TCP Congestion Control Beyond Bandwidth-Delay Product for Mobile Cellular Networks

Paper #66

ABSTRACT

TCP does not work well in modern cellular networks because the current congestion-window (*cwnd*) based congestion control mechanism intimately couples congestion control and packet dispatch, which provides TCP with only indirect control of the effective data rate. The throughput degradation arising from the *cwnd*-based mechanism is especially serious when the uplink is congested. We describe *PropRate*, a new rate-based TCP algorithm that directly regulates the packets in the bottleneck buffer to achieve a trade-off in terms of delay and throughput along a more efficient frontier than conventional *cwnd*-based TCP variants. To the best of our knowledge, PropRate is the first TCP algorithm that allows an application to set and achieve a target latency, if the network conditions allow for it. Also, unlike the *cwnd*-based TCP mechanism, our new rate-based TCP mechanism is significantly more resilient to saturated uplinks in cellular networks. PropRate does not require modifications at the receiver and is amenable to immediate practical deployment.

Introduction

Mobile cellular networks have become a dominant mode of Internet access due to the increasing popularity of mobile devices and improvements in cellular technology. Global mobile data traffic was reported to have grown by 63% in 2016, and this growth is expected to continue [1]. End-to-end latency is often the dominant component of the overall response time for real-time communication (RTC) applications, such as video conferencing and online gaming, which are more sensitive to latency than throughput [18]. There is significant interest in improving latency for transport protocols [7, 23, 22, 6, 27], with Skype and Google developing their own transport layer protocols to address high latency [5, 11].

The design of the traditional window-based (*cwnd*-based) mechanism is fundamentally incompatible with the features of modern mobile cellular networks. Traditional *cwnd*-based TCP suffers from the bufferbloat problem because it is designed to saturate the buffer to fully utilize the available bandwidth and to detect congestion [7]. This design works well in wired networks where routers have small buffers to quickly signal network congestion to the sender. However, mobile cellular networks use large buffers to buffer the underlying variations in the link conditions to ensure high link utilization. Because the buffers are large relative to the available bandwidth, saturation results in large latencies [25].

We propose to replace traditional loss-based congestion signaling with buffer-delay-based congestion detection and incorporate an oscillation feedback loop that keeps the sending rate around the estimated bottleneck bandwidth. We found that it is sometimes impossible to achieve low latency if the packets in flight are kept at the bandwidth delay product (BDP). In fact, to achieve low latency, *it might sometimes be necessary to empty the buffer*. Existing latency-optimized algorithms implicitly do this, but none of them explicitly seek to do so. As a proof-of-concept, we developed *PropRate*, a rate-based congestion control protocol. We also observed that a simple feedback loop is not always sufficient to regulate the latency accurately. We address this problem by introducing an additional higher-level negative-feedback loop. To the best of our knowledge, PropRate is the first TCP algorithm that allows an application to set and successfully achieve a target latency, if the network conditions allow for it.

PropRate uses the buffer delay as the congestion signal, and performs rate control by having the sending rate oscillate around the estimated receive rate. In particular, the sending rate is set to be proportional to the estimated receive rate. By doing so, the operating parameters can be determined solely by measuring the one-way delay. Oscillation is introduced as a means to probe for spare network capacity to utilize extra link bandwidth when it becomes available, and to give up some bandwidth when there is network congestion. Unlike most TCP variants, which have a fixed operating point, PropRate is a tunable protocol that can achieve a range of performance trade-offs between delay and throughput.

We compared the performance of PropRate to the publicly-available implementations of the state-of-the-art algorithms, like BBR [4], LEDBAT [19], Sprout [22], PCC [6] and Verus [27], and to traditional TCP variants implemented in the Linux kernel, using trace-driven emulation, on actual 4G/LTE cellular networks and on traditional wired networks. Our results show that PropRate can outperform the state-of-the-art algorithms on mobile cellular networks. PropRate can be tuned to achieve low delays similar to Sprout and PCC, but at higher throughputs with lower overhead. We can also match the throughput for TCP CUBIC and BBR, while keeping the average latency low in our LTE traces.

Our key contribution is that we have shown that a buffer-management-based approach to congestion control is a viable and effective alternative for existing mechanisms for mobile cellular networks, and potentially for traditional wired networks. We showed that there is no compelling reason to predict the future [22, 23] or to do link profiling [6, 27] for mobile cellular networks. It is sufficient to have a control loop that can react sufficiently fast to the changing network conditions. However, our work also suggests that there is still significant room for innovation in the design of the TCP control and feedback loop for mobile cellular networks.

PropRate is implemented as a Linux kernel module and requires modifications only to the TCP sender. It is compatible with existing TCP receivers, making it suitable for immediate deployment on servers and middleboxes commonly deployed in the wild.

Related Work

The body of work related to TCP congestion control is vast and it is not possible to do justice to them within this paper. Thus, we will describe only the works that are most closely related to PropRate.

BBR [4] is the state-of-the-art congestion control protocol that is most similar to PropRate. However, its design philosophy is very different. Cardwell et al. were working on the assumption that events such as loss or buffer occupancy are only weakly correlated with congestion and proposed a control algorithm based on two operating parameters: (i) estimated bandwidth and (ii) roundtrip time. BBR attempts to operate at an operating point that optimizes for throughput, while minimizing latency as the secondary objective. We believe that it is important to not solely maximize throughput, but also to consider the trade-off between throughput and latency in a congestion control algorithm. PropRate achieves this by directly regulating the filling and draining of the bottleneck buffer in a link. This has resulted in two key differences in our feedback mechanism design: (i) we consider the one-way delay instead of RTT as it presents a more accurate picture of latency, and (ii) we relax the constraint that the buffer should not be empty. As a result, PropRate can achieve a range of trade-offs between latency and throughput instead of being limited to a single operating point.

This particular flexibility has shown itself to be attractive in recent times as apparent from the numerous attempts to improve the latency of mobile cellular networks, mostly by attempting to forecast network performance [23, 22, 27]. Xu et al. conducted extensive experiments to show that network conditions within a small time window are correlated, and used a regression tree to predict the available bandwidth to set the sending rate [23]. Winstein et al. forecasted the number of packets that can be sent in the next window to bound the latency and achieve high throughput [22]. Zaki et al. used the link history to map the desired latency to the appropriate congestion window [27]. While these techniques are generally effective in improving the network latency, they incur significant CPU overhead.

Others have tried to improve performance by defining a utility function of some network metrics, such as one-way delay, loss rate and throughput, and adjusting *cwnd* to find a value that maximizes it [21, 6]. For a given utility function, Remy trains the model for common operating conditions [21], while PCC adjusts the sending rate dynamically [6]. We show in §5 that these approaches are not as effective in practice as BBR or PropRate.

Active Queue Management (AQM) schemes like CoDel are designed for routers that attempt to address the bufferbloat problem [17]. The drawback of these schemes is that they require modifications to intermediate routers. We decided that our algorithm should be end-to-end and require only modifications to the sender to facilitate practical deployment. That said, our solution could be interpreted as an end-to-end implementation of CoDel within the transport layer, which exposes parameters that can be directly controlled by the application.

Besides PropRate and BBR, there have been many rate-based TCP algorithms in the literature [3, 9, 13, 12, 19], but most are not true rate-based algorithms because they still send packets based on a congestion window (*cwnd*) that is clocked by returning ACK packets. Such algorithms suffer from two main drawbacks: (i) large latencies due to bufferbloat [7], where congestion is triggered only by a loss event which happens when the buffer overflows, and (ii) ACK delays from asymmetric channels [15] since new packets are only sent when an ACK packet is received.

To the best of our knowledge, WTCP [20] was the first true rate-based mechanism that also decouples the sending of packets from received ACKs. WTCP was designed for CPDP networks (an early predecessor of modern cellular networks), and there are significant differences in approach and implementation from PropRate. For example, WTCP uses packet loss rates to signal congestion, packet-pairs to estimate bandwidth and ignores RTO due to unreliable RTTs. PropRate, on the other hand, uses buffer delay as a congestion signal, packet trains to estimate bandwidth and retains the RTO since the RTT is kept stable by maintaining low buffer occupancy. RRE [15] adopts a similar rate regulation mechanism as PropRate, but it was designed to mitigate uplink ACK delays in cellular networks by eliminating ACK clocking and did not have a buffer management model like that for PropRate, as described in §3. RRE attempts to keep the buffer full to achieve high link utilization and throughput, so it suffers from high latency.

Algorithm Design

While TCP congestion control is typically analyzed in terms of the bandwidth delay product, our key insight is that the critical metric is not the roundtrip time but the one-way delay from the sender to the receiver. In cellular networks, it is possible for the forward and return paths to be asymmetric. We thus consider a simplified model of the pipe: (i) there is an abstract bottleneck buffer somewhere along the pipe that restricts the data to a rate ρ, which means that the measured receive rate at the receiver would also be ρ; and (ii) we are able to measure changes in the one-way delay from the sender to the receiver with reasonable accuracy after some delay of approximately one RTT.

We observe that we can achieve full link utilization by sending packets at a rate that matches the available bandwidth. However, simply matching the sending rate to the receive rate is not sufficient. Doing so naively will allow us to detect a drop in the available bandwidth, but we will not be able to tell if the available bandwidth has increased. Also, the receive rate is difficult to estimate with high accuracy because the available bandwidth can vary by over an order of magnitude in a period of several minutes [25].

Our key idea is to continuously probe the available network capacity by having the sending rate oscillate around the estimated receive rate by filling and draining the bottleneck buffer. We send at a rate σ_f, which is faster than the receive rate ρ when we want to fill the buffer (*Buffer Fill*), and send at a slower rate σ_d when we want to drain the buffer (*Buffer Drain*). To decide whether to fill or drain the buffer, we estimate the instantaneous buffer delay t_{buff} from the observed one-way delay, and switch between filling and draining when the one-way delay reaches a threshold T. The details on how we achieve this in practice are described in §4.

In PropRate, we set $\sigma_f = k_f \rho$ and $\sigma_d = k_d \rho$, i.e., the sending rate in the Buffer Fill and Buffer Drain states are proportional to the receive rate ρ. In Figures 1 and 2, we illustrate two possible cases for how the instantaneous buffer delay t_{buff} might vary over time: a buffer full case where the buffer is never emptied and a buffer emptied case where the buffer is periodically emptied, respectively. While it might seem surprising that we switch between the two states at the same threshold T, the reason why this works is because we are not switching based on the instantaneous buffer delay, but based on the observed buffer delay after a delay of $RTT + t_{buff}$. The hysteresis in the system creates the sawtooth pattern.

Clearly, if we want to fully utilize the available bandwidth, the buffer should never be allowed to be emptied at any time. However, not allowing the buffer to fully drain will impose a limit on how low the latency can go. If we need to achieve really low latencies, we might need to operate in the latter regime where the buffer is periodically emptied. We denote the utilization $\mathbb{U}$ of the link with:

$$\mathbb{U} = \frac{t_f + t_d}{t_f + t_d + t_e} \tag{1}$$

where t_f is the time spent in the *Buffer Fill* state. The time spent in the *Buffer Drain* state is divided into t_d, the time needed to drain the buffer, and t_e, the time during which the buffer is empty. If $t_e = 0$, the buffer is never empty and $\mathbb{U} = 1$. Furthermore, the average buffer delay $\bar{t}_{buff}$ is the area of the shaded parts over one period, i.e.,

$$\bar{t}_{buff} = \begin{cases} \frac{D_{max}+D_{min}}{2}, & \text{Buffer Full Case} \\ \frac{D_{max}}{2}\mathbb{U}, & \text{Buffer Emptied Case} \end{cases} \tag{2}$$

D_{max}, D_{min} and $\mathbb{U}$ are essentially determined by the buffer delay threshold T and the parameters k_f and k_d for the *Buffer Fill* and *Buffer Drain* states, respectively. Thus, we need to understand how these parameters affect the shape of the waveforms. More precisely, k_f determines the gradient of the rising slope as well as the height of the sawtooth wave (D_{max}). Likewise, k_d determines the gradient of the descending slope as well as the depth of the trough of the wave (D_{min}).

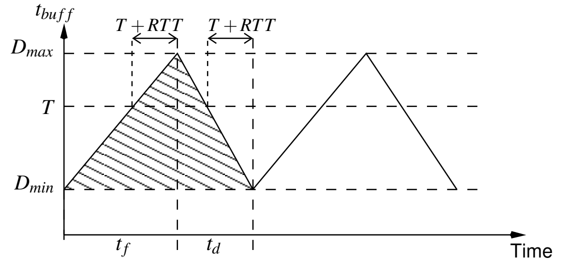


Figure 1: Buffer full (throughout optimal) case: buffer delay oscillates between D_{max} and D_{min}.

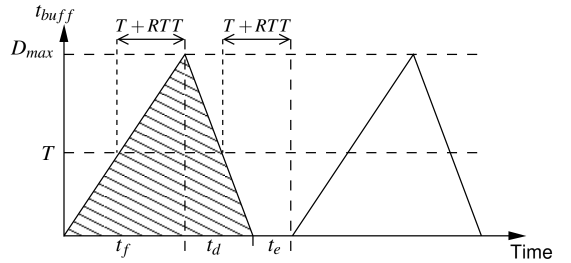
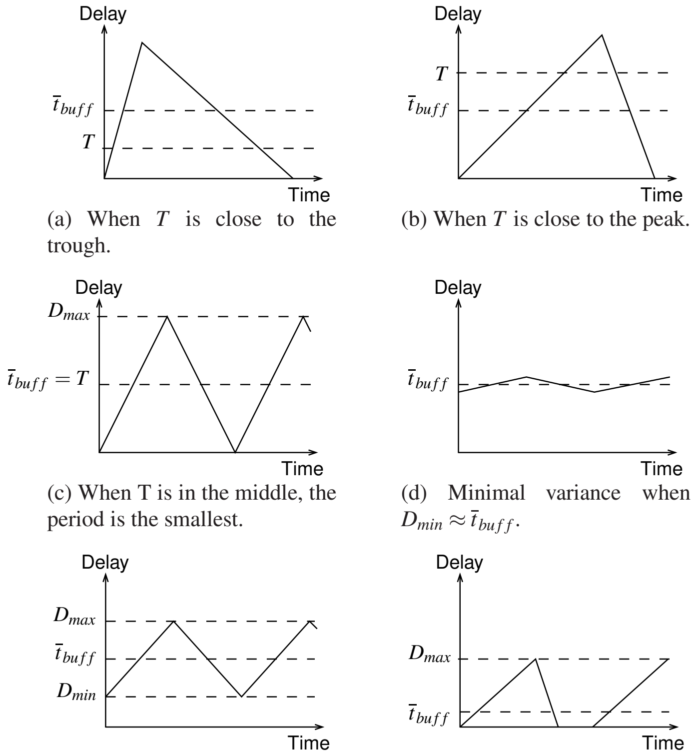


Figure 2: Buffer emptied (delay sensitive) case: buffer delay oscillates between D_{max} and 0.

Optimizing for Throughput. First, we consider the more straightforward case where we try to optimize for throughput while minimizing latency as a secondary objective. Figures 3(a), 3(b) and 3(c) show how the "skew" of the sawtooth waveform changes as T varies for a fixed D_{max} and D_{min}. As T approaches the trough (Figure 3(a)) or peak (Figure 3(b)) of the waveform, the period of the waveform increases and the algorithm will remain in one state longer than the other. If the time to switch between states is longer, we will be less responsive to changes in the network. To minimize the period, we set $T = \bar{t}_{buff}$, so that we obtain a symmetric waveform (Figure 3(c)) where $t_f = t_d = 2(T + RTT)$.

Next, we observe that the peak (D_{max}) and trough (D_{min}) can be set independently from the period while keeping t_{buff} constant. Figures 3(a) to 3(c) show cases with maximum peak-to-trough height with $D_{min} = 0$ and $D_{max} = 2\bar{t}_{buff}$. In these scenarios, the buffer is empty only instantaneously at some points. Throughput variations in a real network could however cause the buffer to be emptied for a longer period, resulting in periods where the link is not utilized. This suggests that we want $D_{min} > 0$, so that we can achieve full utilization of the link at all times. If we were to increase D_{min} to a value close to $\bar{t}_{buff}$ as shown in Figure 3(d), the variance of t_{buff} would become negligible, but this is undesirable because in order for us to accurately probe for the available bandwidth, we need a minimum amount of variation. As both extremes are undesirable, we choose the middle ground by



(a) When T is close to the trough.

(b) When T is close to the peak.

(c) When T is in the middle, the period is the smallest.

(d) Minimal variance when $D_{min} \approx \bar{t}_{buff}$.

(e) Chosen configuration when optimizing for throughput.

(f) Chosen configuration when optimizing for latency.

Figure 3: Different shapes of the waveform under i) Buffer Full case with the same average buffer delay ($\bar{t}_{buff}$) (a) to (e), and ii) Buffer Emptied case when $\bar{t}_{buff}$ is low (f).

setting:

$$\begin{aligned} D_{max} - D_{min} &= \bar{t}_{buff} \\ D_{min} &= \frac{\bar{t}_{buff}}{2} \end{aligned} \tag{3}$$

as shown in Figure 3(e).

Optimizing for Latency. End-to-end delay consists of processing delay, transmission delay, propagation delay and queuing delay, in which transmission delay and queuing delay are the dominant components. Assuming that a latency-sensitive application has some maximum tolerable latency L_{max} that it can work with, the sum of the round-trip delay RTT and the maximum buffer delay D_{max} should not be larger than L_{max}, i.e., $RTT + D_{max} \leq L_{max}$.

When we tried setting D_{max} as $L_{max} - RTT$, we found that it results in a very steep gradient when $\mathbb{U}$ is low, which causes the latency to often overshoot the target especially when there are variations in the network. This suggests that D_{max} needs to be moderated by a function of $\mathbb{U}$, i.e., D_{max} needs to be smaller than $L_{max} - RTT$ when $\mathbb{U}$ is low. After some experimentation, we found that a simple cubic function of $\mathbb{U}$ works well:

$$D_{max} = \mathbb{U}^3(L_{max} - RTT) \tag{4}$$

In other words, Equation (4) determines the $\mathbb{U}$ required when we operate in the buffer emptied regime. To the best of our knowledge, PropRate is the first algorithm that explicitly attempts to cause the bottleneck buffer to be emptied periodically. Other algorithms that try to keep latency low [22, 6] would also do the same, but it happens implicitly. PropRate's deliberate control over the filling and emptying of the bottleneck buffer allows us to obtain a smooth performance frontier that enables us to trade-off throughput for latency, or vice versa as shown in §5.2.

Setting Parameters for PropRate

In general, we expect an application to specify a maximum end-to-end roundtrip latency L_{max} that it can tolerate to guarantee good user experience. For example, L_{max} for RTC applications like Skype and Facetime should be below 100 ms, while that for throughput-intensive applications could potentially be larger than 200 ms. Given the measured RTT, we expect the target average buffer delay $\bar{t}_{buff}$ to be $\leq L_{max} - RTT$.

Next, we need to decide whether PropRate will operate in the buffer full or buffer emptied regime. From Equations (2) and (4), we know that for the buffer emptied regime:

$$\bar{t}_{buff} = \frac{1}{2}\mathbb{U}^4(L_{max} - RTT) \tag{5}$$

Hence, we will operate in the buffer emptied regime only if $\mathbb{U} < 1$, or in other words when

$$\bar{t}_{buff} < \frac{L_{max} - RTT}{2} \tag{6}$$

where L_{max} is the maximum end-to-end latency described above and RTT is the round-trip time *excluding the buffer delay*. RTT is unlikely to change in the short term and can be estimated dynamically during operation.

The derivation of the parameters involves only simple math and is omitted here due to space constraints. Initially, we set $T = \bar{t}_{buff}$. The remaining parameters can be derived as functions of T. For the buffer full case we use the following parameters:

$$\begin{aligned} k_f &= \frac{\frac{3}{2}T + RTT}{T + RTT} \\ k_d &= \frac{\frac{1}{2}T + RTT}{T + RTT} \end{aligned} \tag{7}$$

For the buffer emptied case ($\mathbb{U} < 1$), we apply these instead:

$$\begin{aligned} \mathbb{U} &= \left(\frac{2T}{L_{max} - RTT}\right)^{\frac{1}{4}} \\ k_f &= \frac{\frac{2}{\mathbb{U}}T + RTT}{T + RTT} \\ k_d &= \frac{RTT - \frac{1-\mathbb{U}}{\mathbb{U}}k_f t_f}{\frac{1}{\mathbb{U}}T + RTT - \frac{1-\mathbb{U}}{\mathbb{U}}t_f} \end{aligned} \tag{8}$$

where t_f, the time spent in the *Buffer Fill* state, can be obtained by $t_f = \frac{D_{max}}{k_f - 1}$. k_f and k_d are set as functions of the threshold T and the round-trip time *excluding the buffer delay RTT* (note that $\mathbb{U}$ is also a function of T).

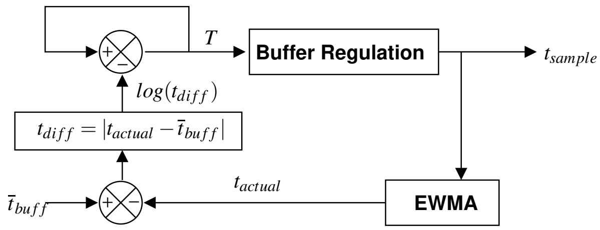


Figure 4: The negative-feedback loop to adjust T dynamically.

Updating the Threshold T

Since our network model does not consider network volatility, and we make decisions based on a snapshot of the network conditions after a delay of about one RTT, it is not surprising that the resulting buffer delay would be different from the target buffer delay $\bar{t}_{buff}$. Our key insight is that we can view *buffer regulation* (see §4.1) as a black box where an input T produces a resulting effective buffer latency t_{actual} and thereby improve the performance by introducing a higher-level negative feedback loop to update T until t_{actual} converges to the required $\bar{t}_{buff}$. This is illustrated in Figure 4.

We can measure the instantaneous buffer delay from each ACK packet (described in §4.1). For each BDP window of packets, we compute the instantaneous buffer delay t_{sample}. t_{actual} is computed from these instantaneous values with an exponentially weighted moving average (EWMA):

$$t_{actual} = \frac{7}{8} t_{actual} + \frac{1}{8} t_{sample} \tag{9}$$

We adjust T after each BDP window of packets, and T is incremented by $\log(t_{actual} - \bar{t}_{buff})$ in the Buffer Fill state and the actual average latency $t_{actual} > \bar{t}_{buff}$. In the Buffer Drain state, T would be decremented by the same amount if $t_{actual} < \bar{t}_{buff}$. We use $\log(\bar{t}_{buff} - t_{actual})$ instead of $\bar{t}_{buff} - t$ because network variations could result in t_{actual} becoming much larger than $\bar{t}_{buff}$, so we need a means to scale down the observed difference.

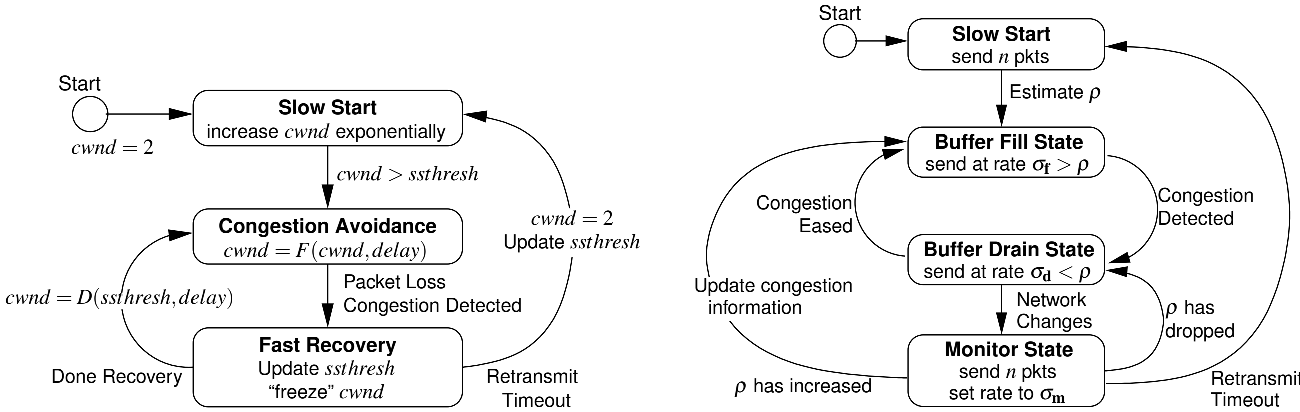
Implementation

To explain how PropRate was implemented, we thought it would be helpful to compare it to the implementation of conventional *cwnd*-based TCP. Figure 5(a) illustrates a general *cwnd*-based TCP implementation. Different TCP variants will set the *cwnd* and *ssthresh* according to different policies, represented by the functions F and D.

Both the conventional *cwnd*-based implementation and our new implementation begin in a Slow Start state. For the *cwnd*-based stack, the *cwnd* is initially set to 2 and is doubled every RTT, until the *cwnd* exceeds *ssthresh*. In PropRate, a burst of 10 packets are sent to obtain an initial estimate of the receive rate ρ. If this is not sufficient to achieve an estimate, then the number of packets is doubled and another burst is sent. This is repeated until a rate estimate is obtained. We decided on 10 packets by taking reference from Google [8], which argued that the TCP initial congestion window can be increased to 10 to reduce latency at large without causing congestion.

After Slow Start, the conventional *cwnd*-based implementation will enter a Congestion Avoidance state. In this state, the *cwnd* is now increased or decreased based on the function $F(cwnd, delay)$ for each ACK received. For traditional AIMD protocols like New Reno, $F = \frac{1}{cwnd}$, while delay-based protocols like Vegas set F differently according to whether there is network congestion. This process continues until network congestion is detected, presumably due to a packet loss and it goes into the Fast Recovery state. For our rate-based TCP implementation, we have an analogous state called the Buffer Fill state. In this state, the sending rate is set to $\sigma_f > \rho$ which will essentially cause the buffer to fill. Once the congestion is detected by the congestion detection module, it switches to the Buffer Drain state. In the Fast Recovery state of the *cwnd*-based TCP implementation, the *cwnd* is controlled by the fast recovery algorithm to retransmit packets and prevent the pipe from draining. Once the sender has received the ACKs for the retransmitted packets, recovery is complete, the *cwnd* is set to *ssthresh*, and the state returns to the Congestion Avoidance state. If fast recovery fails and the retransmission times out, the implementation will reset *cwnd* to 2 and return to the Slow Start state. Similarly, our rate-based TCP implementation sends packets at a rate of $\sigma_d < \rho$ to drain the buffer in the Buffer Drain state. Once the congestion is eased, it switches back to the Buffer Fill state. Also, if there is a retransmission timeout, the implementation returns to the Slow Start state. In this way, the rate-based implementation provides protocol designers with the flexibility to set the sending rate based on different requirements of latency and throughput. In particular, PropRate sends at the rates $\sigma_f = k_f \rho$ and $\sigma_d = k_d \rho$ ($k_f > 1, k_d < 1$) in the Buffer Fill state and Buffer Drain state respectively, which are proportional to the measured receive rate of the receiver, hence the name *PropRate*.

The key difference between PropRate and the *cwnd*-based TCP implementation is that we have an additional *Monitor state*. In the Buffer Drain state, we send at a rate that is slower than the receive rate to ease the congestion. However, if we stay in the Buffer Drain state for an extended period of time, it implies that something is wrong with either the congestion detection module or the measured receive rate ρ. In this case, we switch to the Monitor state. In the Monitor state, we use a burst of n packets to obtain a new estimate of the receive rate. While the new estimate is being obtained, the sending rate σ_m is conservatively set to $\sigma_d/2$ to avoid flooding the buffer. If the new rate estimate ρ is greater than or equal to the current estimate, we conclude that the network condition is indeed good. We update congestion detection module and switch to the Buffer Fill state. Otherwise, it means that the network is still congested, and we simply return to the Buffer Drain state to continue draining the buffer. The *cwnd*-based TCP implementation does not need a state that is equivalent to the Monitor state because the *cwnd*-based congestion control implementation relies on packet loss to signal congestion, and thus does not need to worry about underlying changes in the



(a) Conventional *cwnd*-based mechanism.

(b) Rate-based mechanism.

Figure 5: Comparison of TCP packet regulation mechanisms.

network.

Buffer Regulation

The congestion detection module determines the efficiency of the negative oscillation feedback loop. Instead of using packet drops to detect congestion, PropRate adopts the technique used in recent works [15, 19, 22] to detect the congestion by estimating the buffer delay. Specifically, PropRate switches to the Buffer Fill state and Buffer Drain state when the estimated buffer delay t_{buff} is above and below a threshold T respectively. Because the underlying one-way delay might change due to variations in mobile networks, it is possible for the buffer to be completely emptied, even when the measured buffer delay remains above T. This happens when the underlying one-way delay suddenly increases due to network changes. To address this scenario, we switch to the Monitor state when the algorithm remains in the Buffer Drain state for an extended period of time. As for the congested uplink case where returning ACK packets are delayed, using a timer here might cause the unnecessary switch to the Monitor state due to the delayed ACK packets. Hence we decided to limit the number of packets transmitted during the Buffer Drain state before switching to the Monitor state. The cap is set at $RTT * \rho$ here.

The level of congestion in the network is inferred by estimating the buffer delay t_{buff} from the observed changes in the one-way delay of the received packets. When the buffer is empty, this is simply the propagation delay of the packet. When the packet is queued in the buffer, the one-way delay will increase to include the time it spends in the buffer. Thus, we estimate t_{buff} by taking the difference between the currently observed one-way delay RD and the minimum one-way delay observed in the recent past RD_{min}, as illustrated in Figure 6(a). The relative one-way delay RD is computed by subtracting the timestamp t_r when the packet is received from the timestamp t_s, which is the point when a packet is queued for delivery at the sender. Since these two timestamps are available in ACK packets, the sender can easily compute the relative one-way delay as $RD = t_r - t_s$ from the returned ACK, and estimate the buffer delay as $t_{buff} = RD - RD_{min}$.

Packet losses need to be handled. When a packet loss occurs, `TSecr` is set to the `TSval` of the last in-sequence data packet before the packet loss, rather than to the most recently received packet. As the sender already records the sending time of each packet to determine retransmission timeouts, it can obtain the sending time of any given packet.

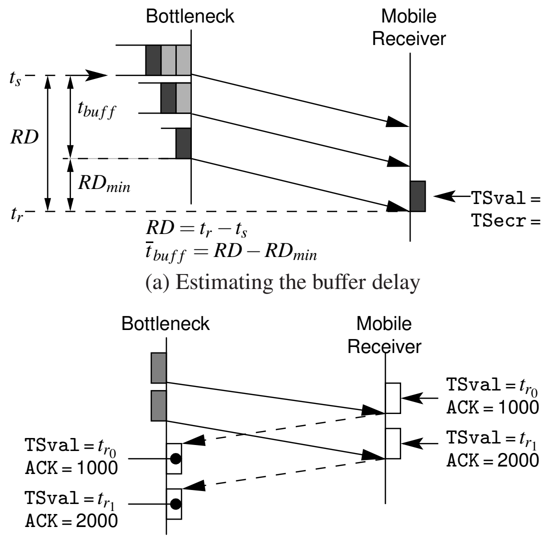
As astute reader would likely highlight at this point that the bottleneck might not occur at a cellular base station and so it is incorrect to assume that the buffer will only be filled by the packets from our PropRate. It is possible that the buffer might be filled by cross traffic instead. The truth is that we had initially attempted to model the network dynamics, but we were not able to do a good job, so what PropRate effectively attempts to do is to make control decisions based on a snapshot of the network, taking into account the one RTT delay in the measurements. This seems to work well in practice. Our model is abstract and makes no assumptions on the location of the bottleneck or that the bottleneck will not move. A change in the position of the bottleneck will seem to the network to be a change in the receive rate. Similarly, cross traffic would also be observed as a decrease in the receive and a possible increase in the baseline one-way delay. We see in §5.4 that PropRate's performance is comparable to that for CUBIC and BRR on Internet-scale paths.

Bandwidth Estimation and Rate Control

A natural approach to obtain an estimate of the receive rate or bottleneck bandwidth, is for the receiver to perform the estimation based on the received packets and to send the estimate back to the sender [22, 24]. The same approach could also be adopted for PropRate. However, congestion control is strictly a sender-side initiative and does not depend upon the receiver's TCP stack. Thus, to maintain this compatibility by avoiding modifications to the TCP receiver, we decided to adopt an indirect method that allows the rate estimation to be performed at the sender when the TCP timestamp option is enabled, which makes our stack compatible with existing *cwnd*-based protocols at the receiver. The TCP timestamp option is enabled by default on Android and iPhone devices, which together account for more than 95% of the cur-

Table 1: Implemented kernel APIs.

Functions	Parameters	Returned value	Description
`update`	$tstamp$, $bytes\_recv$	ρ	Update the module with new information of packet reception.
`get_rate`	$curr\_state$, ρ	σ_{curr_state}	Obtain new sending rate from kernel module.
`threshold`	$curr\_state$	T_{curr_state}	Obtain threshold value from which congestion is triggered.



(a) Estimating the buffer delay

(b) Estimating the receive rate.

Figure 6: Using TCP timestamps for estimation at the sender.

rent smartphone market [2]. The only caveat is that the TCP timestamp granularity of the receiver needs to be known to the sender. Since most mobile devices currently use 10 ms as the timestamp granularity, we can assume this granularity by default. Even when this assumption is wrong, we also have developed a simple technique that will allow us to estimate the granularity within one RTT.

When the TCP timestamp option is enabled, a TCP receiver will send ACK packets with the `TSval` set to its current time. This timestamp is the same as the receiving time of the data packet. Thus, the packet arrival times are effectively embedded in the ACK packets. From the ACK number and the timestamp value, the sender can determine the number of bytes received by the receiver. In the example illustrated in Figure 6(b), the sender can determine that 1,000 bytes have been received in the time period between t_{r_0} and t_{r_1}.

Packet losses, which will cause the ACK number to stop increasing, need to be handled. We can obtain reasonably good estimates by assuming that each duplicate ACK corresponds to one MSS packet received. Also, when enabled, SACK blocks in the ACKs can be used to accurately determine the exact number of bytes received.

PropRate uses an exponentially weighted moving average (EWMA) of the instantaneous throughput measured from a sliding window to estimate ρ from the timestamps of packet arrivals. As suggested by Xu et al., we use a window of 50 bursts to estimate the instantaneous throughput [25]. In other words, instead of sizing our sliding window in terms of packets, the sliding window consists of either 50 consecutive and distinct timestamps with reported packet arrivals (which would contain at least 50 packets), or 200 ms (if there are fewer than 50 distinct timestamps in this period). We found that while this approach is suitable for very fast flows, the length of the sliding window can become as long as a few seconds when the throughput is low. Since using a long history of timestamps to estimate the throughput would not reflect the instantaneous throughput accurately, we limit the maximum length of the sliding window to 500 ms.

Kernel Implementation Details

We implemented PropRate in the Linux 3.13 kernel in a manner that is similar to the existing *cwnd*-based TCP congestion control modules. The decoupled components of congestion detection, rate control and packet dispatching are abstracted as API functions so that new rate-based TCP variants can be implemented as new kernel modules. The three functions are summarized in Table 1.

While both of the traditional *cwnd*-based mechanism and our new rate-based mechanism allow a congestion control module to determine the sending rate, the regulation of the sending rate is done differently. Traditional *cwnd*-based congestion control modules will adjust the *cwnd*, which indirectly controls the sending rate by determining the number of unacknowledged packets that the TCP stack can have in flight at any time. The sending of new data packets is clocked by the received ACK packets. In our new rate-based mechanism, the congestion control module explicitly sets a rate, and packets will be sent continuously (clocked by a timer) at a given rate as long as there are available packets to be sent.

To implement our new mechanism for the Linux kernel, we had to modify the kernel to add hooks to our new module. In total, we added about 200 lines of code to the kernel and the kernel module is about 1,500 lines of code. The following is a brief description of some of the significant modifications.

Sending packets. At every kernel tick interval, the control mechanism determines how many packets should be sent based on the sending rate obtained from the congestion control module. If the result is a non-integer number, we round up this value in the Buffer Fill state and round down this value in the Buffer Drain and Monitor states, as it is preferable to send full-size packets of 1 MSS. A history of the exact number of bytes sent is kept so that we can make up for the rounding discrepancy over a few ticks.

Receiving ACK. Every ACK packet received will be used to compute the one-way delay and the `update` function of the congestion control module will be called. When packets are lost, SACK is used to determine the number of newly received bytes. In order to avoid code duplication, we added hooks into the TCP SACK processing routine to pass this information directly to our module.

Handling packet losses. When there are packet losses, the Linux TCP stack enters the Fast Recovery state, where lost packets are automatically retransmitted based on the SACK information. We use the same mechanism by simply ignoring the *cwnd* and continue transmitting at the specified rate. Basically, PropRate does not require packet losses to be specially handled by fast retransmission algorithms like PRR [7].

Performance Evaluation

The network conditions of cellular data networks can vary greatly even over a short period of time [24]. To ensure that the comparison is consistent among the algorithms, we performed our main evaluation using the trace-driven network emulator Cellsim, which was used by Winstein et al. in evaluating Sprout [22]. Cellsim acts as a transparent proxy between two devices, and controls the forwarding of packets by buffering and releasing them according to the timing and bandwidth in a given trace. The original Cellsim source code implements an infinite buffer. Since traditional *cwnd*-based algorithms like CUBIC react to packet loss due to buffer overflow, we enhanced Cellsim by introducing a finite drop-tail buffer for a fairer evaluation.

We collected network traces from three cellular ISPs using a custom Android application by saturating the network with UDP packets. `tcpdump` was used to capture the packet traces. We used UDP to measure the available network bandwidth because we did not find any evidence of UDP traffic throttling by the ISPs. Two sets of traces were collected from each ISP—one set with the phone in a stationary position and the other set taken on board a vehicle that was driven around campus. We followed the evaluation methodology and used the same emulation parameters as Winstein et al. [22], using both the uplink and downlink traces in the network emulator. The propagation delay was set to 20 ms and `iperf` was used to generate TCP traffic.

We also evaluated various algorithms over real LTE networks using an Android phone as the receiver. To mitigate the variations in a real network, we performed a large number of experiments over a 6-month period and collected over 60 GB of 4G/LTE trace data. Finally, to prove that PropRate works and competes well in traditional wired networks, we evaluated PropRate in both local and inter-continental scenarios using a sender in Singapore and receivers on AWS in the US, the UK, Australia and Singapore. We did not use AWS for the sender to avoid measurement errors from virtualization [26].

We compared PropRate to the algorithms listed in Table 2. They range from traditional TCP congestion control algorithms, such as CUBIC and Vegas, to state-of-the-art algorithms like LEDBAT [19], PROTEUS [23], Sprout [22], PCC [6], Verus [27] and BBR [4]. For PCC, we only used its default delay-sensitive utility function, as we found in our experiments that its throughput-sensitive mode was too aggressive in practice and caused buffer overflow almost all the time.

Table 2: Comparison of state-of-the-art algorithms.

Algorithm	Sending Regulation	Congestion Trigger
PropRate	Rate-based (+ window-capped)	Buffer Delay
RRE [15]	Rate-based	Buffer Delay
BBR [4]	Rate-based	Bandwidth Delay Product
PCC [6]	Rate-based	Utility Function
PROTEUS [23]	Rate-based	Rate Forecast
Sprout [22]	Window-based	Rate Forecast
Verus [27]	Window-based	Utility Function
LEDBAT [19]	Window-based	Buffer Delay + Packet Loss
CUBIC [10]	*cwnd*-based	Packet Loss
Vegas [3]	*cwnd*-based	Packet Loss
Westwood [16]	*cwnd*-based	Packet Loss

We obtained the source code for all the algorithms above and made minor modifications to make them work locally on our machines. We found an implementation of BBR in the Linux kernel 4.10 network development branch. The only exception was PROTEUS [23], which we implemented following the description in the paper because the source code was not available.

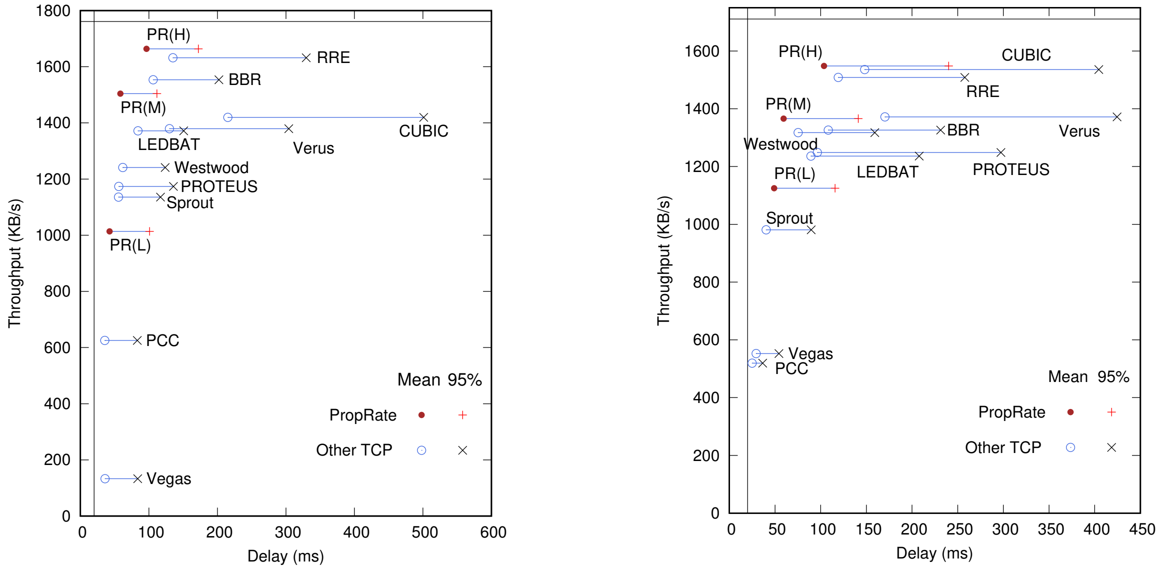
PropRate can be configured to operate at a range of operating points by setting the target average buffer delay $\bar{t}_{buff}$. In our experiments, we present the results of $\bar{t}_{buff} = 20$ ms, 40 ms and 80 ms, and denote them as PR(L), PR(M) and PR(H), respectively. PR(L) operates in the Buffer Emptied regime, PR(H) operates in the Buffer Full regime, and PR(M) is approximately at the crossover point between the 2 regimes.

Trace-based Emulation

In Figure 7, we plot the mean and 95th-percentile one-way packet delay against the total average throughput of different algorithms for both the stationary and mobile traces for one ISP. The results for the two remaining ISPs are omitted because of space constraints, and because they were similar and did not yield additional insight. The 95th-percentile value follows the evaluation metric earlier proposed by Winstein et al. [22]. The black dotted lines indicate the maximum average throughput and minimum latency for each trace.

We make several observations from these results: (i) PropRate outperforms all existing state-of-the-art algorithms. Modern forecast-based algorithms like Sprout, PCC and PROTEUS that are optimized for low latency in cellular networks do achieve low latencies, but at significant penalties to the resulting throughput. (ii) As expected, all the algorithms seem to perform worse in the mobile traces since the underlying bandwidth variations are larger than those for the stationary traces, which are very stable. (iii) BBR performs surprisingly well even though it was not specifically designed for mobile cellular networks.

Handling poor connectivity. Our mobile cellular ISPs had generally good performance. We also repeated our experiments using the mobile traces used by Winstein et al. [22], which were collected by driving around Boston. The results for the 5 traces were similar, with the exception of the Sprint trace shown in Figure 9. Upon investigation, we found that



(a) Stationary (b) Mobile

Figure 7: Comparison of performance for stationary and mobile ISP traces.

the main difference between our traces and the Sprout Sprint trace is that the Sprint trace had significantly lower bandwidth, high network variations and many *periods of network outage* (54% of the time!). We found that PropRate does not perform well because the outages resulted in many packet losses. While it might seem that RRE, Westwood and CUBIC achieved higher throughput, the absolute throughput for the network was so low that the difference is not significant. These algorithms are very aggressive, which resulted in high latencies (note the log scale in Figure 9). What we found surprising was that BBR seemed to be relatively robust to the network outages. We are currently investigating how BBR achieves this robustness. We also noted that PCC was not able to cope well with the outages. That said, we should not read too much into the results for this trace because it represents a poorly performing mobile network, which should not be the norm moving forward. We are highlighting this trace merely because it reveals the relative robustness of existing algorithms to network outage.

Effectiveness of Negative Feedback Loop (NFL). To investigate the effectiveness of the negative-feedback loop (see §3.2) in achieving convergence to the desired latency, we compared PropRate with and without the negative-feedback loop for both the stationary and mobile traces, with the expected buffer delay $\bar{t}_{buff}$ varying from 20 to 120 ms. In Figure 8, we plot $\bar{t}_{buff}$ against the observed average buffer delay for the mobile trace. The diagonal line represents the perfect match between target and achieved latency. We can see that the negative-feedback loop allows PropRate to converge closer to the target latency.

We omit the results for the stationary trace because there is no significant difference between the two cases (NFL vs. no NFL). Upon further analysis, we found that the negative-feedback loop only helps if there were significant network

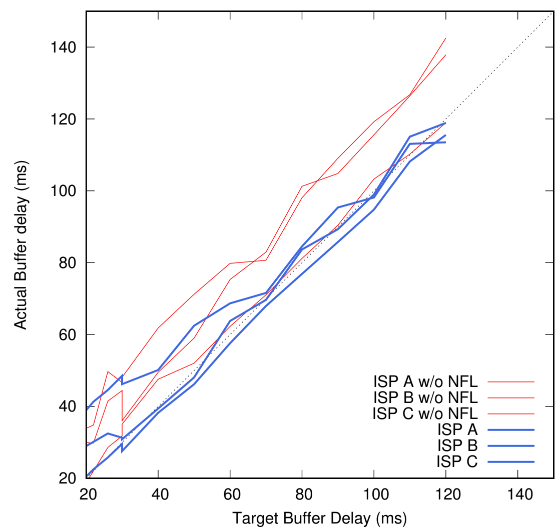


Figure 8: Average latency performance of PropRate versus the target buffer delay on mobile traces.

volatility and outages. Our network model does not take into account network volatility and so the negative-feedback loop acted as a mechanism to correct for high volatility.

Understanding the Performance Frontier

PropRate can be tuned to achieve a range of performance trade-offs between throughput and latency. To visualize the performance frontier that is achievable by PropRate, we plot more configurations for PropRate by varying $\bar{t}_{buff}$ from 12 ms to 30 ms in steps of 1 ms and from 30 ms to 120 ms in steps of 4 ms in Figure 11 for the mobile trace shown earlier in Figure 7(b). We also reproduce the results for CUBIC, BBR, Sprout and PCC for comparison. To the best of our knowledge, PropRate is the only algorithm that achieves such a smooth and continuous frontier. Sprout claims to achieve this using a parameter called the *confidence percentage* and we

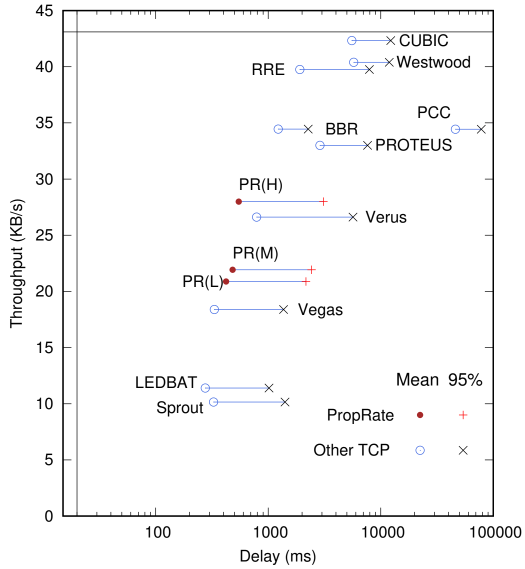


Figure 9: Results for the Sprint (mobile) trace [22].

tried it on the various traces. Unlike PropRate, Sprout was not able to produce a smooth frontier for most of the traces. Furthermore, PropRate's frontier is significantly closer to optimal performance.

Practical 4G/LTE Networks

To validate our emulation results, we evaluated the performance of PropRate over real cellular data networks. Like the emulation experiments, we used the same configurations described in §5.1. We used `iperf` to generate the test traffic and started a 30-second TCP transfer from our server to an Android smartphone over the LTE networks. We could not get Sprout to compile natively on Android, so we tethered the phone to a laptop that could run Sprout. We measured the one-way delay using a loopback configuration, where both sender and receiver were on the same machine.

The operating conditions for real cellular networks can vary significantly over time, even when the mobile device is at a stationary position. Thus, each experiment was repeated many times for each algorithm. We plot the results for one ISP in Figure 12. Due to space constraints, we omit the results for the remaining ISPs, as they are similar. The key takeaway from our results for the real cellular networks is that they are similar to and validate the results for our trace-based emulation experiments in Figure 7. After having worked with Cellsim for many years, we are pleased to report that Cellsim is an excellent emulation tool. Its emulation results tend to be very close to that obtained from experiments with real LTE networks.

Playing Well with Others in the Wild

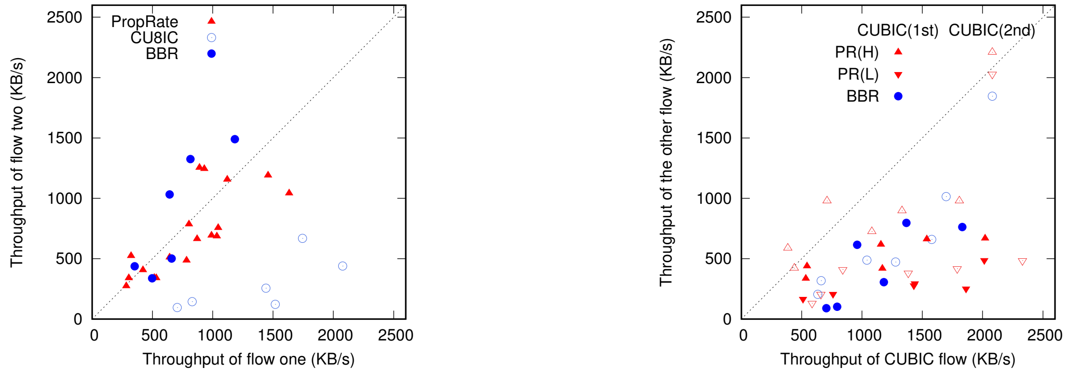
One of the key concerns for a new TCP variant is incremental deployability. Given that CUBIC is the dominant TCP of the Internet and it is known to be very aggressive in terms of ramping up its congestion window to fill the available buffer, can a new latency-sensitive TCP variant work in such an environment?

As a baseline, we first investigated how two flows of the same algorithm would affect each other using our stationary and mobile traces for each of the three ISPs. We conducted a series of experiments using the same PropRate configurations described in §5.1 and also with BBR and CUBIC. In each experiment, we first started one TCP flow and then a second flow after 30 s. Both flows continue for another 60 s, and we compute the average throughput for each flow during the latter 60 s. In Figure 10(a), we can see that both PropRate and BBR are generally more friendly to themselves than CUBIC. In terms of the ratio of the throughput of the second flow to that of the first flow, we found that the average throughputs were only 23% for CUBIC and close to 100% for BBR and PropRate.

It is clear that an algorithm designed to minimize latency would not be able to contend effectively against CUBIC. Also, with CUBIC cross traffic, it would be impossible to achieve low latency since CUBIC would fill up the bottleneck buffer and cause packet losses. Nevertheless, we investigated how well existing algorithms contend with CUBIC to understand how they would perform if deployed in the wild. Due to space constraints, we only plot the results for PropRate and BBR in Figure 10(b), because these are the only ones that can contend effectively against CUBIC, and BBR is already deployed by Google. The experimental setup is the same as that for self-contention.

We make several observations: (i) the latency-minimizing configuration for PropRate (PR(L)) receives a smaller share of the bandwidth, but is not completely starved; (ii) BBR is less aggressive than CUBIC; (iii) the throughput-maximizing configuration for PropRate (PR(H)) contends reasonably well with CUBIC. In fact, if the PropRate flow starts before the CUBIC flow, it is sometimes possible for the PropRate flow to get a slightly larger share of the bandwidth. This was surprising for us.

We also evaluated PropRate on the Internet at large using Amazon AWS servers in the US, the UK, Australia and Singapore with the TCP sender in Singapore (and not on AWS). For each sender-receiver pair, we used `iperf` to generate traffic for 30 seconds and measured the throughput, and repeated this measurement 30 times for each pair for each algorithm. Figure 13 shows the average throughput for CUBIC, BBR, PropRate(L) and PropRate(H). We found that CUBIC achieved the highest average throughput and that the throughput for BBR was generally lower than that for CUBIC, unlike the results reported for the Google B4 network [4]. As expected, PropRate(L) had lower throughput, but the difference was less than 30%. PropRate(H) performed slightly worse than BBR and CUBIC. We presented PropRate(L) and PropRate(H) here for a fair comparison with the emulations in the previous sections. We also tried larger values of $\bar{t}_{buff}$ and it comes as no surprise that throughput increases with $\bar{t}_{buff}$ until about $\frac{1}{2}RTT$, where the maximum PropRate(max) is obtained. The performance of PropRate(max) is close to that for CUBIC. This suggests that if PropRate were to be



(a) Self-contention.

(b) Contention against CUBIC.

Figure 10: Understanding contention between PropRate, BBR and CUBIC on the stationary and mobile traces of 3 local ISPs.

Table 3: CPU utilization for the state-of-the-art TCP protocols.

	Sprout	LEDBAT	PCC	Verus	BBR	CUBIC	RRE	PropRate
Intel i7-2600 @ 3.4GHz	2.4%	0.1%	12.7%	21.8%	0.1%	0.1%	0.1%	0.1%
Intel Xeon E5640 @ 2.67GHz	2.8%	0.1%	12.8%	22.6%	0.1%	0.1%	0.1%	0.1%
Intel Q9550 @ 2.83GHz	3.5%	3.0%	25.4%	25.2%	1.5%	0.3%	0.3%	0.3%

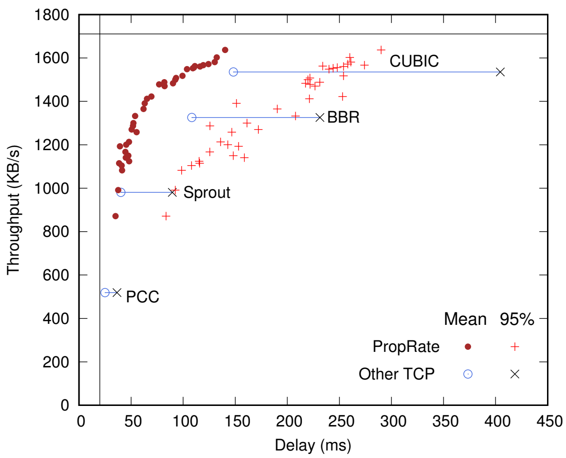


Figure 11: Performance frontier achievable by PropRate on mobile trace.

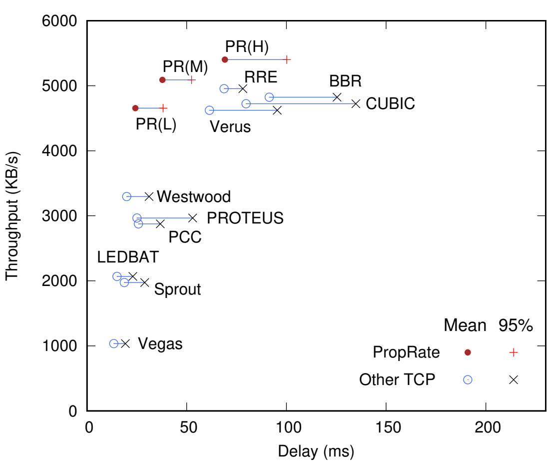


Figure 12: Throughput and latency performance for a real LTE network.

deployed on the Internet, it would be interesting to investigate how $\bar{t}_{buff}$ should be set to achieve optimal throughput,

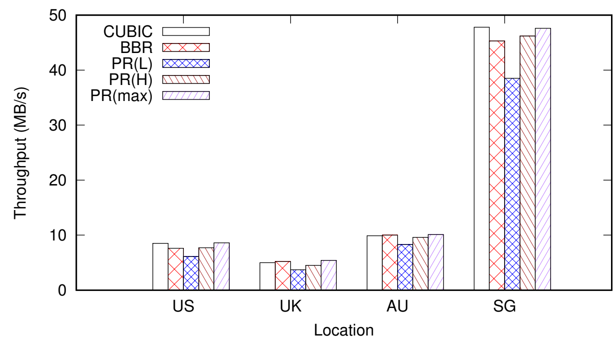


Figure 13: Throughput performance of CUBIC, BBR and PropRate for inter-continental Internet traffic.

but we leave this as future work.

Computation Overhead

In Table 3, we compare the measured CPU overhead of PropRate to the state-of-the-art algorithms. We used `iperf` to generate TCP traffic and measured the CPU utilization ratio at the sender for three different CPUs. The key takeaway is that PropRate has a low CPU footprint comparable to CUBIC. On the other hand, forecast-based algorithms like Sprout, Verus and PCC incur significantly higher computation overhead.

Discussion

PropRate represents a new approach to TCP congestion control by directly regulating the bottleneck buffer. We have demonstrated that it is a viable and promising approach. Nevertheless, some questions and issues remain to be resolved.

Network volatility. In PropRate, we assumed a steady state model of the buffer. While it might seem obvious that we would do better if we could take into account network volatility, we found that it was not straightforward for two reasons. First, network volatility is a second-order statistic

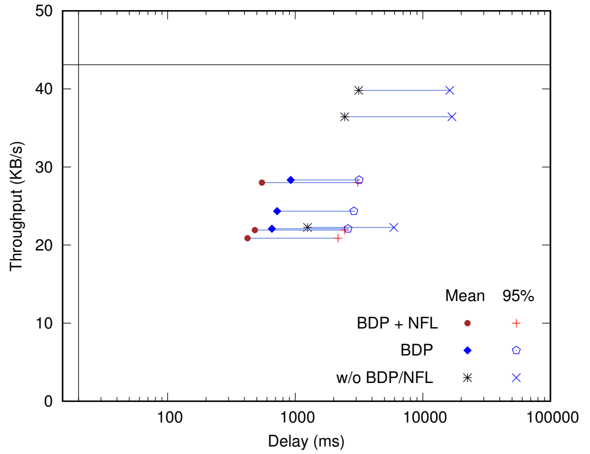


Figure 14: Demonstration of effectiveness of 2 × BDP packet cap and the negative-feedback loop for Sprout Sprint (mobile) trace [22].

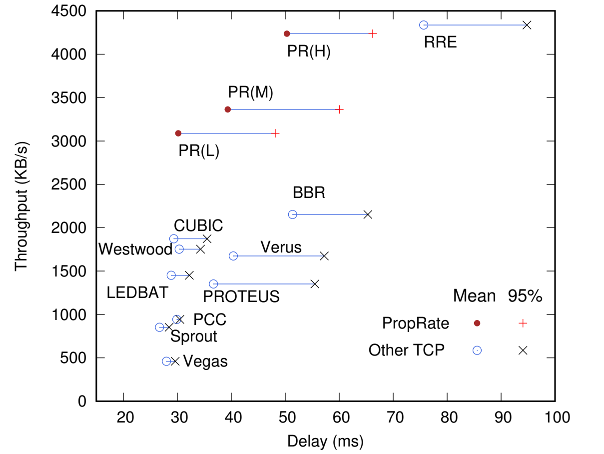


Figure 15: Downstream throughput and delay in the presence of a concurrent upstream TCP flow in a real LTE network.

that is not easy to measure accurately in practice. Second, a complicated feedback loop was prone to unstable behavior in a practical implementation. The buffer regulation algorithm that is presented in this paper is the only one we tested that could achieve stable and good performance. Nevertheless, we believe that incorporating network volatility in a clever way could potentially improve the model.

Rate-Based Fragility. *cwnd*-based ACK clocking is robust to network outage since data packets are not sent, if ACKs are not received. It would be difficult for a rate-based packet dispatch strategy like PropRate to detect network outage and stop sending packets without receiver-side coordination. As a case in point, we found that a naive implementation of PropRate performed poorly for the Sprout Sprint trace [22] due to the numerous network outages in the trace. Fortunately, we found a simple solution that we incorporated into PropRate to address this problem. We observed that if a TCP flow were operating optimally, the total number of packets in flight is roughly equal to the BDP. Therefore, we avoid having too many packet drops in the event of a network outage by temporarily stopping packets from being dispatched when the number of unacknowledged packets in flight reaches a multiple of the estimated BDP. We found that an upper limit of 2× BDP worked well in practice, as shown in Fig 14. We also verified that, as expected, this upperbound on the unacknowledged packets has negligible impact on flows in the normal case when there was no network outage. In addition, the negative-feedback loop also helps mitigate this problem to some extent.

Link Asymmetry. BBR works with RTT, so there is an underlying assumption that the link is symmetric. PropRate works with the one-way delay. This means that PropRate is able to handle the problem of congested uplinks in cellular networks [24], where a congested uplink can sufficiently delay the returning ACK of a download to result in the underutilization of the downlink. To simulate a congested uplink, we started a TCP CUBIC uplink flow simultaneously with the downlink flow that was measured. The results for a real LTE network are shown in Figure 15. Like RRE, which was developed to address this problem, PropRate is able to achieve high throughput.

Practical Deployment. Mobile cellular networks typically provide a private queue for each subscriber and implement some form of proportionate fair queuing at the base station. In other words, the buffer delay for an individual subscriber is effectively independent of the other users in the network [14] and cellular networks would be the ideal environment for PropRate to be deployed. Mobile ISPs also often have transparent proxies deployed at their base stations. This makes PropRate immediately suitable for deployment at these middleboxes.

Given that CUBIC is the dominant TCP variant on the Internet, it would be difficult for algorithms like BBR and PropRate to fully achieve their potential in reducing TCP latency. Nevertheless, as shown in §5.4, BBR and PropRate are amenable for immediate deployment since they can achieve good performance even in the existing environment. In time, we believe that there is a good chance that the dominant TCP on the Internet would evolve into a more latency-friendly variant, like BBR or PropRate.

Conclusion

In spite of the decades of work on TCP, we agree with Cardwell et al [4] that it is timely to rethink congestion control in a fundamental way. Notwithstanding the work that has already been done in BBR and PropRate, we believe that there is still room for innovation in TCP by framing it as a control and feedback loop with delayed inputs, in terms of measurable network metrics. We have shown that our buffer-regulation-based approach is a promising alternative to the state-of-the-art BDP-based approach (BBR). Our approach not only works well for mobile cellular networks, but has the additional benefit of allowing applications to choose their operating point on a wide range of performance trade-off points between latency and throughput.

References

[1] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2016, https://goo.gl/shrbpf.
[2] IDC worldwide mobile phone tracker, 2015.
[3] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE J. Sel. Area Comm.*, 13(8):1465–1480, 1995.
[4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):50:20–50:53, Oct. 2016.
[5] L. D. Cicco and S. Mascolo. A mathematical model of the Skype VoIP congestion control algorithm. *IEEE Trans. Autom. Control*, 55(3):790–795, Mar. 2010.
[6] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting congestion control for consistent high performance. In *Proceedings of NSDI '15*, May 2015.
[7] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional rate reduction for TCP. In *Proceedings of IMC '11*, Nov. 2011.
[8] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. *SIGCOMM CCR*, 40:26–33, June 2010.
[9] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. *SIGCOMM Comput. Commun. Rev.*, 30:43–56, Aug. 2000.
[10] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *SIGOPS OSR*, July 2008.
[11] R. hhHamilton, J. Iyengar, I. Swett, and A. Wilk. QUIC: A UDP-based secure and reliable transport for HTTP/2. IETF Working Draft, Jan. 2016.
[12] C. Jin, D. Wei, and S. Low. Fast TCP: Motivation, architecture, algorithms, performance. In *Proceedings of INFOCOM '04*, Mar. 2004.
[13] A. Karnik and A. Kumar. Performance of TCP congestion control with explicit rate feedback: Rate adaptive TCP (RATCP). In *Proceedings of Globecom '00*, Dec. 2000.
[14] H. J. Kushner and P. A. Whiting. Convergence of proportional-fair sharing algorithms under general conditions. *Trans. Wireless. Comm.*, 3(4):1250–1259, July 2004.
[15] W. K. Leong, Y. Xu, B. Leong, and Z. Wang. Mitigating egregious ACK delays in cellular data networks by eliminating TCP ACK clocking. In *Proceedings of ICNP '13*, Oct. 2013.
[16] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of MobiCom '01*, July 2001.
[17] K. Nichols and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20–34, May 2012.
[18] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of SOSP '13*, Nov. 2013.
[19] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low extra delay background transport (LEDBAT). IETF Working Draft, Oct. 2011.
[20] P. Sinha, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. WTCP: A reliable transport protocol for wireless wide-area networks. In *Proceedings of MobiCom '99*, 1999.
[21] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-generated congestion control. In *Proceedings of SIGCOMM '13*, Aug. 2013.
[22] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of NSDI '13*, Apr. 2013.
[23] Q. Xu, S. Mehrotra, Z. Mao, and J. Li. PROTEUS: Network performance forecast for real-time, interactive mobile applications. In *Proceeding of MobiSys '13*, June 2013.
[24] Y. Xu, W. K. Leong, B. Leong, and A. Razeen. Dynamic regulation of mobile 3G/HSPA uplink buffer with receiver-side flow control. In *Proceedings of ICNP '12*, Oct. 2012.
[25] Y. Xu, Z. Wang, W. K. Leong, and B. Leong. An end-to-end measurement study of modern cellular data networks. In *Proceedings of PAM '14*, Mar. 2014.
[26] B. N. Yunjing Xu, Zachary Musgrave and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of NSDI '13*, 2013.
[27] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of SIGCOMM '15*, 2015.