# Tackling Bufferbloat in 3G/4G Networks

Haiqing Jiang[1], Yaogong Wang[1], Kyunghan Lee[2], and Injong Rhee[1]
North Carolina State University, USA [1] Ulsan National Institute of Science and Technology, Korea [2]
{hjiang5, ywang15, rhee}@ncsu.edu khlee@unist.ac.kr

## ABSTRACT

The problem of overbuffering in the current Internet (termed as bufferbloat) has drawn the attention of the research community in recent years. Cellular networks keep large buffers at base stations to smooth out the bursty data traffic over the time-varying channels and are hence apt to bufferbloat. However, despite their growing importance due to the boom of smart phones, we still lack a comprehensive study of bufferbloat in cellular networks and its impact on TCP performance. In this paper, we conducted extensive measurement of the 3G/4G networks of the four major U.S. carriers and the largest carrier in Korea. We revealed the severity of bufferbloat in current cellular networks and discovered some ad-hoc tricks adopted by smart phone vendors to mitigate its impact. Our experiments show that, due to their static nature, these ad-hoc solutions may result in performance degradation under various scenarios. Hence, a dynamic scheme which requires only receiver-side modification and can be easily deployed via over-the-air (OTA) updates is proposed. According to our extensive real-world tests, our proposal may reduce the latency experienced by TCP flows by 25% $\sim$ 49% and increase TCP throughput by up to 51% in certain scenarios.

## Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

## General Terms

Design, Measurement, Performance

## Keywords

Bufferbloat, Cellular Networks, TCP, Receive Window

## 1. INTRODUCTION

Bufferbloat, as termed by Gettys [10], is a phenomenon where oversized buffers in the network result in extremely long delay and other performance degradation. It has been observed in different parts of the Internet, ranging from ADSL to cable modem users [5, 17, 26]. Cellular networks are another place where buffers are heavily provisioned to accommodate the dynamic cellular link (Figure 1). However, other than some ad-hoc observations [24], bufferbloat in cellular networks has not been studied systematically.

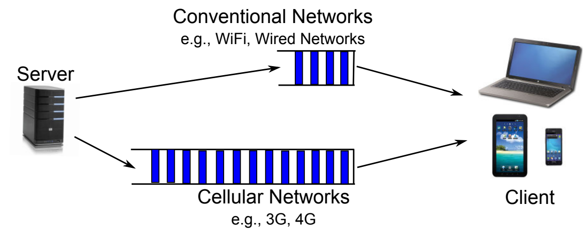



**Figure 1: Bufferbloat has been widely observed in the current Internet but is especially severe in cellular networks, resulting in up to several seconds of round trip delay.**

In this paper, we carried out extensive measurements over the 3G/4G networks of all four major U.S. carriers (AT&T, Sprint, T-Mobile, Verizon) as well as the largest cellular carrier in Korea (SK Telecom). Our experiments span more than two months and consume over 200GB of 3G/4G data. According to our measurements, TCP has a number of performance issues in bufferbloated cellular networks, including extremely long delays and sub-optimal throughput. The reasons behind such performance degradation are two-fold. First, most of the widely deployed TCP implementations use loss-based congestion control where the sender will not slow down its sending rate until it sees packet loss. Second, most cellular networks are overbuffered to accommodate traffic burstiness and channel variability [20]. The exceptionally large buffer along with link layer retransmission conceals packet losses from TCP senders. The combination of these two facts leads to the following phenomenon: the TCP sender continues to increase its sending rate even if it has already exceeded the bottleneck link capacity since all of the overshot packets are absorbed by the buffers. This results in up to several seconds of round trip delay. This extremely long delay did not cause critical user experience problems today simply because 1) base stations typically has separate buffer space for each user [20] and 2) users do not multitask on smart phones very often at this point. This means only a single TCP flow is using the buffer space. If it is a short-lived flow like Web browsing, queues will not build up since the traffic is small. If it is a long-lived


*IMC'12,* November 14–16, 2012, Boston, Massachusetts, USA.

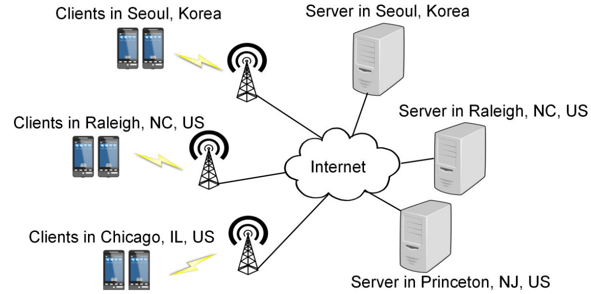
flow like downloading a file, long queues will build up but users hardly notice them since it is the throughput that matters rather than the delay. However, as the smart phones become more and more powerful (e.g., several recently released smart phones are equipped with quad-core processors and 2GB of memory), users are expected to perform multi-tasking more often. If a user is playing an online game and at the same time downloading a song in the background, severe problem will appear since the time-sensitive gaming traffic will experience huge queuing delays caused by the background download. Hence, we believe that bufferbloat in 3G/4G networks is an important problem that must be addressed in the near future.

This problem is not completely unnoticed by today's smart phone vendors. Our investigation into the open source Android platform reveals that a small untold trick has been applied to mitigate the issue: the maximum TCP receive buffer size parameter (*tcp_rmem_max*) has been set to a relatively small value although the physical buffer size is much larger. Since the advertised receive window (*rwnd*) cannot exceed the receive buffer size and the sender cannot send more than what is allowed by the advertised receive window, the limit on *tcp_rmem_max* effectively prevents TCP congestion window (*cwnd*) from excessive growth and controls the RTT (round trip time) of the flow within a reasonable range. However, since the limit is statically configured, it is sub-optimal in many scenarios, especially considering the dynamic nature of the wireless mobile environment. In high speed long distance networks (e.g., downloading from an oversea server over 4G LTE (Long Term Evolution) network), the static value could be too small to saturate the link and results in throughput degradation. On the other hand, in small bandwidth-delay product (BDP) networks, the static value may be too large and the flow may experience excessively long RTT.

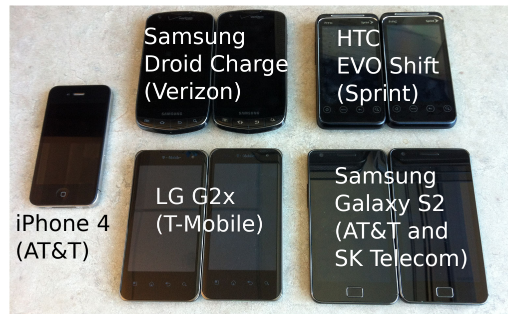
There are many possible ways to tackle this problem, ranging from modifying TCP congestion control algorithm at the sender to adopting Active Queue Management (AQM) at the base station. However, all of them incur considerable deployment cost. In this paper, we propose dynamic receive window adjustment (DRWA), a light-weight, receiver-based solution that is cheap to deploy. Since DRWA requires modifications only on the receiver side and is fully compatible with existing TCP protocol, carriers or device manufacturers can simply issue an over-the-air (OTA) update to smart phones so that they can immediately enjoy better performance even when interacting with existing servers.

DRWA is similar in spirit to delay-based congestion control algorithms but runs on the receiver side. It modifies the existing receive window adjustment algorithm of TCP to indirectly control the sending rate. Roughly speaking, DRWA increases the advertised window when the current RTT is close to the minimum RTT we have observed so far and decreases it when RTT becomes larger due to queuing delay. With proper parameter tuning, DRWA could keep the queue size at the bottleneck link small yet non-empty so that throughput and delay experienced by the TCP flow are both optimized. Our extensive experiments show that DRWA reduces the RTT by 25% $\sim$ 49% while achieving similar throughput in ordinary scenarios. In large BDP networks, DRWA can achieve up to 51% throughput improvement over existing implementations.

In summary, the contributions of this paper include:




(a) Experiment testbed




(b) Experiment phones

**Figure 2: Our measurement framework spans across the globe with servers and clients deployed in various places in U.S. and Korea using the cellular networks of five different carriers.**

- We conducted extensive measurements in a range of cellular networks (EVDO, HSPA+, LTE) across various carriers and characterized the bufferbloat problem in these networks.
- We anatomized the TCP implementation in state-of-the-art smart phones and revealed the limitation of their ad-hoc solution to the bufferbloat problem.
- We proposed a simple and immediately deployable solution that is experimentally proven to be safe and effective.

The rest of the paper is organized as follow. Section 2 introduces our measurement setup and highlights the severity of bufferbloat in today's 3G/4G networks. Section 3 then investigates the impact of bufferbloat on TCP performance and points out the pitfalls of high speed TCP variants in cellular networks. The abnormal behavior of TCP in smart phones is revealed in Section 4 and its root cause is located. We then propose our solution DRWA in Section 5 and evaluate its performance in Section 6. Finally, alternative solutions and related work are discussed in Section 7 and we conclude our work in Section 8.

## 2. OBSERVATION OF BUFFERBLOAT IN CELLULAR NETWORKS

Bufferbloat is a phenomenon prevalent in the current Internet where excessive buffers within the network lead to exceptionally large end-to-end latency and jitter as well as throughput degradation. With the recent boom of smart phones and tablets, cellular networks become a more and more important part of the Internet. However, despite the

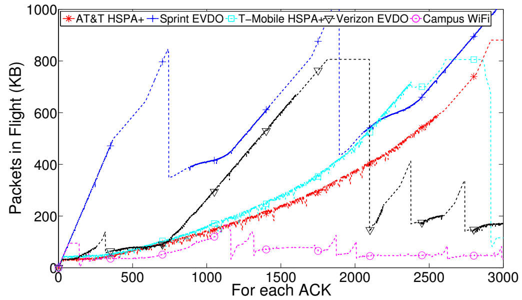



**Figure 3: We observed exceptionally fat pipes across the cellular networks of five different carriers. We tried three different client/server locations under both good and weak signal case. This shows the prevalence of bufferbloat in cellular networks. The figure above is a representative example.**

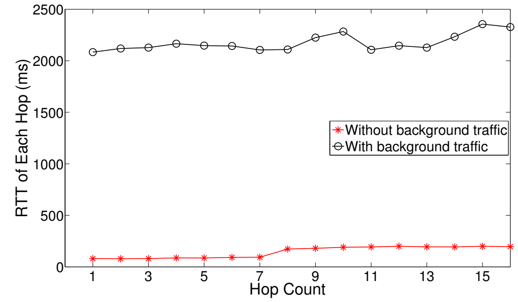



**Figure 4: We verified that the queue is built up at the very first IP hop (from the mobile client).**

abundant measurement studies of the cellular Internet [4, 18, 20, 23, 14], the specific problem of bufferbloat in cellular networks has not been studied systematically. To obtain a comprehensive understanding of this problem and its impact on TCP performance, we have set up the following measurement framework which is used throughout the paper.

## 2.1 Measurement Setup

Figure 2(a) gives an overview of our testbed. We have servers and clients deployed in various places in U.S. and Korea so that a number of scenarios with different BDPs can be tested. All of our servers run Ubuntu 10.04 (with 2.6.35.13 kernel) and use its default TCP congestion control algorithm CUBIC [11] unless otherwise noted. We use several different phone models on the client side, each working with the 3G/4G network of s specific carrier (Figure 2(b)). The signal strength during our tests ranges from -75dBm to -105dBm so that it covers both good signal condition and weak signal condition. We develop some simple applications on the client side to download data from the server with different traffic patterns (short-lived, long-lived, etc.). The most commonly used traffic pattern is long-lived TCP flow where the client downloads a very large file from the server for 3 minutes (the file is large enough so that the download never finishes within 3 minutes). Most experiments have been repeated numerous times for a whole day with a one-minute interval between each run. That results in more than 300 samples for each experiment based on which we calculate the average and the confidence interval.

For the rest of the paper, we only consider the performance of the downlink (from base station to mobile station) since it is the most common case. We leave the measurement of uplink performance as our future work. Since we are going to present a large number of measurement results under various conditions in this paper, we provide a table that summaries the setup of each experiment for the reader's convenience. Please refer to Table 1 in Appendix A.

## 2.2 Bufferbloat in Cellular Networks

The potential problem of overbuffering in cellular networks was pointed out by Ludwig et al. [21] as early as 1999 when researchers were focusing on GPRS networks. However, overbuffering still prevails in today's 3G/4G networks. To estimate the buffer space in current cellular networks, we set up the following experiment: we launch a long-lived TCP flow from our server to a Linux laptop (Ubuntu 10.04 with 2.6.35.13 kernel) over the 3G networks of four major U.S. carriers. By default, Ubuntu sets both the maximum TCP receive buffer size and the maximum TCP send buffer size to a large value (greater than 3MB). Hence, the flow will never be limited by the buffer size of the end points. Due to the closed nature of cellular networks, we are unable to know the exact queue size within the network. Instead, we measure the size of packets in flight on the sender side to estimate the buffer space within the network. Figure 3 shows our measurement results. We observed exceptionally fat pipes in all four major U.S. cellular carriers. Take Sprint EVDO network for instance. The peak downlink rate for EVDO is 3.1 Mbps and the observed minimum RTT (which approximates the round-trip propagation delay) is around 150ms. Therefore, the BDP of the network is around 58KB. But as the figure shows, Sprint is able to bear more than 800KB of packets in flight!

As a comparison, we ran a similar experiment of a long-lived TCP flow between a client in Raleigh, U.S. and a server in Seoul, Korea over the campus WiFi network. Due to the long distance of the link and the ample bandwidth of WiFi, the corresponding pipe size is expected to be large. However, according to Figure 3, the size of in-flight packets even in such a large BDP network is still much smaller than the ones we observed in cellular networks.

We extend the measurement to other scenarios in the field to verify that the observation is universal in current cellular networks. For example, we have clients and servers in various locations over various cellular networks in various signal conditions (Table 1). All the scenarios prove the existence of extremely fat pipes similar to Figure 3.

To further confirm that the bufferbloat is within the cellular segment rather than the backbone Internet, we designed the following experiment to locate where the long queue is built up. We use *Traceroute* on the client side to measure the RTT of each hop along the path to the server and compare the results with or without a background long-lived TCP flow. If the queue is built up at hop $x$, the queuing delay should increase significantly at that hop when background traffic is in place. Hence, we should see that the RTTs before hop $x$ do not differ much no matter the background traffic is present or not. But the RTTs after hop $x$ should have a notable gap between the case with background traffic and the case without. The results shown in Figure 4 demonstrate that the queue is built up at the very first hop. Note that there could be a number of components between the mo-

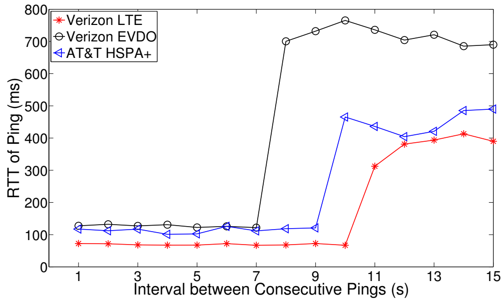


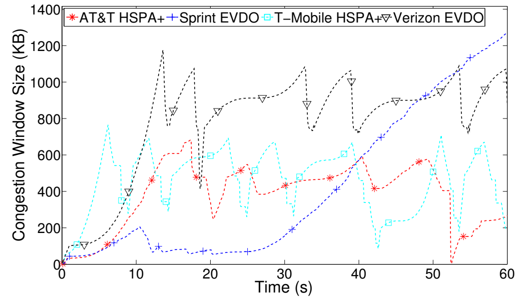

**Figure 5: RRC state transition only affects short-lived TCP flows with considerable idle periods in between (> 7s) but does not affect long-lived flows.**

bile client and the first IP hop (e.g., RNC, SGSN, GGSN, etc.). It may not be only the wireless link between the mobile station and the base station. Without administrative access, we are unable to diagnose the details within the carrier's network but according to [20] large per-user buffer are deployed at the base station to absorb channel fluctuation.

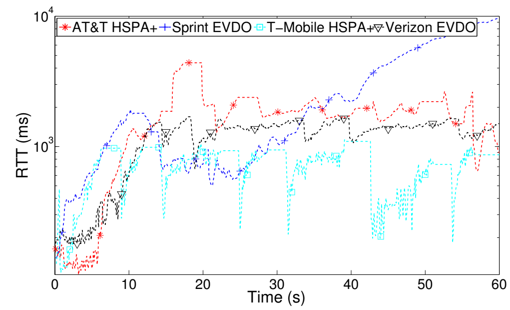
Another concern is that the extremely long delays we observed in cellular networks are due to Radio Resource Control (RRC) state transitions [1] rather than bufferbloat. We set up the following experiment to demonstrate that these two problems are orthogonal. We repeatedly *ping* our server from the mobile station with different intervals between consecutive *ping*s. The experiment has been carried out for a whole day and the average RTT of the *ping* is calculated. According to the RRC state transition diagram, if the interval between consecutive *ping*s is long enough, we should observe a substantially higher RTT due to the state promotion delay. By varying this interval in each run, we could obtain the threshold that would trigger RRC state transition. As shown in Figure 5, when the interval between consecutive *ping*s is beyond 7 seconds (specific threshold depends on the network type and the carrier), there is a sudden increase in RTT which demonstrates the state promotion delay. However, when the interval is below 7 seconds, RRC state transition does not seem to affect the performance. Hence, we conclude that RRC state transition may only affect short-lived TCP flows with considerable idle periods in between (e.g., Web browsing), but does not affect long-lived TCP flows we were testing since their packet intervals are typically at millisecond scale. When the interval between packets is short, the cellular device should remain in CELL_DCH state and state promotion delays would not contribute to the extremely long delays we observed.

# 3. TCP PERFORMANCE OVER BUFFERBLOATED CELLULAR NETWORKS

Given the exceptionally large buffer size in cellular networks as observed in Section 2, in this section we investigate its impact on TCP's behavior and performance. We carried out similar experiments to Figure 3 but observed the congestion window size and RTT of the long-lived TCP flow instead of packets in flight. As Figure 6 shows, TCP congestion window keeps probing even if its size is far beyond the BDP of the underlying network. With so much overshooting, the extremely long RTT (up to 10 seconds!) as shown in Figure 6(b) is not surprising.




(a) Congestion Window Size



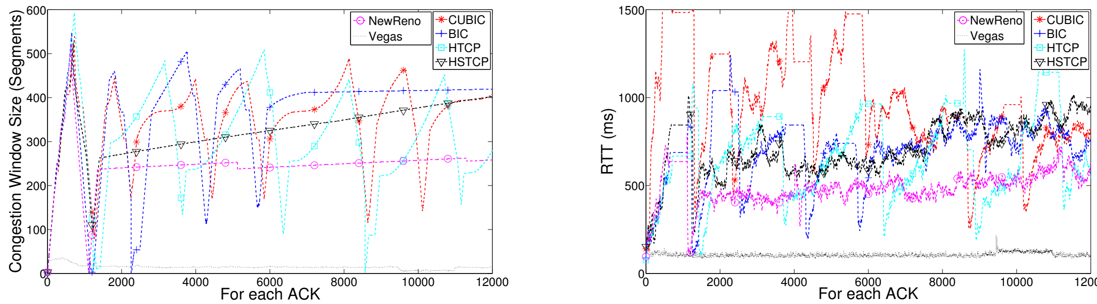

(b) Round Trip Time

**Figure 6: TCP congestion window grows way beyond the BDP of the underlying network due to bufferbloat. Such excessive overshooting leads to extremely long RTT.**

In the previous experiment, the server used CUBIC as its TCP congestion control algorithm. However, we are also interested in the behaviors of other TCP congestion control algorithms under bufferbloat. According to [32], a large portion of the Web servers in the current Internet use high speed TCP variants such as BIC [30], CUBIC [11], CTCP [27], HSTCP [7] and H-TCP [19]. How these high speed TCP variants would perform in bufferbloated cellular networks as compared to less aggressive TCP variants like TCP NewReno [8] and TCP Vegas [2] is of great interest.

Figure 7 shows the *cwnd* and RTT of TCP NewReno, Vegas, CUBIC, BIC, HTCP and HSTCP under AT&T HSPA+ network. We left CTCP out of the picture simply because we are unable to know its internal behavior due to the closed nature of Windows. As the figure shows, all the loss-based high speed TCP variants (CUBIC, BIC, HTCP, HSTCP) overshoot more often than NewReno. These high speed variants were originally designed for efficient probing of the available bandwidth in large BDP networks. But in bufferbloated cellular networks, they only make the problem worse by constant overshooting. Hence, the bufferbloat problem adds a new dimension in the design of an efficient TCP congestion control algorithm.

In contrast, TCP Vegas is resistive to bufferbloat as it uses a delay-based congestion control algorithm that backs off as soon as RTT starts to increase. This behavior prevents *cwnd* from excessive growth and keeps the RTT at a low level. However, delay-based TCP congestion control has its own problems and is far from a perfect solution to bufferbloat. We will further discuss this aspect in Section 7.




(a) Congestion Window Size

(b) Round Trip Time

**Figure 7: All the loss-based high speed TCP variants (CUBIC, BIC, HTCP, HSTCP) suffer from the bufferbloat problem more severely than NewReno. But TCP Vegas, a delay-based TCP variant, is resistive to bufferbloat.**

# 4. CURRENT TRICK BY SMART PHONE VENDORS AND ITS LIMITATION

The previous experiments used a Linux laptop with mobile broadband USB modem as the client. We have not looked at other platforms yet, especially the exponentially growing smart phones. In the following experiment, we explore the behavior of different TCP implementations in various desktop (Windows 7, Mac OS 10.7, Ubuntu 10.04) and mobile operating systems (iOS 5, Android 2.3, Windows Phone 7) over cellular networks.

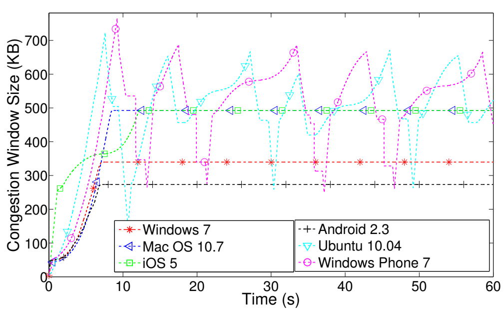



**Figure 8: The behavior of TCP in various platforms over AT&T HSPA+ network exhibits two patterns: "flat TCP" and "fat TCP".**

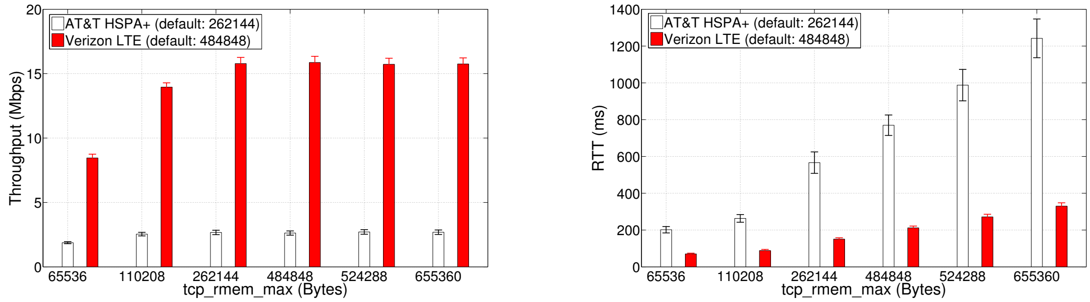
Figure 8 depicts the evolution of TCP congestion window when clients of various platforms launch a long-lived TCP flow over AT&T HSPA+ network. To our surprise, two types of *cwnd* patterns are observed: "flat TCP" and "fat TCP". Flat TCP, such as observed in Android phones, is the phenomenon where the TCP congestion window grows to a constant value and stays there until the session ends. On the other hand, fat TCP, such as observed in Windows Phone 7, is the phenomenon that packet loss events do not occur until the congestion window grows to a large value far beyond the BDP. Fat TCP can easily be explained by the bufferbloat in cellular networks and the loss-based congestion control algorithm. But the abnormal flat TCP behavior caught our attention and revealed an untold story of TCP over cellular networks.

## 4.1 Understanding the Abnormal Flat TCP

How could the TCP congestion window stay at a constant value? The static *cwnd* first indicates that no packet loss is observed by the TCP sender (otherwise the congestion window should have decreased multiplicatively at any loss event). This is due to the large buffers in cellular networks and its link layer retransmission mechanism as discussed earlier. Measurement results from [14] also confirm that cellular networks typically experience close-to-zero packet loss rate.

If packet losses are perfectly concealed, the congestion window may not drop but it will persistently grow as fat TCP does. However, it unexpectedly stops at a certain value and this value is different for each cellular network or client platform. Our inspection into the TCP implementation in Android phones (since it is open-source) reveals that the value is determined by the *tcp_rmem_max* parameter that specifies the maximum receive window advertised by the Android phone. This gives the answer to flat TCP behavior: the receive window advertised by the receiver crops the congestion windows in the sender. By inspecting various Android phone models, we found that *tcp_rmem_max* has diverse values for different types of networks (refer to Table 2 in Appendix B for some sample settings). Generally speaking, larger values are assigned to faster communication standards (e.g., LTE). But all the values are statically configured.

To understand the impact of such static settings, we compared the TCP performance under various *tcp_rmem_max* values in AT&T HSPA+ network and Verizon LTE network in Figure 9. Obviously, a larger *tcp_rmem_max* allows the congestion window to grow to a larger size and hence leads to higher throughput. But this throughput improvement will flatten out once the link is saturated. Further increase of *tcp_rmem_max* brings nothing but longer queuing delay and hence longer RTT. For instance, when downloading from a nearby server, the RTT is relatively small. In such small BDP networks, the default values for both HSPA+ and LTE are large enough to achieve full bandwidth utilization as shown in Figure 9(a). But they trigger excessive packets in flight and result in unnecessarily long RTT as shown in Figure 9(b). This demonstrates the limitation of static parameter setting: it mandates one specific trade-off point in the system which may be sub-optimal for other applications. Two realistic scenarios are discussed in the next subsection.




(a) Throughput (b) Round Trip Time

**Figure 9: Throughput and RTT performance of a long-lived TCP flow in a *small* BDP network under different *tcp_rmem_max* settings. For this specific environment, 110208 may work better than the default 262144 in AT&T HSPA+ network. Similarly, 262144 may work better than the default 484848 in Verizon LTE network. However, the optimal value depends on the BDP of the underlying network and is hard to be configured statically in advance.**

## 4.2 Impact on User Experience

**Web Browsing with Background Downloading:** The high-end smart phones released in 2012 typically have quad-core processors and more than 1GB of RAM. Due to their significantly improved capability, the phones are expected to multitask more often. For instance, people will enjoy Web browsing or online gaming while downloading files such as books, music, movies or applications in the background. In such cases, we found that the current TCP implementation incurs long delays for the interactive flow (Web browsing or online gaming) since the buffer is filled with packets belonging to the background long-lived TCP flow.

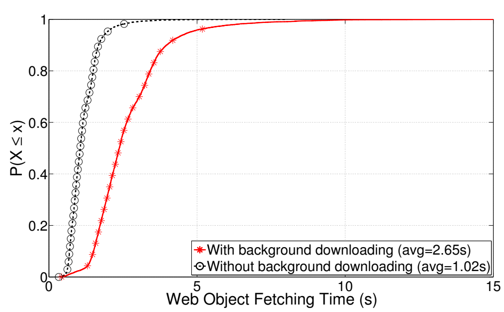



**Figure 10: The average Web object fetching time is 2.6 times longer when background downloading is present.**

Figure 10 demonstrates that the Web object fetching time is severely degraded when a background download is under way. In this experiment, we used a simplified method to emulate Web traffic. The mobile client generates Web requests according to a Poisson process. The size of the content brought by each request is randomly picked among 8KB, 16KB, 32KB and 64KB. Since these Web objects are small, their fetching time mainly depends on RTT rather than throughput. When a background long-lived flow causes long queues to be built up, the average Web object fetching time becomes 2.6 times longer.

**Throughput in Large BDP Networks:** The sites that smart phone users visit are diverse. Some contents are well maintained and CDNs (content delivery networks) are assisting them to get "closer" to their customers via replication. In such cases, the throughput performance can be satisfactory since the BDP of the network is small (Figure 9). However, there are still many sites with long latency due to their remote locations. In such cases, the static setting of *tcp_rmem_max* (which is tuned for moderate latency case) fails to fill the long fat pipe and results in sub-optimal throughput. Figure 11 shows that when a mobile client in Raleigh, U.S. downloads contents from a server in Seoul, Korea over AT&T HSPA+ network and Verizon LTE network, the default setting is far from optimal in terms of throughput performance. A larger *tcp_rmem_max* can achieve much higher throughput although setting it too large may cause packet loss and throughput degradation.

In summary, flat TCP has performance issues in both throughput and delay. In small BDP networks, the static setting of *tcp_rmem_max* may be too large and cause unnecessarily long end-to-end latency. On the other hand, it may be too small in large BDP networks and suffer from significant throughput degradation.

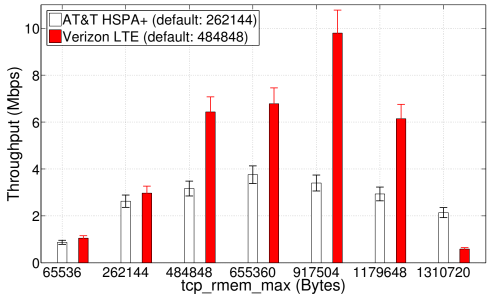
# 5. OUR SOLUTION

In light of the limitation of a static *tcp_rmem_max* setting, we propose a dynamic receive window adjustment algorithm to adapt to various scenarios automatically. But before discussing our proposal, let us first look at how TCP receive windows are controlled in the current implementations.

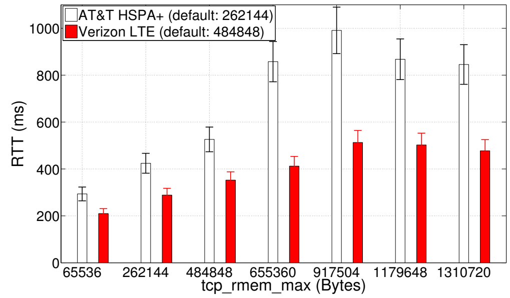
## 5.1 Receive Window Adjustment in Current TCP Implementations

As we know, the TCP receive window was originally designed to prevent a fast sender from overwhelming a slow receiver with limited buffer space. It reflects the available buffer size on the receiver side so that the sender will not send more packets than the receiver can accommodate. This is called TCP flow control, which is different from TCP congestion control whose goal is to prevent overload in the network rather than at the receiver. Flow control and congestion control together govern the transmission rate of a TCP sender and the sending window size is the minimum of the advertised receive window and the congestion window.

With the advancement in storage technology, memories are becoming increasingly cheaper. Currently, it is common




(a) Throughput




(b) Round Trip Time

**Figure 11: Throughput and RTT performance of a long-lived TCP flow in a *large* BDP network under different *tcp_rmem_max* settings. The default setting results in sub-optimal throughput performance since it fails to saturate the long fat pipe. 655360 for AT&T and 917504 for Verizon provide much higher throughput.**

to find computers (or even smart phones) equipped with gigabytes of RAM. Hence, buffer space on the receiver side is hardly the bottleneck in the current Internet. To improve TCP throughput, a receive buffer auto-tuning technique called Dynamic Right-Sizing (DRS [6]) was proposed. In DRS, instead of determining the receive window based by the available buffer space, the receive buffer size is dynamically adjusted in order to suit the connection's demand. Specifically, in each RTT, the receiver estimates the sender's congestion window and then advertises a receive window which is *twice the size* of the estimated congestion window. The fundamental goal of DRS is to allocate enough buffer (as long as we can afford it) so that the throughput of the TCP connection is never limited by the receive window size but only constrained by network congestion. Meanwhile, DRS tries to avoid allocating more buffers than necessary.

Linux adopted a receive buffer auto-tuning scheme similar to DRS since kernel 2.4.27. Since Android is based on Linux, it inherits the same receive window adjustment algorithm. Other major operating systems also implemented customized TCP buffer auto-tuning (Windows since Vista, Mac OS since 10.5, FreeBSD since 7.0). This implies a significant role change for the TCP receive window. Although the functionality of flow control is still preserved, most of the time the receive window as well as the receive buffer size is undergoing dynamic adjustments. However, this dynamic adjustment is unidirectional: DRS increases the receive window size only when it might potentially limit the congestion window growth but never decreases it.

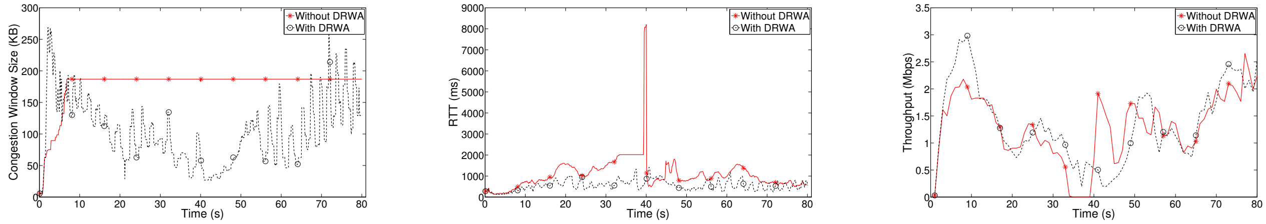
## 5.2 Dynamic Receive Window Adjustment

As discussed earlier, setting a static limit on the receive window size is inadequate to adapt to the diverse network scenarios in the mobile environment. We need to adjust the receive window dynamically. DRS is already doing this, but its adjustment is unidirectional. It does not solve the bufferbloat problem. In fact, it makes it worse by incessantly increasing the receive window size as the congestion window size grows. What we need is a bidirectional adjustment algorithm to rein TCP in the bufferbloated cellular networks. At the same time it needs to ensure full utilization of the available bandwidth. Hence, we build our DRWA proposal on top of DRS and Algorithm 1 gives the details.

DRWA uses the same technique as DRS to measure RTT on the receiver side when the TCP timestamp option [15] is not available (Line 8). However, if the timestamp option is available, DRWA uses it to obtain a more accurate estimation of the RTT (Line 10–12). TCP timestamp can provide multiple RTT samples within an RTT whereas the traditional DRS way provides only one sample per RTT. With the assistance of timestamps, DRWA is able to achieve robust RTT measurement on the receiver side. We also surveyed that both Windows Server and Linux support TCP timestamp option as long as the client requests it in the initial SYN segment. DRWA records the minimum RTT ever seen in this connection and uses it to approximate the

**Algorithm 1** DRWA

1: Initialization:
2: $tcp\_rmem\_max \leftarrow a\ large\ value$;
3: $RTT_{min} \leftarrow \infty$;
4: $cwnd_{est} \leftarrow data\_rcvd$ in the first $RTT_{est}$;
5: $rwnd \leftarrow 0$;
6:
7: RTT and minimum RTT estimation:
8: $RTT_{est} \leftarrow$ the time between when a byte is first acknowledged and the receipt of data that is at least one window beyond the sequence number that was acknowledged;
9:
10: **if** TCP timestamp option is available **then**
11: $\quad RTT_{est} \leftarrow$ averaging the RTT samples obtained from the timestamps within the last RTT;
12: **end if**
13:
14: **if** $RTT_{est} < RTT_{min}$ **then**
15: $\quad RTT_{min} \leftarrow RTT_{est}$;
16: **end if**
17:
18: DRWA:
19: **if** data is copied to user space **then**
20: $\quad$ **if** $elapsed\_time < RTT_{est}$ **then**
21: $\quad\quad$ return;
22: $\quad$ **end if**
23:
24: $\quad cwnd_{est} \leftarrow \alpha * cwnd_{est} + (1 - \alpha) * data\_rcvd$;
25: $\quad rwnd \leftarrow \lambda * \frac{RTT_{min}}{RTT_{est}} * cwnd_{est}$;
26: $\quad$ Advertise $rwnd$ as the receive window size;
27: **end if**




(a) Receive Window Size (b) Round Trip Time (c) Throughput

**Figure 12: When the smart phone is moved from a good signal area to a weak signal area and then moved back, DRWA nicely tracks the variation of the channel conditions and dynamically adjusts the receive window size, leading to a constantly low RTT but no throughput loss.**

round-trip propagation delay when no queue is built up in the intermediate routers (Line 14–16).

After knowing the RTT, DRWA counts the amount of data received within each RTT and smooths the estimated congestion window by a moving average with a low-pass filter (Line 24). $\alpha$ is set to 7/8 in our current implementation. This smoothed value is used to determine the receive window we advertise. In contrast to DRS who always sets $rwnd$ to $2 * cwnd_{est}$, DRWA sets it to $\lambda * \frac{RTT_{min}}{RTT_{est}} * cwnd_{est}$ where $\lambda$ is a tunable parameter larger than 1 (Line 25). When $RTT_{est}$ is close to $RTT_{min}$, implying the network is not congested, $rwnd$ will increase quickly to give the sender enough space to probe the available bandwidth. As $RTT_{est}$ increases, we gradually slow down the increment rate of $rwnd$ to stop TCP from overshooting. Thus, DRWA makes bidirectional adjustment of the advertised window and controls the $RTT_{est}$ to stay around $\lambda * RTT_{min}$. More detailed discussion on the impact of $\lambda$ will be given in Section 5.4.

This algorithm is simple yet effective. Its ideas stem from delay-based congestion control algorithms but work better than they do for two reasons. First, since DRWA only *guides* the TCP congestion window by advertising an adaptive receive window, the bandwidth probing responsibility still lies with the TCP congestion control algorithm at the sender. Therefore, typical throughput degradation seen in delay-based TCP will not appear. Second, due to some unique characteristics of cellular networks, delay-based control can work more effectively: in wired networks, a router may handle hundreds of TCP flows at the same time and they may share the same output buffer. That makes RTT measurement noisy and delay-based congestion control unreliable. However, in cellular networks, a base station typically has separate buffer space for each user [20] and a mobile user is unlikely to have many simultaneous TCP connections. This makes RTT measurement a more reliable signal for network congestion.

However, DRWA may indeed suffer from one same problem as delay-based congestion control: inaccurate $RTT_{min}$ estimation. For instance, when a user move from a location with small $RTT_{min}$ to a location with large $RTT_{min}$, the flow may still memorize the previous smaller $RTT_{min}$ and incorrectly adjust the receive window, leading to potential throughput loss. However, we believe that the session time is typically shorter than the time scale of movement. Hence, this problem will not occur often in practice. Further, we may supplement our algorithm with an accelerometer monitoring module so that we can reset $RTT_{min}$ in case of fast movement. We leave this as our future work.

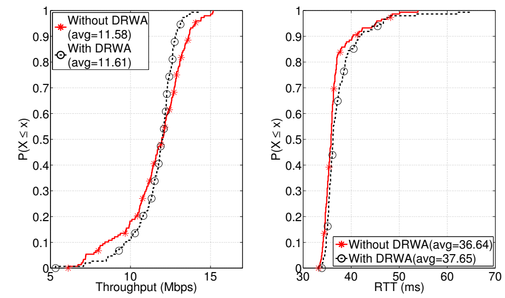



**Figure 13: DRWA has negligible impact in networks that are not bufferbloated (e.g., WiFi).**

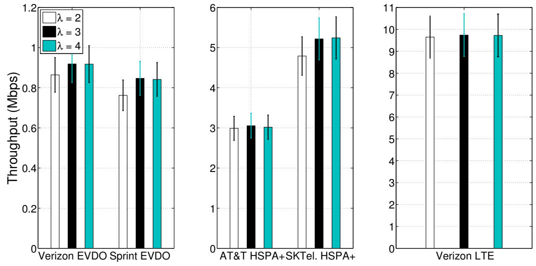
## 5.3 The Adaptive Nature of DRWA

DRWA allows a TCP receiver to dynamically report a proper receive window size to its sender in every RTT rather than advertising a static limit. Due to its adaptive nature, DRWA is able to track the variation of the channel conditions. Figure 12 shows the evolution of the receive window and the corresponding RTT/throughput performance when we move an Android phone from a good signal area to a weak signal area (from 0 second to 40 second) and then return to the good signal area (from 40 second to 80 second). As shown in Figure 12(a), the receive window size dynamically adjusted by DRWA well tracks the signal strength change incurred by movement. This leads to a steadily low RTT while the default static setting results in an ever increasing RTT as the signal strength decreases and the RTT blows up in the area of the weakest signal strength (Figure 12(b)). With regard to throughput performance, DRWA does not cause any throughput loss and the curve naturally follows the change in signal strength.

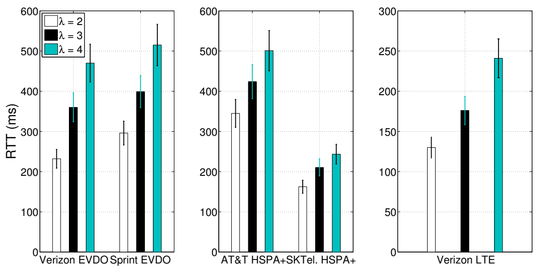
In networks that are not bufferbloated, DRWA has negligible impact on TCP behavior. That is because, when the buffer size is set to the BDP of the network (the rule of thumb for router buffer sizing), packet loss will happen before DRWA starts to rein the receive window. Figure 13 verifies that TCP performs similarly with or without DRWA in WiFi networks. Hence, we can safely deploy DRWA in smart phones even if they may connect to non-bufferbloated networks.

## 5.4 The Impact of $\lambda$

$\lambda$ is a key parameter in DRWA. It tunes the operation region of the algorithm and reflects the trade-off between

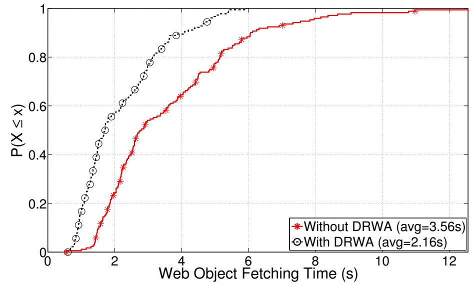



(a) Throughput

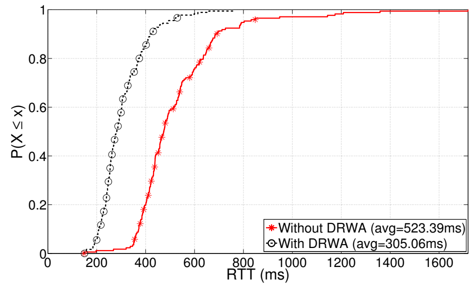



(b) Round Trip Time

**Figure 14: The impact of $\lambda$ on the performance of TCP: $\lambda = 3$ seems to give a good balance between throughput and RTT.**




(a) Web Object Fetching Time




(b) Round Trip Time

**Figure 15: DRWA improves the Web object fetching time with background downloading by 39%.**

throughput and delay. Note that when $RTT_{est}/RTT_{min}$ equals to $\lambda$, the advertised receive window will be equal to its previous value, leading to a steady state. Therefore, $\lambda$ reflects the target RTT of DRWA. If we set $\lambda$ to 1, that means we want RTT to be exactly $RTT_{min}$ and no queue is allowed to be built up. This ideal case works only if 1) the traffic has constant bit rate, 2) the available bandwidth of the wireless channel is also constant and 3) the constant bit rate equals to the constant bandwidth. In practice, Internet traffic is bursty and the channel condition varies over time. Both necessitate the existence of some buffers to absorb the temporarily excessive traffic and drain the queue later on when the load becomes lighter or the channel condition becomes better. $\lambda$ determines how aggressive we want to be in keeping the link busy and how much delay penalty we can tolerate. The larger $\lambda$ is, the more aggressive the algorithm is. It will guarantee the throughput of TCP to be maximized but at the same time introduce extra delays. Figure 14 gives the performance comparison of different values of $\lambda$ in terms of throughput and RTT[1]. This test combines multiple scenarios ranging from small to large BDP networks, good to weak signal, etc. Each has been repeated 400 times over the span of 24 hours in order to find the optimal parameter setting. As the figure shows, $\lambda = 3$ has some throughput advantage over $\lambda = 2$ under certain scenarios. Further increasing it to 4 does not seems to improve throughput but only incurs extra delay. Hence, we set $\lambda$ to 3 in our current implementation. A potential future work is to make this parameter adaptive.

[1]We plot different types of cellular networks separately since they have drastically different peak rates. Putting LTE and EVDO together will make the throughput differences in EVDO networks indiscernible.

## 5.5 Improvement in User Experience

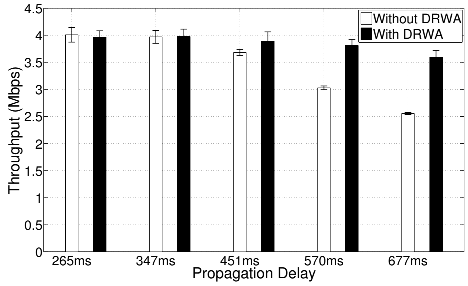
Section 4.2 lists two scenarios where the static setting of *tcp_rmem_max* may have a negative impact on user experience. In this subsection, we demonstrate that, by applying DRWA, we can dramatically improve user experience in such scenarios. More comprehensive experiment results are provided in Section 6.

Figure 15 shows Web object fetching performance with a long-lived TCP flow in the background. Since DRWA reduces the length of the queue built up in the cellular networks, it brings 42% reduction in RTT on average, which translates into 39% speed-up in Web object fetching. Note that the absolute numbers in this test are not directly comparable with those in Figure 10 since these two experiments were carried out at different time.

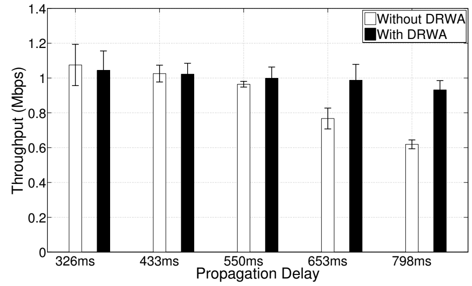
Figure 16 shows the scenario where a mobile client in Raleigh, U.S. launches a long-lived TCP download from a server in Seoul, Korea over both AT&T HSPA+ network and Verizon LTE network. Since the RTT is very long in this scenario, the BDP of the underlying network is fairly large (especially the LTE case since its peak rate is very high). The static setting of *tcp_rmem_max* is too small to fill the long fat pipe and results in throughput degradation. With DRWA, we are able to fully utilize the available bandwidth and achieve 23% $\sim$ 30% improvement in throughput.

# 6. MORE EXPERIMENT RESULTS

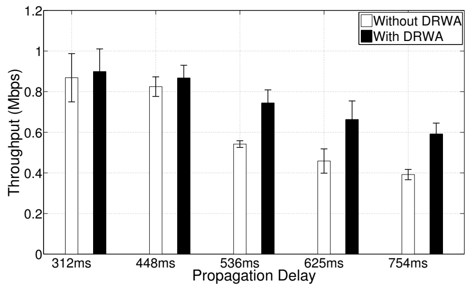
We implemented DRWA in Android phones by patching their kernels. It turned out to be fairly simple to implement DRWA in the Linux/Android kernel. It only takes around 100 lines of code. We downloaded the original kernel source codes of different Android models from their manufacturers' website, patched the kernels with DRWA and recompiled



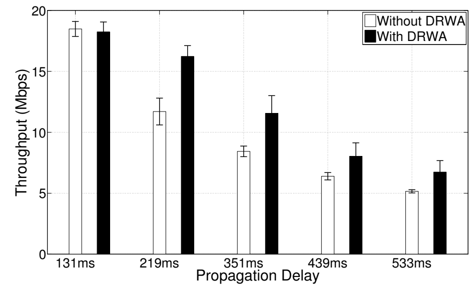

(a) Improvement in AT&T HSPA+: -1%, 0.1%, 6%, 26% and 41%




(b) Improvement in Verizon EVDO: -3%, -0.3%, 4%, 29% and 51%




(c) Improvement in Sprint EVDO: 4%, 5%, 37%, 45% and 51%




(d) Improvement in Verizon LTE: -1%, 39%, 37%, 26% and 31%

**Figure 17: Throughput improvement brought by DRWA over various cellular networks: the larger the propagation delay is, the more throughput improvement DRWA brings. Such long propagation delays are common in cellular networks since all traffic must detour through the gateway [31].**

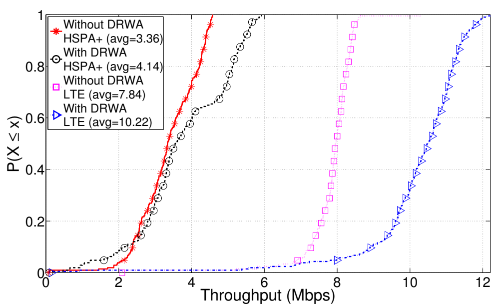



**Figure 16: DRWA improves the throughput by 23% in AT&T HSPA+ network and 30% in Verizon LTE network when the BDP of the underlying network is large.**

them. Finally, the phones were flashed with our customized kernel images.

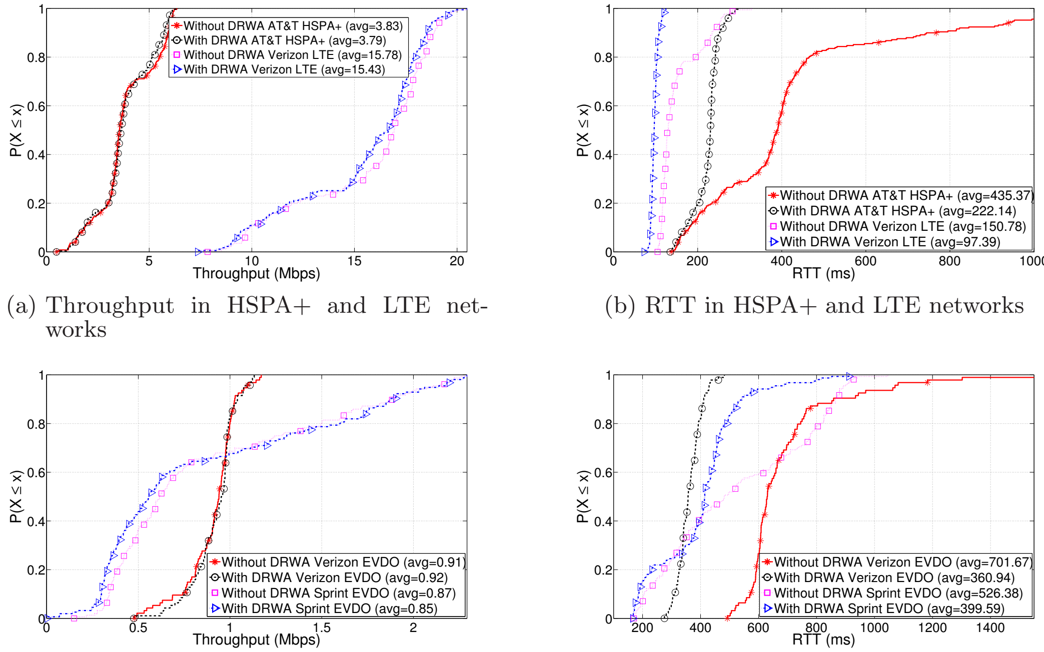
## 6.1 Throughput Improvement

Figure 17 shows the throughput improvement brought by DRWA over networks of various BDPs. We emulate different BDPs by applying *netem* [12] on the server side to vary the end-to-end propagation delay. Note that the propagation delays we have emulated are relatively large (from 131ms to 798ms). That is because RTTs in cellular networks are indeed larger than conventional networks. Even if the client and server are close to each other geographically, the propagation delay between them could still be hundreds of milliseconds. The reason is that all the cellular data have to go through a few IP gateways [31] deployed across the country by the carriers. Due to this detour, the natural RTTs in cellular networks are relatively large.

According to the figure, DRWA significantly improves the TCP throughput in various cellular networks as the propagation delay increases. The scenario over the Sprint EVDO network with the propagation delay of 754ms shows the largest improvement (as high as 51%). In LTE networks, the phones with DRWA show throughput improvement up to 39% under the latency of 219ms. The reason behind the improvement is obvious. When the latency increases, the static setting of *tcp_rmem_max* fails to saturate the pipe, resulting in throughput degradation. In contrast, networks with small latencies do not show such degradation since the static value is large enough to fill the pipe. According to our experiences, RTTs between 400 ms and 700 ms are easily observable in cellular networks, especially when using services from oversea servers. In LTE networks, TCP throughput is even more sensitive to *tcp_rmem_max* setting. The BDP can be dramatically increased by a slight RTT increase. Therefore, the static configuration easily becomes sub-optimal. However, DRWA is able to keep pace with the varying BDP.

## 6.2 RTT Reduction

In networks with small BDP, the static *tcp_rmem_max* setting is sufficient to fully utilize the bandwidth of the network. However, it has a side effect of long RTT. DRWA manages to keep the RTT around $\lambda$ times of $RTT_{min}$, which is substantially smaller than the current implementations in networks with small BDP. Figure 18 shows that the reduction in RTT brought by DRWA does not come at the cost of the throughput. We see a remarkable reduction of RTT




(a) Throughput in HSPA+ and LTE networks

(b) RTT in HSPA+ and LTE networks

(c) Throughput in EVDO networks

(d) RTT in EVDO networks

**Figure 18: RTT reduction in small BDP networks: DRWA provides significant RTT reduction without throughput loss across various cellular networks. The RTT reduction ratios are 49%, 35%, 49% and 24% for AT&T HSPA+, Verizon LTE, Verizon EVDO and Sprint EVDO networks respectively.**

up to 49% while the throughput is guaranteed at a similar level (4% difference at maximum).

Another important observation from this experiment is the much larger RTT variation under static *tcp_rmem_max* setting than that with DRWA. As Figures 18(b) and 18(d) show, the RTT values without DRWA are distributed over a much wider range than that with DRWA. The reason is that DRWA intentionally enforces the RTT to remain around the target value of $\lambda * RTT_{min}$. This property of DRWA will potentially benefit jitter-sensitive applications such as live video and/or voice communication.

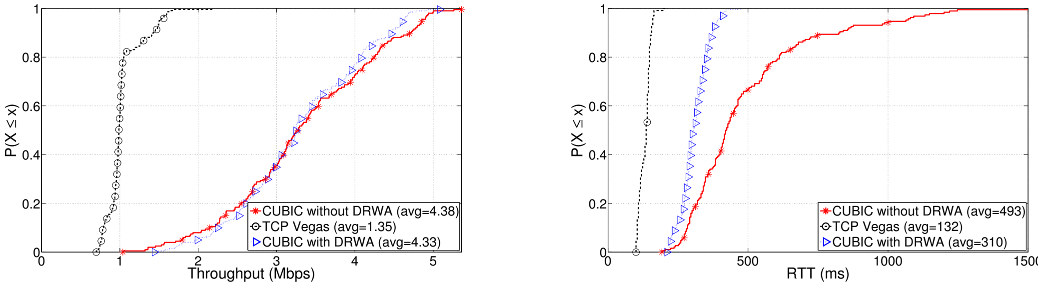
# 7. DISCUSSION

## 7.1 Alternative Solutions

There are many other possible solutions to the bufferbloat problem. One obvious solution is to reduce the buffer size in cellular networks so that TCP can function the same way as it does in conventional networks. However, there are two potential problems with this simple approach. First, the large buffers in cellular networks are not introduced without a reason. As explained earlier, they help absorb the busty data traffic over the time-varying and lossy wireless link, achieving a very low packet loss rate (most lost packets are recovered at link layer). By removing these extra buffer space, TCP may experience a much higher packet loss rate and hence much lower throughput. Second, modification of the deployed network infrastructure (such as the buffer space on the base stations) implies considerable cost.

An alternative to this solution is to employ certain AQM schemes like RED [9]. By randomly dropping certain packets before the buffer is full, we can notify TCP senders in advance and avoid long RTT. However, despite being studied extensively in the literature, few AQM schemes are actually deployed over the Internet due to the complexity of their parameter tuning, the extra packet losses introduced by them and the limited performance gains provided by them. More recently, Nichols et al. proposed CoDel [22], a parameterless AQM that aims at handling bufferbloat. Although it exhibits several advantages over traditional AQM schemes, they suffers from the same problem in terms of deployment cost: you need to modify all the intermediate routers in the Internet which is much harder than updating the end points.

Another possible solution to this problem is to modify the TCP congestion control algorithm at the sender. As shown in Figure 7, delay-based congestion control algorithms (e.g., TCP Vegas, FAST TCP [28]) are resistive to the bufferbloat problem. Since they back off when RTT starts to increase rather than waiting until packet loss happens, they may serve the bufferbloated cellular networks better than loss-based congestion control algorithms. To verify this, we compared the performance of Vegas against CUBIC with and without DRWA in Figure 19. As the figure shows, although Vegas has a much lower RTT than CUBIC, it suffers from significant throughput degradation at the same time. In contrast, DRWA is able to maintain similar throughput while reducing the RTT by a considerable amount. Moreover, delay-based congestion control protocols have a number of other issues. For example, as a sender-based solution, it requires modifying all the servers in the world as compared to the cheap OTA updates of the mobile clients. Further, since not all receivers are on cellular networks, delay-based flows will compete with other loss-based flows in other parts of the network where bufferbloat is less severe. In such sit-




(a) Throughput

(b) Round Trip Time

**Figure 19: Comparison between DRWA and TCP Vegas as the solution to bufferbloat: although delay-based congestion control keeps RTT low, it suffers from throughput degradation. In contrast, DRWA maintains similar throughput to CUBIC while reducing the RTT by a considerable amount.**

uations, it is well-known that loss-based flows unfairly grab more bandwidth from delay-based flows [3].

Traffic shaping is another technique proposed to address the bufferbloat problem [26]. By smoothing out the bulk data flow with a traffic shaper on the sender side, we would have a shorter queue at the router. However, the problem with this approach is how to determine the shaping parameters beforehand. With wired networks like ADSL or cable modem, it may be straightforward. But in highly variable cellular networks, it would be extremely difficult to find the right parameters. We tried out this method in AT&T HSPA+ network and the results are shown in Figure 20. In this experiment, we again use *netem* on the server to shape the sending rate to different values (via token bucket) and measure the resulting throughput and RTT. According to this figure, lower shaped sending rate leads to lower RTT but also sub-optimal throughput. In this specific test, 4Mbps seems to be a good balancing point. However, such static setting of the shaping parameters could suffer from the same problem as the static setting of *tcp_rmem_max*.

In light of the problems with the above-mentioned solutions, we handled the problem on the receiver side by changing the static setting of *tcp_rmem_max*. That is because receiver (mobile device) side modification has minimum deployment cost. Vendors may simply issue an OTA update to the protocol stack of the mobile devices so that they can enjoy a better TCP performance without affecting other wired users. Further, since the receiver has the most knowledge of the last-hop wireless link, it could make more informed decisions than the sender. For instance, the receiver may choose to turn off DRWA if it is connected to a network that is not severely bufferbloated (e.g., WiFi). Hence, a receiver-centric solution is the preferred approach to transport protocol design for mobile hosts [13].

## 7.2 Related Work

Adjusting the receive window to solve TCP performance issues is not uncommon in the literature. Spring et al. leveraged it to prioritize TCP flows of different types to improve response time while maintaining high throughput [25]. Key et al. used similar ideas to create a low priority background transfer service [16]. ICTCP [29] instead used receive window adjustment to solve the incast collapse problem for TCP in data center networks.

There are a number of measurement studies on TCP performance over cellular networks. Chan et al. [4] evaluated

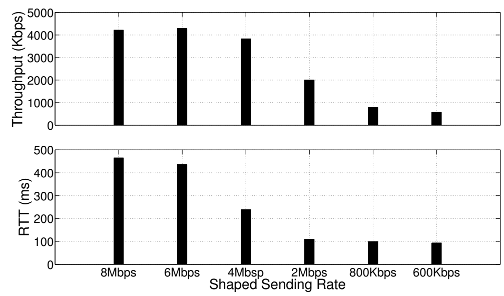



**Figure 20: TCP performance in AT&T HSPA+ network when the sending rate is shaped to different values. In time-varying cellular networks, it is hard to determine the shaping parameters beforehand.**

the impact of link layer retransmission and opportunistic schedulers on TCP performance and proposed a network-based solution called Ack Regulator to mitigate the effect of rate and delay variability. Lee [18] investigated long-lived TCP performance over CDMA 1x EVDO networks. The same type of network is also studied in [20] where the performance of four popular TCP variants were compared. Prokkola et al. [23] measured TCP and UDP performance in HSPA networks and compared it with WCDMA and HSDPA-only networks. Huang et al. [14] did a comprehensive performance evaluation of various smart phones over different types of cellular networks operated by different carriers. They also provide a set of recommendations that may improve smart phone users experiences.

# 8. CONCLUSION

In this paper, we thoroughly investigated TCP's behavior and performance over bufferbloated cellular networks. We revealed that the excessive buffers available in the existing cellular networks void the loss-based congestion control algorithms and the ad-hoc solution that sets a static *tcp_rmem_max* is sub-optimal. A dynamic receive window adjustment algorithm was proposed. This solution requires modifications only on the receiver side and is backward-compatible as well as incrementally deployable. Experiment results show that our scheme reduces RTT by 24% $\sim$ 49% while preserving similar throughput in general cases or improves the throughput by up to 51% in large BDP networks.

The bufferbloat problem is not specific to cellular networks although it might be most prominent in this environment. A more fundamental solution to this problem may be needed. Our work provides a good starting point and is an immediately deployable solution for smart phone users.

# 9. ACKNOWLEDGMENTS

Thanks to the anonymous reviewers and our shepherd Costin Raiciu for their comments. This research is supported in part by Samsung Electronics, Mobile Communication Division.

# 10. REFERENCES

[1] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: a Measurement Study and Implications for Network Applications. In *IMC'09*, 2009.

[2] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *ACM SIGCOMM*, 1994.

[3] L. Budzisz, R. Stanojevic, A. Schlote, R. Shorten, and F. Baker. On the Fair Coexistence of Loss- and Delay-based TCP. In *IWQoS*, 2009.

[4] M. C. Chan and R. Ramjee. TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation. In *ACM MobiCom*, 2002.

[5] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In *IMC'07*, 2007.

[6] W.-c. Feng, M. Fisk, M. K. Gardner, and E. Weigle. Dynamic Right-Sizing: An Automated, Lightweight, and Scalable Technique for Enhancing Grid Performance. In *PfHSN*, 2002.

[7] S. Floyd. HighSpeed TCP for Large Congestion Windows. IETF RFC 3649, December 2003.

[8] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. IETF RFC 2582, April 1999.

[9] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1:397–413, August 1993.

[10] J. Gettys. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing*, 15(3):96, May-June 2011.

[11] S. Ha, I. Rhee, and L. Xu. CUBIC: a New TCP-friendly High-speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42:64–74, July 2008.

[12] S. Hemminger. Netem - emulating real networks in the lab. In *Proceedings of the Linux Conference*, 2005.

[13] H.-Y. Hsieh, K.-H. Kim, Y. Zhu, and R. Sivakumar. A Receiver-centric Transport Protocol for Mobile Hosts with Heterogeneous Wireless Interfaces. In *ACM MobiCom*, 2003.

[14] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing Application Performance Differences on Smartphones. In *ACM MobiSys*, 2010.

[15] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. IETF RFC 1323, May 1992.

[16] P. Key, L. Massoulié, and B. Wang. Emulating Low-priority Transport at the Application Layer: a Background Transfer Service. In *ACM SIGMETRICS*, 2004.

[17] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the Edge Network. In *IMC'10*, 2010.

[18] Y. Lee. Measured TCP Performance in CDMA 1x EV-DO Networks. In *PAM*, 2006.

[19] D. Leith and R. Shorten. H-TCP: TCP for High-speed and Long-distance Networks. In *PFLDnet*, 2004.

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

[21] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, and A. Joseph. Multi-layer Tracing of TCP over a Reliable Wireless Link. In *ACM SIGMETRICS*, 1999.

[22] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5):20:20–20:34, May 2012.

[23] J. Prokkola, P. H. J. Perälä, M. Hanski, and E. Piri. 3G/HSPA Performance in Live Networks from the End User Perspective. In *IEEE ICC*, 2009.

[24] D. P. Reed. What's Wrong with This Picture? The end2end-interest mailing list, September 2009.

[25] N. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. Bershad. Receiver Based Management of Low Bandwidth Access Links. In *IEEE INFOCOM*, 2000.

[26] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband Internet Performance: a View from the Gateway. In *ACM SIGCOMM*, 2011.

[27] K. Tan, J. Song, Q. Zhang, and M. Sridharan. Compound TCP: A Scalable and TCP-Friendly Congestion Control for High-speed Networks. In *PFLDnet*, 2006.

[28] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Transactions on Networking*, 14:1246–1259, December 2006.

[29] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *ACM CoNEXT*, 2010.

[30] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-distance Networks. In *IEEE INFOCOM*, 2004.

[31] Q. Xu, J. Huang, Z. Wang, F. Qian, A. Gerber, and Z. M. Mao. Cellular Data Network Infrastructure Characterization and Implication on Mobile Content Placement. In *ACM SIGMETRICS*, 2011.

[32] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP Congestion Avoidance Algorithm Identification. In *IEEE ICDCS*, 2011.

# APPENDIX

## A. LIST OF EXPERIMENT SETUP

See Table 1.

## B. SAMPLE *TCP_RMEM_MAX* SETTINGS

See Table 2.

| Figure | Client | | | | Server | | Network | |
|---|---|---|---|---|---|---|---|---|
| | Location | Signal Strength | Model | Traffic Pattern | Location | Congestion Control | Carrier | Network Type |
| 3 | Raleigh<br>Chicago<br>Seoul | Good<br>Weak | Linux Laptop | Long-lived TCP | Raleigh<br>Princeton<br>Seoul | CUBIC | AT&T<br>Sprint<br>T-Mobile<br>Verizon<br>SK Telecom | HSPA+<br>EVDO<br>LTE<br>WiFi |
| 4 | Raleigh | Good | Galaxy S2 | Traceroute<br>Long-lived TCP | Raleigh | CUBIC | AT&T | HSPA+ |
| 5 | Raleigh | Good | Galaxy S2<br>Droid Charge | Ping | Raleigh | - | AT&T<br>Verizon | HSPA+<br>EVDO<br>LTE |
| 6 | Raleigh | Good | Linux Laptop | Long-lived TCP | Raleigh | CUBIC | AT&T<br>Sprint<br>T-Mobile<br>Verizon | HSPA+<br>EVDO |
| 7 | Raleigh | Good | Linux Laptop | Long-lived TCP | Raleigh | NewReno<br>Vegas<br>CUBIC<br>BIC<br>HTCP<br>HSTCP | AT&T | HSPA+ |
| 8 | Raleigh | Good | Linux Laptop<br>Mac OS 10.7 Laptop<br>Windows 7 Laptop<br>Galaxy S2<br>iPhone 4<br>Windows Phone 7 | Long-lived TCP | Raleigh | CUBIC | AT&T | HSPA+ |
| 9 | Raleigh | Good | Galaxy S2<br>Droid Charge | Long-lived TCP | Raleigh | CUBIC | AT&T<br>Verizon | HSPA+<br>LTE |
| 10 | Raleigh | Good | Galaxy S2 | Short-lived TCP<br>Long-lived TCP | Raleigh | CUBIC | AT&T | HSPA+ |
| 11 | Raleigh | Good | Galaxy S2<br>Droid Charge | Long-lived TCP | Seoul | CUBIC | AT&T<br>Verizon | HSPA+<br>LTE |
| 12 | Raleigh | Good<br>Weak | Galaxy S2 | Long-lived TCP | Raleigh | CUBIC | AT&T | HSPA+ |
| 13 | Raleigh | Good | Galaxy S2 | Long-lived TCP | Princeton | CUBIC | - | WiFi |
| 14 | Raleigh<br>Seoul | Good<br>Weak | Galaxy S2<br>Droid Charge<br>EVO Shift | Long-lived TCP | Raleigh | CUBIC | AT&T<br>Verizon<br>Sprint<br>SK Telecom | HSPA+<br>LTE<br>EVDO |
| 15 | Raleigh | Good | Galaxy S2 | Short-lived TCP<br>Long-lived TCP | Raleigh | CUBIC | AT&T | HSPA+ |
| 16 | Raleigh | Good | Galaxy S2<br>Droid Charge | Long-lived TCP | Seoul | CUBIC | AT&T<br>Verizon | HSPA+<br>LTE |
| 17 | Raleigh | Good | Galaxy S2<br>Droid Charge<br>EVO Shift | Long-lived TCP | Raleigh | CUBIC | AT&T<br>Verizon<br>Sprint | HSPA+<br>LTE<br>EVDO |
| 18 | Raleigh | Good | Galaxy S2<br>Droid Charge<br>EVO Shift | Long-lived TCP | Raleigh | CUBIC | AT&T<br>Verizon<br>Sprint | HSPA+<br>LTE<br>EVDO |
| 19 | Raleigh | Good | Galaxy S2 | Long-lived TCP | Raleigh | CUBIC<br>Vegas | AT&T | HSPA+ |
| 20 | Raleigh | Good | Galaxy S2 | Long-lived TCP | Raleigh | CUBIC | AT&T | HSPA+ |

**Table 1: The setup of each experiment**

| | Samsung Galaxy S2 (AT&T) | HTC EVO Shift (Sprint) | Samsung Droid Charge (Verizon) | LG G2x (T-Mobile) |
|---|---|---|---|---|
| WiFi | 110208 | 110208 | 393216 | 393216 |
| UMTS | 110208 | 393216 | 196608 | 110208 |
| EDGE | 35040 | 393216 | 35040 | 35040 |
| GPRS | 11680 | 393216 | 11680 | 11680 |
| HSPA+ | 262144 | - | - | 262144 |
| WiMAX | - | 524288 | - | - |
| LTE | - | - | 484848 | - |
| Default | 110208 | 110208 | 484848 | 110208 |

**Table 2: Maximum TCP receive buffer size (*tcp_rmem_max*) in bytes on some sample Android phones for various carriers. Note that these values may vary on customized ROMs and can be looked up by looking for "setprop net.tcp.buffersize.*" in the *init.rc* file of the Android phone. Also note that different values are set for different carriers even if the network types are the same. We guess that these values are experimentally determined based on each carrier's network conditions and configurations.**