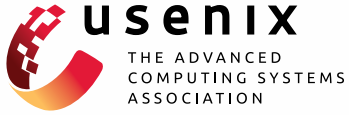


PCC: Re-architecting Congestion Control for Consistent High Performance

Mo Dong and Qingxi Li, *University of Illinois at Urbana-Champaign;* **Doron Zarchy,** *Hebrew University of Jerusalem;* **P. Brighten Godfrey,** *University of Illinois at Urbana-Champaign;* **Michael Schapira,** *Hebrew University of Jerusalem*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/dong

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

PCC: Re-architecting Congestion Control for Consistent High Performance

Mo Dong[*], Qingxi Li[*], Doron Zarchy[**], P. Brighten Godfrey[*], and Michael Schapira[**]

[*]University of Illinois at Urbana-Champaign
[**]Hebrew University of Jerusalem

Abstract

TCP and its variants have suffered from surprisingly poor performance for decades. We argue the TCP family has little hope of achieving consistent high performance due to a fundamental architectural deficiency: hardwiring packet-level events to control responses. We propose Performance-oriented Congestion Control (PCC), a new congestion control architecture in which each sender continuously observes the connection between its *actions* and *empirically experienced performance*, enabling it to consistently adopt actions that result in high performance. We prove that PCC converges to a stable and fair equilibrium. Across many real-world and challenging environments, PCC shows consistent and often 10× performance improvement, with better fairness and stability than TCP. PCC requires no router hardware support or new packet format.

1 Introduction

In the roughly 25 years since its deployment, TCP's congestion control architecture has been notorious for degraded performance. TCP performs poorly on lossy links, penalizes high-RTT flows, underutilizes high bandwidth-delay product (BDP) connections, cannot handle rapidly changing networks, can collapse under data center incast [24] and incurs very high latency with bufferbloat [28] in the network.

As severe performance problems have accumulated over time, protocol "patches" have addressed problems in specific network conditions such as high BDP links [31,52], satellite links [23,42], data center [18,55], wireless and lossy links [38,39], and more. However, the fact that there are so many TCP variants suggests that each is only a point solution: they yield better performance under specific network conditions, but break in others. Worse, we found through real-world experiments that in many cases these TCP variants' performance is *still far away from optimal even in the network conditions for which they are specially engineered.* Indeed, TCP's low performance has impacted industry to the extent that there is a lucrative market for special-purpose high performance data transfer services [1,2,11,13].

Thus, the core problem remains largely unsolved: *achieving consistently high performance over complex real-world network conditions*. We argue this is indeed a very difficult task within TCP's rate control architecture, which we refer to as **hardwired mapping**: certain predefined packet-level events are hardwired to certain predefined control responses. TCP reacts to events that can be as simple as "one packet loss" (TCP New Reno) or can involve multiple signals like "one packet loss and RTT increased by $x\%$" (TCP Illinois). Similarly, the control response might be "halve the rate" (New Reno) or a more complex action like "reduce the window size w to $f(\Delta RTT)w$" (Illinois). The defining feature is that the control action is a direct function of packet-level events.

A hardwired mapping has to make *assumptions* about the network. Take a textbook event-control pair: a packet loss halves the congestion window. TCP *assumes* that the loss indicates congestion in the network. When the assumption is violated, halving the window size can severely degrade performance (e.g. if loss is random, rate should stay the same or increase). It is fundamentally hard to formulate an "always optimal" hardwired mapping in a complex real-world network because the actual optimal response to an event like a loss (i.e. decrease rate or increase? by how much?) is sensitive to network conditions. And modern networks have an immense diversity of conditions: random loss and zero loss, shallow queues and bufferbloat, RTTs of competing flows varying by more than 1000×, dynamics due to mobile wireless or path changes, links from Kbps to Gbps, AQMs, software routers, rate shaping at gateways, virtualization layers and middleboxes like firewalls, packet inspectors and load balancers. These factors add complexity far beyond what can be summarized by the relatively simplistic assumptions embedded in a hardwired mapping. Most unfortunately, when its assumptions are violated, TCP still rigidly carries out the harmful control action.

We propose a new congestion control architecture: Performance-oriented Congestion Control (PCC). PCC's goal is to understand what rate control actions improve performance based on *live experimental evidence*, avoiding TCP's assumptions about the network. PCC sends at a rate r for a short period of time, and observes the results (e.g. SACKs indicating delivery, loss, and latency of each packet). It aggregates these packet-level events into a utility function that describes an objective like "high throughput and low loss rate". The result is a single numerical performance utility u. At this point, PCC has run a single "micro-experiment" that showed send-

ing at rate r produced utility u. To make a rate control decision, PCC runs multiple such micro-experiments: it tries sending at two different rates, and moves in the direction that empirically results in greater performance utility. This is effectively *A/B testing for rate control* and is the core of PCC's decisions. PCC runs these micro-experiments continuously (on every byte of data, not on occasional probes), driven by an online learning algorithm that tracks the empirically-optimal sending rate. Thus, rather than making assumptions about the potentially-complex network, PCC adopts the actions that *empirically* achieve consistent high performance.

PCC's rate control is selfish in nature, but surprisingly, using a widely applicable utility function, competing PCC senders provably converge to a fair equilibrium (with a single bottleneck link). Indeed, experiments show PCC achieves similar convergence time to TCP with significantly smaller rate variance. Moreover, the ability to express different objectives via choice of the utility function (e.g. throughput or latency) provides a flexibility beyond TCP's architecture.

With handling real-world complexity as a key goal, we experimentally evaluated a PCC implementation in large-scale and real-world networks. Without tweaking its control algorithm, PCC achieves consistent high performance and significantly beats *specially engineered* TCPs in various network environments: **(a.)** in the wild on the global commercial Internet (often more than **10×** the throughput of TCP CUBIC); **(b.)** inter-data center networks (**5.23×** vs. TCP Illinois); **(c.)** emulated satellite Internet links (**17×** vs TCP Hybla); **(d.)** unreliable lossy links (**10 − 37×** vs Illinois); **(e.)** unequal RTT of competing senders (an **architectural cure** to RTT unfairness); **(f.)** shallow buffered bottleneck links (up to **45×** higher performance, or **13×** less buffer to reach 90% throughput); **(g.)** rapidly changing networks (**14×** vs CUBIC, **5.6×** vs Illinois). PCC performs similar to ICTCP [55] in **(h.)** the incast scenario in data centers. Though it is a substantial shift in architecture, PCC can be deployed by only replacing the sender-side rate control of TCP. It can also deliver real data today with a user-space implementation at `speedier.net/pcc`.

2 PCC Architecture

2.1 The Key Idea

Suppose flow f is sending a stream of data at some rate and a packet is lost. How should f react? Should it slow the sending rate, or increase, and by how much? Or leave the rate unchanged? This is a difficult question to answer because real networks are complex: a single loss might be the result of *many* possible underlying network scenarios. To pick a few:

- f may be responsible for most of congestion. Then, it should decrease its rate.

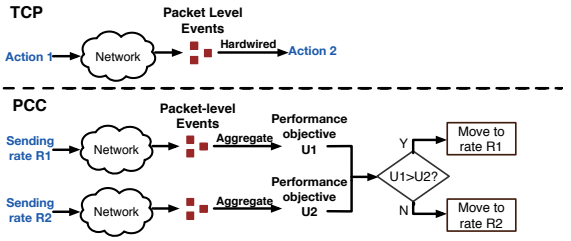


Figure 1: ***The decision-making structure of TCP and PCC.***

- f might traverse a shallow buffer on a high-BDP link, with the loss due to bad luck in statistical multiplexing rather than high link utilization. Then, backing off a little is sufficient.
- There may be a higher-rate competing flow. Then, f should maintain its rate and let the other back off.
- There may be random non-congestion loss somewhere along the path. Then, f should maintain or increase its rate.

Classically, TCP assumes a packet loss indicates non-negligible congestion, and that halving its rate will improve network conditions. However, this assumption is false and will degrade performance in three of the four scenarios above. Fundamentally, picking an optimal *predefined and hardwired* control response is hard because for the same packet-level events, a control response optimal under one network scenario can decimate performance in even a slightly different scenario. The approach taken by a large number of TCP variants is to use more sophisticated packet-level events and control actions. But this does not solve the fundamental problem, because the approach *still hardwires predetermined events to predetermined control responses*, thus inevitably embedding unreliable assumptions about the network. When the unreliable assumptions are violated by the complexity of the network, performance degrades severely. For example, TCP Illinois [38] uses both loss and delay to form an event-control mapping, but its throughput collapses with even a small amount of random loss, or when network conditions are dynamic (§4). More examples are in §5.

Most unfortunately, if some control actions are indeed harming performance, TCP can still blindly "jump off the cliff", because it does not notice the control action's actual effect on performance.

But that observation points toward a solution. Can we design a control algorithm that directly understands whether or not its actions actually improve performance?

Conceptually, no matter how complex the network is, if a sender can directly measure that rate r_1 results in better performance than rate r_2, it has some evidence that r_1 is better than sending at r_2 — at least for this one sender. This example illustrates the key design rationale behind **Performance-oriented Congestion Control** (PCC): PCC makes control decisions based on *em-*

pirical evidence pairing ***actions*** *with directly* ***observed performance*** *results*.

PCC's control action is its choice of sending rate. PCC divides time into continuous time periods, called *monitor intervals* (MIs), whose length is normally one to two RTTs. In each MI, PCC tests an action: it picks a sending rate, say r, and sends data at rate r through the interval. After about an RTT, the sender will see selective ACKs (SACK) from the receiver, just like TCP. However, it does not trigger any predefined control response. Instead, PCC aggregates these SACKs into meaningful performance metrics including throughput, loss rate and latency. These performance metrics are combined to a numerical utility value, say u, via a *utility function*. The utility function can be customized for different data transmission objectives, but for now the reader can assume the objective of "high throughput and low loss rate", such as $u = T - L$ (where T = throughput and L = loss rate) which will capture the main insights of PCC. The end result is that PCC knows when it sent at rate r, it got utility of u.

The preceding describes a single "micro-experiment" through which PCC associates a specific *action* with an observed *resulting utility*. PCC runs these micro-experiments continuously, comparing the utility of different sending rates so it can track the optimal action over time. More specifically, PCC runs an online learning algorithm similar to gradient ascent. When starting at rate r, it tests rate $(1+\varepsilon)r$ and rate $(1-\varepsilon)r$, and moves in the direction (higher or lower rate) that empirically yields higher utility. It continues in this direction as long as utility continues increasing. If utility falls, it returns to a decision-making state where it again tests both higher and lower rates to find which produces higher utility.

Note that PCC does not send occasional probes or use throwaway data for measurements. It observes the results of its actual control decisions on the application's real data and does not pause sending to wait for results.

We now return to the example of the beginning of this section. Suppose PCC is testing rate 100 Mbps in a particular interval, and will test 105 Mbps in the following interval. If it encounters a packet loss in the first interval, will PCC increase or decrease? In fact, there is no specific event in a single interval that will always cause PCC to increase or decrease its rate. Instead, PCC will calculate the utility value for each of these two intervals, and move in the direction of higher utility. For example:

- If the network is congested as a result of this flow, then it is likely that sending at 100 Mbps will have similar throughput and lower loss rate, resulting in higher utility. PCC will decrease its rate.
- If the network is experiencing random loss, PCC is likely to find that the period with rate 105 Mbps has similar loss rate and slightly higher throughput, resulting in higher utility. PCC will therefore increase its rate despite the packet loss.

Throughout this process, PCC makes no assumptions about the underlying network conditions, instead observing which actions empirically produce higher utility and therefore achieving consistent high performance.

Decisions with noisy measurements. PCC's experiments on the live network will tend to move its rate in the direction that improves utility. But it may also make some incorrect decisions. In the example above, if the loss is random non-congestion loss, it may randomly occur that loss is substantially higher when PCC tests rate 105 Mbps, causing it to pick the lower rate. Alternately, if the loss is primarily due to congestion from this sender, unpredictable external events (perhaps another sender arriving with a large initial rate while PCC is testing rate 100 Mbps) might cause a particular 105 Mbps microexperiment to have higher throughput and lower loss rate. More generally, the network might be changing over time for reasons unrelated to the sender's action. This adds noise to the decision process: PCC will on average move in the right direction, but may make some unlucky errors.

We improve PCC's decisions with **multiple randomized controlled trials (RCTs)**. Rather than running two tests (one each at 100 and 105 Mbps), we conduct four in randomized order—e.g. perhaps $(100, 105, 105, 100)$. PCC only picks a particular rate as the winner if utility is higher in *both* trials with that rate. This produces increased confidence in a causal connection between PCC's action and the observed utility. If results are inconclusive, so each rate "wins" in one test, then PCC maintains its current rate, and we may have reached a local optimum (details follow later).

As we will see, without RCTs, PCC already offers a dramatic improvement in performance and stability compared with TCP, but RCTs further reduce rate variance by up to 35%. Although it might seem that RCTs will double convergence time, this is not the case because they help PCC make *better* decisions; overall, RCTs improve the stability/convergence-speed tradeoff space.

Many issues remain. We next delve into fairness, convergence, and choice of utility function; deployment; and flesh out the mechanism sketched above.

2.2 Fairness and Convergence

Each PCC sender optimizes its utility function value based only on locally observed performance metrics. However, this local selfishness does not imply loss of global stability, convergence and fairness. We next show that when selfish senders use a particular "safe" utility function and a simple control algorithm, they provably converge to fair rate equilibrium.

We assume n PCC senders $1, \ldots, n$ send traffic across a bottleneck link of capacity $C > 0$. Each sender i chooses its sending rate x_i to optimize its utility function

u_i. We choose a utility function expressing the common application-level goal of "high throughput and low loss":

$$u_i(x_i) = T_i \cdot Sigmoid_\alpha(L_i - 0.05) - x_i \cdot L_i$$

where x_i is sender i's sending rate, L_i is the observed data loss rate, $T_i = x_i(1-L_i)$ is sender i's throughput, and $Sigmoid_\alpha(y) = \frac{1}{1+e^{\alpha y}}$ for some $\alpha > 0$ to be chosen later.

The above utility function is derived from a simpler starting point: $u_i(x_i) = T_i - x_i \cdot L_i$, i.e., i's throughput minus the production of its loss rate and sending rate. However, this utility function will make loss rate at equilibrium point approach 50% when the number of competing senders increases. Therefore, we include the sigmoid function as a "cut-off". When α is "big enough", $Sigmoid_\alpha(L_i - 0.05)$ will rapidly get closer to 0 as soon as L_i exceeds 0.05, leading to a negative utility for the sender. Thus, we are setting a barrier that caps the overall loss rate at about 5% in the worst case.

Theorem 1 *When $\alpha \geq \max\{2.2(n-1), 100\}$, there exists a unique stable state of sending rates $x_1^*, \ldots, x_n^*$ and, moreover, this state is fair, i.e., $x_1^* = x_2^* = \ldots = x_n^*$.*

To prove Theorem 1, we first prove that $\Sigma_j x_j$ will always be restricted to the region of $(C, \frac{20C}{19})$. Under this condition, our setting can be formulated as a concave game [46]. This enables us to use properties of such games to conclude that a unique rate equilibrium exists and is fair, i.e. $x_1^* = x_2^* = \ldots = x_n^*$. (Full proof: [6])

Next, we show that a simple control algorithm can converge to that equilibrium. At each time step t, each sender j updates its sending rate according to $x_j^{t+1} = x_j^t(1+\varepsilon)$ if j's utility would improve if it unilaterally made this change, and $x_j^{t+1} = x_j^t(1-\varepsilon)$ otherwise. Here $\varepsilon > 0$ is a small number ($\varepsilon = 0.01$, in the experiment). In this model, senders concurrently update their rates, but each sender decides based on a utility comparison as if it were the only one changing. This model does not explicitly consider measurement delay, but we believe it is a reasonable simplification (and experimental evidence bears out the conclusions). We also conjecture the model can be relaxed to allow for asynchrony. We discuss in §3 our implementation with practical optimizations of the control algorithm.

Theorem 2 *If all senders follow the above control algorithm, for every sender j, x_j converges to the domain $(\hat{x}(1-\varepsilon)^2, \hat{x}(1+\varepsilon)^2)$, where $\hat{x}$ denotes the sending rate in the unique stable state.*

It might seem surprising that PCC uses *multiplicative* rate increase and decrease, yet achieves convergence and fairness. If TCP used MIMD, in an idealized network senders would often get the same back-off signal at the same time, and so would take the *same multiplicative decisions in lockstep*, with the ratio of their rates never changing. In PCC, senders make *different* decisions. Consider a 100 Mbps link with sender A at rate 90 Mbps and B at 10 Mbps. When A experiments with slightly higher and lower rates $(1 \pm \varepsilon)90$ Mbps, it will find that it should decrease its rate to get higher utility because when it sends at higher than equilibrium rate, the loss rate dominates the utility function. However, when B experiments with $(1 \pm \varepsilon)10$ Mbps it finds that loss rate increase is negligible compared with its improved throughput. This occurs precisely because B is responsible for little of the congestion. In fact, this reasoning (and the formal proof of the game dynamics) is *independent of the step size* that the flows use in their experiments: PCC senders move towards the convergence point, even if they use a heterogeneous mix of AIMD, AIAD, MIMD, MIAD or other step functions. Convergence behavior does depend on the choice of utility function, however.

2.3 Utility Function: Source of Flexibility

PCC carries a level of flexibility beyond TCP's architecture: the same learning control algorithm can cater to different applications' heterogeneous objectives (e.g. latency vs. throughput) by using different utility functions. For example, under TCP's architecture, latency based protocols [38, 52] usually contain different hardwired mapping algorithms than loss-based protocols [31]. Therefore, without changing the control algorithm, as Sivaraman et al. [47] recently observed, TCP has to rely on different in-network active queue management (AQM) mechanisms to cater to different applications' objectives because *even with fair queueing*, TCP is blind to applications' objectives. However, by literally changing one line of code that describes the utility function, PCC can flip from "loss-based" (§2.2) to "latency-based" (§4.4) and thus caters to different applications' objectives without the complexity and cost of programmable AQMs [47]. That said, alternate utility functions are a largely unexplored area of PCC; in this work, we evaluate alternate utility functions only in the context of fair queueing (§4.4).

2.4 Deployment

Despite being a significant architectural shift, PCC needs only isolated changes. **No router support:** unlike ECN, XCP [35], and RCP [25], there are no new packet fields to be standardized and processed by routers. **No new protocol:** The packet format and semantics can simply remain as in TCP (SACK, hand-shaking and etc.). **No receiver change:** TCP SACK is enough feedback. What PCC does change is the control algorithm within the sender.

The remaining concern is how PCC safely replaces and interacts with TCP. We observe that there are many scenarios where critical applications suffer severely from TCP's poor performance and PCC can be safely deployed by fully replacing or being isolated from TCP.

First, **when a network resource is owned by a single entity** or can be reserved for it, the owner can replace TCP entirely with PCC. For example, some Content Delivery Network (CDN) providers use dedicated network infrastructure to move large amounts of data across continents [9, 10], and scientific institutes can reserve bandwidth for exchanging huge scientific data globally [26]. Second, PCC can be used in challenging network conditions **where per-user or per-tenant resource isolation is enforced** by the network. Satellite Internet providers are known to use per-user bandwidth isolation to allocate the valuable bandwidth resource [15]. For data centers with per-tenant resource isolation [30, 43, 44], an individual tenant can use PCC safely within its virtual network to address problems such as incast and improve data transfer performance between data centers.

The above applications, where PCC can fully replace or be isolated from TCP, are a significant opportunity for PCC. But in fact, **PCC does not depend on any kind of resource isolation to work.** In the public Internet, the key issue is TCP friendliness. Using PCC with the utility function described in §2.2 is not TCP friendly. However, we also study the following utility function which incorporates latency: $u_i(x) = (T_i \cdot Sigmoid_\alpha(L_i - 0.05) \cdot Sigmoid_\beta(\frac{RTT_{n-1}}{RTT_n} - 1) - x_i \cdot L_i)/RTT_n$ where RTT_{n-1} and RTT_n are the average RTT of the previous and current MI, respectively. In §4.3.1 we show that with this utility function, PCC successfully achieves TCP friendliness in various network conditions. Indeed, it is even possible for PCC to be TCP friendly while achieving much higher performance in challenging scenarios (by taking advantage of the capacity TCP's poor control algorithm leaves unused). Overall, this is a promising direction but we only take the first steps in this paper.

It is still possible that individual users will, due to its significantly improved performance, decide to deploy PCC in the public Internet with the default utility function. It turns out that the default utility function's unfriendliness to TCP is comparable to the common practice of opening parallel TCP connections used by web browsers today [3], so it is unlikely to make the ecosystem *dramatically* worse for TCP; see §4.3.2.

3 Prototype Design

We implemented a prototype of PCC in user space by adapting the UDP-based TCP skeleton in the UDT [16] package. Fig. 2 depicts our prototype's components.

3.1 Performance Monitoring

As described in §2.1 and shown in Fig. 3, the timeline is sliced into chunks of duration of T_m called the *Monitor Interval* (MI). When the Sending Module sends packets (new or retransmission) at a certain sending rate instructed by the Performance-oriented Control Module, the Monitor Module will remember what packets are sent

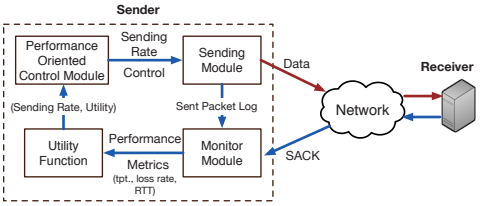


Figure 2: ***PCC prototype design***

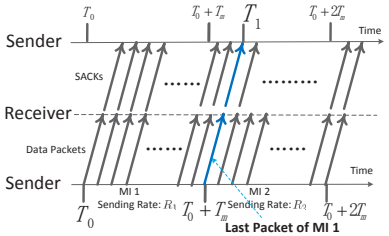


Figure 3: ***Performance monitoring process***

out during each MI. As the SACK comes back from receiver, the Monitor will know what happened (received? lost? RTT?) to each packet sent out during an MI. Taking the example of Fig. 3, the Monitor knows what packets were sent during MI1, spanning T_0 to $T_0 + T_m$. At time T_1, approximately one RTT after $T_0 + T_m$, it has received the SACKs for all packets sent out in MI1. The Monitor aggregates these individual SACKs into meaningful performance metrics including throughput, loss rate and average RTT. The performance metrics are then combined by a utility function; unless otherwise stated, we use the utility function of §2.2. The result of this is that we associate a control action of each MI (sending rate) with its performance result (utility). This pair forms a "micro-experiment" and is used by the performance oriented control module.

To ensure there are enough packets in one monitor interval, we set T_m to the maximum of (a) the time to send 10 data packets and (b) a uniform-random time in the range $[1.7, 2.2]$ RTT. Again, we want to highlight that PCC *does not* pause sending packets to wait for performance results, and it *does not* decide on a rate and send for a long time; packet transfer and measurement-control cycles occur continuously.

Note that the measurement results of one MI can arrive after the next MI has begun, and the control module can decide to change sending rate after processing this result. As an optimization, PCC will immediately change the rate and "re-align" the current MI's starting time with the time of rate change without waiting for the next MI.

3.2 Control Algorithm

We designed a practical control algorithm with the gist of the simple control algorithm described in §2.2.

Starting State: PCC starts at rate $2 \cdot MSS/RTT$ (i.e., $3KB/RTT$) and doubles its rate at each consecutive monitor interval (MI), like TCP. Unlike TCP, PCC does not exit this starting phase because of a packet loss. Instead,

it monitors the utility result of each rate doubling action. Only when the utility decreases, PCC exits the starting state, returns to the previous rate which had higher utility (i.e., half of the rate), and enters the *Decision Making State*. PCC could use other more aggressive startup strategies, but such improvements could be applied to TCP as well.

Decision Making State: Assume PCC is currently at rate r. To decide which direction and amount to change its rate, PCC conducts **multiple randomized controlled trials (RCTs)**. PCC takes four consecutive MIs and divides them into two pairs (2 MIs each). For each pair, PCC attempts a slightly higher rate $r(1+\varepsilon)$ and slightly lower rate $r(1-\varepsilon)$, each for one MI, in random order. After the four consecutive trials, PCC changes the rate back to r and keeps aggregating SACKs until the Monitor generates the utility value for these four trials. For each pair $i \in 1,2$, PCC gets two utility measurements U_i^+, U_i^- corresponding to $r(1+\varepsilon), r(1-\varepsilon)$ respectively. If the higher rate consistently has higher utility ($U_i^+ > U_i^- \ \forall i \in \{1,2\}$), then PCC adjusts its sending rate to $r_{new} = r(1+\varepsilon)$; and if the lower rate consistently has higher utility then PCC picks $r_{new} = r(1-\varepsilon)$. However, if the results are inconclusive, e.g. $U_1^+ > U_1^-$ but $U_2^+ < U_2^-$, PCC stays at its current rate r and re-enters the Decision Making State with larger experiment granularity, $\varepsilon = \varepsilon + \varepsilon_{min}$. The granularity starts from ε_{min} when it enters the Decision Making State for the first time and will increase up to ε_{max} if the process continues to be inconclusive. This increase of granularity helps PCC avoid getting stuck due to noise. Unless otherwise stated, we use $\varepsilon_{min} = 0.01$ and $\varepsilon_{max} = 0.05$.

Rate Adjusting State: Assume the new rate after Decision Making is r_0 and $dir = \pm 1$ is the chosen moving direction. In each MI, PCC adjusts its rate in that direction faster and faster, setting the new rate r_n as: $r_n = r_{n-1} \cdot (1 + n \cdot \varepsilon_{min} \cdot dir)$. However, if utility falls, i.e. $U(r_n) < U(r_{n-1})$, PCC reverts its rate to r_{n-1} and moves back to the *Decision Making State*.

4 Evaluation

We demonstrate PCC's architectural advantages over the TCP family through diversified, large-scale and real-world experiments: §4.1: PCC achieves its design goal of **consistent high performance**. §4.2: PCC can actually achieve much **better fairness and convergence/stability tradeoff** than TCP. §4.3: PCC is **practically deployable** in terms of flow completion time for short flows and TCP friendliness. §4.4: PCC has a huge potential to flexibly **optimize for applications' heterogenous objectives** with fair queuing in the network rather than more complicated AQMs [47].

Transmission Pair	RTT	PCC	SABUL	CUBIC	Illinois
GPO → NYSERNet	12	818	563	129	326
GPO → Missouri	47	624	531	80.7	90.1
GPO → Illinois	35	766	664	84.5	102
NYSERNet → Missouri	47	816	662	108	109
Wisconsin → Illinois	9	801	700	547	562
GPO → Wisc.	38	783	487	79.3	120
NYSERNet → Wisc.	38	791	673	134	134
Missouri → Wisc.	21	807	698	259	262
NYSERNet → Illinois	36	808	674	141	141

Table 1: PCC significantly outperforms TCP in inter-data center environments. RTT is in msec; throughput in Mbps.

4.1 Consistent High Performance

We evaluate PCC's performance under 8 real-world challenging network scenarios with with *no algorithm tweaking for different scenarios. Unless otherwise stated, all experiments using the same default utility function of §2.2*. In the first 7 scenarios, PCC significantly outperforms specially engineered TCP variants.

4.1.1 Inter-Data Center Environment

Here we evaluate PCC's performance in scenarios like inter-data center data transfer [5] and dedicated CDN backbones [9] where **network resources can be isolated or reserved for a single entity**.

The GENI testbed [7], which has reservable bare-metal servers across the U.S. and reservable bandwidth [8] over the Internet2 backbone, provides us a representative evaluation environment. We choose 9 pairs of GENI sites and reserved **800Mbps** end-to-end dedicated bandwidth between each pair. We compare PCC, SABUL [29], TCP CUBIC [31] and TCP Illinois [38] over 100-second runs.

As shown in Table 1, PCC significantly outperforms TCP Illinois, by 5.2× on average and up to 7.5×. It is surprising that even in this very clean network, specially optimized TCPs still perform far from optimal. We believe some part of the gain is because the bandwidth-reserving rate limiter has a small buffer and TCP will overflow it, unnecessarily decreasing rate and also introducing latency jitter that confuses TCP Illinois. (TCP pacing will not resolve this problem; §4.1.5.) On the other hand, PCC continuously tracks the optimal sending rate by continuously measuring performance.

4.1.2 Satellite Links

Satellite Internet is widely used for critical missions such as emergency and military communication and Internet access for rural areas. Because TCP suffers from severely degraded performance on satellite links that have excessive latency (600ms to 1500ms RTT [14]) and relatively high random loss rate [42], special modifications of TCP (Hybla [23], Illinois) were proposed and special infrastructure has even been built [32, 50].

We test PCC against TCP Hybla (widely used in real-world satellite communication), Illinois and CUBIC under emulated satellite links on Emulab parameterized

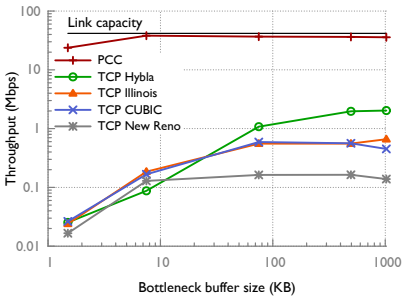


Figure 4: ***PCC outperforms special TCP modifications on emulated satellite links***

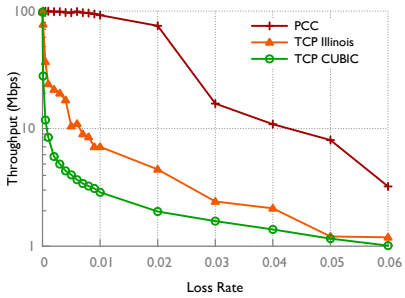


Figure 5: ***PCC is highly resilient to random loss compared to specially-engineered TCPs***

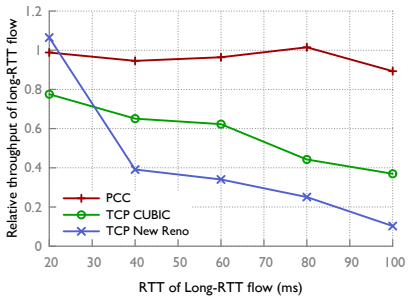


Figure 6: ***PCC achieves better RTT fairness than specially engineered TCPs***

with the real-world measurements of the WINDs satellite Internet system [42]. The satellite link has **800ms RTT, 42Mbps capacity and 0.74% random loss**. As shown in Fig. 4, we vary the bottleneck buffer from 1.5KB to 1MB and compare PCC's average throughput against different TCP variants with 100 second trials. PCC achieves 90% of optimal throughput even with only a 7.5KB buffer (5 packets) at the bottleneck. However, even with 1MB buffer, the widely used TCP Hybla can only achieve 2.03Mbps which is 17× worse than PCC. TCP Illinois, which is designed for high random loss tolerance, performs 54× worse than PCC with 1MB buffer.

4.1.3 Unreliable Lossy Links

To further quantify the effect of random loss, we set up a link on Emulab with **100Mbps bandwidth, 30ms RTT and varying loss rate**. As shown in Fig. 5, PCC can reach $>95\%$ of achievable throughput capacity until loss rate reaches 1% and shows relatively graceful performance degradation from 95% to 74% of capacity as loss rate increases to 2%. However, TCP's performance collapses very quickly: CUBIC's performance collapses to 10× smaller than PCC with only 0.1% loss rate and 37× smaller than PCC with 2% random loss. TCP Illinois shows better resilience than CUBIC but throughput still degrades severely to less than 10% of PCC's throughput with only 0.7% loss rate and 16× smaller with 2% random loss. Again, PCC can endure random loss because it looks at real utility: unless link capacity is reached, a higher rate will always result in similar loss rate and higher throughput, which translates to higher utility.

PCC's performance does decrease to 3% of the optimal achievable throughput when loss rate increases to 6% because we are using the "safe" utility function of §2.2 that caps the loss rate to 5%[1].

4.1.4 Mitigating RTT Unfairness

For unmodified TCP, short-RTT flows dominate long-RTT flows on throughput. Subsequent modifications of TCP such as CUBIC or Hybla try to mitigate this problem by making the expansion of the congestion window independent of RTT. However, the modifications cause new problems like parameter tuning (Hybla) and severely affect stability on high RTT links (CUBIC) [31]. Because PCC's convergence is based on real performance not the control cycle length, it acts as an architectural cure for the RTT unfairness problem. To demonstrate that, on Emulab we set **one short-RTT (10ms) and one long-RTT (varying from 20ms to 100ms) network path sharing the same bottleneck link of 100Mbit/s bandwidth** and buffer equal to the BDP of the short-RTT flow. We run the long-RTT flow first for 5s, letting it grab the bandwidth, and then let the short-RTT flow join to compete with the long-RTT flow for 500s and calculate the ratio of the two flows' throughput. As shown in Fig. 6, PCC achieves much better RTT fairness than New Reno and even CUBIC cannot perform as well as PCC.

4.1.5 Small Buffers on the Bottleneck Link

TCP cannot distinguish between loss due to congestion and loss simply due to buffer overflow. In face of high BDP links, a shallow-buffered router will keep chopping TCP's window in half and the recovery process is very slow. On the other hand, the practice of over-buffering networks, in the fear that an under-buffered router will drop packets or leave the network severely under-utilized, results in bufferbloat [28], increasing latency. This conflict makes choosing the right buffer size for routers a challenging multi-dimensional optimization problem [27,45,51] for network operators to balance between throughput, latency, cost of buffer memory, degree of multiplexing, etc.

Choosing the right buffer size would be much less difficult if the transport protocol could efficiently utilize a network with very shallow buffers. Therefore, we test how PCC performs with a tiny buffer and compare with TCP CUBIC, which is known to mitigate this problem. Moreover, to address the concern that the performance gain of PCC is merely due to PCC's use of packet pacing, we also test an implementation of TCP New Reno with pacing rate of $(congestionwindow)/(RTT)$. We set

[1] Throughput does not decrease to 0% because the sigmoid function is not a clean cut-off.

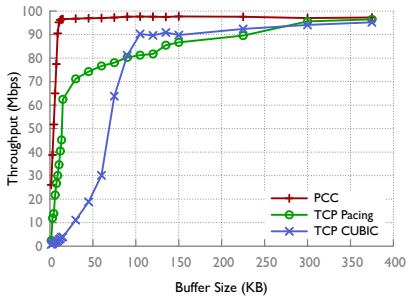


Figure 7: ***PCC efficiently utilizes shallow-buffered networks***

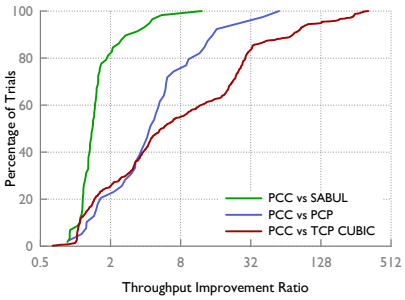


Figure 8: ***Across the public Internet, PCC has*** $\geq 10\times$ ***the performance of TCP CUBIC on*** 41% ***of tested pairs***

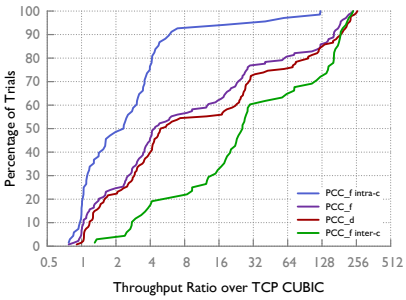


Figure 9: ***PCC's performance gain is not merely due to TCP unfriendliness***

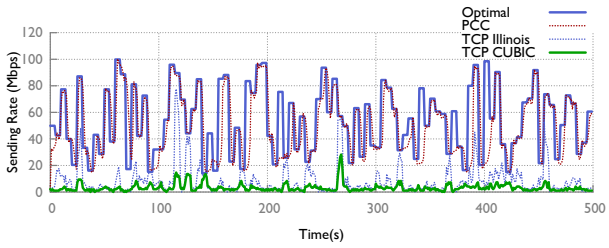


Figure 10: ***PCC can always track optimal sending rate even with drastically changing network conditions***

up on Emulab a network path with **30ms RTT, 100Mbps bottleneck bandwidth and vary the network buffer size from 1.5KB (one packet) to 375KB** ($1 \times BDP$) and compare the protocols' average throughput over 100s.

As shown in 7, PCC only requires $6 \cdot MSS$ (9 KB) buffer to reach 90% capacity. With the same buffer, CUBIC can only reach 2% capacity and even TCP with packet pacing can only reach 30%. CUBIC requires $13\times$ more buffer than PCC to reach 90% throughput and takes $36\times$ more buffer to close the 10% gap. Even with pacing, TCP still requires $25\times$ more buffer than PCC to reach 90% throughput. It is also interesting to notice that with just a *single-packet* buffer, PCC's throughput can reach 25% of capacity, $35\times$ higher throughput than TCP CUBIC. The reason is that PCC constantly monitors the real achieved performance and steadily tracks its rate at the bottleneck rate without swinging up and down like TCP. That means with PCC, network operators can use shallow buffered routers to **get low latency without harming throughput.**

4.1.6 Rapidly Changing Networks

The above scenarios are static environments. Next, we study a network that changes rapidly during the test. We set up on Emulab a network path where **available bandwidth, loss rate and RTT are all changing every** 5 **seconds.** Each parameter is chosen independently from a uniform random distribution with bandwidth ranging from 10Mbps to 100Mbps, latency from 10ms to 100ms and loss rate from 0% to 1%.

Figure 10 shows available bandwidth (optimal sending rate), and the sending rate of PCC, CUBIC and Illinois. Note that we show the PCC control algorithm's chosen sending rate (not its throughput) to get insight into how PCC handles network dynamics. Even with all network parameters rapidly changing, PCC tracks the available bandwidth very well, unlike the TCPs. Over the course of the experiment (500s), PCC's throughput averages 44.9Mbps, achieving 83% of the optimal, while TCP CUBIC and TCP Illinois are $14\times$ and $5.6\times$ worse than PCC respectively.

4.1.7 Big Data Transfer in the Wild

Due to its complexity, the commercial Internet is an attractive place to test whether PCC can achieve consistently high performance. We deploy PCC's receiver on 85 globally distributed PlanetLab [17] nodes and senders on 6 locations: five GENI [7] sites and our local server, and ran experiments over a 2-week period in December 2013. These 510 sending-receiving pairs render a very diverse testing environment with BDP from 14.3 KB to 18 MB.

We first test PCC against TCP CUBIC, the Linux kernel default since 2.6.19; and also SABUL [16], a special modification of TCP for high BDP links. For each sender-receiver pair, we run TCP iperf between them for 100 seconds, wait for 500 seconds and then run PCC for 100 seconds to compare their average throughput. PCC improves throughput by $5.52\times$ at the median (Fig. 8). On 41% of sender-receiver pairs, PCC's improvement is more than $10\times$. This is a conservative result because 4 GENI sites have 100Mbps bandwidth limits on their Internet uplinks.

We also tested two other non-TCP transport protocols on smaller scale experiments: the public releases of PCP [12,20] (43 sending receiving pairs) and SABUL (85 sending receiving pairs). PCP uses packet-trains [33] to probe available bandwidth. However, as discussed more in §5, this bandwidth probing is different from PCC's control based on empirically observed action-utility pairs, and contains unreliable assumptions that can yield very inaccurate sample results. SABUL, widely used for scientific data transfer, packs a full set of boost-

ing techniques: packet pacing, latency monitoring, random loss tolerance, etc. However, SABUL still mechanically hardwires control action to packet-level events. Fig. 8 shows PCC outperforms PCP[2] by 4.58× at the median and 15.03× at the 90th percentile, and outperforms SABUL by 1.41× at the median and 3.39× at the 90th percentile. SABUL shows an unstable control loop: it aggressively overshoots the network and then deeply falls back. On the other hand, PCC stably tracks the optimal rate. As a result, SABUL suffers from 11% loss on average compared with PCC's 3% loss.

Is PCC's performance gain merely due to TCP unfriendliness of the default utility function? In fact, PCC's high performance gain should not be surprising given our results in previous experiments, none of which involved PCC and TCP sharing bandwidth. Nevertheless, we ran another experiment, this time with PCC using the more TCP-friendly utility function described in §2.4 with $\beta = 10$ (its TCP friendliness is evaluated in § 4.3.1), from a server at UIUC[3] to 134 PlanetLab nodes in February 2015. Fig. 9 compares the results with the default utility function (PCC_d) and the friendlier utility function (PCC_f). PCC_f still shows a median of 4.38× gain over TCP while PCC_d shows 5.19×. For 50 pairs, PCC_d yields smaller than 3% higher throughput than PCC_f and for the remaining 84 pairs, the median inflation is only 14%. The use of the PCC_f utility function does not fully rule out the possibility of TCP unfriendliness, because our evaluation of its TCP friendliness (§4.3.1) does not cover all possible network scenarios involved in this experiment. However, it is highly suggestive that the performance gain is not merely due to TCP unfriendliness.

Instead, the results indicate that PCC's advantage comes from its ability to deal with complex network conditions. In particular, geolocation revealed that the large-gain results often involved cross-continent links. On cross-continent links (68 pairs), PCC_f yielded a median gain of 25× compared with 2.33× on intra-continent links (69 pairs). We believe TCP's problem with cross-continent links is not an end-host parameter tuning problem (e.g. sending/receiving buffer size), because there are paths with similar RTT where TCP can still achieve high throughput with identical OS and configuration.

4.1.8 Incast

Moving from wide-area networks to the data center, we now investigate TCP incast [24], which occurs in high bandwidth and low latency networks when multiple senders send data to one receiver concurrently, causing throughput collapse. To solve the TCP incast problem, many protocols have been proposed, including ICTCP [55] and DCTCP [18]. Here, we demonstrate PCC can achieve high performance under incast without special-purpose algorithms. We deployed PCC on Emulab [53] with **33 senders and 1 receiver**.

Fig. 11 shows the goodput of PCC and TCP across various flow sizes and numbers of senders. Each point is the average of 15 trials. When incast congestion begins to happen with roughly ≥ 10 senders, PCC achieves roughly 60-80% of the maximum possible goodput, or 7-8× that of TCP. Note that ICTCP [55] also achieved roughly 60-80% goodput in a similar environment. Also, DCTCP's goodput degraded with increasing number of senders [18], while PCC's is very stable.

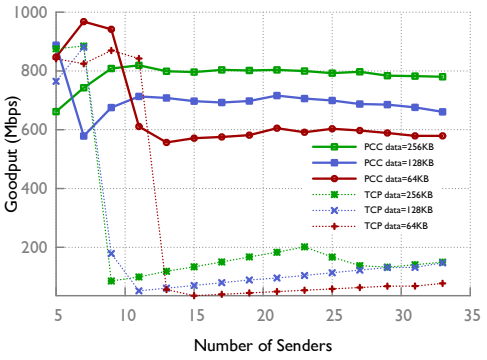
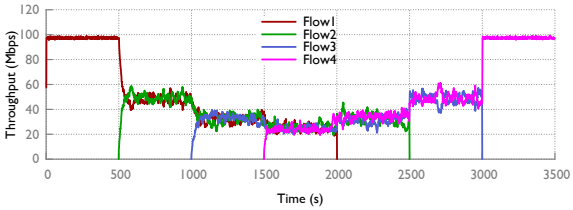
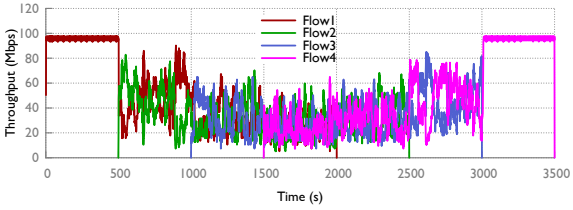


Figure 11: ***PCC largely mitigates TCP incast in a data center environment***



(a) PCC maintains a stable rate with competing senders



(b) TCP CUBIC shows high rate variance and unfairness at short time scales

Figure 12: ***PCC's dynamics are much more stable than TCP CUBIC with senders competing on a FIFO queue***

4.2 Dynamic Behavior of Competing Flows

We proved in §2.2 that with our "safe" utility function, competing PCC flows converge to a fair equilibrium from any initial state. In this section, we experimentally show

[2] $initial - rate = 1Mbps$, $poll - interval = 100\mu s$. PCP in many cases abnormally slows down (e.g. 1 packet per 100ms). We have not determined whether this is an implementation bug in PCP or a more fundamental deficiency. To be conservative, we excluded all such results from the comparison.

[3] The OS was Fedora 21 with kernel version 3.17.4-301.

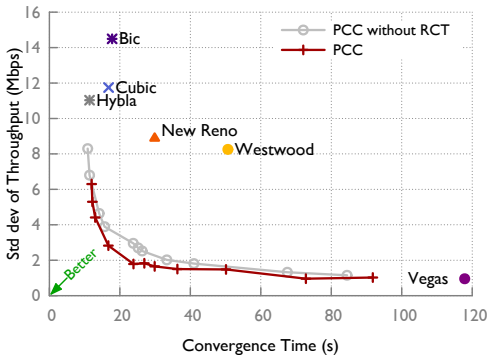


Figure 16: ***PCC has better reactiveness-stability tradeoff than TCP, particularly with its RCT mechanism***

that **PCC is much more stable, more fair and achieves a better tradeoff between stability and reactiveness than TCP.** PCC's stability can immediately translate to benefits for applications such as video streaming where stable rate in presence of congestion is desired [34].

4.2.1 PCC is More Fair and Stable Than TCP

To compare PCC and TCP's convergence process in action, we set up a dumbbell topology on Emulab with **four senders and four receivers sharing a bottleneck link with** 30**ms RTT,** 100**Mbps bandwidth**. Bottleneck router buffer size is set to the BDP to allow CUBIC to reach full throughput.

The data transmission of the four pairs initiates sequentially with a 500s interval and each pair transmits continuously for 2000s. Fig. 12 shows the rate convergence process for PCC and CUBIC respectively with 1s granularity. It is visually obvious that PCC flows converge much more stably than TCP, which has surprisingly high rate variance. Quantitatively, we compare PCC's and TCP's fairness ratio (Jain's index) at different time scales (Fig. 13). Selfishly competing PCC flows achieve better fairness than TCP at all time scales.

4.2.2 PCC has better Stability-Reactiveness tradeoff than TCP

Intuitively, PCC's control cycle is "longer" than TCP due to performance monitoring. Is PCC's significantly better stability and fairness achieved by severely sacrificing convergence time?

We set up two sending-receiving pairs sharing a bottleneck link of 100Mbps and 30ms RTT. We conduct the experiment by letting the first flow, flow A, come in the network for 20s and then let the second flow, flow B, begin. We define the convergence time in a "forward-looking" way: we say flow B's *convergence time* is the smallest t for which throughput in each second from t to $t+5s$ is within $\pm 25\%$ of the ideal equal rate. We measure stability by measuring the standard deviation of throughput of flow B for $60s$ after convergence time. All results are averaged over 15 trials. PCC can achieve various points in the stability-reactiveness trade-off space by adjusting its parameters: higher step size ε_{min} and lower monitor interval T_m result in faster convergence but higher throughput variance. In Fig. 16, we plot a trade-off curve for PCC by choosing a range of different settings of these parameters.[4] There is a clear convergence speed and stability trade-off: higher ε_{min} and lower T_m result in faster convergence and higher variance and vice versa. We also show six TCP variants as individual points in the trade-off space. The TCPs either have very long convergence time or high variance. On the other hand, PCC achieves a much better trade-off. For example, PCC with $T_m = 1.0 \cdot RTT$ and $\varepsilon_{min} = 0.02$ achieves the same convergence time and $4.2\times$ smaller rate variance than CUBIC.

Fig. 16 also shows the **benefit of the RCT mechanism** described in §3.2. While the improvement might look small, it actually helps most in the "sweet spot" where convergence time and rate variance are both small, and where improvements are most difficult and most valuable. Intuitively, with a long monitor interval, PCC gains enough information to make a low-noise decision even in a single interval. But when it tries to make reasonably quick decisions, multiple RCTs help separate signal from noise. Though RCT doubles the time to make a decision in PCC's Decision State, the convergence time of PCC using RCT only shows slight increase because it makes *better* decisions. With $T_m = 1.0 \cdot RTT$ and $\varepsilon_{min} = 0.01$, RCT trades 3% increase in convergence time for **35% reduction in rate variance**.

4.3 PCC is Deployable

4.3.1 A Promising Solution to TCP Friendliness

		30ms	60ms	90ms
$\beta = 10$	10Mbit/s	0.94	0.75	0.67
	50Mbit/s	0.74	0.73	0.81
	90Mbit/s	0.89	0.91	1.01
$\beta = 100$	10Mbit/s	0.71	0.58	0.63
	50Mbit/s	0.56	0.58	0.54
	90Mbit/s	0.63	0.62	0.88

Table 2: ***PCC can be TCP friendly***

To illustrate that PCC does not have to be TCP unfriendly, we evaluate the utility function proposed in § 2.4. We initiate two competing flows on Emulab: a *reference flow* running TCP CUBIC, and a *competing flow* running either TCP CUBIC or PCC, under various bandwidth and latency combinations with bottleneck buffer always equal to the BDP. We compute the ratio of average (over five runs) of throughput of reference flow when it competes with CUBIC, divided by the same value when it competes with PCC. If the ratio is smaller than 1, PCC is more friendly than CUBIC. As shown in

[4] We first fix ε_{min} at 0.01 and vary the length of T_m from $4.8\times$RTT down to $1\times$RTT. Then we fix T_m at $1\times$RTT and vary ε_{min} from 0.01 to 0.05. This is not a full exploration of the parameter space, so other settings might actually achieve better trade-off points.

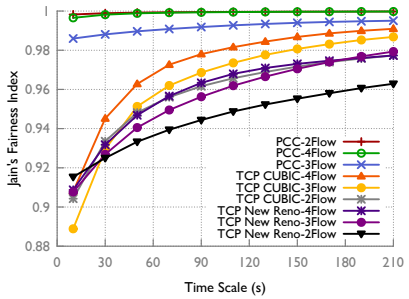


Figure 13: ***PCC achieves better fairness and convergence than TCP CUBIC***

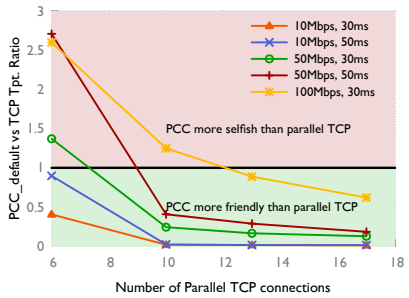


Figure 14: ***The default PCC utility function's TCP unfriendliness is similar to common selfish practice***

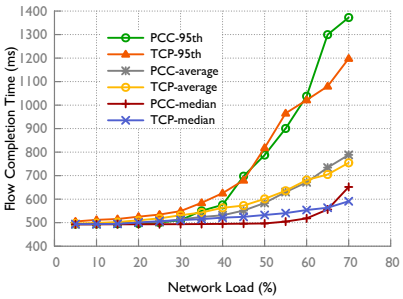


Figure 15: ***PCC can achieve flow completion time for short flows similar to TCP***

Table 2, PCC is already TCP friendly and with $\beta = 100$, PCC's performance is dominated by TCP. We consider this only a first step towards a TCP friendliness evaluation because these results also indicate PCC's friendliness can depend on the network environment. However, this initial result shows promise in finding a utility function that is sufficiently TCP friendly while also offering higher performance (note that this same utility function achieved higher performance than TCP in § 4.1.7).

4.3.2 TCP Friendliness of Default Utility Function

Applications today often adopt "selfish" practices to improve performance [3]; for example, Chrome opens between 6 (default) and 10 (maximum) parallel connections and Internet Explorer 11 opens between 13 and 17. We compare the unfriendliness of PCC's default utility function with these selfish practices by running two competing streams: one with a single PCC flow and the other with parallel TCP connections like the aforementioned web browsers. Fig. 14 shows the ratio of PCC's throughput to the total of the parallel TCP connections, over 100 seconds averaging over 5 runs under different bandwidth and RTT combinations. As shown in Fig. 14, PCC is similarly aggressive to 13 parallel TCP connections (IE11 default) and more friendly than 17 (IE11 maximum). Therefore, even using PCC's default utility function in the wild may not make the ecosystem dramatically worse for TCP. Moreover, simply using parallel connections cannot achieve consistently high performance and stability like PCC and initiating parallel connections involves added overhead in many applications.

4.3.3 Flow Completion Time for Short Flows

Will the "learning" nature of PCC harm flow completion time (FCT)? In this section, we resolve this concern by showing that with a startup phase similar to TCP (§3), PCC achieves similar FCT for short flows.

We set up a link on Emulab with 15 Mbps bandwidth and 60ms RTT. The sender sends short flows of 100KB each to receiver. The interval between two short flows is exponentially distributed with mean interval chosen to control the utilization of the link. As shown in Fig. 15, with network load ranging from 5% to 75%, PCC achieves similar FCT at the median and 95th percentile. The 95th percentile FCT with 75% utilization is 20% longer than TCP. However, we believe this is a solvable engineering issue. The purpose of this experiment is to show PCC does not fundamentally harm short flow performance. There is clearly room for improvement in the startup algorithm of all these protocols, but optimization for fast startup is intentionally outside the scope of this paper because it is a largely separate problem.

4.4 Flexiblity of PCC: An Example

In this section, we show a unique feature of PCC: expressing different data transfer objectives by using different utility functions. Because TCP is blind to the application's objective, a deep buffer (bufferbloat) is good for throughput-hungry applications but will build up long latency that kills performance of interactive applications. AQMs like CoDel [41] limits the queue to maintain low latency but degrades throughput. To cater to different applications' objective with TCP running on end hosts, it has been argued that programmable AQMs are needed in the network [47]. However, PCC can accomplish this with simple per-flow fair queuing (FQ). We only evaluate this feature in a per-flow fair queueing (FQ) environment; with a FIFO queue, the utility function may (or may not) affect dynamics and we leave that to future work. Borrowing the evaluation scenario from [47], an interactive flow is defined as a long-running flow that has the objective of maximizing its throughput-delay ratio, called the *power*. To make our point, we show that PCC + Bufferbloat + FQ has the same power for interactive flows as PCC + CoDel + FQ, and both have higher power than TCP + CoDel + FQ.

We set up a transmission pair on Emulab with 40Mbps bandwidth and 20ms RTT link running a CoDel implementation [4] with AQM parameters set to their default values. With TCP CUBIC and two simultaneous interactive flows, TCP + CoDel + FQ achieves 493.8$Mbit/s^2$, which is 10.5× more power than TCP + Bufferbloat + FQ (46.8$Mbit/s^2$).

For PCC, we use the following utility function modified from the default to express the objective of inter-

active flows: $u_i(x_i) = (T_i \cdot Sigmoid(L_i - 0.05) \cdot \frac{RTT_{n-1}}{RTT_n} - x_i L_i)/RTT_n$ where RTT_{n-1} and RTT_n are the average RTT of the previous and current MIs, respectively. This utility function expresses the objective of low latency and avoiding latency increase. With this utility function, we put PCC into the same test setting of TCP. Surprisingly, PCC + Bufferbloat + FQ and PCC + CoDel + FQ achieve essentially the same power for interactive flows ($772.8 Mbit/s^2$ and $766.3 Mbit/s^2$ respectively). This is because PCC was able to keep buffers very small: we observed *no packet drop* during the experiments even with PCC + CoDel + FQ so PCC's self-inflicted latency never exceeded the latency threshold of CoDel. That is to say, CoDel becomes useless when PCC is used in end-hosts. Moreover, PCC + Bufferbloat + FQ achieves 55% higher power than TCP + CoDel + FQ, indicating that even with AQM, TCP is still suboptimal at realizing the applications' transmission objective.

5 Related work

It has long been clear that TCP lacks enough information, or the right information, to make optimal rate control decisions. XCP [35] and RCP [22] solved this by using explicit feedback from the network to directly set the sender's rate. But this requires new protocols, router hardware, and packet header formats, so deployment is rare.

Numerous designs modify TCP, e.g. [23,31,38,52,55], but fail to acheive consistent high performance, because they still inherit TCP's hardwired mapping architecture. As we evaluated in § 4, they partially mitigate TCP's problems in the specially assumed network scenarios but still suffer from performance degradation when their assumptions are violated. As another example, FAST TCP [52] uses prolonged latency as a congestion signal for high BDP connections. However, it models the network queue in an ideal way and its performance degrades under RTT variance [21], incorrect estimation of baseline RTT [49] and when competing with loss-based TCP protocols.

The method of Remy and TAO [48, 54] pushes TCP's architecture to the extreme: it searches through a large number of *hardwired mappings* under a network model with assumed parameters, e.g. number of senders, link speed, etc., and finds the best protocol under that simulated scenario. However, like all TCP variants, when the real network deviates from Remy's input assumption, performance degrades [48]. Moreover, random loss and many more real network "parameters" are not considered in Remy's network model and the effects are unclear.

Works such as PCP [20] and Packet Pair Flow Control [37] utilize techniques like packet-trains [33] to probe available bandwidth in the network. However, bandwidth probing protocols do not observe real performance like PCC does and make unreliable assumptions about the network. For example, real networks can easily violate the assumptions about packet inter-arrival latency embedded in BP (e.g. latency jitter due to middleboxes, software routers or virtualization layers), rendering incorrect estimates that harm performance.

Several past works also explicitly quantify utility. Analysis of congestion control as a global optimization [36, 40] implemented by a distributed protocol is not under the same framework as our analysis, which defines a utility function and finds the global Nash equilibrium. Other work explicitly defines a utility function for a congestion control protocol, either local [19] or global [48, 54]. However, the resulting control algorithms are still TCP-like hardwired mappings, whereas each PCC sender optimizes utility using a learning algorithm that obtains direct experimental evidence of how sending rate affects utility. Take Remy and TAO again as an example: there is a global optimization goal, used to guide the choice of protocol; but at the end of day the senders use hardwired control to attempt to optimize for that goal, which can fail when those assumptions are violated and moreover, one has to change the hardwired mapping if the goal changes.

6 Conclusion

This paper made the case that Performance-oriented Congestion Control, in which senders control their rate based on direct experimental evidence of the connection between their actions and performance, offers a promising new architecture to achieve consistent high performance. Within this architecture, many questions remain. One major area is in the choice of utility function: Is there a utility function that provably converges to a Nash equilibrium while being TCP friendly? Does a utility function which incorporates latency—clearly a generally-desirable objective—provably converge and experimentally perform as well as the default utility function used in most of our evaluation? More practically, our (userspace, UDP-based) prototype software encounters problems with accurate packet pacing and handling many flows as it scales.

7 Acknowledgement

We sincerely thank our shepherd Jon Howell and our reviewers for their valuable comments. We thank Srinivasan Keshav, Jeff Mogul, and Artur Makutunowicz for providing detailed and valuable comments at various stages of the project. We thank Google's QUIC team, including Ian Swett, for ongoing help with PCC's integration with the QUIC framework and Abdul Kabbani for helpful suggestions on software scalability. Finally, we gratefully acknowledge the support of NSF Grant 1149895, a Google Research Award, and an Alfred P. Sloan Research Fellowship.

References

[1] http://goo.gl/lGvnis.

[2] http://www.dataexpedition.com/.

[3] Browsers usually opens parallel TCP connections to boost performance. http://goo.gl/LT0HbQ.

[4] Codel Linux implementation. http://goo.gl/06VQqG.

[5] ESNet. http://www.es.net/.

[6] Full proof of Theorem 1. http://web.engr.illinois.edu/~modong2/full_proof.pdf.

[7] GENI testbed. http://www.geni.net/.

[8] Internet2 ION service. http://webdev0.internet2.edu/ion/.

[9] Level 3 Bandwidth Optimizer. http://goo.gl/KFQ3aS.

[10] Limelight Orchestrate(TM) Content Delivery. http://goo.gl/M5oHnV.

[11] List of customers of a commercial high speed data transfer service provider. http://goo.gl/kgecRX.

[12] PCP user-space implementation. http://homes.cs.washington.edu/~arvind/pcp/pcp-ulevel.tar.gz.

[13] Report mentioning revenue range of one high speed data transfer service company. http://goo.gl/7mzB0V.

[14] Satellite link latency. http://goo.gl/xnqCih.

[15] TelliShape per-user traffic shaper. http://tinyurl.com/pm6hqrh.

[16] UDT: UDP-based data transfer. udt.sourceforge.net.

[17] PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services. July 2010. http://www.planet-lab.org.

[18] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *Proc. ACM SIGCOMM*, September 2010.

[19] T. Alpcan and T. Basar. A utility-based congestion control scheme for Internet-style networks with delay. *Proc. IEEE INFOCOM*, April 2003.

[20] T. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan. PCP: efficient endpoint congestion control. *Proc. NSDI*, May 2006.

[21] H. Bullot and R. L. Cottrell. Evaluation of advanced tcp stacks on fast longdistance production networks. *Proc. PFLDNeT*, February 2004.

[22] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and K. van der Merwe. Design and implementation of a routing control platform. *Proc. NSDI*, April 2005.

[23] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, August 2004.

[24] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. *Proc. ACM SIGCOMM Workshop on Research on Enterprise Networking*, August 2009.

[25] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control and why this means we need new algorithms. *ACM Computer Communication Review*, January 2006.

[26] ESnet. Virtual Circuits (OSCARS), May 2013. http://goo.gl/qKVOnS.

[27] Y. Ganjali and N. McKeown. Update on buffer sizing in internet routers. *ACM Computer Communication Review*, October 2006.

[28] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the Internet. *ACM Queue*, December 2011.

[29] Y. Gu, X. Hong, M. Mazzucco, and R. Grossman. SABUL: A high performance data transfer protocol. *IEEE Communications Letters*, 2003.

[30] S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: a slice abstraction for software-defined networks. *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, August 2012.

[31] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 2008.

[32] Y. Hu and V. Li. Satellite-based Internet: a tutorial. *Communications Magazine*, 2001.

[33] M. Jain and C. Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. *Proc. Passive and Active Measurement (PAM)*, March 2002.

[34] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *Proc. CoNEXT*, December 2012.

[35] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *Proc. ACM SIGCOMM*, August 2002.

[36] F. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 1998.

[37] S. Keshav. *The packet pair flow control protocol*. ICSI, 1991.

[38] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 2008.

[39] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang. TCP Westwood: bandwidth estimation for enhanced transport over wireless links. *Proc. ACM Mobicom*, July 2001.

[40] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking (ToN)*, 2000.

[41] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 2012.

[42] H. Obata, K. Tamehiro, and K. Ishida. Experimental evaluation of TCP-STAR for satellite Internet over WINDS. *Proc. Autonomous Decentralized Systems (ISADS)*, June 2011.

[43] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. *Proc. ACM SIGCOMM*, August 2012.

[44] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: practical work-conserving bandwidth guarantees for cloud computing. *Proc. ACM SIGCOMM*, August 2013.

[45] R. Prasad, C. Dovrolis, and M. Thottan. Router buffer sizing revisited: the role of the output/input capacity ratio. *Proc. CoNEXT*, December 2007.

[46] J. Rosen. Existence and uniqueness of equilibrium point for concave n-person games. *Econometrica*, 1965.

[47] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No silver bullet: extending SDN to the data plane. *Proc. HotNets*, July 2013.

[48] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An experimental study of the learnability of congestion control. *Proc. ACM SIGCOMM*, August 2014.

[49] L. Tan, C. Yuan, and M. Zukerman. FAST TCP: Fairness and queuing issues. *IEEE Communication Letter*, August 2005.

[50] VSAT Systems. TCP/IP protocol and other applications over satellite. `http://goo.gl/E6q6Yf`.

[51] G. Vu-Brugier, R. Stanojevic, D. J. Leith, and R. Shorten. A critique of recently proposed buffer-sizing strategies. *ACM Computer Communication Review*, 2007.

[52] D. Wei, C. Jin, S. Low, and S. Hegde. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 2006.

[53] B. White, J. Lepreau, L. Stoller, R. Ricci, G. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *Proc. OSDI*, December 2002.

[54] K. Winstein and H. Balakrishnan. TCP ex Machina: computer-generated congestion control. *Proc. ACM SIGCOMM*, August 2013.

[55] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data center networks. *Proc. CoNEXT*, November 2010.