MTU-size packets.

Takeaways. *NCF's quick (i.e., sub-RTT) feedback is effective in preventing a sender from overshooting the bottleneck link capacity, thereby, significantly reducing retransmissions.*

D. Fairness Between Similar Flows

We now evaluate NCF's ability to facilitate an equitable bandwidth sharing between flows. To this end, we run a staggered-flow experiment (similar to that of HPCC [32]), where we introduce four flows, one after another, in a single-bottleneck setting (Fig. 6(a)). More precisely, we add a new flow to the testbed every 25s until we reach four concurrent flows, and then remove one flow every 25s until we have only one left. The four flows are *similar*: They use the same CCA and have same (20ms) RTT. We measure how well the flows share the bottleneck bandwidth (of 200Mbps) as they join and leave.

The staggered flows struggle to acquire their fair share when using BBR (Fig. 10(a)), both during flow arrivals and departures. More importantly, changes in the number of active flows substantially affect the throughput of existing flows, and, consequently, the bottleneck queue occupancy (Fig. 10(d)). Cubic and NewReno show similar behavior, and we present them, hence, in Appendix B. NCF, in contrast, allows the flows to converge quickly to their fair shares (Fig. 10(b)). We also observe substantially smaller throughput fluctuations in NCF flows than those of the other CCAs. We periodically (i.e., every 80 ms) compute the Jain's fairness index (JFI) for concurrent flows, and the CDF of the JFI of the flows over time (Fig. 10(c)) shows that flows share bandwidth more equitably when using NCF than when using other CCAs. NCF also minimizes the bottleneck queue usage (Fig. 10(d)): The median queue occupancy for BBR, NewReno, and Cubic are at least 66% more than that of NCF.



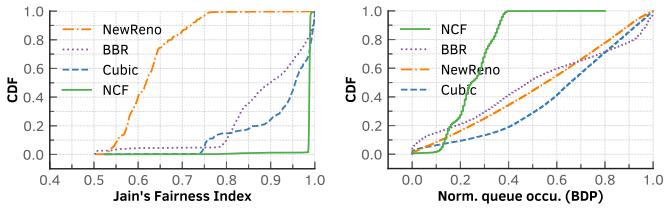
(a) *JFI vs Queue usage; 25 s*

(b) *JFI vs Queue usage; 5 s*

Fig. 11. NCF scores high on fairness with minimal queue occupancy across a range of bottleneck link delays, even if we reduce the arrival and departure intervals from 25 s to 5 s.

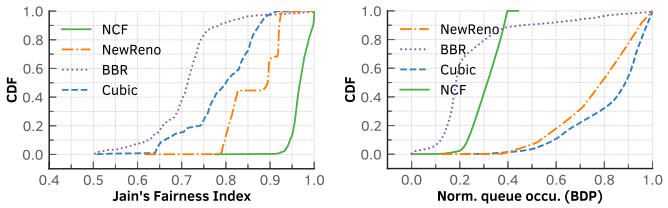
To analyze the implications of latency on the fairness and bottleneck queue occupancy, we repeat the staggered-flow experiment across a range of bottleneck delay settings (Fig. 11(a)). NCF scores high on the Jain's fairness index with minimal queue use compared to all other CCAs. BBR also scores high on fairness, but with queue occupancies nearly twice as large as that for NCF. NewReno and Cubic show significant variation in fairness and queue occupancy, and perform poorly at low delays. We then reduce the time between flow arrivals and departures—to 5s instead of the 25s used earlier—to evaluate how the CCAs perform when network traffic volume (across the bottleneck link) changes quickly. We observe that NCF still scores high on fairness with low queue occupancies (Fig. 11(b)). The other CCAs perform much worse on fairness—with NewReno and Cubic scoring sometimes as low as 0.85 and 0.75, respectively, on fairness—when traffic volume changes rapidly.

Takeaways. *NCF enables fair sharing of the bottleneck bandwidth across flows while minimizing bottleneck buffer*



(a) *JFI* (b) *Queue occupancy*

Fig. 12. Shallow buffer scenario: NCF offers (a) high fairness and (b) low queue occupancy compared to other CCAs for two dissimilar elephant flows—one with 10 ms and the other with 80 ms RTT.



(a) *JFI* (b) *Queue occupancy*

Fig. 13. Deep buffer scenario: NCF offers (a) excellent fairness and (b) low queue occupancy compared to other CCAs for two dissimilar elephant flows—with 10 ms and 80 ms RTTs.

occupancy, across a wide range of bottleneck link delays and regardless of whether flows arrive at and depart from the bottleneck link rapidly or gradually.

E. Fairness Between Dissimilar Flows

We now evaluate the behavior of various CCAs when *dissimilar* flows (i.e., flows experiencing different RTTs or using different CCAs) share a bottleneck link.

① ***RTT fairness:*** Evaluations concerning fairness of CCAs in prior work (e.g., [65], [66]) typically consider flows with similar RTTs. In practice, outside the realm of data-centers, flows typically experience different RTTs. Such RTT differences between flows sharing a bottleneck link, even if they all use the same CCA, make it challenging for the CCA to ensure a fair share of bandwidth across the flows. The signals typically used by a sender (e.g., packet loss or delay), for determining how much it can send and/or how fast, must traverse the path from the bottleneck to the receiver, which then echoes it back to the sender. The RTT, hence, dictates a sender's response time: the larger the RTT the slower the sender reacts to changes in network conditions.

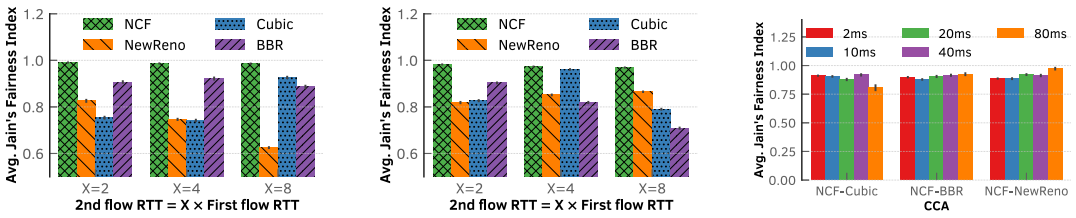
We now evaluate the behavior of various CCAs when *dissimilar* flows contend for bandwidth at a bottleneck. To this end, we stagger the starts of two elephant flows to be 15s apart, and we measure the fairness and queue occupancy across different CCAs. The first flow has a smaller RTT than the second flow, and we set the RTT of the latter as an integral multiple of that of the former. Since the size of the bottleneck queue can be large or small depending on the location of the bottleneck (e.g., whether it is in close proximity to a CDN's edge or content provider's infrastructure, or far away), we repeat our evaluation under both *shallow* and *deep* buffer configurations. We use 250KB, the bandwidth delay product (BDP) for flows with 10ms RTTs, for a shallow buffer, and we set the size of deep buffers to one of $\{500KB, 1000KB, 2000KB\}$ corresponding to BDPs of 20ms, 40ms and 80ms, respectively. We set the bottleneck buffer to the BDP of the flow with the shorter RTT for the shallow buffer and the longer RTT for the deep buffer scenarios, and we set the bottleneck bandwidth to 200Mbps.

In the shallow buffer configuration, even when the RTTs of the flows differ by a factor of eight, NCF ensures that the flows share the bandwidth equitably (Fig. 12(a)). None of the other CCAs offer comparable fairness, and NewReno performs the worst—the smaller-RTT flow enjoys a substantially larger share than the other; we present the throughput timelines for each CCA in Appendix C. The fairness offered by NCF is rooted in its ability to keep the queue occupancy small (Fig. 12(b)), despite the dissimilar RTTs of the flows as well as the sub-RTT feedback. One aspect that benefits NCF is that the feedback delay for both competing flows is similar, which should be the case for most of the envisioned deployment scenarios. Next, we keep the 10ms RTT of the first flow intact, but vary the RTT of the second one as a multiple of the first. Regardless of the differences in RTTs, per Fig. 14(a), NCF ensures a fair share between the flows.

To emulate the deep buffer scenario, we set the bottleneck buffer to the BDP of the flow with larger RTT. NCF once again assures fairness (Fig. 13(a)) despite the vast difference in RTTs between the flows, i.e., 10ms and 80ms. Its queue usage is slightly larger than that of BBR, albeit its fairness is substantially better than BBR (Fig. 13(b)). As with the shallow-buffer experiment, we vary the RTT of the second flow as a multiple of the first one, and measure the fairness of CCAs in each configuration. Regardless of how dissimilar the RTTs of the flows are, Fig. 14(b) shows that NCF offers better fairness than the other CCAs in all configurations.

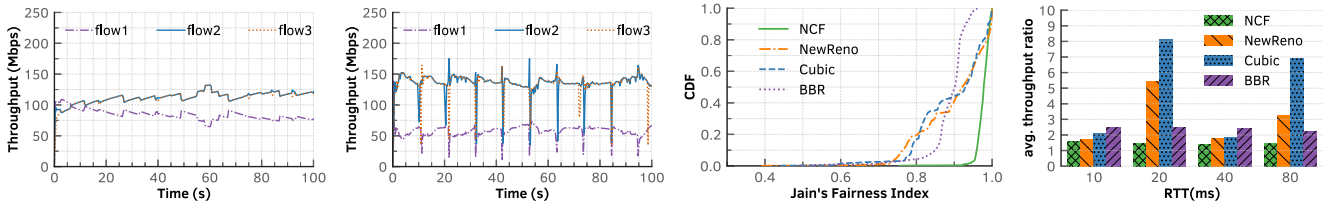
Takeaways. Regardless of how dissimilar flows are (with respect to their RTTs), NCF guarantees fairness of flows under both deep and shallow buffer configurations.

② ***Inter-CCA fairness:*** When an NCF flow competes with a non-NCF (i.e., Cubic, or BBR, or NewReno) flow for bandwidth, the former acquires less than its fair share of the bottleneck bandwidth, even when the flows experience similar RTTs (Fig. 14(c)). This outcome should not be surprising, since NCF reacts to the buffer filling up much quicker than any of the other CCAs, owing to its short control loop. Naturally, NCF backs off faster than other CCAs when the buffer fill up. Although NCF loses some of its bandwidth share, it fares well against BBR, which relies on monitoring the RTT to pace the sender, even though the latter is known to be aggressive [65]. NCF becomes reasonably competitive when we increase RTTs (say, to 80ms) to capture delivery over long-distance links. We can, hence, adopt learning-based methods to help NCF adapt to the network conditions [6], [67]. In the scenarios where we typically expect NCF to be deployed, including edge servers of CDNs and cloud providers, we can easily monitor NCF's performance and safely customize or tune the protocol on the fly. As such, NCF does not threaten the performance or stability of competing non-NCF flows. We



(a) *JFI in shallow buffer settings* (b) *JFI in deep buffer settings* (c) *JFI in inter-CCA scenarios*

Fig. 14. When we vary the RTT of the second flow as a multiple of the first (with 10 ms RTT), NCF offers better fairness than other CCAs in both (a) shallow and (b) deep buffer configurations in all tested scenarios. (c) NCF does not harm other CCAs in the WAN, but it backs off more quickly than other CCAs, sacrificing some of its bandwidth share.



(a) *NCF; RTT: 40ms* (b) *BBR; RTT: 40ms* (c) *JFI* (d) *Avg. throughput ratio*

Fig. 15. For elephant flows in a multiple bottleneck scenario (Fig. 6(b)), (a) NCF enables flows to achieve a fair share while (b) BBR struggles to offer fairness. NCF also achieves a (c) high Jain's fairness index, and (d) near ideal average throughput ratios compared to other CCAs under various RTT settings.

leave the optimization of NCF to cater to specific network conditions, deployment strategies, and evaluation of learning-based approaches for fine-tuning NCF to future work.

Takeaways. NCF can be safely deployed in the Internet. Though it loses some of its fair share to competing CCAs, it performs competitively in WAN settings, highlighting the opportunities for research into CCA optimizations.

F. Multiple Bottleneck Scenarios

Prior work demonstrated that a flow that traverses multiple bottlenecks (marked in purple in Fig. 6(b)) suffers substantial throughput degradation compared to single-bottleneck flows (marked in blue and tawny) [68]. Below, we evaluate how an NCF flow performs under such conditions by emulating the parking-lot topology with the "too many red lights" problem [68]. For this evaluation, we configured the NLEs (Section V-A) such that all flows have the same (40ms) RTT.

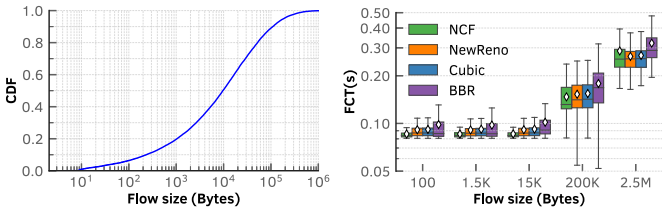
All elephant flows: When all three flows in Fig. 6(b) are elephant flows, we observe that the multi-bottleneck (NCF) flow, in purple, receives less throughput than either of the single-bottleneck (NCF) flows (Fig. 15(a)). When the flows use BBR, per Fig. 15(b), the multi-bottleneck flow struggles to even receive a quarter of the bottleneck bandwidth. Cubic and NewReno exhibit similar behavior (refer Appendix D). NCF clearly outperforms all these CCAs with respect to fairness (Fig. 15(c)). Regardless of the source of the NCF_{tp} signals, NCF_{sl} reacts to *at most one* NCF_{tp} per RTT, independent of the source of the NCF_{tp} (Section IV). We show the ratio of the average throughput of the single-bottleneck flows to that of the multi-bottleneck flow, which evaluates the ability of a CCA to provide an equitable share to the multi-bottleneck flow [69], for the various CCAs in Fig. 15(d). The ratios in this figure demonstrate that NCF consistently offers a ratio close to 1, which is the ideal value, for a wide range of RTTs.

Facebook workload. Next, we evaluate NCF using a realistic workload from Facebook. We replace each of the three flows in Fig. 6(b) with a distribution of heavy-tailed flows (shown in Fig. 16), based on the Facebook workload [62]. More concretely, we use Harpoon [63] to generate flows (with each sender generating up to 400 concurrent flows) with the flow sizes and inter-arrival times sampled from the empirical distributions of corresponding metrics from the Facebook workload (refer Section V-A). All flows have 40 ms RTT, regardless of whether they are single-bottleneck or multi-bottleneck flows. Per Fig. 16(b) NCF offers the lowest FCTs of all CCAs for the short flows in bins labeled "100", "1.5K" and "15K" (indicating sizes in bytes).

Takeaways. Even in challenging multiple bottleneck scenarios, NCF ensures fairness across all flows. In our evaluations using the widely used Facebook workload, NCF offers the lowest FCTs among all other CCAs for short flows, which represent interactive or latency-sensitive applications.

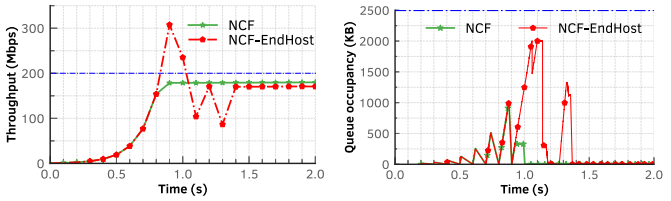
G. Other Network-Assisted Mechanisms

We now compare NCF to the state-of-the-art prior work that use similar network support for congestion control. We refrain from comparing NCF to ECN since ECN signal is not rich (i.e., a 1-bit signal) compared to NCF_{tp}. We analyze, however, the performance of a "rich" ECN-like version of NCF that we call NCF-EndHost. This scheme is similar to NCF in that it isolates mice and elephant flow types using separate queues and uses same sender logic (i.e., NCF_{sl}) as NCF. Instead of directly sending NCF_{tp} to the senders (of elephant flows), it, however, relies on the receiver to reflect the signal (similar to ECN). To help receivers reflect the NCF_{tp}



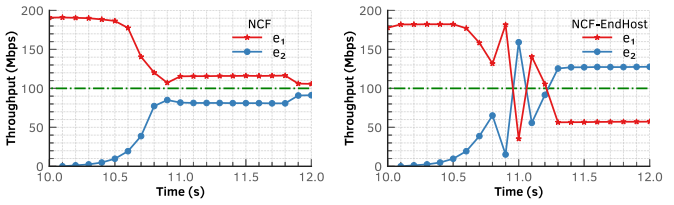
(a) *Flows in Facebook workload* (b) *FCTs; RTT: 40ms*

Fig. 16. In a multiple bottleneck settings with flow sizes sampled from the Facebook workload, NCF offers the lowest FCTs for the short flows in a multiple bottleneck scenario.



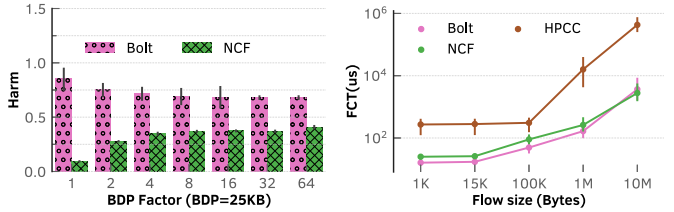
(a) *Sstart throughput* (b) *Sstart total queue occupancy*

Fig. 17. Though NCF and NCF-EndHost both use NCF_{tp} to congestion-control the sender, the sub-RTT signaling in the former helps it perform much better than the latter.



(a) *NCF tput. convergence* (b) *NCF-EndHost tput convergence*

Fig. 18. NCF-EndHost convergence slower and achieves lesser fairness than NCF.



(a) *Harm factor* (b) *FCTs: median and IQR*

Fig. 19. (a) When a flood of mice flows share a bottleneck link with elephant flows in WAN environments, NCF isolates the flow types better than Bolt. (b) In a simulation of datacenter traffic based on the Facebook's Hadoop workload, NCF offers much better FCTs than HPCC, and performs similar to Bolt, despite not being optimized for such environments.

data, we run a *sniffer* script (on the receiver side) that captures NCF_{tp} and reflects it back to the sender. This design eliminates interfering with the flow's TCP state-machine handling logic at the receiver side.

Per Fig. 17, the rich telemetry data (NCF_{tp}), by itself, does not help with taming the sender from overshooting the bottleneck bandwidth. In this experiment, we started an elephant flow and let it run on a path with the bottleneck bandwidth of 200Mbps, 100ms RTT, and a 1 BDP buffer. Though NCF and NCF-EndHost both use NCF_{tp}, the sub-RTT feedback in case of the former allows it to eliminate slow-start throughput spikes and, as a consequence, minimize bottleneck queue occupancy.

We now evaluate how NCF-EndHost fares compared to NCF in enabling competing elephant flows to share the bottleneck equitably. We start the experiment with an elephant flow, and we let it run for 10 s before starting a second elephant flow. We keep the network conditions identical to that of the prior experiment. Once again, the sub-RTT signaling in NCF enables the flows to converge quicker and achieve higher fairness (in sharing) than NCF-EndHost (Fig. 18). By the time, the rich congestion feedback reaches the sender NCF-EndHost would have likely sent a train of segments (of size approximately equivalent to the BDP), and this inflight data drastically degrades its fairness and throughput convergence properties.

To the best of our knowledge, *we are the first to design a rich, sub-RTT congestion-control mechanism for use in the public WAN or Internet.* We, hence, compare our approach to Bolt [9], which explores a similar idea, albeit in a controlled datacenter environment. Since the assumptions about operating or deployment environments for the two approaches differ drastically, we use a two-fold evaluation to perform a fair comparison.

First, we evaluate Bolt's performance in WAN-like environments where NCF excels. We simulate a flood of mice flows to compete with elephant flows in NS-3, similar to the experiment described earlier in Section V-B in a single-bottleneck setting. Since Bolt was designed for datacenter environments, we refrain, however, from using large RTTs (i.e., few tens of milliseconds) typical in the Internet, but set flow RTTs to 2ms—the minimum typically observed in WAN links [69]. We set the bandwidth to 100Mbps and vary the queue size as an integral multiple of the BDP. The harm factors in Fig. 19(a) demonstrate that NCF fares well compared to Bolt; the latter requires large queues to constrain the harm caused by mice flows to the elephant flows. The performance of NCF decreases with increasing queue sizes, because the queue occupancy is typically well below the threshold for sending congestion notifications (refer Section IV). Though the threshold must be changed when queue size changes, we kept the threshold static to demonstrate the benefits of NCF even when adhering to our default configuration. Per this figure, NCF's queue separation approach is still effective in isolating the performance of the two flow types.

Second, we test whether NCF's ideas would generalize to datacenter environments. We simulate, hence, a datacenter scenario and compare NCF to Bolt as well as HPCC [32], another well-known prior work. We simulate traffic using the Facebook's Hadoop workload from [62] at 80% load, set bottleneck bandwidth to 25 Gbps and RTT to 11.6 µs (based on [9]). Per Fig. 19(b), NCF offers much smaller FCTs than HPCC owing to its short control loop. Most important, NCF's performance is quite similar to Bolt, despite NCF not implementing any of Bolt's key optimizations such as proactive ramp-up [9].

Takeaways. The state-of-the-art prior work on congestion control in datacenter networks do not, unsurprisingly, generalize to the WAN settings; NCF's queue separation logic gives it a clear edge over Bolt in such settings. Furthermore, though NCF is not optimized for datacenters, it is competitive against Bolt and HPCC in such environments.

VI. Related Work

There is an extensive body of prior work on congestion control, with numerous CCAs schemes designed to target diverse network environments, meet various application requirements, and leverage advancements in network infrastructure. Huang et al. [70] provide a comprehensive survey of this space.

Traditional CCA proposals, however, suffer from two key shortcomings: They depend on imprecise signals (e.g., loss and delay), and rely on the receiver to reflect the signal back to the sender. This dependency on the receiver results in a delay of at least one RTT before the sender can apply corrective measures to its delivery rate. Cubic [64], the default Linux kernel CCA, for instance, detects congestion by inferring losses based on the ACKs. Cubic, as a result, is typically slow to react to congestion and consequently fills up the bottleneck queue [71]. BBR, a relatively modern CCA, in contrast, combines window-based and rate-based mechanisms to minimize queue buildup [4]. BBR has seen widespread adoption [72], [73], [74], and our evaluation of NCF compares its performance to BBR, Cubic, and NewReno [75], an earlier loss-based CCA.

The idea of eliciting explicit network support for congestion control derives its roots from DECBit [76], RED [77], and ECN [78]. These approaches broadly rely on two techniques to signal congestion: mark one or more bits in the packet headers that the receiver will reflect back to the sender (e.g., ECN [78]), or preemptively drop packets before the buffer overflows (e.g., RED [77] and CoDel [79]). The congestion signals used are quite narrow in scope (for instance, they do not indicate the degree of congestion), and some require receiver support.

Recent advancements, such as the Low Latency, Low Loss, Scalable throughput (L4S) framework [80], improve congestion quantification and reduce notification delays, but their deployment still requires support from the receivers. L4S combines a scalable CCA (e.g., TCP Prague [81]), Accurate ECN (AccECN) [82], and a dual-queue AQM [83] that isolates L4S and non-L4S traffic at the bottleneck. The current L4S system uses a single ECN bit to be set at the sender side, enabling the switch (with Dual Queue AQM) to distinguish L4S traffic from classic traffic [80]. Our approach, in contrast, does not rely on the sender to tag or classify flows. The switch dynamically classifies flows as either mice or elephants; our approach ensures that long-lived flows can react to queue build up, without HOL-blocking flows that do not (or have the ability to) benefit from congestion signals. Our approach improves the utility of the congestion signaling by ensuring that flows unable to react to congestion do not trigger unnecessary congestion signals, as such flows would have completed their transfers by the time they receive these signals.

Other early work on in-network solutions for congestion control pursued a "clean-slate" approach. XCP, for instance, used a congestion header to communicate flow state (e.g., cwnd and RTT) between routers and receivers [31], while RCP required routers along the path to compute per-flow rates [33]; both required support at each hop along the path. More recent efforts, however, have shifted focus to datacenter environments, (e.g., DCTCP [5] and TIMELY [84]). TIMELY [84], for instance, relies on accurately estimating RTT; feasible in datacenter environments where end hosts support hardware time stamping. The advent of programmable switches has further led to exploring novel uses for In-band Network Telemetry (INT) data provided by these switches. HPCC, for instance, enriches the congestion signal with rich data such as the current load on the switch egress port, but it, unfortunately, relies on receivers to reflect this signal back to senders [32]. NCF is inspired by prior work, but extends them in non-trivial ways: It does not require receiver support, offers sub-RTT signaling, and is suitable for public Internet.

Related approaches investigated various methods of directly notifying senders of congestion, albeit in the datacenter context. QCN [85] was an early effort to provide direct feedback to the sender, though its congestion notifications were restricted to the L2 domain. FastLane, in contrast, requests switches to send high-priority notifications directly to senders as soon as a packet drop is detected, however, waiting until a packet drop might already be too late [86]. RoCC [67] and BFC [87] also use sub-RTT congestion signaling at intermediate nodes. RoCC tracks elephant flows at the switch and calculates each flow's fair rate using the queue size, but was designed specifically for RDMA traffic.[3] BFC is a per-hop, per-flow control mechanism that requires dedicated queues per-flow for optimum performance [67]. It also requires upstream switches to pause queues when a bottleneck sends congestion signals, which is impractical in WAN environments. Bolt uses precise congestion information from the switch to proactively control a sender's rate [9]. Although conceptually similar to NCF, Bolt is designed for datacenters (with microsecond delays), relies on support from endpoints (for proactive ramp-up) and does not take into account that mice flows cannot be congestion controlled, though they can easily fill up queues [39], [41].

More recently, Zhuge [13] shortened the control loop in wireless networks by modifying the access point (AP) to inform the senders of the delay experienced by its packets at the AP. Zhuge is, nevertheless, designed to handle congestion in the last mile and their approach of estimating packet delays is not applicable to the WAN scenarios that we consider (refer Section I). None of these prior efforts isolate mice and elephant flow types, which is crucial for congestion control in the WAN (refer Section V-B).

[3] We could not evaluate RoCC since it is not publicly available.

VII. Discussion

NCF presents a quick (i.e., sub-RTT) and anticipatory congestion signaling through $\texttt{NCF}_{\texttt{tp}}$, and the rich telemetry data in the feedback enables a robust and stable response from the senders ($\texttt{NCF}_{\texttt{s1}}$). Below, we discuss a few key practical considerations for adopting NCF.

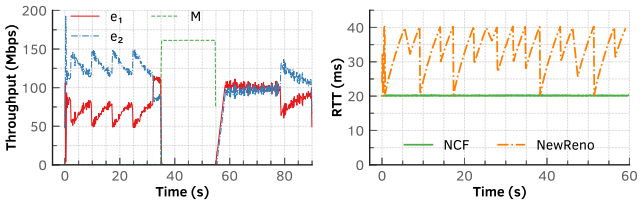
Protocol independence: NCF signal is not specific to any TCP variant. Although the prototype implementation uses some TCP flags (e.g., RST and FIN) for flow tracking, we can use simple timeouts for supporting other protocols (e.g., QUIC). The choice of a timeout value, however, has implications for flow-tracking accuracy and may depend on deployment scenarios. There is, hence, a clear need for experimentation with timeouts for flow-tracking.

Incremental deployment: We can safely and incrementally deploy NCF in the Internet: Not all routers on the path need to support NCF and our approach does not require receiver support. Since traffic volume is typically higher in the downstream direction, i.e., from content or cloud providers to users, than in the upstream direction (from the end users), content providers can incrementally deploy NCF at their edge to improve end-users' quality of experiences (QoEs) [16], [18]. Network operators today are investing in L4S for improving the performance of latency-sensitive applications; although L4S is orthogonal to NCF, these deployment locations are ripe candidates for NCF, and networks would immensely benefit by combining the two.

Control "knobs." The design and implementation of NCF offers guidelines for choosing the thresholds, queue sizes, and data structures, and our evaluations show that they work well in a range of conditions. Some of these parameters, however, might require "tweaks" over time. As an example, the choice of 12 for ρ, the threshold for identifying mice and elephant flow types assumes that the widely used initial-cwnd for TCP is 10 packets. As networks and applications evolve, network operators would benefit from sampling the traffic and changing this value based on their observations. Other protocols (e.g., QUIC) also warrant additional investigation into the choice of such parameters.

Spoofing $\texttt{NCF}_{\texttt{tp}}$. The ability to control a sender's rate via $\texttt{NCF}_{\texttt{tp}}$ raises a few security concerns, similar to that of "source quench" [30]. Such attacks require the attacker to guess the five-tuple identifier as well as the SEQ number. The attack is, hence, no different from generating fake, duplicate ACKs, or even RST packets—NCF does not per se introduce any new problems. The sender can validate $\texttt{NCF}_{\texttt{tp}}$ signals by correlating it with the receiver's feedback, i.e., the ACK stream. We generate $\texttt{NCF}_{\texttt{tp}}$ within the data plane, and they do not introduce a new switch-overload vulnerability.

Telemetry information leaks. The congestion signaling via $\texttt{NCF}_{\texttt{tp}}$ leaks information about the capacity (or lack thereof) of a network to other networks on the path traversed by the $\texttt{NCF}_{\texttt{tp}}$ signal towards the senders. Most traffic is, however, between their customers or customers of their peers: Information leakage should not be a major concern. Refraining from implementing NCF while your upstream and downstream peers deploy the solution might, in contrast, reflect poorly on the network: It clearly shows that any performance degradation is likely due to the network that is refraining from adopting NCF. We can also limit telemetry data, as required, at a given deployment, without losing all the benefits of NCF.



(a) *NewReno; RTT: 20 ms* (b) *Mice flow RTTs; RTT: 20 ms*

Fig. 20. NewReno: (a) a flood of mice flows seizes virtually all available bandwidth from the elephant flows, and (b) when recurring mice flows compete with elephant flows on a bottleneck, the lack of isolation between flow types causes the RTTs of mice flows to vary substantially.

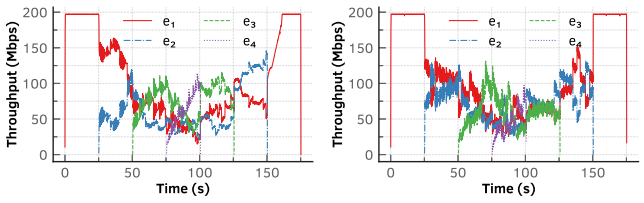
VIII. Conclusion

In this work, we presented the design, implementation, and evaluation of a rich, sub-RTT congestion signaling mechanism for use in the public Internet and demonstrated how a CCA or sender logic ($\texttt{NCF}_{\texttt{s1}}$) at the end hosts can exploit this feedback. We showed through rigorous evaluations how well our scheme fares against widely used CCAs in the Internet and how it equitably shares bandwidth with other flows in a range of network conditions. We demonstrated the generality of our approach by deploying it in datacenter-like environments and comparing its performance to Bolt and HPCC that are optimized for datacenters. We will release the data-plane logic ($\texttt{NCF}_{\texttt{dl}}$) for generating $\texttt{NCF}_{\texttt{tp}}$, sender logic ($\texttt{NCF}_{\texttt{s1}}$) or CCA, test-bed configurations and datasets as *open-source artifacts* upon publication of this work.

Our evaluations showed that NCF is robust and safe for deployment in the public Internet. The need to guarantee low-latency to applications is quite evident from network operators' interests in approaches such as L4S. Emerging applications are, however, highly demanding, than ever before [88], [89]: They are not demanding of low latencies, but also of high bandwidths, and these requirements might also vary over time, i.e., over the course of a data transfer or session. NCF's ability to isolate mice and elephant flow types dynamically would be crucial in meeting the stringent performance requirements of such applications. In today's Internet where CDNs, cloud, and content providers account for most of the Internet traffic, NCF, which only requires sender-side support, offers a new, practical, safe, and robust framework for experimenting with congestion control.

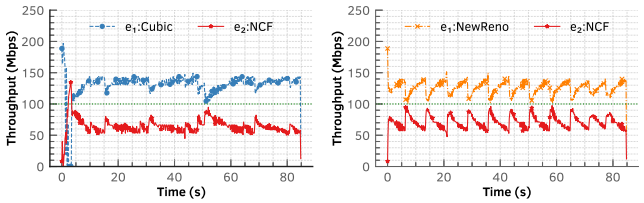
Appendix A
Flow-Type Isolation: NewReno vs. NCF

NewReno (similar to Cubic Fig. 7(a) and BBR Fig. 7(b)) fails to acquire a share of bandwidth when competing with a flood of mice saturating the link. The mice flows, in this experiment, constitute a flood of TCP handshakes. The queue is shared between all the flows, but the flood of mice flows are not amenable to congestion control. Fig. 20(a) illustrate



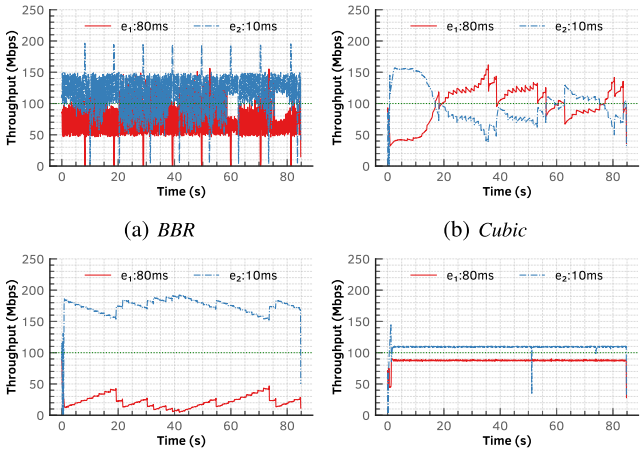
(a) *Cubic; RTT: 20 ms* (b) *NewReno; RTT: 20 ms*

Fig. 21. Fairness offered by (a) NewReno and (b) Cubic when we stagger the arrivals and departures of four flows at a bottleneck link of 200 Mbps by 25 s.



(a) *NCF vs Cubic RTT: 20 ms* (b) *NCF vs NewReno RTT: 20 ms*

Fig. 22. When NCF competes with (a) Cubic and (b) NewReno, it backs off quickly, resulting in the NCF flow achieving lower throughput than its fair share.



(a) *BBR* (b) *Cubic*

(c) *NewReno* (d) *NCF*

Fig. 23. Shallow buffers: dissimilar flows.

the harm that mice flood can inflict on two elephant flows of NewReno. Loss-based CCAs, e.g., NewReno and Cubic, rely on packet loss to detect congestion, which occurs when the queue overflows. Regardless of how the sender changes its cwnd (i.e., using an exponential, linear, or cubic function), the CCA will not notice the overload until a loss occurs. Besides, since loss is interpreted via three duplicate ACKs or a retransmission timeout, it will cause the host to take a long time (i.e., at least an RTT) to respond to congestion. In the absence of dedicated queues for the two flow types, mice flows could end up being head-of-line blocked by elephant flows: As a result, as shown in Fig. 20(b), they experience substantially large fluctuations in RTT, whereas they experience minimal RTTs (with virtually no fluctuations) when we use NCF.

Appendix B
Fairness Between Similar Flows: Cubic and NewReno

Both Cubic (Fig. 21(a)) and NewReno (Fig. 21(b)), similar to BBR in Fig. 10(a), struggle to maintain an equitable share in the staggered-flow experiment. The arrivals of new flows either causes other flows to experience substantial losses or the newly joining flow experiences packet losses well before it can ramp up to its fair share. NCF, in contrast, quickly sends rich feedback signals to senders and helps them converge to their fair shares despite new flow arrivals or departures (Fig. 10(b)), making it ideal for the public Internet.

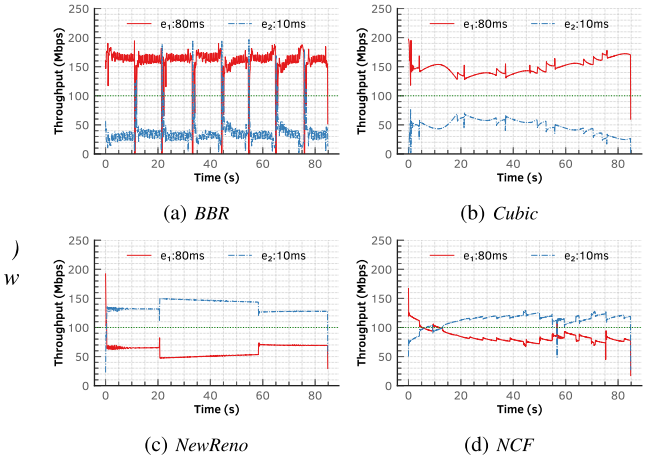
Appendix C
Dissimilar Flows: Throughput and Queue Occupancy

Inter-CCA: We use two flows each with a different CCA, and we make the second one to contend with the first at the bottleneck after 15 s. Both when competing with Cubic (Fig. 22(a)) and BBR (Fig. 22(b)), NCF backs off quickly leaving the other flows to gain more throughput than their fair shares. NCF's ability to react quickly to bottleneck bandwidth changes is evident, however, from it obtaining its fair share during BBR's RTT probing phases. Deploying NCF, hence, does not inflict substantial harm to existing CCAs—quite unlike, for instance, BBRv3 [90]. To ensure that the network or content provider deploying NCF obtain their fair share of bandwidth, they can tune the performance of NCF. Network operators are also fast deploying L4S, which if combined with NCF, will offer both existing CCAs and new NCF-based schemes a fair playground to contend. We plan to investigate such design in future work.

Multi-RTT: We now compare the behaviors of BBR, Cubic, and NewReno against that of NCF when dissimilar flows (i.e., experiencing different RTTs) contend for bandwidth under both shallow (Fig. 23) and deep buffer (Fig. 24) settings. In the shallow buffer (i.e., 1 BDP of 10ms RTT flow) setting, all CCAs perform quite poorly (Fig. 23). In case of BBR, throughputs fluctuate substantially; with Cubic one of the flows obtains most of the available bandwidth; and with NewReno the short-RTT flow dominates the long-RTT flow. Unlike these CCAs, NCF (Fig. 23(d)) allows the flows to acquire their fair shares without substantial throughput fluctuations. In the deep buffer setting (with the bottleneck set to 1 BDP of the long-RTT flow), BBR, Cubic, and NewReno flows do not ever converge to their fair shares, while NCF offers the flows to converge, achieving substantially high fairness than the rest of the CCAs (Fig. 24).

Appendix D
Elephant Flows in Multiple Bottleneck Settings

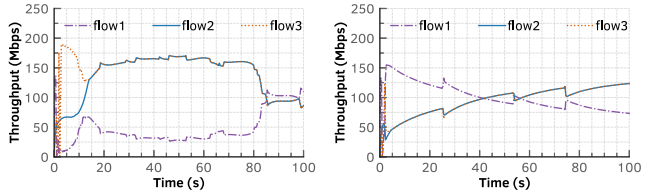
We run an experiment with three long-lived flows in a multiple-bottleneck scenario, similar to that in Fig 15(a), using Cubic and NewReno. We set the duration to 300s with a bottleneck link capacity of 200Mbps and 40ms RTT. The multi-bottlenecked Cubic flow ($flow_1$) only obtains a quarter of the bandwidth after slow-start (Fig. 25(a)). Despite the good



(a) *BBR* (b) *Cubic*

(c) *NewReno* (d) *NCF*

Fig. 24. Deep buffers: Dissimilar flows.



(a) *Cubic RTT: 40 ms* (b) *NewReno RTT: 40 ms*

Fig. 25. Throughput of all elephant flows.

start, NewReno (Fig. 25(b)), flows end up diverging from their fair shares, resulting in poor fairness.

REFERENCES

[1] J. Mo and J. Walrand, "Fair end-to-end window-based congestion control," *IEEE/ACM Trans. Netw.*, vol. 8, no. 5, pp. 556–567, Oct. 2000.

[2] L. Yu, J. Sonchack, and V. Liu, "Cebinae: Scalable in-network fairness augmentation," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2022, pp. 219–232.

[3] A. A. Philip, R. Ware, R. Athapathu, J. Sherry, and V. Sekar, "Revisiting TCP congestion control throughput models & fairness properties at scale," in *Proc. ACM Internet Meas. Conf. (IMC)*, 2021, pp. 96–103.

[4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *ACM Queue*, vol. 14, no. 5, pp. 58–65, Oct. 2016.

[5] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2010, pp. 63–74.

[6] M. Dong et al., "PCC Vivace: Online-learning congestion control," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 343–356.

[7] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proc. ACM Workshop Res. Enterprise Netw. (WREN)*, 2009, pp. 73–82.

[8] V. Arun, M. Alizadeh, and H. Balakrishnan, "Starvation in end-to-end congestion control," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2022, pp. 177–192.

[9] S. Arslan, Y. Li, G. Kumar, and N. Dukkipati, "Bolt: Sub-RTT congestion control for ultra-low latency," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2023, pp. 219–236.

[10] R. Zhou, D. Dong, S. Huang, and Y. Bai, "FastTune: Timely and precise congestion control in data center network," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl. Big Data Cloud Comput. Sustain. Comput. Commun. Social Comput. Netw. (ISPA/BDCloud/SocialCom/SustainCom)*, 2021, pp. 238–245.

[11] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2017, pp. 239–252.

[12] C. Lee, C. Park, K. Jang, S. Moon, and D. Han, "Accurate latency-based congestion feedback for datacenters," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Jul. 2015, pp. 403–415.

[13] Z. Meng et al., "Achieving consistent low latency for wireless real-time communications with the shortest control loop," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2022, pp. 193–206.

[14] P. Goyal, A. Agarwal, R. Netravali, M. Alizadeh, and H. Balakrishnan, "ABC: A simple explicit congestion controller for wireless networks," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Feb. 2020, pp. 353–372.

[15] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[16] E. Pujol, I. Poese, J. Zerwas, G. Smaragdakis, and A. Feldmann, "Steering hyper-giants' traffic at scale," in *Proc. ACM CoNEXT*, 2019, pp. 82–95.

[17] B. Chandrasekaran, G. Smaragdakis, A. Berger, M. Luckie, and K.-C. Ng, "A server-to-server view of the Internet," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2015, pp. 1–13.

[18] A. Saeed et al., "Annulus: A dual congestion control loop for datacenter and wan traffic aggregates," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2020, pp. 735–749.

[19] F. Wohlfart, N. Chatzis, C. Dabanoglu, G. Carle, and W. Willinger, "Leveraging interconnections for performance: The serving infrastructure of a large CDN," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2018, pp. 206–220.

[20] A. Gupta et al., "SDX: A software defined Internet exchange," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 551–562, 2014.

[21] I. Poese, B. Frank, G. Smaragdakis, S. Uhlig, A. Feldmann, and B. Maggs, "Enabling content-aware traffic engineering," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 21–28, 2012.

[22] A. Dhamdhere et al., "Inferring persistent interdomain congestion," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2018, pp. 1–15.

[23] A. Milolidakis, R. Fontugne, and X. Dimitropoulos, "Detecting network disruptions at colocation facilities," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 2161–2169.

[24] M. Luckie, A. Dhamdhere, D. Clark, B. Huffaker, and K. C. Claffy, "Challenges in inferring Internet interdomain congestion," in *Proc. ACM Internet Meas. Conf. (IMC)*, 2014, pp. 15–22.

[25] F.-Y. Ling, S.-L. Tang, M. Wu, Y.-X. Li, and H.-Y. Du, "Research on the net neutrality: The case of Comcast blocking," in *Proc. Int. Conf. Adv. Comput. Theory Eng. (IEEE ICACTE)*, 2010, pp. 488–491.

[26] J. Postel, "Internet control message protocol," DARPA Network Working Group, RFC 792, IETF, 1981.

[27] R. Pan and B. Prabhakar, "CHOKe: A stateless mechanism for providing quality of service in the Internet," in *Proc. Allerton Conf. Commun. Control Comput.*, vol. 37, 1999, pp. 1–10.

[28] F. Baker, "Requirements for IP version 4 routers," RFC 1812, IETF, Jun. 1995.

[29] F. Gont, "ICMP attacks against TCP," RFC 5927, IETF, Jul. 2010.

[30] F. Gont, "Deprecation of ICMP source quench messages," RFC 6633, IETF, May 2012.

[31] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2002, pp. 89–102.

[32] Y. Li et al., "HPCC: High precision congestion control," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2019, pp. 44–58.

[33] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, "Processor sharing flows in the Internet," in *Proc. Int. Workshop Qual. Service (IWQoS)*, 2005, pp. 271–285.

[34] H. Ohsaki, M. Murata, H. Suzuki, C. Ikeda, and H. Miyahara, "Rate-based congestion control for ATM networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 2, pp. 60–72, Apr. 1995.

[35] A. Feldmann, B. Chandrasekaran, S. Fathalli, and E. N. Weyulu, "P4-enabled network-assisted congestion feedback: A case for NACKs," in *Proc. Workshop Buffer Sizing*, 2019, pp. 1–7.

[36] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, "Increasing TCP's initial window," RFC 6928, IETF, Apr. 2013.

[37] J. Rüth, C. Bormann, and O. Hohlfeld, "Large-scale scanning of TCP's initial window," in *Proc. ACM Internet Meas. Conf. (IMC)*, 2017, pp. 304–310.

[38] C. Kenjiro, M. Koushirou, and K. Akira, "Traffic data repository at the wide project," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2000, pp. 1–10.

[39] Y. Joo, V. Ribeiro, A. Feldmann, A. C. Gilbert, and W. Willinger, "TCP/IP traffic dynamics and network performance: A lesson in workload modeling, flow control, and trace-driven simulations," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 2, pp. 25–37, Apr. 2001.

[40] L. Guo and I. Matta, "The war between mice and elephants," in *Proc. IEEE Int. Conf. Netw. Protocols (ICNP)*, 2001, pp. 180–188.

[41] M. P. Grosvenor et al., "Queues don't matter when you can JUMP them!" in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, May 2015, pp. 1–14.

[42] Z. Liu, H. Jin, Y.-C. Hu, and M. Bailey, "Practical proactive DDoS-attack mitigation via endpoint-driven in-network traffic control," in *Proc. IEEETON*, 2018, pp. 1–14.

[43] M. Alizadeh et al., "pFabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, 2013.

[44] L. L. Peterson, L. Brakmo, and B. S. Davie, *TCP Congestion Control: A Systems Approach*. Tucson AZ, USA: Systems Approach LLC, 2022.

[45] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. ACM SIGCOMM Softw. Defined Netw. Res. (SOSR)*, 2017, pp. 164–176.

[46] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[47] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2016, pp. 101–114.

[48] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "Qpipe: Quantiles sketch fully in the data plane," in *Proc. ACM CoNEXT*, 2019, pp. 285–291.

[49] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "SketchLib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 1–17.

[50] Y. Zhang et al., "CocoSketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2021, pp. 207–222.

[51] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2018, pp. 561–575.

[52] "Barefoot networks: Open Tofino," 2023. Accessed: Aug. 27, 2025. [Online]. Available: https://github.com/barefootnetworks/Open-Tofino/blob/ /include/traffic_mgr/traffic_mgr_q_intf.h

[53] M. Handley et al., "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol., Archit. Protocols Comput. Commun. (SIGCOMM)*, 2017, pp. 1–6.

[54] C. Raiciu and G. Antichi, "NDP: Rethinking datacenter networks and stacks two years after," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 5,. pp. 112–114, Oct. 2019.

[55] "Intel® tofino 6.4tbps, 4 pipelines," 2017. Accessed: Sep. 20, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/218643/intel-tofino-6-4-tbps-4-pipelines/specifications.html

[56] "Understanding the alpha parameter in the buffer configuration of Mellanox spectrum switches." NVIDIA. 2022. Accessed: Aug. 27, 2025. [Online]. Available: https://enterprise-support.nvidia.com/s/article/understanding-the-alpha-parameter-in-the-buffer-configuration-of-mellanox-spectrum-switches

[57] "NS-3 network simulator." Accessed: Sep. 20, 2023. [Online]. Available: https://www.nsnam.org/

[58] R. Ware, M. K. Mukerjee, S. Seshan, and J. Sherry, "Beyond Jain's fairness index: Setting the bar for the deployment of congestion control algorithms," in *Proc. SIGCOMM Workshop Hot Topics Netw. (HotNets)*, 2019, pp. 1–7.

[59] K. Sasaki, T. Hirofuchi, S. Yamaguchi, and R. Takano, "An accurate packet loss emulation on a DPDK-based network emulator," in *Proc. Asian Internet Eng. Conf. (AINTEC)*, 2019, pp. 1–9.

[60] F-Stack, "High performance network framework based on DPDK," 2017. Accessed: Sep. 20, 2023. [Online]. Available: http://www.f-stack.org,

[61] C. W. Parsonson, J. L. Benjamin, and G. Zervas, "Traffic generation for benchmarking data centre networks," *Opt. Switch. Netw. J.*, vol. 46, Nov. 2022, pp. 100695.

[62] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2015, pp. 123–137.

[63] J. Sommers, H. Kim, and P. Barford, "Harpoon: A flow-level traffic generator for router and network tests," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Sci.*, vol. 32, Jun. 2004, p. 392.

[64] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[65] R. Ware, M. Mukerjee, S. Seshan, and J. Sherry, "Modeling BBR's interactions with loss-based congestion control," in *Proc. ACM Internet Meas. Conf. (IMC)*, 2019, pp. 1–21.

[66] Y. Cao, A. Jain, K. Sharma, A. Balasubramanian, and A. Gandhi, "When to use and when not to use BBR: An empirical analysis and evaluation study," in *Proc. ACM Internet Meas. Conf. (IMC)*, 2019, pp. 1–14.

[67] P. Taheri, D. Menikkumbura, E. Vanini, S. Fahmy, P. Eugster, and T. Edsall, "ROCC: Robust congestion control for RDMA," in *Proc. ACM CoNEXT*, 2020, pp. 1–6.

[68] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 1–20.

[69] D. Hayes, D. Ros, L. L. Andrew, and S. Floyd, "Common TCP evaluation suite," Internet-Draft draft-irtf-iccrg-tcpeval-01, IETF, Fremont, CA, USA, Jul. 2014. [Online]. Available: https://datatracker.ietf.org/doc/draft-irtf-iccrg-tcpeval/01/

[70] S. Huang, D. Dong, and W. Bai, "Congestion control in high-speed lossless data center networks: A survey," *Future Gener. Comput. Syst.*, vol. 89, pp. 360–374, Dec. 2018.

[71] S. Alfredsson, G. Del Giudice, J. Garcia, A. Brunstrom, L. De Cicco, and S. Mascolo, "Impact of TCP congestion control on bufferbloat in cellular networks," in *Proc. Int. Symp. World Wireless, Mobile Multimedia Netw. (WoWMoM)*, 2013, pp. 1–7.

[72] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi, and B. Leong, "The great Internet TCP congestion control census," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, p. 45, 2019.

[73] A. Mishra, L. Rastogi, R. Joshi, and B. Leong, "Keeping an eye on congestion control in the wild with Nebby," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2024, pp. 136–150.

[74] R. Ware, A. A. Philip, N. Hungria, Y. Kothari, J. Sherry, and S. Seshan, "CCAnalyzer: An efficient and nearly-passive congestion control classifier," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2024, pp. 181–196.

[75] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for TCP," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 1996, pp. 270–280.

[76] K. K. Ramakrishnan and R. Jain, "A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 1988, pp. 151–181.

[77] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[78] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ECN) to IP," RFC 3168, IETF, Sep. 2001.

[79] K. Nichols and V. Jacobson, "Controlling queue delay," *Commun. ACM*, vol. 55, no. 7, pp. 42–50, Jul. 2012.

[80] B. Briscoe, K. D. Schepper, M. Bagnulo, and G. White, "Low latency, low loss, and scalable throughput (L4S) Internet service: Architecture," RFC 9330, IETF, Jan. 2023.

[81] K. D. Schepper, O. Tilmans, B. Briscoe, and V. Goel, "Prague congestion control," Internet-Draft draft-briscoe-iccrg-prague-congestion-control-03, IETF, Fremont, CA, USA, Jul. 2024.

[82] B. Briscoe, M. Kühlewind, and R. Scheffenegger, "More accurate explicit congestion notification (ECN) feedback in TCP," Internet-Draft draft-ietf-tcpm-accurate-ecn-34, IETF, Fremont, CA, USA, Jul. 2024.

[83] K. D. Schepper, B. Briscoe, and G. White, "Dual-queue coupled active queue management (AQM) for low latency, low loss, and scalable throughput (L4S)," RFC 9332, IETF, Jan. 2023.
[84] R. Mittal et al., "TIMELY: RTT-based congestion control for the datacenter," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2015, pp. 537–550.
[85] R. Pan, B. Prabhakar, and A. Laxmikantha. "QCN: Quantized congestion notification: IEEE 802.1," 2007. [Online]. Available: https://web.stanford.edu/balaji/presentations/au-prabhakar-qcn-description.pdf
[86] D. Zats et al., "FastLane: Making short flows shorter with agile drop notification," in *Proc. ACM Symp. Cloud Comput.*, 2015, pp. 84–96.
[87] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson, "Backpressure flow control," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Apr. 2022, pp. 1–4.
[88] L. Hsiao, B. Krajancich, P. Levis, G. Wetzstein, and K. Winstein, "Towards retina-quality VR video streaming: 15ms could save you 80% of your bandwidth," *ACM SIGCOMM Comput. Commun. Rev. (CCR)*, vol. 52, no. 1, pp. 10–19, Mar. 2022.
[89] S. S. Krishnan and R. K. Sitaraman, "Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs," in *Proc. ACM Internet Meas. Conf. (IMC)*, 2012, pp. 211–224.
[90] D. Zeynali, E. N. Weyulu, S. Fathalli, B. Chandrasekaran, and A. Feldmann, "BBRv3 in the public Internet: A boon or a bane?" in *Proc. Appl. Netw. Res. Workshop*, 2024, pp. 97–99.

Seifeddine Fathalli received the engineering degree in computer science from INSAT, Tunisia, in 2012, and the masters (M.Sc.) degree in computer engineering from the Technical University of Berlin in 2019. He is currently a Ph.D. candidate in Computer Science at the Max Planck Institute for Informatics and Saarland University, Saarbrücken, Germany. His research interests focuses on leveraging the data-plane programmability within software defined networks to solve challenges related to network performance and management.

Emilia N. Weyulu received the B.Sc. degree in computer science and statistics from the University of Namibia in 2011, the master's degree in informatics from the Tokyo University of Information Sciences in 2018, and the Ph.D. degree in computer science from the Max Planck Institute for Informatics and Saarland University, Saarbrücken, Germany, in 2025. Her research interests include the design and performance evaluation of transport protocols across diverse network environments. She is currently a Network Engineer with Cloudflare, Lisbon, Portugal.

Danesh Zeynali received the M.Sc. degree in computer networks from the Sharif University of Technology, Tehran, Iran. He is currently pursuing the Ph.D. degree in computer science with the Max Planck Institute for Informatics and Saarland University, Saarbrücken, Germany. His research focuses on congestion control and transport protocols, with an emphasis on systematic evaluation of state-of-the-art algorithms and the design of enhanced congestion feedback mechanisms.

Balakrishnan Chandrasekaran (Member, IEEE) received the Ph.D. degree in computer science from Duke University, Durham, NC, USA, in 2016. He was a Senior Researcher with the Max Planck Institute for Informatics, Saarbrücken, Germany. He is an Assistant Professor of Computer Science with Vrije Universiteit Amsterdam, The Netherlands. His research focuses on the performance and security aspects of networked systems.

Anja Feldmann received the degree in computer science from Universität Paderborn and Carnegie Mellon University. She is a Director with the Max Planck Institute for Informatics, Saarbrücken, Germany, where she leads the Internet Network Architectures Group. She previously held research and faculty positions with AT&T Labs Research, Saarland University, and the Technical University of Munich, and previously served as a Professor of Internet Network Architectures, Telekom Innovation Laboratories, Technische Universität Berlin. Her research focuses on Internet measurement, network architectures, and large-scale networked systems. She is an ACM Fellow for contributions to the data-driven analysis of operational networks and has received numerous prestigious awards, including the IEEE Koji Kobayashi Computers and Communications Award, the Konrad Zuse Medal of the German Informatics Society, the Gottfried Wilhelm Leibniz Prize, and the IETF/IRTF Applied Networking Research Prize. He is a Fellow of ACM.