Open access • Proceedings Article • DOI:10.1109/ICNP.2012.6459976

Dynamic regulation of mobile 3G/HSPA uplink buffer with Receiver-side Flow Control

Yin Xu, Wai Kay Leong, Ben Leong, Ali Razeen

Institutions: National University of Singapore, Duke University

Published on: 30 Oct 2012 - International Conference on Network Protocols

Topics: Upload and Cellular network

Dynamic Regulation of Mobile 3G/HSPA Uplink Buffer with Receiver-Side Flow Control

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

Ali Razeen
Department of Computer Science
Duke University
alrazeen@cs.duke.edu

***Abstract*—We show that the performance of downloads in a 3G/HSPA mobile network can be significantly degraded by a concurrent upload that saturates the uplink buffer on the mobile device. In particular, we found in some instances that download speeds can be reduced by over an order of magnitude from 2,000 kbps to 100 kbps. To mitigate this problem, we propose a new algorithm called *Receiver-side Flow Control* (RSFC) that regulates the uplink buffer on 3G/HSPA data senders. It uses a feedback loop to monitor the available upload capacity and dynamically adjusts the TCP receiver window (`rwnd`) accordingly. We evaluated RSFC on the 3G/HSPA networks of three different mobile ISPs and show that for one of them, RSFC can improve the download throughput from less than 400 kbps to up to 1,400 kbps. In the presence of a concurrent upload, RSFC can also reduce website load times from more than 2 minutes to less than 1 minute 90% of the time. Our technique is compatible with existing TCP implementations and can easily be deployed at 3G web proxies without requiring any modification to existing mobile devices.**

I. Introduction

The increasing popularity of mobile devices and online social networks has caused simultaneous uploads and downloads to become commonplace in 3G mobile networks. For example, the fans at a recent sports event uploaded 40% more data (such as photos and videos) than they downloaded [4]. It would not therefore be surprising to find users attempting to access websites while photos and video are being uploaded in the background. However, we found in a measurement study that in the presence of a simultaneous background upload, 3G download speeds can be drastically reduced from more than 1,000 kbps to less than 100 kbps. With 3G poised to become even more ubiquitous [8], there is an urgent need to understand and address this performance issue associated with mobile uploads.

Since upload speeds in 3G/HSPA mobile networks are typically lower compared to the download speeds, the downstream ACKs will be queued behind the data packets in the uplink buffer when there are concurrent flows in both directions. The ACKs can sometimes be severely delayed and cause the download speeds to slow to a crawl. While one might be tempted to think that this is a manifestation of the well-known ACK compression problem [26], Heusse et al. recently demonstrated that ACK compression rarely occurs in practice and even if it does, it has little effect on performance [11]. Instead, they show that the degradation in performance is a result of the uplink buffer not being appropriately sized for the available link capacity.

TCP buffer sizing is also a well-studied problem. There is an old rule of thumb that the size of a buffer should be set to the bandwidth-delay product [25] (BDP). More recently, it was found that it should be set to $\mathtt{BDP}/\sqrt{n}$, where n is the number of long lived flows [3]. Unfortunately, these rules cannot be applied to 3G mobile networks directly because such networks exhibit significant spatial and temporal variation. Depending on the ISP, we have found that the available uplink bandwidth can vary by more than an order of magnitude from 30 kbps to 1,600 kbps at a single location. Therefore, to fully utilize the available uplink capacity, we cannot use a fixed buffer size. Instead, the size of the uplink buffer needs to be dynamically adjusted according to the available bandwidth.

In this paper, we describe *Receiver-side Flow Control* (RSFC), a method to dynamically control the uplink buffer of a 3G data sender from the receiver. Our approach uses a feedback loop to continuously estimate the available uplink bandwidth and advertises an appropriate TCP receiver window (`rwnd`). The technique of using `rwnd` to control a TCP flow has been employed in other contexts [2], [15], [24]. However, to the best of our knowledge, we are the first to apply this technique to improve the utilization of a 3G mobile downlink in the presence of concurrent uploads. The key innovation in our algorithm is a method to dynamically adapt to variations in the 3G link capacity.

In this paper, we make two key contributions. First, we demonstrate clearly that in a 3G/HSPA network, a concurrent upload can cause significant degradation to the download performance. Second, we developed RSFC, a new algorithm that can dynamically regulate the utilization of the uplink buffer of a 3G mobile device by actively adjusting the TCP receiver window `rwnd`.

We evaluated RSFC extensively on three 3G mobile ISPs using Android phones and found that RSFC can significantly improve downlink utilization, especially with an ISP with consistently low upload speeds. In our experiments, we found that RSFC can improve download speeds from lower than 400 kbps to up to 1400 kbps. RSFC can also reduce the time taken to load websites in the presence of concurrent uploads from more than 2 minutes to less than 1 minute some 90% of the time. We also show that RSFC is compatible with existing

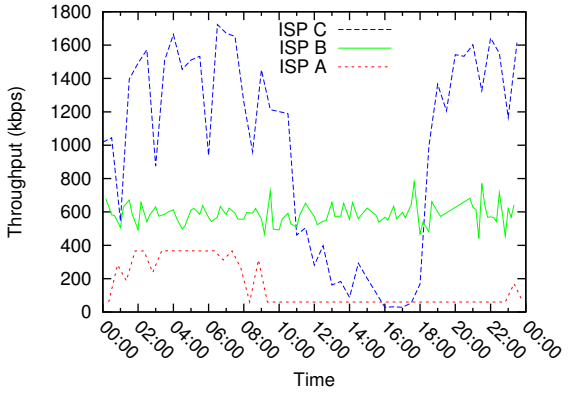


Fig. 1. TCP upstream throughput measured in the lab over 24 hours for the local mobile ISPs A, B and C. Measurements were obtained by uploading 1 MB of data every 15 minutes.

TCP implementations.

The rest of this paper is organized as follows: in Section II, we show the problems caused by a saturated upstream buffer in 3G networks. In Section III, we describe RSFC and explain how it can dynamically regulate the number of packets in the uplink buffer of a mobile device. In Section IV, we evaluate the effectiveness of RSFC with experiments on three 3G/HSPA networks. Finally, we provide an overview of related work in Section V and conclude in Section VI.

II. The Problem of Saturated Uplink

In this section, we demonstrate and explain why a concurrent upload can cause significant degradation to the download throughput. All the experiments were performed in our lab with three local mobile ISPs: A, B, and C. We anonymized the names of the ISPs because we do not want our study to be used as evidence of one ISP's superior performance over the others. The performance of an ISP will vary due to a variety of factors including the location of the mobile devices and the time of 3G access [17]. In Fig. 1, we show that the upload throughput of the three ISPs are quite different in our lab over a 24-hour period, even though the 3G data plans used for all ISPs have the same advertised maximum download and upload throughputs of 7.2 Mbps and 2 Mbps respectively. This turns out to be fortuitous as we were able to obtain a holistic view of the 3G network performance from a single location.

Impact of Saturated Uplink. To verify that a saturated uplink can negatively impact downstream performance, we ran an experiment with three independent sets of TCP flows: (i) a download-only flow (d_0) that downloads 1 MB of data from a server, (ii) an upload-only flow (u_0) that sends 1 MB of data to the server, and (iii) an upload flow (u_1) that continuously sends random data to the server while a concurrent download flow (d_1) downloads 1 MB of data from the server. In a single run, these three sets of flows are run immediately one after another to minimize temporal variations. This experiment was conducted continuously over several days at 15-min intervals.

In Fig. 2, we plot the performance of d_1 against d_0 and observe that a saturated uplink can significantly increase RTTs and reduce the downlink utilization. However, this phenomenon is not observed for all the experiments. We further

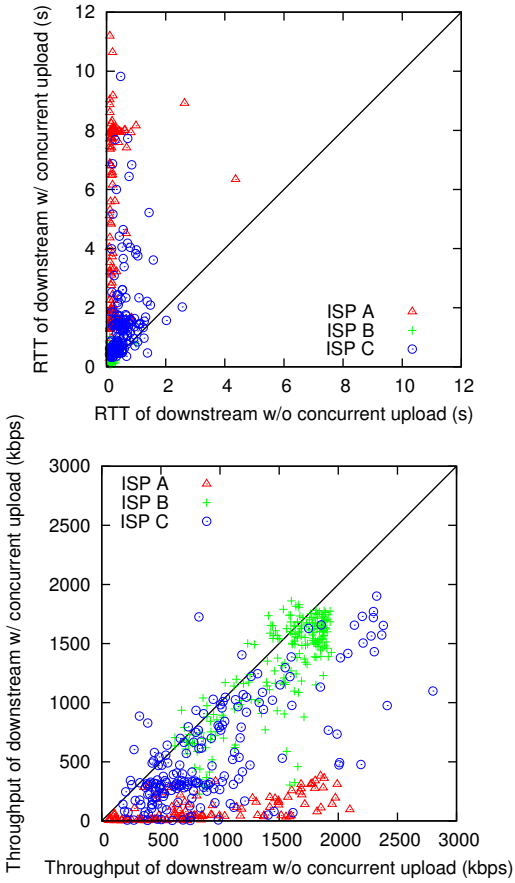
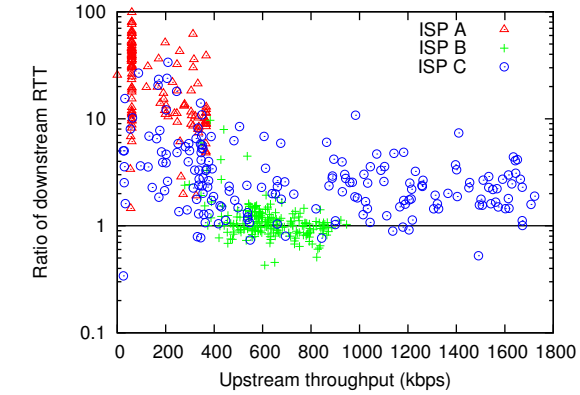


Fig. 2. Comparison of RTT and throughput for downloads with and without uplink saturation.

investigated the relationship between the upload throughput and the downlink performance degradation by plotting the performance ratio, d_1/d_0, against the throughput of u_0 in Fig. 3. We observed that the downstream performance is poorest when the upload speeds are low. For instance, ISP A consistently has low upload throughput and hence the downstream performance is severely degraded. We show later in this section that this poor downlink utilization can partially be explained by ACKs being delayed at the uplink buffer.

We found it interesting that even when an ISP has a consistently high upload speed, as in the case of ISP B, the downlink utilization is still reduced by a small amount. We investigated this further by repeating our experiments with UDP flows. Even with UDP, the throughput of a download flow is degraded by a concurrent upload. Since there are no ACKs for UDP flows, this suggests that mobile downloads are naturally degraded by a concurrent upload, possibly because of ISP-level air time scheduling. As this phenomenon is beyond our control, the focus of this paper is on mitigating the problem of ACK delays, which clearly significantly exacerbates the degradation when upload speeds are low.

Measuring One-Way Delays. To confirm our hypothesis that delayed ACKs is a major reason for the poor downlink utilization, we set up an experiment in a loopback configuration. In this configuration, an Android phone was tethered to a server machine via USB. Next, upload and download TCP



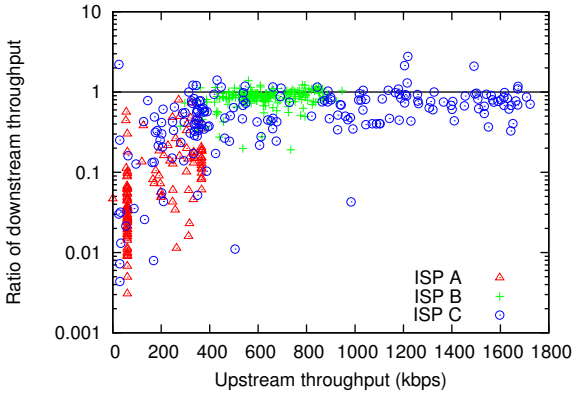


Fig. 3. Plot of ratio of downstream RTT and throughput, with and without upload saturation, against the upload throughput.

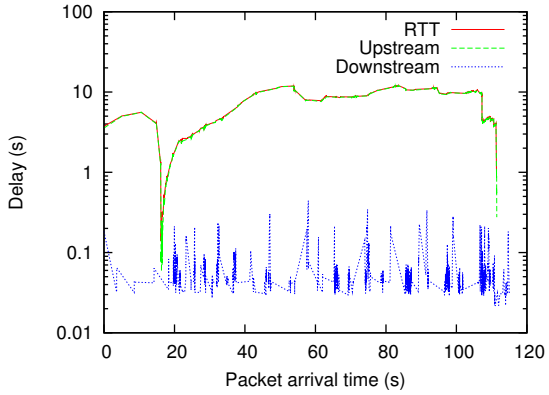


Fig. 4. The breakdown of the downstream RTT into the one-way upstream delay and the one-way downstream delay.

flows were initiated on the server and these flows were routed through the phone's 3G link via the USB connection and back to the server via the wired network. As the server is both the source and destination of all the TCP packets, the timestamps are fully synchronized and we can measure the one-way delay of the downlink (for data packets from the server to the phone), and the one-way delay of the uplink (for ACK packets from the phone to the server). In Fig. 4, we plot the results of this experiment conducted on ISP A. We found that there is significant asymmetry in the delays and that the uplink one-way delay can be up to two orders of magnitude larger than the downlink one-way delay.

III. Receiver-Side Flow Control

In Section II, we showed that the downstream performance can be severely degraded at a mobile sender when the uplink bandwidth is low and the ACKs for the downstream data are delayed in the uplink buffer. One straightforward way to eliminate the delay would be to use a small uplink buffer. However, doing so will cause the uplink to be under-utilized when the available uplink bandwidth increases at a later time. Another possible approach is to control the total amount of data in flight so that the number of packets in the uplink buffer is large enough to fully utilize the uplink capacity and yet not so large as to inflate the uplink one-way delay unnecessarily. In fact, TCP Vegas [7] works in a similar manner by using packet delays to detect impending congestion (arising from packets building up in a buffer) in order to regulate the packet send rate. However, TCP Vegas does not contend well with the common default TCP implementations and it is thus not likely to be adopted as the default TCP implementation for mobile devices anytime soon.

In our approach, which we call *Receiver-side Flow Control* (RSFC), we regulate the number of packets in the uplink buffer of a mobile sender at the receiver by dynamically adjusting the advertised TCP receiver window (`rwnd`). The key challenge is for the TCP receiver to accurately estimate the current uplink capacity and to determine the appropriate `rwnd` to be advertised so that the number of packets in the uplink buffer is kept small without causing the uplink to become under-utilized.

Similar to previous work [16], we estimate delays using TCP timestamps, i.e. using the `TSval` and `TSecr` fields in a received TCP packet. Our approach relies only on the relative differences in the received timestamps and does not require synchronization between the TCP sender and receiver. One caveat is that the granularity of TCP timestamps is not standardized and different devices may increment the timestamps at different rates. In this paper, we work with a granularity of 10 ms on both the mobile sender and the receiver. We expect this issue to be less of a concern in the future because an IETF working group is currently working on the standardization of the TCP timestamp granularity [21].

In the following sections, we first describe how our algorithm works in the general case and then we analyze how the uplink buffer at the mobile sender behaves. Next, we explain how RSFC adapts to changes in the underlying delays of the network. Finally, we briefly discuss how our algorithm can be deployed in practice.

A. RSFC Algorithm

A typical sequence of packets in an upstream TCP flow is illustrated in Fig. 5. At time $t = t_{s_0}$, a packet is sent and it is received at time $t = t_{r_1}$ after a transmission delay t_u. Note that t_s and t_r are recorded with respect to the sender's and receiver's local clocks respectively. The receiver then sends an ACK packet with timestamp fields $\texttt{TSecr} = t_{s_0}$ and $\texttt{TSval} = t_{r_1}$. The sender receives the ACK packet after the transmission delay t_d at time $t = t_{s_2}$ and immediately pushes

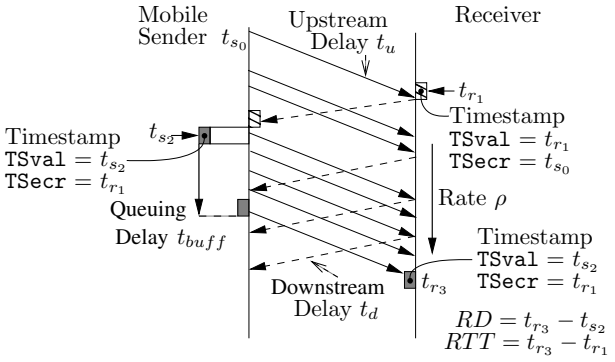


Fig. 5. Packet flow diagram illustrating the various metrics. Solid lines represent data packets, while dotted lines represent ACK packets.

a new data packet into the buffer, with $\texttt{TSecr} = t_{r_1}$ and $\texttt{TSval} = t_{s_2}$. After spending some time t_{buff} in the uplink buffer, the packet is sent and after the transmission delay t_u, the packet is received at time $t = t_{r_3}$. Clearly, there is also a buffer at the downlink, but we do not consider it in our model because as shown in Fig. 4, the downlink one-way delay is negligible and so the amount of time that a packet will spend in the downlink buffer is also negligible.

The receiver tracks the following metrics for each mobile sender as soon as packets are received: (i) the relative one-way delay ($RD = t_{r_3} - t_{s_2}$), (ii) the round-trip time ($RTT = t_{r_3} - t_{r_1}$), and (iii) the rate at which packets are received (ρ). RD and RTT will vary for different packets and the smallest RD and RTT observed are recorded as RD_{min} and RTT_{min} respectively. Clearly,

$$RD = t_{buff} + t_u + t_{offset} \quad (1)$$

$$RTT = t_{buff} + t_u + t_d \quad (2)$$

where t_{offset} is the clock offset between the sender's and the receiver's clocks. If the transmission delays t_u and t_d are constant, it is reasonable to assume that RD_{min} and RTT_{min} are obtained when $t_{buff} \approx 0$, i.e.

$$RD_{min} \approx t_u + t_{offset} \quad (3)$$

$$RTT_{min} \approx t_u + t_d \quad (4)$$

In other words, $RD - RD_{min}$ is a good estimate $\hat{t}_{buff}$, the time that a packet spends in the uplink buffer. Note that we only estimate $\hat{t}_{buff}$ from RD and RD_{min} when there are no packet losses and no re-ordering.

The effective TCP sliding window at the mobile sender is the minimum of the TCP congestion window (`cwnd`) and the `rwnd`. By adjusting the value of `rwnd`, the receiver can cap the growth of the sliding window. Recall that our objective is to regulate the uplink buffer at the mobile sender so that the available capacity is utilized while simultaneously minimizing the uplink delay. Therefore, using this strategy, we advertise an appropriate value of `rwnd` based on the estimated $\hat{t}_{buff}$.

If $\hat{t}_{buff}$ is larger than a threshold T, we say that the system is in the *fast* state because there are too many packets in the uplink buffer. Conversely, if $\hat{t}_{buff} \leq T$, we say the system is in the *slow* state since we can possibly allow more packets to be sent or buffered. It remains for us to determine a value for T, which achieves a good trade-off between delay and link utilization. With a small value of T, we can achieve a lower delay at a higher risk of under-utilizing the upstream link. We evaluated different choices of T and found that setting $T = RTT_{min}$ keeps the RTT below 1 s 80% of the time, while keeping the upload throughput similar to that of TCP Cubic. Reducing T below RTT_{min} will reduce the throughput without much reduction to the RTT. Thus, we set $T = RTT_{min}$ in our implementation.

Fast State. In the fast state, our algorithm will freeze the growth of the TCP buffer to prevent excessive delays, by advertising the `rwnd` as $min(rwnd, \lceil(\rho \times RTT_{min})/MSS\rceil \times MSS)$, where ρ is the rate at which packets are received, MSS is the Maximum Segment Size, and $rwnd$ is the original `rwnd` computed by the kernel. This sets the advertised window to the estimated bandwidth-delay product (BDP) for the transmission. Note that we cannot advertise more than what is originally allowed by the kernel to prevent overflowing the receiver's buffer. Upon receiving the new `rwnd`, the mobile sender will stop queuing more packets and its buffer will start to empty. At some point, $\hat{t}_{buff}$ will drop below T, causing the algorithm to enter the slow state.

Slow State. In the slow state, we want to send more packets to ensure that we fully utilize the available uplink capacity. To do so, we need to increase the advertised `rwnd`. Like TCP slow start, we increase the `rwnd` by one MSS after the receiver receives each data packet. Eventually, the TCP buffer at the sender would begin to fill and $\hat{t}_{buff}$ will start to increase until it exceeds T, and the algorithm returns to the fast state.

B. Maximum Buffer Utilization

As the algorithm oscillates between the fast and slow states, the number of packets in the uplink buffer would fluctuate over time. Hence, we need to ask and answer the following question: what is the maximum time that we expect a packet to stay in the uplink buffer? The answer will allow us understand the effect of selecting different values for the threshold T, and yield important insight into how RSFC adapts to changing network conditions. In Fig. 6, we illustrate the change in the number of packets in the uplink buffer over time.

Suppose that the mobile sender starts in the slow state (i.e. $\hat{t}_{buff} < T$). The receiver will continuously increase the advertised `rwnd` by one MSS for every data packet received. The sender will add more packets to the buffer and at some point, it will add a packet that needs to spend a time $t_{buff} = T$ in the buffer before it is transmitted (See shaded packet in Fig. 6). As this shaded packet moves to the head of the buffer, the sender will continue to receive ACKs for the data packets sent earlier, and the uplink buffer will continue to grow by one MSS for each ACK. When the shaded packet reaches the head of the buffer, by definition, the buffer queue would have reached a point such that the next packet queued will have a $t_{buff} = 2T$, since it took the shaded packet an amount of time equal to T to reach the head.

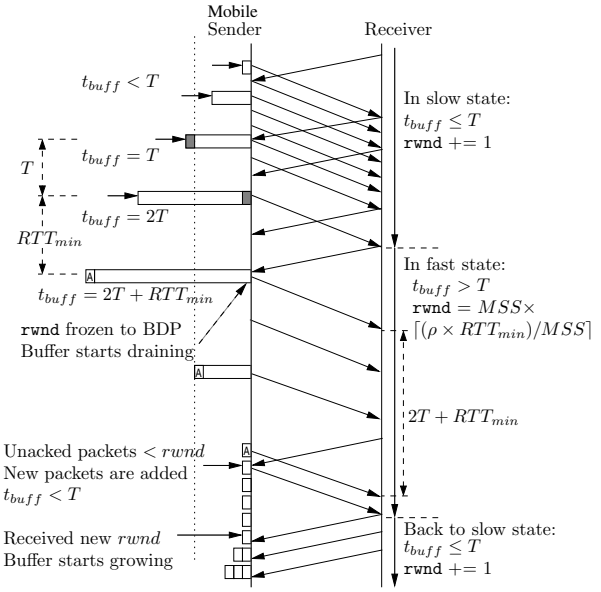


Fig. 6. Packet flow diagram illustrating a typical scenario for buffer inflation.

Once the receiver receives the shaded packet, it switches to the fast state and freezes the advertised rwnd. However, it will take a roundtrip time of RTT_{min} from the time of transmission of the shaded packet for this frozen rwnd to reach the sender. Therefore, until the new rwnd is received, the sender would keep queuing new packets as it receives ACKs with increasing rwnd. When the new rwnd is finally received, the buffer queue would be such that the next packet queued (see packet A in Fig. 6) will have $t_{buff} = 2T + RTT_{min}$. This will be the maximum buffer delay because the rwnd is now capped at the estimated BDP $\lceil(\rho \times RTT_{min})/MSS\rceil \times MSS$. Since we set T as RTT_{min} in our implementation, the largest expected buffer delay t_{buff} is $3 \times RTT_{min}$.

At this point, the uplink buffer will start to drain and a new packet will only be pushed into the uplink buffer when the number of unacknowledged packets drops below rwnd. As rwnd is set to the BDP, which is the number of packets in flight, the buffer will eventually be completely emptied. When this happens, the new packet sent will spend a time $t_{buff} = 0 < T$ in the buffer. Hence, when the receiver receives this new packet, the algorithm will revert to the slow state and start increasing the rwnd by one MSS for each packet. Once again, it will take a time of RTT_{min} from the transmission of the packet for the new rwnd to reach the sender. Hence, in a typical scenario, the algorithm will spend at most $2T + 2RTT_{min}$ in the fast state before going back to the slow state.

C. Handling Changes in the Network

Because the 3G uplink is a shared resource, the base station will allocate an amount of air time to each mobile device depending on factors such as signal quality and the number of connected devices. Hence, the available bandwidth and associated network delays are expected to change over time. In other words, we expect the variables ρ, t_d and t_u to vary over time and RSFC needs to adapt to these changes.

If there is a change only in the available bandwidth that results in a change of the receive rate ρ, there will not be any impact on the algorithm because ρ is only used to estimate the BDP in the fast state. When ρ changes, we simply update our estimate of the BDP with the new ρ. Also, the switching between the fast and slow states is independent of ρ and depends only on the delays t_u and t_d.

Similarly, if there is a decrease in network delays t_u and t_d, no special handling is required. A decrease in either t_u or t_d will cause RD or RTT to drop lower than the previously recorded values of RD_{min} or RTT_{min}, and these two variables will simply be updated. Even if we do not update RTT_{min} accordingly, the older (and larger) value of RTT_{min} will simply result in an over-estimation of the BDP. This means that the rwnd set in fast state will not allow the buffer to empty completely and $\hat{t}_{buff}$ will not be reduced to zero. As long as $\hat{t}_{buff}$ remains small, the uplink is still fully utilized and rwnd is still capped, there will be an upper bound on the delay.

The challenge arises when either of the network delays, t_u or t_d, increase. This will eventually cause the algorithm to enter the fast state, since the increasing rwnd in the slow state will cause the sender to queue packets until the $\hat{t}_{buff} > T$. In the fast state, the BDP will be under-estimated because the estimated $RTT_{min} < t_u + t_d$. This causes rwnd to be set at a value lower than the actual capacity of the link. The buffer will eventually empty, yet the low rwnd prevents new packets from being sent, which causes the link to be under-utilized. In other words, an increase in t_u might be interpreted as an increase in $\hat{t}_{buff}$, which means that $\hat{t}_{buff}$ can never fall to zero. There are two possible situations:

(i) *Enters Slow State*. When the increase in t_u is not large and $\hat{t}_{buff} \leq T$, rwnd will start increasing, allowing the sender to send more packets for every ACK received. Since the link was previously under-utilized, this will result in ρ increasing at the receiver. Eventually the algorithm will return to the fast state. Hence, if we detect an increase in ρ in the most recent cycle of slow and fast states, we can safely assume that link utilization had dropped at some point and we update (increase) RD_{min} and RTT_{min} to the minimum RD and RTT values observed in the most recent cycle of slow and fast states.

(ii) *Stuck in Fast State*. If the increase in t_u is sufficiently large such that $\hat{t}_{buff} > T$, the algorithm will end up getting stuck in the fast state and the link will always be under-utilized. We showed earlier in Section III-B that the algorithm will typically spend at most $2T+2RTT_{min}$ in the fast state before reverting to the slow state. Hence, we can deduce that something is wrong if the algorithm spends significantly longer than $2T + 2RTT_{min}$ in the fast state. In this light, if we find that the algorithm stays in the fast state for longer than $2T + 3RTT_{min}$, we will switch to a third special *monitor* state. Essentially, we add an extra RTT_{min} to provision for possible transient changes in delays.

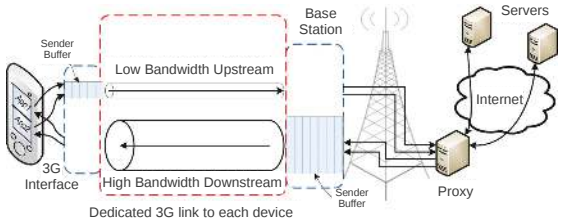


Fig. 7. The bottleneck 3G link is virtually dedicated to each device. Multiplexing is done by the ISP in a schedule which is assumed to be fair.

Monitor State. Like in the slow state, `rwnd` is increased by 1 for every data packet received and we monitor ρ. There are two possible scenarios:

(a) If an increase is detected in ρ, we switch to the slow state. Simultaneously, RD_{min} and RTT_{min} is updated to the minimum RD and RTT observed while in the monitor state and `rwnd` is updated to $\lceil(\rho \times RTT_{min})/MSS\rceil \times MSS$ accordingly with the new value of RTT_{min}.

(b) If ρ does not seem to increase even after RD increases by T, it suggests that the link is still fully utilized, i.e. there are still packets in the uplink buffer. We then halve both RD_{min} and RTT_{min} and switch to the fast state. This will likely force the buffer to empty and cause RSFC to switch back to monitor state. However, this time in the monitor state, we will likely end up in case (a) above and both RD_{min} and RTT_{min} will be set to the correct values.

We demonstrate in Section IV-E that the monitor state is crucial for RSFC to achieve good link utilization in the presence of network variations.

D. Practical Deployment

RSFC requires minor modifications to the TCP stack at the TCP receiver, but *no modification* needs to be made to an existing mobile device, though it does require the TCP timestamp option to be enabled. A quick survey of the available smartphones suggests that the TCP timestamp option is enabled by default for both Android and iPhone (which together constitute about three quarters of the global smartphone market [9]) and disabled by default for Windows Mobile phones. In this light, we believe that the majority of smartphones are RSFC-ready at present.

The current architecture of existing 3G mobile networks makes it relatively straightforward to deploy our algorithm. In particular, we found that all three local mobile ISPs implement a transparent web proxy that intercepts all HTTP connections, and effectively converts them into split TCP [18] connections. These proxies are used to improve 3G performance by caching commonly accessed web content (such as images on popular websites) and in some cases, to perform QoS filtering on the traffic. The current situation suggests we can deploy RSFC for an entire 3G network easily by modifying the TCP stack on these proxies as illustrated in Fig. 7. Unfortunately, this also suggests that it would not be helpful to deploy RSFC on individual servers (unless the connections are not made on the HTTP port) because the mobile connections would be likely be routed through a web proxy and hence RSFC would not be able to have any direct effect on the mobile device.

IV. Performance Evaluation

In this section, we present our evaluation results for RSFC. We begin by investigating RSFC's effectiveness at reducing the RTT and in improving throughput. Next, we evaluate how RSFC performs in two possible application scenarios: (i) when users surf the web while there is a concurrent background upload, and (ii) when there are simultaneous uploads. Finally, we demonstrate the necessity of having to adapt to changing network conditions and also show that RSFC is compatible with other TCP congestion control algorithms.

All of the experiments in this section were conducted in our lab, on Android phones, on all three local ISPs. The congestion control algorithm used on the phones is TCP Cubic [10], which is the default TCP implementation in the Android kernel. It is clear from our measurement results in Section II that ISP A has the poorest upload performance in our lab. As to be expected, RSFC achieves the greatest improvement in performance with ISP A's network. For ISPs B and C, where the upload speeds are relatively high, RSFC was less effective, though it does not perform worse than the default TCP Cubic. Because our goal is to show that RSFC can significantly improve downloads when mobile connectivity is less than ideal (and almost all mobile users will find themselves at such locations every once in a while), and because of space constraints, we present only the results for ISP A, except where stated otherwise.

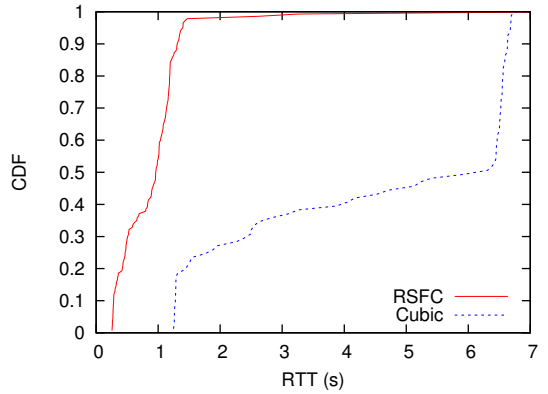
A. Reduction in RTT

To verify that RSFC can reduce the RTT for a TCP upload, we ran an experiment where we uploaded 1 MB of data from the phone using TCP Cubic and then repeated the process with RSFC. This experiment was repeated periodically every 15 min over several days and the CDF of our results is shown in Fig. 8. We observe that in general, using RSFC results in much lower RTT compared to TCP Cubic. TCP Cubic's RTT is greater than 6 s 50% of the time while the RTT with RSFC is close to 1 s more than 90% of the time. It is also apparent that the upload throughput achieved by RSFC is almost identical to that for TCP Cubic. This simple experiment shows that RSFC can achieve a significant reduction in RTT without suffering any loss in upload throughput.

B. Improving Downstream Throughput

Next, we measured the improvement in downlink utilization achieved in the presence of a concurrent upload by running the following sets of experiments: (i) a single TCP Cubic download flow (d_0), (ii) a single TCP Cubic upload flow (u_0), (iii) a TCP Cubic download flow with a concurrent TCP Cubic upload flow (d_1 and u_1), and (iv) a TCP Cubic download flow with a concurrent RSFC upload flow (d_2 and u_2). These experiments were run one after the other to minimize the effect of temporal variance. In Fig. 9, we plot the CDF of the throughputs achieved.

As expected, the throughput of d_1 is poor. It is lower than 400 kbps all the time and it is close to 10 kbps 30% of the time. With RSFC, the downstream throughput improves significantly and 50% of the time, it is greater than 400 kbps. When we



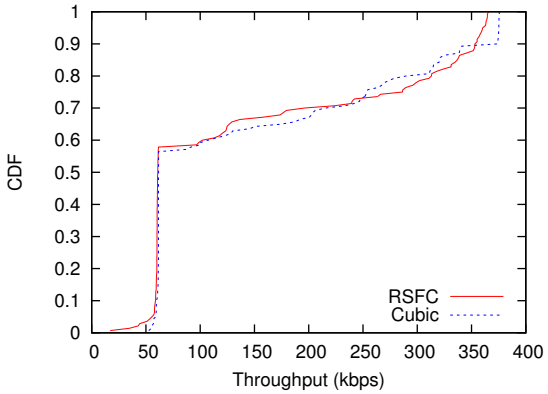
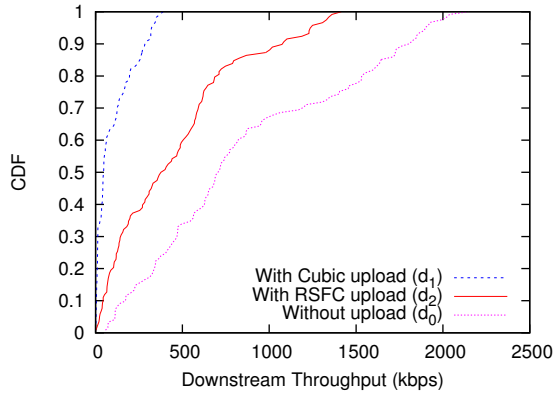


Fig. 8. CDF of RTT and throughput for TCP Cubic and RSFC uploads.



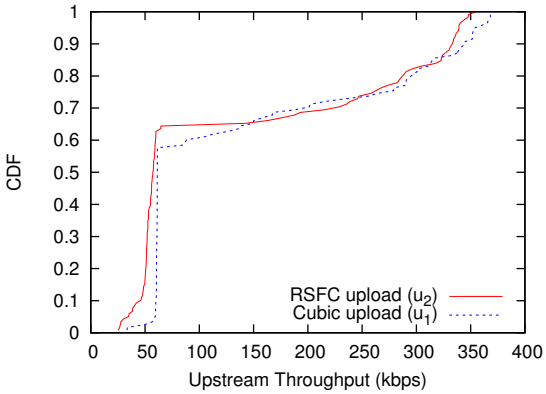


Fig. 9. CDF of the throughput achieved by the downstream and upstream flows under different conditions.

compare the throughputs of u_2 and u_1, we found that they are similar in spite of the increase in downlink utilization for RSFC. This suggests that our approach of regulating the uplink buffer size is effective; the buffer always has packets to send but the packets are not unnecessarily delayed. However, the throughput of d_2 is still not as good as the d_0 benchmark. As discussed in Section II, this is likely due to the interactions between the concurrent flows arising from scheduling at the 3G layer, which is beyond our control.

We also investigated how the current uplink capacity will affect the improvement in the downlink utilization. In Fig. 10, we compare RSFC to TCP Cubic by plotting the throughput ratio d_2/d_1 against u_0. We included the data points from all three ISPs for a better overview. It is clear that RSFC achieves the greatest improvement when the upload throughput is below 400 kbps. Fig. 10 also shows that RSFC does not achieve much improvements for ISPs B and C. These two ISPs typically have higher uplink capacity, which makes it more unlikely for the uplink buffer to be saturated. In such cases, RSFC has a similar performance as TCP Cubic. Note that our experiments were conducted at a fixed location where ISPs B and C seem to have significantly better performance than ISP A. However, they could have lower upload speeds at other locations and in those instances, RSFC would be helpful for ISPs B and C as well.

C. Improving Web Surfing

Next, we investigate how RSFC performs in a common scenario: surfing the web while uploading data in the background.

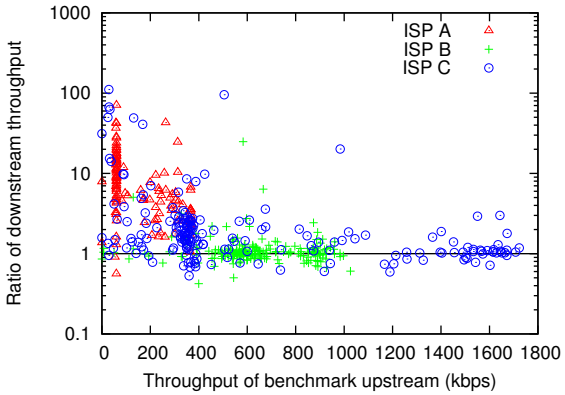


Fig. 10. Plot of ratio between RSFC's downstream throughput to that of TCP Cubic against the throughput of the benchmark upstream flow.

In this experiment, we visited the Alexa Top 100 websites [1] under three conditions: (i) with a concurrent RSFC upload, (ii) with a concurrent TCP Cubic upload, and (iii) without any concurrent uploads. In Fig. 11, we plot the CDF of the time taken to receive the last byte of the last HTTP response. Without a concurrent upload, 90% of these websites take less than 30 s to load. However, with a TCP Cubic upload, 70% of them can take more than 2 mins to load. Such performance degradation will severely impact user-perceived performance of the mobile web access. RSFC mitigates the impact of a concurrent upload as it reduce the load time to within 30 s for 70% of the websites and to within 1 min for 90% of them.

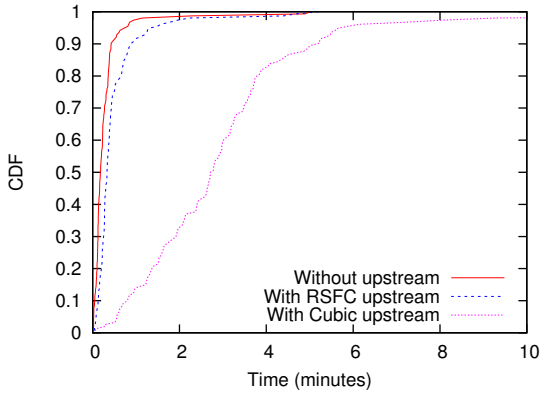


Fig. 11. CDF of the time taken to load the top 100 websites under different conditions.

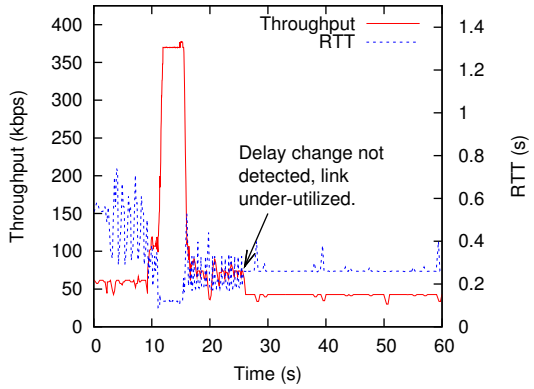


Fig. 13. Plot of the average throughput achieved and RTT using RSFC variant without RD_{min} and RTT_{min} update mechanism.

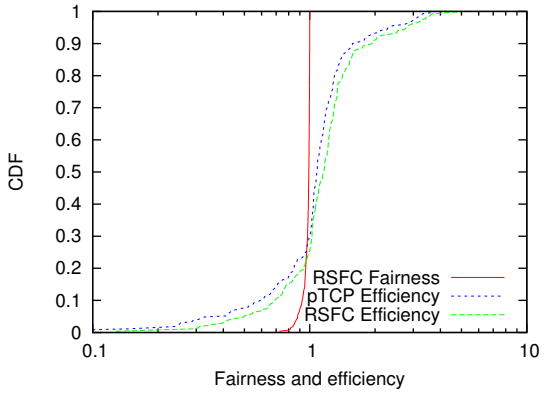


Fig. 12. CDF of the fairness between two RSFC uploads and the efficiency of two RSFC uploads compared to a single TCP Cubic upload.

D. Fairness of Competing RSFC Uploads

If there are multiple upload flows from a single 3G connection, RSFC will be applied to each flow independently. The rate estimation and `rwnd` advertisement for a flow would be done without considering the other flows. We attempt to understand how well this simple scheme will perform in practice by running the following experiment. We first upload 1 MB of data using a single TCP Cubic flow as a reference. After the upload is complete, we start two concurrent RSFC upload flows from the phone and upload 1 MB of data in total. This experiment was repeatedly executed over a 24-hour period.

We quantified the fairness of the throughput utilization between the two RSFC flows using the Jain fairness index [13], 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. We also compared the efficiency of the throughput utilization for the RSFC flows to the TCP Cubic flow by computing the ratio $(R_1 + R_2)/C$, where C is the throughput of the TCP Cubic flow. A CDF of our findings is shown in Fig. 12.

We found that the two RSFC flows achieve very similar throughput. It turns out that the oscillating nature of our algorithm, between the fast and slow states, ensures that neither flow dominates the uplink. Hence, we conclude that RSFC flows are relatively fair when they contend with each other. In terms of utilization efficiency, the two concurrent RSFC flows seem to perform no worse than a single TCP Cubic flow about 80% of the time and can occasionally achieve better throughput since parallel TCP flows can generally better utilize the link capacity compared to a single flow.

For the remaining 20% of the time, the two RSFC flows seemed to perform worse than a single TCP Cubic flow. To further investigate why this might be the case, we ran a benchmark experiment where we compared the efficiency of two parallel TCP Cubic upload flows against a single TCP Cubic flow. The results of this experiment is labeled in Fig. 12 as "pTCP Efficiency". Since this new curve was remarkably similar to the curve for the parallel RSFC flows, it suggests that it is likely that the two RSFC flows performed worse 20% of the time because of temporal variations in the 3G link.

E. Adapting to Changing Network Conditions

In Section III-C, we described how RSFC updates RD_{min} and RTT_{min} to handle variations of the 3G link capacity to maintain good link utilization. To evaluate the necessity of updating RD_{min} and RTT_{min}, we ran an experiment where we uploaded data for 60 s using a variant of RSFC that keeps the values to the lowest measured from the start of the flow. We plot the resulting throughput and RTT achieved in Fig. 13.

At the start of the flow, the estimated RD_{min} and RTT_{min} are accurate and the uplink is fully utilized. There are network fluctuations in the middle of the flow and after 25 s, the throughput is lower than that achieved during the first 10 s even though the network conditions for both periods are similar. This is because the underlying network delays decreased then subsequently increased, but their estimates RD_{min} and RTT_{min} were not updated. Hence, the `rwnd` advertised is smaller than the ideal value and the link is under-utilized.

We repeated this experiment with the default version of RSFC and our results are shown in Fig. 14. We can infer from the oscillations in the RTT that the feedback mechanism of RSFC is operating correctly even though like before, there was a temporary decrease in the network delay at 5 s and a subsequent increase at 20 s. This demonstrates that the mechanism to update RD_{min} and RTT_{min} is effective and necessary to adapt to changing network conditions.

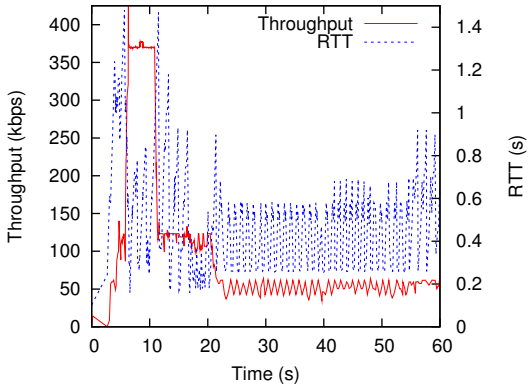


Fig. 14. Plot of the average throughput achieved and RTT using full RSFC algorithm.

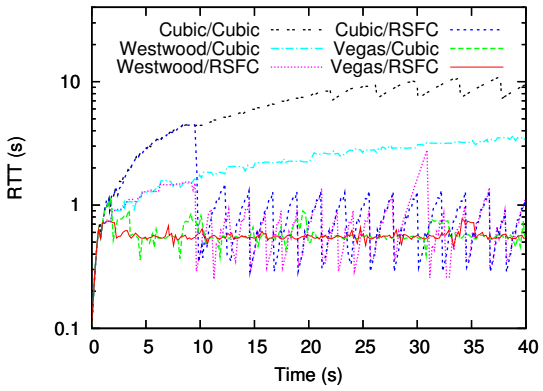


Fig. 15. Plot of the RTT for the transfer of 1 MB file using different TCP variants at both sender and receiver side. In the legend, we indicate first the mobile sender followed by the receiver.

F. Compatibility with other TCP variants

We investigated RSFC's compatibility with other sender-side TCP congestion control algorithms by running a new set of experiments. In these experiments, we uploaded 1 MB of data from the phone to a receiver and alternated the TCP implementation on the phone between Westwood [19], Vegas [7], New Reno [12] and TCP Cubic. We also varied the receiver-side algorithm on the server between TCP Cubic and RSFC.

In Fig. 15, we plot the RTT achieved in the different experiments (the results of New Reno at the sender is not shown because it is similar to TCP Cubic). We see that if the receiver uses RSFC, then the RTT can be improved regardless of the algorithm used at the sender. We also ran the simultaneous upload and download experiments described in Section IV-B and plot our results in Fig. 16. Once again, as long as RSFC is enabled at the receiver, the downlink utilization can be improved. This shows that RSFC is compatible with other TCP variants.

We noticed that the performance of Vegas/Cubic is slightly better than that for Cubic/RSFC. This is to be expected as Vegas is more aggressive in controlling the delays compared to RSFC. Moreover, TCP Vegas can immediately effect changes on the sender whereas RSFC's changes are delayed by at least

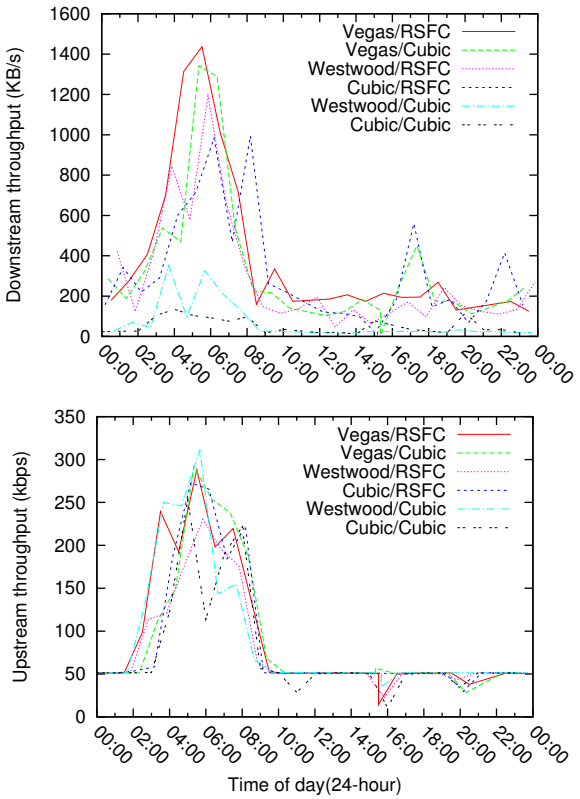


Fig. 16. Plot of downstream throughput when the upstream is saturated with different algorithms over a 24-hour period. In the legend, we indicate first the mobile sender followed by the receiver.

one RTT_{min}. However, RSFC is fully compatible with TCP Vegas and is a compelling alternative to deploying TCP Vegas on mobile devices.

V. Related Work

In this section, we discuss prior work in the literature that are related to our work.

Saturated Uplinks. The impact of saturated uplink buffers on download performance is a well-studied problem, and it was first characterized as the *ACK compression* problem. When upload speeds are low, the downstream TCP ACKs get compressed in the buffer and are sent out in bursts, causing the self-clocking mechanism in TCP to break [14], [26]. However, more recently, Heusse et al. showed that in practice, the *Data Pendulum* effect is more prevalent than ACK compression [11]. According to them, when there are concurrent upload and download connections, it is possible for the buffers on both sides of the connections take turns to fill up and fully utilize their links while the other idles, provided the buffer sizes are configured correctly. However, when the buffer sizes are misconfigured relative to the link capacities, the link with the lower capacity will become the sole bottleneck. We have verified that indeed, the uplink can become a bottleneck and cause the downlink to be under-utilized in 3G networks.

Previous Approaches. Two previous proposals for mitigating the problem of under-utilized downlinks are prioritizing the ACKs and optimizing how the ACKs are sent [5], [6],

and using separate queues for ACKs and data packets [14], [20]. There are also some sender side congestion control algorithms that are designed to achieve low delay, like Vegas [7] and LEDBAT [22]. These solutions all require client-side modifications. Another possible solution is to use parallel TCP connections [23]. However, we verified that parallel TCP flows are not sufficient to improve link utilization because a slow uplink will ultimately delay the ACKs and become the bottleneck.

Receiver-side Congestion Control. The technique of controlling the advertised receive window, `rwnd`, to regulate a TCP flow is not new. Neil et al. used this technique to improve the performance of interactive network applications that are competing with bulk-transfer connections for bandwidth [24]; The explicit window adaptation scheme proposed by Lampros et al. also uses `rwnd` to control the downstream queue size to achieve fairness between window-based and rate-based congestion control algorithm [15]. Andrew et al. used it to fairly share the available bandwidth between users [2]. We solve a different problem from these previous proposals and our goal is to improve downlink utilization in 3G networks by controlling the uplink delay.

TCP Buffer Management. The classic rule of thumb is to set the buffer size to the bandwidth-delay product ($\overline{RTT} \times C$) [25] and more recently, it was suggested that it should be sufficient to set the buffer size to $(\overline{RTT} \times C)/\sqrt{n}$ [3], where C is the data rate of the connection and n is the number of long-lived flows. A fixed-sized approach to buffer sizing is however not feasible in a 3G environment, since unlike routers which typically have fixed throughput, there can be significant variations in the link speed of over an order of magnitude.

VI. Conclusion

We have shown that the performance of a TCP download in a 3G mobile network can be significantly degraded by a simultaneous TCP upload and that *Receiver-side Flow Control* (RSFC) is effective at mitigating this problem, especially in situations where mobile connectivity is poor. RSFC is compatible with existing sender-side TCP congestion control algorithms, and is a practical technique that can be easily deployed at 3G web proxies without requiring any modification of the existing mobile devices. Given that simultaneous upload and downloads are likely going to become more common in 3G networks, we believe that RSFC is an important mechanism for improving the user-perceived performance of the Internet for mobile devices, even though its benefits may seem somewhat indirect.

Acknowledgments

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

References

[1] Alexa. Top Global Sites. http://www.alexa.com/topsites. Accessed March 2012.

[2] L. L. Andrew, S. V. Hanly, and R. G. Mukhtar. Active Queue Management for Fair Resource Allocation in Wireless Networks. *IEEE Transactions on Mobile Computing*, 7(2):231–246, February 2008.

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

[4] AT&T. A Different Take on the Big Game - Stats from the Stands. http://www.attinnovationspace.com/innovation/story/a7780988, February 2012. Accessed September 2012.

[5] H. Balakrishnan, V. N. Padmanabhan, G. Fairhurst, and M. Sooriyabandara. TCP Performance Implications of Network Path Asymmetry. RFC 3449 (Best Current Practice), December 2002.

[6] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The Effects of Asymmetry on TCP Performance. *ACM Mobile Networks and Applications*, 4(3):219–241, October 1999.

[7] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE JSAC*, October 1995.

[8] Forex Pros. Qualcomm: Much to Like Behind the Numbers. http://www.forexpros.com/analysis/qualcomm-much-to-like-behind-the-numbers-121093, April 2012. Accessed September 2012.

[9] Gartner. Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth. http://www.gartner.com/it/page.jsp?id=1924314, February 2012. Accessed September 2012.

[10] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Operating Systems Review*, July 2008.

[11] M. Heusse, S. A. Merritt, T. X. Brown, and A. Duda. Two-way TCP Connections: Old Problem, New Insight. *ACM Computer Communications Review*, 41(2):5–15, April 2011.

[12] J. C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *Proceedings of SIGCOMM '96*, August 1996.

[13] 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, September 1984.

[14] 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.

[15] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. Explicit Window Adoption: A Method to Enhance TCP Performance. *IEEE/ACM Transactions on Networking*, 10(3):338–350, June 2002.

[16] M. Kühlewind and B. Briscoe. Chirping for Congestion Control - Implementation Feasibility. In *Proceedings of PFLDNeT '10*, November 2010.

[17] X. Liu, A. Sridharan, S. Machiraju, M. Seshadri, and H. Zang. Experiences in a 3G Network: Interplay between the Wireless Channel and Applications. In *Proceedings of MobiCom '08*, September 2008.

[18] D. A. Maltz and P. Bhagwat. TCP splicing for application layer proxy performance. *Journal of High Speed Networks*, 8(3):225–240, 1999.

[19] 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.

[20] M. Podlesny and C. Williamson. Improving TCP Performance in Residential Broadband Networks: a Simple and Deployable Approach. *SIGCOMM Computer Communications Review*, 42(1):61–68, January 2012.

[21] R. Scheffenegger and M. Kuehlewind. Additional Negotiation in the TCP Timestamp Option Field during the TCP Handshake. IETF Working Draft, October 2011.

[22] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). IETF Working Draft, October 2011.

[23] H. Sivakumar, S. Bailey, and R. Grossman. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. In *Proceedings of SC '00*, November 2000.

[24] N. T. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. Bershad. Receiver Based Management of Low Bandwidth Access Links. In *Proceedings of IEEE INFOCOM '00*, March 2000.

[25] C. Villamizar and C. Song. High Performance TCP in ANSNET. *SIGCOMM Computer Communications Review*, 24(5):45–60, October 1994.

[26] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proceedings of SIGCOMM '91*, September 1991.