A PCC-Vivace Kernel Module for Congestion Control

Nathan Jay*, Tomer Gilad , Nogah Frankel****

Tong Meng*, Brighten Godfrey*, Michael Schapira**

Jae Won Chung*, Vikram Siwach***, Jamal Hadi Salim*****
University of Illinois Urbana-Champaign*, Hebrew University of Jerusalem in Israel**, Verizon***

Abstract

The introduction of a high performance packet scheduler to the Linux kernel and modular congestion control system from BBR makes it possible to draw research congestion control algorithms into the Linux kernel. In this paper, we discuss the introduction of the PCC family of congestion control algorithms into the Linux kernel. We implement both loss- and latency-based congestion control using the rate-based PCC architecture and discuss possible interfaces for choosing congestion control parameters.

Keywords

Linux, networking, TCP, low latency, PCC

Introduction

Research on Internet congestion control has produced a variety of transport layer implementations in the past decades (*e.g.*, [6, 3, 4, 2, 10, 1, 8], *etc.*). Many research algorithms have stayed in the realm of research because of former challenges in implementing congestion control in modern operating systems. Thankfully, the recent introduction of rate-based mechanisms for congestion control [2] and the creation of congestion control hooks has made it much easier to implement novel congestion control algorithms in the Linux kernel. These new mechanisms allow developers and researchers alike to deploy research algorithms in the open-source world.

Ideally, congestion control should deliver consistently high performance, *i.e.*, close-to-capacity throughput with close-to-minimum latency, even in the presence of many competing flows. This objective involves a tradeoff: obtaining full link utilization on a bottleneck link requires an algorithm to keep packets in the buffer preceding the bottleneck. If the bottleneck link has unpredictably variable capacity as LTE links may [cite], then the congestion controller must keep sufficient data in the bottleneck buffer to take advantage of sudden increases in capacity, and it must back off to keep latency low if the link capacity decreases. This challenging scenario is already extremely common, with some sources estimating that the majority of webpage traffic is now served to mobile devices [9].

Congestion control must also serve a variety of applications whose goals may differ. Loading static webpages, downloading bulk files and streaming buffered or live videos may have different optimal operating points for throughput and latency. Often, the only way for network operators or developers to choose different operating points is to choose a completely different congestion control algorithms. Unfortunately, the objective of each congestion control algorithm may not be clear, forcing network operators to test a variety of algorithms and develop in-house implementations to meet their needs.

Recognizing these challenges for congestion control, and the great opportunity afforded by the improved Linux networking code, we implement PCC-Vivace [4] with both loss- and latency-based utility functions in the Linux kernel and discuss further opportunities for high-performance congestion control and configurability.

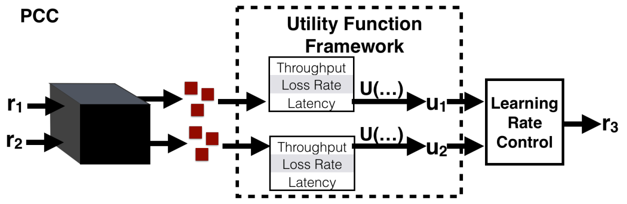

Figure 1: PCC Architecture

Background

Our kernel module is an implementation of PCC-Vivace congestion control algorithm, so we provide a brief description here. Interested readers can check the full paper for more information [4].

PCC's core architecture is based on RTT-length monitor intervals and an explicit utility function. By sending at a variety of rates, each for a full RTT and monitoring the resulting network statistics, PCC can compute the utility of each rate and choose a sending rate that optimizes utility. The rates tested and resulting rate decision will depend on the current state of the PCC state machine as described below. For utility functions, we generally reward throughput while penalizing loss or latency increases. Our module includes two utility functions, *Allegro* and *Vivace*, named for their source implementations [3, 4].

The Allegro utility function rewards sending rate but penalizes the flow for lost data. This essentially optimizes

$$\mathrm{U}_V(r) = r * (1 - \alpha \frac{dRTT}{dt} - \beta L)$$
$$\mathrm{U}_A(r) = \frac{r}{1+e^{100L}} - rL$$

Figure 2: Allegro and Vivace utility functions where r is the rate, L is the loss rate. α and β are weights on latency inflation and loss chosen to match the original Vivace implementation [4].

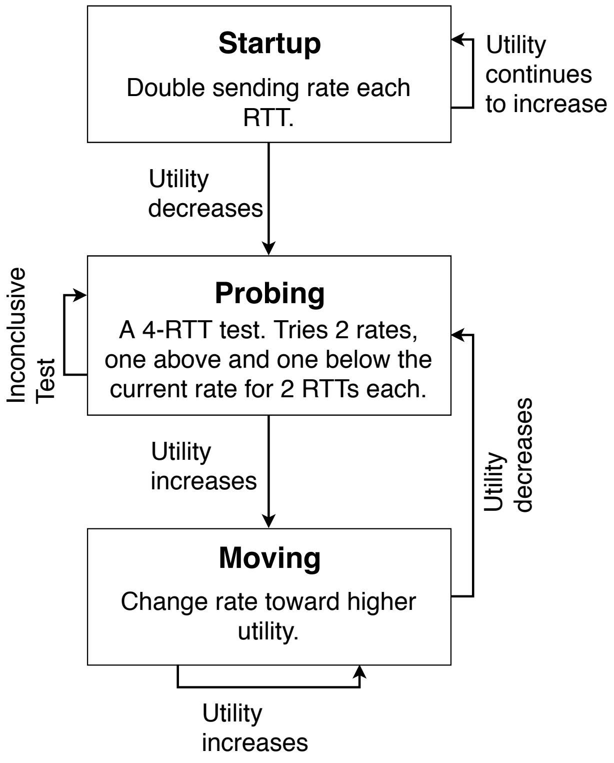
Startup
Double sending rate each RTT.
Utility continues to increase
Utility decreases
Probing
A 4-RTT test. Tries 2 rates, one above and one below the current rate for 2 RTTs each.
Inconclusive Test
Utility increases
Utility decreases
Moving
Change rate toward higher utility.
Utility increases

Figure 3: PCC State Machine

for throughput without regard to latency, and with an explicit tradeoff against loss. The Vivace utility function rewards throughput, or decreases in latency, while penalizing lost packets and increases in latency. This means that Vivace will allow queues to drain if it has a link to itself, or if other flows allow the queues to drain, but in the present of a buffer-filling flow like CUBIC, it will tradeoff between throughput and loss rate. Vivace will back off quickly as latency increases.

PCC State Machine

PCC-Vivace's full implementation has three states, *startup*, *probing* and *moving*.

In startup, PCC repeatedly doubles its sending rate (once per RTT) and observes the utility, decreasing its rate by half and transitioning to the probing state when the observed utility decreases.

The probing state is intended to determine if the best sending rate is above or below the current rate. In the probing state, PCC chooses one rate slightly above and one rate slightly below the current sending rate. It then tests each rate twice (with each test lasting one RTT) and determines which rate has higher utility. If the two tests have the same result, PCC enters the moving state in the direction of greater utility, if the tests are inconclusive, PCC repeats the probing process.

In the moving state, PCC repeatedly changes the sending rate toward the direction of greater utility. After each RTT, PCC computes the utility of the latest rate and compute the gradient of utility with respect to rate. It uses the accelerating gradient ascent mechanism described in [4]. When the observed utility decreases, PCC transitions back to the probing state.

Benefits of PCC

PCC offers two key beneifts: in the short term, it provides a high performance, low latency congestion control algorithm in the form of PCC-Vivace; in the long term, it provides an explicit utility-based framework that will allow others to specify new utility functions that suit their applications.

PCC-Vivace is a recently-published algorithm that has shown promising results in several congestion control metrics like loss tolerance, convergence loss rate, and convergence latency. PCC's architecture contributes to these benefits in three ways. First, PCC's monitor interval algorithm takes an empirical approach to determining the best sending rate. This approach makes few assumptions about the network, so it works well in many network conditions. Second, PCC leverages a well-known gradient ascent technique to quickly find the rate that maximizes utility (with some restrictions on the utility function). By tracking the gradient of utility with respect to rate, PCC take larger steps when it is far from the optimal rate and smaller steps as it nears optimal. Finally, PCC's architecture takes advantage of game-theoretic utility functions whose convergence point is a fair share of link bandwidth. Subject to measurement errors and noise, this means that multiple PCC senders independently making rate control decisions should all gain a fair share of bandwidth.

PCC's architecture makes it easy for others to implement utility functions that optimize for application-specific goals. Our current utility functions show two possible operating points: low latency, and throughput without regard to latency. Ongoing research considers even more interesting operating points for congestion control, including scavenger congestion control that takes bandwidth only if the path appears to be underutilized. These are only a handful of possible utility functions. One could imagine making a video-specific utility function that has information about bitrate and rebuffering, or a browser-based utility that considers page loading rate. This initial implementation of PCC can provide a basis for all of these future directions.

Kernel Implementation

Previous implementations of PCC were done in user-space based on the UDT library [5]. These implementations had several heavy-weight mechanisms available and made assumptions about feedback (per packet acks and timestamps, unique packet numbers, etc.) that could not be used in a high-performance kernel implementation. In this section, we discuss the changes we made to PCC to adapt to the Linux kernel environment.

ACK Feedback

The user-space implementation of PCC relies on acks and RTT estimates for every packet to accurately measure changes in latency, and to easily attribute lost packets to the sending rate at the time those packets were lost. Unfortunately, tracking this much data is prohibitively costly (and contributes to the user-space version's bandwidth limitation of about 1Gbps). We address this issues in two ways. First, instead of measuring per-packet RTT, we use the smoothed RTT estimate provided by the TCP socket. We sample this RTT at the beginning and end of each sending interval to determine what effect our rate choice had on latency. Second, because we now have only two samples for latency per RTT (instead of one per packet), we significantly increase the low-pass filter for latency inflation from 1% to 3%. Our testing showed that this may slow the reaction to latency inflation, but reduces false-positive signals of inflation significantly.

Utility Gradient Calculation

Calculating the gradient of utility with respect to sending rate tells us which direction to move the sending rate to achieve greater utility, but it can be very noisy. The equation we use is:

$$\frac{utility_2 - utility_1}{rate_2 - rate_1}$$

For rates that are very similar with a small amount of network noise (say one more packet of 100 is lost in interval 2), the gradient can be extremely steep because the observed utility is quite different and the rates are close together. The user-space implementation of PCC attempts to address this by averaging several gradients, but outliers can still lead to incorrect decisions. We reduce the impact of noise by introducing a 2% minimum rate difference for gradient computations, and a similar minimum rate change required in the moving state.

Packet-Rate Association

One critical facility of PCC is associating packet results (delivered or lost) with the rate at which packets were sent. If this mechanism is inaccurate, we will make incorrect rate decisions. In the user-space implementations, this is done by assigning unique IDs to each packet and recording the interval those packets belong to. In the kernel, keeping any per-packet data is impractical. While the kernel does provide data in the form of rate_samples, this data is overlapping and provides no additional information compared to the tcp_sock structure alone. We instead modify PCC monitor intervals in two ways. First, monitor intervals record the TCP socket's data_segs_out when they start and end. Then using the socket's lost and delivered values, we can determine approximately when acks are probably for packets in our interval of interest. This mechanism assumes that reordering will mostly occur on a smaller scale than 1-RTT. Second, we ignore the last few acks (at most 20% in each monitor interval), and we use this first part of the monitor interval to calculate utility. This mechanism is intended to reduce the effect of reordering and compensate for the fact that we often know about lost packets later than we know about acked packets.

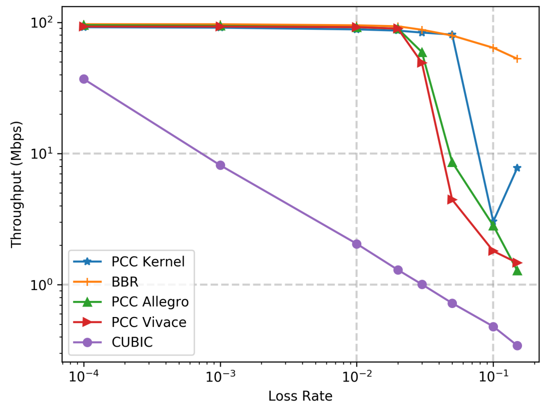

Figure 4: Throughput on emulated links with 100Mbps capacity and various random loss rates. The PCC schemes (both user-space and current kernel) have higher loss resilience than CUBIC but lower than BBR.

Evaluation

We install our kernel module into the 4.16 Linux kernel and test it in emulated network scenarios that demonstrate its core use cases. We perform the tests using Pantheon [8] (which uses mahimahi [7] and iperf) to emulate networks locally. We compare our kernel implementation to the existing CUBIC and BBR implementations, as well as the user-space implementations of PCC-Allegro and PCC-Vivace. We also partner with a major cellular service provider to test our kernel implementation in a realistic setting.

Random Loss Resilience

We test random loss resilience by running local tests with Pantheon. We configured out test link to with 100Mbps capacity, 30ms delay, 750KB buffer, and varied random loss. We run each test for two minutes and repeat them three times. As expected, CUBIC backs off to just 2Mbps throughput under 1% random loss, while all other algorithms manage about 90% throughput. BBR maintains 75Mbps throughput at 5% loss and degrades to 50Mbps throughput at 15% loss. Throughput degrades significantly for the user-space PCC impelementations around 3 to 5% loss, while the kernel implementation maintains 80Mbps up to 5% loss before falling off quickly.

We attribute the additional loss resilience of the kernel implementation to the approximate packet-rate association of our implementation. When we cannot accurately attribute acks and losses that are counted near the edge of a monitor interval definitively, we do not count them. Since many losses are realized later than acks, this means that we disproportionately ignore losses, both decreasing observed loss rate (which

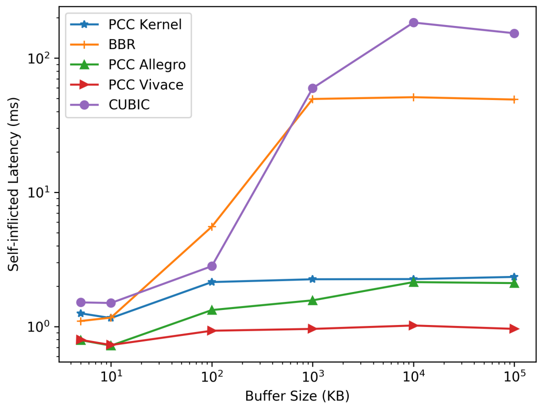

Figure 5: Self-inflicted latency by buffer size.

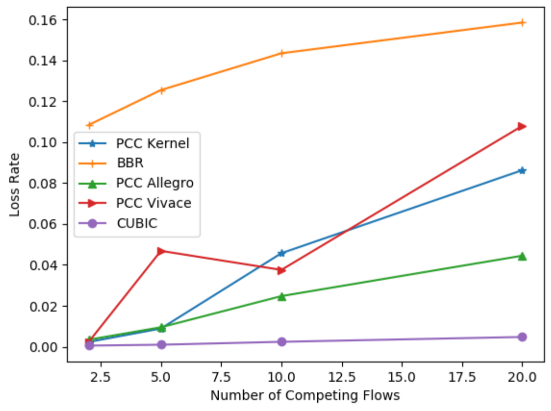

Figure 6: Loss rate by number of competing senders on a 100Mbps bottleneck link.

improves resilience), and potentially impacting convergence. In practice, we find that our implementation still converges to a reasonable loss rate of about 5% for 10 flows, discussed in more detail later.

Low Buffer Bloat

We test buffer bloat by running local Pantheon tests with varied buffer sizes and measuring the resulting self-inflicted latency. Our test uses an emulated link with 100Mbps and a base 30ms rtt. We ran three tests of each buffer size, which varied from 2KB to 1MB. CUBIC and BBR both completely filled the buffer, resulting in up to 200% increases in round trip time. The PCC variants, including our kernel implementation, increase latency only slightly (about 1ms), regardless of buffer size.

Loss Rate at Convergence

While TCP maintains a very low loss rate at convergence, BBR often has a much higher loss rate, around 10%. We demonstrate low loss rate at convergence for our module by emulating a 100Mbps, 30ms rtt bottleneck link with a 750KB buffer, no random loss and a varied number of senders. Each test lasts for two minutes and is repeated three times. We report the average loss rate for the entire duration of the test even though the latency sensitive algorithms (PCC-Kernel and PCC-Vivace) may only induce loss during the first few seconds of startup for low numbers of senders.

For just two flows, BBR's loss rate is at 10%, while all other algorithms induce less than 0.5% loss. As the number of flows grows, the algorithms have increasing convergence loss rate, with TCP increasing most slowly. Even with 10 senders though, PCC-Kernel induces a convergence loss rate of just 5%.

Discussion

Fairly recent updates to the Linux kernel have made it substantially easier to implement novel congestion controllers, but our implementation of PCC was somewhat limited compared to what we imagine may be possible and useful. We present two areas of interest for which interfaces from user-space to the kernel may be beneficial: congestion control parameters and (PCC specific) user-specified utility functions.

Congestion Control Parameters It seems unavoidable that congestion control algorithms work best in the environments at the center of their design range. TCP requires a buffer long enough to avoid losses while filling, but short enough to avoid extreme delays, and it assumes that losses are always a signal of congestion. BBR makes assumptions about reasonable intervals for probing link capacity (mobile networks can be highly variable though), and it gets high throughput by filling buffers to a degree that may be unreasonable for low-latency applications. While PCC was designed without assumptions about network traffic or conditions, it has many parameters determined empirically based on a variety of tests: the dynamic rate bound starts at 10%, the latency inflation filter is 3%, the gradient step size starts at 400Kbps. These parameters may be more easily updated than the assumptions of CUBIC, but they will not be ideal for everyone. Our choice of parameters seems sensible for current networks and technology, but network operators with different traffic may find better parameters for their networks. Instead of forcing network operators to consider other congestion controllers or provide kernel modules of their own, an interface for specifying congestion control parameters may broaden the use case for each congestion controller, resulting in fewer, more configurable control algorithms.

User-Specified Utility Functions In our implementation, we provide two congestion controllers in one module by registering two different congestion_ops structs. The controllers differ only in utility function with one reacting only to loss, and the other reacting to both loss and latency inflation. While these utility functions have a number of sensible properties, they are far from the only functions one might imagine. Perhaps a distributed AR-VR application requires stable latency of less than 120ms and would prefer to stop briefly than make its users sick. This application might use its own transport protocol, place checks for the required latency, probe network conditions and operate only in the desired latency range. Developing, testing and maintaining this protocol would represent a significant effort. On the other hand, the simple utility function shown below can accomplish a similar goal in the PCC module we present with just a few lines of code.

A small amount of parameter passing can be done with Netlink sockets, and switching between a small number of algorithms can be done by simply registering more congestion_ops, but PCC brings the potential for even greater configurability. As more and more applications become network-dependent, a richer interface may allow faster innovation in Linux networking.

Conclusion

Improvements to the Linux networking stack have allowed us to implement the recently-published PCC-Vivace rate control algorithm. This initial Linux kernel implementation delivers a number of the benefits of the user-space research version while adapting to the high-performance, low-overhead environment of the Linux kernel. We have shown promising performance results in several cases of interest and hope to continue improving performance and robustness. Additionally, PCC's explicit utility functions makes it more adaptable than many algorithms, hopefully serving the Linux kernel well as increasingly diverse applications and devices rely on networking.

References

[1] Arun, V. 2018. Copa: Practical delay-based congestion control for the internet.

[2] Cardwell, N.; Cheng, Y.; Gunn, C. S.; Yeganeh, S. H.; and Jacobson, V. 2016. BBR: Congestion-based congestion control. *ACM Queue* 14(5):50.

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

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

[5] Gu, Y. 2005. *UDT: a high performance data transport protocol*. University of Illinois at Chicago.

[6] Ha, S.; Rhee, I.; and Xu, L. 2008. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review* 42(5):64–74.

[7] Netravali, R.; Sivaraman, A.; Da, S.; Goyal, A.; Winstein, K.; Mickens, J.; and Balakrishnan, H. 2015. Mahimahi: Accurate record-and-replay for http.

[8] Pantheon. http://pantheon.stanford.edu/.

[9] Statista. 2018. Percentage of all global web pages served to mobile phones from 2009 to 2018. https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/.

[10] Winstein, K., and Balakrishnan, H. 2013. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM*.