Mitigating Egregious ACK Delays in Cellular Data Networks by Eliminating TCP ACK Clocking

Wai Kay Leong, Yin Xu, Ben Leong, Zixiao Wang
Department of Computer Science, National University of Singapore
{waikay, xuyin, benleong, zixiao}@comp.nus.edu.sg

***Abstract*—It is not uncommon for the uplink buffers of cellular data networks to be saturated when the uplink bandwidths are low. This can cause the ACK packets for a downlink TCP flow to be severely delayed. Since existing TCP implementations are ACK-clocked, the downstream flow will suffer significant degradation, causing the downlink to be under-utilized. We present a new TCP variant, called *TCP Receiver-Rate Estimation* (TCP-RRE), that addresses this problem directly by eliminating ACK clocking. Instead, it uses TCP timestamps to estimate the receiving rate at the receiver, which it then uses to determine the sending rate. We show that TCP-RRE is able to improve download speeds by 2 to 4 times compared to existing TCP variants in both simulation and on real commercial cellular data networks. Our solution is practical because it is compatible with existing TCP implementations, requires no modifications to existing mobile devices, and is thus immediately deployable in existing ISP proxies.**

I. Introduction

Recent studies have shown that the significant asymmetric bandwidth in cellular data networks can cause significant degradation for a downstream TCP flow to a mobile device in the presence of a concurrent upstream flow. This problem can be mitigated if the upstream flow regulates its sending rate to avoid saturating the uplink buffer [21]. Unfortunately, this approach only works if the receiver of the upstream cooperates by implementing the solution, and will not work with a non-TCP upstream flow. Furthermore, we have found that extremely low uplink bandwidth (<200 kb/s) is not uncommon in 3.5G/HSPA mobile networks even when the downlink bandwidth is high. This is especially common in crowded areas during peak hours, like in the subway or in a shopping mall. Under such circumstances, the ACK packets of a downstream TCP flow will inevitably be severely delayed in the uplink buffer. As TCP relies on *ACK clocking* to regulate its data flow, severely delayed ACK packets can cause the downstream TCP throughput to be reduced significantly and the downlink to be under-utilized.

Naively, we can improve the utilization of the downlink by sending data packets at a rate that saturates the downlink buffer without waiting for the ACK packets. However, by doing so, we will likely lose packets from buffer overflow and cause significant delays. This is known as the bufferbloat problem [12]. In this paper, we present a sender-side rate-based algorithm, called *TCP Receiver-Rate Estimate (TCP-RRE)*, that tries to do better. Not only do we fully utilize the downlink, we also keep the occupancy of the downlink buffer low.

While there have been previous work on rate-based congestion control algorithms both for TCP [15, 17] and for UDP [6], to the best of our knowledge, TCP-RRE is the first attempt at *completely eliminating* ACK clocking in the TCP stack. Previous rate-based TCP variants all estimated the appropriate sending rates in order to determine the appropriate congestion window `cwnd`. There are many reasons why rate-based approaches are not commonly deployed. First, rate-based congestion control algorithms are typically less aggressive than existing window-based TCP variants like TCP-CUBIC [7]. This means that rate-based algorithms would contend poorly against them in the core Internet. Second, they often require modifications to the TCP stacks of both the sender and receiver, which is often hard to achieve in practice. Finally, it is fundamentally difficult to estimate the available bandwidth accurately in the presence of network fluctuations. If it is not possible to estimate the available bandwidth accurately, then we certainly cannot set the sending rate correctly.

Keshav had previously shown that rate-based flow control is optimal for networks that implement fair queuing [16]. We argue that with the emergence and growing popularity of modern cellular data networks, it is timely to revisit rate-based algorithms because cellular data networks typically maintain separate queues for individual subscribers at each base station and enforce a non-FIFO scheduling policy. It turns out that it is also common for mobile ISPs to deploy transparent proxies in their networks and split TCP is the norm and not the exception [21]. This means that if implemented at such a proxy, a rate-based algorithm would not have to contend with more aggressive window-based TCP variants. We will show in Section IV that even if implemented at a server, and not at a proxy, our rate-based approach can still achieve improved utilization compared to TCP-CUBIC.

In addition, we argue that our approach is practical because it requires only modifications at the TCP sender (which can easily be incorporated in a proxy) and does not require changes to the TCP stack of the receiving mobile devices. Because TCP-RRE estimates the available downlink bandwidth by deducing the receive rate at the sender via passive observation of the TCP timestamps on the ACK packets, it only requires that the TCP timestamp option be enabled. This option is currently enabled by default for both Android and iOS, which together currently account for some 80% of the available mobile devices.

Since we are only able to obtain a rough estimate of the available downlink bandwidth and there are often significant variation in network conditions, our key insight is to exploit

978-1-4799-1270-4/13/$31.00 © 2013 IEEE

the downlink buffer to take up the slack. We implement the rate control algorithm with a feedback loop with two states: (i) a "buffer fill" state, where the sending rate is set slightly higher than the estimated receive rate, and (ii) a "buffer drain" state, where the sending rate is set below the estimated receive rate. By oscillating between these two states depending on the observed changes in the one-way packet delays, TCP-RRE keeps the buffer occupancy low but ensures that there are always sufficient packets to maintain high link utilization. The TCP-RRE rate control algorithm also adapts naturally to the fluctuations in the network conditions.

We evaluated TCP-RRE with both the `ns-2` simulator and also with a Linux implementation over two commercial 3.5G/HSPA mobile networks. Our simulation results show that TCP-RRE is able to fully utilize the downlink bandwidth when the uplink bandwidth is low relative to the downlink bandwidth. TCP-RRE is able to improve download speeds by up to 10 times in our `ns-2` simulations and by about 2 times for our Linux implementation in real cellular data networks, compared to TCP-CUBIC. When link conditions are good, TCP-RRE's delay-based congestion detection mechanism also helps to mitigate buffer-bloat [12], by reducing the buffer occupancy by more than an order of magnitude compared to TCP-CUBIC.

While multiple TCP-RRE flows contend fairly among themselves, TCP-RRE might be prone to starvation if it needs to contend with more aggressive window-based TCP variants. This suggests that TCP-RRE should ideally be deployed at a proxy at the last hop mobile link of a mobile ISP. That said, experiments with our Linux implementation, on a server not located within an ISP, suggests that even if not, TCP-RRE is still able to achieve higher download speeds than TCP-CUBIC.

II. TCP Receiver-Rate Estimation (TCP-RRE)

In Fig. 1, we plot the ratio of the average one-way delay of 400 different 1 MB TCP downloads over a 3.5G/HSPA link—half of them with a concurrent TCP upload in the background and the other half without. The measurements were taken by tethering an Android phone to a server via USB in loopback configuration. What these results show is that when the uplink buffer is relatively empty, the uplink delay is typically smaller than the downlink delay. However, when the uplink buffer is saturated, the one-way delay of the uplink can inflate to almost 100 times to that of the downlink.

It is clear that under such circumstances, the ACK packets for the downlink flow will be severely delayed and because TCP is ACK-clocked, the downlink will be under-utilized. Our key insight is that to achieve good utilization, it suffices if *on average, we can match the sending rate to the effective link bandwidth* even if the ACK packets are delayed. While this could be easily done with the cooperation of the receiver, we do not think that it is feasible to expect all existing TCP stacks to be replaced. To avoid modifying the existing TCP stacks in the mobile devices, we developed a sender-side technique called *TCP Receiver-Rate Estimation (TCP-RRE)* to estimate the receive rate by exploiting the TCP timestamp option. In our work, we have found that it is very common for mobile ISPs

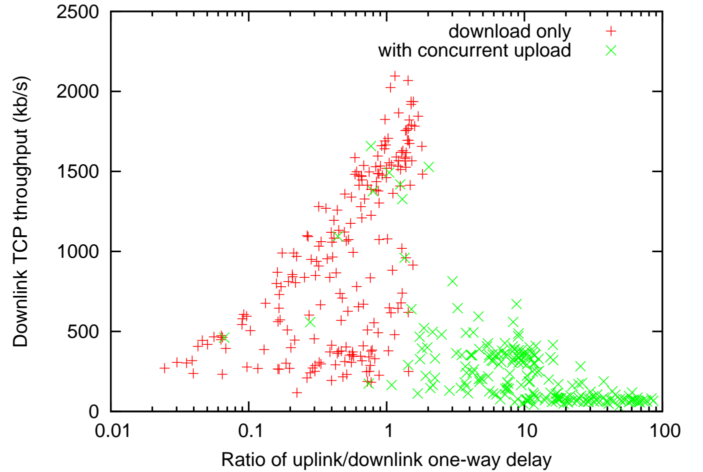

Fig. 1: Ratio of one-way delay against ratio of downlink throughput.

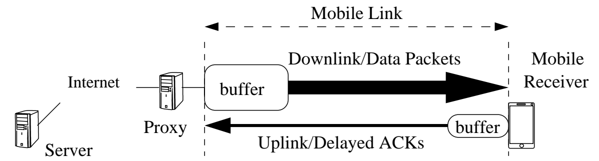

Fig. 2: Deployment scenario for TCP-RRE.

to deploy transparent proxies in their networks [21], especially on port 80. We have found this to be true for all three local ISPs and in this light, TCP-RRE is designed to be deployed at such proxies as shown in Fig. 2. In the rest of this section, we will describe how we successfully eliminated ACK clocking at the TCP sender with our new algorithm.

The general idea is to set the sending rate so that, on average, it matches the available downlink bandwidth. However, it is hard to estimate the available downlink bandwidth accurately from the receiving rate, because (i) the time granularity of TCP timestamps is too coarse to achieve high accuracy, and (ii) the available bandwidth could increase over time and if the sending rate were not increased accordingly, the receiving rate would remain the same, resulting in the underestimation of the available bandwidth.

Our approach addresses these challenges by oscillating the sending rate around a coarsely-estimated receive rate and exploiting the buffer to take up the slack. We first send a small burst of packets that is not likely to saturate or cause the downlink buffer to overflow. From the timestamps of the ACKs for these packets, the sender estimates the receive rate ρ and sends packets at a rate that is slightly larger than ρ. This should cause the downlink buffer to start filling up. We continuously estimate the receive rate ρ and also monitor the buffer growth by tracking the relative one-way delay. If the delay grows above a certain threshold, we switch instead to a "buffer drain" state, where the sending rate is reduced to a rate slightly lower than ρ, which causes the buffer to drain. When the observed one-way delay drops below the threshold, we again switch back to the initial "buffer fill" state. In this manner, we can achieve an average sending rate that matches the available downlink bandwidth while maintaining link utilization without requiring an accurate estimate of the receive rate.

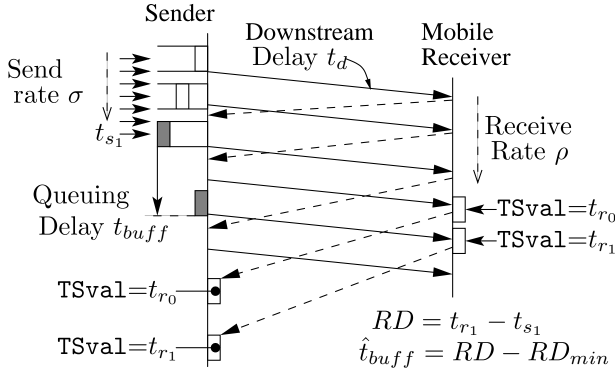

Fig. 3: Estimation of algorithm parameters.

A. Preliminaries

We will first define the various parameters and metrics that are used by TCP-RRE. Instead of being ACK-clocked, TCP-RRE packets are sent at a rate σ and packets are received at some rate ρ. This is illustrated in Fig. 3.

Estimating Receive Rate ρ. We estimate ρ using the TCP timestamp option. When the TCP timestamp option is set, both the sender and receiver will mark packets with two timestamp values: (i) `TSval`, the current kernel timestamp when the packet was sent, and (ii) `TSecr`, the "echo reply" which is the `TSval` of the oldest unacknowledged data packet received from the sender. Since we found that the delay between the reception of a data packet and the sending of the corresponding ACK packet is negligible (below 1 ms from our measurements), the `TSval` of ACK packets provides us with a good estimate of the local timestamp when the data packet was received.

Normal in-sequence ACK packets will advance the ACK sequence number, which indicates the number of bytes received at the recorded timestamp. By tracking the byte counts between consecutive ACK packets, we can estimate the receive rate. The only caveat is that the timestamp granularity of the receiver is known to the sender, but since most devices use a 10 ms granularity, this is hardly a concern. The situation becomes slightly complicated when packets are reordered or lost. To handle this, we use the TCP SACK option, which allows us to compute the exact amount of data received for each ACK. When there is no change to the SACK blocks, we simply assume one MSS of data was retransmitted. Typical TCP implementations only acknowledge every other packet (c.f. delayed ACK), but this does not affect the accuracy of our estimate, though it introduces a small, but negligible delay.

Estimating Buffering Time t_{buff}. En route from the sender to the receiver, a data packet will often spend some time t_{buff} in the downlink buffer of the 3.5G base station if the channel is fully utilized. Suppose the shaded packet in Fig. 3 is sent at time t_{s_1} and queued in the buffer for time t_{buff} until it is transmitted. It arrives at the receiver at time t_{r_1} and the receiver replies with an ACK, with `TSval` set to t_{r_1} and `TSecr` set to t_{s_1}. In principle, we should be able to determine from the `TSecr` value that the ACK is associated with the earlier shaded packet, but in practice, `TSecr` does not reflect the `TSval` of the packet ACKed when there is a delayed ACK or a lost packet. So instead, the packet sent time is obtained from the transmission buffer maintained by the kernel. We define the *relative one-way delay* RD as $t_{r_1} - t_{s_1}$. Clearly, the minimum observed relative one-way delay RD_{min} would be observed when the buffer is empty, so we can estimate t_{buff} with $RD - RD_{min}$.

B. Detailed Algorithm

TCP-RRE is organized in two key stages: (i) initial receive rate estimation and (ii) buffer management mode. We use a burst of n packets to provide us with an initial estimate of the receive rate. Thereafter, we adjust the sending rate to regulate the number of packets in the downlink buffer so as to keep the downlink fully utilized.

Initial Receive Rate Estimation. We start a new connection with a burst of n packets, to intentionally saturate and cause a small backlog of packets at the downlink buffer. Since the buffer is backlogged, the drain rate and thereby the receive rate would be approximately equal to the maximum available bandwidth. Ideally, we would want to fill the buffer to more than $1 \times$ BDP (bandwidth-delay product) because this will ensure that the buffer remains filled until the first ACK packet returns. Furthermore, because we know the maximum capacity of a 3.5G/HSPA link (e.g. 7.2 Mb/s downlink bandwidth and 50 ms RTT), we should, in principle, set n to the maximum BDP, which is about 30 packets. Since downlink buffers are typically in the order of several thousand packets in size, a burst of 30 packets is relatively small and will not likely cause the downlink buffer to overflow. However, in practice, because the typical initial TCP receiver window is only 10 packets, packets will be dropped if n is larger than 10. Also, Dukkipati et al. from Google Inc. recently argued for the TCP initial congestion window to be increased to 10, which they claim can reduce latency without causing congestion [5]. Hence, we use a setting of $n = 10$.

After the initial burst, two new data packets are sent for each ACK received in this state. This is analogous to TCP *Slow Start*. Once we receive enough ACK packets to obtain a good estimate of ρ, TCP-RRE switches into buffer management mode.

Buffer Management Mode. In buffer management mode, our goal is to keep the number of packets in the buffer B oscillating around a threshold value T. When the number of packets in the buffer is less than T, we will set the sending rate σ at a value that is higher than the estimated receive rate ρ to fill the buffer; otherwise, we will set σ to a lower value so as to drain the buffer. We discuss how we determine an appropriate value for T in Section II-E below. In practice, the number of packets in the buffer B will fluctuate between a value $B_{max} > T$ and a value $B_{min} < T$ because of a delay in the feedback from the receiver.

We know that it takes t_{buff} time for a packet to move from the tail to the head of a queue of length B, so we can estimate

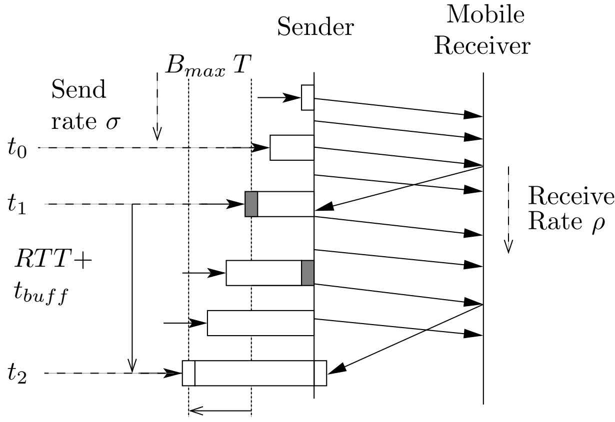

Fig. 4: Evolution of buffer during buffer fill state.

B from t_{buff} as follows:

$$B \times MSS = \rho \times t_{buff} \quad (1)$$

$$B = \frac{\rho \times t_{buff}}{MSS} \quad (2)$$

where MSS is the maximum segment size, typically 1,420 bytes.

(a) Buffer Fill State ($B < T$). When the estimated number of packets in the buffer B is less than T, we set $\sigma > \rho$ so that the buffer starts to fill. We will not be able to observe B directly and so we infer B from t_{buff} using Equation (2). In other words, we keep filling the buffer if:

$$t_{buff} < \frac{T \times MSS}{\rho} \quad (3)$$

We determine σ by analyzing the evolution of the buffer as shown in Fig. 4. It is clear from the figure that once B reaches T (shaded packet), it takes $t_{buff} + RTT$ for the sender to receive the feedback. During this time, the buffer would have increased in size at a rate $\sigma - \rho$, so

$$B_{max} = T + \frac{\sigma - \rho}{MSS} \times RTT \quad (4)$$

$$\sigma = \rho + \frac{B_{max} - T}{MSS \times RTT} \quad (5)$$

where B_{max} is the expected maximum number of packets in the buffer (to be discussed in Section II-E).

(b) Buffer Drain State ($B \geq T$). As we keep sending packets at a rate that is higher than the receive rate, t_{buff} will eventually exceed the threshold value given in Equation (3). Once this occurs we will need to reduce the sending rate σ so that it falls below ρ, as follows:

$$\sigma = \rho - \frac{T - B_{min}}{MSS \times RTT} \quad (6)$$

where B_{min} is the expected minimum number of packets in the buffer (to be discussed in Section II-E). This is completely analogous to the buffer fill state with the following caveat: after the sending rate σ is reduced when we enter this state, σ is only allowed to increase and not allowed to decrease, even if the estimated receive rate ρ drops or if the RTT decreases.

Under normal circumstances, the buffer will start to empty after σ is reduced. Eventually, t_{buff} will fall below the threshold $\frac{T \times MSS}{\rho}$, in which case we will switch back to the buffer fill state. However, ρ should remain constant as long as there are packets in the buffer. Thus, if ρ were to decrease and eventually match σ, it indicates that buffer has completely emptied. If we were to always keep $\sigma < \rho$ as ρ decreases, the lower sending rate will directly result in a lower receiving rate. Thus, both σ and ρ will eventually be reduced to zero when the buffer is empty. This is why σ is not allowed to decrease in this state. Note that it takes a while before the effect of any state change is observed by the sender due to RTT and buffer delays.

C. Adapting to Changes in Underlying Network

Our algorithm tries to match the sending rate σ to the receive rate ρ, so it will naturally adapt to observed changes in the available bandwidth. However, because it uses the estimated relative one-way delay to determine when to switch between the buffer fill and buffer drain states, RD_{min} has to be updated as the underlying one-way delay changes.

Decrease in one-way delay. During the buffer fill stage, this will result in an underestimation of the number of packets in the buffer and result in the algorithm switching to the buffer drain state later. This means that the number of packets in the buffer would oscillate about a value that is higher than the T which we had intended. This might increase the delay slightly but would not have much impact on the efficiency. It is plausible that during the buffer drain stage, the buffer might drain sufficiently, so that RD falls below the earlier observed RD_{min}, in which case RD_{min} is updated with the new value.

Increase in one-way delay. This will cause packets to spend more time on the link as the BDP increases, which will in turn cause the buffer level to drop while TCP-RRE remains oblivious to the changes. This causes no harm if the buffer never empties as the link will still be fully utilized. However, if the buffer does empty, the receiving rate ρ will be limited by the sending rate σ as discussed previously and ρ will eventually decrease to match σ. Naively, we could simply update RD_{min} to the current RD. However, it might not always be the case that the buffer is empty when ρ falls to match σ. The presence of another flow, or simply network fluctuations could also cause ρ to reduce. We thus introduce a special monitor state to probe the network, before deciding what to do next.

Monitor State. When transiting into this state, a small burst of n packets is sent, similar to the initial fill stage. This is to probe if the buffer is empty, or network conditions had indeed changed. Thereafter, as a precaution and to further drain the buffer, the sending rate σ is halved while we wait for the feedback from the initial small burst. When the feedback is received, it gives us the new receive rate estimate ρ'. If ρ' is close to the previous value ρ, it suggests that the network bandwidth did not actually change and perhaps the buffer was either empty, or a competing flow had reduced the rate temporarily. Either way, RD_{min} is updated and we switch back to buffer fill state with the new $\rho = \rho'$.

If ρ' is indeed much lower than ρ, this most likely suggests that the link bandwidth had reduced and the buffer was not

yet empty. Thus, we return to the buffer drain state with the updated $\rho = \rho'$ without changing RD_{min}.

In addition, it can be derived from the equations in Section II-B that it should typically take approximately 3 × RTT for the buffer to sufficiently drain and switch back to the buffer fill state. If the algorithm stays in the buffer drain state longer than 4 × RTT, it suggests that the buffer is not draining as expected, and so we also switch to the monitor state.

D. Handling Packet Losses

Traditional TCP congestion control uses packet losses to trigger a congestion event, under the assumption that the losses were due to buffer overflow. Because TCP-RRE will keep buffer occupancy low and will not overflow the buffer, packet losses due to buffer overflow are likely caused by competing (non-TCP-RRE) flows.

There are several existing TCP fast recovery schemes to handle packet losses that uses SACK [2, 4, 18]. All of them focus on estimating the number of outstanding packets in flight in order to reduce the `cwnd` to an optimal value to minimize delays. On the other hand, TCP-RRE does not need to estimate a `cwnd` because its sending rate is not clocked by ACK packets. In this light, as long as TCP-RRE continues to correctly estimate the receiving rate and relative one-way delay from the SACK information, it can continue to use the same sending rate for both retransmission as well as new data packets.

To avoid inadvertently causing further buffer overflow, TCP-RRE will switch to the buffer-drain state, i.e. we will slightly reduce the send rate, when there is a packet loss. If successive packet losses cause TCP-RRE to stay in the buffer-drain state for a long time, it will eventually trigger a transition to the monitor state, where TCP-RRE aggressively drains the buffer. While we did not specifically design TCP-RRE to handle packet losses, the basic algorithm was able to adapt to packet losses naturally.

E. Parameter Tuning

Other than the size of the initial burst n, TCP-RRE needs to determine the values of parameters T, B_{max} and B_{min}. Clearly, $B_{min} \geq 0$ and B_{max} should not be larger than the available downlink buffer, which is determined by the mobile ISP and is not under our control (though it is relatively easy to estimate the size of the buffer with a simple experiment).

We experimented with different settings and found that a large T will cause slower feedback due to the increased buffer delay. While this does not affect the resulting receiving rate, it makes our algorithm slow to react to network fluctuations. Conversely, setting T too low might inadvertently cause the buffer to empty, resulting in underutilization of the downlink. Also, the higher the bandwidth of the link, the faster the buffer will drain. Thus, a value of T that is suitable for low bandwidths might be too low when the bandwidth is high. A simple solution is to make T a function of the bandwidth and RTT. Because we know the receive rate ρ and the RTT, we set $T = \rho \times RTT_{min}$, which is the estimated bandwidth-delay product (BDP) and this seems to work well in practice. We note that Nichols and Jacobson's CoDel also sets 1 × RTT as the threshold to invoke early dropping of packets in the buffer [19].

B_{max} and B_{min} determine the responsiveness of TCP-RRE and how fast it will converge to T on each oscillation. If the difference between them and T is large, TCP-RRE will respond with more aggressive changes in the sending rate (See Equations (5) and (6)). Because T is set to the BDP, we set B_{max} and B_{min} to $T + \frac{BDP}{2}$ and $T - \frac{BDP}{2}$ respectively. We found that in practice, because of the imprecision in estimating the RTT and receive rate, the fluctuations in the buffer size will tend to overshoot these maximum and minimum values. This problem is exacerbated when the bandwidth is low (so T is small), and the buffer might empty completely, leading to underutilization. We address this issue by simply setting a minimum value for T at 30 packets. Likewise, to prevent excessive use of the buffer, a maximum value of T can also be imposed. We did not set a maximum as we found the typical ISP downlinks buffers are sufficiently large. Instead, we set the lower and upper limit on B_{max} and B_{min} to $+10$ and -10 packets respectively.

III. `ns-2` SIMULATION

We evaluated TCP-RRE with the `ns-2` simulator to understand and show the correctness of the protocol under a controlled environment. While we also have a Linux implementation that can be run over a real 3.5G/HSPA network, we are not able to replicate a consistent test environment over a commercial 3.5G/HSPA cellular data network. Because we are aware that our `ns-2` model cannot perfectly model real 3.5G/HSPA links, we attempt to obtain good simulation parameters by performing a measurement study of existing cellular data networks.

We evaluate TCP-RRE under three scenarios: (i) when the uplink bandwidth is very low, (ii) when the uplink bandwidth is good, and (iii) in the presence of a concurrent upstream flow. We compare it against TCP-Reno, the classic congestion control algorithm, as well as TCP-CUBIC [7], which is the current default TCP congestion control algorithm deployed in Linux and Android. We also compared TCP-RRE to TCP Vegas, which is delay-based, and to TCP Westwood, which implements a form of receive rate estimation.

A. Network Model & Parameters

In our simulations, we used the simple dumbbell topology shown in Fig. 5 to model the mobile wireless link. While a typical connection from a mobile device to the Internet will involve more nodes and links, this configuration is sufficient for us to obtain an understanding of TCP-RRE, because we expect the bottleneck to be at the mobile wireless link. What remains is to set the model parameters (link bandwidth, buffer size, RTT and loss rate) appropriately, so that we have some confidence that the resulting evaluations are meaningful for a practical cellular data network.

To determine the parameters for our model, we conducted

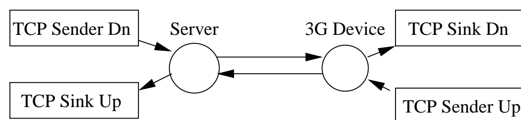

Fig. 5: Network topology for `ns-2` simulation.

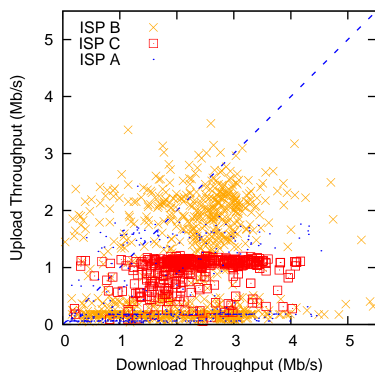

Fig. 6: Scatter plot of the upstream and downstream throughput for different mobile ISPs.

a measurement study to characterize the 3.5G/HSPA networks of the three local mobile ISPs, which we anonymously label A, B and C. While 4G/LTE plans have very recently become available, we did not manage to get access to them as there were no locally available plans when we first started our experiments. We believe that our results are likely to be applicable to 4G/LTE networks as well. We wrote a custom Android app which we installed on the phones of several volunteers to collect background measurements of local mobile networks when their phones were idle.

Link Bandwidth. To determine the bandwidth of the networks, we use UDP to send a small flood of packets of around 300 KB between a server and a mobile device and record the received throughput. We recorded over 2,000 data points over a period of several weeks, which we plot in Fig. 6. We see from the results that the available bandwidth is distributed across a large range up to 5 Mb/s downstream and 3 Mb/s upstream. All three mobile ISPs offered mobile plans with advertised rates of 7.2 Mb/s downstream and 2 Mb/s upstream. There were a small number of instances where the downstream bandwidth reached 8 Mb/s. While we omitted these samples from the graph for clarity, the bandwidth parameters in our simulations were up to 8 Mb/s for the downlink and 3 Mb/s for the uplink.

Our measurement results seem to suggest that there is no clear correlation between the uplink and downlink bandwidth. At the same time, there are significant differences in the network characteristics for different mobile ISPs. For example, ISP C seems to impose a cap on the upload bandwidth that is significantly lower than the 2 Mb/s advertised rate. Furthermore, it is not uncommon for the uplink bandwidth to be very low while the downlink remains disproportionately high. We found that not only do certain locations tend to exhibit such characteristics, they also typically occur in crowded areas like in a shopping mall or in the subway during peak hours. One possible explanation is that the mobile device might not have sufficient transmission power to overcome the interference at

TABLE I: Buffer sizes for different ISPs and phone models.

ISP	Downlink Buffer
A	2.8 MB
B	2.8 MB
C	<600 KB

Phone Model	Uplink Buffer
iPhone 5	150 kB
HTC Desire	200 kB
Galaxy Nexus	1 MB
HTC Rhyme	1 MB

certain locations. Another explanation is that there might be significant contention on the uplink due to a high volume of subscribers.

RTT & Packet Loss. We observed RTTs that varied between 50 ms to 200 ms, so the RTT parameter for our simulations was also varied within this range. The observed packet loss rate was less than 0.04% overall, which agrees with Huang et al.'s measurements that packet losses over cellular networks are rare [9]. We did simulations both with no link losses and with 0.04% link losses, and found that there was hardly any difference in the results.

Buffer Size. We estimated the uplink and downlink buffer sizes by sending a flood of UDP packets at the advertised link rate. Because packet losses are uncommon over the 3.5G/HSPA link due to its Hybrid-ARQ mechanism, any losses can most likely be attributed to buffer overflow. By examining the outstanding packets-in-flight (pif), we can deduce the buffer size from the steep increase in packet losses when the pif reaches a plateau. We also measured the downlink buffer for each ISP and found that the downlink buffer for ISP A and B was about 2.8 MB. For ISP C, the pif did not plateau but peaked in spikes, with packet losses. We suspect that ISP C has implemented some form of RED in their network, and thus did not use it in our evaluations. Another interesting observation we made from our measurements is that separate downlink buffers are maintained for each mobile device from running the measurements simultaneously with phones side-by-side. Similarly, we measured the uplink buffer for a few different phone models and found that certain phone models, especially older ones, have significantly smaller buffers. We suspect it could be because the newer models are designed to support the higher HSPA+ uplink speeds. However, these large buffers would exacerbate the ACK delay problem that we study in this paper when these phones operate on the older 3.5G/HSPA networks. Our results are summarized in Table I. Based on these results, we set the downlink and uplink buffer sizes to 2.8 MB and 1 MB in our simulations.

B. Single Download with Slow Uplink

To understand how a slow uplink can degrade a TCP flow downstream, we varied the uplink and downlink bandwidths for a 1 MB data flow downlink using TCP-CUBIC. In Fig. 7, we plot the average downlink utilization against uplink bandwidth. As expected, the utilization is independent of the uplink bandwidth when the uplink bandwidth is high, but the utilization drops once the uplink bandwidth falls below a certain threshold (See dotted line in Fig. 7). This threshold increases as the downlink bandwidth increases, since we need a higher rate of returning ACK packets to clock the TCP sender.

TCP-RRE is specifically designed to address scenarios where

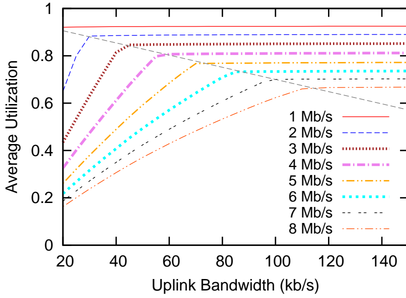

Fig. 7: Plot of downlink utilization against uplink bandwidth for TCP-CUBIC.

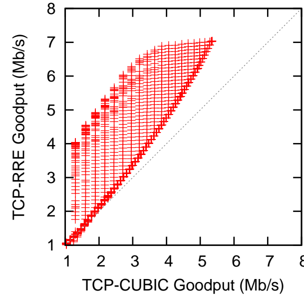

Fig. 8: Scatter plot comparing downstream goodput of TCP-RRE to TCP-CUBIC.

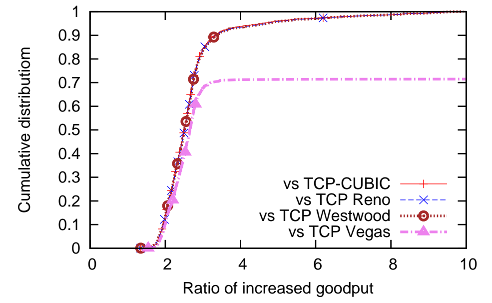

Fig. 9: Cumulative distribution function of the ratio of TCP-RRE goodput to TCP-CUBIC and TCP-Reno, in the presence of a concurrent upload.

the uplink is the limiting factor. To understand how TCP-RRE improves downlink performance, we uniformly sampled configurations of (uplink, downlink) pairs that fall below this threshold by varying the uplink bandwidth at 5 kb/s intervals and the downlink bandwidth at 0.25 Mb/s intervals. We run experiments to compare the resulting goodput of TCP-CUBIC and TCP-RRE for each of these configurations and plot the results in the scatter-plot shown in Fig. 8. These results clearly demonstrate that TCP-RRE is able to achieve a higher goodput than TCP-CUBIC when the uplink is a bottleneck. The achieved improvement depends on how close the uplink is to the threshold. It is greatest when the uplink bandwidth is significantly smaller than the threshold. While it is not shown in the figure, TCP-RRE is able to achieve a downlink utilization close to 80% for 90% of the scenarios.

C. Download with Concurrent Upload

Next, we investigate how TCP-RRE performs when the uplink is congested. To simulate a congested uplink, we simply start a single continuous upload using TCP-CUBIC. After a short delay to allow the uplink flow to saturate the uplink buffer, we start a downstream TCP transfer of 1 MB using different TCP variants. We varied the delay from 1 s to 10 s at 1 s intervals, the uplink bandwidth from 250 kb/s to 3,000 kb/s at intervals of 250 kb/s, and the downlink bandwidth was varied from 500 kb/s to 8,000 kb/s at intervals of 500 kb/s. In total, we obtained 1,620 data points for each of the TCP variants: TCP-RRE, TCP-Reno, TCP-CUBIC, TCP Vegas and TCP Westwood. The RTT was set at 100 ms.

In Fig. 9, we plot the cumulative distribution of the ratio of goodput achieved by TCP-RRE against that for the other TCP variants on a pairwise basis. We make three observations: (i) the achieved goodput for TCP-Reno, TCP-CUBIC and TCP Westwood are extremely similar. There are three distinct lines for these three algorithms in Fig. 9, but it is hard to tell them apart. The reason for the similarity is that all three algorithms have similar behavior during slow start, which dominates the duration of the 1 MB data transfer. (ii) TCP Vegas performs relatively poorly and is starved about 30% of the time by the concurrent upload. (iii) TCP-RRE is able to achieve downlink goodput that is between 2 to 4 times of that for the other window-based ACK-clocked TCP variants.

D. Single Download under Normal Conditions

While we have shown that TCP-RRE performs as expected and can improve downstream TCP goodput under poor uplink conditions, we now examine how TCP-RRE compares against other TCP variants under normal conditions. Here, we transferred 10 MB downstream so as to allow the downstream buffer a chance to fill. The downlink bandwidth was varied between 0.5 Mb/s to 8 Mb/s at intervals of 0.5 Mb/s, and the uplink bandwidth was set at a level that is above the threshold levels described in Section III-B. The RTT was set at 100 ms.

In Fig. 10, we plot the average downstream goodput of various TCP variants and note that the achieved downstream goodput are all comparable. A minor observation is that TCP-RRE performs slightly better than the other variants when the downlink bandwidth is high because TCP-RRE does not require several RTTs during slow start to inflate the `cwnd` like the other (ACK-clocked) variants. Instead, it quickly estimates the correct sending rate. This can be clearly seen in Fig. 11 where we plot the time traces of the various single TCP flows against time. We can see that the average goodput of TCP-RRE increases much more rapidly to the steady value than the other TCP variants. We can observe also in the time traces that TCP-Reno and TCP-CUBIC both experience a drop in goodput after about 2 s due to packet losses from buffer overflow. Because SACK was used in our simulation, the sender only had to retransmit the packets lost when the buffer overflowed. Thus, upon reception of the lost packets, the goodput sharply returns to normal.

E. Handling Network Fluctuations

One important design goal of a congestion control algorithm is that it must be able to adapt to changing network conditions promptly and gracefully. To investigate how TCP-RRE handles changes in network conditions, we ran a long-lived TCP-RRE flow with a starting downlink bandwidth of 3 Mb/s and the RTT initially set at 100 ms. The underlying network conditions are changed at various points: (i) at 4 s, the RTT was increased by 50 ms to 150 ms; (ii) at 8 s, the RTT was further increased to 200 ms; (iii) at 15 s, the bandwidth was decreased to 2 Mb/s; (iv) at 20 s, the RTT was restored to the original level of 100 ms; (v) at 24 s, the bandwidth was drastically increased to 5 Mb/s; and finally (vi) at 30 s, the bandwidth was restored to the original value of 3 Mb/s. The resulting trace is shown in

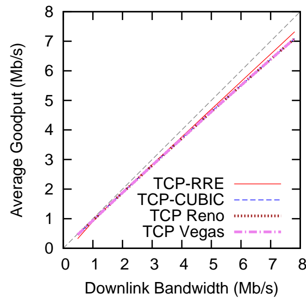

Fig. 10: Plot of average downstream goodput against downstream bandwidth for different TCP variants.

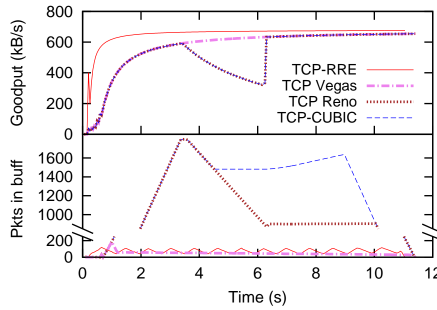

Fig. 11: Sample time traces for different TCP variants.

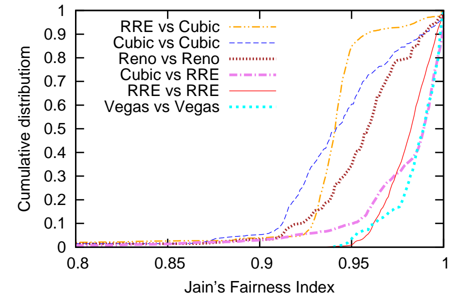

Fig. 12: Jain's fairness index for contending TCP flows.

Fig. 13. We plot also a trace for TCP-CUBIC under the same conditions for comparison.

We see from these traces that TCP-CUBIC has a relatively stable sending rate, but it also keeps buffer occupancy relatively high. While the sending rate for TCP-RRE oscillates, the achieved receive rate is comparable to TCP-CUBIC and also relatively stable. TCP-RRE reacts to the changes rather quickly and typically converges to the correct sending rate within a few seconds. The worst response was at $t = 8$ when the RTT increases a second time. Nevertheless, by $t = 10s$, TCP-RRE has successfully detected the change in the network and adjusted its sending rate accordingly.

F. TCP Friendliness

Next, we investigate how TCP-RRE contends with other TCP flows. To do so, we ran two downstream TCP flows concurrently, with the second flow started with delay after the first. The experiment was repeated with the delay varied between 1 s to 10 s at intervals of 1 s. The rest of the parameters were identical to those in Section III-D.

We then computed the average goodput for each pair of flows and the associated Jain's fairness index [11], i.e. $(R_1 + R_2)^2/(2 \times ({R_1}^2 + {R_2}^2))$, where R_1 and R_2 are the throughput of the two flows. In Fig. 12, we plot the cumulative distribution of the resulting data points. We make two interesting observations: (i) TCP-RRE and TCP Vegas are significantly more fair when contending with the same variant (and achieves a fairness index value consistently above 0.95), compared to TCP-CUBIC and TCP-Reno; (ii) how well TCP-RRE contends with TCP-CUBIC depends on which flow starts first, which explains why there are two lines, one labeled "RRE vs CUBIC" and one labeled "CUBIC vs RRE." Surprisingly, if we start a TCP-RRE flow first, a subsequent TCP-CUBIC flow would aggressively flood the buffer and cause TCP-RRE to back-off and significantly reduce its rate below the "fair" rate. On the other hand, if a TCP-CUBIC flow starts first, a subsequent TCP-RRE flow is able to acquire a reasonably fair share of the available bandwidth.

Our results suggest that TCP-RRE, like other rate-based congestion control algorithms, does not contend well against TCP-CUBIC, which means that TCP-RRE might not be suitable for deployment "in the wild." However, because transparent proxies are commonly deployed in existing mobile ISPs, TCP-RRE can be easily deployed by modifying the mobile-device-facing TCP stacks at such proxies where they would not have to contend with other TCP variants in the core Internet. Surprisingly, we will show in the next section, that our Linux TCP-RRE implementation is often able to achieve better goodput than TCP-CUBIC on existing 3.5G/HSPA networks, even if deployed on a server that is not within a mobile ISP.

IV. Linux Implementation

TCP-RRE was implemented as a kernel module for Linux kernel version 3.2. Because TCP-RRE uses completely different mechanisms from regular TCP congestion control, it cannot be implemented using the Linux pluggable TCP congestion control module. Instead, we inserted hooks into the TCP process flow to intercept incoming ACK packets and outgoing data packets. The receive rate is estimated with every ACK received using an exponentially-weighted moving-average (EWMA) and the sending rate is updated accordingly. Because the send routine in the regular TCP stack is only called when an ACK packet is received, we introduced a timer to clock the sending instead. A short history of bytes sent is maintained to determine the number of packets to be sent at each tick.

In addition, the retransmission routine also has to be modified to handle packet retransmissions, especially after a transmission time-out. This is to give priority to retransmitting lost packets over sending new data. We also added a hook in the handling of SACK packets to extract the information identifying which data sequence was acknowledged. All other TCP functions were left as is. In total, we added slightly less than 1,000 lines of code.

The modified kernel was installed on a server in our lab and we evaluated it over 3.5G/HSPA networks for ISPs A and B. We were unable to test with LTE/HSPA+ because we were not able to obtain any locally available data plans. We ran sets of experiments at various locations, such as in our laboratory, at various residences and at shopping malls, for several hours each. In our experiments, we downloaded 1 MB of data from the server to a mobile phone, and we used two different models of Android phones: HTC Desire (200 kB uplink buffer) and the newer Samsung Galaxy Nexus (1 MB uplink buffer). One set of experiments consists of 4 tests: (i) a single TCP-CUBIC download, (ii) a single TCP-RRE download, (iii) a TCP-CUBIC download with a concurrent TCP upload, and (iv) a TCP-RRE download with a concurrent TCP upload. In the latter two tests, we started the download 10 s after we start

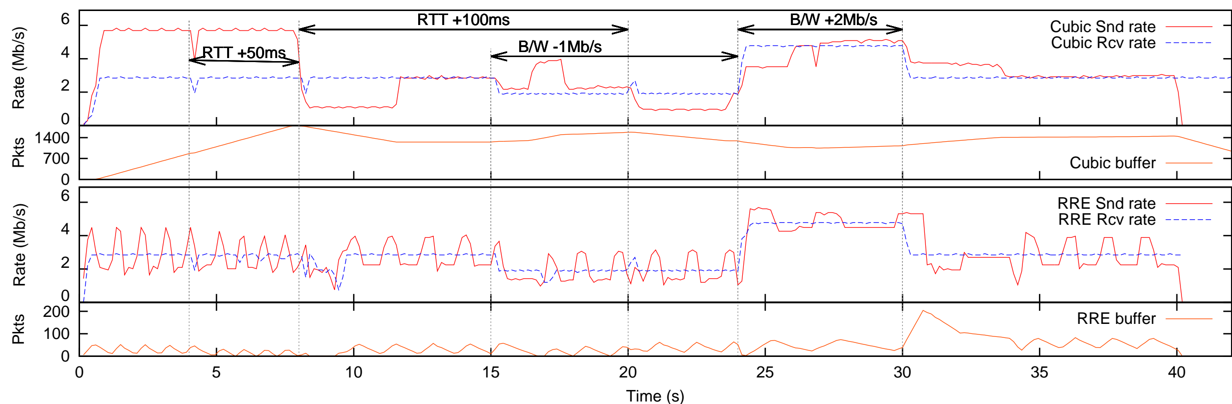

Fig. 13: Time trace comparing how TCP-RRE reacts under changing network conditions to TCP-CUBIC.

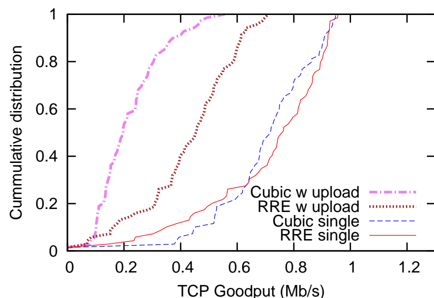

Fig. 14: Cumulative distribution of measured downlink goodput in the laboratory for ISP A on HTC Desire.

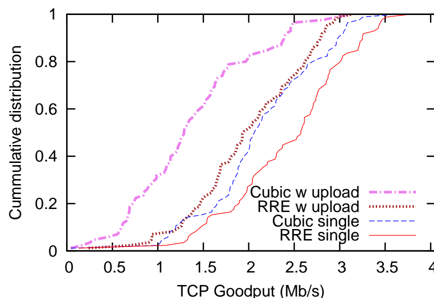

Fig. 15: Cumulative distribution of measured downlink goodput in the laboratory for ISP B with Galaxy Nexus.

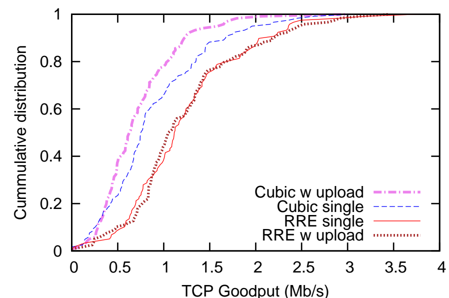

Fig. 16: Cumulative distribution of measured downlink goodput at a residence for ISP B on Galaxy Nexus.

the continuous upload. The experiments were done in sets of 4 tests and each set was run approximately every minute. Since we stayed for several hours at each location, we obtained about 100 to 200 data points for each test at each location. While we collected many sets of data, we are only able to present three sets of data because of space constraints.

In Fig. 14, we plot the results of our experiment carried out in our lab using the older HTC Desire phone over ISP A. The results show that the achieved goodput for a single TCP download for both TCP-CUBIC and TCP-RRE are comparable. However, in the presence of a concurrent upload, the goodput drops significantly, though the drop for TCP-CUBIC is much more significant than that for TCP-RRE. We suspect that this large drop was caused by the combination of a small 200 kB uplink buffer and a relatively high measured uplink throughput of 800 kb/s compared to the download. This combination likely caused the background TCP uplink flow to flood the uplink buffer aggressively causing significant ACK losses and delays for the downlink flow.

In Fig. 15, we plot the results for experiments carried out in the same location, but using the newer Galaxy Nexus phone over ISP B. The downlink speeds were much higher but the uplink throughput was slower, with a median value of 500 kb/s. Finally, in Fig. 16, we plot the results for experiments carried out at a residence over ISP B. The uplink at the residence was even slower, with the median rate below 200 kb/s. While the data presented in Figs. 14 to 16 were specially chosen to illustrate scenarios where TCP-RRE performed better than TCP-CUBIC, we do not mean to suggest that TCP-RRE always performs better. In experiments where the uplink bandwidth was high, the performance of TCP-RRE and TCP-CUBIC were comparable. We note however that in none of our experiments did TCP-RRE perform noticeably worse than TCP-CUBIC.

In summary, what our results for actual 3.5G/HSPA networks suggest is that TCP-RRE is able to improve download throughput under two scenarios: (i) when the uplink bandwidth is low relative to the uplink buffer and (ii) when the uplink buffer is saturated by a concurrent upload. While TCP-RRE was predicted to perform in our simulations some 2 to 4 times faster than TCP-CUBIC under such scenarios, the observed improvements were somewhat smaller in practice. We believe that a plausible explanation for the difference is that the server in our experiments was not located within the ISP and so there were likely losses arising from contention between our TCP-RRE flow and other TCP flows in the core Internet routers.

V. Related Work

In this section, we provide an overview of the prior work in the literature that are related to our work.

TCP congestion control is a well-studied subject and many TCP variants have been proposed [3, 7, 15, 17]. All existing TCP variants follow the basic ACK-clocking mechanism first proposed by Jacobson [10] and differ on how the congestion window `cwnd` is determined and adjusted. TCP Vegas keeps the buffer occupancy low by computing the `cwnd` based on the RTT observed [3]. TCP Westwood+ optimizes for wireless networks by estimating the rate using the observed

RTT [17]. However, since cellular networks are prone to significant delays, TCP Westwood+ is not likely to be able to estimate the required rate accurately. TCP-CUBIC [7] is the default TCP implementation in current Android and iOS mobile devices. The key difference between TCP-RRE and previous TCP variants is that we have done away with the congestion window *cwnd* and ACK clocking, and we regulate the sending rate directly.

The idea of using rate information to control a TCP flow is not new. Keshav showed that the rate control method is preferred when there are rate allocating servers [16]. RATCP is another TCP congestion control using rate information feedback from the receiver [14]. Thus, it requires a modified receiver and is not compatible with TCP, unlike TCP-RRE. TCP-friendly, equation-based congestion control was proposed to regulate UDP flows [6]. The proposed equation is based on the packet lost rate which is very low in cellular networks. TCP Rate-based Pacing (RBP) is a technique to pace out the sending of packets instead of sending a burst of `cwnd` packets to avoid saturating the buffer [15, 20]. None of these techniques can address the ACK delay problem solved by TCP-RRE.

The uplink saturation problem has recently received much attention and shown to be a significant cause of performance degradation in 3.5G/HSPA networks [21]. Heusse et al. showed that in practice, the *Data Pendulum* effect is more prevalent [8] than the classic *ACK compression* problem [13] under such scenarios. Xu et al. proposed a receiver-side algorithm called *Receiver-side Flow Control* (RSFC) that regulates the sending rate of a TCP upload from a mobile device to avoid saturating the uplink buffer. TCP-RRE directly addresses the uplink buffer saturation problem from the sender-side, and in addition, addresses it in a more general way and hence improves download performance even under scenarios where low uplink bandwidths would effectively throttle the downlink.

The bufferbloat problem was also cited in a recent measurement study of 3G/4G networks [12]. While bufferbloat is typically solved by sizing the buffer appropriately [1], Nichols and Jacobson recently revisited this classic problem and proposed CoDel, where packets are dropped according to the time they spend in the buffer [19]. This is similar in spirit to TCP-RRE, which infers the buffer growth by measuring the relative one-way delay. By making the delay oscillate about a constant threshold that is a function of the RTT, TCP-RRE effectively creates a dynamic virtual buffer. Unlike CoDel, which requires specific hardware feedback support and is designed for deployment in routers with FIFO queues, TCP-RRE works end-to-end.

VI. Conclusion

While there have been previous work on rate-based congestion control algorithms for both TCP and UDP, to the best of our knowledge, TCP-RRE is the first attempt at completely eliminating ACK clocking in the TCP stack. When link conditions are good, the performance of TCP-RRE is comparable to that of existing TCP variants. When uplink conditions are poor and there are severe delays in the ACKs within the uplink buffer, TCP-RRE is able to achieve high utilization of the downlink by deducing the receive rate at the sender via passive observation of the TCP timestamps on the ACK packets and setting the sending rate accordingly.

TCP-RRE is compatible with existing TCP implementations and can be easily deployed at existing mobile ISP proxies without requiring any modifications to existing mobile devices. As mobile uploads become more common, and the number of cellular data subscribers continues to increase, we believe TCP-RRE is an useful enhancement for modern cellular data networks.

Acknowledgments

This work was supported by the Singapore Ministry of Education grant T1 251RES1006.

References

[1] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing Router Buffers. In *Proceedings of SIGCOMM '04*, Aug. 2004.

[2] E. Blanton, M. Allman, K. Fall, and L. Wang. A conservative SACK-based loss recovery algorithm for TCP. RFC 3517, April 2003.

[3] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM '94*, Aug. 1994.

[4] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional rate reduction for TCP. In *Proceedings of IMC '11*, Nov. 2011.

[5] 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.

[6] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proceedings of SIGCOMM '00*, Aug. 2000.

[7] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *SIGOPS OSR*, July 2008.

[8] M. Heusse, S. A. Merritt, T. X. Brown, and A. Duda. Two-way TCP connections: Old problem, new insight. *SIGCOMM CCR*, 41(2):5–15, Apr. 2011.

[9] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of MobiSys '10*, June 2010.

[10] V. Jacobson. Congestion avoidance and control. *SIGCOMM CCR*, Aug. 1988.

[11] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer system. DEC Research Report TR-301, Sept. 1984.

[12] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3G/4G networks. In *Proceedings of IMC '12*, Nov. 2012.

[13] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. Improving TCP throughput over two-way asymmetric links: Analysis and solutions. In *Proceedings of SIGMETRICS '98*, June 1998.

[14] 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.

[15] J. Ke and C. Williamson. Towards a rate-based TCP protocol for the web. In *Proceedings of MASCOT '00*, Sept. 2000.

[16] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of SIGCOMM '91*, 1991.

[17] 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.

[18] M. Mathis and J. Mahdavi. TCP rate-halving with bounding parameters. Dec. 1997.

[19] K. Nichols and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20:20–20:34, May 2012.

[20] V. Visweswaraiah and J. Heidemann. Rate based pacing for TCP. http://www.isi.edu/lsam/publications/rate_based_pacing/, 1997.

[21] 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.