Reproducing Fidelity and Scalability of Congestion Control Plane Algorithms

Kevin Rosendahl
Stanford University
kedero@cs.stanford.edu

Robert Neff
Stanford University
rneff@cs.stanford.edu

ABSTRACT
Congestion control algorithms implementation and research continues to be important area of study as new ideas require extensive testing on increasing numbers of application-network stacks or “ datapaths” and seek to provide different functionality. This report presents our reproduction of Restructuring Endpoint Congestion Control by Narayan et al. [7], which explores placing algorithms in a user-space agent outside the datapath. In recreating Figure 10 we find that algorithms run in this mode match congestion window, throughput and delay fidelity of their kernel counterparts fairly accurately under a variety of simulated network conditions. We further replicate Figure 13, noting that multi-flow throughput and CPU utilization scale according to the paper and find that the additional time overhead costs are dominated by context switches. Finally, we briefly attempt to implement TCP Reno using their code for datapath setup [8].

1 INTRODUCTION
In Restructuring Endpoint Congestion Control by Narayan et al. the authors present the Congestion Control Plane (CCP), a framework and system for writing reusable congestion control algorithms that can be deployed to multiple datapaths (i.e. modules connecting applications to network hardware). It follows that re-implementation of congestion control algorithms in various different datapaths (ex. Linux kernel, QUIC and DPDK TCP implementations) becomes an easier task with the help of CCP. That is, developers can simply implement an algorithm once in user space via CCP and deploy anywhere.

1.1 Motivation
As previously mentioned, a dominant motivation behind CCP is in reducing time spent understanding new datapaths and redoing implementation work for them. Historically, most applications have used preexisting TCP congestion control implementations provided by their operating system kernel. Recently, however, alternatives such as protocols on top of UDP (ex. QUIC [6]) and kernel-bypass techniques (ex. DPDK [1]), among other strategies, have given applications more reliable transport options.

To handle this growing pool of data transmission and reception options, CCP creates a single interface through which datapaths and algorithms can communicate. Consequently, once a datapath is supported by CCP it can run any CCP-based algorithm without additional effort. To highlight the importance of this result Narayan et al. point out that newer datapaths such as mTCP (a DPDK TCP implementation) only support TCP Reno. Even QUIC, which is supported by Google, is missing implementations of many algorithms present in Linux kernel TCP. As more alternate datapaths emerge, this problem will only get worse without systems like CCP.

In addition to the so-called “write-once, run anywhere” [7] benefit, developers further profit through the ability to write algorithms in user space with modern languages such as Rust or Python. This not only leads to possible faster implementation pace, but also allows people to harness capabilities that are not readily available within kernel space. For example, algorithms can make use of signal processing and machine learning techniques that use operations and complexity not possible without floating point numbers or tied to the short-timescale ACK-clock.

1.2 Related Work
Kernel-bypass approaches allow applications running in user-mode to send packets on the NIC directly (i.e. without delegating with other parts of the stack). Given this, applications must then also implement their own congestion avoidance protocols, which limits algorithm availability in newer datapaths (ex. mTCP [5]).

The congestion manager (CM) [3] presents a datapath API which utilizes a kernel-agent to manage congestion control for flows. Given that CM control lives directly in the kernel, knowledge of the datapath is not easily extensible to additional algorithms like CCP is. Furthermore, CM only supports other kernel-based datapaths through its API which further limits its capabilities.

QUIC [6] builds congestion control on top of UDP sockets and provides a sender interface through its user-space library. This library-dependence means algorithms made for QUIC only work with QUIC. However, CCP can use QUIC’s congestion signal primitives to abstract away the datapath
specificity and have QUIC support generic CCP-based algorithms. Linux provides an analogous ‘plugable’ TCP API [4] that allows kernel modules to get/set various congestion signals on connection events. This pluggable format is essential to CCP’s ability to readily support a variety of datatypes, which we understand in exploring CCP’s structure next.

2 ORIGINAL CCP

2.1 Construction Overview

The CCP system is built upon two primary components: an agent that runs congestion control algorithms in user space and a datapath specific module (ex. a kernel API for interfacing with Linux kernel TCP and its congestion signals). The CCP-agent communicates data on behalf of the user algorithm to a datapath program and vice versa supported by the a shared library, libccp [7], which is common to all datatypes (and handles congestion signal callbacks).

CCP algorithm implementations consist of two event handlers, namely onCreate() and onReport(), which are registered with the CCP-agent. The onCreate() handler specifies a program written in a small domain specific language (DSL) that runs within the datapath, collecting connection statistics relevant to the congestion avoidance algorithms. When a new flow is created the CCP datapath component launches the DSL-specified (Lisp-like syntax) program, which uses user-defined fold-functions to accumulate statistics such as the number of bytes acknowledged or the occurrence of a timeout.

On specific conditions (defined by CCP-based algorithm), control is returned to user space, triggering the onReport() handler. This communication between the datapath component and the CCP-agent is performed via an IPC mechanism, which in turn invokes the onReport() handler and supplies the gathered statistics to the congestion algorithm. The user space algorithm can then employ this information to calculate and update desired congestion window and pacing rate properties. The datapath DSL is also capable of performing some simple actions (ex. slow start cwnd increment) to avoid expensive switches between the datapath and the CCP-agent.

3 REPRODUCTION

After presenting the CCP system, Narayan et al. evaluate both the CCP algorithm development pace and how well CCP algorithms reproduce the performance and behaviour of native Linux kernel (chosen due to being a more mature datatype for analysis) TCP algorithms.

Our reproduction repository is forked from that made for the original paper [8]. We use and extend this repository in our reproduction attempt (detailed in the subsections below).

3.1 Fidelity

In evaluating how closely CCP-based algorithms behave to their datapath counterparts, Narayan et al. compare achieved throughput, delay and flow completion times (not replicated here) for Linux kernel TCP NewReno and Cubic against the same respective CCP algorithms (implemented by paper authors and assumed by our test to be correct). They further plot congestion window update decisions made by each TCP Cubic algorithm from their throughput/delay tests.

They assess throughput and delay performance using Mahimahi [9] to generate three scenarios (all with a single flow):

- Fixed: a fixed 96Mb/s rate link using 1 bandwidth delay product of buffering.
- Drop: a fixed 96Mb/s rate link using infinite buffering and 0.01% stochastic loss.
- Cell: a Verizon LTE trace with 100 packet buffering.

These scenarios all use 20ms RTTs and present different network situations, i.e. predictable, bufferbloated and mobile. We see their original distribution plots in Figure 1.

Our reproduction was performed using an Ubuntu 17.10 VirtualBox [10] VM (with four 4.0GHz cores and 8GB of memory), per their repository, as certain newer Linux kernels do not support tcprobe, which is needed to collect data. We use less memory than the original tests due to machine limitations and the observation that single flow fidelity tests rarely use more than 1GB of memory.

We run the same simulated scenarios, each run for ten 30-second experiments and with per-RTT CCP algorithm reporting. Our results can be seen in Figure 2. While the distributions are similar in most cases, the original and presented TCP Cubic drop scenarios deviate significantly from one another. In investigating this result, we find that the CCP algorithm stays in the slow-start phase much longer than the kernel version. The congestion window sizes used by the CCP implementation are then higher on average and incorrectly approximate network conditions leading to lower throughput and more delay (see Figure 3). As we use slightly shorter test periods than the original paper, this issue is likely more pronounced, as without timeouts slow-start only happens at the beginning of a connection.

We note that this drop case is deterministic such that loss happens at the same time for the kernel and CCP algorithms (for better comparison). The congestion window decisions made by the CCP algorithm also become deterministic with this condition, however the kernel decisions are not. Perhaps variations in kernel entropy (that are disconnected from CCP kernel module) are the contributors to differences here.

Finally, in looking at the congestion window decisions for various iterations of our fixed scenario test for TCP Cubic, we notice that the kernel and CCP curves do not always overlap.
as well as in the original paper (see Appendix Figures 9 and 10). That is, window updates by CCP are often delayed or occasionally missed, causing the kernel and CCP algorithms to become out of sync. However, the difference between the top and bottom of the saw-tooth is consistent between kernel and CCP, leading to the same throughput on a fixed rate link. A potential cause for this discrepancy may come from randomization in the simulation inherent to the scenario.

We performed limited testing with a TMobile LTE bandwidth trace and additional rates/delays/queue-sizes for each scenario but did not find anything outstanding. As there are many variable parameters for the test cases presented in this section, we feel more extensive testing would yield insight as to where/why CCP algorithms differ from their Linux TCP kernel counterparts.

3.2 Scalability

In addition to testing the fidelity results presented by Narayan et al., we sought to investigate the scalability and performance of the CCP system. The authors investigated both maximum achievable localhost throughput, as well as the CPU utilization while saturating a 10 Gb/s link. They measured and compared the performance of CCP against the kernel while varying the number of flows.

Our reproductions were conducted on GCP VMs configured with 8 virtual cores and 64 GB of memory, which allows for a maximum of 16 Gb/s egress bandwidth.

3.2.1 Localhost Throughput. The authors ran a localhost microbenchmark, measuring achieved throughput on the loopback interface. We reproduced the benchmark, as can be seen in Figure 5. While our absolute numbers differ, this can be expected. The overall shape of the charts match up well. For the most part, running CCP with a 10ms reporting interval is able to achieve the throughput of its kernel counterpart. Unlike the authors, we find that the per-ACK reporting interval seems to reliably achieve less throughput than the 10ms reporting interval or kernel implementations.

3.2.2 CPU Utilization. Narayan et al. also measured the CPU utilization of CCP against the Linux kernel. In their evaluation they claim that "the difference between CCP and
the kernel is most pronounced in the region between 16 and 64 flows, where CCP uses 2.0× as much CPU than the kernel on average, although for 16 flows their graph seems to suggest that CCP averaged over 7.5% CPU utilization versus the Linux kernel’s 2.5%, which is a 3.0× factor difference. Additionally, they note that CPU utilization remains below 8% in their tests.

In our reproduction of CPU utilization scalability, we found that the relative factor of CCP vs kernel utilization was similar to the results found in the paper, but the absolute CPU utilization was higher. This could be explained by a different CPU/bandwidth ratio between our GCP instances and the system used by the authors. We find that that for all number of flows, CCP uses more CPU than its kernel counterpart, with the difference growing with the number of flows added (see Figures 6 and 7).

The authors also note that “the differences in CPU utilization stem from the necessarily greater number of context switches as more flows send measurements to CCP.” This claim is mostly substantiated by our investigation of the breakdown of CPU utilization in Figure 8, however we do note a noticeable difference in the amount of user CPU utilization when using CCP compared to the Linux kernel.

### 3.3 Ease of Development

A key design decision for CCP was providing a good abstraction with which developers and designers can write congestion control algorithms without worrying about underlying architecture. This allows for easier development/deployment, maintenance and updates and in providing new capabilities with aggregated data. To drive this point home, Narayan et al. claim the authors of the Copa [2] (a model-based congestion algorithm) were both able to re-implement Copa using CCP and find a bug that greatly hindered the throughput of their prior UDP-based implementation, quickly fixing it thereafter.

In working with CCP we briefly examined how the original paper re-implemented TCP NewReno and Cubic in Rust for CPP. From this experience, we attempted to re-implement TCP NewReno and Vegas (based off the Linux kernel versions) in Python using Narayan et al.’s Portus [7] package. We were quickly able to test a basic rendition NewReno that matched the kernel in the fixed rate tests (see Figure 11 as an example). On the flip side, we were largely unsuccessful in putting together a passable Vegas implementation. Given
the inherent complexity of both following congestion control specifications and mimicking kernel semantics, we were not surprised by this outcome. We can however recommend CCP’s event-driven language and use of congestion signal primitives, as programming was simple and intuitive.

4 EVALUATION

Overall, our reproduction results are fairly consistent with those demonstrated in Restructuring Endpoint Congestion Control. Despite various efforts to break fidelity or cause performance overheads to be exceedingly high, we were largely unsuccessful in finding any severe inconsistencies and flaws in the system. We do however find a few peculiarities that manifest under specific test conditions and so cause deviations from the expected outcome. In section 3 we reason about the root causes behind these issues, yet deeper understanding of the CCP system and additional testing would further clarify our conclusions.

While we don’t dispute the results presented by Narayan et al., we do note that they do not make an effort to give context for how the presented results should be evaluated. If the goal of CCP is simply to prove the possibility of implementing such a system, we have shown that they have succeeded in doing so.

If CCP is intended to be a tool for prototyping and researching congestion control algorithms (similar to Pantheon [11]), then the fact that the fidelity experiments were not quite perfect would require more scrutiny.

If CCP is intended to actually be deployed, questions such as the following would need to be addressed:

- Is it feasible to install CCP into the datapath and run the agent? For example, in a containerized environment many users may share the same Linux kernel, potentially rendering it impossible for each tenant to choose their algorithm implementation.
- Is the CPU overhead acceptable for the targeted applications? Can it be reduced?
- Is there an impedance mismatch between the datapath and CCP? For example, mTCP is designed to be used in performance sensitive applications that want to avoid expensive context switches and time in the kernel. Would adding CCP negate some of these benefits?

5 FUTURE WORK

Long-term work on this project would lead us to dig deeper into the plethora of test options we could not address. Some of these areas include:

- Testing network environment parameters such as link delay, flow count or loss rate.
- Explore CCP algorithm reporting periods (ex. per-ACK, per-RTT, custom) and the trade-offs associated with fidelity and performance overhead under assorted circumstances.
- Try more congestion control algorithms (ex. BBR, Copa) that are implemented in CCP (or can easily be added). This would let us gauge how well CCP supports all algorithms types, especially those dubbed as offering new capabilities that were not present in our tests.
- Install the CCP module on alternative datapaths (like QUIC and mTCP) to validate it works as intended.
- Assess more closely where specifically system overhead is incurred by CCP via a modern tracing tool (ex. KUTrace). This could lead to potential optimizations and verify the scalability concerns of the original paper.

6 REPRODUCTION CODE

To reproduce our results or view further plots and data go to: https://github.com/kevindrosendahl/eval-scripts/tree/master/reproduction.

7 APPENDIX

7.1 Congestion Window Fidelity

Figure 9: Original TCP Cubic congestion window update decisions for fixed rate link test case (with per-RTT reporting and 20ms RTT). This is also Figure 9 in the original paper.

Figure 10: Reproduced example of TCP Cubic congestion window update decisions for fixed rate link test case (with per-RTT reporting and 20ms RTT).
REFERENCES