A Rate-based TCP Congestion Control Framework for Cellular Data Networks

LEONG WAI KAY
B.Comp. (Hons.), NUS

A THESIS SUBMITTED

FOR THE DEGREE OF PH.D. IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2014

Acknowledgement

First and above all, I thank and praise almighty God, for providing me the opportunity and the capability to accomplish everything. This thesis would also not have been possible without the help and influence of people in my life.

I want to express my thanks and gratitude to my supervisor, Prof. Ben Leong, for his guidance and mentoring through my graduate studies and research. His patience, belief and support in me has taught me many valuable lessons in life's journey.

I would like to acknowledge my friends and collaborators: Yin Xu, Wei Wang, Qiang Wang, Zixiao Wang, Daryl Seah, Ali Razeen and Aditya Kulkarni. Thank you for all the long nights spent together performing experiments and writing papers. I am glad to have been a part of your research as well as sharing your graduate life experiences.

Special thanks also to my wife, Nicky Tay, the most beautiful woman in the world for her 100% support in my work. She is a pillar of strength and encouragement during trying times and gives me the assurance I need to carry on.

Thanks also to my family, friends and carecell for all your prayers and support.

Publications

- Wei Wang, Qiang Wang, Wai Kay Leong, Ben Leong, and Yi Li. "Uncovering a Hidden Wireless Menace: Interference from 802.11x MAC Acknowledgment Frames." In *Proceedings of the 11th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON 2014)*. Jun. 2014.

- Yin Xu, Zixiao Wang, Wai Kay Leong, and Ben Leong. "An End-to-End Measurement Study of Modern Cellular Data Networks." In *Proceedings of the 15th Passive and Active Measurement Conference (PAM 2014)*. Mar. 2014.

- Wai Kay Leong, Aditya Kulkarni, Yin Xu and Ben Leong. "Unveiling the Hidden Dangers of Public IP Addresses in 4G/LTE Cellular Data Networks." In *Proceedings of the 15th International Workshop on Mobile Computing Systems and Applications (HotMobile 2014)*. Feb. 2014.

- Wei Wang, Raj Joshi, Aditya Kulkarni, Wai Kay Leong and Ben Leong. "Feasibility study of mobile phone WiFi detection in aerial search and rescue operations." In *Proceedings of the 4th ACM Asia-Pacific Workshop on Systems (APSys 2013)*. Oct. 2013.

- Wai Kay Leong, Yin Xu, Ben Leong and Zixiao Wang. "Mitigating Egregious ACK Delays in Cellular Data Networks by Eliminating TCP ACK Clocking." In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP 2013)*, Oct. 2013.

- Yin Xu, Wai Kay Leong, Ben Leong, and Ali Razeen. "Dynamic Regulation of Mobile 3G/HSPA Uplink Buffer with Receiver-side Flow Control." In *Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP 2012)*, Oct. 2012.

- Daryl Seah, Wai Kay Leong, Qingwei Yang, Ben Leong, and Ali Razeen. "Peer NAT Proxies for Peer-to-Peer Games". In *Proceedings of the*

8th Annual Workshop on Network and Systems Support for Games (NetGames 2009). Nov. 2009.

- Ioana Cutcutache, Thi Thanh Nga Dang, Wai Kay Leong, Shanshan Liu, Kathy Dang Nguyen, Linh Thi Xuan Phan, Joon Edward Sim, Zhenxin Sun, Teck Bok Tok, Lin Xu, Francis Eng Hock Tay and Weng-Fai Wong. "BSN Simulator: Optimizing Application Using System Level Simulation." In *Proceedings of the Sixth International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2009)*, Jun. 2009.

Abstract

Modern 3G/4G cellular data networks have vastly different characteristics from other wireless networks such as Wi-Fi networks. It is also becoming more pervasive with the reducing cost of smartphones and cellular data plans. In this thesis, we investigate the major issues of cellular data networks and propose a radical TCP congestion control mechanism to overcome these problems.

Firstly, cellular data networks are highly asymmetric. Downstream TCP flows are thus affected by a concurrent uplink flow or a congested and slow uplink due to the ACK packets being delayed. Secondly, packet losses are very rare due to the hybrid-ARQ scheme used in the link-level protocol. Thus, this causes the *cwnd* in the TCP congestion control algorithm to grow until the buffer overflows. As ISPs typically provision huge buffers, this causes the bufferbloat problem where the end-to-end delay becomes very large. Thirdly, recent stochastic forecasting techniques have been used to predict the network bandwidth to prevent excessive sending of packets to reduce the overall delay. However, such techniques are complicated and require a long computation or initialization time and often overly sacrifice on throughput.

To address these issues, we propose a new rate-based congestion control technique and developed a TCP congestion control framework upon which various algorithms can be built on. In our rate-based framework, the sending rate is set by estimating the available bandwidth from the receive rate at the receiver. To achieve stability and to adapt to changing network conditions, we oscillate the sending rate above and below the receive rate which will fill and drain the buffer respectively. By observing the buffer delay, we can choose when to switch between the filling and draining of the buffer. By controlling the various parameters, we can control the algorithm to optimize for link utilization by keeping the buffer always occupied, or for latency by keeping buffer occupancy low.

We implemented our framework into the TCP stack of the Linux kernel and developed two rate-based algorithms, RRE and PropRate. The algorithms were evaluated using the `ns-2` simulator as well as using a trace-driven

network emulator, and also tested on real cellular data networks. We show that by controlling the various parameters, the algorithms can optimize and tradeoff between throughput and delay. In addition, we also implemented two state-of-the-art forecasting techniques Sprout and PROTEUS into our framework and evaluated them using our network traces. We found that while forecasting techniques can reduce the delay, a quick reacting rate-based algorithm can perform just as well, if not better by maintaining a higher throughput.

Finally, our work advances the current TCP congestion control technique by introducing a new framework upon which new algorithms can be built upon. While we have showed that our new algorithms can achieve good trade-offs with certain parameters, how the parameters can be chosen to match the current network conditions has room for further research. Similar to how many *cwnd*-based congestion control algorithms have been developed in the past, we believe that our framework opens new possibilities in the research community to explore a rate-based congestion control for TCP in emerging networks. In addition, because our framework is compatible with existing TCP stacks, it is suitable for immediate deployment and experimentation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cellular data networks are becoming more and more commonplace with the higher penetration of 3G-enabled, and more recently, 4G/LTE-enabled smart-phones. Cheap data plans and widespread coverage in Singapore has made 3G/4G networks one of the main modes of accessing the Internet. Social media and networking trends are also increasing along with mobile apps which allow users to post feeds and uploading photos on the go.

The transport protocol of the Internet however, has largely remain unchanged from the wired medium of the past. With new modern wireless networks having vastly different characteristics from traditional wired or WiFi networks, it is timely to examine and update the transport layer protocol, in particular the congestion control of TCP. There is also a rising trend for users to upload media such as images and video over their mobile devices [49], hence resulting in a shift from Internet usage being mostly downstream to a mix of up and downstream.

In this thesis, we investigate mobile cellular data networks and found

that i) the downlink performance of TCP flows is severely affected by ACK packets being delayed due to a concurrent uplink flow or congestion causing a slow uplink; ii) TCP flows typically have high latencies as the low packet loss rate combined with the ISPs provisioning deep buffers, allows the *cwnd* to grow large, thus increasing buffer delays; and iii) stochastic forecasting of the link throughput can reduce the overall latency but overly sacrifices on throughput. To address these issues, we thereby propose a new rate-based approach to TCP congestion control. We show that with our framework, we can achieve high throughput/utilization in the presence of a saturated or congested uplink or achieve low latencies by controlling some parameters.

1.1 Measurement Study of Cellular Data Networks

Although also being wireless, cellular data networks behave differently from 802.11 Wi-Fi networks because it operates in a licensed band and uses different access protocols such as HSPA and LTE. Thus, it is important to first understand the characteristics of the network before we can propose improvements to the performance.

Our measurement study investigates the UDP and TCP performance of three different local telcos/ISPs across different periods of the day. The experiments were obtained from a fixed position, mainly in our lab. Our results uncovered an interesting issue with mobile network with regards to concurrently uploading and downloading with TCP. For example, though

the upstream and downstream protocols in HSPA networks function independently, a downstream TCP flow is hindered by an upstream flow because its ACKs are delayed. In particular, simultaneous uploads and downloads can reduce download rates from over 1,000 kb/s to less than 100 kb/s. While a properly-sized uplink buffer that matches the available uplink bandwidth would probably be sufficient to address this problem, the available bandwidth on the uplink varies too widely over time for a fixed size uplink buffer to be practical.

1.2 Rate-based Congestion Control for TCP

Following this measurement study, we investigate how a new rate-based TCP congestion control algorithm that eliminates ACK-clocking can improve the network performance of cellular data networks.

Previous work on improving TCP performance for 3G networks focussed on adapting to the significant variations in delay and rate and avoiding bursty packet losses and ACK compression [16, 17]. Our problem is quite different in nature from these previous problems because the crux of the issue is not that too many ACKs are received in a burst, but that ACKs are not being received in a timely manner. To the best of our knowledge, this uplink saturation problem in cellular data networks has not previously been cited in the literature.

In addition, end-to-end network delay is an important performance metric for mobile applications as it is often the dominant component of the overall response time [57]. Because cellular data networks often experience rapidly

varying link conditions, they typically have large buffers to ensure high link utilization [76]. However, if the application or transport layers send packets too aggressively, the saturation of the buffer can cause long delays [19]. Sprout [71] and PROTEUS [73] were recently proposed to address this problem by forecasting the network conditions. Their key idea is that if we can forecast network performance accurately, then packets can be sent at an appropriate rate to avoid causing long queuing delays in the buffer. However, Sprout requires intensive computations and sacrifices a significant amount of throughput to achieve low delays, whereas PROTEUS requires some 32 s of training time, which at LTE speeds, would be equivalent to 90 MB worth of data.

While forecasting has been shown to be effective at improving mobile network performance, our key insight is that it is possible to achieve similarly low delays, while maintaining a much higher throughput, by simply using a fast feedback mechanism to control the sending rate. In other words, there is no compelling need to try to predict the future. Just reacting sufficiently fast to the changing mobile network conditions is good enough.

To this end, we developed a new *rate-based* TCP congestion control mechanism that uses the buffer delay as the feedback signal to regulate the sending rate. Our mechanism uses ACK packets to estimate the current receive rate at the mobile receiver instead of using them as a clocking mechanism to decide when to send more packets, thus solving the problem of egregious ACK delays. Our key insight is that to achieve full link utilization, it suffices if *we can accurately estimate the effective maximum receive rate at the receiver and match the sending rate at the sender to it.* However, matching the sending

rate to the receive rate can not be done precisely in practice as network variations are common in cellular data networks. Thus, our sending mechanism uses a feedback-loop based on the estimated buffer delay to oscillate the sending rate. Together, these techniques combine to form a rate-based congestion control framework which enables a new-class of tunable rate-based congestion control algorithms to be designed, potentially allowing mobile applications to achieve the desired tradeoff between delay and throughput.

We validated our framework by implementing two proof-of-concept algorithms RRE and PropRate, as well as implementing the forecasting techniques of Sprout and PROTEUS in a rate-based algorithm. The algorithms were evaluated using the `ns-2` simulator as well as using a trace-driven network emulator with an actual Linux implementation. We also tested our framework over a real cellular data network.

1.3 Contributions

The key contribution of this thesis is the development of a new rate-based TCP congestion control mechanism as opposed to the traditional *cwnd*-based mechanism. TCP congestion control has always been done using a congestion window to restrict the number of outstanding unacknowledged packets. Thus, new packets are only sent when ACK packets are received. While this scheme has worked well over the years, this ACK-clocking mechanism is affected by egregious ACK delays in cellular data networks.

This thesis presents not only a new rate-based technique to overcome the problem of egregious ACK delays, but also a new framework that en-

ables new possibilities of TCP congestion control. As a proof-of-concept, we present two new algorithms for the framework and show that they can be optimized between maximizing throughput or minimizing delay. In addition, we implemented and integrated two state-of-the-art forecasting algorithms, Sprout and PROTEUS, into our framework, showing that our framework can be used with current as well as future algorithms and techniques.

Finally, we show that while forecasting techniques can decrease the delay in cellular data networks, it is not necessary as a quick reacting rate-based algorithm can also achieve similar performance. By varying the parameters used in our rate-based framework over the same network trace, we can obtain all possible tradeoff between throughput and delay. The frontier of the points show that algorithms using our rate-based framework can achieve similar, if not better performance than existing forecasting schemes.

Our work suggests that there is scope for developing new TCP congestion control algorithms that can perform significantly better than existing *cwnd*-based algorithms for mobile cellular networks. In particular, by adjusting the control parameters in our new rate-based TCP framework, mobile application developers can potentially achieve the desired performance tradeoff between delay and throughput on per application basis. Exactly how this should be done is room for future research.

1.4 Organization of Thesis

The rest of this thesis is organised as follows: In Chapter 2 we discuss the related works, followed by the measurement study in Chapter 3. We then

present our rate-based TCP congestion control algorithm and framework in Chapter 4. Thereafter, we examine in Chapter 5, RRE, a rate-based algorithm that achieves good link utilization when the uplink is saturated or congested. In Chapter 6, we present another rate-based algorithm PropRate, and compare its performance with other stochastic forecasting techniques in achieving low delays in cellular data networks. Finally, we discuss the future direction in Chapter 7.

Chapter 2

Related Work

In this section, we provide an overview of TCP congestion control protocols, especially those closely related to our work. Next, we discuss TCP performance issues and techniques to mitigate the issues in both early 2G/3G networks and the modern 3G/4G networks.

2.1 TCP Congestion Control

TCP congestion control is a well-studied subject which was first proposed by Jacobson [32] as a means to prevent "congestion collapse", a condition where too much traffic in the network causes excessive packet losses from buffer overflow. In today's TCP, the crux of congestion control is adjusting the congestion window variable (*cwnd*), which determines how many unacknowledged packets the sender can send. Different congestion control algorithms mainly determine how the `cwnd` should be increased for each incoming ACK packet and how the *cwnd* should decrease for every congestion event.

2.1.1 Traditional *cwnd*-based Congestion Control Algorithms

TCP NewReno [29] is most widely cited as the basic congestion control algorithm, which is the base algorithm implemented in the Linux TCP stack. It uses the traditional additive-increase, multiplicative-decrease (AIMD) to control the *cwnd*. Simply put, NewReno increases the `cwnd` linearly by one packet for every round-trip time (RTT) and decreases it by half for every congestion event. One good property of AIMD algorithm is that it allowed the *cwnd* of multiple flows through a link to converge to a fair value. There are two main approaches to detect congestion: 1) packet losses, and 2) increasing delay.

TCP Vegas [8] was the first algorithm that proposed using packet delay or RTT over packet loss as the main signal for congestion. It records the minimum RTT value and uses it to calculate an expected rate. The expected rate is then compared with the actual rate and the *cwnd* is additively increased, kept constant, or additively decreased based on two threshold values α and β. One advantage of delay-based algorithm is that it detects congestion before it happens whereas algorithms based on packet loss like TCP Reno detects congestion after it has happened. However, because of this early detection, TCP Vegas tends to back off before other co-existing flows using packet loss detection like TCP Reno, giving them more bandwidth. Thus TCP Vegas is not widely used as it is not able to contend fairly with other algorithms.

Recently, newer "high-speed" congestion control algorithms have been developed for use with the modern high-bandwidth networks such as ADSL and

Cable which have become commonplace for domestic Internet subscribers. The Linux kernel uses CUBIC [27] as its default congestion control module while Microsoft has developed Compound TCP (CTCP) [65] for use in its own operating systems.

CUBIC deviates from the traditional AIMD algorithms in that the *cwnd* increases according to a cubic function of time since the last congestion event. The point of inflexion of the cubic function is the *cwnd* value of the last congestion event before it was decreased. Thus Cubic aims to quickly return the *cwnd* to the previous value, plateaus around the value for some time before aggressively increasing to probe for more bandwidth.

Microsoft's CTCP algorithm combines the traditional TCP Reno AIMD window algorithm with an additional delay-based window. The final *cwnd* is the sum of these two windows. The delay-based window increases when the RTT is small to quickly probe for more bandwidth. When queueing is detected from an increasing RTT, the delay-based window is decreased to keep the total `cwnd` constant. This approach combines both packet loss and RTT to detect congestion.

H-TCP [41] works similarly to CUBIC by increasing the *cwnd* as a function of time. It toggles between conventional TCP and a high-speed mode based on some threshold. In the high-speed mode, the *cwnd* is increased by a quadratic function. When a congestion event is encountered, instead of decreasing the *cwnd* by a fixed scale, H-TCP estimates the link capacity using the RTT and scales the *cwnd* to match the throughput to that before the congestion event.

Hi-Speed TCP (HSTCP) [22] is an IETF proposal to tweak the AIMD

response function of TCP for high-speed gigabit networks. The traditional Reno AIMD functions can be generalized to a linear increase factor of 1, and a multiplicative decrease factor of $\frac{1}{2}$. When the *cwnd* is below a certain threshold value, the traditional factors are used. When it is above the threshold, the increase and decrease factor is set to a function proportional to the current *cwnd* value.

TCP Westwood (TCPW) [47] was proposed for use over 802.11 WiFi links to mitigate the effects of packet losses being mis-interpreted as congestion events due to the nature of a lossy channel. Instead of halving the *cwnd* at the onset of a congestion event, TCPW attempts to estimate the bandwidth by tracking the rate of ACKs being received. A Westwood+ algorithm was later proposed to enhance TCPW's bandwidth estimation algorithm to better handle ACK compression [26]. The enhanced algorithm counts more carefully duplicate and delayed ACK segments and employs a low-pass filter because congestion events occurs in low frequency.

These delay-based methods work by using the RTT as a parameter. Martin et al. used increases in RTT as an indicator of congestion and future loss [46]. However, the RTT is not a stable parameter in cellular data networks because of significant variance in the delays [16]. TCP Hybla [10] was developed for use in satellite connections, as they too experience large RTTs. When the RTT is large, the *cwnd* grows at a slower rate than flows with shorter RTT. To overcome this slow growth, TCP Hybla takes as reference, the RTT of a fast wired connection and increases the *cwnd* more aggressively to match the throughput to the reference connection.

All these algorithms work by adjusting the *cwnd* which determines the

maximum number of outstanding unacknowledged packets that is allowed. Thus, the sending is clocked by incoming ACK packets when the *cwnd* value of outstanding unacknowledged packets is reached.

2.1.2 Rate Based Congestion Control Algorithms

The idea of using rate information to control the sending rate of flows is not new. Padhye et al. were first to propose an equation-based approach for congestion control that adjusts the send-rate based on observed loss events [54, 23]. Ke et al. suggested pacing out the sending of packets based on the current rate instead of sending them back-to-back so as to avoid multiple packet losses [38]. However, they require precise estimates of RTT, which are not easily available and are not actually accurate indicators of link quality in cellular data networks. Another proposal of performing TCP congestion control using the rate information is *RATCP* [37], which is not a practical approach in our context as it requires the network to explicitly feedback the available rate to the TCP source. A similar rate technique is used in TCP Rate-based Pacing (TCP-RBP) to ramp up the *cwnd* after a slow start from idle [68]. However, their technique to estimate the bandwidth is analogous to TCP Vegas, which used the RTT as a parameter and not one-way delays. Their aim is to restart the ACK clocking mechanism as quickly as possible, which we have shown in our circumstances to be ineffective.

2.2 Improving TCP Performance

2.2.1 Asymmetry in TCP

In the early days, the slow delivery of ACKs was mainly due to asymmetry in the upstream and downstream bandwidth. The ACKs of a downstream TCP flow collates or gets compressed at the uplink buffer when the uplink bandwidth is low. The ACKs are then sent and received in bursts, causing the TCP sender to send data packets in spikes, further aggravating the situation. This ACK compression effect was reported by Zhang et al. while studying simulations of bi-directional TCP flow in a single link [78]. Mongul confirmed such occurrences in practice by studying real-world traces of busy segments of the Internet [51]. Kalampoukas et al. examined the methods of prioritizing ACKs and restricting the sending buffer [36], and suggested that a form of QoS to be used to allocate a minimum bandwidth per flow. This will guarantee a minimum throughput to slow flows while isolating them from the effects of faster flows.

Balakrishnan et al. proposed several techniques to overcome the problem of ACK compression caused by two-way traffic over asymmetric links [6]. Their techniques focus on regulating the ACKs by using an ACK congestion control to regulate the sending of the ACKs as well as prioritizing ACK packets at the bottleneck router of the return path. Ming-Chit et al. further suggests that ACKs should not be sent for every other data packet, but the number of data packets each ACK should acknowledge should be varied according to the estimated congestion window of the sender [50]. These techniques eventually form the RFC 3449 [5].

The asymmetric effect on TCP has also been studied in different networks. Shekhar et al. developed an operational model called the "AMP model" to understand TCP dynamics in asymmetric networks [64]. Their model is used to guide the design of buffers and scheduling schemes to improve TCP performance. Louati et al. proposed an Adaptive Class-based Queuing mechanism for classifying ACK and data packets at the link entry [43]. The mechanism adapts the weight of both classes according to the crossing traffic at the link. For ADSL networks, Brouer and Hansen argues that in general, the uplink capacity do not result in ACK compression unless the uplink is congested [9]. They showed that the ACK traffic on the uplink can be significant with larger networks of approximately 200 users.

The IEEE 802.16 WiMAX protocol has a configurable upload/download ratio in the wireless links. Chiang et al. concluded with `ns-2` simulations that the ratios for both long-lived uplink and downlink TCP flows should be 1, in order to avoid asymmetry and maximize the aggregated throughput of simultaneous bi-directional transfers [18]. Eshete et al. further investigated the impact other WiMAX operating parameters have on both network symmetry and TCP performance [21]. Wu et al. investigated how the schemes proposed by Balakrishnan et al. [6] can be used in IEEE 802.16e WiBro [72]. Yang et al. takes this one step further by exploiting the flexibility of WiMAX MAC layer to propose an adaptive modulation and coding scheme for the returning ACK uplink to improve the spectral efficiency [77]. Their focus is to reduce ACK losses which they claim contributes the most in degrading TCP performance.

In a unique case where a high-speed simplex satellite distribution system

uses a low-speed terrestrial link as a return path, Samaraweera developed "ACK compaction" and "ACK spacing" [58]. These are ACK filtering techniques to reduce ACK packets through an IP-tunnel on the return link and regenerate a suitable number of ACKs at the other end to maintain the self-clocking mechanism at the sender.

While these works solve the ACK compression problem, Heusse et al. recently showed that modern networks suffer more from the *data pendulum* effect than from ACK compression [28]. In highly asymmetrical networks, the ACK compression effect has only a minor effect on the network performance. Instead, the data pendulum effect is the primary problem in the interactions of two-way TCP connections. The data pendulum effect is when utilization of the link oscillates between the upstream and downstream flows, with each flow taking turns to fill and then drain the buffers. As cellular data networks are highly asymmetric by nature, it is likely that they will face the same problems. Heusse et al. analysis shows that using a very small upload buffer will greatly reduce the harmful interference between uploads and download. However, it is not easy to fix a small buffer size as the link bandwidth of cellular data networks tends to vary greatly.

In light of this, Podlesny and Williamson demonstrated performance degradation of two-way concurrent flows in asymmetric ADSL and cable links and proposed an *Asymmetric Queueing* (AQ) mechanism. The idea of AQ is to separate TCP data and ACK packets into different queues and prioritizing them according to some mathematical model. These previous two works however, only evaluated TCP New-Reno and not with the more aggressive CUBIC or high-speed CTCP that are the two main algorithms

used today.

2.2.2 Improving TCP over Cellular Data Networks

One issue with early 2G/3G cellular networks is that the delay varies greatly and TCP connections may spuriously timeout. Inamura et al. suggested using large window sizes and enabling the TCP Timestamp option to improve RTO estimates in order to avoid spurious timeouts [31]. ACK compression remains an issue in the early 2G/3G networks. Chan and Ramjee were amongst the first to attempt to address the poor performance of TCP over 3G networks [16]. They showed that the variable rate and delays in 3G links result in ACK compression, where the TCP source receives ACKs in bursts and hence, sends data packets in bursts. They proposed deploying an *ACK Regulator* at the ISP to control the rate at which ACKs are sent to the source based on the buffer usage at the ISP. In their follow-up work, Chan and Ramjee proposed a *Window Regulator* technique which advertises the wireless link conditions to the TCP source via the receiver window field in the ACK packet [15]. Their motivation is to control the send rate of the TCP source so that congestion losses are reduced.

Alcaraz et al. proposed combining a technique similar to the above with active queue management (AQM) algorithms at the ISP [2]. Chakravorty and Pratt also identified high latencies on the mobile downlink for 2G networks [14, 13]. They proposed the use of a mobile proxy and inflating *cwnd* to overcome its slow growth due to the large BDP. Albeit being on the older GPRS network, this shows that the problem still exists even in todays high-

speed HSPA networks. Previous approaches rely on ACKs and so cannot adequately address the mismatch between TCP ACK-clocking and the link layer design of 2G/3G protocols.

2.2.3 TCP Over Modern 3.5G/4G Networks

Xu et al. developed a receiver-side flow control (RSFC) for cellular data networks that allows the receiver of a mobile upload to limit the amount of outstanding data to be sent [75]. This in effect simulates a small upload buffer and mitigates the problems encountered with concurrent two-way TCP flows. However, this solution only works if the receiving party implements the solution. In addition, the link channels in cellular data networks are shared among other subscribers and thus uplink congestion can happen due to external factors such as crowding.

Reducing Delay

End-to-end network delay is an important performance metric for mobile applications as it is often the dominant component of the overall response time [57]. Bufferbloat describes the problem of extremely long delays being caused by huge buffers [25] and it is common in modern cellular data networks [34]. Jiang et al. prosed a dynamic receive window adjustment (DRMA) scheme to tackle the bufferbloat in 3G/4G networks [35] from a receiver side perspective. Silimar to RSFC, DRMA limits the upload flow using the receiver advertised window and increases it only when the current RTT is close to the observed minimum RTT and decreases it otherwise.

CoDel is a recent AQM scheme designed for routers that attempts to address the bufferbloat problem [52]. Packets are timestamped when they enter the queue and are dropped with high probability if they exceed a certain threshold time of staying in the buffer. The purpose is to trigger congestion in conventional TCP algorithms to prevent the buffer delay from exceeding the threshold value.

Sprout [71] and PROTEUS [73] are recent techniques that work at the sender side by limiting the amount of data to be sent to prevent excessive buffer queuing. Sprout attempts to forecast the available network bandwidth by modeling the link as a doubly-stochastic process of a Poisson process whose mean models a Brownian motion. However, the complexity of the forecast computation takes a significant amount of time and the forecasted sending rate tend to tradeoff too much throughput to achieve low delays. PROTEUS determines the sending rate by using a regression tree constructed from a history of past samples taken across time windows of 500 ms. One drawback with PROTEUS is that the suggested parameters of 500 ms and history of 64 time windows results in a long initialization time of 32 s to initialize the regression tree.

LEDBAT [63] is another flow control protocol targeted for background flows to prevent them from causing delays to other competing flow. Although LEDBAT is not a TCP congestion control algorithm, it uses the same *cwnd* mechanism as TCP to control the sending rate. Delay is kept low by estimating the buffer delay and using a proportional-integral-derivative (PID) controller to adjust the *cwnd* value. WebRTC [44] is an application layer framework for mobile networks to improve the performance for real-time

communication (RTC) applications by using Real-Time Protocol (RTP) [60], an application level protocol that runs over TCP or UDP. These techniques are application level techniques and require both sender and receiver to be running the same protocol.

Delay-centric TCP algorithms such as Vegas do keep the delay low by being very conservative in growing the *cwnd*. TCP Nice [66] and TCP Low Priority (TCP-LP) [40] are both TCP congestion control algorithms that use delay to trigger congestion. TCP Nice extends upon Vegas by increasing the sensitivity to delay and being more aggressive by halving the *cwnd* value when delay is detected. TCP-LP uses the one-way delay estimated from packet timestamps to trigger congestion. While these algorithms do prevent large delays, they tend to over tradeoff throughput and being conventional TCP algorithms, are affected by egregious ACK delays in cellular data networks.

Chapter 3

Measurement Study

In this chapter, we first present an overview of the High-Speed Packet Access (HSPA) protocol that is used in 3.5G mobile networks and the LTE protocol in the 4G networks. Next, we present the results of our measurement study of existing 3.5G/HSPA mobile networks in Singapore.

3.1 Overview of 3.5G/HSPA and 4G/LTE Networks

3.1.1 3.5G/HSPA Networks

The characteristics of the physical layer for the *High-Speed Packet Access (HSPA)* protocol that is common in modern 3.5G networks is quite different from that for IEEE 802.11x (WiFi) and other wireless networks. HSPA consists of two different sub-protocols: High-Speed Downlink Packet Access (HSDPA) and High-Speed Uplink Packet Access (HSUPA). In both sub-

protocols, several radio channels are used concurrently to send and receive coordination commands between the mobile device and base station, while a dedicated data channel is used for transmitting the data frames. A Hybrid Automatic Repeat-Request (HARQ) protocol encodes Forward Error-correction into each data frame to reduce frame corruption errors and automatically retransmits frames that cannot be recovered. This significantly reduces the packet loss rate due to random wireless losses but potentially introduces significant packet reordering.

When transmitting data, both HSDPA and HSUPA use Time-division Multiple Access (TDMA) to share the access among users on the data channel. The transmit slot size is typically 2 ms. The slot scheduling is coordinated by the base station based on several matrices which may include signal quality or even data price plan of the user.

In HSDPA, Code-division Multiple Access (CDMA) is also used over the data channel to multiplex up to 15 codes, allowing concurrent data transfer to 15 different devices, or all to a single user. This is not possible on HSUPA because there is insufficient power on the phone to enable higher levels of coding, or to coordinate concurrent CDMA from different sources. This is not an issue for HSDPA because the base station has access to a power source and it is broadcasting from a single source.

In general, the downlink HSDPA protocol is generally able to transmit data at a significantly higher rate than its uplink HSUPA counterpart. On the other hand, devices wanting to upload data on HSUPA has to share time slots with other users to transmit their payload leading to potentially significant delays when the uplink is congested. In other words, asymmetry

is inherent in the design of the physical layers of the HSPA protocol because of fundamental power constraints.

3.1.2 4G/LTE Networks

LTE was designed as a completely new standard and does not build upon previous GSM/UMTS standards. Orthogonal frequency-division multiplexing (OFDMA) or a variant orthogonal frequency-division multiple access (OFDMA) are now used in the downlink protocol instead of CDMA [59]. This allows the signal to be split into multiple narrow-band sub-carriers of different frequency. OFDMA further supports multiple users by using TDMA or FDMA to divide the sub-carriers. Having several narrow-bandwidths are easier to scale than a single wide-bandwidth, making higher bandwidths available using OFDM than with CDMA.

On the uplink protocol, a variant of OFDM known as single-carrier frequency-division multiple access (SC-FDMA) is used. While in OFDMA, a user uses several sub-carrier channels in parallel, each user is only assigned one sub-carrier channel in SC-FDMA, hence the name single-carrier. Resource blocks both in the time and frequency domain are scheduled to users by the basestation, allowing concurrent uplink transmissions from multiple users. While higher speeds can now be supported, the 3GPP standards still specifies an asymmetric link with an instantaneous downlink peak at 100 Mb/s and uplink at 50 Mb/s [62].

3.2 Measurement Methodology

For our experiments, we used the HTC Desire mobile phone with the Android 2.3 (Gingerbread) operating system. The HTC Desire is equipped with a 1 GHz Qualcomm Snapdragon processor and has 576 MB of RAM. The phone supports up to 7.2 Mbps download and 2 Mbps upload speeds on 3G. We brought to phones to three different locations for the tests: in our laboratory, in a shopping mall and a residential apartment. The servers used for our measurement study are Intel Core2Duo or better Linux machines with more than 4 GB RAM located in our lab connected to the university network. We wrote a custom Android application which is installed on the phones to run and coordinate various data transfer tests over 3G plans which we purchased from the three local telcos. The plans used were advertised at 7.2 Mbps.

Various network measurement tools such as iPerf are available for basic throughput measurements. However, we required more control over certain socket parameters like selecting congestion control algorithm that existing tools did not provide. In addition, the 3G networks are behind NATs, allowing only client-to-server connections, but not vice versa. Existing tools do not take this into account and thus can only perform a single direction test from client to server. We also needed to coordinate the tests between the phone app and the server application. Therefore, we were left with little choice but to write a new testing tool to generate the traffic and capture the packet traces using *tcpdump* on both the sender and receiver.

There are some practical challenges in measuring practical 3G networks.

For example, the switching of a mobile phone to the RRC (Radio Resource Control) state will have an influence on the measurement results. Another anomaly that we observed was that sometimes there would be an initial delay at the start of a test, where packets get buffered and are received in a larger burst than usual. We observed this behavior in both TCP and UDP flows. Because we could not control the state of the radio directly or eliminate the initial buffering, we ran each measurement test several times and took the average in order to reduce the impact of these variations.

Also, it is possible for the first connection of each battery of tests to experience an additional slight delay arising from the need to initiate channel access. This spurious delay was eliminated in our experiments by first negotiating a initial connection before starting the bulk data transfer experiments.

3.2.1 Loopback Configuration

To accurately measure delays and packets in flight, we also set up an experiment in a loop-back configuration. In this configuration, the Android phone was tethered to the server machine via USB. Next, the upload and download TCP 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 all 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). We can also determine the exact number of packets

in flight at any point in time.

3.3 Measurement Results

From our measurement study, we first look if packet size matters as the 3G/HSPA frame size is much smaller than an IP packet MSS. Next we examine buffer size as it affects TCP performance. Finally we examine the throughput performance of both UDP and TCP together with concurrent up and down flows.

3.3.1 Does Packet Size Matter?

Intuitively, sending large packets over a shared wireless network would degrade performance since this increases the probability of packet collisions. Korhonen and Wang reported a correlation between frame sizes and transport delay in 802.11b WiFi networks, although the difference in application packet sizes does not significantly affect performance [39]. HSPA uses small frame sizes between 120 and 360 bytes (depending on modulation), which is significantly smaller than corresponding WiFi packets. To this end, we investigated if packet sizes would have an impact on the performance and loss tolerance in HSPA networks.

In this experiment, we saturated the mobile link using UDP streams with datagrams of varying sizes for each test. We repeated this experiment with different send rates and found that there was no significant difference in the raw throughput or loss rates. The goodput of the UDP streams with smaller packets was naturally lower because of the additional overhead in the packet

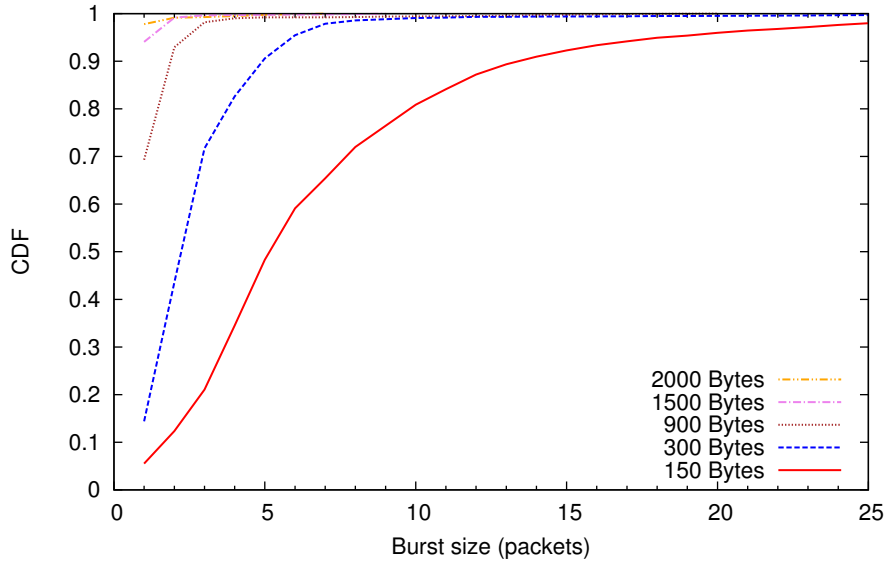


Figure 3.1: Distribution of packets coalescing in a burst for downstream UDP at 600 kb/s send rate observed with `tcpdump`.

headers.

However, we observed an interesting pattern of packets arriving in bursts, often with no more than 1 ms separating the first and last packet in each burst. These bursts tend to arrive at intervals that are multiples of 10 ms. In Figure 3.1, we plot the distribution of the size of these packet bursts when sent at a very low sending rate to prevent packet losses. We found that larger packets have a lower tendency to arrive in bursts compared to smaller packets. We also found that when the send rate was increased, more packets tend to arrive in larger bursts.

We initially suspected that this "bursty" behavior is caused by the scheduling algorithm at the base station. However, on closer inspection, we found that the amount of data in each burst is larger than the amount of data that can be transmitted in a single HSDPA time slot. However, when using WiFi, we did not find such bursts in the flows. Hence, this suggests that

the behavior is more likely due some polling cycle or hardware limitation of the cellular radio. Regardless, this observation suggests that algorithms that rely on observing the pattern of packet arrival timings such as packet trains [55, 12] are not likely to work well in the mobile 3G environment. This was examined more in detail in a related thesis [74].

3.3.2 Buffer Size

Buffer sizing is an important parameter which affects TCP performance. A rightly sized buffer will contain sufficient packets to utilize the link when the TCP sender reduces its send window when it detects congestion. Having a buffer that is too small will lead to link under-utilization, while having a buffer that is too large will cause additional delays and high RTT. There is a classic rule of thumb that the buffer should be sized to at least the bandwidth-delay product [67] (BDP). In recent times, it was found that the buffer size can be reduced to $\texttt{BDP}/\sqrt{n}$, where n is the number of long-lived flows [3].

The buffer sizes of both the downstream and upstream of our local ISPs were examined in our earlier work [76]. By flooding the channel with UDP packets sent at a high rate. This will induce a buffer overflow. Because the link layer automatically corrects for loss or corrupt packets, any packet loss can be mostly attributed to buffer overflow. As the server and phone is synchronized using USB before each experiement, we can know for certain by examining the timestamps in the network trace how many packets were present in the network in any point of time. By observing the number of

Table 3.1: Buffer sizes of the various ISPs obtained from our related work [76].

ISP	Network	Buffer Size	Drop Policy
ISP A	HSPA(+)	4,000 pkts	Drop-tail
	LTE	($\leq$ 800 ms)	AQM
ISP B	HSPA(+)	400 pkts	Drop-head
	LTE	600 pkts	Drop-tail
ISP C	HSPA(+)	2,000 pkts	Drop-tail
	LTE	2,000 pkts	Drop-tail

packets in flight, correlating when large packet losses start to kick in, and accounting for the bandwidth delay product, the buffer size can hence be deduced.

The results from our previous work is shown in Table 3.1. From this we can see that 2 out of 3 of our local ISPs provision very large buffers of over 2,000 packets. Furthermore, we observed that one of the ISPs implemented some form of active queue management (AQM) which dropped packets that remained in the queue for longer than 800 ms. It suggests that the ISP might be experimenting with CoDel [52] to alleviate the bufferbloat issue.

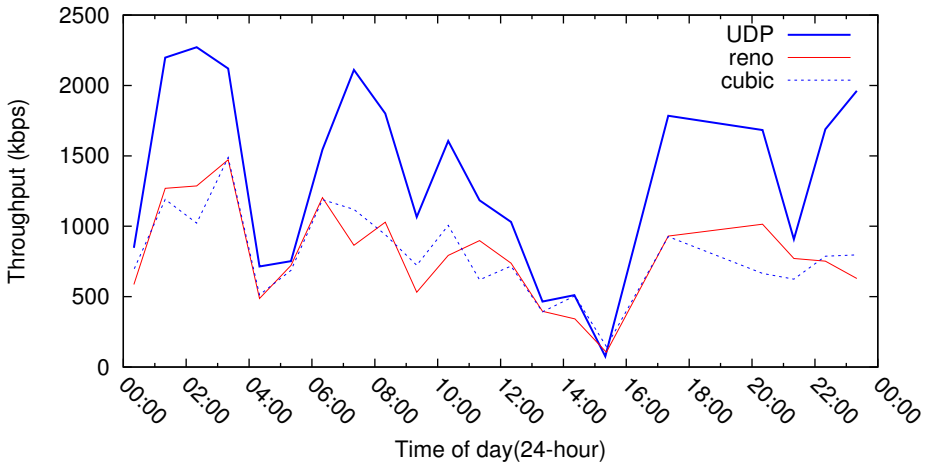
3.3.3 Throughput

To investigate the performance of the public 3G networks in Singapore as perceived by local consumers, we took throughput measurements at three different locations: (i) in our lab in campus, (ii) in a residential apartment, and (iii) in a busy shopping mall. These locations have different amounts of human traffic at different times of the day. Our experiment consists of a

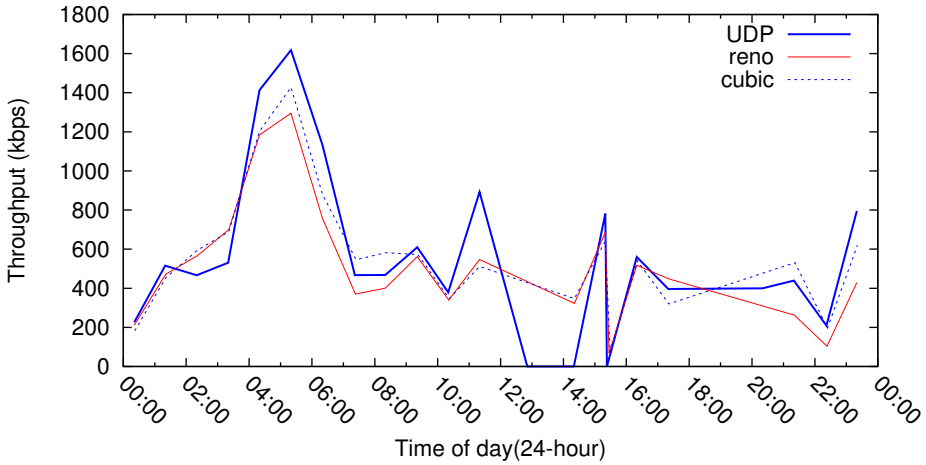
battery of interleaved tests that measure the performance of the uplink and downlink for UDP and different variants of TCP variants every hour over a period of several days. Tests were run back-to-back to minimize the impact of possible temporal variations to allow us to compare the effects of different parameter variations fairly.

In order to obtain accurate measurements of the one-way packet delay in the 3G uplink and downlink, we also set up a special loop-back network configuration. The phone was tethered to the server machine via USB cable and packets were routed through the phone's 3G/HSPA connection and back through the server's regular network interface. This allowed us to run both the server and client application on the same machine, removing the need to perform clock synchronization at the expense of mobility. We could replicate this setup in both our lab and at the residential apartment (via a domestic broadband line), but not at the shopping mall.

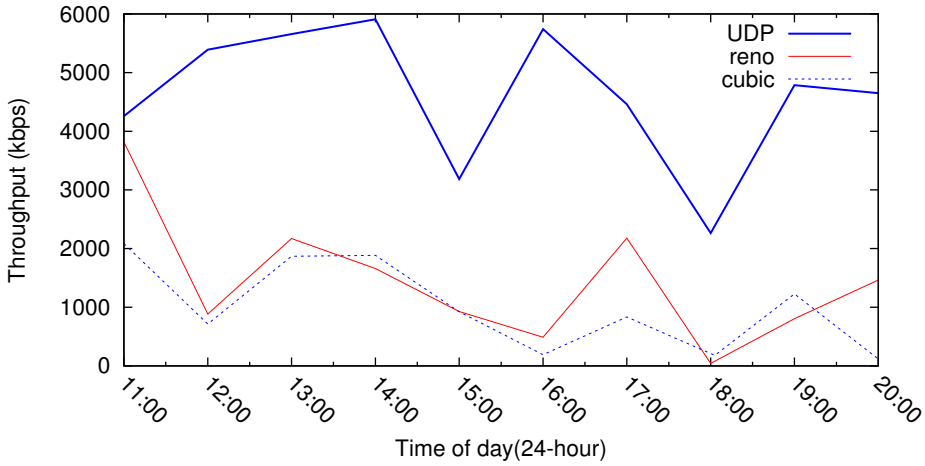
In Figures 3.2, 3.3 and 3.4, we plot the variation in the average TCP throughput (for TCPs Reno and Cubic [27]) for a bulk transfer for all three local mobile ISPs at three different locations over the course of a typical day. For the lab and apartment, the measurements were taken at hourly intervals over several days. For shopping mall, the duration of test is limited to the opening hours of the mall (11 am to 9 pm). To measure the upper bound on performance, we use a UDP stream that sends a constant stream of packets that can fully saturate the downstream bandwidth in order to measure the maximum available download capacity. Three measurements are taken per configuration to minimize random errors and the figures are sample 24-hour traces. There are minor variations between days, but the



(a) Lab on Campus

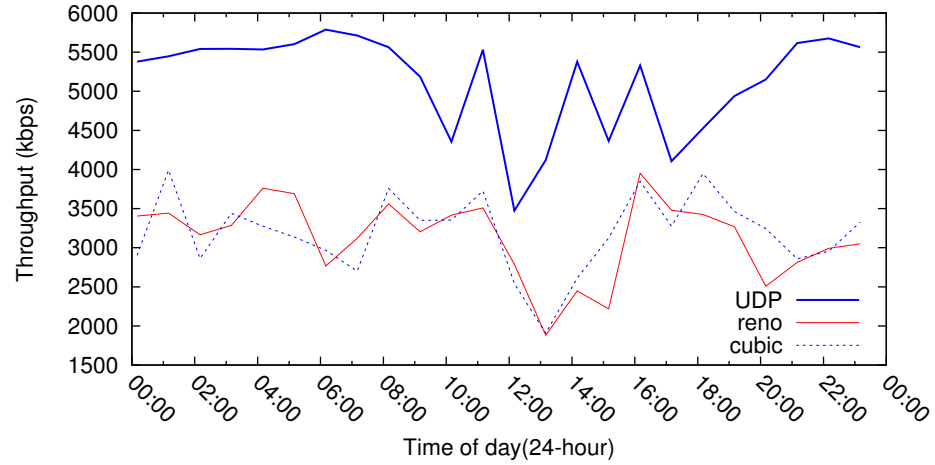


(b) Residential Apartment

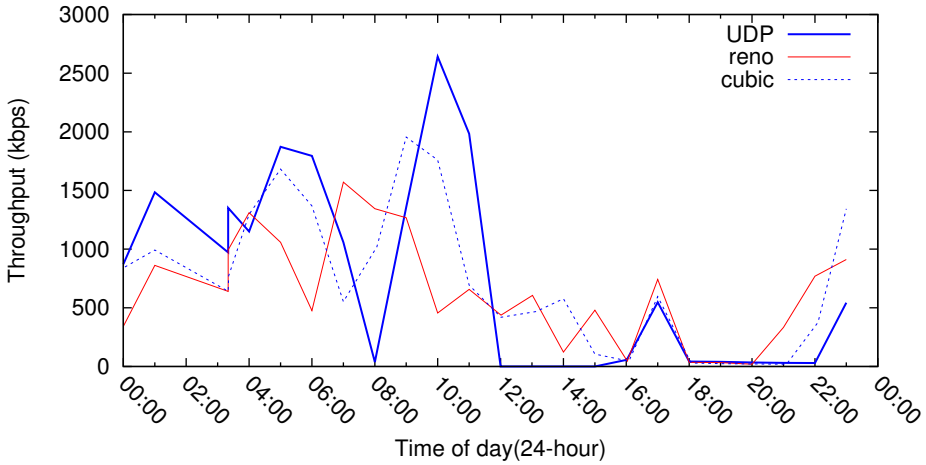


(c) A Shopping Mall

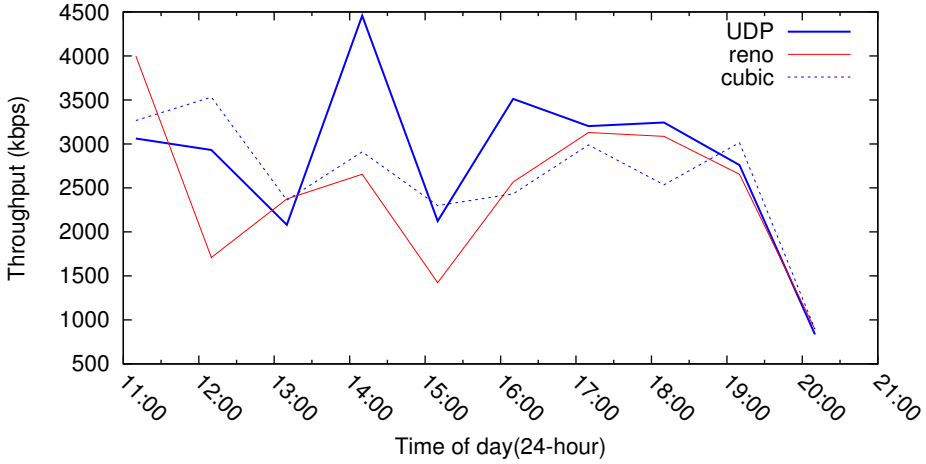
Figure 3.2: 24 hour downstream throughput of UDP and TCP for ISP A.



(a) Lab on Campus

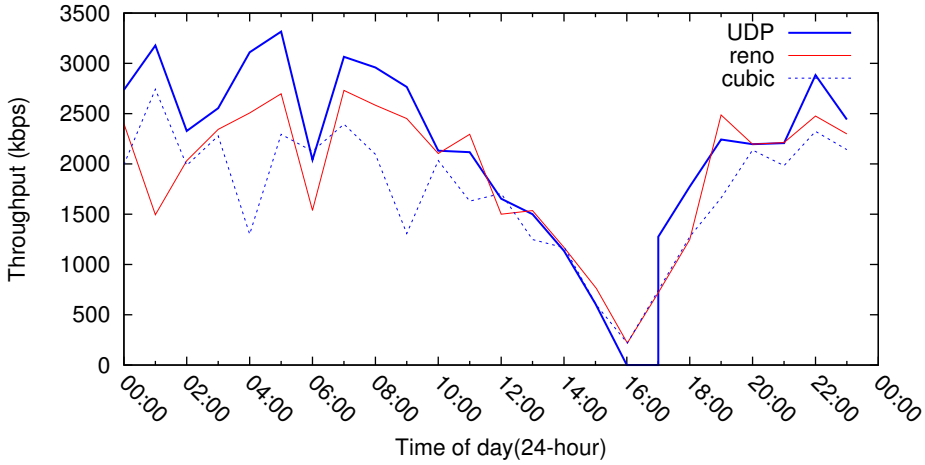


(b) Residential Apartment

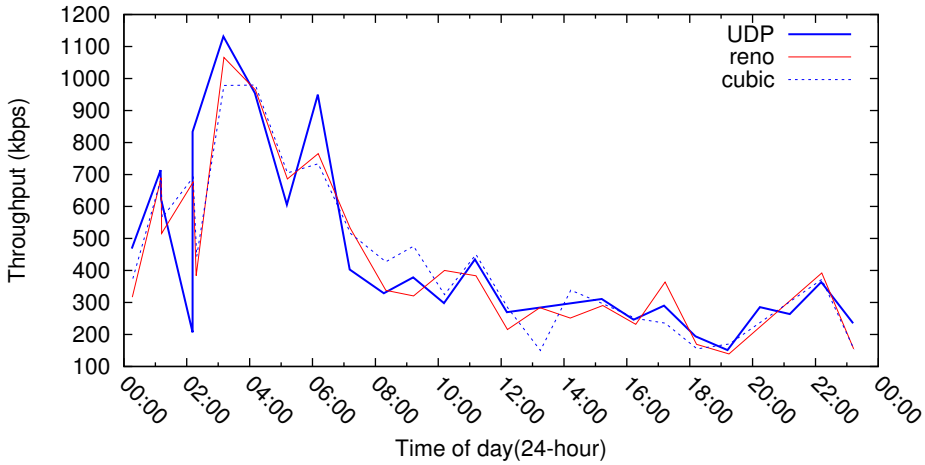


(c) A Shopping Mall

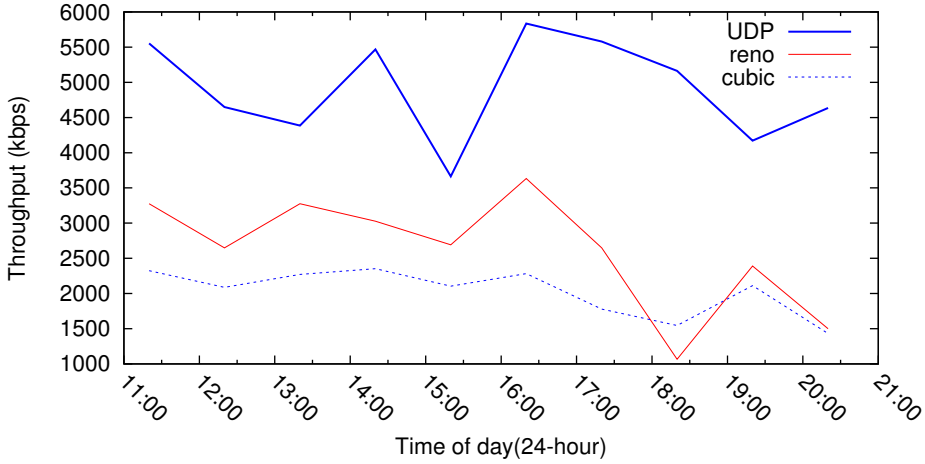
Figure 3.3: 24 hour downstream throughput of UDP and TCP for ISP B.



(a) Lab on Campus



(b) Residential Apartment



(c) A Shopping Mall

Figure 3.4: 24 hour downstream throughput of UDP and TCP for ISP C.

trends are consistent. There were also some points where the throughput was close to zero, possibly due to some temporal network outages during the time we ran the experiment.

As expected, we found that the throughput fluctuated over the course of a day and varies between ISP and location. In general, we found that there are little difference between Reno and Cubic, so we used TCP Cubic, which is currently the default TCP implementation for the Android kernel, for subsequent experiments. Also, as expected, TCP throughput is typically lower than that for UDP. We plot the ratio of achieved download throughput of UDP to TCP for the all the data points obtained our experiments in Figure 3.5. We see that the performance varied quite significantly across the various ISPs. The throughput for UDP is typically no worse than that for TCP, except for ISP C, for which UDP performance is slightly worse about 25% of the time, which leads us to suspect that ISP C might have implemented some QoS-like scheme in their network that preferentially drops UDP packets. What is of significant interest is that some 20 to 50% of the time, TCP uses less than 50% of the available bandwidth.

We next compare the difference between the downlink and uplink for each of the three ISPs. Figure 3.6 shows the downlink and uplink throughput obtained from another experiment done in our lab over a 24-hour period on a typical weekday. While large variations in the throughput is again observed, there is a large difference between the downlink and uplink speeds for all three ISPs.

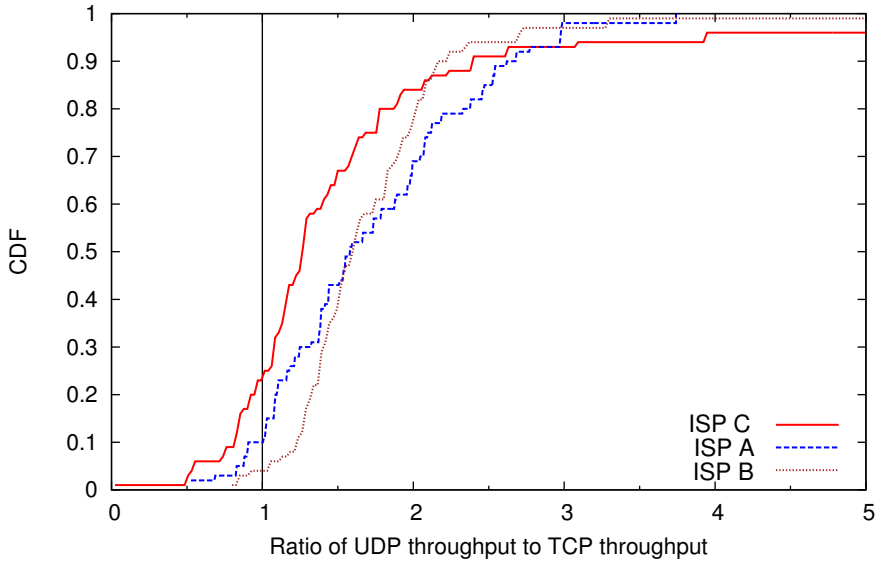


Figure 3.5: CDF of the ratio of UDP throughput to TCP throughput for the various ISPs.

3.3.4 Concurrent Flows

To investigate how concurrent uploads can affect downloads, we ran three independent TCP measurements: 1) downloading 1 MB of data, 2) uploading 1 MB of data, and 3) downloading 1 MB of data while concurrently uploading a huge data file in the background. These three tests were run at 15-min intervals over several days in a back-to-back manner (to control for temporal variations).

In Figure 3.7, we plot the results for ISP A over a 48-hour period during a weekend, where there tends to be fewer people on campus and therefore less 3G interference from other users. From the resulting traces, we see that we can actually achieve the advertised 7.2 Mbps rates for raw UDP transfers during the night hours. It seems that ISP A caps the upload bandwidth and so upload bandwidth is uniformly low during the entire period. The

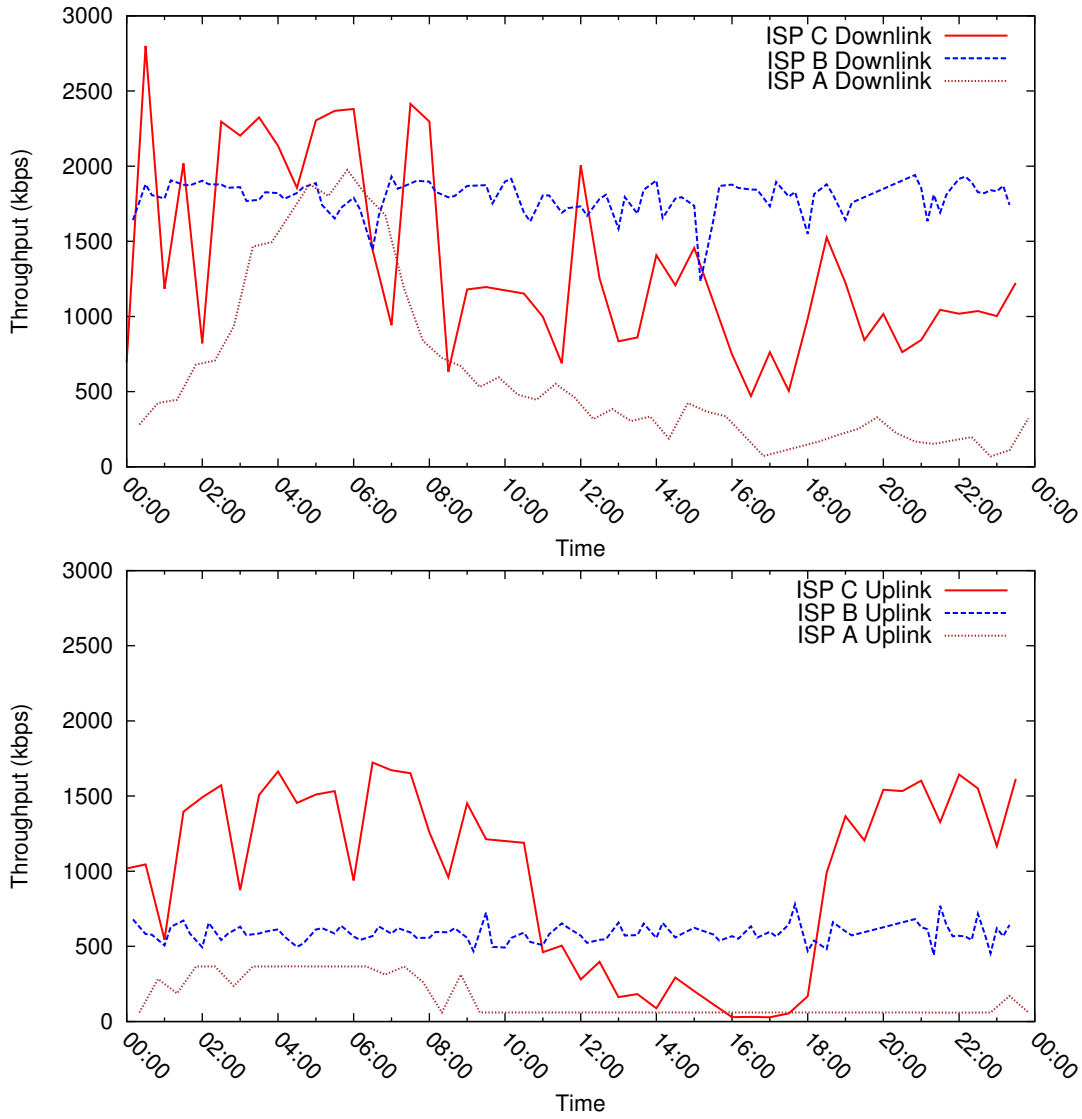


Figure 3.6: TCP throughput for three different mobile ISPs over a 24 hour period on a typical weekday.

throughput for a raw TCP download is significantly lower than that of the raw UDP throughput, but in the presence of a concurrent background TCP upload, it is even more significantly degraded. The performance gap between UDP and TCP for 3G networks is well-studied and well-understood [16]. We focus on understanding and addressing the performance degradation arising from the concurrent upload, which to the best of our knowledge, has not been studied extensively.

Degradation Caused by Concurrent Upload

We next examine the performance of downlink TCP flows with and without the presence of a concurrent upload flow. For each run, we performed one complete TCP download that has no concurrent upload back-to-back with another TCP download having a concurrent upload. This is to reduce the temporal effects of the network.

We plot the average RTT and throughput of each run in Figure 3.8. The results clearly show that the RTT is increased when there is a concurrent upload as compared to without and the effect is especially significant for ISP A. Associated with the increase in RTT is also a significant drop in the throughput in most instances.

In our experiment setup, we tether the mobile phone directly to the server so that we can measure the one-way delays between the server and the mobile phone and back accurately. In Figure 3.9, we plot the throughput against the ratio of the uplink one-way delay to the downlink one-way delay. Our results show that when the uplink one-way delay is comparable to the downlink one-way delay, we get good TCP throughput. On the other hand, with a

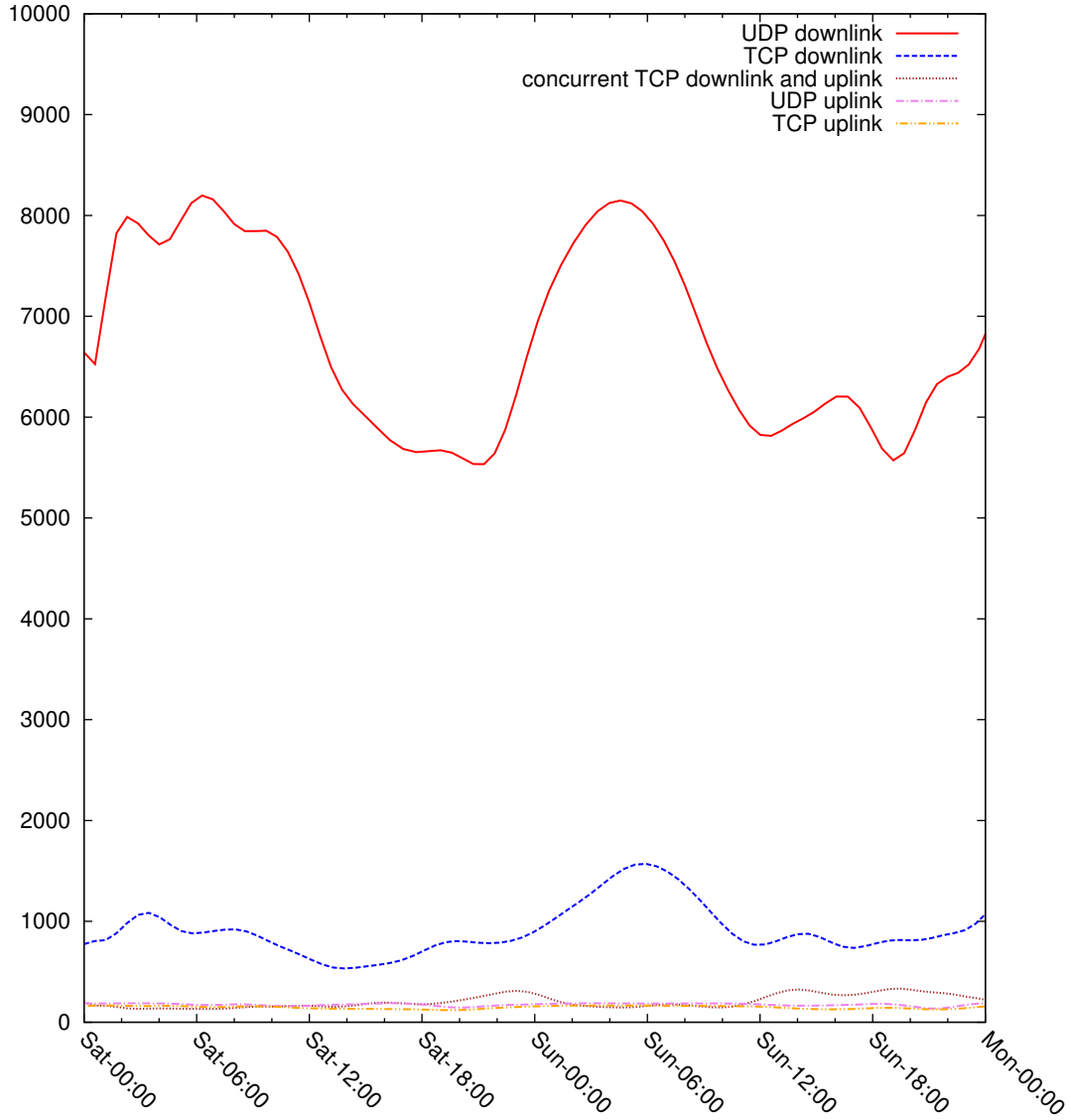


Figure 3.7: Measured throughput for ISP A over a weekend.

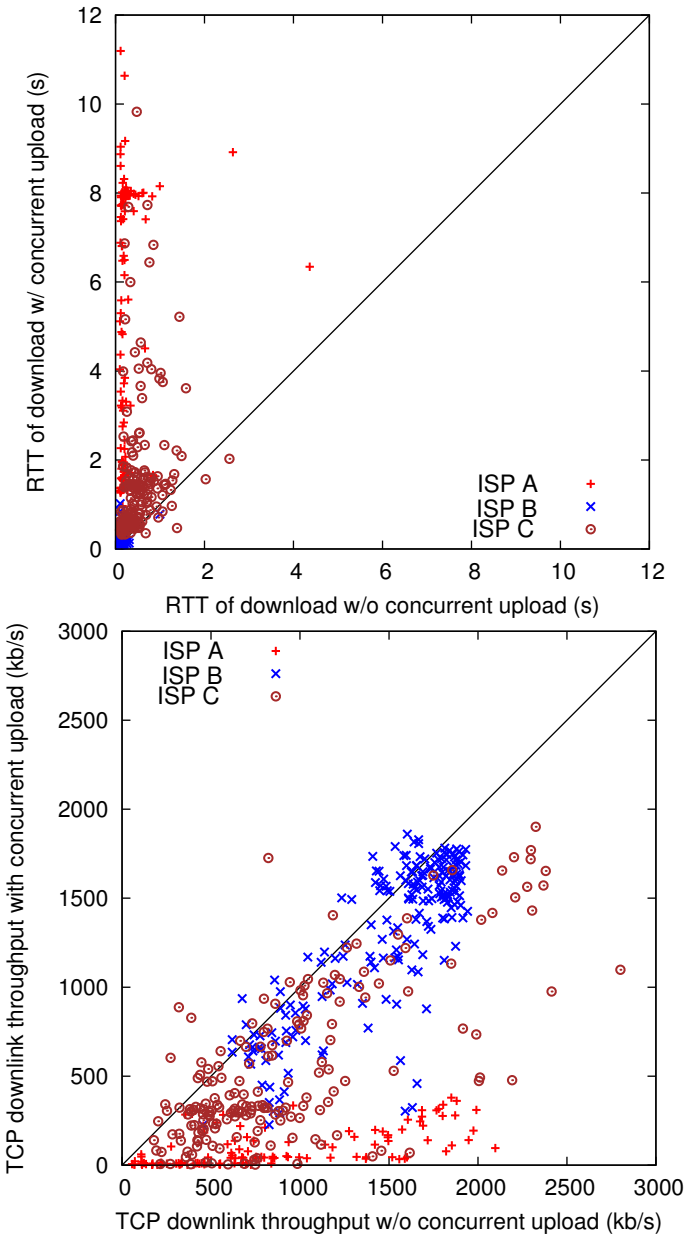


Figure 3.8: Comparison of RTT and throughput for downloads with and without uplink saturation.

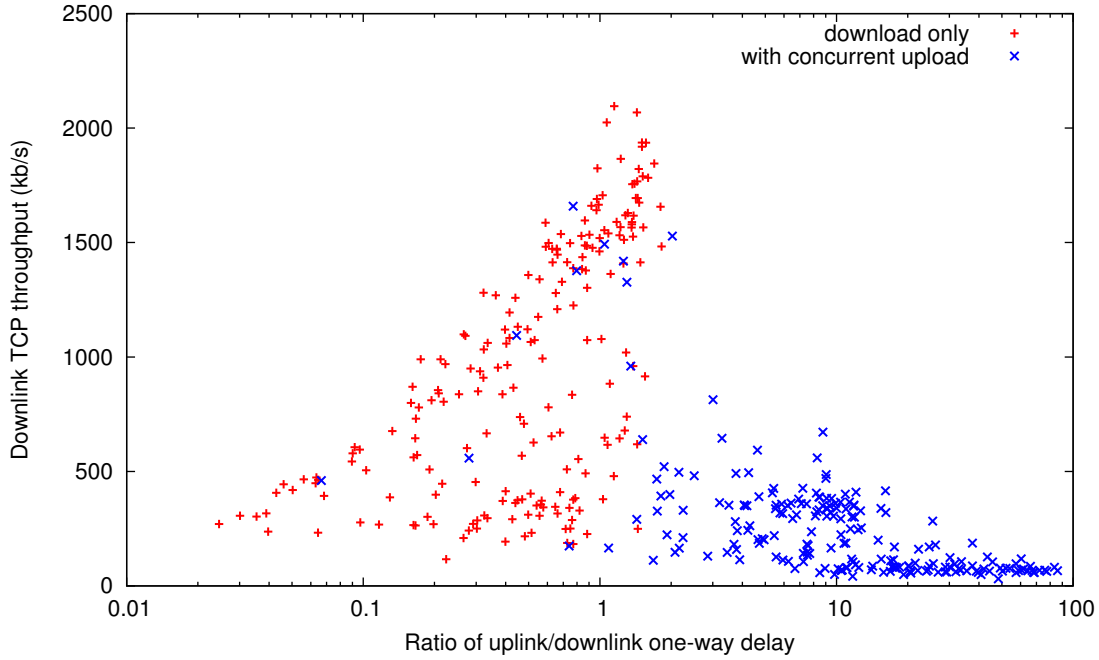


Figure 3.9: Ratio of one-way delay against ratio of downlink throughput.

concurrent upload, the uplink one-way delay becomes significantly larger and we get significantly reduced TCP throughput. In Figure 3.10, we illustrate a typical uplink saturation scenario. The RTT is dominated by the uplink one-way delay since the uplink one-way delay can be some ten to a hundred-fold larger than the downlink delay (which is typically about 70 to 100 ms).

Bandwidth Under-Utilization

Because we have the full trace of all packets sent and received, we can deduce the number of packets in flight at each instant in time. In Figure 3.11, we plot the cumulative distributions of the number of packets in flight over time for the two different scenarios. It is quite clear that in the presence of a concurrent upload, the downlink is significantly under-utilized. To put things into perspective, at a bandwidth of 7.2 Mbps and an RTT of 150 ms, we ex-

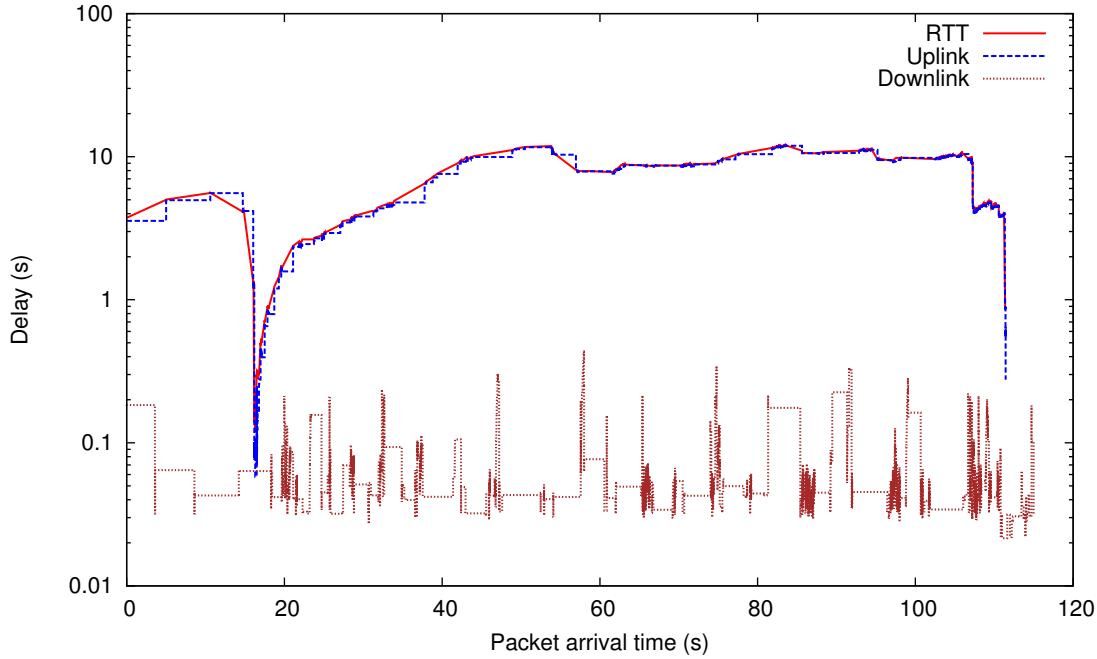


Figure 3.10: Breakdown of the RTT into the one-way uplink delay and the one-way downlink delay under uplink saturation.

pect the bandwidth delay product to be about 90 (for 1,500 byte MSS). This means that we expect about 45 packets to be in flight and unacknowledged if the channel is symmetric and fully utilized.

3.4 Summary

In summary, we have shown in our measurement study that there is significant performance degradation in the downlink of cellular data networks when then uplink is saturated. The uplink can be saturated due to a concurrent flow from the mobile device, or due to high user traffic at the basestation. This causes the returning TCP ACK packets from a downstream flow to be delayed, resulting in under-utilization of the downlink. In the next chapter,

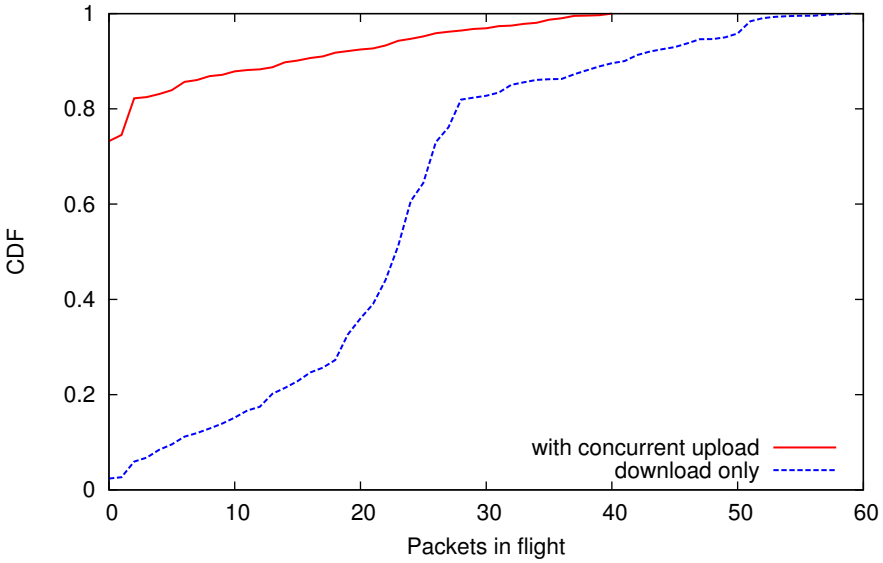


Figure 3.11: Distribution of the number of packets in flight for TCP download both with and without a concurrent upload.

we present our solution to address this problem.

Chapter 4

Rate Based TCP Congestion Control Framework

In chapter 3, we showed that there is significant performance degradation in cellular data networks when the uplink buffer is saturated, as illustrated in Figure 4.1.

We now describe our approach of addressing the problem by eliminating ACK clocking with a new rate-based congestion control algorithm. We observed that the mobile downlink remains the bottleneck for the system and that as shown in Figure 3.10, the uplink one-way delay can increase to tens of seconds while the downlink one-way delay remains small when the uplink buffer is saturated. Our key insight is that under such circumstances, ACK clocking will clearly result in under-utilization since the ACKs are delayed. To achieve good utilization, it suffices if *we can accurately estimate the effective maximum receive rate at the receiver and match the send rate at the sender to it.* Also, for our solution to be easily deployable, it should

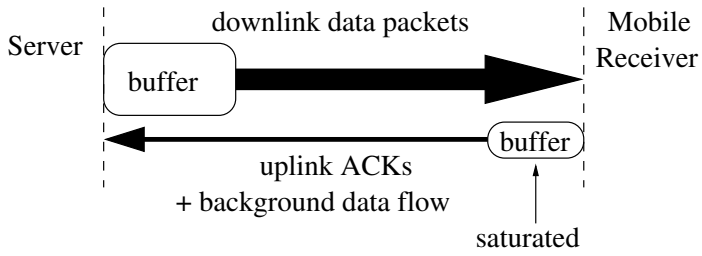


Figure 4.1: Model of uplink buffer saturation problem.

remain compatible with an unmodified TCP stack at the receiver. To do so, we developed a technique to estimate the receive rate by exploiting the TCP Timestamp option. This works because even when the uplink buffer is saturated, the downlink one-way delay remains small and is relatively stable.

4.1 Rate-Based Congestion Control

In principle, we can achieve high link utilization in mobile cellular networks by sending packets at a rate that matches the available bandwidth. One way to estimate the available bandwidth is to send a burst of packets at a rate that saturates the bottleneck link and measure the receive rate of the receiver [71]. However, there are two key challenges. First, the receive rate is difficult to estimate with high accuracy given the bursty nature of packets in cellular data networks. Also, the available bandwidth can vary by over an order of magnitude over a period of several minutes [76]. Second, simply matching the sending rate to the receive rate is not sufficient. It is relatively easy to detect a drop in the available bandwidth, but it is much harder to tell if the available bandwidth has increased. Not reacting in a timely manner when the available bandwidth increases would result in link under-utilization.

TCP currently uses a *cwnd*-based congestion control mechanism that works reasonably well for conventional wired networks, but which falls short for mobile cellular networks for three structural reasons: (i) the TCP *cwnd*-based mechanism regulates the sending rate indirectly, and so often reacts too slowly to the rapid network variations observed in mobile cellular networks [75]; (ii) ACK-clocking can result in significant downlink under-utilization if the ACK packets are delayed at a saturated uplink buffer [42]; and (iii) the *cwnd*-based TCP mechanism depends on packet losses to detect congestion, which requires that buffers be filled and naturally introduces significant delays which are detrimental to RTC applications [73]. To address these shortcomings, we proposed a new rate-based congestion control mechanism where we directly control the sending rate of packets in response to the estimated receive rate. To achieve the same stability inherent in an ACK-clocked system, we employ an oscillating negative feedback mechanism.

4.1.1 Congestion Control Mechanism

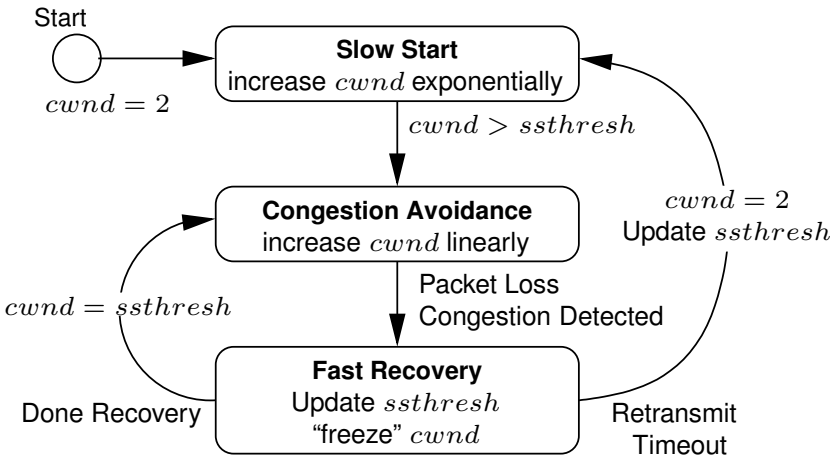
Our key idea is to use a feedback loop to regulate the sending rate so that the number of packets in the bottleneck buffer oscillates around a constant value, thus preventing link under-utilization while keeping latency low. We compare our new rate-based TCP stack to the conventional *cwnd*-based stack in Figure 4.2. Note that Figure 4.2(a) illustrates a generic *cwnd*-based TCP stack and not a specific TCP implementation as there are *cwnd*-based implementations, e.g., CUBIC, that set the *cwnd* and *ssthresh* according to a different algorithm. Likewise, the illustration for our rate-based stack is also

generic as the parameters are defined by the specific algorithms.

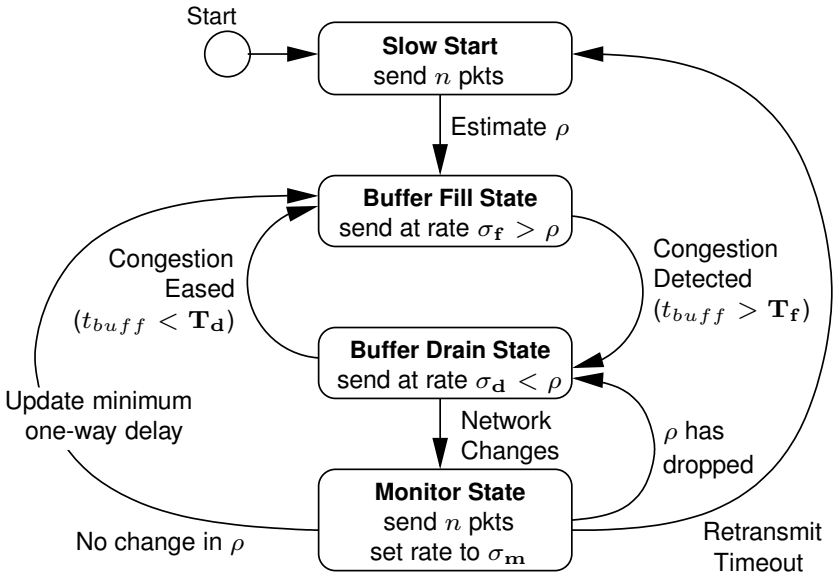
Both the conventional *cwnd*-based TCP stack and our rate-based stack begin in a Slow Start state. For the *cwnd*-based stack, the *cwnd* is initially set to 2 and is doubled every RTT, until the *cwnd* exceeds $ssthresh$. In our rate-based stack, a burst of n packets are sent to obtain an initial estimate of the receive rate ρ. If n packets are not sufficient to estimate the rate, then n is doubled and another burst is sent. This is repeated until a rate estimate is obtained.

After Slow Start, the conventional *cwnd*-based stack will enter a Congestion Avoidance state. In this state, the *cwnd* is now increased by $\frac{1}{cwnd}$ for each ACK received. This process continues until a packet loss occurs, presumably due to buffer overflow and it goes into the Fast Recovery state. For our rate-based stack, we have an analogous state called the Buffer Fill state. In this state, the sending rate is set to $\sigma_f > \rho$ which will essentially cause the buffer to fill. Instead of using packet drops to detect congestion, we adopt the technique used in recent works [63, 71] and instead detect congestion by estimating the buffer delay. When the estimated buffer delay t_{buff} crosses a threshold T_f, the stack switches to the Buffer Drain state.

In the Fast Recovery state of the *cwnd*-based stack, the *cwnd* is controlled by the fast recovery algorithm to retransmit packets and prevent the pipe from draining. Once the sender has received the ACKs for the retransmitted packets, recovery is complete, the *cwnd* is set to $ssthresh$, and the state returns to the Congestion Avoidance state. If fast recovery fails and the retransmission times out, the stack will reset *cwnd* to 2 and return to the Slow Start state. Similarly, our rate-based stack sends packets at a rate of



(a) Conventional $cwnd$-based mechanism.



(b) Rate-based mechanism.

Figure 4.2: Comparison of TCP congestion control mechanisms.

$\sigma_d < \rho$ to drain the buffer in the Buffer Drain state. Once the buffer delay falls below T_d, the stack switches back to the Buffer Fill state. Also, if there is a retransmission timeout, the stack returns to the Slow Start state.

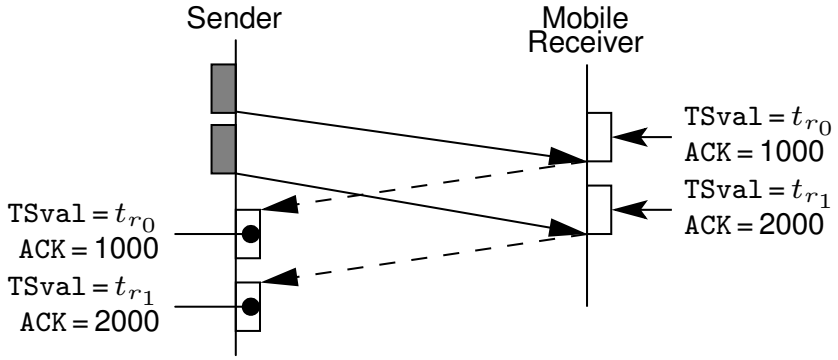
The key difference between our rate-based stack and the *cwnd*-based stack is that we have an additional *Monitor state*. Because the underlying one-way delay might change due to variations in the mobile networks and our feedback mechanism detects congestion by estimating the buffer delay t_{buff}, it is possible for the buffer to be completely emptied, even when the buffer delay remains above T_d. To address this scenario, we switch to the Monitor state when the algorithm remains in the Buffer Drain state for an extended period of time. In this Monitor state, we use a burst of n packets to obtain a new estimate of the receive rate. While the new estimate is being obtained, the sending rate is conservatively set to $\sigma_m < \sigma_d$ to avoid flooding the buffer. If the new rate estimate ρ is greater or equal to the current estimate, we conclude that the buffer is indeed empty. We update the one-way delay measurements and switch to the Buffer Fill state. Otherwise, it means that the buffer is still not yet empty, and we simply return to the Buffer Drain state to continue draining the buffer. The *cwnd*-based stack does not need a state that is equivalent to our Monitor state because the *cwnd*-based congestion mechanism relies on packet loss to signal congestion, and thus does not need to worry about underlying changes in the network delays.

4.1.2 Estimating the Receive Rate

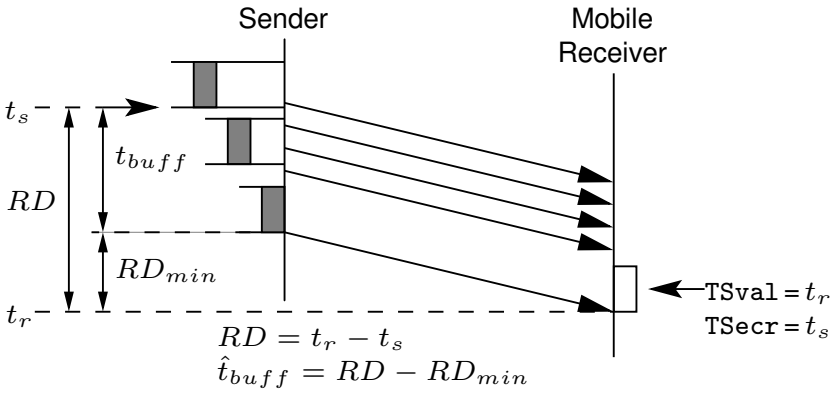
A natural approach to obtain an estimate of the receive rate or bottleneck bandwidth, is for the receiver to perform the estimation based on the received packets and to send the estimate back to the sender [71, 75]. The same approach could also be adopted to implement our rate-based mechanism. However, to keep all modifications at the TCP sender and avoid modifications to the TCP receiver, we decided to adopt an indirect method that allows the rate estimation to be performed at the sender, when the TCP timestamp option is enabled. The TCP timestamp option is enabled by default on Android and iPhone devices, which together account for more than 90% of the current smartphone market [1].

When the TCP timestamp option is enabled, a TCP receiver will send ACK packets with the `TSval` set to its current time. This timestamp is the same as the receiving time of the data packet. Thus, the packet arrival times are effectively embedded in the ACK packets. From the ACK number and the timestamp value, the sender can determine the number of bytes received by the receiver. In the example illustrated in Figure 4.3(a), the sender can determine that 1,000 bytes have been received in the time period between t_{r_0} and t_{r_1}.

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



(a) Estimating the receive rate.



(b) Estimating the buffer delay

Figure 4.3: Using TCP timestamps for estimation at the sender.

4.1.3 Inferring Congestion

The level of congestion in the network is inferred by estimating the buffer delay t_{buff} from the observed changes in the one-way delay of the received packets. The one-way delay is the time difference between the point where a packet is queued for delivery at the sender to when it is received. When the buffer is empty, this is simply the propagation delay of the packet. When the packet is queued in the buffer, the one-way delay will increase to include the time it spends in buffer. Thus, we estimate t_{buff} by taking the difference between the currently observed one-way delay from the minimum one-way

delay observed in the recent past. Because what matters is not the absolute value but the relative increase of the one-way delay, the clocks for the sender and receiver need not be synchronized.

Like the receive rate, t_{buff} can also be estimated using TCP timestamps, as illustrated in Figure 4.3(b). The sender will timestamp the packet with time t_s when sending the packet. The packet will be queue in the buffer, and then be transmitted and arrive at the receiver. The receiver will timestamp the ACK with its receiving time t_r, and in addition, echo the sending time t_s. Thus, the sender can easily compute the relative one-way delay as $RD = t_r - t_s$ from the returned ACK, and estimate t_{buff} by subtracting the minimum observed one-way delay RD_{min}.

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

4.1.4 Adapting to Changes in Underlying Network

Our algorithm tries to match the sending rate σ to the receiving 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 under-estimation of the number of packets in the buffer and result in the algorithm switching to buffer drain 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 in the link as the BDP increases, which will in turn cause the buffer levels to drop while 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. Naïvely 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 fails 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 test the current network conditions.

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 in our implementations while we wait for the feedback from the initial small burst. We feel it is better to err on the side of caution than to cause further conges-

tion, but like other parameters, a particular algorithm might decide on some other value.

When the feedback is received, it gives us the new receive rate estimate ρ'. If ρ' is close to the previous value ρ, it indicates that the network bandwidth did not actually change and the buffer is either empty or a competing flow reduced the rate momentary. Either way, RD_{min} is updated and the state is switched back to buffer fill state with the new $\rho = \rho'$.

If ρ' is indeed much lower than ρ, this most likely indicates that the link bandwidth had reduced and the buffer was not yet empty. We return to the buffer drain state with the updated $\rho = \rho'$ without changing RD_{min}.

In addition, we also switch to the monitor state if the algorithm spends too much time in the buffer-drain state. This indicates that somehow the buffer is not appearing to drain even though we are attempting to do so.

4.1.5 Mechanism Parameters

By adjusting the parameters for our rate-based congestion control mechanism, we can tune the achieved throughput-delay tradeoff. This is similar to how different *cwnd*-based TCP variants can achieve different tradeoffs in performance. There are three key components that can be replaced or adjusted: i) the receive rate estimation algorithm, ii) the policy for adjusting the sending rate in each of the states, and iii) the policy for changing between the states.

Receive rate estimation. There are many possible ways to estimate the instantaneous receive rate. For example, a simple approach would be

to use buckets of fixed time intervals and compute the rate by dividing the total bytes received in each time interval by the size of the interval. More sophisticated approaches would be to use a sliding window of a fixed time interval to count the bytes received, or to use a fixed number of bursts [76]. We can also use existing methods [71, 73] to forecast the receive rate instead of simply measuring the instantaneous rate.

Sending rate. After obtaining an estimate of the receive rate ρ, our rate-based mechanism needs to determine the sending rate according to the current state of the algorithm. The sending rate in the Buffer Fill state σ_f needs to be set to a value larger than the estimated receive rate in order to fill the bottleneck buffer. On the other hand, the sending rate in the Buffer Drain state σ_d needs to be set to a value lower than the receive rate in order to drain the buffer. We provide the flexibility to specify a sending rate in the Monitor state σ_m that is different from the Buffer Fill and Buffer Drain rates. As the network is probed for changes during the Monitor state, σ_m should be set to a relatively low rate in practice.

The design of the rate-based mechanism allows the sending rate to be abstracted as a separate module that allows different new rate-based TCP variants to set their sending rates according to their needs. For example, if low latency is required, then a new rate-based TCP variant might choose to fill the buffer more slowly and drain the buffer more quickly compared to another variant that tries to maximize throughput.

The current Linux TCP congestion control implementation does not enforce any specific policy on how the *cwnd* value be set, e.g., CUBIC does not follow the AIMD scheme. Similarly, our rate-based TCP stack does not

impose any constraints on how the sending rates should be set. For example, a new rate-based TCP variant could very well decide not to fill the buffer even in the Buffer Fill state, if so desired.

Switching Between States. Congestion is detected when the estimated buffer delay t_{buff} increases over a threshold T_f , and is said to have eased when it drops below the threshold T_d. These two thresholds are different to allow some hysteresis to be introduced if necessary. To avoid uncontrolled oscillations, the constraint $T_f \geq T_d$ is imposed. Because there is delay in the feedback due to queuing, these two parameters will also affect the latency and throughput tradeoff. Larger threshold values would ensure the buffer does not underflow when entering the Buffer Drain state, but at the cost of introducing longer queuing delays in the buffer.

Other parameters. There are other optional parameters that can be set by the congestion control algorithm, namely the initial burst of n packets and the timeout to switch from the buffer drain state to the monitor state. These parameters are optional to reduce the complexity of implementing custom congestion control algorithms and the default values will be used if they are unspecified. The default value of n is set to 10 as the typical initial TCP receiver window is only 10 packets in size. In addition, Dukkipati et al. from Google Inc. recently argued for the TCP initial congestion window to be increased to 10 which they claim reduces latency without causing congestion [20]. The default value for the monitor state timeout is set to $4 \times RTT$ in our implementations as the sending rate parameters chosen would not take longer than that to empty the buffer.

Table 4.1: Basic API functions for rate-based mechanism.

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

4.2 Implementation

We implemented our rate-based TCP congestion control mechanism in the Linux kernel (version 3.2.24) in a manner that is similar to the existing *cwnd*-based TCP congestion control modules. The three components described in Section 4.1.5 are abstracted as API functions so that new rate-based TCP variants can be implemented as new kernel modules. The three functions are summarized in Table 4.1.

4.2.1 `update`

The `update(`*timestamp*`,` *bytes_received*`)` function is a callback that provides the module with the timestamp together with the amount of data received by the receiver, as estimated from the TCP timestamps in the ACK packets. The timestamp value provided is not the current system timestamp, but the `TSval` of the ACK packet, which corresponds to the time at which

the receiver had received the given amount of bytes. This function will return the latest estimate of the receive rate ρ.

4.2.2 `get_rate`

The `get_rate(`*curr_state*`,` *rate*`)` function obtains the current rate at which packets are to be sent from the module. The current state of the algorithm, i.e., Buffer Fill, Buffer Drain or Monitor, is supplied as an argument so the congestion module can return the corresponding sending rate. The latest receive rate ρ returned from the `update` function is also supplied as an argument. This function is called at every kernel tick to allow the congestion algorithm to adjust the sending rate even when there are no updates from the receiver. The current time can be obtained from a global variable and thus need not be passed in as an argument.

4.2.3 `threshold`

The `threshold(`*curr_state*`)` function returns the threshold value for the specified state. As before, the current state is passed as an argument so the module can return different threshold values for different state, if so desired.

Figure 4.4 compares the API interactions of the traditional *cwnd*-based TCP stack in the Linux kernel to that for our rate-based stack. While both of them allow a congestion control module to determine the sending rate, the regulation of the sending rate is done differently. Traditional *cwnd*-based congestion control modules will adjust the *cwnd*, which determines the number of unacknowledged packets that the TCP stack can have in flight

at any time and the sending of new data packets is clocked by the received ACK packets. In our new rate-based stack, the congestion control module explicitly sets a rate, and packets will be sent continuously at a given rate as long as there are available packets to be sent.

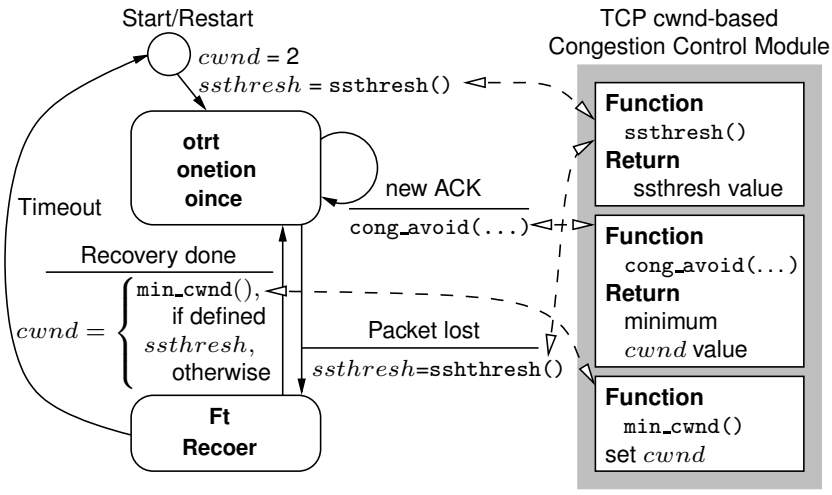
4.3 Linux Kernel Module

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

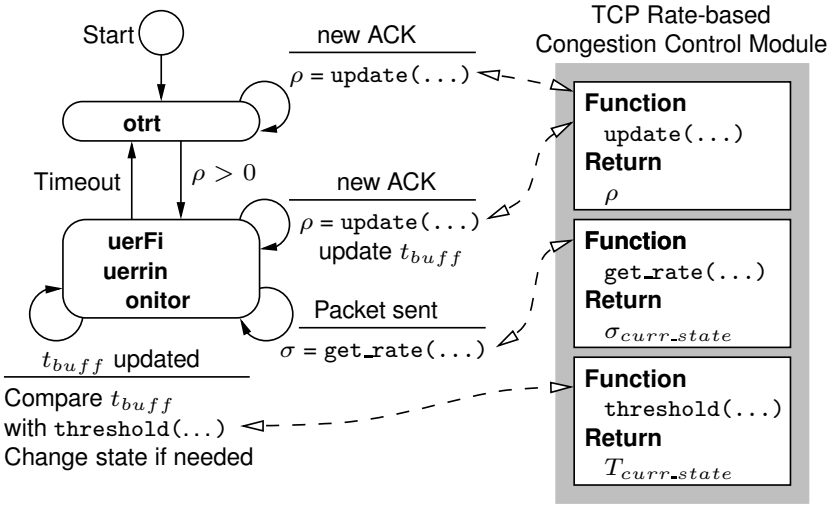
4.3.1 Sending of packets

The kernel function `tcp_write_xmit` is called whenever there is a possibility to send a new data packet. Originally, this function is called whenever there is new data from the application or an ACK packet is received. Because our rate-based mechanism is not ACK-clocked, a new timer is introduced to call this function every tick to allow new packets to be sent continuously.

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



(a) *cwnd*-based Congestion Control



(b) Rate-based Congestion Control

Figure 4.4: Comparison of API interactions between traditional *cwnd*-based congestion control and rate-based congestion control modules.

up for the rounding discrepancy over a few ticks. We modify the function `tcp_transmit_skb` to also perform this logging as it constructs and sends every TCP packet to the network driver.

4.3.2 Receiving ACKs

The function `tcp_ack` is called to handle every ACK packet that is received. The original kernel function examines the acknowledgement number as well as any SACK information and determines the number of packets that is being acknowledged. We simply added our rate-based mechanism to this function and use the reported timestamp values to compute the one-way delay and call the `update` function of the congestion control module. When packets are lost, SACK is used to determine the number of newly received bytes. In order to avoid code duplication, we added hooks into the TCP SACK processing routine, specifically by replacing the function `tcp_sacktag_one` to pass this information directly to our module.

4.3.3 Handling packet losses

When there are packet losses, the Linux TCP stack enters the Fast Recovery state, where lost packets are automatically retransmitted based on the SACK information. It then estimates the number of current outstanding packets in the network and sets the *cwnd* to activate fast retransmit and prevent the *cwnd* growth from stalling. Various fast recovery schemes have been developed [7, 48] and the current algorithm implemented in the Linux kernel is PRR [19].

For our rate-based mechanism, there is no need for a special Fast Recovery state. The packets to be retransmitted are simply given priority to be sent at the current sending rate. This works because congestion is inferred from the estimated buffer delay, and not from packet losses. As cellular networks typically have rather large buffers [76], the buffer delay would have to exceed a reasonable threshold before buffer overflow can happen. However, to be conservative, we switch momentarily to Buffer Drain state when we encounter a packet loss. If successive packet losses cause the algorithm to stay in the buffer-drain state for a long time, it will eventually trigger a transition to the monitor state. Thus, we did not have to specifically design routines to handle packet losses, and this significantly reduces the complexity of the recovery mechanism.

`tcp_retransmit_skb` is called whenever the TCP stack has a packet to retransmit when receiving a third duplicate-ACK or information from SACK blocks. As mentioned earlier, the packet is given priority to be retransmitted immediately in order to prevent delays to the receiving application. This packet is also accounted for in the send history.

`tcp_retransmit_write_queue` is called when there is a retransmit time-out and it originally sends a *cwnd* worth of packets. We have modified it to pace the retransmission of the packets according to the current state of the rate-based algorithm.

4.3.4 Practical Deployment

Our rate-based congestion framework requires minor modifications to the TCP stack at the TCP sender, 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 [24]) and disabled by default for Windows Mobile phones. In this light, we believe that the majority of existing smartphones are compatible with our framework.

The current architecture of existing 3G/4G 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 [45] connections. These proxies are used to improve 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 customized TCP congestion algorithms such as our rate-based framework for an entire network easily by modifying the TCP stack on these proxies as illustrated in Figure 4.5.

4.4 Summary

To evaluate our proposed rate-based congestion mechanism, we implemented a number of rate-based TCP variants. In the next two chapters, we present

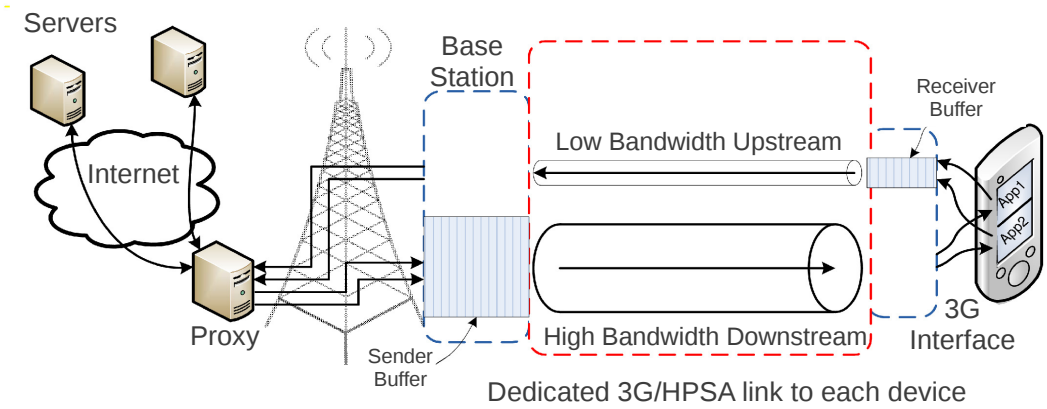


Figure 4.5: Illustration of proxy-based deployment.

two of rate-based congestion control algorithms which we have developed: *Receiver Rate-Estimate* (RRE) and PropRate. RRE was designed to maintain high throughput performance comparable to CUBIC while overcoming egregious ACK delays in a saturated or slow return link. PropRate is a proof-of-concept that shows how throughput can be traded for reduced latency using a simple algorithm. In addition, we implemented two recently developed congestion control algorithms for mobile networks which uses forecasting techniques as modules or our framework.

Chapter 5

Improving Link Utilization

In this chapter, we describe one of the algorithms that we developed for the rate-based congestion control framework called *Receiver-Rate Estimate (RRE)*. RRE was the first TCP congestion control algorithm we developed that uses the receiver-rate estimation technique described in our framework, hence its name. In the following sections, we first explain the parameters chosen for RRE and then evaluate its performance.

5.1 Parameters

RRE was designed to achieve comparable throughput to CUBIC while mitigating the effect of egregious ACK delays. In order to ensure optimal throughput, the buffer of the bottleneck link must remain filled, thus always having packets to send whenever possible. Thus, the objective of RRE during the buffer fill and drain state is to ensure that there are always packets in the buffer as the buffer delay oscillates around the threshold value T. In

practice, the number of MTU-sized 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. In RRE, we set the sending rate to maintain the buffer between a given B_{min} and B_{max} value.

5.1.1 Sending Rate σ

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, hence we can estimate B from t_{buff} as follows:

$$B \times MSS = \rho \times t_{buff} \quad (5.1)$$

$$B = \frac{\rho \times t_{buff}}{MSS} \quad (5.2)$$

where MSS is the maximum segment size, typically 1,500 bytes.

(a) Buffer Fill State ($B < T$). First, we set the send rate such that the buffer starts to fill. Before the number of packets in the buffer B reaches T, we will set the send rate $\sigma > \rho$ 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 (5.2). In other words, we keep filling the buffer if:

$$t_{buff} < \frac{T \times MSS}{\rho} \quad (5.3)$$

We determine σ_f by analyzing the evolution of the buffer as shown in Figure 5.1. It is clear from the figure that once B reaches T, it takes $t_{buff} + RTT$ for the sender to receive the feedback. During this time, the buffer

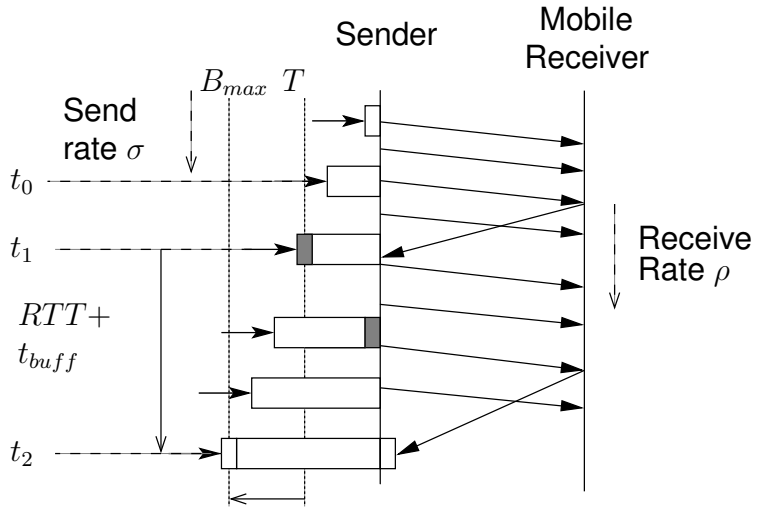


Figure 5.1: Evolution of buffer during buffer fill state.

would have increased in size at a rate $\sigma_f - \rho$, so

$$B_{max} = T + \frac{\sigma_f - \rho}{MSS} \times RTT \tag{5.4}$$

$$\sigma_f = \rho + \frac{B_{max} - T}{MSS \times RTT} \tag{5.5}$$

where B_{max} is the expected maximum number of packets in the buffer (to be discussed in Section 4.1.5).

(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 exceed the threshold value given in Equation (5.3). Once this occurs we will need to reduce the send rate σ_d so that it falls below ρ, as follows:

$$\sigma_d = \rho - \frac{T - B_{min}}{MSS \times RTT} \tag{5.6}$$

where B_{min} is the expected minimum number of packets in the buffer (to be

discussed in Section 4.1.5). This is completely analogous to the buffer fill state with the following caveat: the sending rate σ is set low at the start of this state, and is only allowed to increase, as the estimated receiving rate ρ fluctuates.

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. Therefore, σ is never allowed to further 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. We record the current sequence number upon each state change to aid in the monitoring of network conditions.

For the monitor mode, we simply set $\sigma_m = \frac{\sigma_d}{2}$.

5.1.2 Threshold T

Other than the size of the initial burst n, we need 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 algorithms slow to react to network fluctuations. Conversely, setting too low a T might inadvertently cause the buffer to empty, resulting in under-utilization 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. One solution is to make T a function of the bandwidth and RTT. We note that Nichols and Jacobson's CoDel uses $1 \times$ RTT as the threshold to invoke early dropping of packets in the buffer [52]. Because we know the receive rate ρ and the RTT, we can set $T = \rho \times RTT_{min}$, which is the estimated bandwidth-delay product (BDP) and it seems to work well in practice. Also, we used the same value of T in all the different states.

B_{max} and B_{min} determine the responsiveness of RRE and how fast it will converge to T on each oscillation. If the difference between them and T is large, RRE will respond with more aggressive changes in the send rate (See Equations (5.5) and (5.6)). 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. When the bandwidth is low, T might be too low that the buffer empties. We address this problem by simply setting a minimum value for T at 30 packets, which is much smaller than the buffer size implemented in the ISPs. 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 our downstream

buffers were sufficiently large. Instead, we set the lower and upper limit on B_{max} and B_{min} to $T + 10$ and $T - 10$ respectively.

5.1.3 Receive Rate ρ

The receive rate is estimated using an exponentially weighted moving average (EWMA) of the throughput measured from a sliding window. In a related study [76], we have found that a window of 50 packet bursts is adequate to estimate the instantaneous throughput. As such, we use a sliding window consisting of 50 consecutive and distinct timestamps with reported packet arrivals (which would contain at least 50 packets over at least 50 ms). We found that while this approach is suitable for very fast flows, the length of the sliding window can become as long as a few seconds when the throughput is low. Since using a long history of timestamps to estimate the throughput would hardly reflect the instantaneous throughput accurately, we limited the maximum length of the sliding window to a maximum of 500 ms.

It is common for several packets to have the same arrival timestamp even for slow throughput due to the bursty nature of cellular networks. Thus, the last timestamp is not included in the computation of the average throughput. In other words, the receive rate is only updated when an packet whose arrival time advances the sliding window is received.

5.2 Performance Evaluation

In this section, we present our evaluation of the TCP Receiver-Rate Estimation using both the `ns-2` simulator and also an implementation in the Linux

kernel. First, we evaluate how well RRE performs in the presence of a concurrent upload both in simulation and in a practical 3.5G/HSPA network. Next, we assess the TCP-friendliness of the new TCP variant by comparing it to CUBIC [27], which is the default TCP implementation in Linux and Android.

5.2.1 Evaluation with `ns-2` Simulation

We evaluated 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 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 using data obtained from our previous measurement study.

We evaluate 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 CUBIC [27], which is the current default TCP congestion control algorithm deployed in Linux and Android. We also evaluated TCP Vegas, which is delay-based, and TCP Westwood, which implements a form of receive rate estimation.

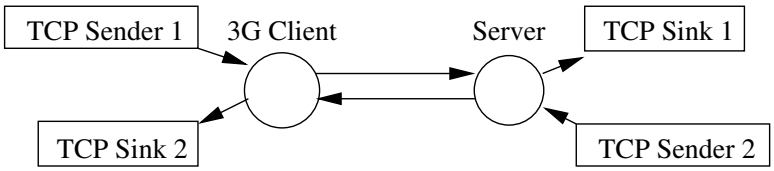


Figure 5.2: Network topology for `ns-2` simulation.

5.2.2 Network Model & Parameters

In our simulations, we use the simple dumbbell topology shown in Figure 5.2 to model the mobile wireless link, where we expect to find the typical bottleneck. While a typical connection from a mobile device to the Internet will involve more nodes and links, this suffices for us to obtain an understanding of RRE. 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 a measurement study to characterize the 3.5G/HSPA network of the three locally available mobile ISP, 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

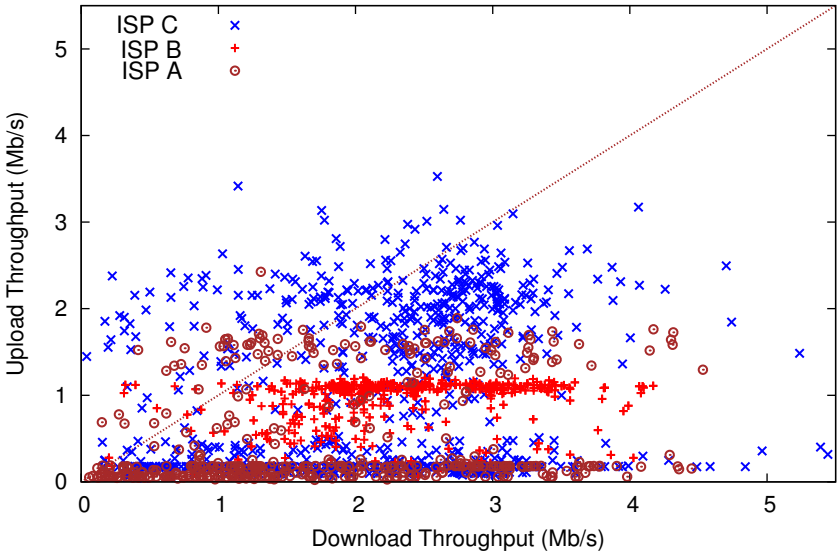


Figure 5.3: Scatter plot of the upstream and downstream throughput for different mobile ISPs.

data points over a period of several weeks, which we plot in Figure 5.3. 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 conditions, 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 certain locations. Another explanation is that there might be significant contention on the uplink due to a high volume of subscribers.

We observed RTTs that varied between 50 ms to 200 ms, so the RTT parameter for our simulations are also varied within this range. The observed packet loss rate was less than 0.04% overall, which concurs with Huang et al. measurements that packet losses over cellular networks are rare [30]. 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.

We set the downlink and uplink buffer sizes to 2,000 packets and 1 MB ($\approx$ 700 packets) in our simulations.

5.2.3 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 CUBIC. In Figure 5.4, 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 Figure 5.4). This threshold increases as the downlink bandwidth increases,

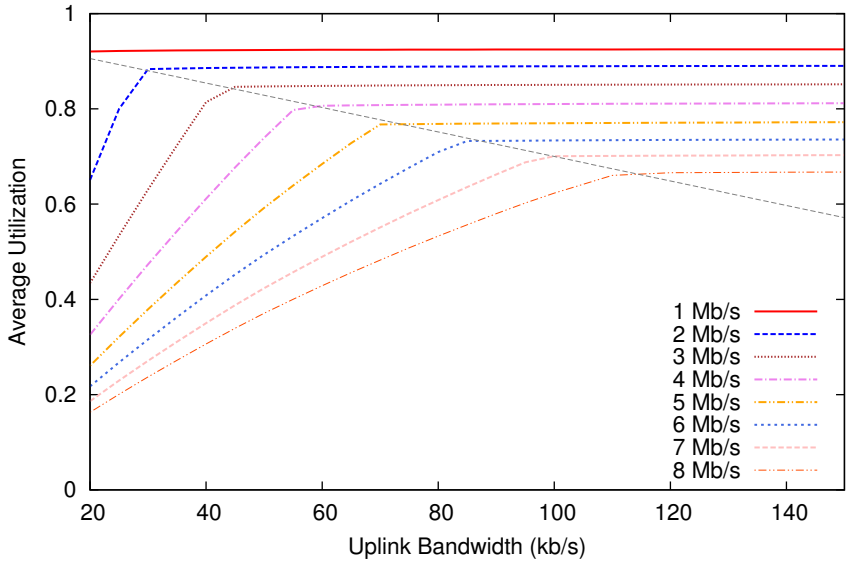


Figure 5.4: Plot of downlink utilization against uplink bandwidth for CUBIC.

since we need a higher rate of returning ACKs to clock the TCP sender.

RRE is specifically designed to address scenarios where the uplink is the limiting factor. To understand how 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 CUBIC and RRE for each of these configurations and plot the results in the scatter-plot shown in Figure 5.5. These results clearly demonstrate that RRE is able to achieve a higher goodput than CUBIC when the uplink is a much slower than the downlink. 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, RRE is able to achieve a downlink utilization close to 80% for 90% of the scenarios.

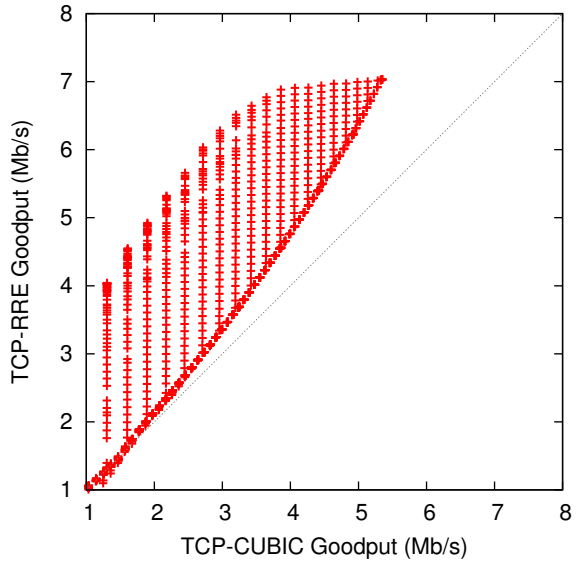


Figure 5.5: Scatter plot comparing downstream goodput of RRE to CUBIC.

5.2.4 Download with Concurrent Upload

Next, we investigate how RRE performs when the uplink is congested. To simulate a congested uplink, we simply start a single continuous upload using 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: RRE, TCP-Reno, CUBIC, TCP Vegas and TCP Westwood. The RTT was set at 100 ms.

In Figure 5.6, we plot the cumulative distribution of the ratio of goodput achieved by RRE against that for the other TCP variants on a pairwise basis. We make three observations: (i) the achieved goodputs for TCP-Reno,

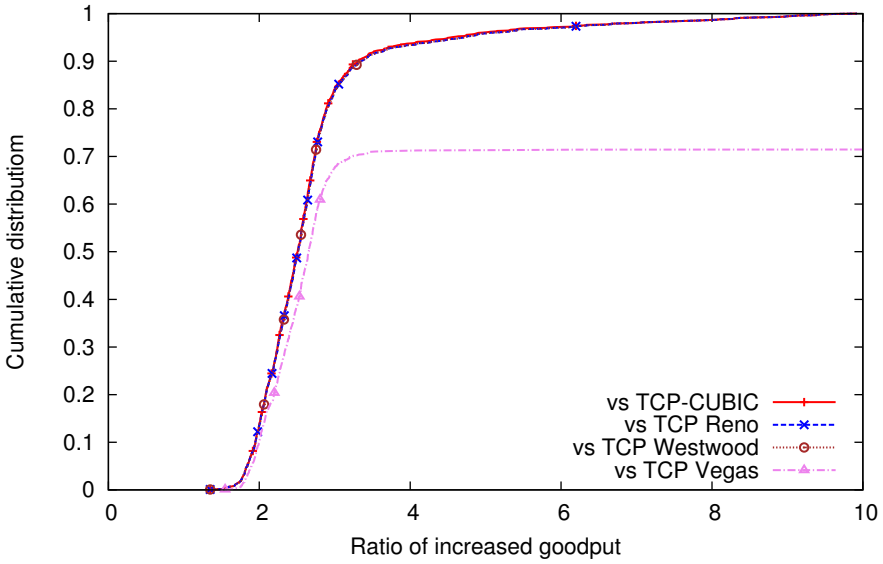


Figure 5.6: Cumulative distribution function of the ratio of RRE goodput to CUBIC and TCP-Reno, in the presence of a concurrent upload.

CUBIC and TCP Westwood are extremely similar. There are three distinct lines for these three algorithms in Figure 5.6, 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) 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.

5.2.5 Single Download under Normal Conditions

While we have shown that RRE performs as expected and can improve downstream TCP goodput under poor uplink conditions, we now examine how 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 in 0.5 Mb/s increments, and the uplink bandwidth was set at a level that is above the threshold levels described in Section 5.2.3. The RTT was set at 100 ms.

In Figure 5.7, 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 RRE performs slightly better than the other variants when the downlink bandwidth is high because 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 rate for sending. This can be clearly seen in Figure 5.8 where we plot the time traces of the various single TCP flows against time. We can see that the average goodput of 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 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 as per normal.

5.2.6 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 RRE handle changes in network conditions, we ran a long-

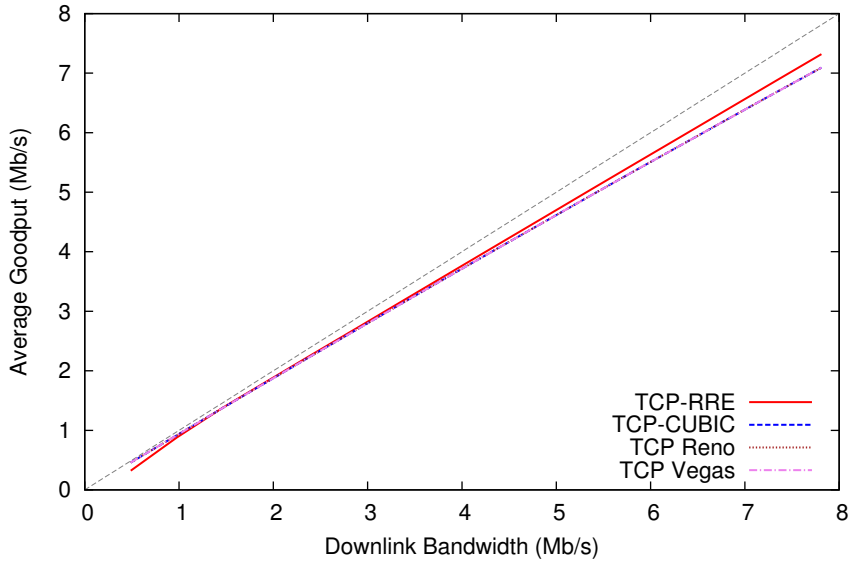


Figure 5.7: Plot of average downstream goodput against downstream bandwidth for different TCP variants.

lived RRE flow with a starting downlink bandwidth of 3 Mb/s and 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 Figure 5.9. We plot also a trace for CUBIC under the same conditions for comparison.

We see from these traces that the CUBIC has a relatively stable send rate, but it also keeps buffer occupancy relatively high. While the sending rate for RRE oscillates quite a bit, the achieved receive rate is comparable to CUBIC and relatively stable. RRE reacts to the changes rather quickly and typically converges to the correct send rate within a few seconds. The worst

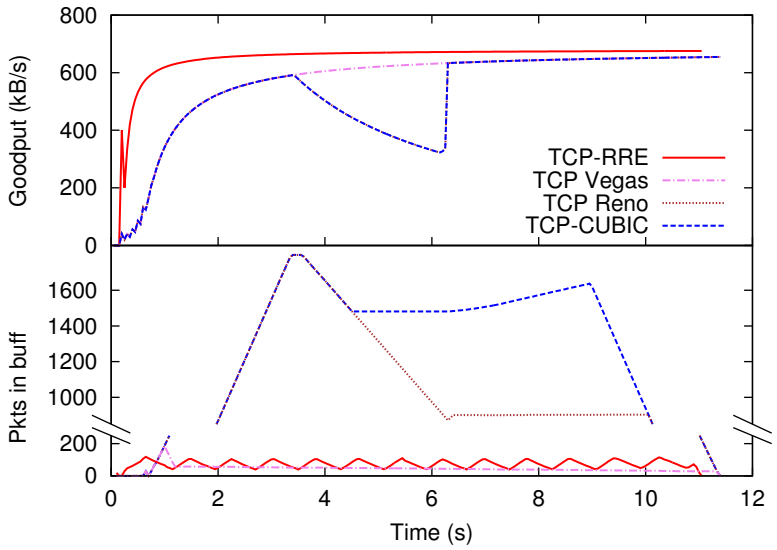


Figure 5.8: Sample time traces for different TCP variants.

response was at $t = 8$ when the RTT increases a second time. Nevertheless, by $t = 10s$, RRE has successfully detected the change in the network and adjusted its send rate accordingly.

5.2.7 TCP Friendliness

Next, we investigate how 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 1 s increments. The rest of the parameters were identical to those in Section 5.2.5.

We then computed the average goodput for each pair of flows and the associated Jain's fairness index [33], i.e. $\frac{(R_1+R_2)^2}{(2\times({R_1}^2+{R_2}^2))}$, where R_1 and R_2 are the throughput of the two flows. In Figure 5.10, we plot the cumulative distribution of the resulting data points. We make two interesting observa-

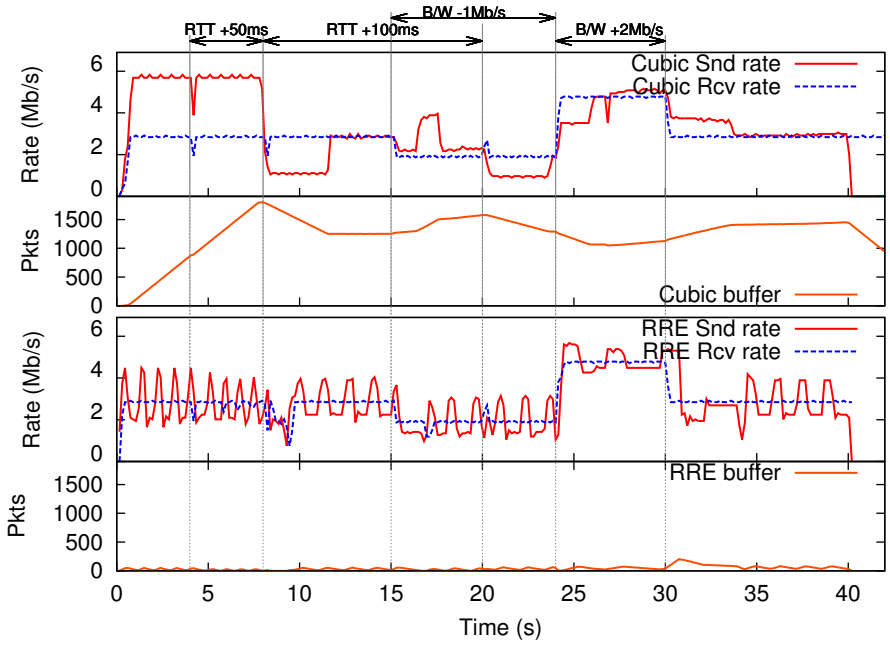


Figure 5.9: Time trace comparing how RRE reacts under changing network conditions to CUBIC.

tions: (i) 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 CUBIC and TCP-Reno; (ii) how well RRE contends with 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 RRE flow first, a subsequent CUBIC flow would aggressively flood the buffer and cause RRE to back-off and significantly reduce its rate below the "fair" rate. On the other hand, if a CUBIC flow starts first, a subsequent RRE flow is able to acquire a reasonably fair share of the available bandwidth.

Our results suggest that RRE, like delay-based congestion control algorithm, does not contend well against CUBIC, which means that RRE might

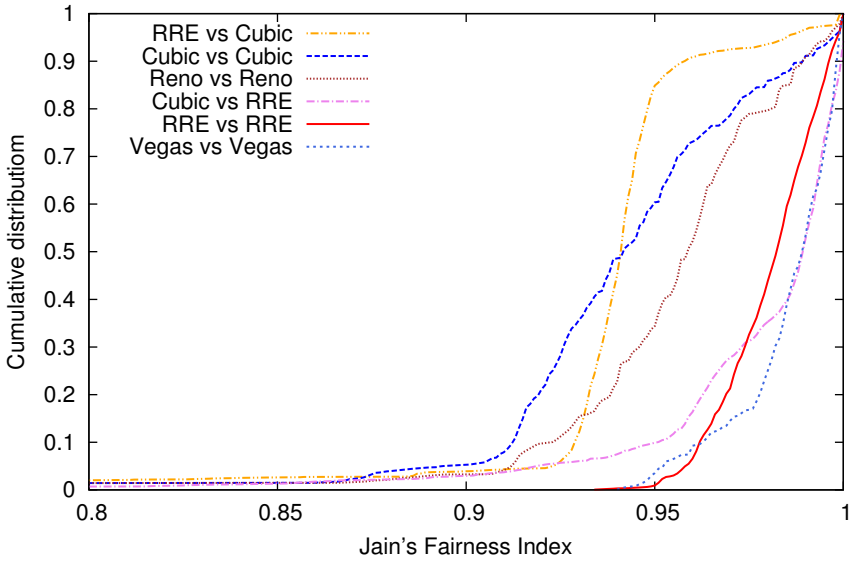


Figure 5.10: Jain's fairness index for contending TCP flows.

not be suitable for deployment "in the wild." However, because transparent proxies are commonly deployed in existing mobile ISPs, 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 in our evaluation of a Linux implementation on existing 3.5G/HSPA networks, we found that even if deployed on a server not within a mobile ISP, RRE is still often able to achieve better goodput than CUBIC under conditions where the uplink is poor or saturated.

5.2.8 Evaluation of the Linux Implementation

RRE was implemented as a kernel module for the Linux 3.2 kernel. The modified kernel was installed on a server in our lab and we evaluated it over the local 3.5G/HSPA networks. We ran sets of experiments at various

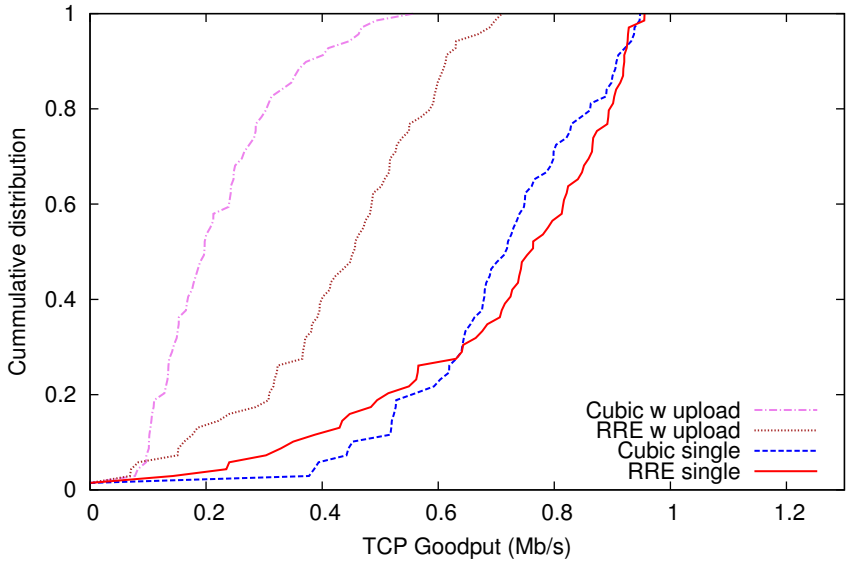


Figure 5.11: Cumulative distribution of measured downlink goodput in the laboratory for ISP A on HTC Desire.

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 and the newer Samsung Galaxy Nexus. One set of experiments consists of 4 tests: (i) a single CUBIC download, (ii) a single RRE download, (iii) a CUBIC download with a concurrent TCP upload, and (iv) a RRE download with a concurrent TCP upload. In the latter two tests, we started the download 10 s after we start 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, they are mainly similar, thus we only present three representative sets of data.

In Figure 5.11, we plot the results of our experiment carried out in our

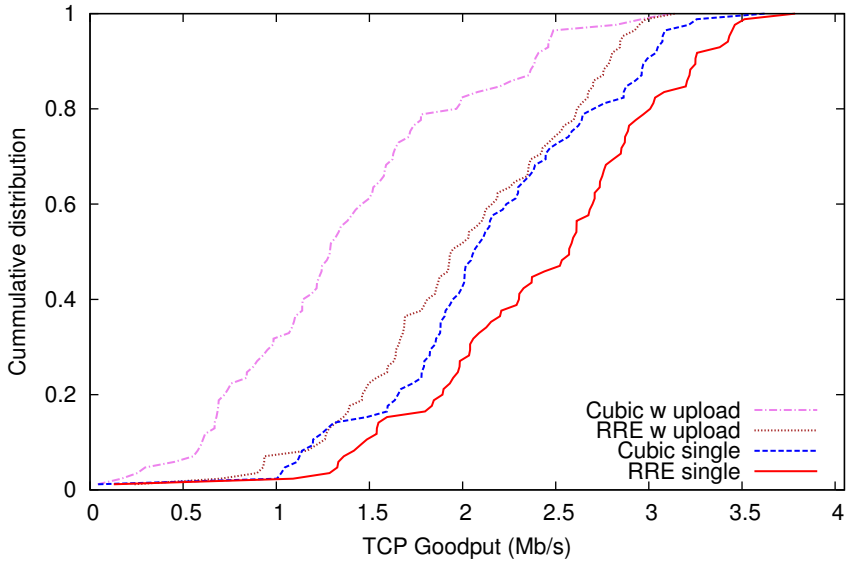


Figure 5.12: Cumulative distribution of measured downlink goodput in the laboratory for ISP C with Galaxy Nexus.

lab using the older HTC Desire phone over ISP A. The results show that the achieved goodput for a single TCP download for both CUBIC and RRE are comparable. However, in the presence of a concurrent upload, the goodput drops significantly, though the drop for CUBIC is much more significant than that for 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 Figure 5.12, we plot the results for experiments carried out in the same location, but using the newer Galaxy Nexus phone over ISP C. The downlink speeds were much higher but the uplink throughput was slower, with a median value of 500 kb/s. Finally, in Figure 5.13, we plot the results for experiments carried out at a residence over ISP C. The uplink at the

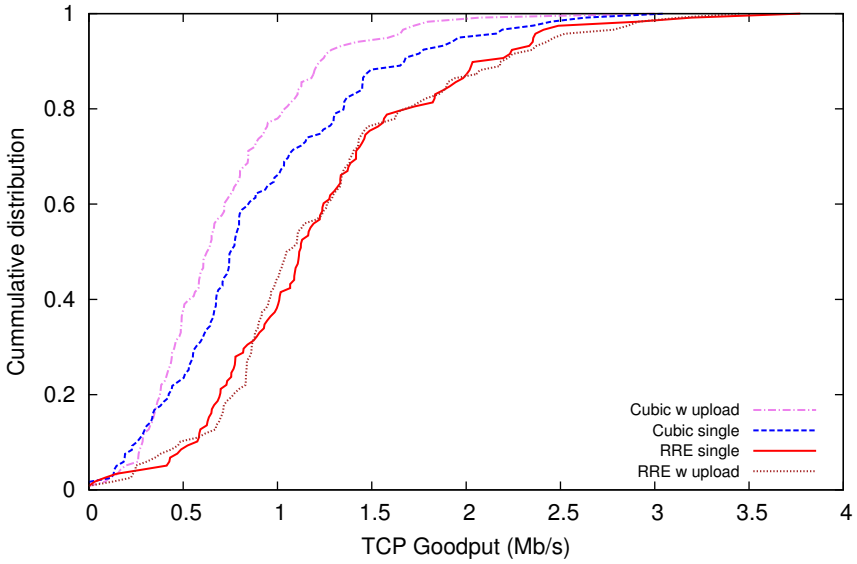


Figure 5.13: Cumulative distribution of measured downlink goodput at a residence for ISP C on Galaxy Nexus.

residence was even slower, with the median rate below 200 kb/s. While the data presented in Figures 5.11 to 5.13 were specially chosen to illustrate scenarios where RRE performed better than CUBIC, we do not mean to suggest that RRE always performs better. In experiments where the uplink bandwidth was high, the performance of RRE and CUBIC were comparable. We note however that in none of our experiments did RRE perform noticeably worse than CUBIC.

In summary, what our results for actual 3.5G/HSPA networks suggest is that 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 RRE was predicted to perform in our simulations some 2 to 4 times faster than 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 RRE flow and other TCP flows in the core Internet routers.

5.3 Summary

We have developed a rate-based congestion control algorithm called RRE, which works on our new TCP framework. RRE adjusts the sending rete so as to keep the buffer occupancy between two given parameters B_{min} and B_{max}. This ensures that the link is kept utilized and the throughput remains at the maximum value.

In this chapter, we showed that not only RRE is able to overcome the problem of egregious ACK delays in two-way concurrent flows, it performs no worse than CUBIC under regular link conditions. RRE also remains unaffected when the uplink return path is slow or congested. Our evaluation shows that RRE is friendly towards other TCP flows and can be feasibly deployed over real cellular data networks.

Chapter 6

Reducing Latency

Recently, there has been a focus on improving end-to-end network delay over cellular networks as it is often the dominant component of the overall response time [57]. Because cellular data networks often experience rapidly varying link conditions, they typically have large buffers to ensure high link utilization [76]. However, if the application or transport layers send packets too aggressively, the saturation of the buffer can cause long delays [19]. Sprout [71] and PROTEUS [73] were recently proposed to address this problem by forecasting the network conditions.

In line with this focus, we show that our rate-based congestion control framework is able to also achieve similarly low delays, while maintaining a much higher throughput, by adjusting the parameters of our rate-based algorithms. To this end, we developed PropRate, a more simplistic version of RRE and show that a fast feedback mechanism works just as well as current forecasting methods. In addition, we also implemented Sprout and PROTEUS as congestion control modules in our framework to show the adapt-

ability of our framework in incorporating new algorithms and techniques.

6.1 Implemented Algorithms

To evaluate our proposed rate-based congestion mechanism, we implemented a number of rate-based TCP variants. As a proof-of-concept, we developed PropRate, a simple proportional-rate congestion control algorithm to demonstrate that our proposed rate-based approach works well. Sprout [71] and PROTEUS [73] are recently proposed algorithms for cellular data networks, and they are essentially different approaches for determining the data sending rate. As such, we investigate how their rate-computation algorithms perform under our proposed rate-based TCP stack. We implemented them as kernel modules that can be loaded into our rate-based framework.

6.1.1 PropRate

PropRate uses the same exponentially weighted moving average (EWMA) computation as RRE in computing the receive rate. Their difference lies in the sending rate and the threshold value. In PropRate, the sending rate σ is set at a fixed function $k\rho$, which is a rate proportional to the estimated receive rate ρ, hence the name PropRate. In the Buffer Fill state, $k > 1$, and in the Buffer Drain state, $k < 1$. We allowed the sending rate σ_d in the buffer drain state to fluctuate, instead of keeping it constant as in RRE. This might cause the buffer to drain too quickly but it allows PropRate to also optimize for delay in addition to throughput.

The thresholds T_f and T_d were set to the same constant T. We show

later in Section 6.2.5 that PropRate can achieve a whole range of tradeoffs along a performance frontier by varying the proportion parameter k and the threshold T.

6.1.2 PROTEUS-Rate

The key idea of PROTEUS [73] is to use a regression tree to forecast the available bandwidth from a history of past samples. PROTEUS computes the instantaneous throughput in disjoint time windows of 500 ms. Then, using a history of 64 time windows, it constructs a regression tree and uses it together with the sending rate of the current window, to forecast the available bandwidth, and thus the number of packets to send, in the next time slot. We implemented the PROTEUS forecasting algorithm as a module called *PROTEUS-Rate*.

Like PropRate, we set the sending rate for PROTEUS-Rate to a fixed proportion of the forecasted rate. We also use a fixed value for T_f and T_d. The PROTEUS algorithm produces relatively good forecasts. Its main drawback is that the suggested parameters of 500 ms and history of 64 time windows [73] require a long initialization time of 32 s to initialize the regression tree. We experimented with shorter time windows and less history to reduce this initialization time, but we found that doing so adversely affected the achieved performance.

6.1.3 Sprout-Rate

We obtained the source code of Sprout from the author's public repository [71], and ported their stochastic forecast algorithm as a kernel module called *Sprout-Rate*. The basic idea of the Sprout algorithm is to model the link as a doubly-stochastic process, where the mean λ of a Poisson process varies every tick according to a Brownian motion with the mean being the previous λ and standard deviation fixed at 200 packets per second per $\sqrt{\text{second}}$. It uses a tick length of 20 ms and the forecast is updated at the end of each tick after observing the number of packets received in the tick.

Although the source code from the authors was freely available, the default Sprout implementation runs in the Linux user space using math libraries and non-blocking IO. We encountered some difficulties in porting it directly as a loadable Linux kernel module and had to make a few modifications. First, we had to develop our own floating point routines to perform non-integer calculations and to compute the Poisson and Gaussian distribution functions in the kernel. The original Sprout algorithm can take up to 90 s to initialize on a 1.66-GHz Atom processor and at least a hundred megabytes of memory to save the Gaussian distribution. By carefully selecting integer values of common factors and hard-coding the standard distributions into the source code, we reduced the start-up time significantly to a few milliseconds and the memory usage was reduced to only a few megabytes.

Second, we found that the original Sprout implementation took between 6 to 10 ms to compute the forecast for each tick, even though it is running in user space with full access to floating point instructions. To prevent blocking,

the forecasting routine runs in a separate thread from the network IO. As we do not have the luxury of floating point instructions and non-blocking IO in the kernel, the forecast routine would block the CPU for more than 50% of the time. To address this, we increased the tick duration to 100 ms so a computation time of 10 ms will only block the CPU for 10% of the time. The 100 ms tick duration also allows the algorithm to be less sensitive to high-frequency network variations.

Third, the original Sprout implementation uses a *cwnd*-like mechanism to clock the sending of packets. In particular, it forecasts the number of packets to send in the next 8 ticks, and sends 5 ticks worth of packets in one burst, which are then used by the receiver to estimate the receive rate. To use their forecast in our module, we computed the sending rate by dividing the forecast over the number of ticks.

6.2 Evaluation

The network conditions of cellular data networks can vary greatly even over short periods of time [75]. To ensure a consistent comparison while maintaining the delays and variations of cellular data networks, we use a trace-driven network emulator as opposed to the `ns-2` simulator. We use the network emulator Cellsim, which was also used by Winstein et al. in evaluating Sprout [71]. The tool forwards packets across two network interfaces according to the packet trace it is given. The Cellsim source obtained from their public repository implements an infinite buffer. Because traditional *cwnd*-based algorithms like CUBIC react to packet loss due to buffer over-

flow, we modified the Cellsim tool to introduce a finite drop-tail buffer for fairer evaluation.

We obtained network traces from three local cellular ISPs using a custom Android application on a Samsung Galaxy S3 LTE smartphone, with the network saturated by UDP packets. Similar to our measurement study, `tcpdump` was used to capture the packet traces. We know from our measurement study that UDP could be used to estimate the available network bandwidth because we did not find any evidence of UDP traffic being throttled by the ISPs. We collected two sets of traces from each of the three local ISPs. One set was obtained with the phone in a stationary position and the other set was taken on board a vehicle that was driven around campus. We followed the evaluation methodology and used the same emulation parameters as Winstein et al. [71], and we used both the uplink and downlink traces in the network emulator. `iperf` was used to send and receive TCP traffic.

We also evaluated the various algorithms over a real LTE network using the Samsung Galaxy S3 LTE phone as the receiver. However, due to the high throughputs of our LTE networks and the limited data quota of our data plans, we could only conduct a limited number of runs for each experiment on the real LTE networks.

In our evaluation, we compared the traditional TCP congestion control algorithms, namely CUBIC, Westwood and Vegas against our rate-based congestion control modules PropRate, PROTEUS-Rate, Sprout-Rate and RRE. In addition, we were also able to evaluate the original Sprout implementation using the source code obtained from their public repository.

Table 6.1: Parameters used for rate-based TCP variants.

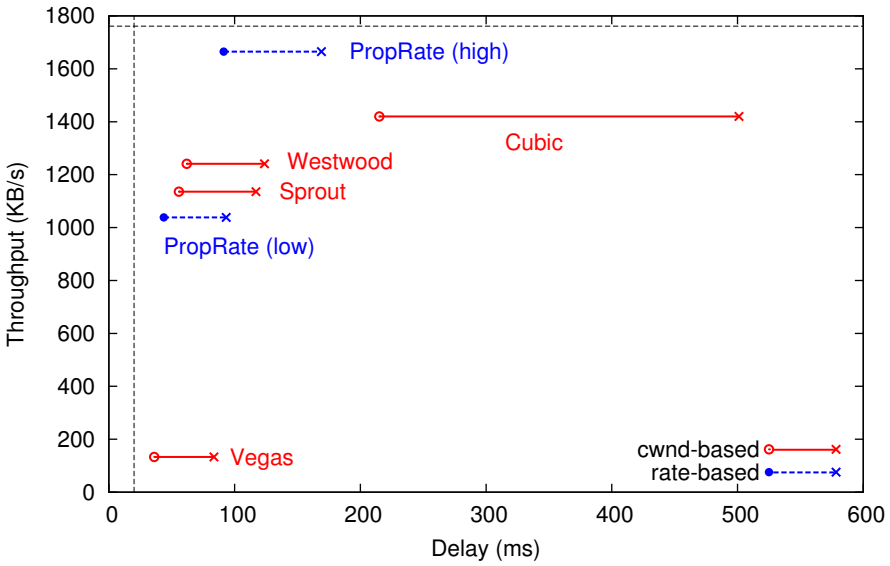
Algorithm	T_f & T_d	σ_f	σ_d
PropRate (High)	60 ms	1.25ρ	0.75ρ
PropRate (Low)	20 ms	1.25ρ	0.25ρ
Proteus-Rate [73]	50 ms	1.00ρ	0.25ρ
Sprout-Rate [71]	50 ms	2.00ρ	0.50ρ
RRE [42]	40 ms	$(1 + \frac{60-40}{40+RTT})\rho$	$(1 - \frac{40-20}{40+RTT})\rho$

6.2.1 Algorithm Parameters

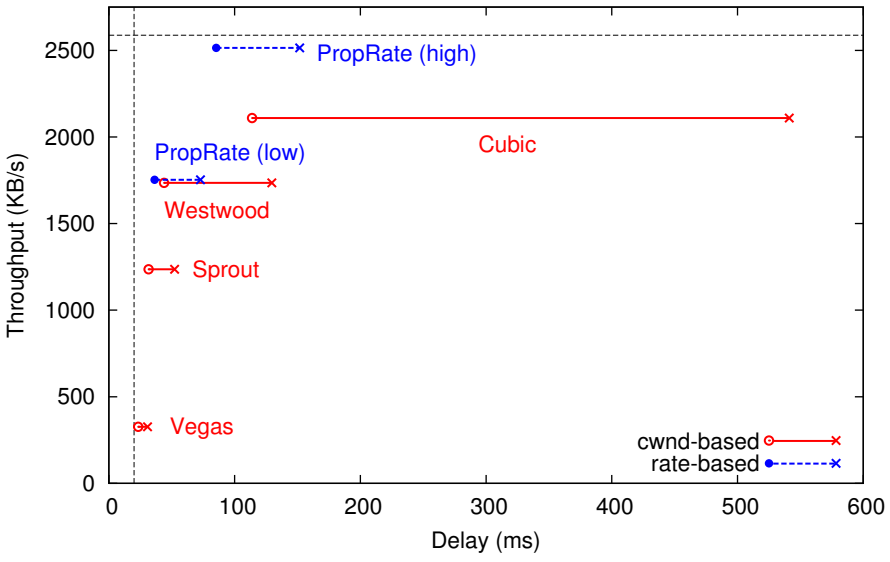
The congestion control algorithms that we evaluated can be divided into two categories: i) the *cwnd*-based algorithms, namely CUBIC, Westwood, Vegas and Sprout; and ii) TCP variants based on our rate-based mechanism, namely PropRate, RRE, PROTEUS-Rate and Sprout-Rate. We consider the Sprout [71] algorithm to be a form of *cwnd*-based algorithm because even though it is implemented in UDP, it adopts a *cwnd*-like counter to clock the sending of packets. Sprout, Sprout-Rate and PROTEUS-Rate are algorithms that attempt to do stochastic forecasting. Table 6.1 summarizes the parameters for the rate-based algorithms.

We investigated two canonical variants of PropRate, which we call PropRate (high) and PropRate (low). The parameters for PropRate (high) is intended to achieve a high throughput, and thus it uses a fixed threshold of 60 ms, and has a low drain rate to prevent the buffer from emptying completely. On the other hand, PropRate (low) is optimized for low delays and uses a fixed threshold of 20 ms, which is half the minimum RTT, and a much higher drain rate to avoid bufferbloat.

For PROTEUS-Rate, we found that with the recommended window size

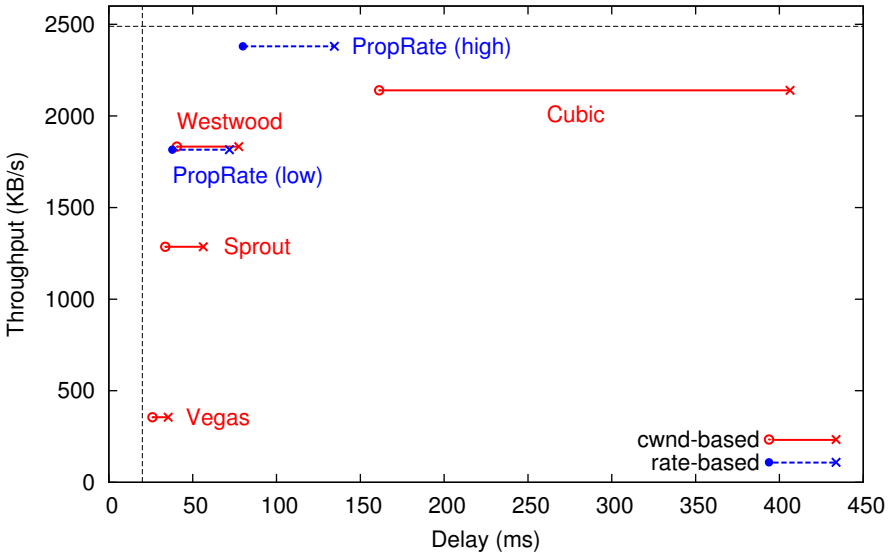


(a) ISP A, Stationary

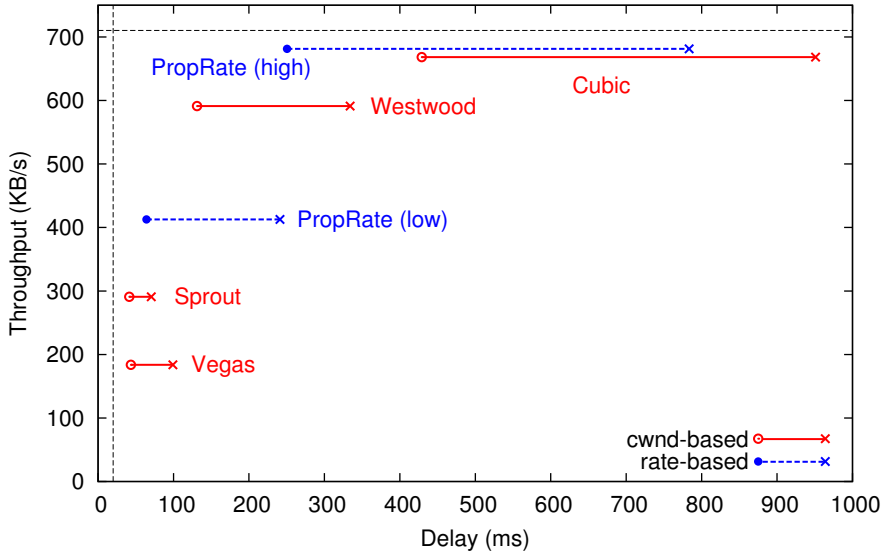


(b) ISP A, Mobile

Figure 6.1: Performance of various algorithms for ISP A traces.

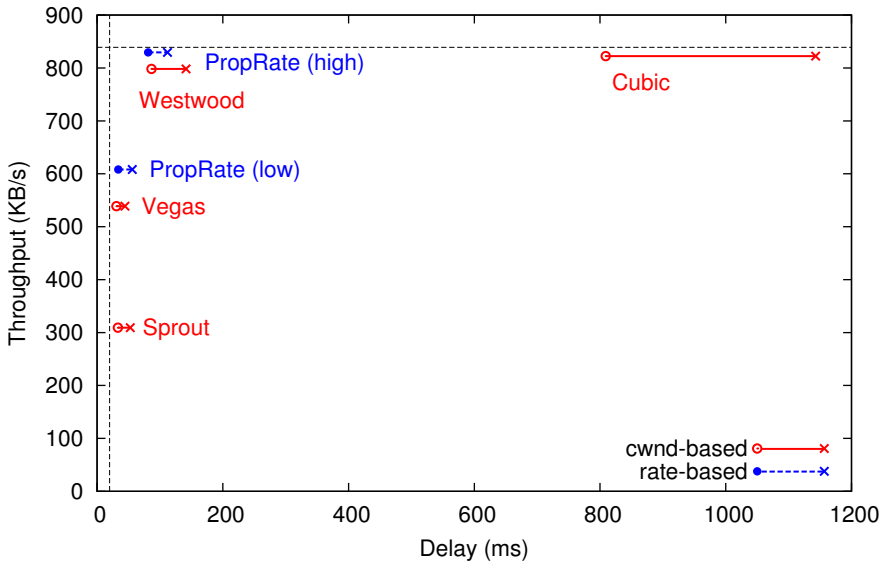


(a) ISP B, Stationary

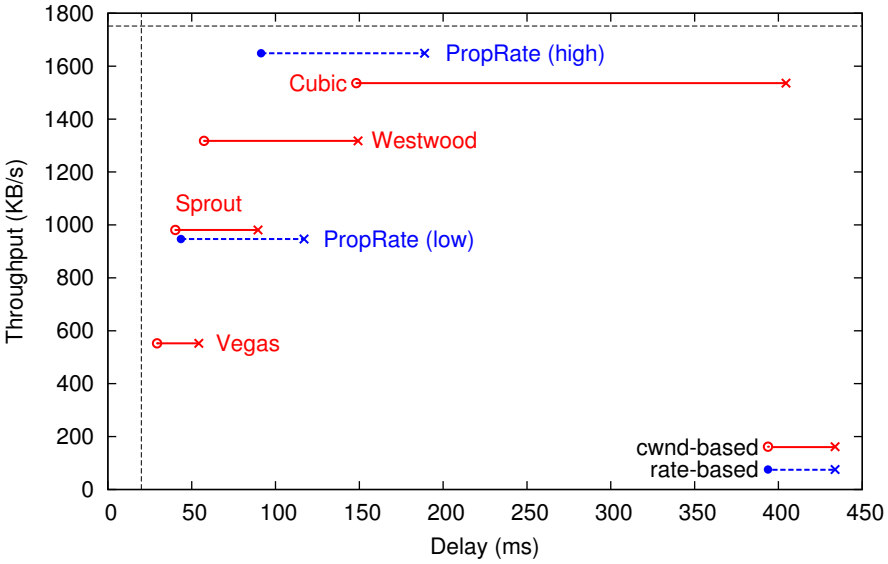


(b) ISP B, Mobile

Figure 6.2: Performance of various algorithms for ISP B traces.



(a) ISP C, Stationary



(b) ISP C, Mobile

Figure 6.3: Performance of various algorithms for ISP C traces.

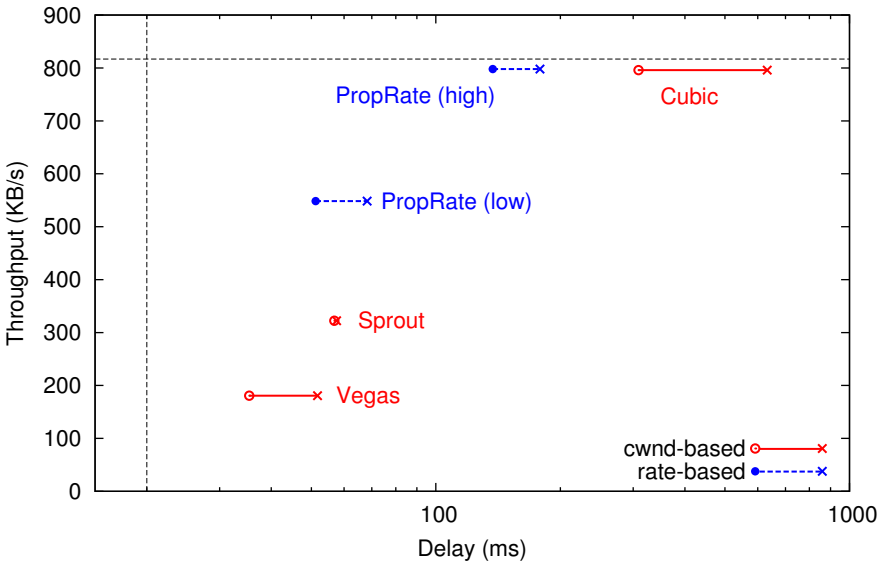
of 500 ms [73], the forecasted bandwidth tends to be slightly higher than the actual bandwidth. Thus, we simply used the forecasted rate for σ_f. We also found that setting σ_d to 0.25ρ allows us to keep the delay low.

For Sprout-Rate, we found that we had to use a rate that was double its forecast in order to get comparable performance. This is because the original Sprout implementation uses a burst of 5-ticks worth of packets to estimate the receive rate while our rate-based mechanism sends packets at a steady rate.

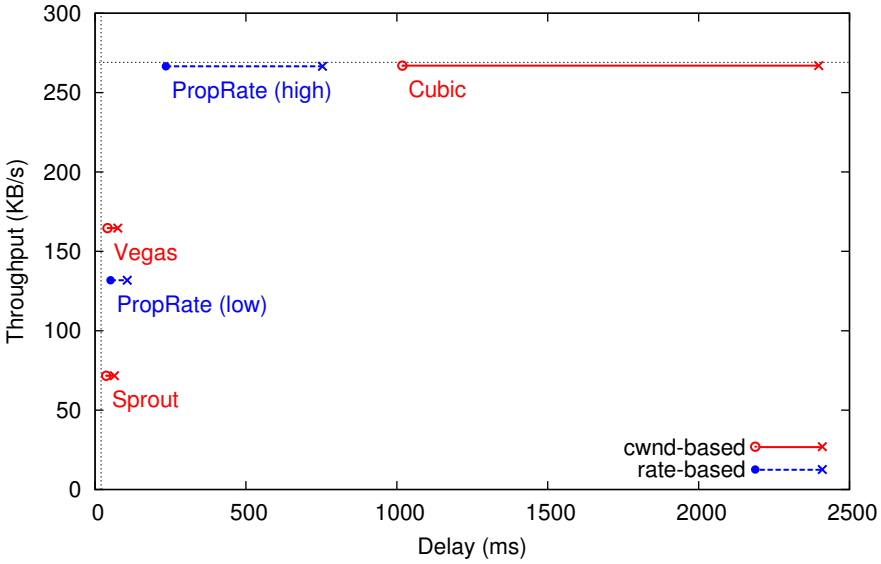
RRE uses a default threshold value that is the minimum RTT, thus we set it to 40 ms. B_{max} and B_{min} are to be set at $\pm\frac{1}{2}$RTT, so we set them to 60 ms and 20 ms respectively. Note that the difference between RRE and PropRate is that the sending rates for the former are expressed in terms of the currently observed minimum RTT, while PropRate uses fixed constants. Finally, we set $T_f = T_d$ and also set $\sigma_m = 0.5\sigma_d$ for all the rate-based algorithms.

6.2.2 Trace-based Emulation

In Figure 6.1, we plot the one-way packet delay against the total average throughput of different algorithms for stationary and mobile traces for three ISPs. The circles in Figure 6.1 (and also subsequent figures) indicate the mean values while the crosses indicate the 95th-percentile values. The 95th-percentile value follows the evaluation metric adopted by Winstein et al. [71]. The dotted lines indicate the maximum throughput and minimum latencies for each trace. The results for PROTEUS-Rate do not include the long initial training period before forecasting begins.

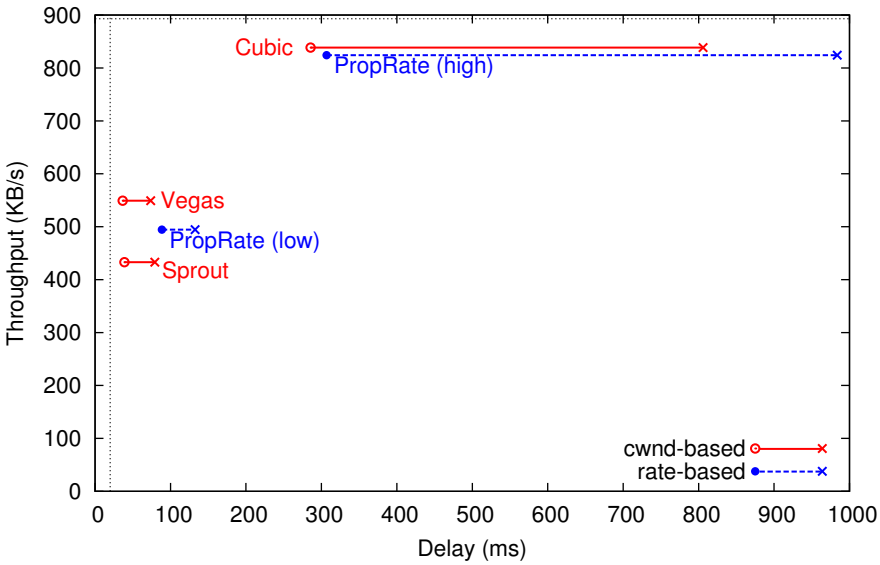


(a) AT&T

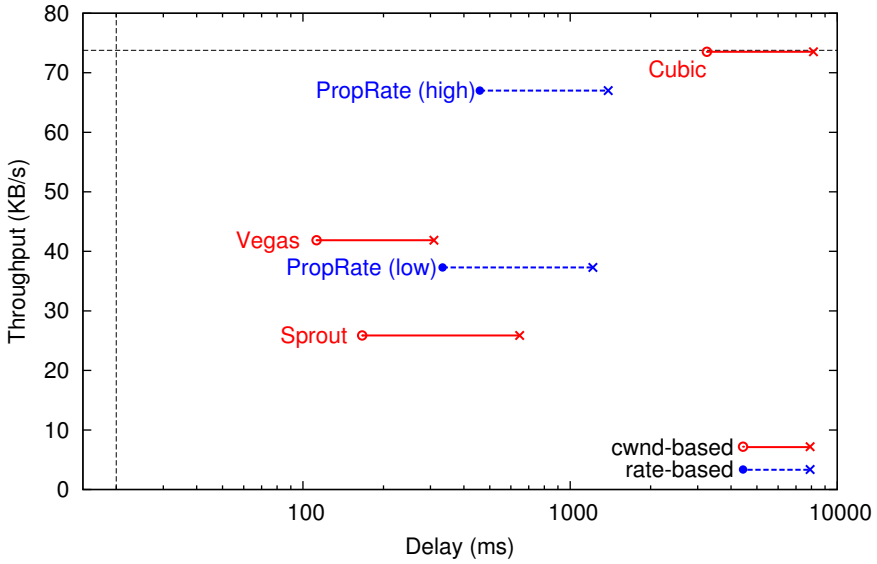


(b) T-Mobile

Figure 6.4: Results using MIT Sprout (mobile) traces [71].

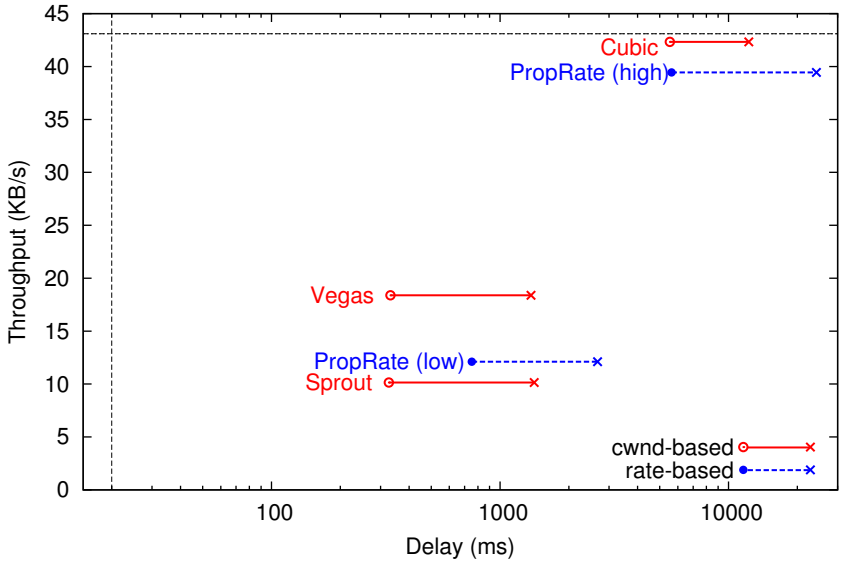


(c) Verizon-4G



(d) Verizon-3G

Figure 6.4: Results using MIT Sprout (mobile) traces [71].



(e) Sprint

Figure 6.4: Results using MIT Sprout (mobile) traces [71].

Our results show that among the traditional *cwnd*-based algorithms, CUBIC achieves a high throughput but also results in higher delays, while Westwood is surprisingly good. Westwood achieves lower delays than CUBIC at slightly lower throughputs. Another surprising observation was that Vegas outperformed Sprout in terms of latency for all the traces. Both had relatively low throughputs and neither seemed superior to the other in terms of throughput for stationary traces. Among the rate-based algorithms, we see that PropRate (high) is able to achieve high throughputs that are relatively close to the optimal value. PropRate (low) is able to achieve low delays comparable to Sprout and Vegas, but at a higher throughput.

Mobile Traces. If we compare the results between the stationary and mobile traces, we can see the maximum available throughput is different for the various ISP traces. The performance for the various algorithms for

the mobile traces is very similar to that for the stationary traces, except that there is typically an increase in the variance of the latencies. Sprout is exceedingly robust and is able to achieve latencies with relatively low variance for mobile traces. We also note that for some ISPs, the mobile traces have higher throughputs than the stationary ones.

We also repeated our experiments using the mobile traces used by Winstein et al. [71], which were collected by driving around Boston. We present the results in Figure 6.4. The main difference between our local traces and the MIT Sprout traces is that the Sprout traces have significantly lower bandwidth. Even though we used the same emulator as Winstein et al., our results for Sprout are slightly different from those presented in [71]. We checked our experimental setup many times and communicated with the authors of [71], and found the key difference was that their evaluations were done on an Amazon EC2 cluster [70]. Our emulation experiments were performed over real physical networks and servers, and we suspect that the difference between the results could be due to the network effects arising from virtualization [69].

We found that PropRate (low) does not perform as well in terms of delay as the original Sprout algorithm for some traces like Verizon-3G and Sprint in terms of latency, so we set out to investigate why. A detailed examination of the MIT Sprout traces reveals that periods of complete outages (i.e., zero bandwidth) were relatively common, unlike our local mobile traces where there was constant connectivity at significantly higher bandwidths. Thus, we believe that much of the performance gains for the Sprout algorithm arising from it being highly optimized to tolerate and recover from complete outages. We believe that PropRate can be further optimized to better cope

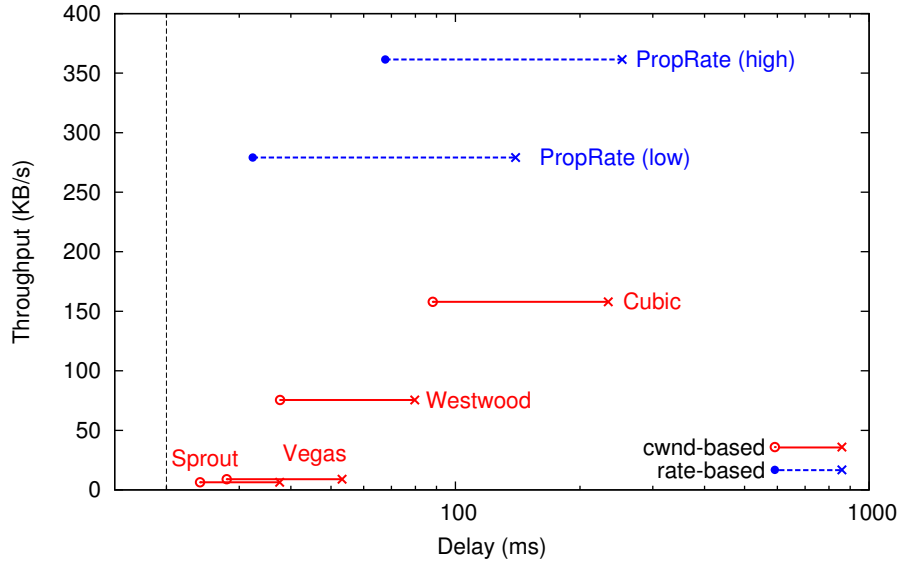


Figure 6.5: Downstream throughput and delay in the presence of a concurrent upstream TCP flow for ISP C.

with periodic complete network outages.

6.2.3 Problem of Congested Uplink

It has been shown that the performance of mobile networks can degrade significantly when there is congestion in the uplink [75]. To simulate a congested uplink, we started a TCP CUBIC uplink flow simultaneously with the downlink flow that was measured. In Figure 6.5, we present the results for the ISP C mobile trace. The uplink bandwidth in this trace was 1.5 Mb/s, which was roughly one-tenth of the downlink bandwidth. Comparing this with Figure 6.3(b), we can see that the background uplink flow will significantly degrade the performance of the downlink by at least 75% for all the algorithms. We see however that our rate-based algorithms are significantly more resilient than the *cwnd*-based algorithms. Both Vegas and Sprout are

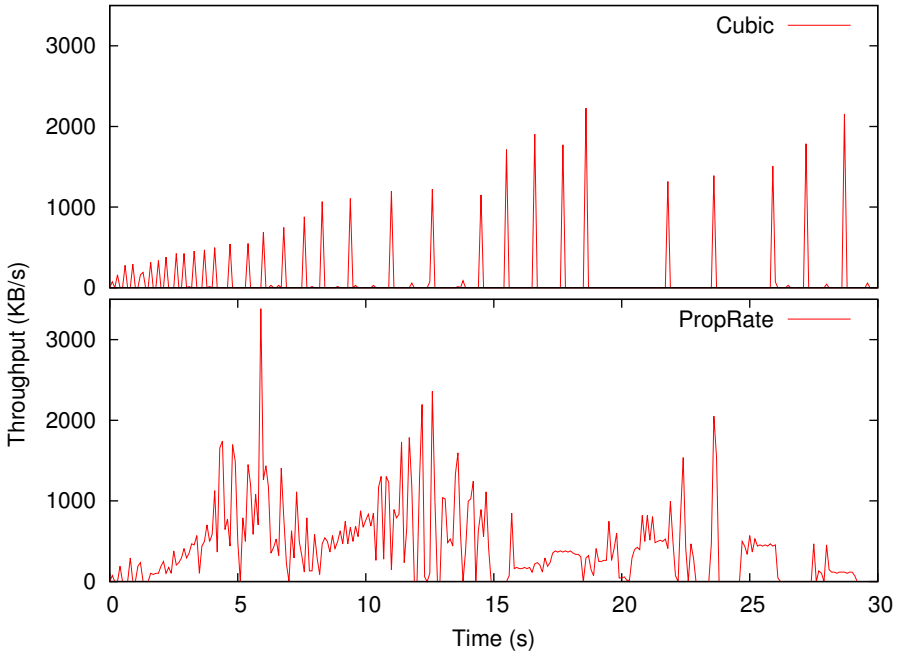


Figure 6.6: Trace of the downstream sending rate for flows in Figure 6.5.

effectively starved.

Figure 6.6 shows the sending rates for CUBIC and PropRate (high) over time for the same network trace. We can see that as expected, the ACK-clocked CUBIC algorithm is very bursty as it sends new data packets only when the ACKs are received. PropRate is however able to maintain a somewhat more steady sending rate. We found that the gaps that appear in the PropRate trace are caused by the default TCP receive window, which prevents packets from being sent once there is one receive window worth of unacknowledged packets. Such gaps will to be reduced if the receive window is set to a higher value.

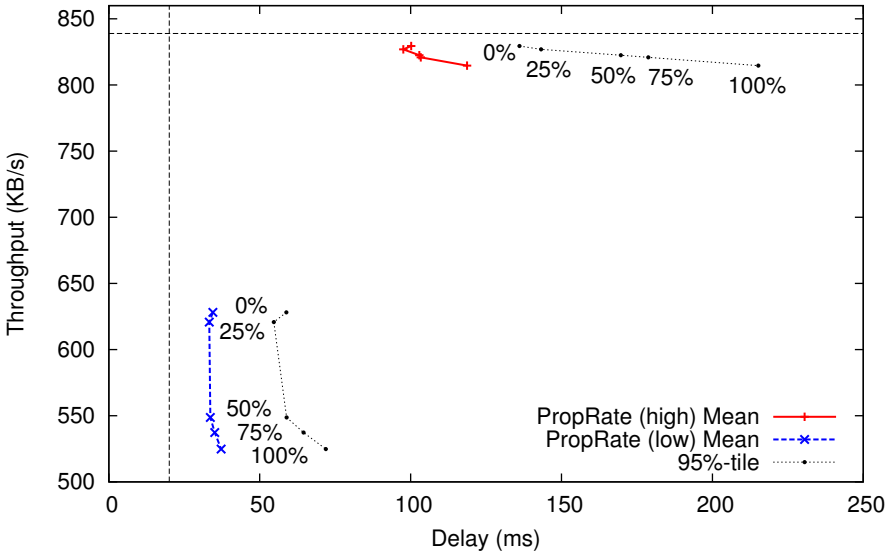


Figure 6.7: Performance when errors are introduced to the rate estimation.

6.2.4 Robustness to Rate Estimation Errors

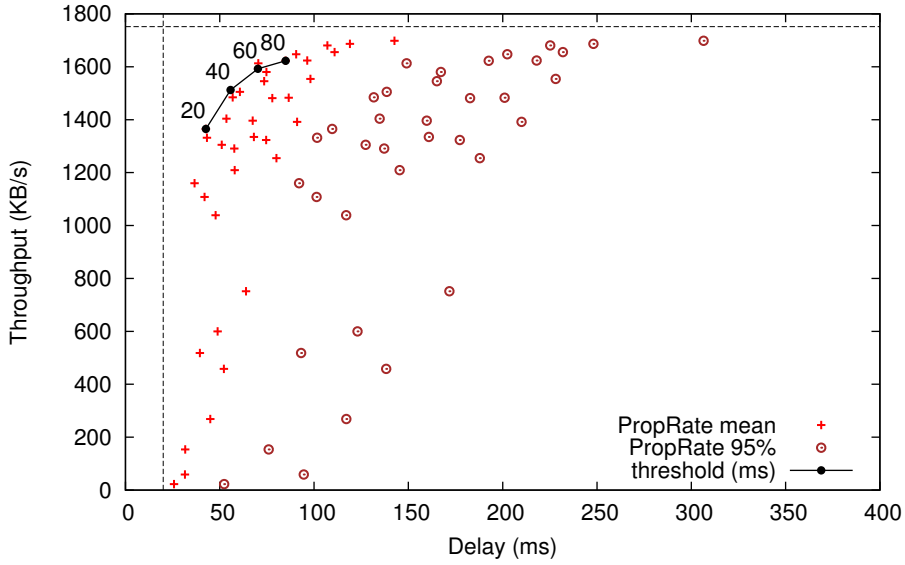
Figure 6.7 shows the results of PropRate (high) and PropRate (low) when we introduced errors into the estimated rate. The introduced error is expressed as in terms of the coefficient of variance of the estimated rate, i.e., ratio of the standard deviation (of the error) to the mean estimated rate. We found that the errors have a marginal effect on the latency, though larger errors will slightly reduce the throughput. Degradation was about 10% when the standard deviation of the errors introduced was up to 100% of the mean of the original estimates. This is not surprising since such errors merely increase the frequency of the switching between the Buffer Fill and Buffer Drain states.

6.2.5 Performance Frontiers

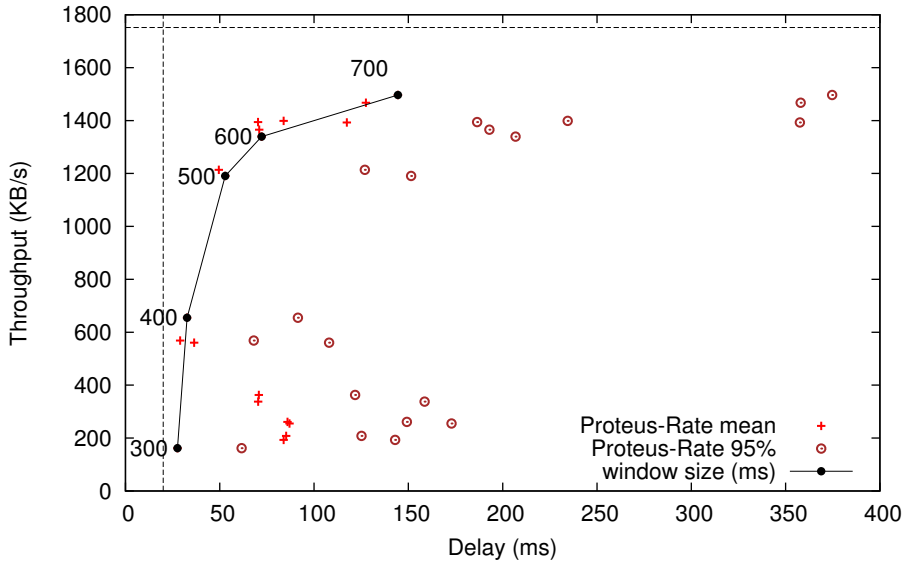
We next investigate how promising our rate-based mechanism is to achieve a good tradeoff point between latency and throughput for mobile applications. We also investigate the effectiveness of forecasting algorithms like Sprout and PROTEUS. To this end, we ran variants of our rate-based algorithms with a wide range of parameters on the ISP C mobile traces to obtain the scatterplots in Figure 6.8. From these plots, we can see the corresponding performance frontiers of the various rate-based algorithms, and that PropRate seems to be able to achieve a frontier that is close to optimal.

PropRate. The results for PropRate in Figure 6.8(a) are obtained by varying the threshold value T and the multiplier constant k for the proportional increase or decrease of the sending rate. In addition, we also plot the points on the frontier corresponding to the threshold values T from 20 ms to 80 ms. Clearly, the achieved latency is lower with a smaller threshold, at the cost of reduced throughput. The throughput however will drop off quite steeply when the threshold T is reduced below 20 ms.

PROTEUS-Rate. PROTEUS-Rate has some extra proprietary parameters which can be adjusted. They are the size of each observation window and the length of training history. We highlight the points on the performance frontier that correspond to different window sizes. The remaining points are obtained by varying the threshold and drain rates. We see our rate-based approach can achieve somewhat lower latencies by using the PROTEUS forecasting algorithm, at the cost of slightly lower throughput compared to PropRate. The main drawback of PROTEUS-Rate is that it requires a relatively

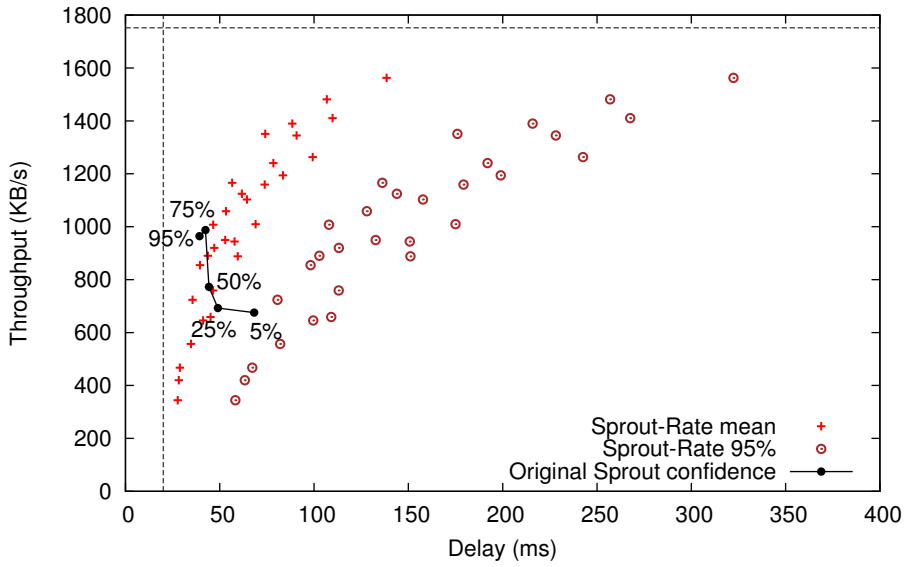


(a) PropRate

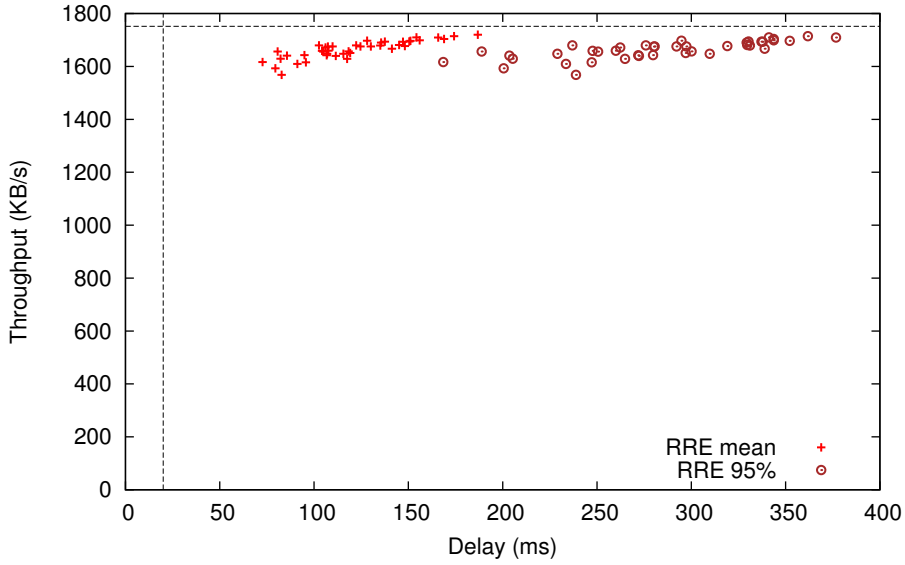


(b) PROTEUS-Rate

Figure 6.8: Performance frontiers achieved by different algorithms with the ISP C mobile trace.



(c) Sprout & Sprout-Rate



(d) RRE

Figure 6.8: Performance frontiers achieved by different algorithms with the ISP C mobile trace.

long training time of about 30 s. Reducing this training time will significantly reduce the accuracy of the forecast.

Sprout-Rate. As evident from Figure 6.8(c), we found that by using the Sprout forecasting algorithm, we again can achieve lower latencies than PropRate, but at a cost to the achieved throughput.

It turns out that the Sprout algorithm incorporates a confidence parameter [71] and we followed the methodology used by Winstein et al. to run the original Sprout algorithm on the same trace while varying this parameter. The varying of this confidence parameter produces a nice performance frontier for the MIT Sprout trace (which we could reproduce and verify). However, this was not the case for our ISP C mobile LTE trace. We found that this is because when a lower confidence parameter was used, the forecast would be too high and this causes more packets than required would be sent in a burst. This significantly increases the delay. Our ISP C mobile trace had significantly higher throughputs than the MIT Sprout trace. This meant that too many packets were sent in a single burst, such that no packets would be sent for several subsequent ticks. This causes the receiver to incorrectly infer that there is a network outage. As a result, this leads to severe oscillations in the forecasts, thereby negatively impacting latency. In other words, for the ISP C mobile trace, reducing the confidence parameter actually makes latency worse (higher) instead of improving it.

RRE. We varied the threshold value T and both the B_{min} and B_{max} variables specified in the algorithm and plot the results in Figure 6.8(d). Though these variables indirectly control the fill and drain rate like PropRate, it was not possible to sufficiently drain the buffer to achieve low latencies by

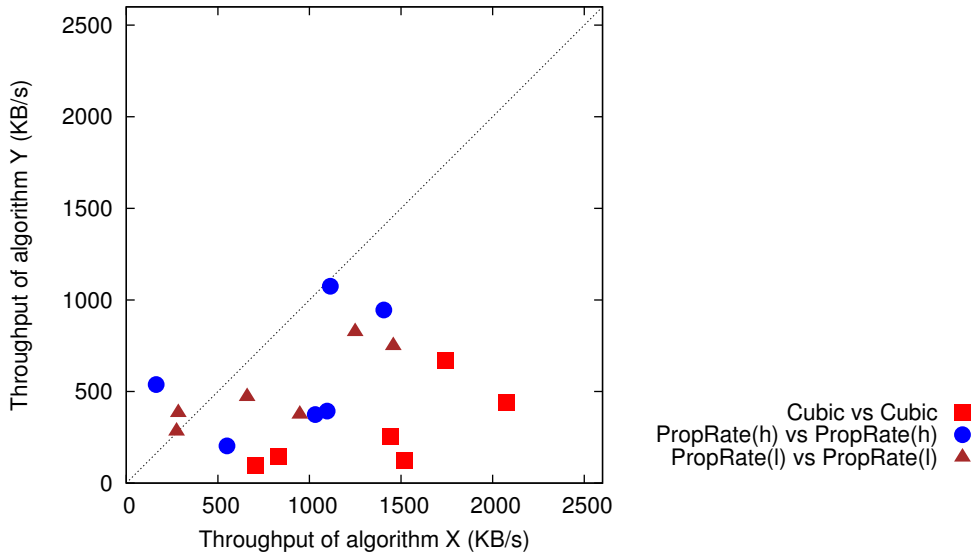
just varying these parameters. This shows that our initial design of RRE was indeed to maximize for throughput rather than delay.

6.2.6 TCP Friendliness

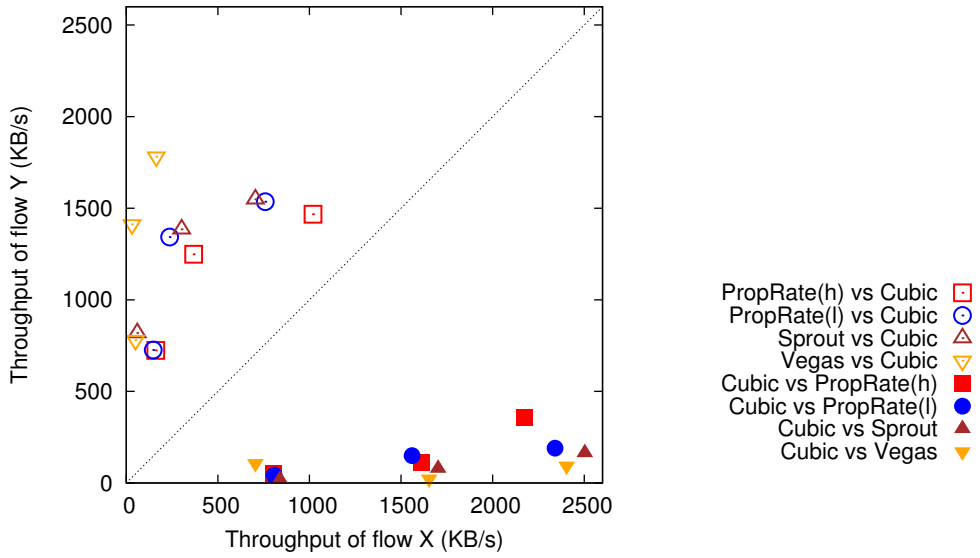
If a rate-based TCP variant, like PropRate, were to be deployed in the wild, it would have to interact with existing *cwnd*-based TCP variants. So, it is important to understand how TCP variants based on our rate-based stack contend with existing *cwnd*-based TCP variants like CUBIC. In each experiment, we first started one TCP flow and then another parallel flow after 30 s. Both flows were then measured for another 30 s, and we computed the average throughput for each flow during the latter 30 s time period.

As a baseline, we first evaluated how two flows of the same algorithm would affect each other and we plot the results in the Figure 6.9(a). A point on the diagonal axis will represent perfect fairness and sharing by the two flows. We see that CUBIC is in fact not very fair when contending with another CUBIC flow and that PropRate is relatively friendly to itself. Because like Sprout [71], we envision that PropRate will be deployed at the mobile link, it is important that PropRate be fair when there is self-contention.

Next, we compared PropRate (high), PropRate (low), Sprout and Vegas to CUBIC for the available stationary traces for three ISPs (hence, the distinct bands) and plot the results in Figure 6.9(b). We found that the results depended on which flow starts first. CUBIC is rather aggressive and if it starts first, then the throughput of the other algorithm is significantly



(a) Self-contention.



(b) Contention against CUBIC.

Figure 6.9: TCP friendliness of Flow X versus Flow Y. Flow Y was started 30 s after Flow X.

reduced, though PropRate still achieves slightly higher throughputs than Sprout and Vegas. If the CUBIC flow were to start later, then PropRate and Sprout are able to obtain a significantly larger share of the available bandwidth than that in the earlier case. As noted by Winstein et al., it is hard to achieve low latencies if there is aggressive cross traffic at the bottleneck link [71]. While PropRate will certainly achieve lower latencies than CUBIC by trading off some throughput and contending less aggressively, how low we can go ultimately depends on the size of the bottleneck buffer. However, our results do suggest that PropRate is potentially usable even when there is cross traffic, especially if it can adjust its "aggressiveness" dynamically.

6.2.7 Practical 4G Networks

Finally, we compared the performance of PropRate to existing algorithms over a real cellular data network. Like the emulation experiments, we used `iperf` to start a 90-second long TCP transfer from our server to a Samsung Galaxy S3 LTE smartphone over an LTE network. We could not get Sprout to compile for Android, so we instead tethered the phone to a laptop that could run Sprout. As our LTE networks have very high bandwidth, we could only run a limited number of tests before exhausting the data quota of our plans.

Since real cellular network conditions are not always stable even at a stationary position, we repeated each experiment several times for each algorithm and plotted the trace with the highest throughput for a fairer comparison. We plot the results for the ISP A LTE network in Figure 6.10. The

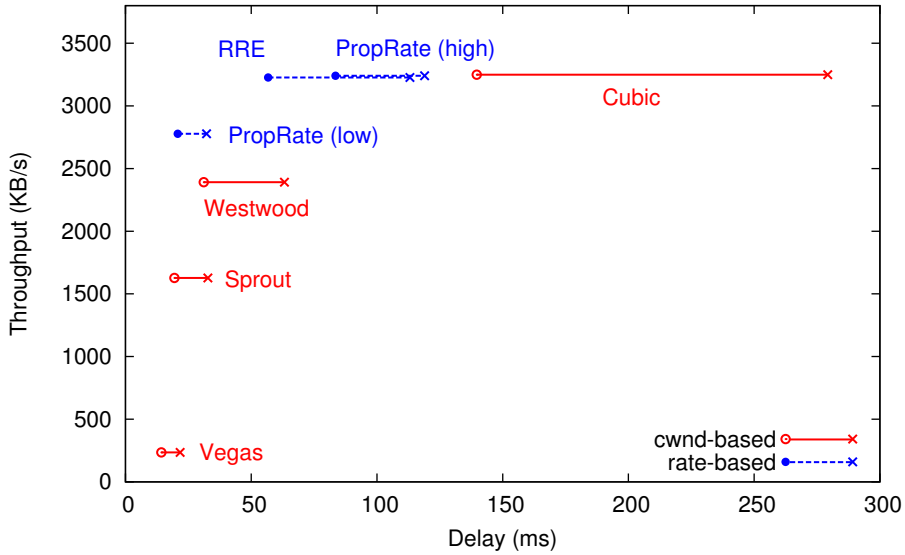


Figure 6.10: Plot of throughput vs delay on ISP A LTE network.

results are very similar to that obtained by emulation in Figure 6.1(a). This also validates our emulation-based evaluations in Sections 6.2.2 to 6.2.6.

6.3 Summary

In this chapter, we introduced PropRate, another rate-based congestion control algorithm for our rate-based TCP framework. We showed that a simple algorithm like PropRate can be adjusted to optimize the algorithm towards throughput or delay. In addition, the performance of PropRate can be as good, if not better than the current forecasting techniques Sprout and PROTEUS, which optimize for delay. By varying the parameters and plotting the throughput/delay tradeoff, we also showed that our rate-based algorithms can achieve a good frontier in our network traces. This shows that it is possible to obtain a good performance tradeoff by simply varying the parameters of our framework.

Chapter 7

Conclusion and Future Work

In this thesis, we investigated mobile cellular data networks and found that i) the downlink performance of two-way concurrent TCP flows is severely affected by ACK packets being delayed in the uplink; ii) TCP flows typically have high latencies as ISPs typically provision deep buffers, and the low packet loss rate allows the *cwnd* to grow large; and iii) stochastic forecasting of the link throughput can reduce the overall latency but overly sacrifices on throughput. To address these issues, we proposed a new rate-based approach to TCP congestion control and implemented a working framework in the Linux kernel.

We showed that by using this framework, we can achieve high throughput/utilization in the presence of a saturated or congested uplink or achieve low latencies by controlling some parameters.

A rate-based approach has several challenges which we have solved in our congestion control framework. The first challenge of obtaining the rate estimate of the link was solved by using the TCP timestamps of ACK packets

to estimate to arrival time of the corresponding data packets. The second challenge is to handle bandwidth variations in the network. This is solved by using a feedback mechanism to oscillate the sending rate about a fixed value, i.e., we intentionally send faster than the estimated bandwidth to probe for potential bandwidth increase, and throttle back when congestion is encountered. This leads to the third challenge of how to determine the onset of congestion. Traditionally, congestion is triggered by packet losses. However, cellular data networks have very low packet losses due to the hybrid-ARQ scheme at the link layer. The very deep buffers typically provisioned at the ISPs results in bufferbloat and increases the end-to-end delay. Thus, in our framework, we directly estimate the buffer queue by observing the relative difference in the TCP timestamps to obtain and estimate of the buffer delay. Together, this self-oscillating feedback mechanism helps achieve the stability that is inherent in the ACK-clocked *cwnd*-based mechanism.

Our rate-based congestion control framework have parameters that can be controlled by different algorithms, namely i) estimating the receiving rate; ii) setting the sending rate; and iii) the congestion threshold. We demonstrated using two control algorithms, RRE and PropRate, that we can optimize the algorithms towards maximizing throughput or minimizing delay. As a further proof-of-concept, we also implemented two current state-of-the-art forecasting techniques, Sprout and PROTEUS, as congestion control algorithms in our framework. We showed that while forecasting might help improve latency of TCP flows in cellular networks, a simple algorithm like PropRate can perform better given the correct parameters.

7.1 Future Work

The development of our rate-based congestion control algorithm opens up new possibilities for further research. We highlight a few of those possibilities.

7.1.1 Navigating the performance frontier

In our current implementation of PropRate, we have chosen two sets of parameters: one which is optimized for good throughput and another for low delay. We have also shown by arbitrarily adjusting the parameters that it is possible to obtain a range of performance trade-off between throughput and delay. However, most of the results are not on the optimal frontier. Therefore it remains to derive an algorithm or formula to obtain a set of parameters whose loci will lie along the optimal frontier.

One such approach would be to first filter the set of parameters whose result lies on the frontier from running PropRate over constant network traces of different throughput, and thereafter, try to fit an equation of each parameter with respect to the delay or throughput. From this, we can obtain an equation on which we can set the parameters based on the desired delay or throughput. Our preliminary investigations on this method have showed some promising result, thus we are continuing research in this approach.

7.1.2 Model of rate-based congestion control

An advantage of a framework is that an algorithm and structure can be well-defined. The interactions between the parameters are based on a general algorithms. This makes it possible to derive a set of formal equations or model

to perform analysis on the algorithm. In the traditional TCP congestion control mechanism, the size of the congestion window is set by the algorithm in response to lost packets, which are assumed to be lost due to buffer overflow. This not only allows TCP throughput to be modelled [53, 11], but also analysis of general AIMD algorithms in regards to buffer sizing [4, 56, 61].

Such analysis and modelling is also possible under our new rate-based TCP congestion control framework. Further research can be done to analyse and compare the performance of rate-based TCP congestion control algorithms.

7.1.3 Explore new rate-base algorithms

Our framework specifies a set of parameters upon which different algorithms can adjust, thereby producing different effects. Not only are the parameters tunable, but the computation method itself is open, e.g., estimating the rate from raw packet timestamps. There are several methods to estimate the rate and we have only investigated using a naive sliding-window technique. This leaves much room for further research in how other techniques of estimation can be done.

Another aspect for more development is the technique in determining the sending rates. While we have investigated a few techniques like using a fixed proportion and stochastic forecasting techniques and regression trees, there are many other techniques that can be implemented such as using proportional-integral-derivative (PID) control equations to control the feedback loop.

7.1.4 Use in other networks

Our work was developed specifically to overcome the bufferbloat and delayed-ACK issues in cellular data networks. However, a rate-base congestion control could also potentially be advantageous over other networks such as Wi-Fi. In addition, as LTE speeds continue to increase, techniques and algorithms have to evolve with the new challenges. Continued research is needed to investigate rate-based congestion control under such conditions.

Bibliography

[1] IDC worldwide mobile phone tracker, November 2013.

[2] Juan J. Alcaraz and Fernando Cerdán. Combining ACK Rate Control and AQM to Enhance TCP Performance over 3G Links. In *Proceedings of PM²HW²N '06*, October 2006.

[3] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing Router Buffers. In *Proceedings of SIGCOMM '04*, August 2004.

[4] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 281–292, New York, NY, USA, 2004. ACM.

[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] Hari Balakrishnan, Randy H. Katz, and Venkata N. Padmanbhan. The effects of asymmetry on TCP performance. *Mob. Netw. Appl.*, 4:219–241, October 1999.

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

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

[9] Jesper D. Brouer and Jørgen S. Hansen. Experiences with reducing TCP performance problems on ADSL. *DIKU - Technical Report 04/07*, May 2004.

[10] Carlo Caini and Rosario Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *INTERNATIONAL JOURNAL OF SATELLITE COMMUNICATIONS AND NETWORKING*, 22, 2004.

[11] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling tcp latency. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1742–1751. IEEE, 2000.

[12] Robert L. Carter and Mark E. Crovella. Measuring bottleneck link speed in packet-switched networks. Technical report, Performance Evaluation, 1996.

[13] Rajiv Chakravorty, Joel Cartwright, and Ian Pratt. Practical experience with TCP over GPRS. In *Proceedings of IEEE GLOBECOM '02*, 2002.

[14] Rajiv Chakravorty and Ian Pratt. WWW performance over GPRS. In *MWCN*, pages 527–531. IEEE, 2002.

[15] Mun Choon Chan and Ram Ramjee. Improving TCP/IP performance over third-generation wireless networks. *IEEE Transactions on Mobile Computing*, 7(4):430–443, 2008.

[16] Mun Choon Chan and Ramachandran Ramjee. TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation. In *Proceedings of MobiCom '02*, September 2002.

[17] Mun Choon Chan and Ramachandran Ramjee. Improving TCP/IP Performance over Third Generation Wireless Networks. In *Proceedings of INFOCOM '04*, March 2004.

[18] Chih-He Chiang, Wanjiun Liao, and Tehuang Liu. Adaptive downlink/uplink bandwidth allocation in IEEE 802.16 (WiMAX) wireless networks: A cross-layer approach. In *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pages 4775–4779, Nov 2007.

[19] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for TCP. In *Proceedings of IMC '11*, November 2011.

[20] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing TCP's initial congestion window. *SIGCOMM Computer Communications Review*, 40, June 2010.

[21] Addisu Eshete, Andrés Arcia, David Ros, and Yuming Jiang. Impact of wimax network asymmetry on TCP. In *Wireless Communications and*

Networking Conference, 2009. WCNC 2009. IEEE, pages 1–6. IEEE, 2009.

[22] Sally Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3481 (Best Current Practice), December 2003.

[23] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications. In *Proceedings of SIGCOMM '00*, pages 43–56, August 2000.

[24] Gartner. Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth. `http://www.gartner.com/it/page.jsp?id=1924314`.

[25] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. *Queue*, 9(11):40–54, November 2011.

[26] Luigi A. Grieco and Saverio Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *SIGCOMM Comput. Commun. Rev.*, 34(2):25–38, April 2004.

[27] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Operating Systems Review*, July 2008.

[28] Martin Heusse, Sears A. Merritt, Timothy X. Brown, and Andrzej Duda. Two-way TCP Connections: Old Problem, New Insight. *ACM Computer Communications Review*, 41(2):5–15, April 2011.

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

[30] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z. Morley Mao, Ming Zhang, and Paramvir Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of MobiSys '10*, June 2010.

[31] Hiroshi Inamura, Gabriel Montenegro, Reiner Ludwig, Andrei Gurtov, and Farid Khafizov. TCP over Second (2.5G) and Third (3G) Generation Wireless Networks. RFC 3481 (Best Current Practice), February 2003.

[32] V. Jacobson. Congestion avoidance and control. *SIGCOMM Computer Communications Review*, August 1988.

[33] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer system. DEC Research Report TR-301, September 1984.

[34] Haiqing Jiang, Zeyu Liu, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Understanding bufferbloat in cellular networks. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, CellNet '12, pages 1–6, New York, NY, USA, 2012. ACM.

[35] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Tackling bufferbloat in 3G/4G networks. In *Proceedings of IMC '12*, November 2012.

[36] Lampros Kalampoukas, Anujan Varma, and K. K. Ramakrishnan. Two-way tcp traffic over rate controlled channels: Effects and analysis. *IEEE/ACM Trans. Netw.*, 6(6):729–743, December 1998.

[37] Aditya Karnik and Anurag Kumar. Performance of TCP congestion control with explicit rate feedback: Rate adaptive TCP (RATCP). In *Proceedings of Globecom '00*, December 2000.

[38] Jun Ke and Carey Williamson. Towards a Rate-Based TCP Protocol for the Web. In *Proceedings of MASCOT '00*, September 2000.

[39] Jari Korhonen and Ye Wang. Effect of packet size on loss rate and delay in wireless links. In *Proceedings of WCNC '05*, 2005.

[40] Aleksandar Kuzmanovic and Edward W. Knightly. TCP-LP: Low-priority service via end-point congestion control. *IEEE/ACM Trans. Netw.*, 14(4):739–752, August 2006.

[41] Douglas Leith and Robert Shorten. H-TCP: TCP for high-speed and long-distance networks. 2004.

[42] Wai Kay Leong, Yin Xu, Ben Leong, and Zixiao Wang. Mitigating egregious ACK delays in cellular data networks by eliminating TCP ACK clocking. In *Proceedings of ICNP '13*, October 2013.

[43] Fatma Louati, Chadi Barakat, and Walid Dabbous. Handling two-way tcp traffic in asymmetric networks. In *High Speed Networks and Multimedia Communications*, pages 233–243. Springer, 2004.

[44] H. Lundin, S. Holmer, and H. Alvestrand. A google congestion control algorithm for real-time communication on the world wide web. IETF Working Draft, Oct. 2012.

[45] David A. Maltz and Pravin Bhagwat. TCP splicing for application layer proxy performance. *JHSN*, 1999.

[46] Jim Martin, Arne Nilsson, and Injong Rhee. Delay-based congestion avoidance for TCP. *IEEE/ACM Transactions on Networking*, 11(3):356–369, June 2003.

[47] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of MobiCom '01*, July 2001.

[48] Matt Mathis and Jamshid Mahdavi. TCP rate-halving with bounding parameters. December 1997.

[49] Mary Meeker. 2013 Internet Trends. Kleiner Perkins Caufield & Byers, 2013.

[50] Ivan Tam Ming-Chit, Du Jinsong, and Weiguo Wang. Improving TCP Performance Over Asymmetric Networks. *ACM Computer Communications Review*, 30(3):45–54, July 2000.

[51] Jeffrey C. Mogul. Observing TCP dynamics in real networks. In *Conference Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '92, pages 305–317, New York, NY, USA, 1992. ACM.

[52] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Queue*, 10(5):20–34, May 2012.

[53] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 303–314, New York, NY, USA, 1998. ACM.

[54] Jitendra Padhye, Jim Kurose, Don Towsley, and Rajeev Koodli. A model based TCP-friendly rate control protocol. In *Proceedings of NOSSDAV '99*, June 1999.

[55] Vern Paxson. End-to-end internet packet dynamics. In *Proceedings of SIGCOMM '97*, SIGCOMM '97, pages 139–152. ACM, 1997.

[56] G. Raina and D. Wischik. Buffer sizes for large multiplexers: TCP queueing theory and instability analysis. In *Next Generation Internet Networks, 2005*, pages 173–180, April 2005.

[57] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of SOSP '13*, Nov. 2013.

[58] Nihal K. G. Samaraweera. Return link optimization for internet service provision using DVB-S networks. *SIGCOMM Comput. Commun. Rev.*, 29(3):4–13, July 1999.

[59] Basic Transmission Scheme. LTE: the evolution of mobile broadband. *IEEE Communications Magazine*, page 45, 2009.

[60] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 3550, 2003.

[61] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic tcp buffer tuning. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 315–323, New York, NY, USA, 1998. ACM.

[62] Stefania Sesia, Issam Toufik, and Matthew Baker. *LTE: the UMTS long term evolution*. Wiley Online Library, 2009.

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

[64] Dibyendu Shekhar, Hua Qin, Shivkumar Kalyanaraman, and Kalyan Kidambi. Performance optimization of TCP/IP over asymmetric wired and wireless links. *Invited paper at European Wireless 2002*, February 2002.

[65] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *Proceedings of IEEE INFOCOM '06*, April 2006.

[66] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, December 2002.

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

[68] Vikram Visweswaraiah and John Heidemann. Rate based pacing for TCP. `http://www.isi.edu/lsam/publications/rate_based_pacing/`, 1997.

[69] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *Proceedings of INFOCOM '10*, April 2010.

[70] Keith Winstein. Personal communication. E-mail, November 2013.

[71] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of NSDI '13*, Apr. 2013.

[72] Jing Wu, Dongho Kim, and Jeonghoon Mo. TCP performance over the WiBrO compatible 802.16e systems. In *Advanced Communication Technology, The 9th International Conference on*, volume 3, pages 1752–1755. IEEE, 2007.

[73] Qiang Xu, Sanjeev Mehrotra, Zhuoqing Mao, and Jin Li. PROTEUS: Network performance forecast for real-time, interactive mobile applications. In *Proceeding of MobiSys '13*, Jun. 2013.

[74] Yin Xu. *Understanding and Mitigating Congestion in Modern Networks*. PhD thesis, National University of Singapore, 2014.

[75] Yin Xu, Wai Kay Leong, Ben Leong, and Ali Razeen. Dynamic regulation of mobile 3G/HSPA uplink buffer with receiver-side flow control. In *Proceedings of ICNP '12*, October 2012.

[76] Yin Xu, Zixiao Wang, Wai Kay Leong, and Ben Leong. An end-to-end measurement study of modern cellular data networks. In *Proceedings of PAM '14*, Mar. 2014.

[77] Xiangying Yang, Muthaiah Venkatachalam, and Shantidev Mohanty. Exploiting the mac layer flexibility of wimax to systematically enhance tcp performance. In *Mobile WiMAX Symposium, 2007. IEEE*, pages 60–65. IEEE, 2007.

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