Restructuring Endpoint Congestion Control

Unpublished Draft

Akshay Narayan
MIT CSAIL
akshayn@csail.mit.edu

Frank Cangialosi
MIT CSAIL
frankc@csail.mit.edu

Deepti Raghavan
MIT CSAIL
deeptir@csail.mit.edu

Prateesh Goyal
MIT CSAIL
prateesh@csail.mit.edu

Srinivas Narayana
MIT CSAIL
alephtwo@csail.mit.edu

Radhika Mittal
UC Berkeley
radhika@eecs.berkeley.edu

Mohammad Alizadeh
MIT CSAIL
alizadeh@csail.mit.edu

Hari Balakrishnan
MIT CSAIL
hari@csail.mit.edu

Abstract

This paper describes the implementation and evaluation of a Congestion Control Plane (CCP), a new way to structure congestion control functions at the sender by removing them from the datapath. With CCP, each datapath such as the Linux Kernel TCP, UDP-based QUIC, or kernel-bypass transports like mTCP/DPDK summarizes information about the round-trip time, packet receptions, losses, ECN, etc. via a well-defined interface, and algorithms running atop CCP can use this information to control the datapath's congestion window or pacing rate. We show that CCP enables, for the first time, congestion control algorithms to be written once and run on multiple datapaths. It also improves both the pace of development and ease of maintenance of congestion control algorithms by providing better, modular abstractions, and enables new capabilities such as aggregate congestion control across groups of connections, all with one-time changes to datapaths. 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. They also demonstrate new capabilities enabled by CCP, such as Copa in Linux TCP, several algorithms run for the first time on QUIC and mTCP/DPDK, a congestion manager from outside the datapath, and sophisticated congestion control using signal processing running on Linux TCP.

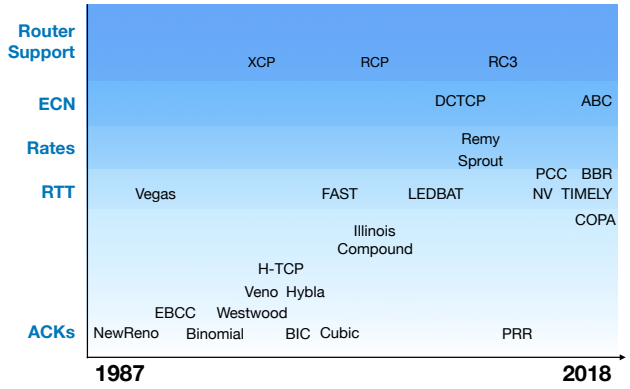


Figure 1: As link characteristics diversify, a developers have developed a battery of congestion control algorithms, from the "long-fat pipe" schemes of the mid-2000s [9, 18, 29, 32] to purely delay-based [6, 7, 33, 42, 45] and hybrid loss-delay [30, 43] schemes, and more recent proposals [3, 10, 13, 19, 47, 48].

1 Introduction

Due to the continual deployment of new applications and network technologies, and changes in workload patterns, research on congestion control has not only remained vibrant since the 1980s, but has flourished in recent years. Figure 1 shows a time-line of innovations in this area.

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 on each TCP connection.

An operating system's TCP software 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, many other datapaths have emerged, including user-space protocols atop UDP (e.g., QUIC [28], WebRTC [26], Mosh [46]), kernel-bypass methods (e.g., mTCP/DPDK [12, 25, 40]), RDMA [50], multi-path TCP (MPTCP) [49], and specialized Network Interface Cards ("SmartNICs" [36]). This trend suggests that the future will likely see many applications using datapaths different from kernel-supported TCP connections.

New datapaths typically do not offer much in terms of congestion control because implementing these algorithms correctly takes considerable time and effort. For instance, the set of available algorithms in mTCP [25], a TCP implementation on DPDK, is limited to a variant of Reno. Lest one dismiss this example as a case of a research project lacking in engineering resources, we note that QUIC, despite Google's imposing engineering resources, does not have most of the algorithms that have been implemented in the Linux kernel over 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. The difficulty isn't the volume of code, but the many subtle correctness and performance issues in various algorithms that are tricky, requiring expertise to understand and resolve.

This paper starts from the observation, made in a recent position paper [34], that congestion control algorithms do not need to be implemented in the datapath. If the datapath encapsulated the information available to it about *congestion signals* like the round-trip time (RTT), packets received and lost, explicit congestion notification (ECN) markings, etc., and periodically provided this information via a well-defined interface to an off-datapath module, then congestion control algorithms could run in the context of that module. Then, 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 of the off-datapath congestion control algorithm.

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:** With CCP, one can write a congestion control algorithm once and run it on any datapath that supports the specified interface. We demonstrate in this paper a variety of 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, Copa on QUIC).

(2) **Higher "velocity" of development:** With the right abstractions, a congestion control designer can focus on the algorithmic essentials without worrying about the details and data structures of the datapath. The result is more expressive, easier to maintain code. We show a deployment mode where CCP and the algorithms run at user level, which means that new algorithms can be deployed in production without the cumbersome "upstreaming" process. (Of course, the modifications we propose to the datapath must be deployed and upstreamed, but that effort does not grow with the number of algorithms).

(3) **New capabilities:** CCP makes it easier to provide new capabilities, such as aggregate control of multiple flows as previously proposed in the congestion manager [4], and writing and debugging new algorithms that require significant computation (e.g., signal processing, machine learning, etc.).

Our design and implementation of CCP includes the following contributions:

- A flexible API to request measurements from and exercise control over the datapath (§2). We define a small language with which algorithms can specify custom summaries over per-packet information.
- A specification of datapath responsibilities (§3). These include congestion signals that a datapath should maintain, as well as a simple framework to execute directives from a CCP program. This design enables "write-once, run-anywhere" protocols. We demonstrate this capability with several examples, showing multiple cases of the same CCP algorithm code running over three different datapaths.
- An exploration of the design space of new capabilities enabled by CCP (§4): the rapid development of new algorithms and implementing an aggregate congestion controller.
- A demonstration of the fidelity of CCP in a variety of link conditions. Our CCP implementation matches the performance of Linux kernel implementations at only a small overhead (5% more CPU utilization in the worst case). Furthermore, we show it is feasible to implement infrequent congestion control even in low-RTT, high bandwidth environments.

2 Design Overview

Our design restructures congestion control by separating it into two distinct components: an off-datapath agent (CCP) and a set of interfaces that the datapath must provide. The CCP agent provides the execution environment for user-specified congestion control algorithms. It receives data about congestion signals from the datapath and invokes the algorithm's code that uses this data. The datapath is responsible for processing feedback (e.g., TCP or QUIC ACKs) from the receiver to provide congestion signals for the algorithms that run in CCP. The datapath also provides interfaces for CCP algorithms to set the congestion window and pacing rate, and express control patterns; these patterns can control transmission times and specify the times at which congestion signals are delivered to CCP.

2.1 CCP-Datapath Isolation

Should CCP run in the same address space as the datapath or not? There are some trade-offs involved in this decision. On the plus side, if run in the same address space, then information could be exchanged between them nearly instantaneously at

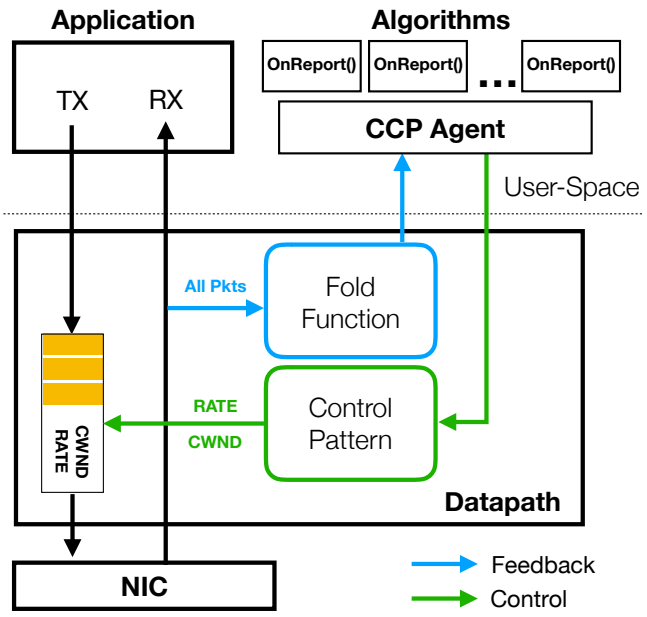


Figure 2: Congestion control algorithms in CCP are distinct from the application and datapath. Users specify control patterns to set the congestion window or rate, and fold functions to define how to summarize datapath messages into reports processed in CCP.

high bandwidth, and a congestion signal could be sent from the datapath to CCP every time feedback from the receiver arrives. But there are two drawbacks to this approach:

Safety. Bugs in CCP or in the algorithm's code could cause datapath crashes, or cause vulnerabilities that trigger privilege escalation attacks on kernel datapaths. To cope, at a minimum, we would have to enforce restrictions on the kernel functions that CCP can use, and adopt methods from recent research on kernel software fault isolation (LXFI [31]). This approach will have a non-trivial performance impact, quite likely defeating the purpose of running CCP in the same address space as the datapath. This approach also requires a careful annotation of kernel functions, and the specification of principals and kernel API integrity rules, a challenging task because the network stack has a large number of functions with subtle behaviors.

Flexibility. In the future, we anticipate running CCP on a different machine from the sender to centralize congestion control policies across groups of hosts. The same-address-space CCP would require a substantial re-design for this mode of operation.

Our design supports both modes, but we focus on how to achieve high performance and fidelity when CCP is in a different address space, including the important case when the datapath is kernel TCP and CCP is in user 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 TCP datapath and user-space CCP, compared to kernel TCP throughput. Per-ACK feedback (0 μs interval) reduces throughput by 32%, a significant amount, while using a 10 ms reporting interval achieves almost identical throughput to the kernel. Results as the number of flows increases are in §6.2.

When CCP is in a different address space, providing per-ACK notifications from the datapath to CCP incurs high overhead, as shown in Figure 1. This experiment shows that for a saturating iperf connection over a loopback interface, the Linux kernel TCP on this machine (with 4 2.8 Ghz cores) achieves 45 Gbit/s running Reno, whereas per-ACK reporting from the kernel datapath to CCP running Reno achieves only 68% of the throughput of the kernel. One way to improve performance is to increase the time between reports sent to CCP. The "Per 10 ms" bar shows that a less frequent reporting period achieves close to the kernel's throughput.

Given that CCP should report measurements only infrequently, a key question is how best to *summarize* congestion signals on the datapath so that CCP algorithms can achieve high fidelity compared to a baseline that implements the algorithm within the datapath. Although we are optimistic that a good design can achieve high fidelity because the natural timescale for end-to-end congestion control is the RTT between sender and receiver, achieving it requires a careful design of the information channel between datapath and CCP.

In §6.1 we show that using a larger reporting period does not affect the fidelity of CCP algorithm implementations relative to native datapath ones.

2.2 Exercising Control over Datapath Functions

Congestion-control algorithms written in CCP are not tied to the traditional "ACK clock", but rather operate on summaries that encapsulate observations over multiple ACKs received by the datapath sender. CCP algorithms specify datapath behavior using two mechanisms: *fold functions* and *control patterns*, whose directives are interpreted by the datapath to summarize congestion signals sent to CCP and control packet transmissions on the datapath, respectively.

Congestion signals and fold functions. All CCP-compatible datapaths should export a well-defined set of congestion signals (Table 2). There are two kinds of congestion signals: `Ack` signals, which are computed directly on each ACK (e.g., bytes acked in order, bytes acked out of order, RTT sample, etc.) and `Flow` signals, which are statistics maintained for the flow (e.g., outgoing and incoming rates). Most of these signals are updated on the arrival of an ACK; the exceptions are:

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

Table 2: Congestion signals that datapaths should report.

Class	Operations
Arithmetic	+,−,∗, /, EWMA
Comparison	==,<,>
Conditionals	If (branching)
Assignment	:=

Table 3: Fold function operators that a datapath must support.

`was_timeout` and `lost_pkts_sample` updated on a timeout; and `bytes_in_flight` and `packets_in_flight` updated on packet transmissions.

CCP may read but not write these signals. The only way to gain access to them is via a fold function, which specifies how the datapath should summarize the congestion signals in a single reporting period.

Fold functions express simple operations over the congestion signal primitives. Table 3 contains the operations each datapath should be able to compute. A CCP algorithm expresses fold functions in a small, restrictive domain-specific language. Each datapath must support this language. For software datapaths, we have developed `libccp`, a library that provides a reference implementation of fold function computations in this language and other features common to all datapaths (see §3.3).

The fold function defines a structure called `Summary`. This structure is maintained by the datapath and encapsulates all measurements reported to CCP by the datapath. The CCP algorithm specifies the fields of the `Summary` structure in the fold function and how to compute each field from congestion signals. The algorithm may define as many fields as it wants in the `Summary` structure. The datapath writes the values of these fields using the logic supplied by the CCP algorithm.

The following fold function shows an example of how to request the number of bytes delivered in order within the reporting period. The sender's datapath runs this function on each received ACK, `A`:

```
(def (Summary.acked 0))
(:= Summary.acked
  (+ Summary.acked A.bytes_acked))
```

In the example above, the fold function defines a field, `Summary.acked`, and specifies that on each invocation (typically every received ACK), it should be incremented by the number of in-order bytes observed since the last invocation. After delivering the summary, the datapath resets each reported `Summary` field to the specified initial value.

Control patterns. CCP algorithms can specify the congestion window or pacing rate using simple functions supported by the datapath. These can be set to either absolute values or relative to current values. The algorithms can also specify timing patterns according to which summary reports from fold functions and window/rate settings are executed by the datapath. The ability to specify timing patterns is useful for two reasons. First, algorithms may wish to specify patterns of sending; BBR [10], for example, sends pulses of different magnitudes for specific time durations. Second, algorithms may wish to specify the precise intervals over which congestion signals should be gathered and reported (e.g., once every half-RTT).

```
SetCwndAbs(...) => WaitRtts(0.5) => Report()
```

Control patterns use `=>` to specify that the directive on the right should happen after the one on the left. This example pattern uses the `WaitRtts()` directive to specify that the datapath should set the congestion window to the specified value, wait for a half-RTT, and then report a summary to CCP. After this report, the pattern will reset Summary state and loop back to `SetCwndAbs()`.

Figure 2 shows the architecture of a CCP-enabled sender, highlighting how the components we have discussed in this section fit together.

3 Datapath Responsibilities

A CCP-compatible datapath must accurately enforce the congestion control algorithm specified by the user-space CCP module. While algorithms themselves are free from considering datapath-specific concerns, from the datapath's perspective it is almost as if the developer were writing their algorithm directly into the datapath. Furthermore, once a datapath implements support for CCP, it automatically enables all CCP algorithms.

A CCP datapath has three responsibilities:

(1) It should communicate with the CCP agent (running on the same host) using an inter-process communication (IPC) mechanism. CCP expects datapaths to multiplex messages for different flows onto a single persistent IPC connection. Datapaths should assign flow IDs for CCP to demultiplex information.

(2) To accurately execute the algorithm, datapaths should enforce the congestion windows and pacing rates specified by the control pattern.

(3) Datapaths should update fold function calculations soon after new information arrives (i.e., upon every ACK)

and report these summaries to CCP as specified by the control pattern.

3.1 Control Patterns

The datapath must observe any congestion windows and pacing rates CCP algorithms specify. The datapath should independently enforce the congestion window and pacing rate unless a CCP algorithm's directives indicate otherwise. Independent enforcement can allow algorithms to specify, for example, a congestion window with a rate cap to prevent bursty transmission. We excerpt the CCP Copa implementation (§4.1) to demonstrate this:

```
SetRateAbs(...) => SetCwndAbs(...) =>
   WaitRtts(0.25) => Report()
```

3.2 Fold Function Support

Fold functions are loop-free collections of arithmetic operations over congestion signals specified by the CCP. While the CCP will compile fold functions into a datapath-compatible format and check for common errors (e.g., use of undefined variables) before installing them in the datapath, it is the datapath's responsibility to ensure safe interpretation of fold functions; for example, datapaths should prevent divide by zero errors when calculating congestion signals and guarantee that fold functions cannot overwrite the congestion primitives.

Table 2 specifies the signals our datapaths currently implement. The more signals a datapath can measure, the more algorithms that datapath can support. For example, CCP can only support DCTCP [1] or ABC [19] on datapaths that provide ECN support; CCP will not run algorithms on datapaths lacking support for that algorithm's requisite primitives. In the future, we imagine the list of primitives could extend beyond TCP to algorithms such as XCP [27].

After every ACK, the datapath must update the signal values, run the fold function, and send summary results to CCP as specified by the control patterns. Fold functions can also specify certain urgent conditions upon which the datapath should bypass the reporting interval specified in the control pattern and immediately send a summary to CCP. For example, a loss-based controller can react promptly to a packet loss with this mechanism:

```
(def (Summary.loss 0))
  (:= Summary.loss
    (+ Summary.loss A.lost_pkts_sample))
(:= isUrgent (> Summary.loss 0))
```

Fold functions also allow per-ACK modification of the congestion window. We discuss this feature in §4.2.

3.3 Implementation

We have implemented a library, `libccp`, which provides a reference implementation of CCP fold function computations and message serialization for software datapaths. 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 demultiplexes the message for the correct flow. It then runs the control pattern, which uses the datapath callbacks. After updating congestion signals, the datapath can call `ccp_invoke()`, which automatically computes the fold functions and sends the summaries to CCP. It is the responsibility of the datapath to ensure that it correctly computes and provides the congestion signals defined in Table 2.

We use `libccp` to implement CCP support for three datapaths: the Linux kernel, QUIC, and mTCP (a user-space TCP stack on top of DPDK).

For the Linux kernel and QUIC datapaths, we implemented CCP support through their pluggable congestion control interfaces. Both the kernel and QUIC support setting congestion windows and pacing rates. Both datapaths already expose most of the congestion signals defined in Table 2. QUIC needs to maintain additional state per flow as well as communicate with the user-space CCP module. We modified QUIC to set up a persistent Unix-domain socket on initialization for communication, and expose the function callbacks specified by the `libccp` API. We patched the Linux kernel to expose both transmission and delivery rates, and communicate with the user-space through Netlink sockets [41].

Unlike QUIC and the Linux kernel, mTCP only implements Reno and does not explicitly expose a congestion control interface for new algorithms. Hence, we modified the mTCP transport code directly: we added pacing support, implemented SACK, and made sure it keeps the right number of packets in flight during loss recovery. mTCP does not implement SACK or packet pacing, and when it observes a loss (defined as a triple duplicate ACK), it retransmits all packets in sequence starting from the lost packet, even if only a single packet was lost. As a result, when initially testing our implementation of Cubic, each time the buffer was filled, CCP would cut the window, but when the cumulative ACK finally advanced, mTCP would immediately burst out a congestion window worth of packets (most of which had already been successfully received and SACKed), forcing another drop, and so on. Our changes enabled our mTCP datapath to expose a consistent behavior as the other datapaths. (§5),

4 Writing Algorithms in CCP

A CCP algorithm specifies three functions: `create`, `onReport`, and `controlPattern`. When an application initiates a new flow, the datapath sends a message to CCP that causes `create` to be called. In this function, the algorithm can initialize its state using information (e.g., MSS) provided by the datapath and specify a fold function for the datapath, as shown in the code snippet below. This fold example calculates the number of bytes delivered (and ACK'd) in order and sets `isUrgent`

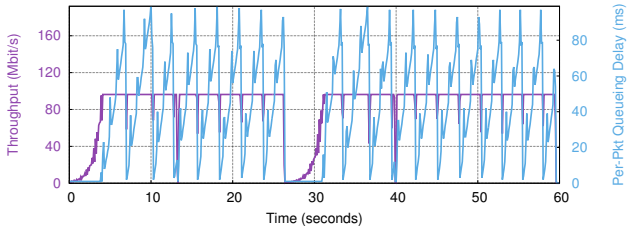


Figure 3: Our CCP implementation of BBR used for a bulk transfer over a 96 Mbit/s link with a 100 ms RTT. The bandwidth probe phase can be seen in the oscillation of the queueing delay, and the RTT probe phase can be seen in the periodic dips in throughput.

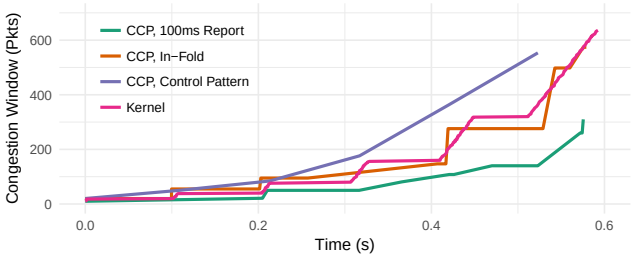


Figure 4: Different implementations of slow start have different window update characteristics. The control pattern implementation is rate-based, so we show the congestion window corresponding to the achieved throughput over each RTT.

to `True` if the flow has experienced any loss. Recall that the summaries are reset to their initial values after every report.

```
fn create(...) -> Self {
  self.scope = datapath.install_fold(info.sock_id, "
     (def (Flow.inorder 0) (Flow.loss 0))
     (:= Flow.inorder (+ Flow.acked Pkt.bytes_acked))
     (:= Flow.loss (+ Flow.loss Pkt.lost_pkts_sample))
     (:= isUrgent (> Flow.loss 0))
  ");
  // more initialization
}
```

We show examples of `onReport` and `controlPattern` later.

4.1 Example: BBR

We discuss how CCP allows a designer to focus on the essential features of their algorithm by outlining the implementation of BBR [10]. BBR is a rate-based scheme that models the network as a single bottleneck link. The sender probes for bandwidth in pulses of 1.25× and 0.75× of the current rate, setting its estimate of the bottleneck rate to a 10-second windowed maximum of the observed rate. If no new minimum RTT is reported for 10 seconds, BBR probes the minimum RTT by briefly draining the queue. BBR caps the congestion window to a fixed multiple (`cwnd_gain`; recommended value is 2) of the "pipe size" (the product of the observed minimum RTT and estimated link rate).

BBR is expressed succinctly with this control pattern:

```
SetRateAbs(1.25 * bottleneck_estimate) =>
SetCwndAbs(bottleneck_estimate * min_rtt
  * cwnd_gain) =>
WaitRtts(1.0) =>
SetRateAbs(0.75 * bottleneck_estimate) =>
WaitRtts(1.0) =>
SetRateAbs(bottleneck_estimate)
```

Here, `bottleneck_estimate`, `min_rtt`, and `cwnd_gain` are variables maintained in the CCP algorithm code, while the `Set...` and `WaitRTT` functions are interfaces provided by the datapath.

To estimate the bottleneck rate, BBR uses a windowed maximum observed rate. For this purpose, the datapath must maintain estimates of the outgoing and incoming rates. The following fold function expresses how this signal as well as the minimum RTT are calculated:

```
(def
   (Summary.bottleneck_estimate 0)
   (Summary.minrtt +infinity))
(:= Summary.minrtt
   (min Summary.minrtt A.rtt_sample_us))
(:= Summary.bottleneck_estimate
   (max Flow.rate_outgoing Flow.rate_incoming))
```

We implement the windowed min in the `OnReport()` handler in the CCP algorithm, not in the fold function. In Figure 3, we show the throughput and queueing delay achieved by our implementation.[1] Both the characteristic BBR pulse, achieved from the control pattern, and the 10-second periodic probe-RTT mode, are evident.

4.2 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 multiplicately 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. With CCP, one way to implement slow start is to retain the logic entirely in the CCP algorithm code, and measure the size of the required window update with a fold function. We show an example in Listing 1. This implementation strategy is semantically closest in behavior to native datapath implementations.

[1] We implemented only the bandwidth and RTT probing phases of BBR.

```
fn create(...) {
 datapath.install_fold(sock_id, "
 (def (Summary.acked 0) (Summary.loss 0))
 (:= Summary.acked (+ Summary.acked A.bytes_acked))
 ");
 datapath.install_pattern(
  SetCwndAbs(init_cwnd) => WaitRtts(1.0) => Report());
}
fn onReport(...) {
 let acked = report.get_field("Summary.acked");
 self.cwnd += acked;
 datapath.install_pattern(
  SetCwndAbs(self.cwnd) => WaitRtts(1.0) => Report());
}
```

Listing 1: A CCP implementation of slow start (ending slow start not shown).

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 4 demonstrates, on a 48 Mbps, 100 ms RTT link, different implementations of slow start exhibit differing window updates relative to the Linux kernel baseline. An implementation with a 1-RTT reporting period lags behind the kernel, but it is also possible to implement slow start within the fold function (Listing 2), or in a control pattern:

```
WaitRtts(1.0) => SetRateRel(2.0)
```

As outlined in §2, the programming model of fold functions 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.

Meanwhile, a control pattern is an easy way to program static behavior into the datapath, but is inflexible: CCP algorithms implementing slow start in a control pattern must be careful to exit slow start at the correct time; the control pattern will continue to scale the sending rate exponentially until it is replaced.

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. It is important to note, however, that some configurations, e.g., making updates to the congestion window both in the fold function as well as via the send pattern makes the resulting algorithm difficult to reason about since the updates are not synchronized. Ultimately, users will be able to choose the algorithm implementation best suited to their congestion control logic and application needs.

```
fn create(...) {
 datapath.install_fold(sock_id, "
 (def (Summary.loss 0))
 (:= Cwnd (+ Cwnd A.bytes_acked))
 (:= isUrgent (> A.lost_packets 0))
 ");
}
fn onReport(...) {
 // exit slow start
}
```

Listing 2: A within-fold implementation of slow start. Note that CCP algorithm code is not invoked at all until the connection experiences its first loss.

5 New Capabilities

We present four new capabilities enabled by CCP: new classes of congestion control, 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.

5.1 New Algorithms

CCP enables a broad new class of congestion control algorithms. Until now, because congestion control has been embedded in the datapath, algorithms have been strictly tied to the ACK clock. As a result, as data rates rise, algorithms have less time to make decisions, which naturally restricts the computation an algorithm can perform. By contrast, CCP algorithms are decoupled from the ACK clock. Although congestion signals must still be acquired and summarized at ever-faster rates, CCP algorithms can to amortize computation across different time scales, and use sophisticated data structures and computations that may be unwise on datapaths like the kernel.

One such algorithmic technique, proposed in concurrently submitted work[2], uses CCP to compute Fourier transforms on estimates of cross traffic and infer whether it is elastic (buffer-filling) or not. A key component of this technique 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 series and infers elasticity if the FFT at particular frequencies is large.

In principle, it may be possible to implement such sinusoidal pulses within the datapath, but computing the FFT is unlikely to be practical both due to the timing requirements of the ACK clock as well as the difficulty of performing complex calculations in restrictive programming environments such as the kernel. Moreover, implementing the transmitter outside the

[2]Paper #97: Elasticity Detection: A Building Block for Delay-Sensitive Congestion Control

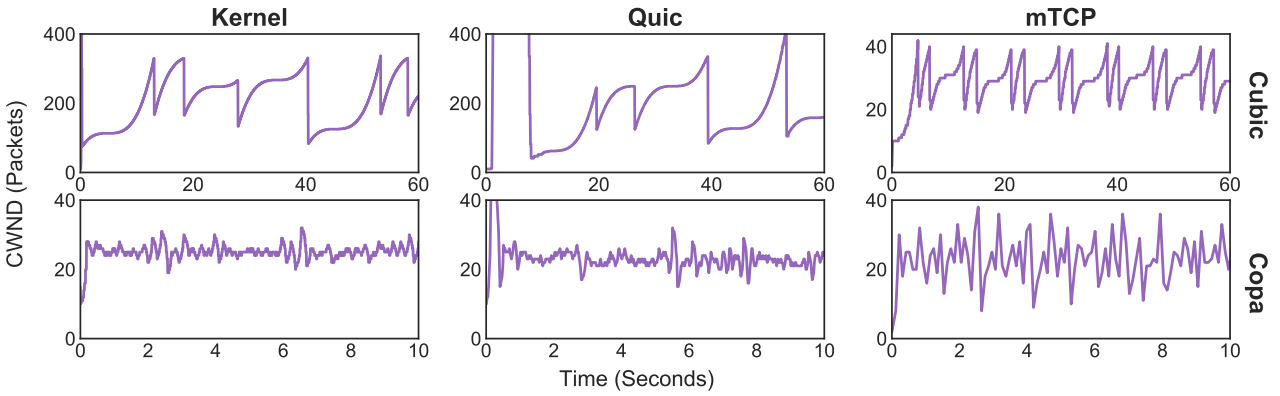


Figure 5: Comparison of *the same* CCP implementation of Cubic and Copa run on three *different* datapaths with a fixed-rate bottleneck link.

datapath in CCP allowed the researchers to iterate through many ideas more quickly than if they had to write 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.

5.2 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. Under three network models, this is shown to provide quantifiably low delay, and fair and efficient allocation of bandwidth. 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.

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 fold function, 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×.

5.3 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. This per-connection paradigm for congestion control has served the Internet well for 30 years, but is a poor fit in many modern traffic control scenarios where a large number of nodes exchange traffic between a few sites. Examples include: traffic between different datacenters [23, 24]; traffic between campuses of an organization; traffic between collaborating universities or companies; traffic between a large content provider (e.g., Netflix) and a network with many clients (e.g., a regional ISP); large-scale data backups from an organization to an external site; etc.

A more modest use case is to implement the per-host aggregation proposed by the Congestion Manager two decades ago [4]. We used CCP to develop 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 minimal changes to 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 (§6). 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 6 shows the aggregator instantaneously apportioning equal windows to each flow in its domain.

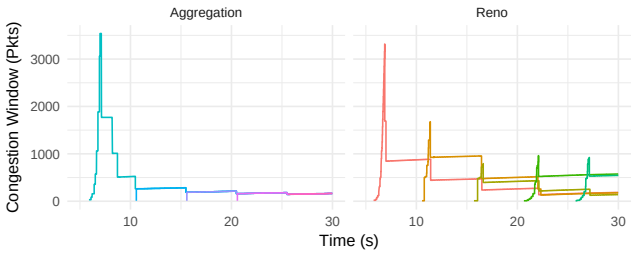


Figure 6: 5 20-second iperf flows with 10 second staggered starts. While Reno (right) must individually probe for bandwidth for each new connection, and aggregating congestion controller is able to immediately set the connection's congestion window to the fair share value.

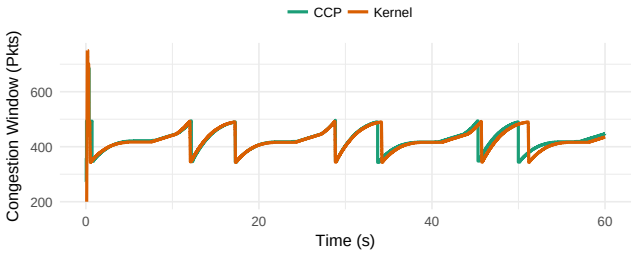


Figure 7: Cubic in CCP matches Cubic in Linux TCP.

5.4 Write-Once, Run-Anywhere

Correct implementation of congestion control algorithms, especially increasingly complicated recent work, is a significant undertaking. As a result, 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.

To exhibit this capability, we ran the same implementation of both Cubic (not currently implemented in mTCP) and Copa (§5.2, not currently implemented in any major datapath) on the three different datapaths and plot the congestion window evolution over time in Figure 5.

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.

6 Evaluation

We implement a user-space CCP in Rust, called Portus, which implements functionality common across independent congestion control algorithm implementations, including a compiler for the fold function language and a serialization library for IPC communication. We also provide a module for common congestion control calculations such as slow start. CCP congestion control algorithms are hence also implemented in Rust; we plan to build Portus bindings for other high-level languages as well.

Unless otherwise specified, we evaluated using Linux 4.14.0 (with a 4-line patch to expose rate measurements to CCP) on a machine with four 2.8 Ghz cores and 64 GB memory.

Fidelity (§6.1). 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 (§6.2). How expensive is communication between CCP and the datapath?

Low-RTT simulations (§6.3). We evaluate CCP in a simulated low-RTT datacenter-like environment.

6.1 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 7 shows one such comparison of congestion window update decisions over time on an emulated 96 Mbit/s fixed-rate Mahimahi [35] 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.

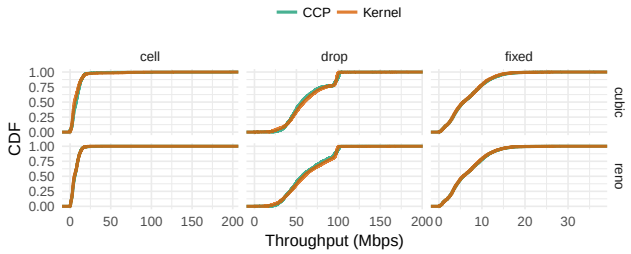
For the remainder of this subsection, we compare the performance of CCP and kernel implementations of NewReno and Cubic on three metrics (throughput and delay in §6.1.1, and FCT in §6.1.2) and three scenarios, all using Mahimahi.

6.1.1 Throughput and Delay We study the following scenarios:

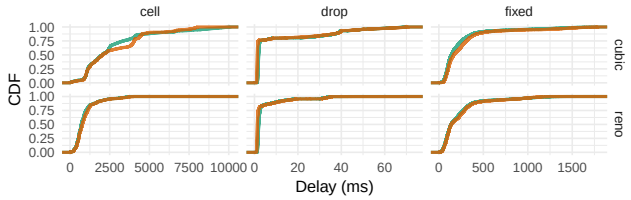
Fixed-rate link ("fixed"). A 20 ms RTT link with a fixed 96 Mbit/s rate and 1 BDP of buffering.

Cellular link ("cell"). A 20 ms RTT variable-rate link with a 100-packet buffer based on a Verizon LTE bandwidth trace [35].

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.

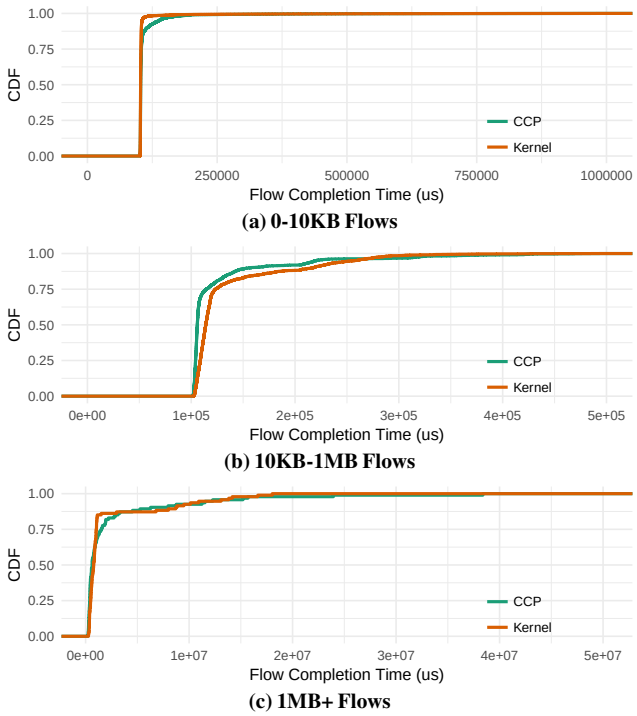


(a) Achieved throughput over 1 RTT periods. Note the different scales on the x-axes for the three scenarios.



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

Figure 8: 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.



(a) 0-10KB Flows

(b) 10KB-1MB Flows

(c) 1MB+ Flows

Figure 9: CDF comparisons of flow completion times. Note the differing x-axes.

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 (8a) and delay (8b), and show the ensuing distributions in Figure 8.

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

6.1.2 Flow Completion Time To measure flow completion times (FCT), we use a flow size distribution compiled from CAIDA Internet traces [8] in a similar setting to the "fixed" scenario above, modified to form an incast scenario; we use a 100 ms RTT and a 192 Mbit/s link. To generate traffic, we use an empirical traffic generator to sample flow sizes from the distribution and send flows of that size in an incast traffic pattern to a single client behind the emulated Mahimahi link. We generate flows with 50% link load, and generate 100,000 flows to the client in a closed loop 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.

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 9a, 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 9b 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 9c).

6.2 Performance

6.2.1 Measurement Staleness Because Portus runs in a different address space than datapath code, there is some delay between the datapath gathering a summary and algorithm code acting upon the report due to overheads in inter-process communication (IPC). In the worst case, a severely delayed

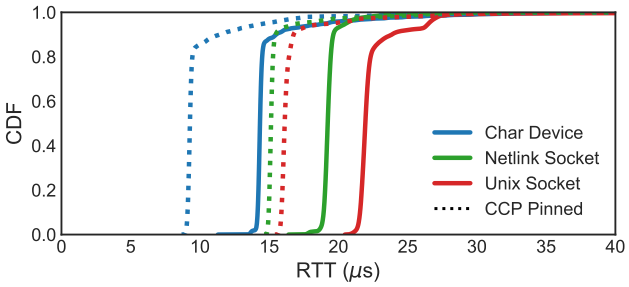


Figure 10: Minimum time required to send information to the datapath and receive a response using different IPC mechanisms. The dotted lines show latency when the process is bound to a single CPU core.

measurement could cause an algorithm to make an erroneous window update.

Fortunately, as Figure 10 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 [39], a convenient and popular IPC mechanism used for communication between user-space processes; Netlink sockets [41], 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. If the sending process is pinned to a CPU core to avoid process scheduling overheads, the 95th percentile drops to 20 μs.

6.2.2 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 1024. In both scenarios, we test CCP's implementation of Reno against the Linux kernel's. We measure average throughput and CPU utilization in 1 second intervals over the course of 10 15-second experiments using iperf [44]. We use a reporting interval of 10 ms.

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 11a, CCP matches the kernel's throughput up to the maximum number of flows tested, 1024.

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 11b 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 64 and 256 flows, where CCP uses 2.0$\times$ as much CPU than the kernel on average.

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.

6.3 Low-RTT and High Bandwidth Paths

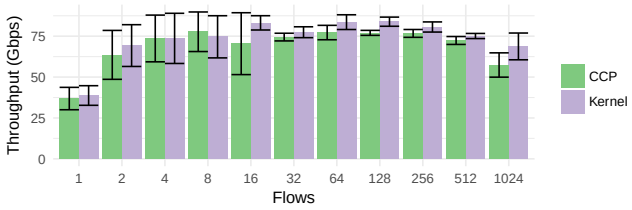
To demonstrate it is feasible to separate congestion control from the datapath even in low-RTT and high bandwidth situations, we simulate a datacenter incast scenario using ns-2 [37]. We model CCP by imposing both forms of delays due to CCP: (i) the period with which actions can be taken (the reporting period) and, (ii) the staleness after which sent messages arrive in CCP. We used our microbenchmarks in §6.2.1 to set the staleness to 20μs, 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 ($<$10μs 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 12 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$\times$ the RTT; even this infrequent reporting period yields only minor degradation.

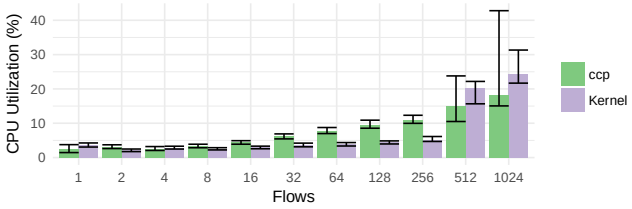
Meanwhile, at 40 Gbit/s the slowdown over the baseline increases with the reporting interval in the case of 100 packet flows, but not with 10 or 1000 packet flows. Similar to the results in §6.1.2, the short flows and long flows are both unaffected by the reporting period because the short flows complete 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.

7 Related Work

eBPF [14] allows developers to define programs that can be safely executed in the Linux kernel. These programs can be attached to kernel functions and used for debugging. We considered compiling CCP's user defined fold functions into eBPF instructions to ensure safe execution in the kernel datapath. However, the congestion signals in Table 2 are not easily accessible by eBPF programs. In addition, implementing our

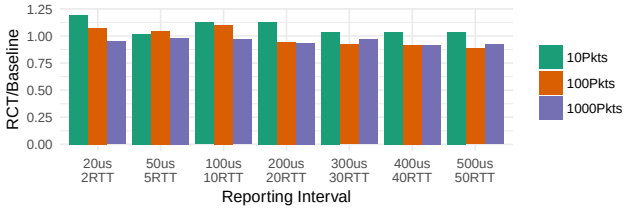


(a) Achieved localhost throughput as the number of flows increases

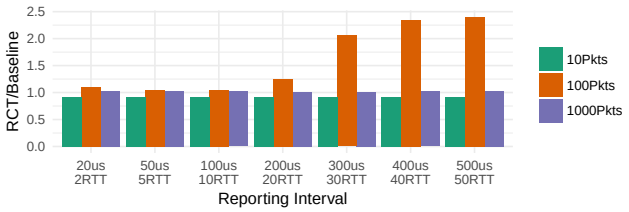


(b) CPU Utilization when saturating a 10 Gbit/s link. Error bars show standard deviation.

Figure 11: CCP can handle many concurrent flows without significant CPU overhead.



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



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

Figure 12: 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.

own fold function language within `libccp` allows portability to other datapaths.

Linux includes a pluggable TCP API [11], 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 [28] also offers pluggable congestion control. CCP extracts common primitives for congestion control across multiple datapaths, supporting off-datapath programmability with the performance of in-datapath implementations.

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. By 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). In addition, the congestion signals exposed by CCP datapaths are not tied to the ACK clock, allowing users to implement a wider class of algorithms.

Structured Streams [17] is a datapath that proposes preventing head-of-line blocking by managing streams between the the same two hosts together, and applying a hereditary structure on streams. CCP does not change the underlying structure of the transport on the datapath; the Linux kernel datapath, for example, uses TCP streams. The congestion manager (CM [4]) introduces an in-kernel agent to perform congestion control operations for a group of flows together. CCP is the first system to both provide flow aggregation capabilities *and* an API for developers to develop novel algorithms, and it does so with few changes to the datapath.

There is a wide range of previous literature on moving kernel functionality into user-space. The Exokernel [16] is an operating system architecture that securely exports hardware functionality into untrusted user-space library operating systems. Arrakis [38] 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 IP and TCP layer 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 support the CCP API as well.

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).

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-NV: Congestion Avoidance for Data Centers. *Linux Plumbers Conference*, 2010.

[7] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

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

[9] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, 2004.

[10] 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.

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

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

[13] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional Rate Reduction for TCP. In *SIGCOMM*, 2011.

[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] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*, 1995.

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

[18] C. P. Fu and S. C. Liew. TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks. In *IEEE JSAC*, volume 21, 2003.

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

[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] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*, 2013.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *SIGCOMM*, 2013.

[25] 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.

[26] A. B. Johnston and D. C. Burnett. *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web*. Digital Codex LLC, 2012.

[27] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[28] 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.

[29] D. Leith and R. Shorten. H-TCP Protocol for High-Speed Long Distance Networks. In *PFLDNet*, 2004.

[30] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 65(6):417–440, 2008.

[31] 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.

[32] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *MobiCom*, 2001.

[33] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[34] A. Narayan, F. J. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan. The Case for Moving Congestion Control Out of the Datapath. In *HotNets*, 2017.

[35] 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.

[36] Netronome. Agilio LX SmartNICs. https://www.netronome.com/products/agilio-lx/. [Online, Retrieved July 28, 2017].

[37] The Network Simulator. https://www.isi.edu/nsnam/ns/.

[38] 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.

[39] D. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63, 1984.

[40] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[41] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol, 2003. RFC 3819, IETF.

[42] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT), 2012. RFC 6817, IETF.

[43] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

[44] 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.

[45] D. Wei, C. Jin, S. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking*, 14(6):1246–1259, 2006.

[46] K. Winstein and H. Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, 2012.

[47] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*, 2013.

[48] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013.

[49] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011.

[50] 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.