Evaluating eBPF as a Platform for Congestion Control Algorithm

Edward Wibowo
Brown University

Bokai Bi
Brown University

Abstract

This paper seeks to explore the viability of developing Congestion Control Algorithms (CCAs) via eBPF. Developing new CCAs has traditionally been a time-consuming and error-prone process. Creating these algorithms directly within the Linux kernel demands a high level of kernel programming expertise while still leaving room for bugs that can crash the system. Although implementing CCAs as kernel modules has traditionally been an accepted alternative [10], it still runs the risk of introducing instability and crashes. Recent research has implemented a Congestion Control Plane (CCP) [12] that allows CCA developers to implement CCAs in user space, eliminating the need to run experimental software directly in the kernel. This approach involves a carefully designed and safety-checked datapath API; however, the Linux kernel datapath API still relies on a kernel module. Simultaneously, developing CCAs in eBPF [9] has emerged as another promising approach, leveraging the eBPF verifier to ensure safety before the code ever runs in the kernel. In this paper, we compare these various approaches – kernel-based, eBPF-based, and user space (via CCP) – along with a new hybrid user space eBPF-based approach (ebpfccp) in terms of behavioral correctness and performance overhead.

1 Introduction

TCP congestion control forms a critical part of the Internet ecosystem. It is an active area of research that involves new algorithms being proposed to improve network performance [7, 8] and discussions about how CCAs influence fairness between users [4, 11]. Hence, congestion control research is critical to the stable operation and continuous improvement of the Internet, even in a post-flow rate fairness era [5, 6].
A key part of congestion control research involves developing and experimenting with new CCAs. Traditionally, this poses two main problems:

1. **Development friction**: CCA implementations are often split between different datapaths. In the Linux kernel, this involves writing kernel code or modules, which presents a high barrier of entry and is prone to bugs that can crash the system.
2. **Distribution difficulties**: Pushing a new CCA upstream to the Linux kernel is difficult due to rigorous review processes and backward compatibility concerns. To see real-world adoption, a new CCA must be user-friendly and easy to adopt in the early stages. Being easy to adopt is often predicated on ensuring safety and performance guarantees.

To accurately evaluate the impacts of a new CCA on the Internet, it requires significant real-world adoption from users around the world. As a result, any CCA wishing to see eventual large-scale adoption must first propose a relatively user-friendly way for people to adopt it in the early stages.
With recent advancements in the eBPF space, the possibility of leveraging eBPF as a platform for implementing CCAs has become more apparent. eBPF [3] can be used to write and install programs that run in the Linux kernel with certain safety guarantees. Using the `struct_ops` API, congestion control-specific hooks can be attached to the TCP stack, allowing for the implementation of new CCAs. This approach has the potential to solve both the development friction and distribution difficulties of developing new CCAs. The eBPF verifier provides powerful guarantees to the safety of the program, protecting against large amounts of hard-to-solve bugs that may lead to kernel crashes. Furthermore, the runtime-attached nature of eBPF makes iterating and testing new versions much easier. On the adoption side, the safety guarantees provided by eBPF would greatly encourage user adoption. While users can be hesitant in installing kernel patches or modules, attaching a eBPF program can be much lower-risk and thus appealing.
However, running CCAs in eBPF can possibly incur extra costs due to running in the eBPF VM. By considering the performance overheads of various different ways of implementing CCAs, this paper explores the degree to which eBPF can be used as a platform for developing CCAs. In addition

to CCAs directly implemented using eBPF, other means of developing CCAs quickly and safely are also tested.

2 Related Work

In order to fully evaluate performance of eBPF-based CCAs against alternative approaches, we looked at prior research aimed at reducing CCA-development overhead. Specifically, CCP [12] was developed to be an abstraction layer where developers can experiment with new CCA ideas without worrying about the intricacies of specific datapath implementations. Most notably, CCP decoupled data gathering and CCA computation. A custom datapath module gathers TCP flow state and sends it to user space via various channels, such as netlink sockets and unix pipes. A user space program then listens to the provided data and sends back any updates to be made to the TCP connection state. Since the computation lives in user space, developers are shielded from the potential of introducing kernel-crashing bugs as long as the underlying CCP datapath implementation is safe.

Due to CCP's datapath-reliant nature, the performance of any CCP-based congestion control algorithm will depend on the implementation of the underlying datapath, as well as the channel used to communicate between user space and kernel space. In our investigations to test the performance of CCP-based CCAs, we built an alternative datapath module that is implemented as an eBPF program rather than a kernel module. In addition to providing more safety guarantees, this implementation also allows us to test how the performance of CCP-based CCAs can vary depending on the underlying datapath implementation.

3 Design Overview

Figure 1 illustrates the current CCP stack. Notably, the CCP algorithm and agent both run in user space, while the datapath is implemented and run completely in kernel space. Conventionally, CCP datapath program implementations such as Linux kernel's TCP stack, mTCP, and Google's QUIC all utilize `libccp` to interface with the CCP agent (Portus) [12].

3.1 eBPF Implementation

Due to eBPF's intense verifier requirements and stipulations, running `libccp` within the eBPF program was deemed difficult. Consequently, we create a user space layer between the CCP agent and the Linux kernel's TCP stack, called eBPFCCP.

eBPFCCP is written in Rust and runs `libccp` in user space to communicate with the CCP agent while also managing and communicating with the eBPF-based datapath program.

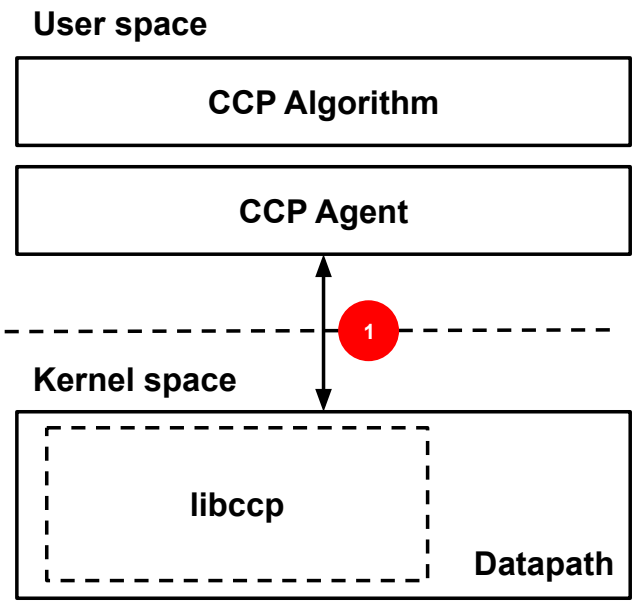

Figure 1: CCP's conventional Linux kernel datapath, implemented as a kernel module, utilizes `libccp` to communicate with the CCP agent. Communication (1) is accomplished via Netlink sockets or a custom character device.

eBPFCCP's role is to collect primitive congestion signals from the eBPF program, send them over to the CCP agent, and execute instructions sent by the CCP agent to influence TCP behavior. As shown in Figure 2, eBPFCCP communicates with the CCP agent via Unix-domain sockets as its IPC mechanism and interfaces with the eBPF datapath program through BPF hash maps and BPF ring buffers.

On the eBPF datapath side, we use eBPF's `struct_ops` API to attach congestion control-specific hooks to the TCP stack. Each of these hooks operate on a per-flow basis. When a new flow is created, the eBPF datapath program records the flow's information in a BPF ring buffer called `create_conn_events`. This signals eBPFCCP to retrieve the new flow's information and pass it to the CCP agent. Similarly, when a flow is terminated, the eBPF datapath program writes an event to the `free_conn_events` ring buffer, which eBPFCCP then reads to perform flow cleanup actions. As new ACKs arrive, the eBPF datapath program reads the TCP state and calculates primitive congestion signals, which are then written to the `signals` ring buffer, read by eBPFCCP, and sent to the CCP agent. Then, messages pertaining to congestion window size and sending rate are written to the `connections` hash map. The BPF maps used by the eBPF datapath and eBPFCCP are summarized in Table 1.

Overall, eBPFCCP aims to act as a bridge between the CCP agent and the eBPF datapath program.

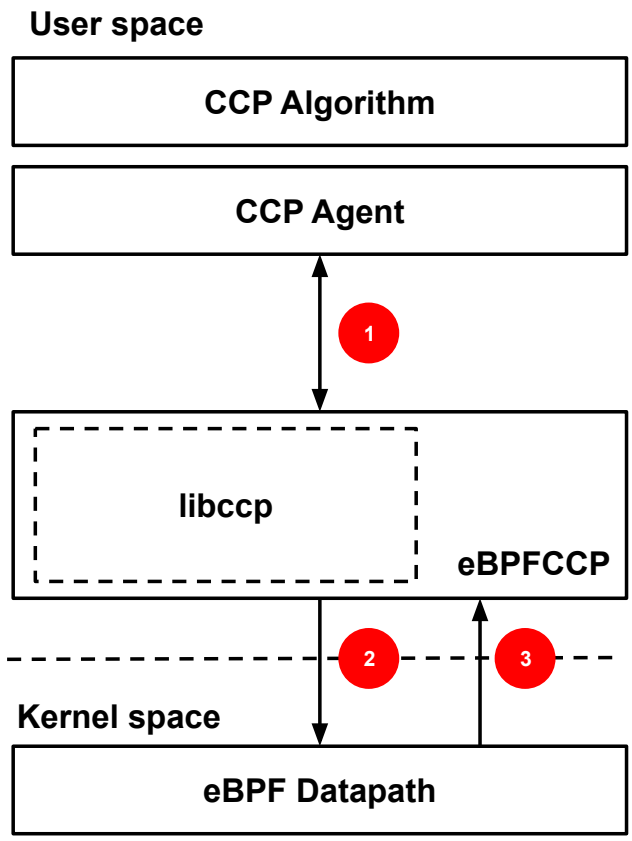

Figure 2: The eBPFCCP stack runs `libccp` in user space, communicating (1) with the CCP agent via Unix-domain sockets and interfacing with the eBPF datapath program via (2) BPF hash maps and (3) BPF ring buffers.

Map	Description
`connections` (BPF Hash)	eBPFCCP writes per-flow congestion window size and pacing rate. The eBPF datapath reads these values and updates the TCP state accordingly.
`signals` (BPF Ringbuf)	The eBPF datapath writes primitive signal measurements into this buffer. The eBPFCCP reads these measurements and forwards them to the CCP agent.
`create_conn_events` (BPF Ringbuf)	Whenever a new flow is created, the eBPF datapath records flow information here. The eBPFCCP then retrieves and passes these details to the CCP agent.
`free_conn_events` (BPF Ringbuf)	Upon flow termination, the eBPF datapath writes an event. The eBPFCCP reads and relays this information to the CCP agent for flow cleanup actions.

Table 1: BPF Maps used by the eBPF datapath and eBPFCCP for managing flow states, signals, and events.

4 Evaluation

4.1 Setup and Method

An alternative implementation of an existing CCA can be evaluated using two metrics. We evaluate eBPFCCP using two metrics:

1. **Behavior**: how accurate is an alternative implementation in replicating original CCA behavior?
2. **Performance**: what is the performance overhead of the alternative implementation?

We included two sets of benchmarks to measure this: (1) a behavioral benchmark aimed to verify the "correctness" of different implementations (TCP Cubic should behave similarly no matter where it is implemented, as indicated by metrics such as bitrate/congestion window size over time), and (2) a performance benchmark aimed to measure the CPU overhead of different implementation approaches. A CCA implementation with smaller CPU overhead would logically lead to a higher maximum bandwidth in an ideal, infinite-speed connection where the CPU running the CCA is the bottleneck. Our benchmarks utilize iperf3 [2]. For each trial, we loop through every CCA we are testing and run iperf3 for 15 seconds. At the same time, we run a Python script that loops infinitely while incrementing a variable. Once the 15 seconds are up, the script then outputs the final number into a file. A higher output number suggests that the script went through more CPU cycles during the trial, indicating better CPU performance. We ran a total of 20 trials.
After all the trials, we average the behavioral metric across the trials and plot the average for each CCA through time as the behavioral benchmark. We then plot the CPU performances for each CCA across trials on a boxplot.
To make sure we are correctly benchmarking the performance impact of CCAs rather than miscellneous confounding factors, we ran the benchmark on an out-of-the-box Debian image on a VM. To ensure the CPU benchmark and the CCA are running on the same core, our VM is set up to use only one core. We also decided to run our benchmarks on various implementations of TCP Cubic due to its wide availability across platforms.

4.2 Results

The results of the behavioral benchmarks are shown in Figures 3, 4, and 5. From the graphs, we can see that the behavior of the same Congestion Control Algorithm remains similar across congestion window size, bitrate, and RTT. The results of the CPU benchmarks are shown in Figure 6. The eBPF implementation of TCP Cubic achieved similar CPU performance compared to the built-in Linux kernel networking

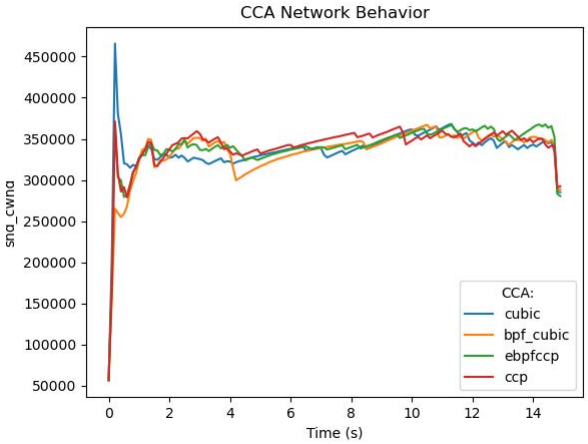

Figure 3: snd-cwnd behavior of TCP Cubic across different platforms of implementation. Averaged over 20 trials.

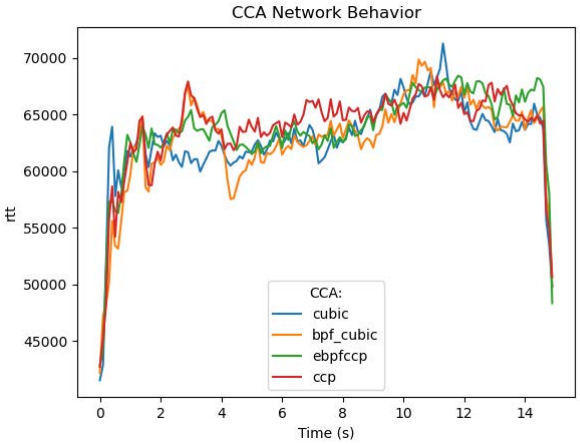

Figure 5: RTT behavior of TCP Cubic across different platforms of implementation. Averaged over 20 trials.

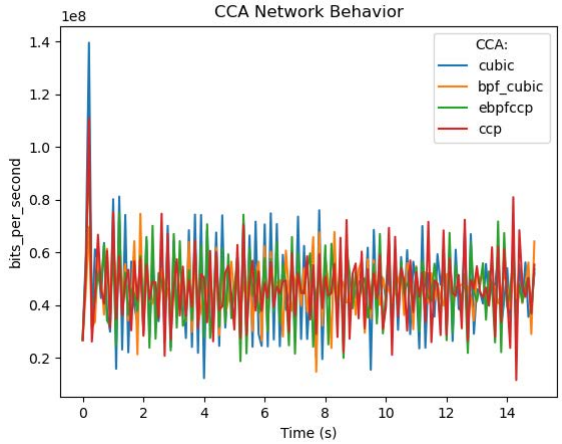

Figure 4: Bitrate behavior of TCP Cubic across different platforms of implementation. Averaged over 20 trials.

stack implementation. CCP-Cubic from the CCP Project [1] combined with the CCP datapath kernel module and netlink slightly lower performance, where its median is at 91 percent of the full kernel implementation value. Our implementation of eBPFCCP ranked lower on the performance chart, but was still able to achieve 80 percent of the full kernel implementation value. Due to the relative comparability of CPU performance between the kernel and eBPF implementations of TCP Cubic, we theorize the gap between eBPFCCP and kernel module CCP was due to our choice of using unix pipes rather than netlink as a channel for communicating between the eBPFCCP and the user space program.

From these results, we can conclude that eBPF is a viable platform for the development CCAs. The minor performance overhead when compared to the kernel native implementation suggests that eBPF-based CCAs can be used as a long-term deployment option outside of research and development stages. With regards to CCP, while the performance overhead of eBPFCCP is non-negligible, it should be noted that the benchmark is performed on a single core VM. In a real-world scenario, the performance overhead of eBPFCCP is likely to be negligible for all but the most performance-sensitive industry networking applications. The safety guarantees provided by eBPFCCP would provide a good intermediate step for users to adopt and experiment with a new CCA before it is natively implemented either directly into the kernel or as an eBPF program.

5 Limitations

One limitation that eBPFCCP faces is that it currently introduces another layer of indirection via BPF maps and ring buffers on top of the existing IPC mechanism. Future work could explore how to integrate eBPFCCP into the CCP agent, removing the need to communicate between two separate user space programs. We envision this work is made easier since both the user space component of eBPFCCP and Portus are written in Rust.

6 Conclusion

In this paper, we introduced an eBPF-based platform called eBPFCCP, which enables the implementation of user space CCAs. We then compared eBPFCCP with the native Linux kernel TCP stack, a pure eBPF implementation, and CCP. Our evaluations focused on the behavioral correctness and performance impact of these implementations. Through our behavior evaluations, we demonstrated that CCAs implemented

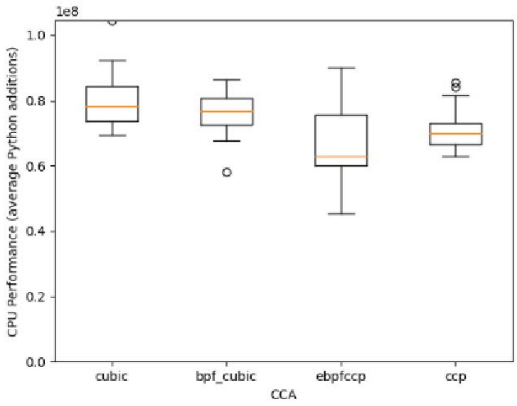

Figure 6: Standard box plot of TCP Cubic CPU performance across different platforms of implementation. Higher is better. Box extends from 25th to 75th percentile. Whiskers extend to the farthest data point within 1.5 IQR from the box. Data gathered over 20 trials.

using alternative platforms achieve the desired behavior. Additionally, we showed that eBPF is a suitable platform for developing CCAs natively, with negligible performance overhead compared to the kernel's native implementation. While our implementation of eBPFCCP had a slightly higher performance overhead, it still achieved a substantial percentage of the full kernel implementation's value. We believe that eBPFCCP can be valuable as an alternative CCP datapath that offers safety guarantees. Overall, the data presented here demonstrates that using eBPF-related CCA platforms not only provides numerous benefits, including enhanced development speed and safety, but also has the potential for long-term deployment beyond research stages.

7 Acknowledgements and Availability

We would like to thank Brown's CS2680 and Professor Akshay Narayan for the help provided in the development of this project. The source code is available at:
https://github.com/ebpfcca/ebpfcca

References

[1] Ccp algorithm: Generic congestion avoidance. https://github.com/ccp-project/generic-cong-avoid.

[2] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.

[3] Linux socket filtering aka berkeley packet filter (bpf). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[4] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 177–192, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Bob Briscoe. Flow rate fairness: dismantling a religion. *SIGCOMM Comput. Commun. Rev.*, 37(2):63–74, March 2007.

[6] Lloyd Brown, Albert Gran Alcoz, Frank Cangialosi, Akshay Narayan, Mohammad Alizadeh, Hari Balakrishnan, Eric Friedman, Ethan Katz-Bassett, Arvind Krishnamurthy, Michael Schapira, and Scott Shenker. Principles for internet congestion management. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 166–180, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, October 2016.

[8] Biyao Che, Yuxiang Wang, Zirui Wan, Ying Chen, Zixiao Wang, Yuan Tian, Jizhuang Zhao, Shuo Wang, and Jiao Zhang. Fcc : A fast-converging low-latency congestion control algorithm for datacenter rdma network. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, page 200–201, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Jörn-Thorben Hinz, Vamsi Addanki, Csaba Györgyi, Theo Jepsen, and Stefan Schmid. Tcp's third eye: Leveraging ebpf for telemetry-powered congestion control. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23. Association for Computing Machinery, 2023.

[10] Nathan Jay, Tomer Gilad, Nogah Frankel, Tong Meng, Brighten Godfrey, Michael Schapira, Jae Won Chung, Vikram Siwach, and Jamal Hadi Salim. A pcc-vivace kernel module for congestion control. 2018.

[11] Yang Ling, Zhang Xiao-fan, and Liu Yu-shan. Analyzing and improving the tcp flow fairness in 802.11 based ad hoc networks. In *2009 International Conference on Wireless Communications & Signal Processing*, pages 1–5, 2009.

[12] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *SIGCOMM*, 2018.