The Case for Moving Congestion Control Out of the Datapath

Akshay Narayan, Frank Cangialosi, Prateesh Goyal,
Srinivas Narayana, Mohammad Alizadeh, Hari Balakrishnan
{akshayn,frankc,prateesh,alephtwo,alizadeh,hari}@csail.mit.edu
MIT CSAIL

Abstract

With Moore's law ending, the gap between general-purpose processor speeds and network link rates is widening. This trend has led to new packet-processing "datapaths" in endpoints, including kernel bypass software and emerging SmartNIC hardware. In addition, several applications are rolling out their own protocols atop UDP (e.g., QUIC, WebRTC, Mosh, etc.), forming new datapaths different from the traditional kernel TCP stack. All these datapaths require congestion control, but they must implement it separately because it is not possible to reuse the kernel's TCP implementations. This paper proposes moving congestion control from the datapath into a separate agent. This agent, which we call the *congestion control plane* (CCP), must provide both an expressive congestion control API as well as a specification for datapath designers to implement and deploy CCP. We propose an API for congestion control, datapath primitives, and a user-space agent design that uses a batching method to communicate with the datapath. Our approach promises to preserve the behavior and performance of in-datapath implementations while making it significantly easier to implement and deploy new congestion control algorithms.

1 Introduction

Traditionally, applications have used the networking stack provided by the operating system through the socket API. Recently, however, there has been a proliferation of new networking stacks as NICs have offered more custom features such as kernel bypass and as application developers have come to demand specialized features from the network [2, 11, 28, 32, 44, 49]. We refer to such modules that provide interfaces for data movement between applications and network hardware as *datapaths*. Examples of datapaths include not only the kernel [17], but also kernel-bypass methods [2, 28, 44], RDMA [49], user-space libraries [32], and emerging programmable NICs ("SmartNICs" [39]).

The proliferation of datapaths has created a significant challenge for deploying congestion control algorithms. For each new datapath, developers must re-implement their desired congestion control algorithms essentially from scratch—a difficult and tedious task. For example, implementing TCP on Intel's DPDK is a significant undertaking [28]. We expect this situation to worsen with the emergence of new hardware accelerators and programmable NICs because many such high-speed datapaths forego programming convenience for performance.

Meanwhile, the diversity of congestion control algorithms—a key component of transport layers that continues to see innovation and evolution—makes supporting all algorithms across all datapaths even harder. The Linux kernel alone implements over a dozen [12, 24, 26, 27, 33, 34] algorithms, and many new proposals have emerged in only the last few years [8, 13, 19, 22, 31, 35, 36, 47, 48]. As hardware datapaths become more widely deployed, continuing down the current path will erode this rich ecosystem of congestion control algorithms because NIC hardware designers will tend to bake-in only a select few schemes into the datapath.

We believe it is therefore time for a new *datapath-agnostic* architecture for endpoint congestion control. We propose to *decouple* congestion control from the datapath and relocate it to a separate agent that we call the *congestion control plane* (CCP). CCP runs as a user-space program and communicates asynchronously with the datapath to direct congestion control decisions. Our goal is to identify a narrow API for congestion control that developers can use to specify flexible congestion control logic, and that datapath developers can implement to support a variety of congestion control schemes. This narrow API enables congestion control algorithms and datapaths to evolve independently, providing three key benefits:

Write once, run everywhere. With CCP, congestion control researchers will be able to write a single piece of software and run it on multiple datapaths, including those yet to be invented. Additionally, datapath developers can focus on implementing a standard set of primitives—many of which are already used today—rather than worry about constantly evolving congestion control algorithms. Once an algorithm is implemented with the CCP API, running on new CCP-conformant datapaths will be automatic.

Ease of programming. Locating CCP in user-space and designing a convenient API for writing congestion control

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotNets-XVI, November 30-December 1, 2017, Palo Alto, CA, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-5569-8/17/11. . . $15.00
https://doi.org/10.1145/3152434.3152438

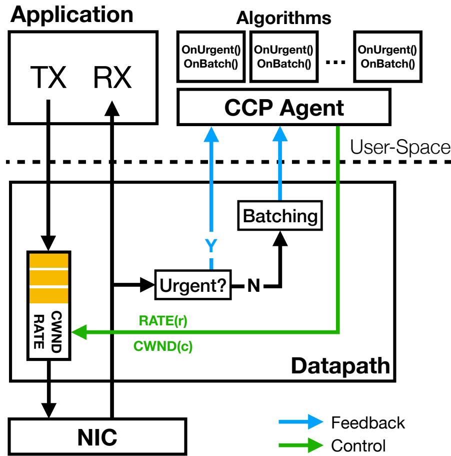


Figure 1: Overview of CCP and application interactions with the datapath. Both the application and CCP operate in user-space, while the datapath may exist in user-space, kernel-space, or hardware. If a packet contains urgent information, e.g., a congestion event, the datapath contacts CCP immediately. Otherwise, it batches user-specified measurements and sends them asynchronously. A CCP algorithm implements a handler for each of these cases. The datapath enforces the rate or congestion window it receives from CCP. The application's interface to the datapath (e.g., POSIX socket API) is unmodified.

algorithms will make it easier to develop and deploy new schemes. This will ease the process of implementing and evaluating novel algorithms. Furthermore, algorithm developers will be free to utilize powerful user-space libraries (e.g., neural nets) and focus on the details of their methods rather than learning low-level datapath APIs.

Performance without compromises. As NIC line rates steadily march upwards to 100 Gbit/s and beyond, it is desirable to offload transport-layer packet processing to hardware to save CPU cycles. Unlike solutions that hard-code a congestion control algorithm in hardware, e.g., [49], removing congestion control from the datapath will enable richer congestion control algorithms while retaining the performance of fast datapaths.

The key challenge in providing such flexibility is achieving good performance. Congestion control schemes traditionally process every incoming ACK, but doing so, CCP would incur unacceptable overheads. We propose to tackle this problem using a new flexible batching method to summarize and communicate information between the datapath and CCP once or twice per RTT rather than on every ACK. We show that this approach can achieve behavior close to native datapath implementations since the fundamental time-scale for end-to-end congestion control is an RTT. To our knowledge, CCP is the first such off-datapath congestion-control architecture that does not process every single packet or ACK.

2 CCP Design

Our system architecture (Figure 1) consists of three components: the congestion control algorithm, the CCP agent, and the CCP modification to the datapath. In the discussion below, we refer to the former two components together as the "CCP." Our architecture does not modify the API between applications and datapaths (e.g., POSIX sockets); applications can run unmodified.

Congestion control algorithms are implemented in user-space and make all important congestion control decisions. The algorithm is independent of the datapath. Developers are free to add new algorithm implementations, and it is possible to run multiple algorithms on the same host, e.g., file downloads and video calls could use different transmission algorithms.

The agent is the "glue" between the congestion control algorithm and the datapath, and imposes policies on the decisions of the congestion control algorithms, e.g., per-connection maximum transmission rates. Crucially, neither the agent nor the algorithm is directly involved with packet transmission or reception. Instead, the CCP agent programs the (modified) datapath asynchronously using a well-defined API. We envision the agent running in user-space to support user-space datapaths and ease of programmability. It is also possible to run the agent in kernel-space when security is critical.

The modification to the datapath enforces the rates and window decisions it receives from the agent. On the receive path, it aggregates measurements from acknowledgments (e.g., for reliability, negative ACKs, etc.) and sends the results to the agent.

2.1 What is the API for congestion control?

Designing a flow-level CCP API is practical only if there is a common set of primitives which most congestion control algorithms build upon, and which datapaths can implement efficiently. In Table 1, we identify a set of control actions and packet measurements that enable the implementation of a wide variety of congestion control algorithms. Consequently, for each flow, we require that datapaths be able to maintain

(1) a given congestion window; and
(2) a given pacing rate on packet transmissions; and
(3) statistics on packet-level round trip times, packet delivery rates, and packet loss, and functions specified over them.

Indeed, many datapaths existing today implement—or can implement—these primitives internally (§4, §5). Making such datapaths CCP-compliant simply involves making these primitives externally programmable by the CCP.

Control. Congestion control algorithms specify a sending rate in one of two ways: either by directly setting a rate, or by setting a congestion window (CWND), which constrains the number of packets allowed to be in flight. Indeed, this second approach is analogous to setting rate $= \frac{\text{CWND}}{\text{RTT}}$.

Some new protocols send traffic in specific patterns. For example, BBR [13] is a rate-control algorithm which sends at $1.25\times$ the desired rate for an RTT, then at $0.75\times$ the rate for another RTT, and then exactly at the desired rate for 6 RTTs before repeating the pattern. Other algorithms are precise about the intervals over which they gather data: Sprout [48] models available network capacity using equally spaced rate measurements.

Hence, we allow developers to write a *control program* using a sequence of control primitives from Table 2. While it is technically possible for CCP to send these commands

Protocol	Measurement	Control Knobs
Reno [26]	ACKs	CWND
Vegas [12]	RTT	CWND
XCP [30]	Packet header	CWND
Cubic [24]	Loss, ACKs	CWND
DCTCP [8]	ECN, ACKs, Loss	CWND
Timely [35]	RTT	Rate
PCC [19]	Loss Rate, Sending Rate, Receiving Rate	Rate
NUMFabric [37]	Packet headers	Rate, Packet headers
Sprout [48]	Sending Rate, Receiving Rate, RTT	Rate
Remy [47]	Sending Rate, Receiving Rate, RTT	Rate
BBR [13]	Sending Rate, Receiving Rate, RTT	Rate (pulses), CWND cap

Table 1: Measurement and control primitives used by classic and modern congestion control algorithms.

Operation	Description
`Measure(·)`	Measure a per-packet metric
`Rate(r)`	`rate` $\leftarrow$ `r`
`Cwnd(c)`	`cwnd` $\leftarrow$ `c`
`Wait(time)`	Gather measurements
`WaitRtts(α)`	`Wait(α * rtt)`
`Report()`	Send measurements to the CCP

Table 2: Primitives in the control language.

Function	Description
`Init(seq, flow)`	Initialize flow state
`OnMeasurement(m)`	Measurements have arrived
`OnUrgent(type)`	An urgent event has occurred
`Install(p)`	Send new control program to the datapath

Table 3: CCP API. The first three functions are user-space event handlers implemented by a CCP algorithm. The last function is provided by CCP as a way for these handlers to modify sending behavior in the datapath.

out every RTT, the control program provides a way for the datapath to synchronize measurements with control actions in the datapath itself. For instance, it is important that BBR can measure increases in the achieved rate during the RTT period right after the high pulse:

```
Measure(rate).
Rate(1.25*r).WaitRtts(1.0).Report().
Rate(0.75*r).WaitRtts(1.0).Report().
Rate(rate).WaitRtts(6.0).Report()
```

Here the dot (·) in the syntax represents sequential execution, i.e., `A().B()` means "execute action `A()` followed by `B()`." Control programs allow developers to specify both the minute details of sending behavior and the precise intervals over which the datapath should gather measurements, without compromising performance. We expect that supporting the execution of simple control programs will be feasible in most datapaths, but it is also possible to support programs purely by issuing commands from the CCP each RTT.

Measurements. When gathering measurements, congestion control algorithms may take as input the packet-level RTT, achieved sending and receiving rates, and a limited number of congestion signals (triple duplicate ACKs, retransmit timeouts, and ECN). Additionally, some algorithms, such as XCP [30] and priority-dependent algorithms, take advantage of customized fields in packet headers. For example, XCP sends using the rate specified by the router in each packet.

Certain events, such as indications of congestion signals (e.g., packet loss, ECN), may be important enough to warrant immediate notification and processing. For this reason we classify measurements into two categories: *urgent* and *batched*. The datapath immediately reports urgent measurements to the CCP; it reports batched measurements at times specified in the control program. Urgent measurements allow the system to react quickly to network signals to avoid congestion collapse and unnecessary packet drops. Table 3 summarizes the three event handlers required to implement a congestion control algorithm in CCP.

2.2 Are CCP algorithms easier to write?

The CCP API allows developers to implement a variety of congestion control algorithms with relative ease. In contrast, kernel-space code must be trusted; as a result, writing kernel-space code for congestion control is complex. The kernel lacks support for floating point arithmetic, memory must be carefully managed, and exceptions from common errors (e.g., division by zero) will crash the operating system. We postulate that the difficulty of kernel programming and the lack of flexible APIs for other high-speed datapaths is a key reason why some new congestion control proposals like PCC [19] or Sprout [48] remain without high-speed implementations.

For example, the Linux kernel implements TCP Cubic's cube-root calculation in 42 lines of C using a lookup table followed by an iteration of the Newton-Raphson algorithm. We show the same per-packet `OnMeasurement` operation in CCP below, which can take advantage of convenient user-space floating point arithmetic packages and is thus simpler.

```
// Other state updates elided
// WlastMax is the window size at last drop
K = pow(max(0.0,
          (c.WlastMax-c.cwnd)/0.4),
       1.0/3.0)
// calculate and set CWND
c.cwnd = WlastMax + 0.4*pow(t-K, 3.0)
```

Furthermore, implementing new congestion control algorithms on emerging hardware datapaths like FPGAs [14] or SmartNICs [39] is likely to be even more challenging than writing kernel-space code. With CCP, hardware datapath

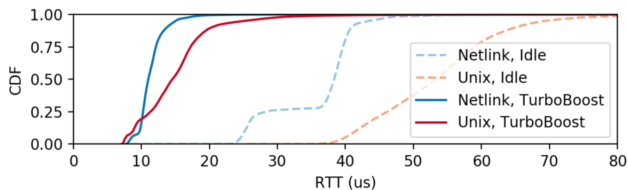


Figure 2: CDF of RTT between a Linux kernel module and user-space (using Netlink sockets) and between two user-space processes (using Unix domain sockets). When there is high CPU utilization, the CPU's clock speed temporarily increases due to Intel TurboBoost.

designers only need to support the CCP primitives to run a variety of CCP algorithms on their datapath.

2.3 Why batch measurements?

In our design, CCP modulates the packet transmission rate or window just a few times each RTT. Processing every acknowledgment in user-space would lead to much higher CPU utilization than necessary. Indeed, batching measurements is in line with optimizations that avoid per-packet processing of acknowledgments on the receive side, such as large receive offload (LRO [7]) and the Linux kernel's new interrupt API (NAPI [16]). For example, processing each acknowledgment (without batching) for a 100 Gbit/s stream with MTU sized packets requires processing 8 million acknowledgments per second. However, with per-RTT batching of acknowledgments, CCP only needs to process 100,000 batches per second at an RTT of 10 μs (e.g., in data centers). With an RTT of 100 ms (e.g., in the WAN), this number drops to 10. The saved CPU cycles can be returned to the application or used by complex congestion control algorithms.

Further, from a control theory perspective, to achieve stability it is sufficient to sample the feedback and actuate the control loop at a time scale comparable to the loop's feedback delay [40]. Since it takes an RTT to observe the effect of any action (e.g., rate update), the natural time scale for congestion control sensing and actions is an RTT.

Finally, it is feasible to batch measurements over RTT timescales. Figure 2 shows the distribution of round-trip times (RTTs) collected from two IPC mechanisms: Unix domain and Netlink sockets. The Unix domain socket measurements were between two user-space processes, while the Netlink measurements were between a Linux kernel module and a user-space process. Note that these numbers capture both latency in the stack (i.e., syscalls and copies) as well as process scheduling overheads. When the CPU is idle, across 60,000 samples the 99th percentile RTT is 48 μs for Netlink and 80 μs for Unix sockets. When the CPU is highly utilized and has Intel Turbo Boost [3] enabled, however, these RTTs drop significantly, with 99th percentile values of only 18 μs for Netlink and 35 μs for Unix sockets. These latencies for control decisions, even in the tail, are negligible compared to WAN RTTs, and may even be acceptable in many datacenter environments. In very low-latency datacenter networks (e.g., with 1-10 μs RTTs [23]), dedicating a CPU core to the CCP may reduce the overall impact on the effective RTT of congestion control, by avoiding the need for process scheduling. We discuss further strategies for very low-latency networks in §5.

2.4 How should datapaths batch measurements?

We describe two possible approaches to batching information from multiple packets in the datapath and illustrate the semantic differences between them in CCP with the example of TCP Vegas. Vegas increases the congestion window when it detects fewer than (say) two queued packets, and decreases the congestion window when there are greater than (say) four. We elide details such as slow start.

Vector of measurements. In the first approach, the datapath appends a small amount of developer-specified measurement data for each packet into a variable-length vector. The datapath sends this vector to the CCP at the time that the control program indicates through the `Report()` statement. The `OnMeasurement()` handler in the CCP is called with this vector as the argument:

```
func (v *Vegas) OnMeasurement(ps []Packet) {
  for p := range ps {
    v.baseRtt = min(v.baseRtt, p.Rtt)
    inQ = (p.Rtt-v.baseRtt)*v.cwnd/v.baseRtt
    if inQ < 2 {
       v.cwnd += 1
    } else if inQ > 4 {
       v.cwnd -= 1
    }
  }

  v.Install(Measure(rtt).
    Cwnd(v.cwnd).WaitRtts(1).Report())
}
```

In the latter half, the `Install()` statement instructs the datapath to measure the packet-level RTT, update its congestion window, and report its measurements after the next RTT.

With this simple approach, it is straightforward to port existing congestion control algorithms, which process packets synchronously in the datapath, to CCP implementations that process a batch of information at a time. Indeed, the vector-style implementation of TCP Vegas is similar to the Linux implementation.

Fold function over measurements. In the second approach, CCP instructs the datapath to summarize information from new packets into a constant amount of measurement state in the datapath. For each new packet, the fold function in the datapath takes an old measurement state and the new packet, and records the new measurement state. The datapath sends the measurement state to the `OnMeasurement()` CCP handler. For example, in TCP Vegas, we define and update `VegasState`, which consists of the minimum RTT and the increment to the congestion window computed from the previous round-trip time period:

```
func (v *Vegas) OnMeasurement(s VegasState) {
  v.cwnd += s.delta
  v.baseRtt = s.baseRtt
  // Define measurement fold parameters
  initState = VegasState {
```

```
    delta: 0, baseRtt: v.baseRtt
  }
  foldFn = func (new St) (old St, p Pkt) {
    new.baseRtt = min(
      old.baseRtt,
      p.Rtt,
    )
    inQ = ((p.Rtt-new.baseRtt) *
           v.cwnd / new.baseRtt)
    if inQ < 2 {
      new.delta = old.delta + 1
    } else if inQ > 4 {
      new.delta = old.delta - 1
    } else {
      new.delta = old.delta
    }
  }

  v.Install(Measure(initState, foldFn).
    Cwnd(v.cwnd).WaitRtts(1).Report())
}
```

While using such fold functions guarantees that the datapath uses a bounded amount of memory, it remains to be seen if this API can realize a support set of congestion control measurements. Fold functions will likely need to be written in a restrictive language to ensure portability across datapaths, particularly, constrained hardware-accelerated datapaths. Recent work in programming switches using high level languages [38, 45] suggests that it is possible for datapaths to implement complex fold functions which fit within tight timing budgets.

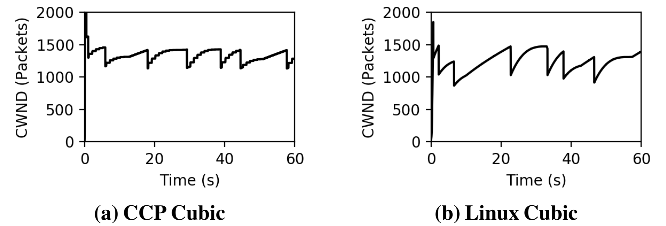
Takeaway. Using a vector of measurements in the CCP provides more flexibility than the folding approach, since all processing is in user-space. However, maintaining and processing vectors also imposes significant performance requirements on the CCP (i.e., per-packet work) and the datapath (i.e., need memory for each packet). Ultimately, the right batching approach will be dictated by the capabilities and flexibility of emerging datapaths.

3 Preliminary Evaluation

We have implemented a prototype CCP [1] in approximately 2500 lines of Go code and an associated Linux kernel datapath [4] in 900 lines of C. The kernel datapath is a kernel module which implements a set of Pluggable TCP callbacks (§4). To accurately report rates to the CCP, we additionally wrote a four line kernel patch on Linux 4.10 [5].

Our datapath implementation currently does not support user-defined measurements, user specification of urgent messages, or either event vectors or general fold functions. Rather, the prototype datapath reports only the most recent ACK and an EWMA-filtered RTT, sending rate, and receiving rate. The purpose of this implementation is to evaluate the feasibility of extracting congestion control logic from the datapath; the full design and implementation of a programmable datapath remains an avenue for future work.

Does CCP change the behavior of current algorithms? Figure 3 compares the congestion window evolution for TCP Cubic on a 1 Gbps link with an RTT of 10 ms and 1 BDP of buffer between the Linux implementation (in 3b) and the CCP implementation (in 3a). The CCP matches the microscopic window evolution of the Linux kernel. Furthermore, the macroscopic performance of the two implementations is similar; the Linux implementation achieved 94.4% utilization and 15.8 ms median RTT compared to the CCP implementation's 95.4% and 16.1 ms.



(a) CCP Cubic **(b) Linux Cubic**

Figure 3: Comparison of window dynamics of a CCP-based Cubic implementation and the Linux kernel implementation.

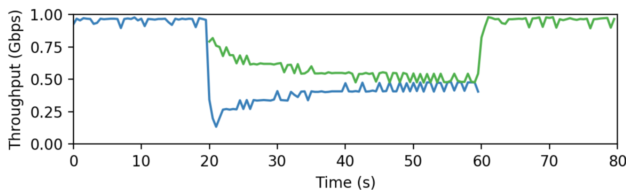
Does CCP change the performance of current algorithms? A potential downside of batching introduced by CCP is slow reactions to changing link conditions. In Figure 4 we start a 60-second CCP (4a) or Linux (4b) NewReno flow at time 0. After 20 seconds, we start a competing flow of the same type. Both implementations exhibit similar convergence dynamics.

Will CCP waste CPU cycles? CCP can *save* cycles by enabling the batch processing of packets. However, more concerning is the prospect of IPC overheads consuming extra CPU cycles. Figure 5 shows the difference in achieved throughput between CCP and the Linux kernel on a 10 Gbps link. As expected, with NIC offloads (segmentation and receive offloads) enabled, the CPU is not the bottleneck and both systems saturate the NIC. With segmentation offloads disabled on the sender, CCP achieves higher throughput than the kernel. We believe this is because CCP sends slightly larger bursts which leads to more efficient processing at the receiver via Generic Receive Offload (GRO) [18]. Indeed, with receive offloads disabled on the receiver as well, CCP and Linux achieve comparable performance. We believe that the remaining difference can be attributed to better interrupt coalescing at the receiver NIC. In future work, we plan to implement smooth congestion window transitions in the datapath to avoid packet bursts due to per-RTT congestion window updates.

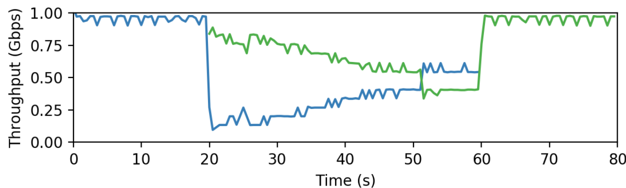
4 Related Work

How does the CCP API improve on existing interfaces to program congestion control on datapaths? Could datapaths support the CCP primitives?

The Linux kernel offers the 'pluggable TCP' API for congestion avoidance [17] that allows a kernel module to set a flow's congestion window or pacing rate [20, 21]. The API also provides access to per-packet delay, RTT-averaged rate samples, and specific TCP congestion events such as ECN marks and retransmission timeouts. In addition to Pluggable TCP, the CCP API exposes both transmitted and delivered rates, implements fold functions over per-packet measurements, and supports CCP control programs. CCP



(a) CCP NewReno



(b) Linux NewReno

Figure 4: Comparison of the reactivity of a CCP-based NewReno implementation and the Linux kernel implementation.

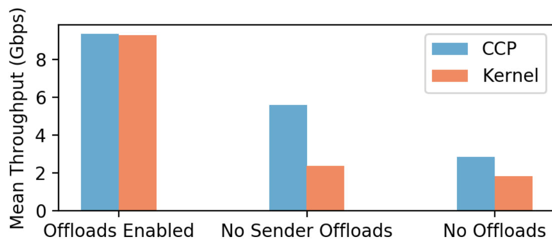


Figure 5: Comparison of achieved throughput with NIC offloads (TSO, GRO, and GSO) enabled and disabled, respectively. Each value is the average across four runs.

enables easy user-space programming of congestion control with support for floating point operations (e.g., cube root) and debugging capabilities lacking in the Linux kernel.

QUIC [32] is a user-space library which implements reliability and congestion control on top of UDP sockets. QUIC provides a protocol sender interface [46] which already supports the necessary basic CCP primitives in Table 2. We believe that supporting control programs on top of these capabilities will be straightforward. Unlike QUIC, congestion control algorithms built using CCP can run atop multiple datapaths. Further, with CCP, applications are not limited to using UDP.

Kernel bypass libraries such as DPDK [2] and netmap [44] allow user-space programs to send packets directly to the NIC from user-space. Applications using kernel bypass must implement their own custom end-to-end congestion control. The mTCP project [28] implements a single congestion control algorithm (NewReno) on top of DPDK. Extending the mTCP implementation to support the more general CCP primitives is a promising way to realize a CCP-compatible kernel-bypass datapath. mTCP already implements congestion windows and useful measurement statistics, and we believe that building packet pacing (e.g., analogous to Linux's FQ queueing discipline [29]) is feasible.

The congestion manager (CM [9]) proposes a datapath wherein an in-kernel agent performs congestion control operations on behalf of all flows. CCP supports this paradigm. However, unlike CM, CCP's congestion control algorithms reside off the critical path of data transmission (CM's algorithm resides in its kernel agent). Further, CCP provides a flexible API to implement new congestion control algorithms (CM's kernel agent implements a single algorithm).

IX [11] is a datapath building on the line of work on library operating systems [41]. However, IX's notion of a datapath includes congestion control. In CCP, the OS still performs hardware multiplexing, but congestion control occurs independently from the datapath.

5 Discussion

This paper made the case for moving congestion control off the datapath of transport protocols and into a separate, reusable, and flexible user-space agent. We conclude by outlining questions for further research related to CCP.

Can emerging hardware-accelerated datapaths support the CCP primitives? At first glance, it may appear counter-intuitive to increase the complexity of hardware datapaths by adding support for CCP primitives. However, RDMA NICs [42] and dedicated TCP engines [15] today already offload congestion control processing—such as maintaining congestion windows and accurate per-packet statistics—from the CPU to the NIC. CCP mainly requires making these capabilities externally programmable to avoid baking the congestion control algorithm into the NIC. Given the advent of programmable NICs [14, 39, 50], we anticipate that supporting measurement fold functions and control programs should be feasible. Rate-based congestion control algorithms can be implemented using packet pacing in hardware [25, 35, 43]. A key impediment to supporting CCP is maintaining per-flow congestion state on the NIC. To tackle this problem, it may be possible to extend the techniques used in receive-side scaling [6].

Could CCP work at low RTTs? Currently, CCP is designed to communicate its congestion control decisions to datapaths a few times per RTT. However, this communication is challenging at very low RTTs (e.g., 1–10 μs) due to IPC overheads. Prior works have proposed making congestion control decisions slower than per-packet, e.g., [10, 35]. However, could congestion control running much slower, e.g., once in a few RTTs, be effective in low RTT scenarios? Alternatively, to retain per-RTT control, could we synthesize the congestion controller into the datapath from the high-level CCP algorithm?

New congestion control schemes. CCP's API will allow the use of powerful user-space libraries (e.g., neural networks) in algorithms. CCP makes it possible to implement congestion control outside the sending hosts, for example to manage congestion for groups of flows that share common bottlenecks. Such offloads could allow efficient use of shared resources.

Is CCP safe to deploy? Misbehaving CCP algorithms should not crash the datapath or consume an inordinate amount of resources. Safe fallback mechanisms and policies within the CCP agent and datapaths could enable a robust platform to ease the deployment of congestion control algorithms.

Acknowledgements. We thank Anirudh Sivaraman, Aurojit Panda, Hariharan Rahul, Radhika Mittal, Ravi Netravali, Shoumik Palkar, and the HotNets reviewers for their useful comments and feedback. This work was supported in part by DARPA contract HR001117C0048 and NSF grant 1407470.

References

[1] CCP user-space. https://github.com/mit-nms/ccp.

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

[3] Intel Turbo Boost. https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[4] Linux kernel datapath for CCP. https://github.com/mit-nms/ccp-kernel.

[5] Linux kernel patch for CCP. https://github.com/ngsrinivas/linux-fork/commit/a84f0a6db796a84411547a7055d619d2792d871c.

[6] Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[7] Segmentation Offloads in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt.

[8] 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.

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

[10] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly-responsive Congestion Control Algorithms. SIGCOMM, 2001.

[11] 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.

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

[13] 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.

[14] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A Cloud-Scale Acceleration Architecture. In *MICRO*, 2016.

[15] Chelsio Communications. TCP Offload Engine (TOE). http://www.chelsio.com/nic/tcp-offload-engine/.

[16] Christian Benvenuti. *Understanding Linux network internals*. O'Reilly Media, 2009.

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

[18] J. Corbet. Generic receive offload. https://lwn.net/Articles/358910/, 2009.

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

[20] E. Dumazet. pkt_sched: fq: Fair Queue packet scheduler. https://lwn.net/Articles/564825/, 2013.

[21] E. Dumazet. TCP: Internal implementation for pacing. https://patchwork.ozlabs.org/patch/762899/, 2017.

[22] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *SIGCOMM*, 2013.

[23] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.

[24] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Operating System Review*, July 2008.

[25] Hanay, Y.S. and Dwaraki, A. and Wolf, T. High-performance implementation of in-network traffic pacing. In *HPSR*, 2011.

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

[27] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[28] E. Jeong, S. Wood, 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.

[29] Jonathan Corbet. TSO sizing and the FQ scheduler. https://lwn.net/Articles/564978/, 2013.

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

[31] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control Without Reliability. In *SIGCOMM*, 2006.

[32] 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.

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

[34] 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.

[35] 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.

[36] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively Cautious Congestion Control. In *NSDI*, 2014.

[37] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM*, 2016.

[38] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.

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

[40] J. L. Ny. Sampling and Sampled-Data Systems. http://www.professeurs.polymtl.ca/jerome.le-ny/teaching/NECS_Spring11/notes/3_sampling/3_sampling.pdf. [Online, Retrieved August 3, 2017].

[41] 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.

[42] J. Pinkerton. The case for RDMA. http://rdmaconsortium.org/home/The_Case_for_RDMA020531.pdf, 2002.

[43] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *NSDI*, 2014.

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

[45] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[46] The Chromium Authors. QUIC sender interface. https://chromium.googlesource.com/chromium/src/net/+/master/quic/core/congestion_control/send_algorithm_interface.h. [Online, Retrieved August 2, 2017].

[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] 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.

[50] N. Zilberman, Y. Audzevich, G. Kalogeridou, N. Manihatty-Bojan, J. Zhang, and A. Moore. NetFPGA: Rapid Prototyping of Networking Devices in Open Source. In *SIGCOMM*, 2015.