The version of this paper published at SIGCOMM 2018 contained a labelling error in Figure 10. This has been corrected in this version.

Restructuring Endpoint Congestion Control

Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal
Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, Hari Balakrishnan
ccp@csail.mit.edu
MIT Computer Science and Artificial Intelligence Laboratory

Abstract

This paper describes the implementation and evaluation of a system to implement complex congestion control functions by placing them in a separate agent outside the datapath. Each datapath—such as the Linux kernel TCP, UDP-based QUIC, or kernel-bypass transports like mTCP-on-DPDK—summarizes information about packet round-trip times, receptions, losses, and ECN via a well-defined interface to algorithms running in the off-datapath Congestion Control Plane (CCP). The algorithms use this information to control the datapath's congestion window or pacing rate. Algorithms written in CCP can run on multiple datapaths. CCP improves both the pace of development and ease of maintenance of congestion control algorithms by providing better, modular abstractions, and supports aggregation capabilities of the Congestion Manager, all with one-time changes to datapaths. CCP also enables new capabilities, such as Copa in Linux TCP, several algorithms running on QUIC and mTCP/DPDK, and the use of signal processing algorithms to detect whether cross-traffic is ACK-clocked. Experiments with our user-level Linux CCP implementation show that CCP algorithms behave similarly to kernel algorithms, and incur modest CPU overhead of a few percent.

CCS Concepts

• **Networks** → **Transport protocols**; *Network protocol design*;

Keywords

Congestion control, Operating systems

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230553

1 Introduction

At its core, a congestion control protocol determines when each segment of data must be sent. Because a natural place to make this decision is within the transport layer, congestion control today is tightly woven into kernel TCP software and runs independently for each TCP connection.

This design has three shortcomings. First, many modern proposals use techniques such as Bayesian forecasts (Sprout [41]), offline or online learning (Remy [40], PCC [11], PCC-Vivace [12], Indigo [43]), or signal processing with Fourier transforms (Nimbus [19]) that are difficult, if not impossible, to implement in a kernel lacking useful libraries for the required calculations. For example, computing the cube root function in Linux's Cubic implementation requires using a table lookup and a Newton-Raphson iteration instead of a simple function call. Moreover, to meet tight performance constraints, in-kernel congestion control methods have largely been restricted to simple window or rate arithmetic.

Second, the kernel TCP stack is but one example of a *datapath*, the term we use for any module that provides data transmission and reception interfaces between higher-layer applications and lower-layer network hardware. Recently, new datapaths have emerged, including user-space protocols atop UDP (e.g., QUIC [25], WebRTC [24], Mosh [39]), kernel-bypass methods (e.g., mTCP/DPDK [13, 23, 33]), RDMA [45], multi-path TCP (MPTCP) [42], and specialized Network Interface Cards ("SmartNICs" [28]). This trend suggests that future applications will use datapaths different from traditional kernel-supported TCP connections.

New datapaths offer limited choices for congestion control because implementing these algorithms correctly takes considerable time and effort. We believe this significantly hinders experimentation and innovation both in the datapaths and the congestion control algorithms running over them. For instance, the set of available algorithms in mTCP [23], a TCP implementation on DPDK, is limited to a variant of Reno. QUIC, despite Google's imposing engineering resources, does not have implementations of several algorithms that have existed in the Linux kernel for many years. We expect this situation to worsen with the emergence of new hardware accelerators and programmable network interface cards (NICs) because high-speed hardware designers tend to forego programming convenience for performance.

Third, tying congestion control tightly to the datapath makes it hard to provide new capabilities, such as aggregating congestion information across flows that share common bottlenecks, as proposed in the Congestion Manager project [4].

If, instead, the datapath encapsulated the information available to it about *congestion signals* like packet round-trip times (RTT), receptions, losses, ECN, etc., and periodically provided this information to an off-datapath module, then congestion control algorithms could run in the context of that module. By exposing an analogous interface to control transmission parameters such as the window size, pacing rate, and transmission pattern, the datapath could transmit data according to the policies specified by the off-datapath congestion control algorithm. Of course, the datapath must be modified to expose such an interface, but this effort needs to be undertaken only once for each datapath.

We use the term *Congestion Control Plane (CCP)* to refer to this off-datapath module. Running congestion control in the CCP offers the following benefits:

(1) **Write-once, run-anywhere:** One can write a congestion control algorithm once and run it on any datapath that supports the specified interface. We describe several algorithms running on three datapaths: the Linux kernel, mTCP/DPDK, and QUIC, and show algorithms running for the first time on certain datapaths (e.g., Cubic on mTCP/DPDK and Copa on QUIC).
(2) **Higher pace of development:** With good abstractions, a congestion control designer can focus on the algorithmic essentials without worrying about the details and data structures of the datapath. The resulting code is easier to read and maintain. In our implementation, congestion control algorithms in CCP are written in Rust or Python and run in user space.
(3) **New capabilities:** CCP makes it easier to provide new capabilities, such as aggregate control of multiple flows [4], and algorithms that require sophisticated computation (e.g., signal processing, machine learning, etc.) running in user-space programming environments.

This paper's contributions include:

- An event-driven language to specify congestion control algorithms. Algorithm developers specify congestion control behavior using combinations of events and conditions, such as the receipt of an ACK or a loss event, along with corresponding handlers to perform simple computations directly in the datapath (e.g., increment the window) or defer complex logic to a user-space component. We show how to implement several recently proposed algorithms and also congestion-manager aggregation.
- A specification of datapath responsibilities. These include congestion signals that a datapath should maintain (Table 2), as well as a simple framework to execute directives from a CCP program. This design enables "write-once, run-anywhere" protocols.
- An evaluation of the fidelity of CCP relative to in-kernel implementations under a variety of link conditions. Our CCP implementation matches the performance of Linux kernel implementations at only a small overhead (5% higher CPU utilization in the worst case).

2 Related Work

The Congestion Manager (CM [4]) proposed a kernel module to separate congestion control from individual flows. CM provides an API for flows to govern their transmissions and a plan to aggregate congestion information across flows believed to share a bottleneck. The CM API requires a flow to inform the CM whenever it wanted to send data; at some point in the future, the CM will issue a callback to the flow granting it permission to send a specified amount of data. Unlike CCP, the CM architecture does not support non-kernel datapaths or allow custom congestion control algorithms. Further, the performance of CM is sub-optimal if the CM and the datapath are in different address spaces, since each permission grant (typically on each new ACK) requires a context switch which reduces throughput and increases latency. We show in §6.3 that CCP can support the aggregate congestion control capabilities of the CM architecture.

eBPF [14] allows developers to define programs that can be safely executed in the Linux kernel. These programs can be compiled just-in-time (JIT) and attached to kernel functions for debugging. TCP BPF [6] is an extension to eBPF that allows matching on flow metadata (i.e., 4-tuple) to customize TCP connection settings, such as the TCP buffer size or SYN RTO. In the kernel datapath, it may be possible for CCP to use the JIT features of eBPF to gather measurements, but not (yet) to set rates and congestion windows. Exploring the possibility of TCP control entrypoints for eBPF, and an implementation of a Linux kernel datapath for CCP based on such control, is left for future work.

Linux includes a pluggable TCP API [10], which exposes various statistics for every connection, including delay, rates averaged over the past RTT, ECN information, timeouts, and packet loss. icTCP [20] is a modified TCP stack in the Linux kernel that allows user-space programs to modify specific TCP-related variables, such as the congestion window, slow start threshold, receive window size, and retransmission timeout. QUIC [25] also offers pluggable congestion control. We use these Linux and QUIC pluggable APIs to implement datapath support for CCP. CCP's API draws from them, but emphasizes asynchronous control over datapaths.

HotCocoa [2] introduces a domain specific language to allow developers to compile congestion control algorithms directly into programmable NICs to increase efficiency in packet processing. In contrast, CCP allows developers to write algorithms

in user-space with the full benefit of libraries and conveniences such as floating point operations (e.g., for Fourier transforms).

Structured Streams (SST [17]) proposed a datapath that prevents head-of-line blocking among packets of applications by managing the transport streams between a given pair of hosts and applying a hereditary structure on the streams. Unlike SST, CCP does not manage the contents of the underlying transport stream: CCP enables deciding when a packet is transmitted, not which packet. We view SST and CCP as complementary architectures which can be combined to provide composable benefits.

Finally, there is a wide range of previous literature on moving kernel functionality into user-space. Arrakis [30] is system that facilitates kernel-bypass networking for applications via SR-IOV. IX [5] is a dataplane operating system that separates the management functionality of the kernel from packet processing. Alpine [15] moves all of TCP and IP into user-space. Whereas these systems use hardware virtualization to allow applications to have finer grained control over their networking resources, CCP exposes only congestion control information to user-space. Moreover, CCP is also agnostic to the datapath; datapaths for library operating systems could be CCP datapaths.

3 CCP Design Principles

To enable rich new congestion control algorithms on datapaths, CCP must provide a low-barrier programming environment and access to libraries (e.g., for optimization, machine learning, etc.). Further, new algorithms should also achieve high performance running at tens of Gbit/s per connection with small packet delays in the datapath.

3.1 Isolating Algorithms from the Datapath

Should congestion control algorithms run in the same address space as the datapath? There are conflicting factors to consider:

Safety. Supporting experimentation with algorithms and the possibility of including user-space code means that programs implementing congestion control algorithms should be considered untrusted. If algorithms and the datapath are in the same address space, bugs in algorithm or library code could cause datapath crashes or create vulnerabilities leading to privilege escalations in the kernel datapath.

Flexibility. Placing congestion control functionality outside the datapath provides more flexibility. For example, we anticipate future use cases of the CCP architecture where a congestion control algorithm may run on a machine different from the sender, enabling control policies across groups of hosts.

Performance. On the other hand, congestion control algorithms can access the datapath's congestion measurements with low delays and high throughput if the two reside in the same address space.

Implementation	Reporting Interval	Mean Throughput
Kernel	-	43 Gbit/s
CCP	Per ACK	29 Gbit/s
CCP	Per 10 ms	41 Gbit/s

Table 1: Single-flow throughput for different reporting intervals between the Linux kernel and CCP user-space, compared to kernel TCP throughput. Per-ACK feedback (0 μs interval) reduces throughput by 32% while using a 10 ms reporting interval achieves almost identical throughput to the kernel. Results as the number of flows increases are in §7.2.

Our design restructures congestion control algorithms into two components in separate address spaces: an off-datapath *CCP agent* and a component that executes in the datapath itself. The CCP agent provides a flexible execution environment in user space for congestion control algorithms, by receiving congestion signals from the datapath and invoking the algorithm code on these signals. Algorithm developers have full access to the user-space programming environment, including tools and libraries. The datapath component is responsible for processing feedback (e.g., TCP or QUIC ACKs, packet delays, etc.) from the network and the receiver, and providing congestion signals to the algorithms. Further, the datapath component provides interfaces for algorithms to set congestion windows and pacing rates.

An alternative design would be to run both the algorithm and the datapath in the same address space, but with fault isolation techniques [9, 16, 26, 31, 36, 38, 44]. However, this approach comes with significantly increased CPU utilization (e.g., 2× [9, 26, 31, 36, 38], resulting from tracing and run-time checks), a restrictive development environment [44], or changes to development tools such as the compiler [16, 38]. These performance and usability impediments, in our view, significantly diminish the benefits of running congestion control algorithms and the datapath in one address space.

3.2 Decoupling Congestion Control from the ACK Clock

Typical congestion control implementations in the Linux kernel are coupled to the so-called "ACK-clock," i.e., algorithm functionality is invoked upon receiving a packet acknowledgment in the networking stack. In contrast, with CCP, algorithms operate on summaries of network observations obtained over multiple measurements gathered in the datapath. Users program the datapath to gather these summaries using a safe domain-specific language (§3.3).

This decoupling of algorithm logic from the ACK clock provides two benefits.

First, users can develop congestion control algorithms free from the strict time restrictions rooted in the inter-arrival time of packet acknowledgments—a useful feature, especially at high link rates. Hence, it is possible to build algorithms that perform complex computations and yet achieve high throughput.

Second, the ability to provide congestion feedback less frequently than per-ACK can significantly reduce the overhead

of datapath-CCP communication. Table 1 shows that for a single saturating `iperf` connection over a loopback interface, Linux kernel TCP on a server machine with four 2.8-Ghz cores achieves 45 Gbit/s running Reno. In comparison, per-ACK reporting from the kernel to the CCP agent achieves only 68% of the kernel's throughput. By increasing the time between reports sent to the slow path to 10 ms (see the "per 10 ms" row), our implementation of Reno in CCP achieves close to the kernel's throughput.

Given that CCP algorithms operate over measurements supplied only infrequently, a key question is how best to summarize congestion signals within the datapath so algorithms can achieve high fidelity compared to a traditional in-datapath implementation. Indeed, in §7.1 we show that reporting on an RTT time-scale does not affect the fidelity of CCP algorithm implementations relative to traditional in-kernel implementations.

3.3 Supporting per-ACK Logic Within the Datapath

How must the datapath provide congestion feedback to algorithms running in the CCP agent? Ideally, a datapath should supply congestion signals to algorithms with suitable granularity (e.g., averaged over an RTT, rather than per ACK), at configurable time intervals (e.g., a few times every RTT) and during critical events (e.g., packet losses). With CCP, users can specify such datapath behavior using a domain-specific language (§4). At a high level, CCP-compatible datapaths expose a number of *congestion signals*, over which users can write *fold functions* to summarize network observations for algorithms. It is also possible to perform *control actions* such as reporting summarized measurements to CCP or setting a flow's pacing rate. Datapath programs can trigger fold functions and control actions when certain conditions hold, e.g., an ACK is received or a timer elapses. Users can thus control how to partition the logic of the algorithm between these two components according to their performance and flexibility requirements (§4.4).

4 Writing Algorithms in CCP

Figure 1 shows the control loop of a congestion control algorithm in CCP. Users implement two callback handlers (`onCreate()` and `onReport()`) in the CCP agent and one or more datapath programs. When a new flow is created, CCP's datapath component invokes the `onCreate()` handler. The implementation of `onCreate()` must install an initial datapath program for that flow. Datapath programs could compute summaries over per-packet congestion signals (such as a minimum packet delay or a moving average of packet delivery rate) and report summaries or high priority conditions (such as loss) to the CCP agent. On a report, the CCP agent invokes the `onReport()` handler which contains the bulk of the logic of the congestion control algorithm. The `onReport()` function computes and installs the flow's congestion window or sending rate using the signals from the datapath report. It may also replace the datapath program entirely with different logic.

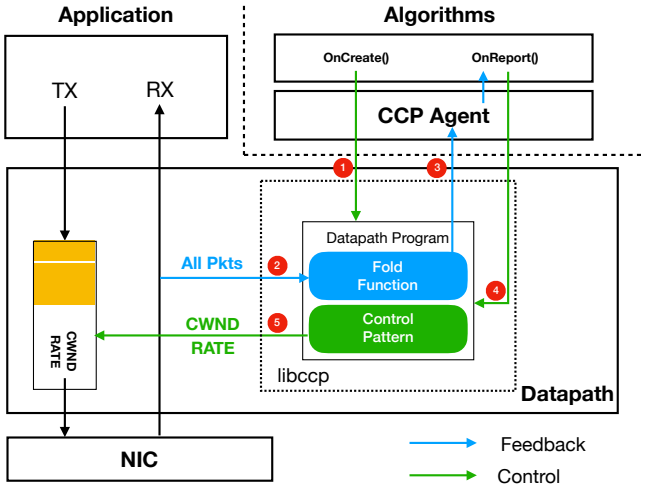


Figure 1: Congestion control algorithms in CCP are distinct from the application and datapath. Users specify an `onCreate()` handler which CCP calls when a new flow begins. In this handler, algorithms install (1) a datapath program. This datapath program aggregates incoming measurements (2) using user-defined fold functions and occasionally sends reports (3) to CCP, which calls the `onReport()` handler. The `onReport()` handler can update (4) the datapath program, which uses its defined control patterns to enforce (5) a congestion window or pacing rate.

```
1 (def (Report (volatile acked 0) (volatile lost 0)))
2 (when true
3   (:= Report.acked (+ Report.acked Ack.bytes_acked))
4   (:= Report.lost  (+ Report.lost  Ack.lost_pkts_sample))
5   (fallthrough))
6 (when (> Report.lost 0) (report))
```

Figure 2: A simple datapath program to count bytes acked and report on losses.

4.1 Datapath Program Abstractions

CCP's datapath programs are written in a simple domain specific language. These programs exist in order to provide a per ACK execution environment, where algorithms can define and update variables per ACK and perform *control actions*, in response to the values of these variables.

Figure 2 shows a program that counts the cumulative number of packets acknowledged and lost and reports these counters immediately upon a loss. The first statement of the program allows users to define custom variables. The "Report" block signifies that these variables should be included in the report message sent to the CCP agent. The `volatile` marker means that these variables should be reset to their initial values, 0, after every report to the CCP agent.

Primitive congestion signals	
Signal	**Definition**
`Ack.bytes_acked`, `Ack.packets_acked`	In-order acknowledged
`Ack.bytes_misordered`, `Ack.packets_misordered`	Out-of-order acknowledged
`Ack.ecn_bytes`, `Ack.ecn_packets`	ECN-marked
`Ack.lost_pkts_sample`	Number of lost packets
`Ack.now`	Datapath time (e.g., Linux `jiffies`)
`Flow.was_timeout`	Did a timeout occur?
`Flow.rtt_sample_us`	A recent sample RTT
`Flow.rate_outgoing`	Outgoing sending rate
`Flow.rate_incoming`	Receiver-side receiving rate
`Flow.bytes_in_flight`, `Flow.packets_in_flight`	Sent but not yet acknowledged

Operators	
Class	**Operations**
Arithmetic	`+`, `-`, `*`, `/`
Assignment	`:=`
Comparison	`==`, `<`, `>`, `or`, and
Conditionals	`If` (branching)

Variable Scopes	
Scope	**Description**
`Ack`	Signals measured per packet
`Flow`	Signals measured per connection
`Timer`	Multi-resolution timer that can be zeroed by a call to `reset`

Table 2: Datapath language: congestion signals, operators, and scopes.

Following the `def` block, *fold functions* provide custom summaries over primitive congestion signals. Datapath programs have read access to these *primitive congestion signals* (prefixed with "`Ack.`" or "`Flow.`" to specify their measurement period), which are exposed by the datapath on every incoming packet. Such signals include the round trip delay sample, the number of bytes the datapath believes have been dropped by the network, and the delivery rates of packets. Table 2 enumerates the primitive congestion signals we support. Users can write simple mathematical summaries over these primitive signals, as shown in Lines 3-4 of Figure 2.

Finally, algorithms can perform control actions in response to conditions defined by the fold function variables, e.g., updating a rate or cwnd or reporting the user defined variables to the CCP agent. As shown in Figure 2, the program defines a series of `when` clauses, and performs the following block only if the condition was evaluated to true.

CCP's datapath program language provides an event driven programming model. The condition (`when true...`) signifies that the body should be evaluated on every packet. This is where the program might calculate fold function summaries. The `when` clauses have access to all the fold function variables, as well as timing related counters. The `report` instruction causes the datapath to transmit the `acked` and `lost` counters to the CCP agent. By default, the program evaluates until one `when` clause evaluates to true; the (`fallthrough`) instruction at the end of the first `when` indicates that subsequent `when` clauses should also be evaluated.

4.2 CCP Algorithm Logic

The `onReport()` handler provides a way to implement congestion control actions in user-space in reaction to reports from the datapath. For example, a simple additive-increase multiplicative-decrease (AIMD) algorithm could be implemented in Python[1] using the `acked` and `lost` bytes reported every round-trip time from the datapath:

```
def onReport(self, report):
  if report["lost"] > 0:
    self.cwnd = self.cwnd / 2
  else:
    acked = report["acked"]
    self.cwnd = self.cwnd + acked*MSS/self.cwnd
  self.update("cwnd", self.cwnd/MSS)
```

We have implemented complex functionality within congestion control algorithms by leveraging slow-path logic, for example, a congestion control algorithm that uses Fast Fourier Transform (FFT) operations [19].

If the round-trip time of the network is a few milliseconds or more, it is possible to locate congestion control algorithm logic entirely within CCP with high fidelity relative to a per-packet update algorithm, as we show in §7.1.

4.3 Example: BBR

As a more involved example, we show below how various components of TCP BBR [8] are implemented using the CCP API. A BBR sender estimates the rate of packets delivered to the receiver, and sets its sending rate to the maximum delivered rate (over a sliding time window), which is believed to be the rate of the bottleneck link between the sender and the receiver.

This filter over the received rate is expressed simply in a fold function:

```
(when true
   (:= minrtt (min minrtt Ack.rtt_sample_us))
   (:= curr_btl_est (max curr_btl_est Flow.rate_incoming))
   (fallthrough))
```

To determine whether a connection can send more than its current sending rate, BBR probes for additional available bandwidth by temporarily increasing its sending rate by a factor ($1.25\times$) of its current sending rate. To drain a queue that may have been created in the process, it also reduces its rate by a reciprocal factor ($0.75\times$) before starting to send at the new estimated bottleneck link rate.

The following excerpt expresses this sending pattern (for simplicity, we show only 2 transitions):

```
(when (== pulseState 0)
  (:= Rate (* 1.25 curr_btl_est))
```

[1] Our CCP implementation is in Rust and exposes Python bindings (§5).

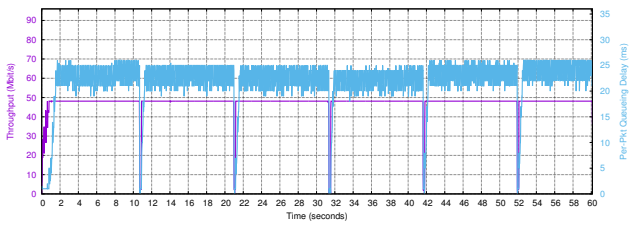


Figure 3: Our CCP implementation of BBR used for a bulk transfer over a 48 Mbit/s link with a 20 ms RTT and 2 BDPs of buffering. The bandwidth probe phase can be seen in the oscillation of the queueing delay, and the RTT probe phase can be seen in the periodic dips in throughput.

```
  (:= pulseState 1))
(when (&& (== pulseState 1)
          (> Timer.micros Flow.rtt_sample_us))
  (:= Rate (* 0.75 curr_btl_est))
  (:= pulseState 2))
```

Here, the variable `pulseState` denotes the state of the sender's bandwidth probing: probing with high sending rate (0) and draining queues with low sending rate (1). Each `when` clause represents a pulse state transition and is conditioned on the resettable timer `Timer.micros`. Upon the transition, the handler sets the `Rate` and advances `pulseState`. After the last phase of the pulse, the handler would reset the timer and `pulseState` to restart the sending pattern (not shown).

Figure 3 shows the impact of BBR's bandwidth probing[2] on the achieved goodput and queueing delays when a single flow runs over a 48 Mbit/s bottleneck link with a 20 ms round trip propagation delay. BBR's windowed min/max operations and the RTT probing phase (showing steep rate dips every 10 seconds) are implemented in the slow path's `onReport()` handler by installing a new fold function. CCP's split programming model enables this flexible partitioning of functionality.

4.4 Case Study: Slow Start

Because algorithms no longer make decisions upon every ACK, CCP changes the way in which developers should think about congestion control, and correspondingly provides multiple implementation choices. As a result, new issues arise about where to place algorithm functionality. We discuss the involved trade-offs with an illustrative example: slow start.

Slow start is a widely used congestion control module in which a connection probes for bandwidth by multiplicatively increasing its congestion window (`cwnd`) every RTT. Most implementations increment `cwnd` per ACK, either by the number of bytes acknowledged in the ACK, or by 1 MSS. One way to implement slow start is to retain the logic entirely in CCP, and measure the size of the required window update from datapath reports. We show an example in Figure 4. This

[2]We only implement BBR's PROBE_BW and PROBE_RTT. Our implementation is here: github.com/ccp-project/bbr.

```
fn create(...) {
  datapath.install("
  (def (Report (volatile acked 0) (volatile loss 0)))
  (when true
    (:= Report.acked (+ Report.acked Ack.bytes_acked)))
  (when (> Micros Flow.rtt_sample_us) (report) (reset))");
}
fn onReport(...) {
  if report.get_field("Report.loss") == 0 {
    let acked = report.get_field("Report.acked");
    self.cwnd += acked;
    datapath.update_field(&[("Cwnd", self.cwnd)]);
  } else { /* exit slow start */ }
}
```

Figure 4: A CCP implementation of slow start.

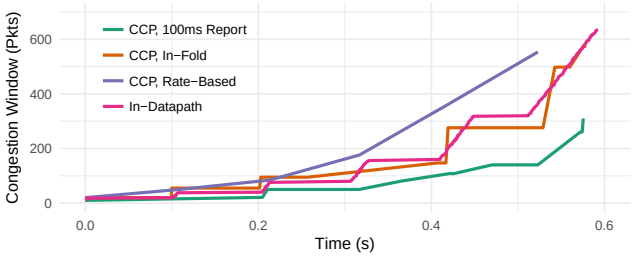


Figure 5: Different implementations of slow start have different window update characteristics. The control pattern implementation is rate-based, so we show the congestion window corresponding to the achieved throughput over each RTT.

implementation strategy is semantically closest in behavior to native datapath implementations.

For some workloads this approach may prove problematic, depending on the parameters of the algorithm. If the reporting period defined is large, then infrequent slow start updates can cause connections to lose throughput. Figure 5 demonstrates that, on a 48 Mbps, 100 ms RTT link, different implementations of slow start exhibit differing window updates relative to the Linux kernel baseline. A version with a 1-RTT reporting period lags behind the native datapath implementation. It is also possible to implement slow start within the datapath either by using congestion window increase (Figure 6), or by using rate based control:

```
(when (> Timer.Micros Flow.rtt_sample_us)
  (:= Rate (* Rate 2))
  (:= Timer.Micros 0))
```

Take-away. As outlined in §3, the programming model of datapath programs is deliberately limited. First, we envision that in the future, CCP will support low-level hardware datapaths—the simpler the fold function execution environment is, the easier these hardware implementations will be. Second, algorithms able to make complex decisions on longer time-scales will naturally do so to preserve cycles for the application and datapath; as a result, complex logic inside the fold function may not be desirable.

```
fn create(...) {
 datapath.install("
 (def (volatile Report.loss 0))
 (when true (:= Cwnd (+ Cwnd Ack.bytes_acked)))
 (when (> Ack.lost_pkts_sample 0) (report))");
}
fn onReport(...) { /* exit slow start */ }
```

Figure 6: A within-fold implementation of slow start. Note that CCP algorithm code is not invoked at all until the connection experiences its first loss.

More broadly, developers may choose among various points in the algorithm design space. On one extreme, algorithms may be implemented almost entirely in CCP, using the fold function as a simple measurement query language. On the other extreme, CCP algorithms may merely specify transitions between in-datapath fold functions implementing the primary control logic of the algorithm. Ultimately, users are able to choose the algorithm implementation best suited to their congestion control logic and application needs.

5 CCP Implementation

We implement a user-space CCP agent in Rust, called Portus[3], which implements functionality common across independent congestion control algorithm implementations, including a compiler for the datapath language and a serialization library for IPC communication. CCP congestion control algorithms are hence implemented in Rust; we additionally expose bindings in Python. The remainder of this section will discuss datapath support for CCP.

5.1 Datapath Requirements

A CCP-compatible datapath must accurately enforce the congestion control algorithm specified by the user-space CCP module. Once a datapath implements support for CCP, it automatically enables all CCP algorithms. An implementation of the CCP datapath must perform the following functions:

- The datapath should communicate with a user-space CCP agent using an IPC mechanism. The datapath multiplexes `reports` from multiple connections onto the single persistent IPC connection to the slow path. It must also perform the proper serialization for all messages received and sent.
- The datapath should execute the user-provided domain-specific program on the arrival of every acknowledgment or a timeout in a safe manner. Datapath programs (§4) may include simple computations to summarize per-packet congestion signals (Table 2) and enforce congestion windows and rates.

[3] github.com/ccp-project/portus

5.2 Safe Execution of Datapath Programs

Datapaths are responsible for safely executing the program sent from the user-space CCP module. While CCP will compile the instructions and check for mundane errors (e.g., use of undefined variables) before installation, it is the datapath's responsibility to ensure safe interpretation of the instructions. For example, datapaths should prevent divide by zero errors when calculating user defined variables and guarantee that programs cannot overwrite the congestion primitives. However, algorithms are allowed to set arbitrary congestion windows or rates, in the same way that any application can congest the network using UDP sockets.

Thankfully, this task is straightforward as datapath programs are limited in functionality: programs may not enter loops, perform floating point operations, define functions or data structures, allocate memory, or use pointers. Rather, programs are strictly a way to express arithmetic computations over a limited set of primitives, define when and how to set congestion windows and pacing rates, and report measurements.

5.3 `libccp`: CCP's Datapath Component

We have implemented a library, `libccp`[4], that provides a reference implementation of CCP's datapath component, in order to simplify CCP datapath development. `libccp` is lightweight execution loop for datapath programs and message serialization. While we considered using eBPF [14] or TCP BPF [6] as the execution loop, including our own makes `libccp` portable to datapaths outside the Linux kernel; the execution loop runs the same code in all three datapaths we implemented.

To use `libccp`, the datapath must provide callbacks to functions that: (1) set the window and rate, (2) provide a notion of time, and (3) send an IPC message to CCP. Upon reading a message from CCP, the datapath calls `ccp_recv_msg()`, which automatically de-multiplexes the message for the correct flow. After updating congestion signals, the datapath can call `ccp_invoke()` to run the datapath program, which may update variable calculations, set windows or rates, and send report summaries to CCP. It is the responsibility of the datapath to ensure that it correctly computes and provides the congestion signals in Table 2.

The more signals a datapath can measure, the more algorithms that datapath can support. For example, CCP can only support DCTCP [1] or ABC [18] on datapaths that provide ECN support; CCP will not run algorithms on datapaths lacking support for that algorithm's requisite primitives.

5.4 Datapath Implementation

We use `libccp` to implement CCP support in three software datapaths: the Linux kernel[5]; mTCP, a DPDK-based datapath;

[4] github.com/ccp-project/libccp

[5] Our kernel module is built on Linux 4.14: github.com/ccp-project/ccp-kernel

Signal	Definition
`Ack.bytes_acked`, `Ack.packets_acked`	`Delta(tcp_sock.bytes_acked)`
`Ack.bytes_misordered`, `Ack.packets_misordered`	`Delta(tcp_sock.sacked_out)`
`Ack.ecn_bytes`, `Ack.ecn_packets`	`in_ack_event: CA_ACK_ECE`
`Ack.lost_pkts_sample`	`rate_sample.losses`
`Ack.now`	`getnstimeofday()`
`Flow.was_timeout`	`set_state: TCP_CA_Loss`
`Flow.rtt_sample_us`	`rate_sample.rtt_us`
`Flow.rate_outgoing`	`rate_sample.delivered / Delta(tcp_sock.first_tx_mstamp)`
`Flow.rate_incoming`	`rate_sample.delivered / Delta(tcp_sock.tcp_mstamp)`
`Flow.bytes_in_flight`, `Flow.packets_in_flight`	`tcp_packets_in_flight(tcp_sock)`

Table 3: Definition of CCP primitives in terms of the `tcp_sock` and `rate_sample` structures, for the Linux kernel datapath.

and Google's QUIC. For both the Linux kernel and QUIC datapaths, we leveraged their respective pluggable congestion control interfaces, which provide callbacks upon packet acknowledgements and timeouts, where the `libccp` program interpreter can be invoked. The kernel module implements the communication channel to CCP using either Netlink sockets or a custom character device, while mTCP and QUIC use Unix domain sockets. We additionally modified the QUIC source code to support multiplexing CCP flows on one persistent IPC connection and to expose the function callbacks required by the `libccp` API.

Unlike QUIC and the Linux kernel, mTCP only implements Reno and does not explicitly expose a congestion control interface for new algorithms. In order to achieve behavior consistent with other datapaths, we also implemented SACK and packet pacing; these features were previously lacking.

The definition of congestion signal primitives, IPC, and window and rate enforcement mechanisms is the only datapath-specific work needed to support CCP. As an example, Table 3 details the mapping of kernel variables to CCP primitives. Most of these definitions are straightforward; the CCP API merely requires datapaths to *expose* variables they are already measuring. All other necessary functionality, most notably interpreting and running the datapath programs, is shared amongst software datapaths via `libccp` (§5.3).

6 New Capabilities

We present four new capabilities enabled by CCP: new congestion control algorithms that use sophisticated user-space programming libraries, rapid development and testing of algorithms, congestion control for flow aggregates, and the ability to write an algorithm once and run it on multiple datapaths.

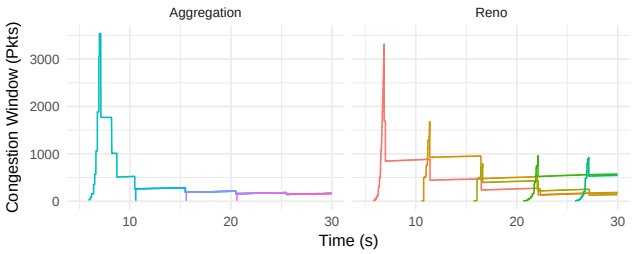


Figure 7: 5 20-second iperf flows with 10 second staggered starts. While Reno (right) must individually probe for bandwidth for each new connection, an aggregating congestion controller is able to immediately set the connection's congestion window to the fair share value.

6.1 Sophisticated Congestion Control Algorithms

CCP makes it possible to use sophisticated user-space libraries, such as libraries for signal processing, machine learning, etc. to implement congestion control algorithms.

One example is Nimbus [19], a new congestion control algorithm that detects whether the cross traffic at a bottleneck link is elastic (buffer-filling) or not, and uses different control rules depending on the outcome. The Nimbus algorithm involves sending traffic in an asymmetric sinusoidal pulse pattern and using the sending and receiving rates measured over an RTT to produce a time-series of cross-traffic rates. The method then computes the FFT of this time-series and infers elasticity if the FFT at particular frequencies is large.

The implementation of Nimbus uses CCP to configure the datapath to report the sending and receiving rates periodically (e.g., every 10 ms), maintains a time-series of the measurements in user-space, and performs FFT calculations using a FFT library in Rust [34].

Although it is possible to implement such algorithms directly in the datapath, it would be significantly more difficult. For instance, one would need to implement the FFT operations with fixed-point arithmetic. Moreover, implementing the algorithm outside the datapath using CCP allows for a tighter development-testing loop than writing kernel code.

We anticipate that in the future, CCP will enable the use of other similarly powerful but computationally-intensive methods such as neural networks.

6.2 Velocity of Development

Copa [3] is a recently proposed model-based congestion control algorithm that seeks to maintain a target rate that is inversely proportional to the queuing delay, estimated as the difference of the current RTT and the minimum RTT. It is robust to non-congestive loss, buffer-bloat, and unequal propagation delays. It includes mechanisms to provide TCP competitiveness, accurate minimum RTT estimation, and imperfect pacing.

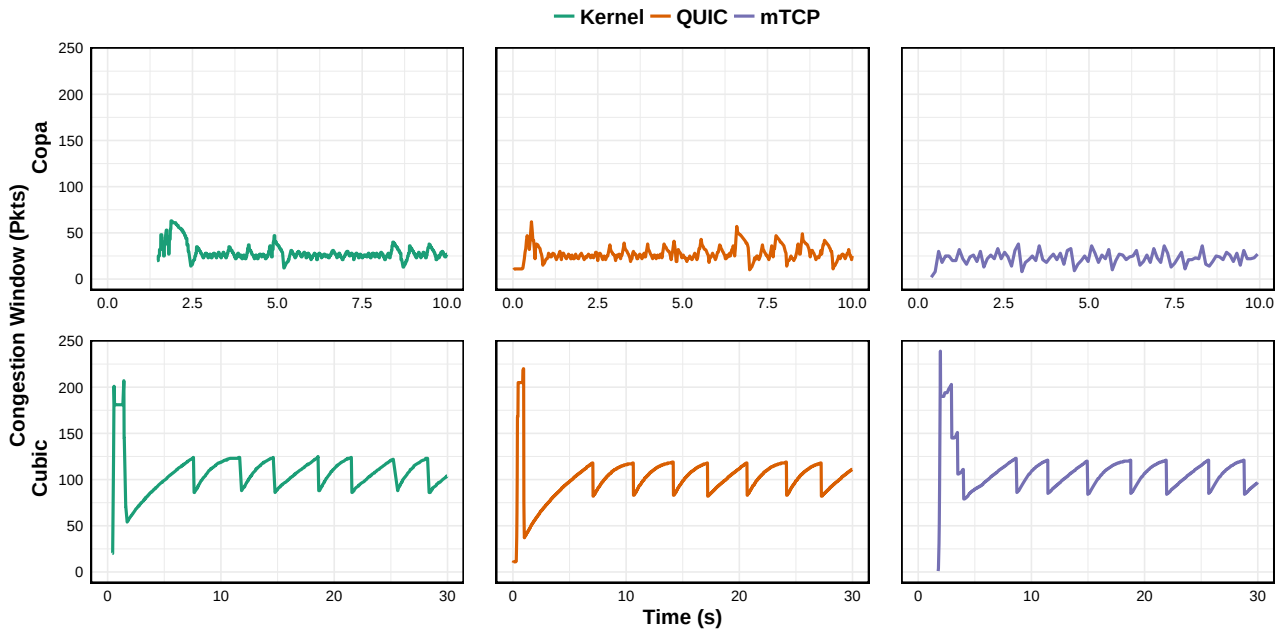


Figure 8: Comparison of the same CCP implementation of Cubic and Copa run on three different datapaths. Copa is run on a fixed 12 Mbps link with a 20 ms RTT; Cubic is run on a fixed 24 Mbps link with a 20 ms RTT.

The authors of Copa used CCP to implement Copa recently, and in the process discovered a small bug that produced an erroneous minimum RTT estimate due to ACK compression. They solved this problem with a small modification to the Copa datapath program, and in a few hours were able to improve the performance of their earlier user-space implementation. The improvement is summarized here:

Algorithm	Throughput	Mean queue delay
Copa (UDP)	1.3 Mbit/s	9 ms
Copa (CCP-Kernel)	8.2 Mbit/s	11 ms

After the ACK compression bug was fixed in the CCP version, Copa achieves higher throughput on a Mahimahi link with 25 ms RTT and 12 Mbit/s rate while maintaining low mean queueing delay. Because of ACK compression, the UDP version over-estimates the minimum RTT by 5×.

6.3 Flow Aggregation

Congestion control on the Internet is performed by individual TCP connections. Each connection independently probes for bandwidth, detects congestion on its path, and reacts to it. Congestion Manager [4] proposed the idea of performing congestion control for aggregates of flows at end-hosts. Flow aggregation allows different flows to share congestion information and achieve the correct rate more quickly.

We describe how to use CCP to implement a host-level aggregate controller that maintains a single aggregate window or rate for a group of flows and allocates that to individual flows—all with no changes to the non-CCP parts of the datapath.

Interface. In addition to the `create()` and `onReport()` event handlers, we introduce two new APIs for aggregate congestion controllers: `create_subflow()` and `aggregateBy()`. CCP uses `aggregateBy()` to classify new connections into aggregates. Then, it calls either the existing `create()` handler in the case of a new aggregate, or the `create_subflow()` handler in the case of an already active one.

These handlers are natural extensions of the existing per-flow API; we implemented API support for aggregation in 80 lines of code in our Rust CCP implementation (§7). Algorithms can aggregate flows using the connection 5-tuple, passed as an argument to `aggregateBy()`.

As a proof of concept, we implement an algorithm which simply aggregates all flows on each of the hosts's interfaces into one aggregate and assigns the window in equal portions to each sub-flow. Figure 7 shows the aggregator instantaneously apportioning equal windows to each flow in its domain.

6.4 Write-Once, Run-Anywhere

Implementing a new congestion control algorithm is difficult because of the subtle correctness and performance issues that require expertise to understand and resolve. New algorithms are often implemented in a single datapath and new datapaths have very few algorithms implemented. CCP enables algorithm designers to focus on building and testing a single solid implementation of their algorithm that users can then run on any (supported) datapath.

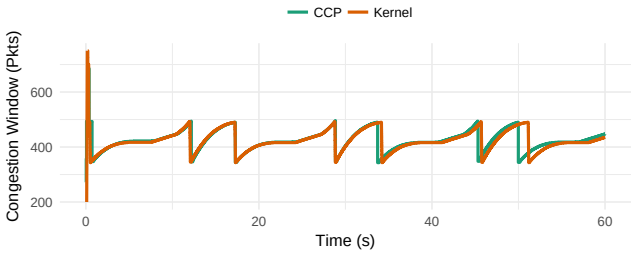


Figure 9: Cubic in CCP matches Cubic in Linux TCP.

To exhibit this capability, we ran the same implementation of both Cubic (not previously implemented in mTCP) and Copa (§6.2, not previously implemented in any widely-used datapath) on the three datapaths and plot the congestion window evolution over time in Figure 8.

As expected, the congestion window naturally evolves differently on each datapath, but the characteristic shapes of both algorithms are clearly visible. Copa uses triangular oscillations around an equilibrium of 1 BDP worth of packets (22 in this case), periodically draining the queue in an attempt to estimate the minimum RTT.

7 Evaluation

We evaluated the following aspects of CCP:

Fidelity (§7.1). Do algorithms implemented in CCP behave similarly to algorithms implemented within the datapath? Using the Linux kernel datapath as a case study, we explore both achieved throughput and delay for persistently backlogged connections as well as achieved flow completion time for dynamic workloads.

Overhead of datapath communication (§7.2). How expensive is communication between CCP and the datapath?

High bandwidth, low RTT (§7.3). We use ns-2 simulations to demonstrate that CCP's method of taking congestion control actions periodically can perform well even in ultra-low RTT environments.

Unless otherwise specified, we evaluated our implementation of CCP using Linux 4.14.0 on a machine with four 2.8 Ghz cores and 64 GB memory.

7.1 Fidelity

The Linux kernel is the most mature datapath we consider. Therefore, we present an in-depth exploration of congestion control outcomes comparing CCP and native-kernel implementations of two widely used congestion control algorithms: NewReno [22] and Cubic [21]. As an illustrative example, Figure 9 shows one such comparison of congestion window update decisions over time on an emulated 96 Mbit/s fixed-rate Mahimahi [27] link with a 20 ms RTT. We expect and indeed observe minor deviations as the connection progresses and small timing differences between the two implementations cause the window to differ, but overall, not only does CCP's implementation of Cubic exhibit a window update consistent with a cubic increase function, but its updates closely match the kernel implementation.

For the remainder of this subsection, we compare the performance of CCP and kernel implementations of NewReno and Cubic on three metrics (throughput and delay in §7.1.1, and FCT in §7.1.2) and three scenarios, all using Mahimahi.

7.1.1 Throughput and Delay. We study the following scenarios:

Fixed-rate link ("fixed"). A 20 ms RTT link with a fixed 96 Mbit/s rate and 1 BDP of buffering.

Cellular link ("cell"). A 20 ms RTT variable-rate link with a 100-packet buffer based on a Verizon LTE bandwidth trace [27].

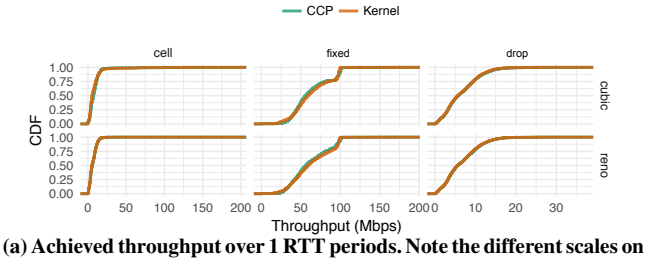
Stochastic drops ("drop"). A 20 ms RTT link with a fixed 96 Mbit/s rate, but with 0.01% stochastic loss and an unlimited buffer. To ensure that both tested algorithms encountered exactly the same conditions, we modified Mahimahi to use a fixed random seed when deciding whether to drop a packet.

These three scenarios represent a variety of environments congestion control algorithms encounter in practice, from predictable to mobile to bufferbloated paths. We calculate, per-RTT over twenty 1-minute experiments, the achieved throughput (10a) and delay (10b), and show the ensuing distributions in Figure 10.

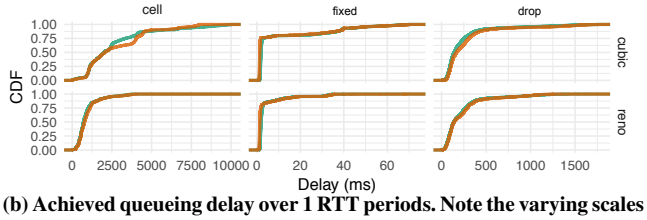
Overall, both distributions are close, suggesting that CCP's implementations make the same congestion control decisions as the kernel.

7.1.2 Flow Completion Time. To measure flow completion times (FCT), we use a flow size distribution compiled from CAIDA Internet traces [7] in a similar setting to the "fixed" scenario above; we use a 100 ms RTT and a 192 Mbit/s link. To generate traffic, we use a traffic generator to sample flow sizes from the distribution and send flows of that size according to a Poisson arrival process to a single client behind the emulated Mahimahi link. We generate flows with 50% average link load, and generate 100,000 flows to the client from 50 sending servers using persistent connections to the client. We used Reno as the congestion control algorithm in both cases. To ensure that the kernel-native congestion control ran under the same conditions as the CCP implementation, we disabled the slow-start-after-idle option.

Of the 100,000 flows we sampled from the CAIDA workload, 97,606 were 10 KB or less, comprising 487 MB, while the 95 flows greater than 1 MB in size accounted for 907 MB out of the workload's total of 1.7 GB.



(a) Achieved throughput over 1 RTT periods. Note the different scales on the x-axes for the three scenarios.



(b) Achieved queueing delay over 1 RTT periods. Note the varying scales on the x-axes for the three scenarios.

Figure 10: Comparison of achieved throughput over 20 ms periods. The achieved throughput distributions are nearly identical across the three scenarios and two congestion control algorithms evaluated.

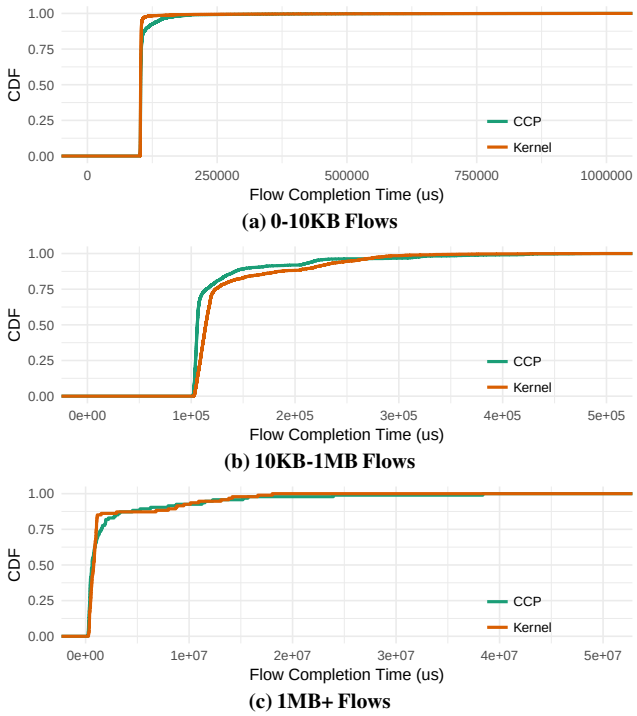


Figure 11: CDF comparisons of flow completion times. Note the differing x-axes.

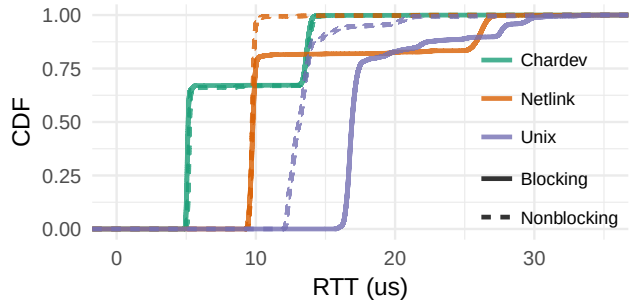


Figure 12: Minimum time required to send information to the datapath and receive a response using different IPC mechanisms.

Across all flow sizes, CCP achieves FCTs 0.02% lower than the kernel in the median, 3% higher in the 75th percentile, and 30% higher in the 95th percentile.

Small flows. Flows less than 10 KB in size, shown in Figure 11a, are essentially unaffected by congestion control. These flows, the vast majority of flows in the system, complete before either CCP algorithms or kernel-native algorithms make any significant decisions about them.

Medium flows. Flows between 10 KB and 1 MB in size, in Figure 11b achieve 7% lower FCT in the median with CCP because CCP slightly penalizes long flows due to its slightly longer update period, freeing up bandwidth for medium size flows to complete.

Large flows. CCP penalizes some flows larger than 1 MB in size compared to the native-kernel implementation: 22% worse in the median (Figure 11c).

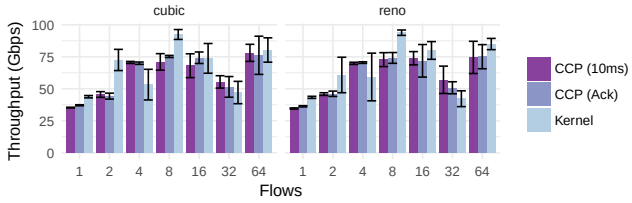
7.2 Performance

7.2.1 Measurement Staleness. Because our CCP implementation, Portus, runs in a different address space than datapath code, there is some delay between the datapath gathering a report and algorithm code acting upon the report. In the worst case, a severely delayed measurement could cause an algorithm to make an erroneous window update.

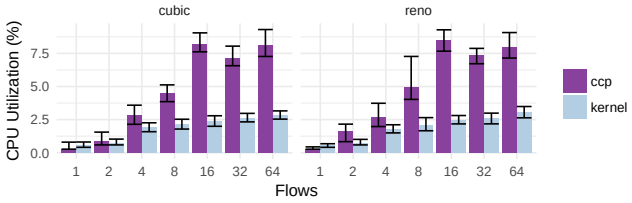
Fortunately, as Figure 12 shows, this overhead is small. We calculate an IPC RTT by sending a time-stamped message to a kernel module (or user-space process in the case of a Unix-domain socket). The receiver then immediately echoes the message, and we measure the elapsed time at the originating process.

We test three IPC mechanisms: Unix-domain sockets [32], a convenient and popular IPC mechanism used for communication between user-space processes; Netlink sockets [35], a Linux-specific IPC socket used for communication between the kernel and user-space; and a custom kernel module, which implements a message queue that can be accessed (in both user-space and kernel-space) via a character device.

In all cases, the 95th percentile latency is less than 30 μs.

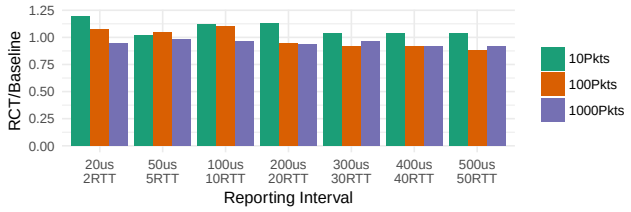


(a) Achieved localhost throughput as the number of flows increases

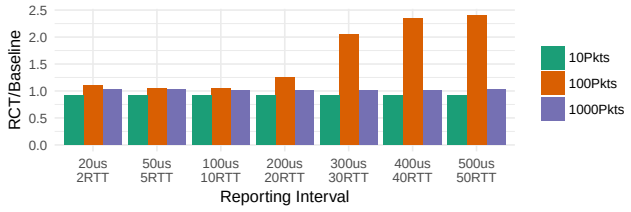


(b) CPU Utilization when saturating a 10 Gbit/s link.

Figure 13: CCP can handle many concurrent flows without significant CPU overhead. Error bars show standard deviation.



(a) Tail flow completion time at 10 Gbit/s



(b) Tail flow completion time at 40 Gbit/s

Figure 14: Mean tail completion across 50 simulations. While at 10 Gbit/s even rare reporting (every 50 RTTs) has limited overhead (at most 20%), at 40 Gbit/s, a 1 ms reporting period is necessary to avoid performance degradation.

7.2.2 Scalability. CCP naturally has nonzero overhead since more context switches must occur to make congestion control decisions in user-space. We test two scenarios as the number of flows in the system increases exponentially from 1 to 64. In both scenarios, we test CCP's implementation of Reno and Cubic against the Linux kernel's. We measure average throughput and CPU utilization in 1 second intervals over the course of 10 30-second experiments using iperf [37]. We evaluate CCP with two fold functions: one which implements a reporting interval of 10 ms, and another which reports on every packet.

We omit mTCP and QUIC from these scalability microbenchmarks and focus on the kernel datapath. The QUIC toy server is mainly used for integration testing and does not perform well as the number of flows increase; we confirmed this behavior with Google's QUIC team. Similarly, after discussion with the mTCP authors, we were unable to run mTCP at sufficient speeds to saturate a localhost or 10 Gbit/sec connection.

Localhost microbenchmark. We measure achieved throughput on a loopback interface as the number of flows increases. As the CPU becomes fully utilized, the achieved throughput will plateau. Indeed, in Figure 13a, CCP matches the kernel's throughput up to the maximum number of flows tested, 64.

CPU Utilization. To demonstrate the overhead of CCP in a realistic scenario, we scale the number of flows over a single 10 Gbit/s link between two physical servers and measure the resulting CPU utilization. Figure 13b shows that as the number of flows increases, the CPU utilization in the CCP case rises steadily. 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; the CPU utilization nevertheless remains under 8% in all cases.

In both the CPU utilization and the throughput microbenchmarks, the differences in CPU utilization stem from the necessarily greater number of context switches as more flows send measurements to CCP. Furthermore, the congestion control algorithm used does not affect performance.

7.3 Low-RTT and High Bandwidth Paths

To demonstrate it is feasible to separate congestion control from the datapath even in low-RTT and high bandwidth situations, we simulate a datacenter incast scenario using ns-2 [29]. We model CCP by imposing both forms of delays due to CCP: (i) the period with which actions can be taken (the reporting period) and, (ii) the staleness after which sent messages arrive in CCP. We used our microbenchmarks in §7.2.1 to set the staleness to 20 μs, and vary the reporting interval since it is controlled by algorithm implementations. We used a 20 μs RTT with a 50-to-1 incast traffic pattern across 50 flows with link speeds of 10 and 40 Gbit/s. To increase the statistical significance of our results, we introduce a small random jitter to flow start times ($<10\mu$s with 10 Gbit/s bandwidth and <2.5 μs with 40 Gbit/s bandwidth) and run each point 50 times with a different simulator random seed value and report the mean.

Figure 14 compares the results with the baseline set to in-datapath window update. We find that at 10 Gbit/s, CCP performance stays within 15% of the baseline across different flow sizes and reporting intervals ranging from 10 μs to 500 μs. Recall that 500 μs is 50× the RTT; even this infrequent reporting period yields only minor degradation.

Meanwhile, at 40 Gbit/s the slowdown over the baseline increases with the reporting interval in the case of 100 packet flows, but not with 10 or 1000 packet flows. Similar to the results in §7.1.2, the short flows and long flows are both unaffected by the reporting period because the short flows complete too quickly and the long flows spend much of their time with large congestion windows regardless of the window update. Indeed, at 100 μs (10 RTTs), the tail completion time is within 10% of the baseline; as the reporting increases, the tail completion time increases to over 2$\times$ the baseline. This nevertheless suggests that when reporting intervals are kept to small multiples of the RTT, tail completion time does not suffer.

8 Conclusion

We described the design, implementation, and evaluation of CCP, a system that restructures congestion control at the sender. CCP defines better abstractions for congestion control, specifying the responsibilities of the datapath and showing a way to use fold functions and control patterns to exercise control over datapath behavior. We showed how CCP (i) enables the same algorithm code to run on a variety of datapaths, (ii) increases the "velocity" of development and improves maintainability, and (iii) facilitates new capabilities such as the congestion manager-style aggregation and sophisticated signal processing algorithms.

Our implementation achieves high fidelity compared to native datapath implementations at low CPU overhead. The use of fold functions and summarization reduces overhead, but not at the expense of correctness or accuracy.

Future work includes: (i) CCP support for customizing congestion control for specific applications such as video streaming and videoconferencing, (ii) CCP on hardware datapaths (e.g., SmartNICs), and (iii) CCP running on a different machine from the datapath to support cluster-based congestion management (e.g., for a server farm communicating with distributed clients).

Acknowledgements. We thank Venkat Arun and the SIGCOMM reviewers for their useful comments and feedback. Special thanks to Eddie Kohler, whose insistence on clarity and brevity greatly improved this paper. We additionally thank Tushar Dhoot and Joe Zhang for catching a labelling error while reproducing our results. This work was supported in part by DARPA contract HR001117C0048 and NSF grant 1407470.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[2] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 108–114, New York, NY, USA, 2017. ACM.

[3] V. Arun and H. Balakrishnan. Copa: Congestion Control Combining Objective Optimization with Window Adjustments. In *NSDI*, 2018.

[4] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM*, 1999.

[5] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014.

[6] L. Brakmo. TCP-BPF: Programmatically tuning TCP behavior through BPF. https://www.netdevconf.org/2.2/papers/brakmo-tcpbpf-talk.pdf.

[7] CAIDA. The CAIDA Anonymized Internet Traces 2016 Dataset - 2016-01-21. http://www.caida.org/data/passive/passive_2016_dataset.xml, 2016.

[8] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5):50:20–50:53, Oct. 2016.

[9] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[10] J. Corbet. Pluggable congestion avoidance modules. https://lwn.net/Articles/128681/, 2005.

[11] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, 2015.

[12] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. PCC Vivace: Online-Learning Congestion Control. In *NSDI*, 2018.

[13] DPDK. http://dpdk.org/.

[14] Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[15] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *USITS*, volume 1, pages 15–15, 2001.

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[17] B. Ford. Structured Streams: A New Transport Abstraction. In *SIGCOMM*, 2007.

[18] P. Goyal, M. Alizadeh, and H. Balakrishnan. Rethinking Congestion Control for Cellular Networks. In *HotNets*, 2017.

[19] P. Goyal, A. Narayan, F. Cangialosi, D. Raghavan, S. Narayana, M. Alizadeh, and H. Balakrishnan. Elasticity Detection: A Building Block for Delay-Sensitive Congestion Control. *ArXiv e-prints*, Feb. 2018.

[20] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying Safe User-level Network Services with icTCP. In *OSDI*, 2004.

[21] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.

[22] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*, 1996.

[23] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[24] A. B. Johnston and D. C. Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-time Web*. Digital Codex LLC, 2012.

[25] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*, 2017.

[26] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *SOSP*, 2011.
[27] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*, 2015.
[28] Netronome. Agilio LX SmartNICs. https://www.netronome.com/products/agilio-lx/. [Online, Retrieved July 28, 2017].
[29] The Network Simulator. https://www.isi.edu/nsnam/ns/.
[30] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *OSDI*, 2014.
[31] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*, 2002.
[32] D. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63, 1984.
[33] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.
[34] rustfft. https://crates.io/crates/rustfft.
[35] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol, 2003. RFC 3819, IETF.
[36] I. Thomas, I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th conference on USENIX Security Symposium*, 2001.
[37] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.
[38] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.
[39] K. Winstein and H. Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, 2012.
[40] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*, 2013.
[41] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013.
[42] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011.
[43] F. Y. Yan, J. Ma, G. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein. Pantheon: The Training Ground for Internet Congestion-Control research. In *USENIX ATC*, 2018.
[44] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, 2009.
[45] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.

A Reproducibility

As discussed in §5, we have published each of the components of CCP at github.com/ccp-project. This includes Portus, the CCP userspace agent (https://github.com/ccp-project/portus), `libccp` (https://github.com/ccp-project/libccp), and the implementation of CCP support for the three datapaths we evaluate. Our Linux kernel (https://github.com/ccp-project/ccp-kernel) datapath is a kernel module which we have tested using kernel version `4.14`. Our mTCP datapath (https://github.com/ccp-project/ccp-mtcp) is a fork of the original authors' mTCP implementation (https://github.com/eunyoung14/mtcp). We are working towards having the CCP changes merged into the upstream implementation. Our QUIC datapath (https://github.com/ccp-project/ccp-quic) is a patch for the Chromium QUIC implementation.

We also implemented a number of congestion control algorithms: Reno and Cubic (https://github.com/ccp-project/generic-cong-avoid), BBR (https://github.com/ccp-project/bbr), and Nimbus (https://github.com/ccp-project/nimbus). Venkat Arun's implementation of Copa is also available at https://github.com/venkatarun95/ccp_copa.

To make replicating our experiments easier, we have published a number of experiment scripts here: https://github.com/ccp-project/eval-scripts. Running the various scripts as outlined in the Readme will reproduce the figures from §7.

Finally, the source for this document is at https://github.com/ccp-project/sigcomm18.