TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control

Jörn-Thorben Hinz
TU Berlin

Vamsi Addanki
TU Berlin

Csaba Györgyi
University of Vienna

Theo Jepsen
Intel

Stefan Schmid
TU Berlin

ABSTRACT

For years, congestion control algorithms have been navigating in the dark, blind to the actual state of the network. They were limited to the course-grained signals that are visible from the OS kernel, which are measured locally (e.g., RTT) or hints of imminent congestion (e.g., packet loss and ECN). As applications and OSs are becoming ever more distributed, it is only natural that the kernel have visibility beyond the host, into the network fabric. Network switches already collect telemetry, but it has been impractical to export it for the end-host to react.

Although some telemetry-based solutions have been proposed, they require changes to the end-host, like custom hardware or new protocols and network stacks. We address the challenges of efficiency and protocol compatibility, showing that it is possible *and practical* to run telemetry-based congestion control algorithms in the kernel. We designed a framework that uses eBPF to run CCAs that can execute different control laws by selecting different types of telemetry. It can be deployed in brownfield environments, without requiring all switches be telemetry-enabled, or kernel recompilation at the end-hosts. When our eBPF program is deployed on hosts without hardware or OS changes, TCP incast workloads experience less queuing (thus lower latency), faster convergence and better fairness.

CCS CONCEPTS

• **Networks** → **Transport protocols**; *Data center networks.*

KEYWORDS

eBPF, Datacenter, TCP, INT, Congestion Control, Linux Kernel.

ACM Reference Format:
Jörn-Thorben Hinz, Vamsi Addanki, Csaba Györgyi, Theo Jepsen, and Stefan Schmid. 2023. TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control. In *1st Workshop on eBPF and Kernel Extensions (eBPF '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609021.3609295

eBPF '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0293-8/23/09…$15.00
https://doi.org/10.1145/3609021.3609295

1 INTRODUCTION

The volume of traffic in datacenters is increasing rapidly over time [6, 31, 33]. The throughput and latency offered by the underlying architecture and the set of protocols plays a critical role in the performance of modern cloud-based applications [26]. To this end, major research efforts over the past decade have been in two main domains: hardware offloading [2, 3, 20] and advanced congestion control [1, 18, 25, 26, 28, 42].

On the one hand, offloading computationally heavy tasks to hardware reduces software overheads but it comes at the cost of programmability and flexibility of the network stack, requiring specialized hardware such as RDMA (Remote Direct Memory Access) NICs. On the other hand, advanced congestion control offers immense benefits in-terms of throughput and latency, but it comes at the cost of bandwidth and computational overheads [12, 27]. These tradeoffs are clearly visible in today's datacenters, which rely on traditional TCP/IP for storage applications [19, 21]. Even in large-scale datacenters with RDMA capabilities, traditional TCP/IP traffic still accounts for up to 30% of the total traffic [5].

In this paper, we lay the groundwork for flexible and low-overhead telemetry-based congestion control algorithms (CCAs) in datacenter networks. Recent advancements in networking have finally made this possible *and practical.* The network data-plane is now programmable, both at the end-hosts and in the network fabric. Just the way the kernel network stack can be programmed with eBPF [16, 23, 37], switches in the fabric can be programmed with P4 [11, 14]. This opens opportunities for network-host signaling co-design. Concretely, exporting fine-grained telemetry from switches to the kernel provides end-host CCAs with rich information — not only the *location*, but also the *degree* of congestion – that can be used to make critical decisions more precisely and more rapidly.

CCAs base their decisions on the state of the network — or, more accurately, based on what they *believe* to be the state of the network. They learn the state of the network based on signals. Today's CCAs generally use signals that are inferred implicitly at the host, or explicitly from the network. These signals are multi-bit or single-bit (binary). Round trip time (RTT) is a multi-bit signal measured at the host, which infers the bottleneck state and intensity of congestion. Packet drops are a single-bit signal from the network indicating that congestion was encountered. Explicit Congestion Notification (ECN) is a single bit sent by the network fabric, indicating whether a packet passed through a congested link [17]. However, these signals are insufficient for precisely controlling modern datacenter traffic [1, 26].

The host-based signals are rich (multi-bit), but delayed. On the other hand, the network fabric-based signals are poor (single-bit),

but immediate. We want both: rich multi-bit signals received immediately from the fabric. This third option is now possible, thanks to advances in networking: In-band Network Telemetry (INT) [38, 39]. ***Our main contribution*** is to give the Linux network stack a "third eye", leveraging INT to see the true state of the fabric (transmission rate, queuing, latency, etc.), empowering end-host CCAs to make informed, immediate decisions.

We are not the first to propose using INT for CCAs. Although there have been proposals for integrating INT [26, 35, 36], they are *impractical*. They have high overheads and are not suitable for brownfield deployments, as they require specialized hardware (e.g., RDMA NICs [26]) and significant changes to the host networking stack [35]. Furthermore, they are not flexible—they are tied to a specific CCA.

To address these shortcomings, we developed a flexible telemetry-based CCA framework. It is built on eBPF and TCP in order to be compatible with existing protocols and networking stacks. The framework's flexibility is enabled by an extensible control model that can integrate various types of network telemetry. To access telemetry, we use TCP-INT [22, 41], a P4-based telemetry system that inserts telemetry (measured locally at the switch) into packets in such a way to minimize data overhead, as well as computational overheads at the host. In essence, our framework can be used to implement existing, as well as new advanced CCAs that can react to the changes in the network *quickly* and *precisely*.

To make such a system performant and practical, we faced several challenges. First, it must be suitable for brownfield deployments with different types of network hardware and software stacks. This requires compatibility with existing protocols (i.e. TCP) and integration into the host network stack (eBPF). Second, the existing approach of appending telemetry data to the packet header at *every hop* in the network increases the bandwidth overhead [26, 27]. One of our key observations is that the end-host CCA only needs information about the *most congested hop*, and hence information from every hop is unnecessary. Receiving telemetry from only the most congested hop reduces bandwidth overhead. Third, the existing approach of inserting telemetry (called "tagging") in *every packet* introduces computational overhead, as it interferes with other parts of the host networking stack [12]. This high tagging frequency hurts throughput without bringing any additional benefits for detecting changes in network conditions; ideally, the tagging frequency should provide a suitable tradeoff between throughput and timely telemetry.

Finally, to demonstrate the flexibility and efficiency of our framework, we use it to implement PowerTCP, a recent CCA that originally used RDMA and NIC offloading [1]. Experiments in our testbed show that our implementation achieves the desired fairness and queuing properties of PowerTCP. Furthermore, we evaluate the overheads and trade-offs of our framework. We discuss several exciting future directions and suggest some desired features and functionality from eBPF that would greatly improve the efficiency and accuracy of advanced CCAs in datacenters.

Overall, our main contributions are:

- A flexible CCA framework that uses telemetry in a novel control model.
- An exploration and evaluation of the trade-offs that make such a system *practical*.
- An eBPF implementation that runs the CCA model, and kernel patches that have already been accepted in Linux 6.0. Our source code is publicly available online at https://github.com/inet-tub/powertcp-linux.

2 BACKGROUND

For decades, the TCP/IP stack has ossified in both end-hosts as well as the network fabric (e.g., switches and routers). Deploying new functionality required major changes either in terms of re-compiling the kernel or changing the hardware altogether. Recent advancements have enabled the programmability of both end-host kernels as well as the network fabric. This presents the opportunity to innovate congestion control algorithms by unlocking the rich network telemetry that has always been available, but invisible, to the end-host.

2.1 eBPF for Congestion Control

Congestion control [29] is a fundamental and integral part of TCP. It ensures that network hosts do not overload the network by transmitting too much data too fast, leading to network congestion, packet loss and re-transmissions. Congestion control algorithms (CCAs) typically detect or infer changes in the network state, and control the sending rate by changing their congestion window (CWND) — the in-flight, un-acknowledged bytes.

When innovating CCAs, it is difficult to deploy in the wild, because it is non-trivial to integrate into existing network stacks, like that in the Linux kernel. CCP [30] proposes a framework for decoupling the CCA from the OS, making it easier to deploy new CCAs. However, this is not yet natively supported in the kernel.

Recently, the kernel added support for implementing CCAs in eBPF, through the `STRUCT_OPS` program type An eBPF CCA must satisfy the same interface as from a kernel module, providing the functions predefined in the `tcp_congestion_ops` structure[1]. The structure defines hooks that are invoked at multiple places in the control flow of Linux' network stack. Various events can trigger the hooks, including TCP state changes, arrival of ACKs, and detection of packet loss. The most important hook is responsible for updating the CWND, and optionally control a socket's pacing rate, after an ACK was received.

Historically, because CCAs don't run in the network fabric but at the hosts, they cannot know the actual state of the network. Instead, they infer the network state using signals like packet loss [34] or round-trip time (RTT) [13]. With a more recent signalling mechanism, Explicit Congestion Notification (ECN) [17], switches probabilistically set a bit in the IP header if queue occupancy exceeds a threshold. This single bit conveys some network state, but it is imprecise. It does not convey the *degree* of congestion. The host must receive many of these packets to calculate the ECN ratio, which slows down its reaction time.

Ideally, the switch would communicate its instantaneous send rate and queue occupancy [4]. Not only would this enable the hosts to detect changes in network conditions immediately, but it would also inform them *how aggressively* to react.

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/net/tcp.h?h=v6.1#n1066

2.2 In-band Network Telemetry

Network switches (including non-programmable) typically collect a wealth of information, including: queue occupancy, transmit counters, timestamps, etc., This information would be invaluable to the protocols at the end-hosts, e.g., to control sending rate by the congestion control algorithms. However, this information has been out of reach for a simple reason: the network was fixed-function. Switch vendors had to bake-in support for a myriad of protocols and processing logic. The lack of flexibility fundamentally limits the ability to export telemetry. The logic to export different types of telemetry for different types of protocols would have led to a combinatorial explosion.

With the advent of P4 [11], the network has become programmable. We can now program network switches, just the way one would write software. This programmability is what has made In-band Network Telemetry (INT) possible in early research efforts, without relying on the manufacturing capacity of major switch vendors.

There are various implementations of INT. For collecting telemetry on a packet travelling through the network, the INT spec [39] describes two modes of operation: *(i)* Postcards: upon receiving a packet, each switch sends a new "postcard" packet containing the telemetry to a collector (a "sink"), without modifying the original packet. *(ii)* Embedded: each switch appends its telemetry to the packet. The last hop receives the original packet, which includes a telemetry header from each switch along the path. While postcards may be useful for passive network monitoring, they are impractical for control. Not only does it have higher overhead—it generates many postcard packets, one for each hop—but it also presents operational challenges. It requires configuring the switch with the destination ("sink") of the postcards. Furthermore, the sink must correlate all these postcards with the actual data packet. Embedded INT, on the other hand, can deliver telemetry straight to the receiver in the same packet, which does not require correlation. Embedded INT is better suited for enabling the host to make immediate decisions.

However, there are still several problems:

- **Interoperability:** The INT spec prepends a new header format to the packet. This may break existing network stacks.
- **Data overhead:** Since the packet contains telemetry from *each* switch along the path, the packet size grows with the number of hops.
- **Compute overhead:** The switch inserts telemetry (*tagging*) in all packets. This causes increased overhead at the receiver, which must process telemetry for *every* packet. Furthermore, the receiver suffers additional compute overhead since it must parse each switch's telemetry (delivered by the packet) to aggregate them (e.g., to find the queue occupancy at the most congested switch).

Although other schemes have been proposed [7, 9, 35, 36], they do not address the aforementioned problems. We need an INT scheme that is inter-operable, efficient and that can be integrated into the Linux kernel.

2.3 INT-based Congestion Control

The idea of using INT for CCAs is not new. Some designs have been proposed, but they target different types of networks or do not address interoperability issues. Further, existing designs require specialized hardware at the end-host [26, 42], where the congestion control algorithm is runs in hardware, which is inflexible.

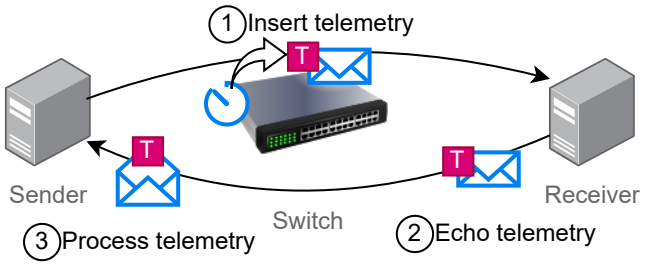


Figure 1: Overview of TCP-INT telemetry delivery.

HPCC [26] uses network telemetry to update the congestion window for RDMA over Converged Ethernet (RoCE). Specifically, HPCC updates its window size based on in-flight bytes in a multiplicative-increase and multiplicative-decrease manner (with a constant additive factor). The recent proposal PowerTCP [1] updates its window size based on the notion of power: the product of arrival rate and the amount of in-flight bytes. These protocols rely on INT to compute in-flight bytes and power. Their header format includes *per-hop* telemetry such as queue length, `tx bytes`, timestamps, etc. Since telemetry is inserted *per-hop*, data overhead as well as compute overhead increases. PINT [7] reduces overhead by aggregating INT (most congested hop), but it is not integrated into the kernel and requires protocol changes. Although many recent proposals use INT, these solutions are impractical as they are not well-integrated into the kernel. They require specialized network cards or protocols, and require changes to the network stack or to the application.

Given the recent advancements in programmability of the kernel as well as the network fabric, we set out with the goal to design a low overhead INT-powered CCA framework for the Linux kernel.

3 TCP'S THIRD EYE

At a high level, our system design can be divided into two parts: *(i)* TCP-INT [22, 41] "sees the network", delivering network telemetry to the kernel; and *(ii)* a CCA model running in eBPF that consumes the telemetry.

3.1 Seeing with TCP-INT

TCP-INT [22] was recently introduced to address the efficiency and compatibility problems of the above-mentioned previous INT solutions. With TCP-INT, switches insert INT into packets and an eBPF-based program at the end-hosts exports the received INT. This is a key enabler for us: we can now use the INT exposed by TCP-INT as inputs to telemetry-powered CCAs.

Figure 1 shows the workflow of INT delivery with TCP-INT. When switches receive a packet (1), they insert telemetry in the packet (which we refer to as *tagging*). Upon receiving a packet, the receiver extracts the telemetry. The receiver then 'echos' the telemetry back to the receiver (2) by inserting it in the next packet it sends (usually an ACK). The sender receives the 'echoed' telemetry (3), which shows the sender the state of the links on the data-path (i.e. its forward sending direction).

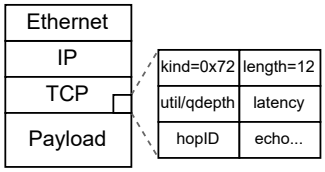


Figure 2: TCP-INT option in TCP header.

In order to maintain compatibility with TCP stacks (switches and hosts), TCP-INT carries INT data *in the TCP header*, using a TCP option [15]. The option format is shown in Figure 2. It contains telemetry fields that are updated by the switch, including queue depth, link utilization, and timestamps. As an optimization, the queue depth and utilization are encoded in the same field, indicating only one of the two (if there is queuing, the link is implicitly fully utilized). The echo fields are not touched by the switch; they contain the telemetry from the data path that is sent back from the receiver.

In order to reduce data and compute overheads, instead of storing the telemetry *per-hop*, the telemetry is *aggregated*. This way, the host only receives telemetry that it needs, from the *most* congested link along the path (a switch only tags the packet if its local congestion is greater than that encountered by the packet so far). The total size of the option is a mere 12 bytes, which is significantly smaller than the current approach of appending INT at *every-hop*.

Typically, INT systems tag all packets, which causes excess computational overhead at the receiver. Not only is the host INT processing logic invoked *for every packet*, but it also causes interference with the Linux network stack. Specifically, it 'breaks' generic receive offload (GRO) [12], which is responsible for coalescing multiple MTU-sized packets into a jumbo packet, and passing it up the stack for TCP processing. GRO only coalesces packets with matching headers (excluding sequence numbers). Because TCP-INT modifies the TCP header, each packet has a different header, so GRO will not coalesce them, but pass them up individually to the TCP stack. This causes a significant hit to performance, which could be avoided by tagging less frequently.

To reduce these computational overheads, TCP-INT tags *some* packets. The *tagratio* parameter controls the probabilistic frequency of tagging packets. E.g., *tagratio*= 1 tags every packet, while *tagratio*= 32, in expectation, tags one in every 32 packets. Furthermore, TCP-INT can do *dynamic* tagging, instead of using a fixed *tagratio*. The sender dynamically adjusts the *tagratio*, based on the sending rate. It finds the lowest frequency (to not hurt throughput) while ensuring timely delivery of INT.

In this work, we always use a static *tagratio*. We leave it to future work to extend our CCA implementation to dynamically change the *tagratio*. We evaluate the impact of the *tagratio* on throughput (see section 5), and provide initial insights on how it can be tuned.

3.2 Control Law Framework

We develop a control law framework that unites INT and congestion control in an eBPF program running in the Linux kernel. Our control law framework supports implementing CCAs within the generalization discussed in [1].

Our eBPF program implements a class of congestion control algorithms that update the window size based on multiplicative increase and multiplicative decrease (MIMD) with an additive increase factor. Specifically, we implement the generalized control law framework presented in [1]. It works as follows:

$$w_i(t+\delta) \leftarrow \gamma \cdot \left(w_i(t) \cdot \frac{e}{f(t)} + \beta_i \right) + (1-\gamma) \cdot w_i(t)$$

where $w_i(t)$ denotes the congestion window size at time t, β_i is a parameter which denotes the additive increase, γ is a parameter which denotes the smoothing factor. Here, e and $f(t)$ act as MIMD. e is the desired equilibrium which is set at initialization time and $f(t)$ is the feedback value measured and calculated by the algorithm. Various algorithms can be captured by our control law framework based on the choice of e and $f(t)$ as described in [1].

4 IMPLEMENTATION

We implemented our CCA model as an eBPF program that uses telemetry delivered by TCP-INT (see section 3) running with Tofino [14] P4 [11] switches. For the control model, we chose power as the input (i.e. $f(t)$) to embody the PowerTCP [1] CCA. In the process, we identified few shortcomings of eBPF, resolved them and submitted patches which have since been merged upstream.

4.1 eBPF Program

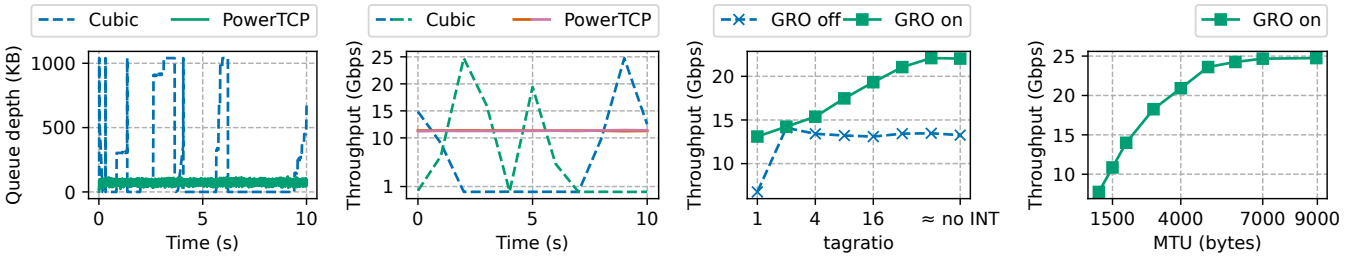
At a high-level, our eBPF program performs three main tasks: *(i)* extract INT from received packets; *(ii)* calculate necessary inputs for the specific control law (i.e. power for PowerTCP); and *(iii)* finally execute the control model and update the congestion window and pacing rate.

We use PowerTCP in the model running in our eBPF program. For PowerTCP, $f(t)$ is based on the notion of power, i.e. set to $(q(t) + b \cdot \tau) \cdot (\dot{q}(t) + \mu)$ where $q(t)$ is the queue length of the bottleneck link, $b \cdot \tau$ is the bandwidth-delay product, $\dot{q}(t)$ is the queue length gradient and μ is the bottleneck link utilization. We next describe the overall workflow of our framework.

Upon acknowledgment: The entry point to our workflow is the reception of an acknowledgment (ACK). Specifically, the kernel by default provides this entry point within the `tcp_ack()` function where it executes any function registered under `cong_control` (or `cong_avoid`) after processing the acknowledgement packet. We register a function `telemetry_cong_control()` in our eBPF program which triggers further processing of the INT data carried by the ACK.

Extracting INT values: For every incoming TCP segment, TCP-INT extracts the custom TCP option from the header if present. Telemetry values from the option are stored in an `SK_STORAGE` map to be retrieved by other eBPF or user space programs. Later, our congestion control framework reads the stored values with `bpf_sk_storage_get()` from the map.

Packet timestamps: CCAs within our control law framework require precise packet timestamps for calculating RTT. While kernel timestamps are readily available, we also support the use of hardware timestamps from the NIC. To do so, we accquire the timestamp value assigned to a packet (from its `struct sk_buff`). The last hardware timestamp received by a socket is stored in a designated `SK_STORAGE` map which we later use for our control law. Unfortunately, `tcp_congestion_ops` functions implemented



(a) PowerTCP maintains near-zero queuing, whereas Cubic has high queueing delays.

(b) PowerTCP fairly shares the bandwidth across two flows whereas Cubic remains unfair at small times scales.

(c) GRO significantly improves throughput (without INT) but GRO breaks under frequent INT i.e, as tagratio gets closer to 1.

(d) As the MTU size increases, the Kernel can cope with throughput even with frequent INT i.e., tagratio = 1.

Figure 3: Visibility into the network fabric via INT brings significant performance benefits for kernel TCP: low latency without losing throughput while maintaining fairness. GRO introduces interesting tradeoffs in our current implementation: packets carry INT within the TCP options which interferes with GRO batching.

in a kernel module have no direct access to `struct sk_buff` and cannot natively access hardware timestamps.

Calculating $f(t)$: We use INT values to compute $f(t)$ depending on the specific choice of $f(t)$. For instance, in the case of PowerTCP, we get $q(t)$ and μ from the INT values. We then use the time difference between two successive acknowledgements to calculate the gradient $\dot{q}(t)$. Finally, we calculate $f(t)$ and apply smoothing with an exponentially weighted moving average (EWMA). The smoothed value of $f(t)$ is then used for updating the window sizes.

Updating the window sizes and pacing rate: We update window sizes based on the control law described above. We then divide the updated window size with base RTT τ (a parameter) to set the pacing rate.

4.2 Kernel Patches

While working on the implementation, we identified two shortcomings in the kernel for eBPF-based CCAs:

- eBPF programs required defining both `cong_avoid()` as well as `cong_control()`, even though the former is not used in the network stack when the latter is defined.
- eBPF programs were not allowed to control the socket pacing rate, i.e. they did not have write access to `sk_pacing_rate` and `sk_pacing_status` in `struct sock`, even when implementing and registering `cong_control()`.

We submitted patches to the eBPF maintainers to resolve both shortcomings, along with minor bug fixes to bpftool and libbpf. Our patches were accepted and are included in the Linux kernel starting with version 6.0.

5 EVALUATION

In evaluating our framework, we have two main questions: *(i)* Does our telemetry-based CCA provide benefits, in terms of throughput, latency and fairness? *(ii)* Is there a performance overhead of the increased visibility (i.e. INT)? Our small-scale testbed consists of 3 hosts connected to a Tofino [14] switch. All the hosts are connected to the switch using 25 Gbps links. We deploy TCP-INT [22] and our eBPF-based CCA model on the hosts. We configure the CCA model to use PowerTCP [1]. We use Linux kernel version 6.1 at the hosts, following the general recommendations in the literature for performance tuning and have an average RTT of $\approx 80\mu s$ between the hosts.

5.1 Performance Benefits

Low latency without losing throughput: We generated an incast by sending two flows from different hosts to a common receiver host. Figure 3a shows that, at full throughput, PowerTCP keeps queuing close to zero. Cubic however, fills up the bottleneck queue, incurring long queuing delays.

Rapid convergence to fairness even at small timescales: First we run two Cubic flows, then two PowerTCP flows. In fig. 3b, we observe how the two flows share the bottleneck link bandwidth. We see that PowerTCP rapidly converges and sustains fairness at very small timescales. Since Cubic relies on loss as a congestion signal, it is unable to converge quickly and does not achieve fairness within this timescale.

5.2 Trade-offs

Given the performance benefits of using INT for congestion control in the Linux kernel, we take a deep dive into the overhead of the framework. Interestingly, we analyze tradeoffs in terms of the freshness and accuracy of INT vs throughput.

Throughput *vs* INT tagging frequency: With higher INT tagging frequency, i.e, with lower *tagratio*, the kernel receives fresh and more accurate information about the state of the bottleneck. To understand the overhead of just processing INT in the kernel, we run TCP-INT with Cubic. The kernel only receives and echos INT but does not use INT for CCA purposes. This allows us to study the overhead without the nuances of congestion control accuracy. In this case, as we see in fig. 3c, a lower *tagratio* significantly impacts the average throughput. As the *tagratio* increases, i.e. hosts receive INT less frequently, the throughput increases. This is caused by GRO, i.e. frequent tagging results in breaking batching. As we see from fig. 3c, at *tagratio* 1, the throughput with GRO enabled is close to the throughput achieved without GRO and INT disabled.

Throughput *vs* MTU size: At a constant throughput, the maximum transmitted size (MTU) plays a key role in kernel's packet

processing rate. As we saw in fig. 3c, which uses a small MTU of 1500, Cubic only comes close to maximum throughput when INT is less frequent. In fig. 3d, we run Cubic along with TCP-INT, but fix the *tagratio* to 1 i.e, every packet is tagged. We observe that as the MTU size increases, the throughput reaches close to full throughput even with *tagratio* 1. This indeed confirms that significant kernel overhead is due to batching (which is performed by GRO), as increasing the MTU reduces the required packet processing rate.

6 LOOKING INTO THE FUTURE

Our eBPF-based control law framework empowers kernel TCP with telemetry-based congestion control algorithms, offering significant performance benefits. We showed that it is possible to obtain the benefits of advanced congestion control without modifying the host stack or overhauling the network. By using a standard protocol (i.e. TCP), it is suitable for brownfield deployment where not all switches are telemetry-enabled. We hope that this will open avenues for widespread adoption of telemetry-based protocols (not just CCAs) at end-hosts, for example routing storage traffic [40], path load balancing [32] and flow scheduling [8]. With future support for offloading eBPF to hardware [10, 24], our telemetry-based CCA could even run directly in the NIC. We believe that standardizing the use of INT at lower-level protocols—like IP header options or a custom header (which are both not yet accessible by eBPF)—would make our framework accessible to protocols beyond TCP, while also alleviating overheads (such as breaking GRO). To this end, a corresponding feature support from the eBPF community would be valuable.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers of this paper for their useful feedback. This work is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, consolidator project Self-Adjusting Networks (AdjustNet), grant agreement No. 864228, Horizon 2020, 2020-2025.

REFERENCES

[1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. 2022. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 51–70. https://www.usenix.org/conference/nsdi22/presentation/addanki

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 93–109. https://www.usenix.org/conference/nsdi20/presentation/arashloo

[3] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. 2021. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21)*. Association for Computing Machinery, New York, NY, USA, 13–26. https://doi.org/10.1145/3482898.3483365

[4] Serhat Arslan and Nick McKeown. 2020. Switches Know the Exact Amount of Congestion. In *Proceedings of the 2019 Workshop on Buffer Sizing (BS '19)*. Association for Computing Machinery, New York, NY, USA, Article 10, 6 pages. https://doi.org/10.1145/3375235.3375245

[5] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering Azure Storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 49–67.

[6] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, et al. 2020. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the ACM SIGCOMM 2020 Conference*. 782–797.

[7] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 662–680. https://doi.org/10.1145/3387514.3405894

[8] Cristian Hetnandez Benet, Andreas Kassler, Gianni Antichi, Theophilus A. Benson, and Gergely Pongracz. 2021. Providing In-network Support to Coflow Scheduling. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 235–243. https://doi.org/10.1109/NetSoft51509.2021.9492530

[9] Ramyashree Venkatesh Bhat, Jetmir Haxhibeqiri, Ingrid Moerman, and Jeroen Hoebeke. 2021. Adaptive transport layer protocols using in-band network telemetry and eBPF. In *2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 241–246. https://ieeexplore.ieee.org/abstract/document/9606378

[10] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 973–990. https://www.usenix.org/conference/osdi20/presentation/brunella

[11] Mihai Budiu and Chris Dodd. 2017. The p416 programming language. *ACM SIGOPS Operating Systems Review* 51, 1 (2017), 5–14.

[12] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[13] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Commun. ACM* 60, 2 (2017), 58–66.

[14] Intel Corporation. 2020. Intel Tofino. (2020). Retrieved June 9, 2023 from https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[15] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. (Aug. 2022). https://doi.org/10.17487/RFC9293

[16] Matt Fleming. 2017. A thorough introduction to eBPF. (2017). https://lwn.net/Articles/740157/

[17] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. 2001. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. https://doi.org/10.17487/RFC3168

[18] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*. 29–42. https://doi.org/10.1145/3098822.3098825

[19] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 127–140. https://www.usenix.org/conference/nsdi20/presentation/hwang

[20] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 239–256. https://www.usenix.org/conference/osdi21/presentation/ibanez

[21] Dell Technologies Blog Ihab Tarazi. 2021. The Future of Software-defined Networking for Storage Connectivity . (2021). Retrieved June 9, 2023 from https://www.dell.com/en-us/blog/the-future-of-software-defined-networking-for-storage-connectivity/.

[22] Grzegorz Jereczek, Theo Jepsen, Simon Wass, Bimmy Pujari, Jerry Zhen, and Jeongkeun Lee. 2022. TCP-INT: Lightweight Network Telemetry with TCP Transport. In *Proceedings of the SIGCOMM '22 Poster and Demo Sessions (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 58–60. https://doi.org/10.1145/3546037.3546064

[23] The kernel development community. 2014. eBPF Instruction Set Specification, v1.0. (2014). https://www.kernel.org/doc/html/latest/bpf/instruction-set.html

[24] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev* 1 (2016).

[25] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*. 514–528. https://doi.org/10.1145/3387514.3406591

[26] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.

[27] Rui Miao, Bo Li, Hongqiang Harry Liu, and Ming Zhang. 2019. Buffer sizing with HPCC. (2019).

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*. 537–550. https://doi.org/10.1145/2785956.2787510

[29] John Nagle. 1984. Congestion Control in IP/TCP Internetworks. RFC 896. (Jan. 1984). https://doi.org/10.17487/RFC0896

[30] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 30–43. https://doi.org/10.1145/3230543.3230553

[31] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. 2022. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 66–85.

[32] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. 2022. PLB: Congestion Signals Are Simple and Effective for Network Load Balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/3544216.3544226

[33] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 123–137.

[34] Pasi Sarolahti and Alexey Kuznetsov. 2002. Congestion Control in Linux TCP. In *USENIX Annual Technical Conference, FREENIX Track*. 49–62. https://www.usenix.org/legacy/event/usenix02/tech/freenix/full_papers/sarolahti/sarolahti_html/

[35] Sandesh Dhawaskar Sathyanarayana, Max Hollingsworth, Wenji Wu, and Richard Cziva. 2022. Design, Implementation, and Evaluation of Host-based In-band Network Telemetry for TCP. In *2022 Global Information Infrastructure and Networking Symposium (GIIS)*. 62–67. https://doi.org/10.1109/GIIS56506.2022.9937001

[36] Siyuan Sheng, Qun Huang, and Patrick PC Lee. 2021. DeltaINT: Toward general in-band network telemetry with extremely low bandwidth overhead. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 1–11.

[37] Alexei Starovoitov and Daniel Borkmann. 2014. *Classic BPF vs eBPF*. https://www.kernel.org/doc/html/latest/bpf/classic_vs_extended.html

[38] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. 2021. In-band network telemetry: A survey. *Computer Networks* 186 (2021), 107763. https://doi.org/10.1016/j.comnet.2020.107763

[39] The P4.org Applications Working Group. 2020. In-band Network Telemetry (INT) Dataplane Specification. (2020). https://p4.org/p4-spec/docs/INT_v2_1.pdf

[40] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, Dai Zhang, Binzhang Fu, Frank Kelly, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. 2022. Predictable VFabric on Informative Data Plane. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 615–632. https://doi.org/10.1145/3544216.3544241

[41] Simon Wass and Jeongkeun Lee. 2022. TCP-INT: Intel's Lightweight Network Telemetry Improves Visibility and Control for TCP Workloads. (2022). Retrieved June 10, 2023 from https://medium.com/intel-tech/tcp-int-intels-lightweight-network-telemetry-improves-visibility-and-control-for-tcp-workloads-74c7c55910e.

[42] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.