PCC Proteus: Scavenger Transport And Beyond

Tong Meng[1] Neta Rozen Schiff[2] P. Brighten Godfrey[1] Michael Schapira[2]
[1]UIUC [2]Hebrew University of Jerusalem

ABSTRACT

Many Internet applications need high bandwidth but are not time sensitive. This motivates a congestion control "scavenger" that voluntarily yields to higher-priority applications, thus improving overall user experience. However, the existing scavenger protocol, LEDBAT, often fails to yield, has performance shortcomings, and requires a codebase separate from other transport protocols.

We present PCC Proteus, a new congestion controller that can behave as an effective scavenger or primary protocol. Proteus incorporates several novel ideas to ensure that it yields to primary flows while still obtaining high performance, including using latency deviation as a signal of competition, and techniques for noise tolerance in dynamic environments. By extending the existing PCC utility framework, Proteus also allows applications to specify a flexible utility function that, in addition to scavenger and primary modes, allows choice of hybrid modes between the two, better capturing application needs. Extensive emulation and real-world evaluation show that Proteus is capable of both being a much more effective scavenger than LEDBAT, and of acting as a high performance primary protocol. Application-level experiments show Proteus significantly improves page load time and DASH video delivery, and its hybrid mode significantly reduces rebuffering in a bandwidth-constrained environment.

CCS CONCEPTS

• **Networks** → **Transport protocols**.

KEYWORDS

Congestion Control; Scavenger

ACM Reference Format:
Tong Meng, Neta Rozen Schiff, P. Brighten Godfrey, Michael Schapira. 2020. PCC Proteus: Scavenger Transport And Beyond. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3387514.3405891

1 INTRODUCTION

It was a scorching summer. A camel and a zebra embarked on a desert expedition. The two companions brought a container of water, which, being best friends, they decided to share equally during their journey. Unfortunately, the zebra suffered serious dehydration, even though the camel could easily have waited until they reached an oasis to quench its thirst. The moral of the story is that equally sharing resources is often not optimal when user requirements are heterogeneous.

The same principle applies to the classic problem of Internet congestion control. Traditional congestion control, dividing bandwidth equally among flows on a common bottleneck, may result in lower network-wide utility. For example, in a typical home, Alice may be watching a high-definition video, while Bob is sleeping at the same time, having left his device downloading a large volume of files from a remote-updated Dropbox folder. Ideally, Alice should enjoy high video quality smoothly as usual, while the Dropbox download could be delayed by hours without Bob even noticing. However, thanks to the "fair" transport layer, Alice suffers from constant video quality degradation.

Among the diverse applications using the network today, there are many with similarly elastic resource requirements, for at least some of their flows: software update, online data backup, background replication of cloud storage (e.g., Dropbox), proactive cache warmup in CDNs, and aggregation of IoT sensor data for offline analytics, among others. Those applications could occupy bandwidth that is excessive for their users, and could have been consumed by more data-intensive applications.

Even those applications that are often time-sensitive sometimes become elastic. For example, a video client may not need to urgently preload chunks as long as the highest bitrate is smoothly streamed, or when the client has little free space in its playback buffer. When a machine learning task is hindered by a slow worker, receiving its input for its next phase of work may be lower priority. Likewise, applications with usually-elastic requirements may at times demand increased priority, *e.g.*, when a Dropbox user specifically requests to view a file. Standard congestion control protocols cannot accommodate such context-sensitive priorities.

We claim that a *scavenger* mode that yields to normal (*primary*) flows helps mitigate this problem, by deprioritizing traffic with elastic requirements. Of course, this approach will not be as close to optimal as a centralized resource allocator, but its deployability makes it a practical approach for general-purpose Internet congestion control, *i.e.*, within end-to-end transport. More specifically, a scavenger has two goals:

(1) **Yielding**: Minimally impact primary flows on a common bottleneck. That is, flows running traditional transport protocols should experience throughput, latency, *etc.*, almost as if the scavenger were not present.
(2) **Performance**: Act like a traditional congestion controller when only scavengers share a bottleneck. For example, competing scavengers should fairly and fully utilize bandwidth while minimizing queueing delay.

The main existing scavenger proposal, Low Extra Delay Background Transport (LEDBAT) [34], tries to defer to high priority

SIGCOMM '20, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405891

flows by never adding more than a target delay in queueing. However, it is designed mainly as a scavenger against TCP CUBIC [21]. As we will see, it is relatively aggressive compared with recently proposed latency-sensitive protocols such as BBR [13] and COPA [8]. When only LEDBAT flows compete, it also has shortcomings: it gives an advantage to latecomers [32], and as its design is based on traditional TCP, it inherits problems like lack of tolerance to random packet loss and poor performance with shallow buffers. Therefore, LEDBAT falls short of both goals above.

Furthermore, as explained above, a flow may dynamically switch between scavenger and primary modes. This is hard for LEDBAT, which, as deployed in Microsoft Windows Server 2019 [7] and BitTorrent [32], is implemented separately from primary protocols. Maintaining separate codebases also imposes an increased software engineering burden, and makes it difficult for improvements in the implementation of one protocol to benefit another.[1] We thus add a third goal:

(3) **Flexibility**: A single transport protocol framework and codebase should be able to easily switch between primary mode and scavenger mode.

We aim to design a congestion control scavenger that meets all three goals. However, this is challenging. The scavenger needs to be both conservative (against primary flows) and aggressive (when alone or among scavengers). It should ideally meet the desired goals whatever the primary protocol is, and we can no longer assume that will always be CUBIC: BBR has widespread deployment, and many research advances are waiting in the wings [8, 16, 17, 42], mostly latency-aware.

Our solution, PCC *Proteus*, extends the utility-based approach in PCC [16, 17] with the following design contributions:

- To achieve our goals of **performance** and **yielding**, we build utility function objectives for both primary (Proteus-P) and scavenger (Proteus-S) senders. The scavenger utility employs a penalty based on latency deviation which provides a sensitive signal of competition and is typically not used by primary flows, allowing Proteus-S to act as a good scavenger even relative to latency-aware protocols. Our theoretical analyses show that competing Proteus-P and Proteus-S senders produce a unique equilibrium, and this equilibrium is fair when all senders use the same utility function.
- We extend the utility design to support more than two modes of service, including a hybrid mode, Proteus-H, with a piecewise utility function that switches between primary and scavenger modes at an adaptive threshold determined by cross-layer application requirements (*e.g.*, maximum bitrate for an online video). The modular architecture of Proteus allows applications to easily select modes and fulfills the goal of **flexibility**.
- To further improve performance in light of Proteus-S's sensitivity to latency deviation, we introduce techniques to better respond to network latency noise (i.e., non-congestion variability in end-to-end latency associated with the channel rather than with the senders' chosen rates) such that the scavenger can achieve robust performance in highly dynamic environments such as wireless networks.

[1]This cost is hard to quantify, but anecdotally, multiple major content providers have expressed this concern to us.

We implement Proteus and evaluate it and LEDBAT along with many primary protocols (CUBIC, BBR, COPA, and PCC Vivace) in emulated networks and the live Internet. To the best of our knowledge, this is the broadest performance test of scavengers currently available. Our results show that Proteus achieves the scavenger goals more effectively:

- *Yielding:* Proteus yields $\geq$ 90% of bandwidth to competing primary flows, while LEDBAT may yield less than 50%, particularly against modern latency-aware protocols like BBR and COPA. In application-level tests on the live Internet, web pages load 33% faster and DASH video delivery receives 2.5× higher bitrate when Proteus, instead of LEDBAT, is scavenging in the background.
- *Performance:* When scavengers compete with themselves, Proteus maintains a Jain's index over 90%, and reaches up to 1.75× higher than LEDBAT. Proteus needs 32× lower buffer to achieve 90% utilization when running alone.
- *Flexibility:* The hybrid mode in Proteus delivers up to 11% higher bitrate for 4K video and 68% lower rebuffering ratio in a video streaming benchmark.

Our code is available open source [4]. This work does not raise any ethical issues.

2 PRELIMINARIES AND MOTIVATION

2.1 When Does Scavenging Makes Sense?

There is a rich literature on prioritizing network bandwidth across flows in ways other than max-min fairness, such as using centralized knowledge of applications [23] or pricing [11], which are generally impractical for the present Internet.

In contrast, a scavenger's prioritization approach is very coarse: flows that are clearly low priority can voluntarily deprioritize themselves. Given that the scavenger may lack incentive to yield for the sake of another flow, and has no idea of other flows' true priorities, when does this make sense? Generally, we believe scavenging will be effective when (1) the scavenger is so time-insensitive that the cost is negligible, and (2) the *application designer* choosing to use a scavenger has some chance of benefiting, perhaps indirectly.

As an example, a mobile phone manufacturer may choose a scavenger for automated software updates. The long-running software update is unlikely to be significantly delayed by occasional higher-priority flows like web page loads, and even if it is, the user is unlikely to notice. The manufacturer benefits because other apps perform better, providing an improved user experience across the whole device.

As another example, some large cloud providers offer multiple popular services ranging from cloud storage to video delivery. Background replication of stored files from the cloud to a particular device can act as a scavenger with negligible cost. The provider benefits from improved video quality of experience on the same device, or on other devices with a shared bottleneck like a home DSL connection.

We believe there are numerous other use cases following this pattern. We will explore several in our evaluation.

2.2 Signaling Scavengers to Yield

Congestion control protocols typically reward and penalize specific control signals. Different sensitivities to these signals cause differential aggressiveness among competing protocols. What metric(s) should a scavenger use in order to yield to primary flows? We consider two approaches.

(1) Same metrics, greater penalty. The scavenger could adopt the same metrics as the primary protocol(s) of interest, but with a greater penalty so it is more conservative. For example, a design in [17] has greater or lesser tolerance for packet loss, for the purpose of proportional bandwidth allocation among senders. This approach has several difficulties.

First, metrics chosen by primary protocols generally represent something very undesirable happening in the network; so if the primary and scavenger protocols have different sensitivities to these important metrics, one or the other of them will sacrifice their performance as a good stand-alone congestion controller (violating our performance goal). For example, the aforementioned proportional allocation design [17] can cause very high loss in order to acquire more bandwidth.

Second, this approach assumes the primary and scavenger rely on the same or similar metrics, which may not be true with a diversity of primary protocol designs. Whatever its target bandwidth share is, the aforementioned proportional allocator of [17] can easily dominate a latency-sensitive sender.

(2) Different metric. Due to the above drawbacks, ideally, a scavenger would somehow take signals from a different metric than primary protocols of interest. To gain some insights on the requirements for this dedicated metric, we start by analyzing performance metrics adopted by existing primary protocols.

As a simplistic example, suppose a scavenger is intended to coexist with a latency-based primary protocol like PCC Vivace or COPA. The scavenger could use **packet loss** as a different metric, but the loss signal will come too late, if ever, since Vivace and COPA avoid filling queues.

LEDBAT's congestion signal is **RTT exceeding a threshold**, e.g., 100 ms above the minimum RTT. This signal often comes earlier than loss, but still fails for primary protocols that react to even earlier signals – as will occur with Vivace's and COPA's latency sensitivity. Even with CUBIC as primary, it will fail if a moderate-size buffer causes loss before latency inflation hits 100 ms.

Another interesting signal is **RTT gradient**, used by Timely [28] and PCC Vivace [17]. For example, Vivace calculates the gradient of recently received RTT samples, and avoids inflation by penalizing positive gradient. This will occur earlier than many other signals, but being in use by certain protocols and having similar latency-awareness to a protocol like COPA, it may not be appropriate for a scavenger. Furthermore, there is a chance that the gradient calculation (*e.g.*, linear regression in [17]) may average out some transient congestion-related RTT fluctuation. In an extreme case, RTT gradient may stay close to zero while the bottleneck buffer is repeatedly inflated and deflated by other concurrent senders.

To sum up, we can't hope to guarantee that a scavenger is robust to *every conceivable* primary protocol. But ideally, its signal of competition should be *typically not used* by primary protocols, and should provide as *early as possible* a signal of competing senders.

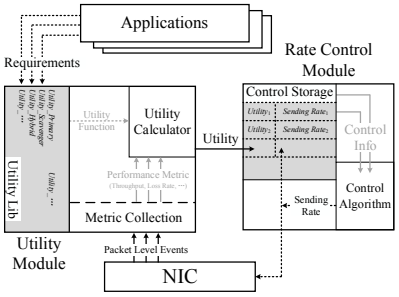


Figure 1: Proteus congestion control architecture

2.3 Motivation for Flexibility

The Internet congestion control domain accumulates massive codebases after decades of effort, ranging from traditional kernel modules to recent user-space implementations (*e.g.*, QUIC [25]). For example, the main existing scavenger protocol, LEDBAT, uses a different implementation from primary protocols such as BBR and CUBIC. As content providers try to optimize use cases with different needs (web traffic, video, real-time voice, scavengers, *etc.*), a proliferation of codebases would impose a nontrivial burden to develop initially and to maintain. The interaction between different protocols, especially the deprioritization of the scavenger, can be challenging to analyze, and brittle even with minor implementation-level code updates or bugfixes.

Separate implementations are also limited to coarse-grained priority changes. Most operating system kernels use a single congestion control protocol for all traffic. Although it is possible to configure different protocols on a per-socket basis or through tools such as *iptables*, this cannot accommodate priority changes mid-flow. For example, when a software update has a deadline requirement, it may want to yield dynamically, only after reaching a certain throughput.

Therefore, it would be of great value if there is a flexible, generic architecture for Internet congestion control that synthesizes both primary and scavenger modes, and eases the formal analysis for intra- and inter-protocol interaction, *i.e.*, scavenger *vs.* scavenger, and scavenger *vs.* primary flow.

3 PROTEUS DESIGN OVERVIEW

Fig. 1 summarizes the PCC Proteus architecture. We begin with a utility-based approach, similar to [16, 17]. Proteus separates congestion control into a utility module and a rate control module. The utility module has a library of utility functions, which may be tailored to different applications' needs. During data transmission, the utility module collects packet-level events (*e.g.*, loss, RTT, timeout), summarizes these metrics in the form of a numeric utility value, and associates the utility with the corresponding sending rate (or window size). Based on the relationship between different sending rates and their corresponding utilities, the rate control module algorithmically adjusts sending rate in the direction that empirically maximizes utility. The sender uses different sending rates in consecutive time intervals called monitor intervals (MIs),

and calculates the utility for each MI when all packets sent in that MI are acknowledged or lost.

We adopt PCC's utility approach because of its decoupled utility design and rate control. We can construct utility functions based on selected performance metrics (§2.2), while allowing these utility functions to share the same rate control algorithm (*e.g.*, the gradient-based rate control in PCC Vivace [17]). In fact, a sender can even switch utility functions dynamically within a running instance of the rate controller, which provides flexibility with minimal overhead (§2.3).

To apply this approach to our setting, Proteus introduces several new components. First, we design a new utility function for scavenger senders, called Proteus-S, that satisfies our **yielding** and **performance** goals by leveraging *latency deviation* as a signal of flow competition. Second, to satisfy our **flexibility** goal, the Proteus system supports dynamic utility function selection. The application may select or re-select a utility function in real-time, even in the middle of a flow. (In our user-space implementation, this is a simple API call.) In addition to Proteus-S, applications may select among a primary-flow utility function called Proteus-P, and a new hybrid-mode utility function that we call Proteus-H, which combines Proteus-S and Proteus-P in an *adaptive piece-wise* function according to applications' throughput requirements.

Within this high-level design, there are two hard problems, which are the subject of the upcoming sections. First, we design the new utility functions, especially the scavenger and its extension to hybrid mode (§4). We employ a game-theoretic analysis of equilibria when senders use the proposed utility functions to show that our performance goal is met for both our primary and scavenger utility functions. Second, because the scavenger utility function is sensitive to non-congestion RTT noise, we design novel noise-tolerant control mechanisms (§5).

4 PROTEUS UTILITY DESIGN

In this section, we present the utility functions employed by Proteus. After introducing the primary-protocol mode, we discuss the key metric employed by our scavenger and then the scavenger utility function for Proteus-S. Finally, we combine Proteus-P and Proteus-S into a hybrid mode (Proteus-H), using a piecewise utility function with cross-layer design, to improve bandwidth allocation.

4.1 Primary Utility Function

We begin with the relatively easy part: for Proteus-P, we use the PCC Vivace utility function [17] with a minor modification – negative RTT gradient is ignored:

$$u_P(x_i) = x_i^t - b \cdot x_i \cdot max\left\{0, \frac{d(RTT_i)}{dt}\right\} - c \cdot x_i \cdot L\,, \qquad (1)$$

where x_i is the sending rate of sender i, L is the observed loss rate, and $d(RTT_i)/dt$ represents RTT gradient. We ignore negative RTT gradient because we found it ultimately slows convergence (the sender tends to reduce its rate significantly below capacity so the queue drains quickly). This change still results in a fair equilibrium among competing Proteus-P senders, similar to [17]. We prove the following theorem in Appendix A).

Theorem 4.1. *In a shared bottleneck, n Proteus-P senders will converge to a fixed rate configuration* $(x_1^*, x_2^*, \cdots, x_n^*)$ *such that* $x_1^* = x_2^* = \cdots = x_n^*$*, and the link is fully utilized.*

The above Proteus-P utility function is latency-aware, and penalizes two performance metrics: RTT gradient and packet loss rate. Its convergence property is determined by three constant parameters as proved in [6]. The exponent t ($0 < t < 1$) should guarantee function concavity, and thus, the existence of a unique equilibrium. The latency coefficient b ($b > 0$) corresponds to a theoretical maximum number of competing senders on a specific bottleneck with no inflation in equilibrium state, *i.e.*, all senders' sending rates sum up to the bottleneck capacity. For example, $b = 900$ is used by PCC Vivace [17], aimed at up to 1000 competing senders on a bottleneck of at most 1000 Mbps. The coefficient c sets a threshold on random loss tolerance, *e.g.*, $c = 11.35$ to tolerate up to 5% random loss rate. In Proteus-P, we use the same default values as in [17] ($t = 0.9$, $b = 900$, $c = 11.35$).

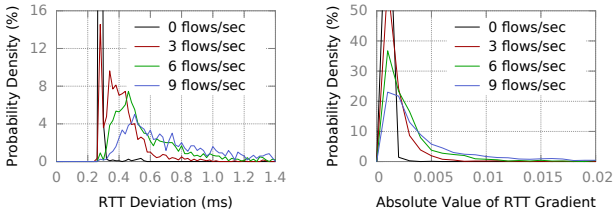
4.2 Competition Indicator: RTT Deviation

As analyzed in §2.2, the ideal signal for a scavenger is not just *ongoing* congestion; we would like to know of *impending* congestion, i.e., competition between flows. The implication is twofold. First, when there are multiple concurrent primary flows with a scavenger on a common bottleneck, if they are under-utilizing the bandwidth, the scavenger does not need to back off since there is no competition. Second, if the bottleneck buffer starts to be inflated and deflated alternatively due to flows probing for bandwidth, the scavenger should identify and react to such an early signal even before persistent congestion is induced. That is crucial to guarantee consistent low priority whether the primary flow is latency-aware or not. Based on that intuition, we choose *RTT deviation* as the indicator for flow competition. RTT deviation is the standard deviation of RTT samples within an MI and is calculated as

$$\sigma(RTT) = \sqrt{\frac{1}{n} \cdot \sum_j \left(RTT_j - \overline{RTT}\right)^2}\,,$$

where n is the number of RTT samples in the corresponding MI, and RTT_j and $\overline{RTT}$ are the j-th and the mean RTT of the MI, respectively.

RTT deviation captures the latency, and thus buffer occupancy dynamics, caused by flow competition. As long as the competing primary flows actively probe for available bandwidth and do not blindly speed up transmission, their competition will cause RTT fluctuation. This is true for both the early scenario when several latency-aware flows have just come close to full bandwidth utilization (where brief random bursts will cause RTT deviation even if the queue isn't persistently growing), and the late scenario when several loss-based flows already bloat the buffer to full occupancy. Even if multiple latency-sensitive flows converge at the steady state, primary senders still repeatedly probe *around* the steady-state rate so that they can adapt to channel dynamics, for example, bandwidth that is freed up when competing flows stop transmitting. That process causes positive RTT deviation, driving scavenger senders to back off. Of course, this is not to imply that RTT deviation is ideal when competing with any possible primary protocol, but we believe the above arguments are broadly true of current protocols.



(a) RTT Deviation (b) RTT Gradient

Figure 2: PDF of RTT deviation/gradient under Poisson arrival CUBIC flows

To show its advantage as a competition indicator, we compare RTT deviation with RTT gradient, a metric used by Proteus-P and Vivace for latency-awareness. More specifically, we will compare whether RTT deviation and absolute value of RTT gradient[2] indeed increase as flow competition increases. Considering that RTT gradient is a metric used by the primary protocol, the scavenger's dedicated performance metric should be something that produces an earlier signal of impending congestion than RTT gradient, so that the scavenger can also yield to latency-sensitive senders.

For this comparison, we set up a 100 Mbps, 60 ms RTT bottleneck with 1500 KB (2 BDP) buffer on Emulab. To emulate impending congestion, we generate short CUBIC flows with uniform flow sizes ranging from [20, 100] KB and Poisson interarrival time. To measure the two metrics, we use a fix-rate UDP flow at 20 Mbps and analyze the RTT gradient and deviation it observes in consecutive 1.5 RTT intervals across a 2-minute run. We test flow arrival rates ranging from 0-9 flows/sec, resulting in average link utilization in the range of 20-24%. This gives an indication of how the metrics perform as an early signal of congestion: congestion is not persistent, but random arrivals will cause occasional brief periods of congestion.

Fig. 2 presents the probability distribution function (PDF) of the two metrics. RTT deviation closely captures the extent of congestion, with its most significant probability peak getting further from the non-congestion peak as the arrival rate of CUBIC flows increases. Specifically, because the arrival rate of 3 flows/sec cannot introduce continuous congestion, RTT deviation shows two peaks, corresponding to non-congestion and congestion cases, respectively. In comparison, RTT gradient has more similar peaks in all cases. To quantify this, we calculate a *confusion probability* from the observed RTT samples, defined as the probability, across uniform-randomly-chosen pairs of (0 flow/sec, 9 flow/sec) samples, that a metric has smaller value in the congested (9 flow/sec) sample than in the non-congested (0 flow/sec) sample. RTT deviation has a confusion probability of 0.6%, significantly lower than 8.0% for RTT gradient. This validates that RTT deviation provides a more sensitive, early signal of RTT fluctuation dynamics, confirming our intuition in §2.2 that the averaging effect of RTT gradient delays or hides important information.

Nevertheless, latency noise (i.e., non-congestion RTT variability) may also cause deviation in some networks, like rapidly changing wireless networks. Proteus's rate controller (from [17]) helps ameliorate possible confusion by trying to distinguish whether the sender's rate is the *cause* of utility changes, by experimenting with different rates. However, we found realistic noise still impacted performance, so we designed noise control mechanisms (§5) for enhanced robustness.

[2]We use the absolute value of RTT gradient since either significantly positive or significantly negative gradients could indicate flow competition. RTT deviation, of course, is never negative.

4.3 Scavenger Utility Function

Given the intuition that RTT deviation can indicate when to yield, we define the utility function for Proteus-S as:

$$u_S(x_i) = u_P(x_i) - d \cdot x_i \cdot \sigma(RTT_i)\,, \tag{2}$$

where $u_P(x_i)$ is the utility function for Proteus-P, $d > 0$ is a parameter, and $\sigma(RTT_i)$ denotes the RTT deviation calculated from a corresponding MI.

We show that this utility function results in a fair equilibrium among competing Proteus-S senders, as required by our performance goal. We prove the following theorem in Appendix A.

Theorem 4.2. *In a shared bottleneck, n Proteus-S senders will converge to a fixed rate configuration* $(x_1^*, x_2^*, \cdots, x_n^*)$ *such that* $x_1^* = x_2^* = \cdots = x_n^*$*, and the link is fully utilized.*

When Proteus-P and Proteus-S senders compete with each other, we prove in the appendix that there exists a unique equilibrium. We leave the formal analysis of Proteus-S senders yielding bandwidth to Proteus-P senders to future work. As informal intuition, a Proteus-S sender yields to a Proteus-P sender because the RTT deviation term in the Proteus-S utility function generates larger penalty, and makes the Proteus-S sender relatively conservative.

When the primary protocol is something other than Proteus-P, the effectiveness of RTT deviation can be informally justified by § 4.2, and validated by our experiments.

Since we adopt the rate control algorithm from PCC Vivace, the above theorem and analysis deal with the existence of equilibria. We leave a study of the dynamics of convergence (e.g., convergence speed) to future work.

4.4 Proteus-H: Hybrid Mode

Network-wide utility can also benefit from applications only occasionally switching to scavenger mode. For example, when watching an online video, users may complain about rebuffering if the throughput cannot fulfill a certain bitrate, but will be satisfied once the video is played in the highest bitrate smoothly. For that purpose, we extend Proteus-P and Proteus-S into Proteus-H, a hybrid mode with a piecewise utility function constructed from Proteus-P and Proteus-S:

$$u_H(x_i) = \begin{cases} u_P(x_i) & \text{if } x_i < threshold, \\ u_S(x_i) & \text{otherwise.} \end{cases} \tag{3}$$

Effectively, Proteus-H switches between scavenger and primary modes based on a threshold. But there is no explicit switch in the control algorithm; it happens implicitly, simply by comparing utility values of different sending rates.

Intuitively, when two senders deviate from the equilibrium sending rate, they will either change towards fair share if they are in the same modes, or the sender in scavenger mode will yield, both of which drive them back to equilibrium. In an ideal situation, then, when two Proteus-H senders with switching threshold r_1 and r_2

($r_1 < r_2$) compete on a bottleneck with capacity C, we would expect them to converge towards the rate pair (x_1^*, x_2^*) where:

$$(x_1^*, x_2^*) = \begin{cases} C/2, C/2 & \text{if } C \in [0,\ 2r_1), \\ r_1, (C - r_1) & \text{if } C \in [2r_1,\ r_1 + r_2), \\ (C - r_2), r_2 & \text{if } C \in [r_1 + r_2,\ 2r_2), \\ C/2, C/2 & \text{if } C \in [2r_2,\ \infty). \end{cases}$$

Cross-Layer Design for Switching Threshold. The threshold in $u_H(x_i)$ should be set adaptively by the application. We develop a threshold policy for video streaming. We start with three observations of video bitrate adaptation:

(1) Users are oblivious to transport throughput as long as the highest video quality is rendered smoothly.
(2) The client will only request the next video chunk if there is enough space in the local playback buffer.
(3) When the video stalls upon rebuffering, the client wants as large throughput as possible to recover.

With this in mind, for bitrate adaptation, we can dynamically set the threshold to the maximum value which satisfies the following two rules:

(1) **Sufficient rate rule:** $threshold \leq G \cdot bitrate_{max}$. We set $G = 1.5$ so there is a sufficient margin of safety to avoid rebuffering.
(2) **Buffer limit rule:** $threshold \leq \frac{1}{2-f} \cdot bitrate_{current}$, where f is the (possibly fractional) number of chunks of free space in the buffer. This rule applies when $f < 2$, and is checked upon requesting a new chunk. The effect is that the threshold will decrease as the buffer approaches full (and therefore loading chunks quickly is not necessary, since anyway, the ABR algorithm will pause transmission if the buffer is full).

Then, whenever rebuffering happens, the following rule will override the switching threshold, until the video resumes.

(3) **Emergency rule:** $threshold = \infty$.

As our experiments show, when a buffer-based adaptation algorithm such as BOLA [35] is used, the above rules effectively increase network-wide efficiency. We should note that we present this as a representative solution for benchmarking; it may not be suitable for bitrate adaptation that uses throughput for control. We leave the incorporation of Proteus-H into other video streaming algorithms, and other types of applications, to future work.

5 HANDLING LATENCY NOISE

Inherent network dynamics, *e.g.*, wireless channel noise, raise challenges for latency-aware congestion control. In Proteus, a noisy utility calculation, which is based on RTT gradient and deviation, can result in incorrect rate change decisions during ramping up, and thus, capacity under-utilization.

For the purpose of noise tolerance, a fixed tolerance threshold for RTT gradient is used by PCC Vivace [17] (any RTT gradient with a smaller magnitude is ignored). But a fixed threshold is ineffective with rapid fluctuations that can occur on the Internet. We therefore design more robust mechanisms.

Per-ACK: RTT Sample Filtering. We found that in dynamic environments such as wireless networks, ACK reception can be bursty even on a non-congested link, possibly due to irregular MAC scheduling. This leads to excessive penalty from both RTT gradient and deviation, and can mislead a Proteus sender into slowing down. To mitigate this, we use the "*ACK interval*" (the time between reception of two consecutive ACKs) to filter out abnormal RTT samples, when the ratio between two consecutive ACK intervals exceeds a threshold (set to 50 in our implementation). All RTT samples are then ignored until an RTT is observed that is below the exponentially weighted moving RTT average.

Per-MI: Regression Error Tolerance. In RTT gradient calculation, the error in linear regression also reflects the accuracy of calculated RTT gradient. Specifically, for the i-th packet sent in the MI whose sent time is $sent\_time_i$ and RTT is RTT_i, we calculate its estimated regression RTT as:

$$RTT_i^* = avg(RTT_i) + rtt\_gradient \cdot (sent\_time_i - avg(sent\_time_i)),$$

where $avg(RTT_i)$ and $avg(sent\_time_i)$ are the average RTT and sent time for all acknowledged packets in the MI, respectively. Then, we calculate regression error based on the residual in linear regression:

$$regression\_error = \sqrt{\frac{1}{n} \cdot \sum_i (RTT_i - RTT_i^*)^2} \cdot \frac{1}{MI\_duration},$$

where n is the number of acknowledged packets in the MI and the final factor simply normalizes by MI duration to produce a relative error. Then, for each MI, if the calculated RTT gradient's magnitude is less than $regression\_error$, we treat both the RTT gradient and the RTT deviation as 0.

MI History: Trending Tolerance. The above per-MI error tolerance may hide a slow but persistent RTT increase, which stays within tolerance for several consecutive MIs, leading to late reaction against inflation. Since RTT deviation is ignored too, a Proteus-S sender may stop behaving as a scavenger until it sees more significant inflation. To avoid such late reaction, Proteus keeps track of latency-related metrics for a longer time period. Specifically, a sender maintains the RTT deviation and average RTT of the most recent k MIs (*e.g.*, $k = 6$ in our experiments for a reasonable trade-off between noise-vulnerability and slow responsiveness), based on which it computes two trending metrics: *trending gradient* and *trending deviation*. Specifically, using linear regression based on the stored MIs' average RTTs, trending gradient is calculated as:

$$\overline{K} = \frac{1}{k} \cdot \sum_{j=1}^{k} j, \quad \overline{RTT} = \frac{1}{k} \cdot \sum_{j=1}^{k} MI_j(RTT),$$

$$trending\_gradient = \frac{\sum_{j=1}^{k} (j - \overline{K})(MI_j(RTT) - \overline{RTT}))}{\sum_i (j - \overline{K})^2},$$

and by taking standard deviation of stored RTT deviations, trending deviation is calculated as:

$$\overline{DEV} = \frac{1}{k} \cdot \sum_{j=1}^{k} MI_j(DEV),$$

$$trending\_deviation = \sqrt{\frac{1}{k} \cdot \sum_{j=1}^{k} (MI_j(DEV) - \overline{DEV})^2}.$$

In the above expressions, $MI_j(RTT)$ and $MI_j(DEV)$ represent the j-th stored MI's average RTT and deviation, respectively.

In addition, we also maintain the exponentially weighted moving average and per-sample deviation for both trending metrics (similar to how smoothed RTT and RTT deviation are updated in the Linux kernel). Then, for each new sample of the two metrics, we compare it with the corresponding average. Our insight is that, when the calculated trending metric sample is several deviations away from its average, it is statistically unlikely to be caused by non-congested noise, and thus, cannot be ignored. We illustrate with the pseudocode below.

```
if |trending_gradient - avg_trend_grad| < G1 * dev_trend_grad :
  rtt_gradient ← 0
  if trending_deviation - avg_trend_dev < G2 * dev_trend_dev :
    rtt_deviation ← 0
```

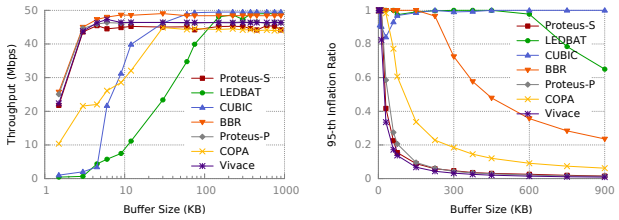
Specifically, the new RTT gradient sample will be ignored if the difference between the updated trending gradient and its moving average (*avg_trend_grad*) is smaller than G_1 times the deviation of trending gradient (*dev_trend_grad*). In that case, RTT deviation is also ignored if the difference between trending deviation and its moving average (*avg_trend_dev*) is smaller than G_2 times its deviation (*dev_trend_dev*). In our implementation, we conservatively select $G_1 = 2$ and $G_2 = 4$ to approximately achieve above 95% confidence with normally-distributed latency noise.

Control Algorithm: Majority Rule. For each rate control decision in its "probing" state, Vivace tries a pair of sending rates (in random order) twice, and changes sending rate only if they imply a consistent rate change direction [16]. That may cause slow rate ramp-up and under-utilization in highly noisy environments, where the sender sees more inconsistent rate change indicators and has to repeatedly test the same pair of sending rates before increasing rate correctly. To improve on that, we let Proteus senders try each pair of sending rates three times (instead of twice), and change the sending rate based on the majority decision from the three pairs of trials. By adding the additional pair, the sender can generally determine the direction of rate change more quickly in noisy networks, while the majority rule effectively avoids frequent false rate change direction.

Note. We do not have enough space to show how each tolerance mechanism contributes to Proteus's performance. Briefly, per-MI regression error tolerance is necessary for Proteus senders to saturate bandwidth even on relatively stable bottleneck, while trending tolerance helps enhance latency sensitivity. The RTT sample filtering mechanism and the usage of majority rule in rate control mainly benefit Proteus in highly dynamic networks, which can be demonstrated to some extent by the performance improvement of Proteus over Vivace on the live Internet (§6.2.1). However, we emphasize that the above tolerance mechanisms are heuristics, and do not have theoretical performance guarantees. Performance may still be impacted when network latency noise appears very bursty on the timescale of a MI (observed in our real-world WiFi test in §6.2.1), causing abnormal samples of RTT gradient and RTT deviation. More robust designs could employ statistical inference techniques, and taking advantage of all available information including in-network feedback [19].

6 EVALUATION

We implemented Proteus by branching off of the existing open-source UDP-based PCC implementation [17] and implementing the



(a) Throughput (b) Latency inflation

Figure 3: Bottleneck saturation with varying buffer size

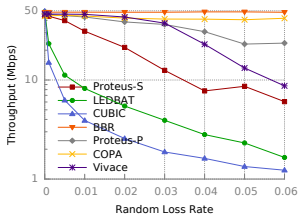


Figure 4: Loss tolerance

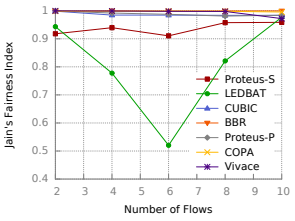


Figure 5: Fairness index

design of the previous sections.[3] For Proteus-P's utility function, we adopt the default parameters from [17]: $t = 0.9$, $b = 900$, and $c = 11.35$. For Proteus-S, we set the RTT deviation coefficient $d = 1500$ (with RTT deviation in units of *seconds*).

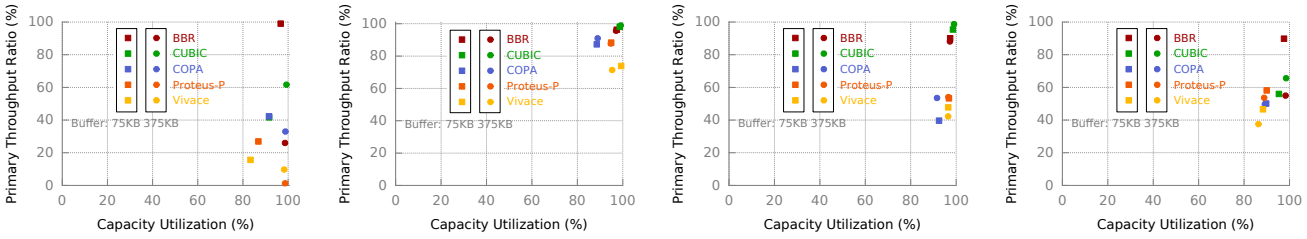
We compare two scavengers – Proteus-S and LEDBAT – and let them compete with various primary protocols: TCP CUBIC [21], BBR [13], COPA [8], PCC-Vivace [17], and Proteus-P. We employ the LEDBAT implementation in the open-source μTorrent Transport Library [2], with the target extra delay set to 100 *ms*, as in the current IETF standard [34] as well as μTorrent's default setting.[4]

To measure transport-level performance, our test environment uses Pantheon [42] to run flows and collect performance metrics, both on Emulab [38] and in the live Internet. Unless otherwise specified, we use Emulab tests with a 50 Mbps bandwidth, 30 ms RTT setup, and show the mean of at least 10 trials in each scenario. We also measure application-level performance (DASH video streaming [1] and webpage loading) to show the benefits of having scavengers competing with primary flows.

For the evaluation of Proteus-H, we implement emulated video streaming on top of our UDP implementation. Specifically, Proteus receiver runs a BOLA [35] agent that takes a DASH video definition as input and consumes the received bytes to maintain an emulated playback buffer. The receiver uses a side channel to notify the sender of: (1) its requested bitrate for each chunk, (2) when to stop/resume transmission due to limited playback buffer space, and (3) the calculated switching threshold if Proteus-H is used.

[3] At the moment, both Proteus-P and Proteus-S, as well as PCC Vivace, are based on UDT [20]. However, we adopt QUIC-compatible APIs [25] in the Proteus implementation, which should facilitate its real-world deployment.

[4] The first LEDBAT IETF draft [33] used a 25 *ms* target. We evaluate its performance as well, and get similarly undesirable results as achieved by 100 *ms*; see Appendix B.



(a) LEDBAT as Scavenger (b) Proteus-S as Scavenger (c) Proteus-P as Scavenger (d) COPA as Scavenger

Figure 6: Scavenger competes with primary protocols

6.1 Scavenger-Only Performance

When there are no primary flows, a good scavenger should have high performance like a normal congestion controller. We evaluate this single-protocol performance with typical congestion control objectives (high throughput, low latency) across different environment variables (buffer size, random loss probability, and number of competing flows).

6.1.1 Latency Awareness. We run a single flow on the above specified Emulab bottleneck link for 100 seconds, with varying buffer size. We compare the protocols' throughput and RTT inflation (Fig. 3).

As shown in Fig. 3(a), both Proteus-P and Proteus-S need as small as 4.5 KB buffer to achieve at least 90% capacity utilization, *i.e.*, 45 Mbps throughput, which is the same as needed by BBR and PCC Vivace. In comparison, both CUBIC and COPA need 5.7× larger buffer to reach the same utilization. LEDBAT, always trying to inflate the RTT by 100 *ms*, needs 150 KB buffer, which is close to the BDP (187.5 KB), and 32.3× larger than needed by Proteus.

We then evaluate latency sensitivity in terms of 95th percentile inflation ratio, calculated as:

$$95th\ inflation\ ratio = \frac{95th\ percentile\ RTT - base\ RTT}{buffer\ size / bottleneck\ bandwidth},$$

which effectively measures the 95th percentile buffer occupancy. We report this value in Fig. 3(b). Both Proteus-S and Proteus-P, similar to Vivace, limit the inflation ratio below 10% as long as the buffer is ≥ 150 KB. Even COPA, which is latency-aware, needs 3× larger buffer (600 KB) to keep inflation ratio below 10%. In comparison, LEDBAT has around 100% inflation ratio until the buffer size is large enough (at least 625 KB) to accommodate its target delay. More specifically, at 2 BDP buffer size (375 KB), Proteus-S has 75.3%, 93.8%, 96.4%, and 96.42% smaller inflation ratio compared with COPA, BBR, CUBIC, and LEDBAT, respectively.

6.1.2 Random Loss Tolerance. Next, when there exists random non-congestion loss, we compare different protocols' average throughput from multiple 100-second runs on the same bottleneck with 375 KB buffer (2 BDP) in Fig. 4. Thanks to its improved noise control, Proteus-P, using a similar utility function as Vivace, performs somewhat better than Vivace on that front, achieving 74% higher throughput with 5% random loss. Proteus-S, on the other hand, has somewhat worse throughput than Vivace, which can be attributed to its RTT deviation-based rate control which causes it to ramp up more conservatively.

LEDBAT is fragile even when facing a 0.001% random loss rate, suffering from 50% degradation compared with Proteus.

We note that COPA and BBR have higher random loss tolerance because they do not directly react to packet losses. In comparison, as explained in §4.1, the loss coefficient in Proteus and PCC Vivace's utility function is set to achieve 5% random loss tolerance. We could tune the coefficient for higher tolerance, although this induces higher congestion loss [17].

6.1.3 Fairness With Competing Flows. To evaluate convergence when multiple senders of the same protocol compete with each other, we use a 30 ms RTT bottleneck link on Emulab. We test $n \in 2, \ldots, 10$ flows with $20n$ Mbps link bandwidth and $300n$ KB buffer size. In each run, a flow is started after waiting 20 seconds for the previous flow to ramp up. We measure mean throughput of each flow during the 200 seconds after all flows are started, and present Jain's fairness index in Fig. 5. We see that Proteus-P, PCC Vivace, CUBIC, BBR and COPA all keep Jain's index around 99%. Proteus-S has lower, but still always above 90%, fairness index.

In comparison, LEDBAT's fairness decreases and then increases with n. The decreasing fairness is known as its latecomer issue [34], which occurs because after one LEDBAT flow is running, the minimum delay observation for any subsequent flow is based on an already-inflated buffer. For example, with 6 competing flows, Proteus-S is 75% more fair than LEDBAT. LEDBAT's fairness begins improving once n is large enough that the sum of the target extra delay of all flows exceeds the maximum inflation allowed by the bottleneck buffer size.

6.2 Yielding to Primary Flows

When a scavenger competes with a primary flow, our goals are that (1) most importantly, the scavenger should have minimal impact on the primary flow (compared to running the primary flow alone); and (2) secondarily, the scavenger should opportunistically use any remaining resources.

Our evaluation uses two flows, one primary followed by one scavenger. Again, we use the specified Emulab link. We consider both shallow (75 KB, *i.e.*, 0.4 BDP) and large buffer (375 KB, *i.e.*, 2 BDP) setups. In addition to LEDBAT, we test Proteus-P in the role of the scavenger, to emphasize the effectiveness of Proteus-S's RTT deviation-based utility function. We calculate two performance metrics. To measure goal (1), we use the *primary throughput ratio*, defined as the primary flow's throughput when running with the

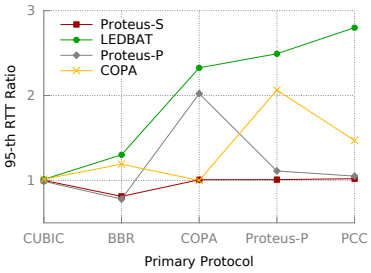


Figure 7: RTT with competition

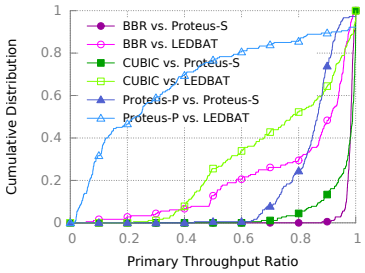


Figure 8: Throughput ratio CDF

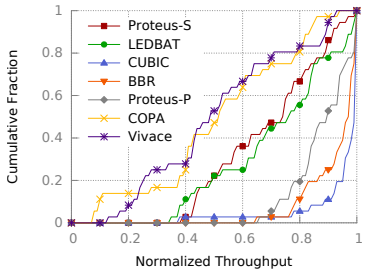


Figure 9: Single flow on WiFi

scavenger divided by its throughput when running alone. To measure goal (2), we use the the total capacity utilization of the two flows.

Although designed as a scavenger against TCP CUBIC, LEDBAT fails to yield to CUBIC when its target extra delay exceeds the maximum inflation allowed by the buffer. This occurs with both buffer setups (Fig. 6(a)). In that situation, it approximately fairly shares the bottleneck with CUBIC. LEDBAT also fails to yield when the competing sender is less aggressive, *e.g.*, it lowers BBR's throughput to 26.0% with 375 KB buffer. Similarly, the other three latency-aware protocols, COPA, PCC Vivace, and Proteus-P, are more significantly impacted by LEDBAT, *e.g.*, they all have lower than 43% throughput ratio when competing with LEDBAT.

In contrast, Proteus-S yields well (Fig. 6(b)): with CUBIC, BBR, COPA, and Proteus-P as the primary flow, the primary throughput ratio is above 98%, 95%, 87%, and 88%, respectively, in all test cases. For primary flows COPA and Proteus-P, regardless of the buffer size, the performance gains are more than 1.1× and 2.3× over LEDBAT. When competing with Vivace, Proteus-S has somewhat lower primary throughput ratio (since Vivace does not have adaptive noise tolerance, and thus may tolerate less RTT fluctuation). However, it is still at least 3.2× better than both LEDBATs.

The other two latency-aware protocols, as expected, do not consistently yield (Fig. 6(c),6(d)). Specifically, Proteus-P competes with COPA and Vivace fairly under both buffer setups, while COPA is friendly (*i.e.*, has fair equilibrium) to all the other protocols except when competing with BBR with a shallow buffer. We also observe that Proteus-P can lower the throughput of BBR to 88%, compared with at least 95% from Proteus-S. This validates our claim in Section 4.2 that RTT deviation signals competition better than RTT gradient, and hence is a better scavenger penalty metric.

Proteus-S also outperforms LEDBAT in our secondary goal of utilizing the remaining bandwidth. When competing with BBR, CUBIC, Proteus-P and PCC Vivace, Proteus-S maintains a joint capacity utilization of at least 95%. Its utilization competing with COPA is 89% (although we note this is the same utilization COPA can achieve when it competes with itself). LEDBAT only delivers around 85% utilization when competing with Proteus-P and Vivace with 75 KB buffer.

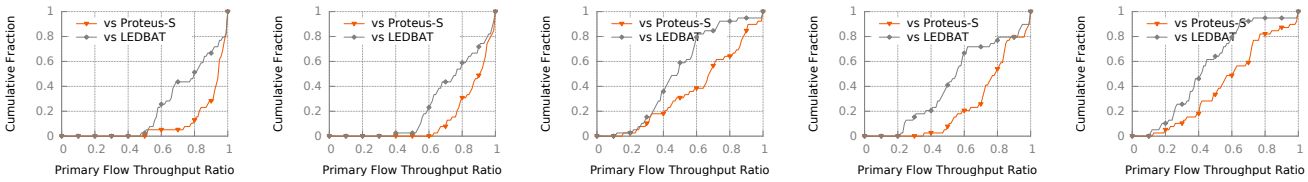
Furthermore, the competition between LEDBAT and primary protocols leads to more significant RTT inflation. Fig. 7 presents the ratio between 95th percentile RTT seen by a primary flow when competing with a scavenger flow and when the primary flow runs alone (achieved with 375 KB buffer). Latency-aware protocols see larger inflation increment, because loss-based protocols such as CUBIC already fill the buffer when they run alone. For instance, COPA sees 2.3× RTT when competing with LEDBAT. Proteus-S, unlike LEDBAT, has negligible influence on RTT, *e.g.*, BBR even sees 18.8% smaller 95-th RTT. Proteus-P and COPA are also inferior, doubling the 95-th RTT when competing with each other.

To further stress Proteus-S's robustness as a congestion control scavenger, we let it compete with BBR, CUBIC, and Proteus-P under the 180 distinct bottleneck configurations representing all combinations of the following parameters: bandwidth chosen from {20, 50, 100, 200, 300, 500} Mbps, RTT chosen from {5, 10, 30, 60, 100, 200} ms, and buffer size chosen from {0.2, 0.5, 1.0, 2.0, 5.0} BDP. For presentation clarity, we only compare Proteus-S with LEDBAT, and present the CDF of primary throughput ratios in Fig. 8. In the median case, the three primary protocols, BBR, CUBIC, and Proteus-P, achieve 7.8%, 28.0%, and 2.8× higher throughput competing with Proteus-S than with LEDBAT. That corresponds to our above conclusion, *i.e.*, the extra delay target used by LEDBAT may be too aggressive for a moderate-sized buffer, and is a late congestion signal especially against latency-sensitive protocols.

One may argue that the inferior yielding performance of LEDBAT can be improved by using a smaller extra target delay. However, as shown by the results with 25 ms extra delay in Appendix B, using a smaller extra target delay induces other performance problems, including more significant latecomer advantage and worse multiflow unfairness. Meanwhile, as a general congestion control protocol, LEDBAT still has high inflation under shallow buffers, and is more aggressive than latency-sensitive protocols such as PCC Vivace and Proteus-P.

6.2.1 Scavenger Performance on the Internet. We now move our test scenarios for the same single-flow and two-flow experiments to the live Internet. Specifically, we use WiFi connections at four different locations (two residential apartments, and two restaurants), and test using the uplink by transmitting from a laptop to an AWS server in each of 16 different regions.[5] For each source-destination pair, we conduct 4 trials, each lasting 2 minutes, and report the median value (i.e., the mean of the two middle values).[6] Finally, to ease visualization, we normalize this median throughput by the highest value obtained by any protocol on that source-destination pair.

[5]These are all of the AWS regions except Hong Kong, Bahrain, and Capetown which we were unable to use for logistical reasons.
[6]We do not observe performance issues due to interference from shared CPU.



(a) BBR as Primary (b) CUBIC as Primary (c) COPA as Primary (d) Proteus-P as Primary (e) PCC Vivace as Primary

Figure 10: Primary throughput ratio in real-world WiFi

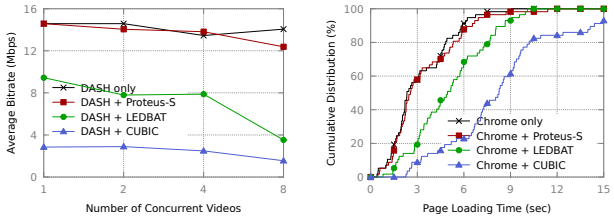
Fig. 9 shows the CDF of normalized median throughputs across the 64 source-destination pairs. The protocols intended as primary flows are, interestingly, among the worst and the best. Two latency-aware primary protocols, COPA and Vivace, have the worst performance, because they are affected by RTT fluctuations. (Even though these are WiFi links and not LTE, we observe significant fluctuation in RTT. The typical RTT deviation is up to 5 ms but RTT occasionally spikes tens of milliseconds higher.) Meanwhile, CUBIC and BBR have highest throughput because they are much more aggressive than other protocols (see latency inflation in Fig. 3(b)). CUBIC is even better than BBR because CUBIC is loss-based and the least latency-sensitive.

Among the scavengers, since LEDBAT relies on (one-way) delay, it is also somewhat prone to noisy measurement. Proteus-S is comparable to LEDBAT.

Overall, in these environments, our latency noise tolerance techniques allow Proteus-P and Proteus-S to each be among the best in their class (primary and scavenger respectively). Specifically, they each have close to the best throughput while being much more latency-aware (Fig. 3(b)) than the other high-throughput protocols. This helps show our design is generally successful in achieving a single codebase which can be either a primary or scavenger. While there might still be room for improvement (such as closing the moderate throughput gap between Proteus-P and BBR), a tradeoff between throughput and latency is to be expected.

We move now to the scavenger goal of yielding to primary flows, quantified by the CDF of primary flow throughput ratio in the same WiFi environments (Fig. 10). When competing with LEDBAT, BBR and CUBIC's median throughput ratios are 80.0% and 76.1% respectively. With Proteus-S as the scavenger, they respectively achieve 17.6% and 19.2% higher throughput ratios. Meanwhile, Proteus-S enables BBR and CUBIC to have at least 90% throughput ratio in 71.8% and 51.3% of all cases, respectively, which are 1.2× and 81.9% higher than LEDBAT. Considering BBR and CUBIC are today the most widely adopted primary protocols, and LEDBAT is perhaps the only deployed scavenger protocol on the Internet, this is an important improvement. Furthermore, Proteus-S has more significant performance gain when the competing primary protocol is latency-aware. Specifically, when competing with Proteus-S, the median throughput ratios of COPA, Proteus-P, and Vivace are 39.3%, 41.0%, and 44.1% higher than achieved when competing with LEDBAT. These results are consistent with those in Fig. 6.

6.2.2 Application Performance Benchmarks. To demonstrate the significance of a congestion control scavenger in practice, we use



(a) DASH Video (b) Webpage Load

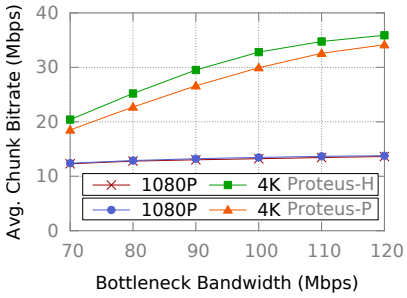
Figure 11: Scavenger with Applications on Internet

the live Internet to compare the influence of Proteus-S, LEDBAT, and CUBIC on two popular applications, DASH video streaming and webpage loading. We use a wired Xfinity downlink of about 100 Mbps. For DASH video streaming, we use the default *dash.js* (version 3.0.1) and request the *Big Buck Bunny* sample video from Akamai. For webpage loading, we randomly request the top 30 sites in United States from *Alexa.com* in a 10-minute run, with a Poisson rate of 1 request per 10 seconds. The Chrome browser is used for both applications. A single scavenger flow may run simultaneously from an AWS server in Virginia to our laptop in the background.

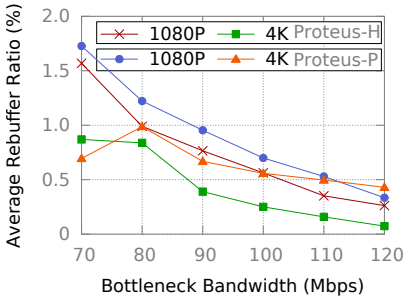
Fig. 11(a) presents the average video chunk bitrate with different number of concurrent videos (started simultaneously). The results when CUBIC runs in the background are included for reference. We can see that although LEDBAT has smaller impact to DASH performance than CUBIC, it still dramatically falls behind Proteus-S. For example, with 8 videos, Proteus-S enables DASH with 2.5× higher bitrate than LEDBAT. Then, Fig. 11(b) presents the CDF of webpage loading time. Proteus-S has almost no impact on page loading, while achieving 48.2% (median) and 33.3% (average) speed-up compared with LEDBAT. Thus, even in a single house with a single router, a congestion scavenger for background flows (*e.g.*, system update, cloud storage synchronization) can still increase competing applications' performance. (That said, we note that Proteus-S's performance gains in Fig. 11 are in part due to the fact that it is latency-aware, rather than specifically because of our scavenger mechanisms.)

6.3 Flexibility of Hybrid Utility

Next, we evaluate our novel capability of supporting a hybrid utility function. As an example application, we compare Proteus-H and Proteus-P in a video streaming test using our emulated adaptive BOLA agent. For that purpose, we generate a corpus of 10 4K and



(a) Video Bitrate



(b) Rebuffer Ratio

Figure 12: Hybrid mode in adaptive video streaming

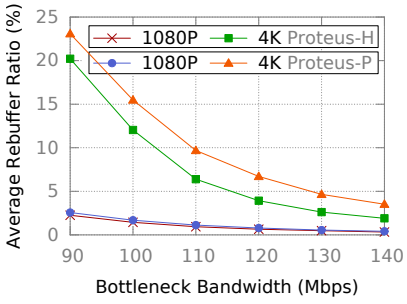


Figure 13: Consistently low rebuffer by hybrid mode

10 1080P videos, all composed of 3-second chunks and at least 3 minutes long, with highest bitrates of above 40 Mbps and 10 Mbps, respectively.

We first use an Emulab bottleneck with 30 ms RTT, 900 KB buffer and varying bandwidth. For both Proteus variants, we randomly select one 4K and three 1080P videos, start them simultaneously and let them stream for 3 minutes. Fig. 12 presents the average bitrates and rebuffer ratios of 4K and 1080P videos separately. Compared with Proteus-P, when all flows use Proteus-H, the average bitrate per 4K video chunk is increased by up to 3 Mbps and up to 11%, without obviously impacting 1080P videos. With 3-second chunk duration and 3-minute streaming period, this increment corresponds to almost half a minute longer duration staying at the highest 4K bitrate. Meanwhile, the 4K bitrate gain comes with

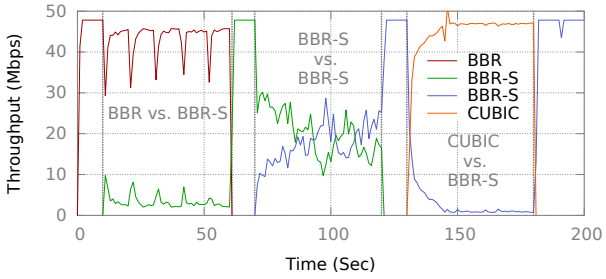


Figure 14: Extend RTT Deviation to BBR

significantly lower rebuffer ratios for both 4K and 1080P videos. For example, with 110 Mbps bandwidth, the 4K and 1080P rebuffer ratios are reduced by 68.0% and 33.5%, respectively. The only exception is under 70 Mbps bandwidth, when Proteus-P does not even try the highest 4K bitrate due to low fair share.

Considering that the above rebuffer ratios are small because of adaptation by BOLA, we force the agent at the highest bitrates, and repeat the above experiments. Fig. 13 shows the achieved rebuffer ratios, which are consistent with Fig. 12(b). Specifically, under the same 110 Mbps bandwidth, Proteus-H has 34.0% lower rebuffer ratio for 4K video. Therefore, the support for an adaptive hybrid mode can indeed increase utilization efficiency of restricted network resources.

7 DISCUSSION

We have seen that Proteus-S has robust performance against various primary protocols, and can help improve network-wide utility, both on emulated networks and in the wild. Nevertheless, much remains to explore in scavenger transport. In this section, we discuss several important insights for practical implementation and future work.

7.1 Real-World Adoption

Proteus can be implemented at the server side without involving the client, consistent with most existing transport designs [13, 16, 17]. Current Linux kernel mechanisms make it difficult (though, we think, not impossible) to implement some aspects of our utility module. But other options exist, such as the CCP platform [29], or a fully user-space transport stack as used in QUIC [25] (there is already a QUIC-based prototype of PCC Vivace [3], and we adopted QUIC-compatible APIs for our implementation of Proteus). Thus, we see several low-overhead paths to adoption for Proteus.

In more advanced deployments, client-side participation may help, including necessary messages required by applications (as in bitrate adaptation) and voluntary feedback by users (e.g., concrete deadline requirements). We leave a full design to the future, but note this feedback can be transmitted through Proteus's APIs and does not necessitate protocol changes like reserved header bits.

The basic idea behind utility design in Proteus, *i.e.*, selection of control signals, can potentially be extended to other primary protocols. For example, any protocol can lower its priority by reacting to RTT deviation. For illustration, we modify the kernel BBR implementation such that whenever the smoothed RTT deviation is larger than 20 ms, the BBR sender is forced into its minimum

RTT probing phase for at least 40 ms,[7] which is a phase when BBR effectively stops further transmission, and maintains minimum in-flight packets to probe for clean channel RTT. We let the modified BBR, denoted BBR-S, compete with BBR, CUBIC, and BBR-S on an Emulab bottleneck with 50 Mbps bandwidth, 30 ms RTT, and 375 KB buffer. Fig. 14 shows the throughput across time. Apparently, BBR-S is able to yield against BBR and CUBIC, while sharing the bottleneck fairly with BBR-S itself. This validates that our techniques may be of interest beyond Proteus itself.

7.2 Robustness in Noise Tolerance

The current Proteus architecture (as well as the utility framework in [16, 17]) relies on a relatively ideal model, *i.e.*, the utility functions and their equilibrium analysis do not formally model inherent RTT fluctuation. Although Proteus mitigates this with noise tolerance mechanisms, they are not perfect. According to results from the live Internet, the relatively more aggressive protocols (BBR and CUBIC) still have better performance as primary flows compared with Proteus-P. Additionally, there are high-fluctuation environments we have not yet tested, such as LTE. Although those networks provide better user isolation using fine-grained resource allocation [41], it is important to consider them in future work on a full-fledged transport design. Thus, we envision designs to deal with noise on a more fundamental level, as well as theoretical tools to analyze these designs. That may involve quantifying confidence in inputs to the utility function, including a specific noise term in the utility function, or turning to alternative methods like neural networks [30].

8 RELATED WORK

The surge of data intensive applications such as online video streaming has driven research on Internet congestion control in both industry and academia. Recent work [8, 13, 17] has shown that traditional TCP variants (*e.g.*, [10, 21, 27], *etc.*) cannot consistently deliver high performance. These variants, and various improvements such as Remy [40], usually correlate packet losses with congestion, which is not always the case, and thus, they cannot tolerate random packet losses. In addition, the widely used TCP CUBIC can cause significant inflation and bufferbloat, which harms the quality of experience for latency-sensitive applications. To solve TCP's inherent performance issues, numerous protocols have been introduced. The idea of performance-oriented congestion control (PCC) was proposed in [16], leveraging a sending rate control logic based on an empirical utility function constructed from observed performance metrics. However, PCC Allegro [16], the first protocol in the PCC family, uses a loss-based utility function, and also suffers from bufferbloat. PCC Vivace [17] has better latency awareness and convergence speed with its latency-aware utility function and gradient-ascent rate control. TCP BBR [13] tries to build a network model from recent measurements of network bandwidth and minimum RTT, so as to maintain high sending rate and avoid excessive inflation. COPA [8] leverages the observed minimum RTT to achieve a target rate that optimizes a natural function of throughput and delay under a Markov packet arrival model. In addition, there are works focusing on congestion control in specific Internet environments such as LTE (*e.g.*, [41, 43]), or for specific flow characteristics such as short flows (*e.g.*, [26]). However, all the above works aim at a fair-sharing equilibrium. Some works such as BBR and COPA explicitly try to achieve fair share when competing with TCP CUBIC for the purpose of TCP friendliness.

The most important work targeting a non-fair scavenger equilibrium is LEDBAT [34]. It tries to control the induced extra delay to the network within a threshold, so that it can back off when competing with other primary flows. However, as shown in our experiments, it often significantly reduces throughput of primary flows – even CUBIC, but to a larger degree for protocols that include some latency awareness – and has a known latecomer advantage issue when competing with itself [5, 14, 32].

Minerva [37] was proposed as an end-to-end transport to improve QoE fairness in video streaming. It differs from our work in that it tries to compete fairly with TCP, but is related in its deviation from fair sharing among video flows to optimize overall QoE. We believe Proteus's and Minerva's ideas could be relevant to a full-fledged cross-layer design for QoE optimization.

Our work is orthogonal to, but may utilize, platforms that ease implementation of new transport protocols, including QUIC's user-space transport which modifies the traditional HTTPS stack [25], and CCP's universal off-datapath congestion controller design [29].

A separate category of congestion control protocols improves performance via the combination of protocol changes and in-network (router/switch) support, either in data centers (e.g., [36, 39]) or in the Internet (e.g., [12, 19, 24]). While this approach can be feasible in some data centers, adoption across the public Internet is difficult.

9 CONCLUSION

We propose PCC Proteus, an architecture for Internet congestion control. Proteus supports interaction from application to transport layer, to tailor congestion control to application requirements, and in particular, to realize a congestion control scavenger. Specifically, based on an online-learning utility framework [16], we design a protocol that can act as either a primary protocol (Proteus-P) or as a scavenger (Proteus-S) using a dedicated scavenger utility function. Through comprehensive experiments on emulated networks and the live Internet, we show the robustness of Proteus-S as a scavenger against various competing protocols. We also extend Proteus to a hybrid scavenger/primary design which achieves higher application-level utility for adaptive bitrate video delivery and web page loading, and demonstrates the flexibility of our approach. We believe this line of research will be increasingly important to deal with Internet environments where constrained bandwidth must be shared between high-priority and traffic with more elastic time requirements.

ACKNOWLEDGEMENTS

We thank Praveen Balasubramanian for insightful discussions about the importance of scavenger transport. We also thank our shepherd, Mohammad Alizadeh, and SIGCOMM reviewers for their valuable comments. This research was supported by Huawei and the Israel Science Foundation.

[7] We use fixed thresholds such as 20 ms RTT deviation for illustration. That said, we are not claiming BBR-S could be a robust scavenger in practice.

REFERENCES

[1] [n.d.]. dash.js. https://github.com/Dash-Industry-Forum/dash.js.
[2] [n.d.]. μTorrent Transport Protocol library. http://github.com/bittorrent/libutp.
[3] [n.d.]. PCC QUIC Implementation. https://github.com/netarch/PCC_QUIC.
[4] [n.d.]. Proteus Implementation. https://github.com/PCCproject/PCC-Uspace.
[5] 2017. LEDBAT++: Low priority TCP Congestion Control in Windows. https://datatracker.ietf.org/meeting/100/materials/slides-100-iccrg-ledbat-low-priority-tcp-congestion-control-in-windows-01.
[6] 2018. Vivace Full Proof of Theorems. http://www.ttmeng.net/pubs/vivace_proof.pdf.
[7] 2018. Windows Transport converges on two Congestion Providers: Cubic and LEDBAT. https://techcommunity.microsoft.com/t5/Networking-Blog/Windows-Transport-converges-on-two-Congestion-Providers-Cubic/ba-p/339819.
[8] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. *Proc. of NSDI* (April 2018).
[9] J.C. Bansal, P.K. Singh, K. Deep, M. Pant, and A.K. Nagar. 2012. *Proceedings of Seventh International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2012): Volume 2.* Springer India. 435–436 pages. https://books.google.co.il/books?id=97mrtf1TlK0C
[10] L. Brakmo, S. Lawrence, S. O'Malley, and L. Peterson. 1994. TCP Vegas: New techniques for congestion detection and avoidance. *Proc. of ACM SIGCOMM* (1994).
[11] Bob Briscoe. 2007. Flow rate fairness: dismantling a religion. *Computer Communication Review* 37, 2 (2007), 63–74. https://doi.org/10.1145/1232919.1232926
[12] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and K. van der Merwe. 2005. Design and implementation of a Routing Control Platform. *Proc. of NSDI* (April 2005).
[13] N. Cardwell, Y. Cheng, C.S. Gunn, S.H. Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *Queue* 14, 5 (2016), 50.
[14] Giovanna Carofiglio, Luca Muscariello, Dario Rossi, Claudio Testa, and Silvio Valenti. 2013. Rethinking the low extra delay background transport (LEDBAT) protocol. *Computer Networks* (2013).
[15] Andrey Chernov. 2019. On Some Approaches to Find Nash Equilibrium in Concave Games. *Automation and Remote Control* 80 (05 2019), 964–988. https://doi.org/10.1134/S0005117919050138
[16] M. Dong, Qingxi Li, Doron Zarchy, Philip Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. *Proc. of NSDI* (March 2015).
[17] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. *Proc. of NSDI* (April 2018).
[18] Eyal Even-Dar, Yishay Mansour, and Uri Nadav. 2009. On the convergence of regret minimization dynamics in concave games. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, Michael Mitzenmacher (Ed.). ACM, 523–532. http://doi.acm.org/10.1145/1536414.1536486
[19] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. 2020. ABC: A Simple Explicit Congestion Controller for Wireless Networks. *Proc. of NSDI* (February 2020).
[20] Yunhong Gu. 2005. *UDT: a high performance data transport protocol.* University of Illinois at Chicago.
[21] S. Ha, I. Rhee, and L. Xu. 2008. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* (2008).
[22] Sergiu Hart and Andreu Mas-Colell. 2015. Markets, correlation, and regret-matching. *Games and Economic Behavior* 93 (2015), 42 – 58. https://doi.org/10.1016/j.geb.2015.06.009
[23] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, and M. Zhu. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM Computer Communication Review* (September 2013).
[24] D. Katabi, M. Handley, and C. Rohrs. 2002. Congestion control for high bandwidth-delay product networks. *Proc. of ACM SIGCOMM* (August 2002).
[25] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* ACM, 183–196.
[26] Q. Li, M. Dong, and P. Godfrey. 2015. Halfback: Running short flows quickly and safely. *Proc. of CoNEXT* (November 2015).
[27] Shao Liu, Tamer Başar, and Ravi Srikant. 2008. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation* (2008).
[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye (Eds.). ACM, 537–550. http://dl.acm.org/citation.cfm?id=2785956
[29] Akshay Narayan, Frank Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. 2017. The case for moving congestion control out of the datapath. *Proc. of HotNets* (December 2017).
[30] P. Brighten Godfrey Michael Schapira Nathan Jay, Noga H. Rotman and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. *Proc. of ICML* (2019).
[31] J. B. Rosen. 1965. Existence and Uniqueness of Equilibrium Points for Concave N-Person Games. *Econometrica* 33 (July 1965), 520–534.
[32] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. 2010. LEDBAT: the new BitTorrent congestion control protocol. *ICCCN* (August 2010).
[33] S. Shalunov. 2009. Low Extra Delay Background Transport (LEDBAT). Draft. https://tools.ietf.org/pdf/draft-ietf-ledbat-congestion-00.pdf
[34] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. 2012. Low Extra Delay Background Transport (LEDBAT). RFC 6817 (Experimental). http://www.ietf.org/rfc/rfc6817.txt
[35] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. 2016. BOLA: Near-optimal bitrate adaptation for online videos. *Proc. IEEE INFOCOM* (April 2016).
[36] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. 2012. Deadline-aware datacenter tcp (d2tcp). *Proc. of ACM SIGCOMM* (August 2012).
[37] Ravichandra Addanki Mehrdad Khani Prateesh Goyal Vikram Nathan, Vibhaalakshmi Sivaraman and Mohammad Alizadeh. 2019. End-to-end transport for video QoE fairness. *Proc. of ACM SIGCOMM* (August 2019).
[38] B. White, J. Lepreau, L. Stoller, R. Ricci, G. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. 2002. An integrated experimental environment for distributed systems and networks. *Proc. of OSDI* (December 2002).
[39] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. *Proc. of ACM SIGCOMM* (August 2011).
[40] K. Winstein and H. Balakrishnan. 2013. TCP ex Machina: Computer-Generated Congestion Control. *Proc. of ACM SIGCOMM* (August 2013).
[41] K. Winstein, A. Sivaraman, and H. Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. *Proc. of NSDI* (March 2013).
[42] F. Y. Yan, J. Ma, G. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein. 2018. Pantheon: the training ground for Internet congestion-control research.
[43] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. 2015. Adaptive congestion control for unpredictable cellular networks. *Proc. of ACM SIGCOMM* (August 2015).

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A PROTEUS EQUILIBRIUM ANALYSIS

We consider a simple model of interaction between senders on a bottleneck link. We show below that for any combination of Proteus-P and Proteus-S senders competing over a single link the induced equilibrium is unique. We leverage this to establish that when only Proteus-P senders, or only Proteus-S senders, share the link, the resulting outcome is fair. We leave the study of dynamics of congestion control in more realistic models (e.g., that incorporate stochastic packet arrivals), and of the impact of different parameter configuration on equilibria when Proteus-P and Proteus-P senders compete, to future research.

A.1 Notation

We let x_i denote the sending rate of Proteus sender i, C the bottleneck capacity, and S the total sending rate of all the senders competing over a common bottleneck ($S = \sum_i x_i$).

As in the proofs for fairness and convergence of PCC-Vivace [17] (assuming tail drop queue), when the buffer is not empty, the RTT gradient is captured by the expression

$$\frac{d(RTT_i)}{dt} = \frac{S - C}{C}.$$

Proteus-S leverages RTT deviation as a signal for competition with primary (Proteus-P) senders. Thus, the interesting scenario to consider for the proof is when the buffer is deep enough for RTT

deviations to be observed and the equilibrium resulted from the interaction between different Proteus senders (P and/or S) does not involve packet loss. Thus, to simplify exposition, we disregard the loss terms in Proteus-P and Proteus-S utility functions in the analysis below. Our formal arguments can be extended to incorporate penalties for loss using the arguments in [6].

Proteus-P is hence expressed in the following form (with loss term omitted):

$$u_P(x_i) = x_i^t - b \cdot x_i \cdot max\left\{0, \frac{S-C}{C}\right\} \quad .$$

We also derive the theoretical (simplified) representation of the scavenger utility function. Note that in our theoretical model, RTT deviation is induced by senders building or draining the buffer. In that process, the difference between two consecutive RTT samples observed by a Proteus-S sender i (assuming MTU-sized data packets) is

$$\Delta_{RTT}(x_i) = \frac{MTU}{x_i} \cdot \frac{d(RTT_i)}{dt}.$$

Thus, RTTs exhibit arithmetic progression when the global sending rate configuration is fixed, and sender i's observed RTT (standard) deviation at an MI takes the form

$$\begin{aligned}\sigma(RTT_i) &= \sqrt{\frac{2 \cdot \sum_{k=1}^{\lfloor n_i/2 \rfloor} (k \cdot \Delta_{RTT}(x_i))^2}{n_i}} \\ &= \sqrt{\frac{(n_i+1)(n_i-1)}{12}} \cdot \frac{MTU}{x_i} \cdot \left|\frac{d(RTT_i)}{dt}\right| \quad ,\end{aligned}$$

where n_i is the number of RTT samples in the corresponding MI.[8] The Proteus-S utility function (again, with the loss term omitted) can now be expressed as

$$\begin{aligned}u_S(x_i) &= u_P(x_i) - d \cdot x_i \cdot \sigma(RTT_i) \\ &= x_i^t - b \cdot x_i \cdot \max\{0, \frac{S-C}{C}\} - d \cdot A \cdot x_i \cdot \frac{|S-C|}{C} \quad ,\end{aligned}$$

where $A = \frac{MTU}{x_i} \cdot \sqrt{(n_i+1)(n_i-1)/12}$.

Since Proteus, similar to [16, 17, 30], employs an RTT-long monitor interval, n_i can be regarded as approximately linear in x_i. Consequently, A can be regarded as a constant for the purpose of our analysis.

A.2 Existence and Uniqueness of Equilibrium

We consider $n \geq 0$ Proteus-P senders and $m \geq 0$ Proteus-S senders competing over the same bottleneck link. We formulate the interaction between these senders as a non-cooperative game G, in which the senders are the players, the strategy of each player is its choice of sending rate, and the payoff for each player from a combination of strategies (global configuration of sending rates) is as specified by its (Proteus-P or Proteus-S) utility function. We prove below that game G has a unique equilibrium point. Our proof consists of the following parts:

(1) We first make the simple observation that, in *any* equilibrium, the total sending rate across all senders can be no less than the capacity, *i.e.*, $S \geq C$ in all equilibria.
(2) We then focus on the *subgame* $G_{S \geq C}$, which is derived from G by only permitting combinations of senders' strategies for which $S \geq C$. We prove that the global rate-configuration is an equilibrium of $G_{S \geq C}$ if an only if it is an equilibrium of G. Thus, we can restrict our attention to analyzing the equilibria of $G_{S \geq C}$.
(3) Lastly, we prove that the restricted game $G_{S \geq C}$ falls within the game-theoretic category of strictly socially concave games [15, 22], for which a unique equilibrium is guaranteed to exist [17, 18, 31]. We then conclude that the original game G is also guaranteed to have a unique equilibrium.

Observation: In any Nash equilibrium of the game G, $S \geq C$.

To see why this is so, suppose, for point of contradiction, the existence of an equilibrium such that $S < C$. Let $\epsilon = C - S$. Since in this equilibrium the total sending rate is strictly less than the link capacity, any of the senders can strictly improve its utility by increasing its rate by less than ϵ (attaining better throughput without increasing the RTT while still satisfying $S < C$)—a contradiction to this being an equilibrium.

We next prove the following lemma.

Lemma A.1. *A configuration of sending rates is an equilibrium of G* ***if and only if*** *it is an equilibrium of $G_{S \geq C}$.*

Proof. Since in any equilibrium of G, $S \geq C$ (see the above observation), and since G allows a strict superset of the strategies available in $G_{S \geq C}$, any equilibrium in G is an equilibrium also in $G_{S \geq C}$.

Now, consider a configuration of sending rates x^* which is an equilibrium in $G_{S \geq C}$. We first handle the case that $S > C$. Let $\epsilon = S - C$. Suppose, for point of contradiction, that x^* is not an equilibrium in G. Hence, some sender i can increase its utility by increasing or decreasing its rate by a certain amount δ. As x^* is an equilibrium in $G_{S \geq C}$, the rate change δ must be outside the strategy space available in $G_{S \geq C}$, i.e., it is a rate decrease where $\delta > \epsilon$. However, since the Proteus (P and S) utility functions are continuous, i must be able to improve its utility also by decreasing its rate by less than ϵ. Observe, however, that this contradicts x^* being an equilibrium in $G_{S \geq C}$ (since the rate-configuration reached after i's rate change is also in $G_{S \geq C}$).

Next, consider the case that x^* is an equilibrium in $G_{S \geq C}$ for which $S = C$. Since this is an equilibrium in $G_{S \geq C}$, no sender can increase its utility by increasing its rate. In addition, no sender can increase its utility by decreasing its rate, because all terms in its utility would decrease or remain the same: sending rate would decrease, the latency gradient term would remain at zero since Proteus (P and S) utility functions do not reward negative latency gradients, and RTT deviation penalty would either remain at zero or become negative. Therefore, no sender can improve its utility and x^* is also an equilibrium in G. □

Lemma A.2. *$G_{S \geq C}$ is strictly socially concave.*

Proof. With $S \geq C$, the Proteus-S and Proteus-P utility functions take the following form:

$$u_S(x_i) = x_i^t - (b + d \cdot A) \cdot x_i \left(\frac{S-C}{C}\right) \quad ,$$

[8] Without loss of generality, we show the expression when n_i is an odd number.

$$u_P(x_i) = x_i^t - b \cdot x_i \left(\frac{S-C}{C}\right) \quad .$$

To prove that $G_{S\geq C}$ is strictly socially concave we show that the following three conditions are satisfied. See [18] for an exposition of (strictly) socially concave games.

(1) Each individual sender i's utility function is strictly concave in its sending rate x_i.
(2) Each individual sender i's utility function is convex in the *other* senders' rates $x_{-i} = \sum_{j\neq i} x_j$.
(3) The sum of sender utilities $\sum_{i\in P_S \bigcup P_P} u(x_i)$, where P_S and P_P are all scavenger and all primary senders, respectively, is concave in the combination of all senders' rates X.

We first show that the utility function $u_S(x_i)$ of Proteus-S sender i is concave in x_i. The first derivative of $u_S(x_i)$ is

$$\frac{\partial u_S(x_i)}{\partial x_i} = t \cdot x_i^{t-1} - (b + d \cdot A)\left(\frac{S-C}{C} + \frac{x_i}{C}\right) \quad .$$

Its second derivative is

$$\frac{\partial^2 u_S(x_i)}{\partial (x_i)^2} = t(t-1)x_i^{t-2} - \frac{2}{C}(b + d \cdot A) \quad .$$

Since $0 < t < 1$ this second derivative is negative, so $u_S(x_i)$ is concave in x_i.

Then, $u_P(x_i)$'s concavity in x_i follows from the fact that when $S \geq C$, $u_P(x_i)$ is identical to PCC Vivace's utility function [17], already shown to be concave in x_i [6].

The utility function of each sender i, whether using Proteus-P or Proteus-S, is convex in x_{-i}, as derived from the fact that $\frac{\partial u_S^2(x_i)}{(\partial x_{-i})^2} = 0$ and $\frac{\partial u_P^2(x_i)}{(\partial x_{-i})^2} = 0$.

Last, we show that the function $g(X) = \sum_{i\in P_S \bigcup P_P} u(x_i)$ is concave in the combination of all senders' rates X:

$$\begin{aligned} g(X) &:= \sum_{i\in P_S} ((x_i)^t - (b + d \cdot A) \cdot x_i(\frac{S-C}{C})) \\ &\quad + \sum_{i\in P_p} ((x_i)^t - b \cdot x_i(\frac{S-C}{C})) \\ &= \sum_{i\in P_S \bigcup P_P} ((x_i)^t) - (S \cdot b + T \cdot d \cdot A)(\frac{S-C}{C})) \quad , \end{aligned}$$

where T is the total sending rate for Proteus-S senders, i.e., $T = \sum_{i\in P_S}(x_i)$.

On that basis, $g(X)$'s first derivative with respect to a Proteus-S sender i's rate x_i is

$$\frac{\partial g(X)}{\partial x_i} = t \cdot x_i^{t-1} - b(\frac{S-C}{C} + \frac{S}{C}) - (d \cdot A)(\frac{S-C}{C} + \frac{T}{C}) \quad ,$$

and the second derivative with respect to the same sender i is

$$\frac{\partial^2 g(X)}{(\partial x_i)^2} = t(t-1)x_i^{t-2} - 2\frac{b + d \cdot A}{C} < 0 \quad .$$

Besides, the second derivative with respect to another Proteus-S sender j is

$$\frac{\partial^2 g_i(X)}{\partial x_i \partial x_j} = -2 \cdot \frac{b + d \cdot A}{C} < 0 \quad ,$$

and the second derivative with respect to a Proteus-P sender j is

$$\frac{\partial^2 g_i(X)}{\partial x_i \partial x_j} = -\frac{2b + d \cdot A}{C} < 0 \quad .$$

The second derivatives of $g(X)$ when the first derivatives are with respect to Proteus-P senders can similarly be shown to be negative. Since all second derivatives are negative, the Hessian is negative semidefinite and so we conclude that $g(X)$ is concave in X [9].

We have shown that the three conditions are satisfied and so $G_{S\geq C}$ is strictly socially concave. □

Strictly socially concave games have a unique equilibrium [17, 18, 31]. This, combined with Lemma A.1, implies that G has a unique equilibrium.

The uniqueness of equilibria immediately implies the fairness in symmetric case as in Theorem 4.1&4.2:

Theorem 4.1 When only Proteus-P senders compete over a bottleneck link the unique equilibrium is fair.

Theorem 4.2 When only Proteus-S senders compete over a bottleneck link the unique equilibrium is fair.

These two theorems follow from the fact that if some sender i's rate in equilibrium x_i is different than another sender j's rate x_j, then the global rate configuration in which i sends at rate x_j and j sends at rate x_i must be a *different* equilibrium. This, however, contradicts the uniqueness of the equilibrium.

B TUNING TARGET EXTRA DELAY CANNOT SAVE LEDBAT

When LEDBAT was first proposed as an IETF draft [33], it employed an extra delay target of 25 *ms*, which is much smaller than 100 *ms* today. Based on our analysis in §4.2, using 25 *ms* extra delay as target should be an earlier congestion signal than 100 *ms*. However, using similar sets of experiments in §6, we demonstrate that both setups fail to serve as robust scavenger against the evaluated primary protocols (LEDBAT-25 and LEDBAT-100 are used to distinguish between two setups).

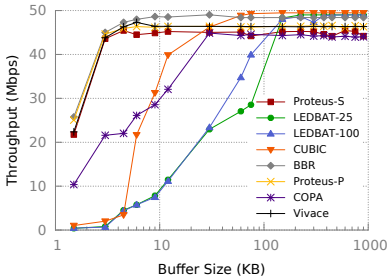
B.1 Performance Goal

First, as a congestion controller itself, LEDBAT-25 also needs large buffer to achieve high utilization. In the meanwhile, it keeps the buffer full until the buffer is large enough to accommodate 25 *ms* additional delay.

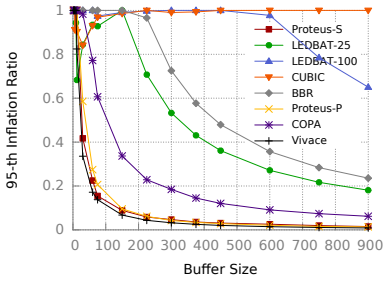
Using the same Emulab bottleneck as in Fig. 3, we have the following updated Fig. 15, where LEDBAT-25 and LEDBAT-100 have similar performance.

Similarly, provided 2 BDP buffer under an Emulab bottleneck of 50 Mbps bandwidth and 30 ms RTT, Fig. 16 shows that LEDBAT-25 has almost identical performance when there exists random loss. This is because they both inherit the design of traditional TCP, *i.e.*, correlating packet losses with in-network congestion.

Furthermore, since with a smaller target extra delay, LEDBAT-25 has even worse multi-flow fairness, because a specific buffer can now accommodate the sum of delay targets of more LEDBAT-25 senders. To validate that, we repeat the multiflow competition experiment as in Fig. 5 with LEDBAT-25. As expected, in Fig.17, LEDBAT-25's fairness index is lower than LEDBAT-100. With n = 10, the Jain's index of LEDBAT-25 is 38.7% smaller than Proteus-S.



(a) Throughput



(b) Latency inflation

Figure 15: Bottleneck saturation with varying buffer size

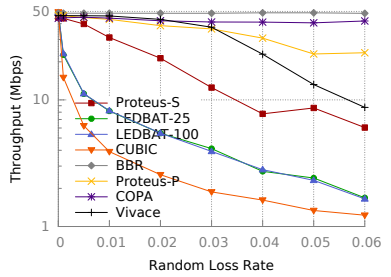


Figure 16: Random loss tolerance

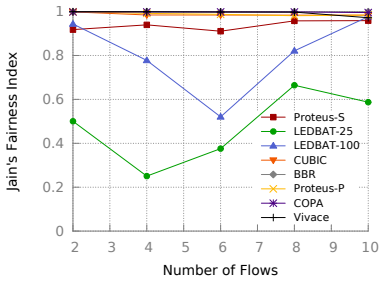


Figure 17: Fairness with competing flows

To further demonstrate the issue intuitively, we show the throughput across time with $n = 4$ in Fig. 18. For LEDBAT-25, each new flow dominates all previous flows because it observes larger "minimum"

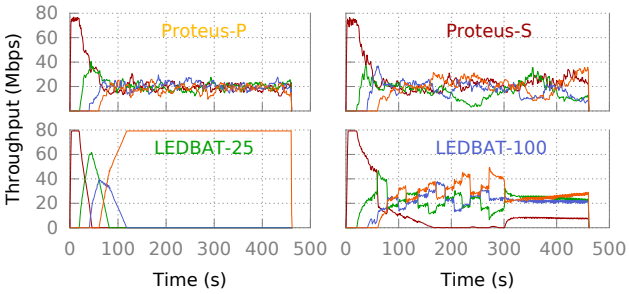


Figure 18: 4-Flow Competition

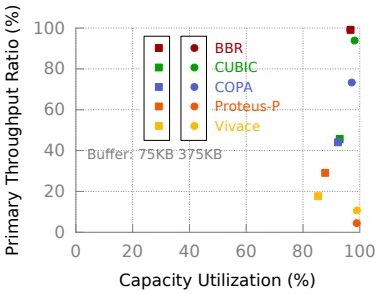


Figure 19: LEDBAT-25 as a Scavenger Competing with Primary Protocols

one-way delay. In the end, the last flow will grab almost all the bandwidth. LEDBAT-100 has better fairness than LEDBAT-25, but the first flow still has the lowest bandwidth share. Both Proteus variants are much more stable and fair, with Proteus-S fluctuating somewhat more than Proteus-P because Proteus-S senders back off (and then recover) in competition more conservatively and frequently.

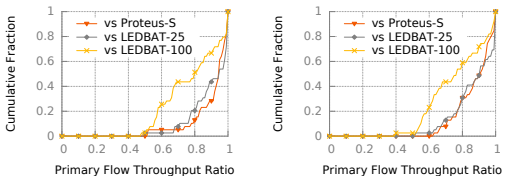
B.2 Yielding Goal

Second, LEDBAT-25, though less aggressive than LEDBAT-100, is still not robust enough as a scavenger against many primary protocols, especially recently-proposed latency-sensitive protocols. To show with, we use the same 50 Mbps bandwidth, 30 ms RTT bottleneck with two buffer setups, and conduct the two-flow competition experiment, letting LEDBAT-25 compete with BBR, CUBIC, COPA, Proteus-P, and PCC.

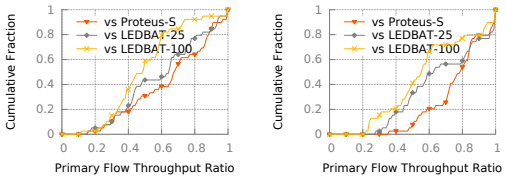
The following is the performance summary for LEDBAT in Fig. 19.

- LEDBAT-25 fails to yield to CUBIC with 75 KB buffer.
- Regardless of the buffer size, compared with LEDBAT-25, the performance of Proteus-S (Fig. 6(b)) is 24% higher when the primary protocol is COPA and 2× higher when the primary protocol is Proteus-P.
- Similar to LEDBAT-100, LEDBAT-25 is even more aggressive against PCC Vivace and Proteus-P.

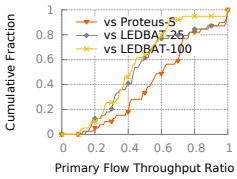
Fig. 20 shows the impact of LEDBAT-25 on RTT of primary protocols. Although COPA can still achieve 73.3% throughput ratio competing with LEDBAT-25 (as in Fig. 6(d)), that comes at the cost of 2.2× RTT.



(a) BBR as Primary (b) CUBIC as Primary



(c) COPA as Primary (d) Proteus-P as Primary



(e) PCC Vivace as Primary

Figure 22: Primary Throughput Ratio in Real-World WiFi

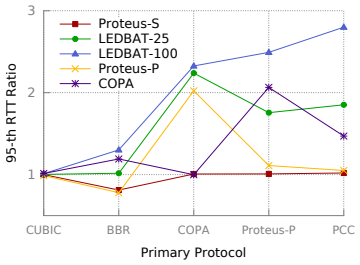


Figure 20: Scavenger's Impact on Congestion RTT (including LEDBAT-25)

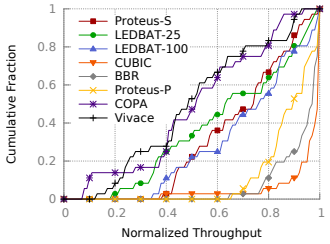


Figure 21: Single Flow Throughput on WiFi (include LEDBAT-25)

B.2.1 WiFi Performance. With a smaller target extra delay, LEDBAT-25 unfortunately is also more sensitive to RTT noise. Fig. 21 shows LEDBAT-25 among our tests of single-flow throughput on real-world WiFi (our laptop to AWS servers). Its throughput CDF is worse than LEDBAT-100 and Proteus-S.

When acting as scavenger sender on the same WiFi test configurations, LEDBAT-25, as expected, is better than LEDBAT-100, but still falls behind Proteus-S, as shown in Figure 22. Specifically, when competing with Proteus-S, the median throughput ratios of COPA, Proteus-P, and PCC Vivace are respectively 5.2%, 24.7%, and 38.6% higher than what they achieve when competing with LEDBAT-25.

B.3 Summary

The key reason that LEDBAT-25 still cannot be a robust scavenger is that LEDBAT uses a late signal for flow competition. Therefore, it is easy for it to have much higher aggressiveness than most latency-aware protocols, such as COPA, PCC Vivace, and Proteus-P.