Adaptive Flow Control for TCP on Mobile Phones

Shruti Sanadhya
shruti.sanadhya@cc.gatech.edu
Georgia Institute of Technology

Raghupathy Sivakumar
siva@ece.gatech.edu
Georgia Institute of Technology

***Abstract*—The focus of this work is to study the efficacy of TCP's flow control algorithm on mobile phones. Specifically, we identify the design limitations of the algorithm when operating in environments, such as mobile phones, where flow control assumes greater importance because of device resource limitations. We then propose an *adaptive flow control* (AFC) algorithm for TCP that relies not just on the available buffer space but also on the application read-rate at the receiver. We show, using $NS2$ simulations, that AFC can provide considerable performance benefits over classical TCP flow control.**

I. Introduction

The flow control mechanism in classical TCP is simple and conservative. It operates based on buffer occupancy, and does not track application read rate directly. For most conventional network scenarios - both wireline and wireless - this is not a serious concern as the application read-rate is rarely the dominant bottleneck. The limitations of a simplistic flow control strategy do not adversely impact a TCP connection's performance if flow control does not kick in very often. However, with the growing use of *mobile phone* platforms for data application access, it is worthwhile studying TCP flow control in more depth. The constrained processing resources on such platforms make it more probable that flow control assumes a more significant role in the throughput enjoyed by a connection.

Thus, *the focus of this work is to study TCP's flow control algorithm, identify its limitations for mobile phones*[1], *and propose a new flow control algorithm for such platforms.* We observe, through experimentation, that the throughput performance of a flow control bottlenecked TCP connection can be as low as 20% of the expected throughput. We identify a variety of reasons for the performance degradation that are directly attributable to the flow control algorithm employed in classical TCP. To better ground our observations we also perform a control theoretic analysis of the TCP flow control algorithm and show that it reduces to an *integral controller*, which in turn has a non decaying oscillation function with an amplitude that is proportional to both the *peak application read-rate* and the *fluctuation frequency* of the read-rate.

We therein motivate a more sophisticated flow control algorithm that not only relies on the available buffer space, but *also explicitly accounts for the application read-rate* in its decisions. We propose such an algorithm called *adaptive flow control (AFC)* for TCP. Besides explicitly tracking the application read-rate, AFC also has a set of key design elements that are targeted toward optimizing performance for connections operating in a flow control dominated regime. We propose AFC as a TCP option so that AFC-enabled network stacks are still backward compatible to communicate with non AFC-enabled stacks. We evaluate AFC using $NS2$ based simulations, and show that AFC delivers considerable performance improvements over classical TCP in flow control dominated regimes, exhibits TCP friendliness, and is robust to a wide variety of network and application characteristics.

This work was supported in part by the National Science Foundation under grants CNS-1017234 and CCF-1017984.

[1]While a majority of our observations and proposed solutions would aid other environments that are flow control dominated as well, we restrict the focus of this paper to only mobile phones.

II. Background and Motivation

A. TCP Flow Control Basics

The basic flow control strategy employed in TCP is for the receiver to *advertise* to the sender, using the $rwnd$ field in the TCP ACK, the available space in the buffer in relation to the highest in-sequence sequence number received [1]. The sender will transmit new segments only if the highest unacknowledged sequence number it has transmitted is smaller than the sum of the lowest unacknowledged sequence number and the $min(rwnd, cwnd)$, where $cwnd$ is the congestion window maintained by the sender. If the rate at which data is consumed by the receiving application is lower than the network rate, the receive buffer occupancy will increase and this in turn will result in lower $rwnd$ values advertised by the receiver. An extreme scenario is when the receive buffer is full and the receiver advertises an $rwnd$ of *zero*. Upon receipt of a such a zero window advertisement, the sender freezes its transmission completely and awaits an *explicit open window advertisement* from the receiver. Eventually, when one MSS worth of space opens up in the receive buffer, the receiver sends an open window by advertising a non-zero $rwnd$ value.

B. Problems with TCP Flow Control on Mobile Phones

1) **Flow control bottlenecks occur more often***:* Mobile phones, in spite of the advances made in their hardware capabilities, continue to be resource limited compared to traditional PCs and laptops. Such limitations span over the processing capabilities, the sizes of the different tiers of storage, and other dimensions of computing. There are a wide variety of reasons for such limitations ranging from the requirement for low power operations, form factor constrains and cost. Figures 1(a) and 1(b) present comparative CPU allocation results for an FTP application running on a laptop (Dell Inspiron 1525 with the Ubuntu 9.10 OS) and a mobile phone (Google G1 with the Android OS) respectively. In both cases, a large file ($\sim$2GB)

is downloaded from an Internet server down to the client. As the download progresses, three competing workloads; email, web browsing and progressive video download - are introduced at different times. The impact on the CPU allocation for the FTP process is measured using the top utility. We observe that on the laptop the FTP client is relatively unaffected by the background processes and remains at around 50% allocation. However, for the FTP client on the mobile phone, the CPU occupancy fluctuates between 60% and $\sim$0% during the download.

Investigating the FTP connection further, we observe no zero window advertisements from the laptop, whereas there are 21 zero window advertisements from the mobile phone. This clearly shows the increase in the impact of flow control on the mobile phone, and we study the performance consequence of this impact next.

2) **TCP flow control is inefficient***:* Even when application read-rate fluctuations occur, an ideal flow control algorithm should still deliver throughput equal to the minimum of the average network rate and the average application read-rate - min(avg. network rate NW, avg. application read-rate AAR). To evaluate TCP's flow control algorithm under fluctuating application read-rate conditions we conduct simulations in $NS2$ with the following setup: sender and receiver connected over a direct link; RTT of 530ms; network rate of 15 Mbps; average application read rate of 4 Mbps, with a fluctuation profile of $<$0, 6, 6$>$ (period of 1 RTT); and receive buffer size equal to the perceived BDP (min(NW, AAR)*RTT = 256KB). While we pick these values as an example (e.g. TCP long-haul connection over a wi-fi last leg), we generalize the values for the parameters in the setup to a broader set both later in this section and in Section V.

The expected throughput for the above setup is equal to 4Mbps ($min(15Mbps, 4Mbps)$) even after taking into account the fluctuations. *However, the aggregate throughput observed (see Figure 1(c)) in the simulations is only 1.45Mbps, a degradation of 63%.* Note that given the high network rate assumed, there are no congestion bottlenecks influencing the performance, and hence this degradation is directly due to the flow control behavior of TCP. We attribute this degradation to TCP flow control's stop and go behavior that does not allow the connection to track the application read rate effectively. We delve into specific design issues next.

C. Design Insights into TCP Flow Control Limitations

We use three different scenarios where TCP flow control leads to under-performance and therein highlight some of the design issues. NS2 simulations are used to determine TCP throughput for the different scenarios[2]. In the different scenarios, the round trip time for each connection is 530ms. The read rate of the receiving application fluctuates in a pattern of $<$AR1,AR2$>$ or $<$0,AR,AR$>$ with a time period of 1 RTT. If the pattern is $<$AR1,AR2$>$, the application reads at AR1 for one RTT, then at AR2 for another RTT and back to AR1. If its $<$0,AR,AR$>$, it does not read any data for one RTT, then reads at the rate of AR for two RTTs and again goes back to not reading, and so on. The scenarios we consider are:

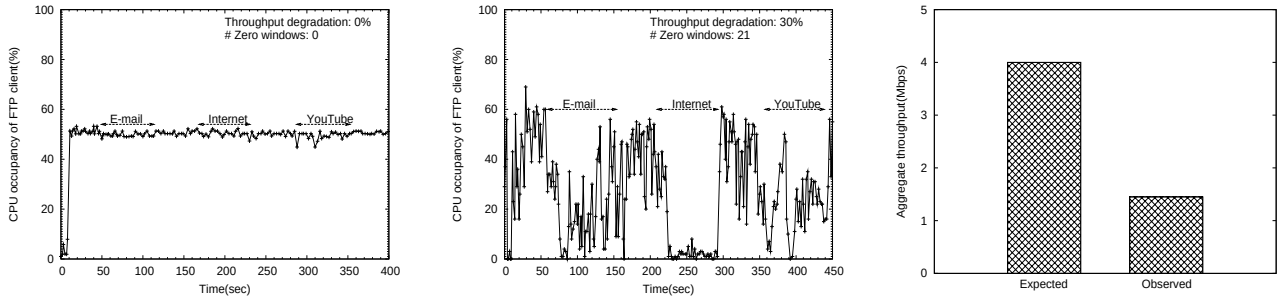
1) ***Fluctuating application rate:*** The variations in application read rate affect the advertised window of a TCP connection. As the window does not converge to a steady value, the throughput of the receiving application also fluctuates, worse than expected. Let's consider the setup: (a) RTT = 1s; (b) Application profile: $<$2, 6$>$ Mbps with the fluctuation interval = 1 RTT; (c) Average Application Rate(AAR) = 4 Mbps; NW = 4 Mbps, i.e. NW $=$ AAR; (d) B is set as min(NW, AAR)*RTT=500KB=4Mb (the ideal BDP).

The expected application throughput is min(NW,AAR)=4 Mbps, but the throughput observed in the experiment is only 2.9 Mbps, a 25% degradation from the expected value. The performance degradation occurs because of TCP's flow control behavior. In steady state the sender tries to send at 4Mbps. If the application is reading at 2Mbps, every half RTT 1Mb of data would be read by the application and 1Mb stored in the buffer. At the end of the first half RTT, the advertised window is 3Mb. At the end of 1RTT, the application would have read another 1Mb and stored 1Mb in the buffer, the advertised window reduces to 2Mb. In the next half RTT, the application reads at the rate of 6Mbps, it reads the 2Mb stored data in the buffer and also the 1Mb received from the sender, which is (3Mb(advertised window an RTT back)-2Mb(outstanding data)). The latest advertised window is now 4Mb. In the next half RTT, the receiver receives another 1Mb, which is 2Mb(the advertised window an RTT back)-1Mb(traffic outstanding in the last RTT). The receiving application reads the entire received 1Mb and advertises a window of 4Mb. The same sequence repeats from there on.

Thus, if the buffer is sized at the prescribed value of the BDP (4Mb), the connection rate is throttled down to 2Mbps when the application read rate is 2Mbps (flow control due to application read rate limitation), but is capped at 4Mbps (flow control due to buffer size) even when the application read rate grows to 6Mbps. The application thus reads 2Mb in the first RTT and 4Mb in the second RTT, and the observed throughput at the application is thus (2+4)/2 Mbps = 3Mbps, while the ideal expected value is 4Mbps.

2) ***Zero windows:*** Extreme fluctuations in application read rate result in zero window advertisements. In TCP's flow control, every zero window advertisement carries with it a deterministic throughput penalty due to the time taken for the window to be re-opened to pre-zero window levels. At any zero window occurrence the sender waits for up to *two round trip times(RTTs) before it can send any* **substantial** *amount of new data* even if the application starts reading immediately after the zero window was advertised; an RTT to wait before sending a zero window probe and another RTT to get a window larger than one to send more data. Hence, a higher frequency

[2]Basic flow control features such as finite-size receive buffer, dynamic advertised window and zero window management were added to the NS2 TCP implementation as NS2 does not support these currently. A configurable application read rate parameter was also added to simulate different application patterns.



(a) FTP on laptop

(b) FTP on Android

(c) Impact of application read rate fluctuations on TCP throughput

Fig. 1. Analysis of TCP Flow Control on Mobile Phones

of zero windows results in a larger number of such under-utilizing periods. We use the following parameters for the evaluation of this scenario: (a) RTT = 530ms; (b) Application profile of <0, 6, 6> (AAR = 4Mbps); (c) NW = 15Mbps; and (d) B is set to 256 KB (perceived BDP).

The expected application throughput is min(NW,AAR)=4 Mbps, but the throughput observed in NS2 is 1.45 Mbps (a 64% degradation), as shown in Figure 1(c). While some of the performance degradation is attributed to the reasons outlined earlier, the higher severity of the degradation is due to the zero window occurrences. When the application stops reading, the receive-buffer fills up, resulting in zero windows being sent and the sender being stalled. As soon as the application starts reading, an open window is sent to the sender and the sender sends one segment. The ACK for this packet, which arrives an RTT later, then allows the sender to send more packets. The receiver thus ends up reading AAR*RTT bytes in 3 RTTs, whenever this happens. In this particular example, 328 zero windows are observed in a connection of 600s, thus 656 out of 1132 RTTs are spent idle. There are no congestion losses.

Thus, whenever the zero window occurrences in the lifetime of a TCP connection increases, the performance degradation (difference between the expected throughput and the observed throughput) increases.

3) ***Fluctuating network rate:*** Apart from the application read rate, the network rate can also fluctuate. This introduces new complications. Ideally the TCP throughput can grow with increase in bandwidth, but the limited buffer or zero window events may prevent the sender from using higher congestion windows. The receiver may never learn of this available bandwidth and be unable to resize its buffer based on techniques like dynamic right sizing[2], auto-tuning[3], etc. We use the following parameters for this scenario: (a) RTT = 530ms; (b) Application profile: <0, 6, 6> Mbps with the fluctuation interval = 1 RTT, AAR=4 Mbps; (c) Network profile: <2, 4, 4> Mbps with the fluctuation interval = 1 RTT; and (d) buffer B set to 128KB/213KB (perceived/ideal Bandwidth Delay Product).

In this scenario, the application is expected to enjoy a throughput of min(average network rate, average application rate), i.e., min(3.3Mbps, 4Mbps). However, to achieve that performance, the receiver needs to make sure that the receive buffer is tuned to the network. Current buffer resizing solutions [2], [3], [4] depend on data rate observed at the receiver to calculate the optimal advertised window and buffer size. In this scenario, zero windows occur while the application is not reading, the sender stalls and while the sender is stalled, the fact that the network rate has increased does not influence the buffer calculation at the receiver. Thus the apparent network rate $N_p \sim 2Mbps$ is much lesser than the actual network rate $N_a = (2{+}4{+}4)/3 = 3.3Mbps$. The observed throughput with a buffer size of 2Mbps*530ms=128KB, is 0.67Mbps, which is 20% of the ideal throughput. Even when the buffer is scaled up to 213KB, i.e. 3.3Mbps*530ms, the observed throughput is still only 1.45Mbps.

Thus, when both the network rate and the application rate fluctuate, the lower throughput rates experienced when the application read rate is low can also impact the achievable network throughput even when the application read rate eventually increases.

D. Study of a Trivial Buffer-based Solution

We now briefly argue for why a buffer provisioning based solution is not desirable to tackle the problems discussed thus far. While we consider a set of different scenarios we present only the worst-case scenario below. Interested readers are referred to [5] for the other scenarios.

Consider a scenario where both the network rate and application read-rate fluctuate. It is possible in this scenario that the connection does not leverage the crests or highs of the network rate *because it is idle due to recovery from zero-windows when the network rate is high.* For example, consider the $Scenario$ where the application rate fluctuates as $(0, 18, 18)$ (period of one RTT), and the network rate fluctuates as $(3, 15, 15)$ (same period). In this scenario, a zero window will be triggered in the first RTT, and the connection will end up idling for the subsequent two round-trip times and hence will not realize that a rate as high as 15Mbps was possible during that period. *In our simulation study of the above scenario, we observe a throughput of 3Mbps in contrast to the expected throughput of 11Mbps.*

This problem can be averted only if the connection is prevented from idling for all round-trip times. While provisioning

the buffer based on the average achievable network rate would suffice, note that the connection has no way of determining the achievable network rate as it will never encounter the high rate periods. Instead, the only deterministic approach to averting the problem is to provision the buffer based on the average application rate. Independent of whether the average application rate is higher or lower than the average network rate, this will suffice. Thus, in order to overcome the idle periods when recovering from zero-windows, the buffer required when both application read-rate and network rate fluctuate is as follows:

$$B_{req} = 3 * AAR * RTT \tag{1}$$

The problem with this strategy, though, is that the AAR for a mobile platform can be arbitrarily high when compared to the possible network rates. For example, on the Android G1 phone, we were able to observe application read-rates as high as 100Mbps (under low CPU load conditions). Hence, the buffer allocation required could be orders of magnitude higher than what the connection throughput will necessitate (e.g. a 2Mbps network rate scenario will ideally need only 125KB of buffer allocation, whereas the provisioning based strategy will necessitate 18.75MB of buffer allocation). Also note that this allocation is on a per connection basis. While requiring orders of magnitude more memory allocation is bad in itself, the demands become onerous when considering the memory limitations of typical mobile phones. Furthermore, even if such allocation can be achieved on the mobile phones, the server (sender) side buffer will have to be of similar proportions in order to support this strategy. Considering a typical web server serving tens and thousands of connections, such onerous buffer allocation quickly becomes untenable. Even assuming that memory is not an issue, the AAR still has to be accurately tracked at the receiver in order to achieve the provisioning. Hence, the question we ask ourselves in the rest of the paper is that if the application read-rate is already being monitored, *could a better solution be derived to achieve the expected performance?*

III. Theoretical Analysis

A. Control theoretic analysis of TCP flow control

TCP is a closed loop system. The sender sends data to the receiver, then waits for feedback from the receiver to determine how much data to send next. We model this control system in the following analysis. For purposes of this analysis we assume that the connection is purely flow control restricted, and the connection rate is TCP, W is the advertised window, AR is the rate at which the data is read at the receiver, B_0 is the receive buffer size and B is the buffer occupancy at any given time. From this we can represent W as follows:

$$W = B_0 - B \tag{2}$$

The buffer is filled in at the rate of TCP and drained by the application at AR. Thus,

$$dB/dt = TCP - AR \tag{3}$$

Differentiating (2) and using (3), we get

$$W' = dW/dt = AR - TCP \tag{4}$$

Note that $0 \leq B \leq B_0$ and $0 \leq W \leq B_0$. Thus,

$$W = min(B_0, \int W' dt) \tag{5}$$

If we consider TCP as a system variable, the target value of TCP is AR and the error err in this variable is the deviation in throughput:$(AR - TCP)$, which is the rate at which W grows:

$$W' = (AR - TCP) = err \tag{6}$$

As network is not the bottleneck, TCP is proportional to the receive window W. Assuming that round trip time RTT remains constant for a connection.

$$TCP = \alpha\ W, \text{ where } \alpha = 1/RTT \tag{7}$$

$$\text{using (5), } TCP = \alpha\ min(B_0, \int W'\ dt) \tag{8}$$

$$\text{using (6), } TCP = \alpha\ min(B_0, \int err\ dt) \tag{9}$$

For now, let's assume B_0 to be unbounded. *Then TCP is entirely dependent on the integral of the deviation from AR.* In control theory, such systems are termed $Integral(I)$ systems [6]. In the following analysis, we look at some characteristics of this system and its implication on TCP's performance.
Eliminating TCP from the equations (6) and (7):

$$W' = AR - \alpha W \tag{10}$$

$$\text{on reorganizing, } W' + \alpha W = AR \tag{11}$$

This is a linear first-order differential equation, where W and AR are functions of time. Solving it by the method of *integrating factor*, we have:

$$\text{Integrating factor} : e^{\alpha t}$$

$$\text{multiplying (11) with integrating factor}$$

$$e^{\alpha t}W' + \alpha e^{\alpha t}W = e^{\alpha t}AR \tag{12}$$

$$\text{on simplifying, } \frac{d}{dt}(e^{\alpha t}W) = e^{\alpha t}AR \tag{13}$$

$$\text{on integrating, } \int_{t=0}^{t} \frac{d}{dt}(e^{\alpha t}W) = \int_{t=0}^{t} (e^{\alpha t}AR)dt \tag{14}$$

Now let us assume that the application fluctuates from 0 to 2 A_0 as a sinusoid function of time with a time-period of T.[3]

$$AR = A_0(1 + \sin \omega t), \text{where } \omega = 2\pi/T \tag{15}$$

using (15) in (14) and simplifying ,

$$e^{\alpha t}W - B_0 = A_0 \int_{t=0}^{t} e^{\alpha t}dt + A_0 \int_{t=0}^{t} e^{\alpha t} \sin \omega t dt \tag{16}$$

$$\text{on solving, } W = e^{-\alpha t}\left[B_0 - \frac{A_0}{\alpha} + \frac{A_0 \sin\theta}{\sqrt{\alpha^2 + \omega^2}}\right] + \frac{A_0}{\alpha} +$$

[3]Note that any other periodic application profile can be represented as a sum of sine/cosine functions[7].

$$A_0 \frac{\sin(\omega t - \theta)}{\sqrt{\alpha^2 + \omega^2}} \text{ ,where } \theta = \tan^{-1}\left(\frac{\omega}{\alpha}\right) \tag{17}$$

The error err in TCP can thus be computed from (6) as:

$$err = W' \tag{18}$$

differentiating (17) and using in (18)

$$err = -\alpha e^{-\alpha t}\left[B_0 - \frac{A_0}{\alpha} + \frac{A_0 \sin\theta}{\sqrt{\alpha^2+\omega^2}}\right] + \frac{A_0\omega}{\sqrt{\alpha^2+\omega^2}}\cos(\omega t - \theta) \tag{19}$$

In steady state: $e^{-\alpha t} \to 0$, thus (19) becomes

$$err = \frac{A_0\omega}{\sqrt{\alpha^2+\omega^2}}(\cos(\omega t - \theta)) \tag{20}$$

$$\text{further, } err = A_0 \sin\theta(\cos(\omega t - \theta)) \tag{21}$$

Thus, *for fluctuating applications, the difference between TCP rate and application read rate exhibits non-decaying oscillations. The amplitude of these oscillations increases with the peak application read rate and cycles with the fluctuation time-period.*

From (7) and (17), TCP is:

$$TCP = \alpha e^{-\alpha t}\left[B_0 - \frac{A_0}{\alpha} + \frac{A_0 \sin\theta}{\sqrt{\alpha^2+\omega^2}}\right] + A_0\left[1 + \frac{\alpha\sin(\omega t - \theta)}{\sqrt{\alpha^2+\omega^2}}\right] \tag{22}$$

which in steady state becomes:

$$TCP = A_0\left[1 + \frac{\alpha}{\sqrt{\alpha^2+\omega^2}}\sin(\omega t - \theta)\right] \tag{23}$$

This has a marked deviation from AR, both in frequency pattern and in the amplitude. As the frequency of oscillations increases, the phase difference in TCP and AR also increases. This lag translates into increased settling time, i.e., time taken to converge to AR, for TCP. Equation (23) presents a control system model for TCP's flow control. In practice, the receive buffer B_0 imposes an upper bound on TCP data rate. Following from (9), the actual TCP data rate is given by:

$$TCP = min\left(\alpha B_0, A_0\left[1 + \frac{\alpha}{\sqrt{\alpha^2+\omega^2}}\sin(\omega t - \theta)\right]\right) \tag{24}$$

Depending on the relation between the two terms in (24), TCP throughput can saturate at the rate of αB_0, i.e., B_0/RTT or grow as much as the application demands. Saturations cause TCP to under-perform. Thus, we conclude that TCP throughput is dependent on the receive buffer size, the application fluctuation frequency and the amplitude of fluctuations in the application read rate.

B. Basis for an Adaptive Flow Control Algorithm

We observe in the previous section that:

1) *Current TCP flow control is an $Integral(I) - only$ control system.* As is well known in control theory, Integral systems are used as corrective components in $Proportional(P)$ control systems. An $I - only$ system can increase settling time(θ in equation (23)), making it respond slower to disturbances/fluctuations.
2) *If B_0 is not large enough to accommodate the application read rate and its fluctuations, TCP send rate is capped by B_0/RTT (as shown in equation* (24)*).*

A corrective term needs to be added in equation (7) to compensate for the impact of integral action and bound of B_0. We propose that this term be AR, i.e. the application read rate. Equation (7) thus takes the form of:

$$TCP = \alpha W + AR \tag{25}$$

working out equation (6)

$$W' = AR - TCP \tag{26}$$

$$\text{using (25), } W' = AR - \alpha W - AR \tag{27}$$

$$\text{i.e., } W' = -\alpha W \tag{28}$$

$$\text{on solving, } W = B_0 e^{-\alpha t} \tag{29}$$

differentiating (29) and using in (18),

$$err = -\alpha B_0 e^{-\alpha t} \tag{30}$$

Note that (30) presents a decaying error in TCP send rate. From equations (25) and (29), TCP takes the form:

$$TCP = \alpha B_0 e^{-\alpha t} + AR \tag{31}$$

which converges to AR at steady state, shows no lag and is not bound by the B_0/RTT limit. Thus, *if TCP starts reacting to the application rate, it would be able to scale up to its target value, even in the face of fluctuations*. In the next section, we discuss how to translate this theory into a practical implementation.

IV. Design elements and algorithm

In this section we present an *adaptive flow control (AFC)* algorithm for TCP that will help achieve expected throughput performance even in a flow control dominated regime. A key goal of the proposed solution is to deliver such performance without requiring a large buffer allocation. We first present an overview of the key design elements in AFC, and then describe the algorithm. Interested readers may refer to [5] for detailed pseudocode.

A. Key Design Elements

*1) **Using Application Read Rate:*** The first design element in AFC follows directly from the theoretical analysis presented in Section III. While classical TCP flow control uses the advertised buffer space from the receiver as the flow control window, AFC relies on both the advertised available buffer space in the receive buffer *and the application read-rate* in determining the flow control window:

$$W_{fc} = B + AR * RTT \tag{32}$$

Just like the advertised buffer space, the application read rate AR is also fed back to the sender from the receiver. We defer details on how the application read rate is monitored and tracked till later in the section. Once the flow control window

W_{fc} is determined, AFC uses the window in exactly the same fashion as in classical TCP. In other words, the number of outstanding packets is controlled to be the minimum of the congestion control window and the flow control window. The use of the application read rate in determining the flow control window thus allows AFC to better react to application read rate changes instead of relying only on buffer over-provisioning.

2) ***Handling Overflows:*** Classical TCP flow control is *conservative* to an extent where the flow control algorithm *will never result in buffer overflows at the receiver*. The TCP sender will at no point send more data than what the receiver buffer can accommodate. Hence, all losses experienced by the connection are directly attributable to congestion.

However, in AFC the flow control window is computed to be a sum of two factors: the available buffer space and the application read rate per RTT. If the application read rate is over estimated or suddenly decreases, overflows at the receive buffer will occur. Such losses however should not be attributed to congestion as the flow control algorithm causes them. Thus, AFC is specifically designed to keep such flow control induced losses from impacting the congestion control algorithm. In classical TCP, when a zero window is received by the sender with an ACK sequence number of S_{zw}, the sender explicitly freezes all congestion control decisions and ignores loss indicators (both triple duplicate ACKs and timeouts) for any sequence numbers greater than S_{zw} till an explicit open window is received from the receiver. In AFC, duplicate ACKs or timeouts may still be triggered by packet drops at the receiver for packets with sequence number S_{oe}, where $S_{oe} > S_{zw} + Receive\ buffer$. These duplicate ACKs can arrive even after the open window event. AFC hides this by *recording the time $ts\_recover$ of the arrival of the open window and further suppressing all congestion indicators till an ACK is received for data sent after $ts\_recover$*. Furthermore, in order to fast track the successful transmission of such overflow data, the next sequence number to transmit(snd_nxt) at the sender side is reset to S_{zw}[4] upon the receipt of an open window. Such fast-tracking of the transmissions beyond S_{zw} prevents those packets from being handled by the (slower) retransmission mechanism in TCP.

The combination of the ignoring of losses after a zero window and the resetting of the snd_nxt averts both congestion control and reliability problems due to the overflow.

3) ***Proactive feedback:*** The receiver in classical TCP sends an ACK *only on the receipt of a segment*. Thus, any feedback from the receiver to the sender is dependent on the arrival of new data. When recovering from a zero window state, this property is clearly undesirable. Even if the application read rate climbs rapidly, the receiver will send the first open window to the sender as soon as one MSS worth of space opens up in the buffer. Thus, for an entire round-trip time after that open window transmission, the receiver cannot send any further feedback to the sender even if the buffer is completely drained. Consequently, the sender will send only one segment for that round-trip time, and wait for the next ACK to arrive before it will expand its flow control window fully. In AFC, this limitation is averted by requiring the receiver to send feedback not just upon receipt of data but *also when there is a drastic change in the buffer state and application read-rate.* Thus, when recovering from a zero window state, the receiver will send not merely the first open window when one MSS worth of buffer is available, but also follow it up with more reports about the AR and B if the application drains the buffer quickly. This allows the sender to take more accurate flow control decisions.

Note that such a design element can also be modulated by a mechanism similar to the delayed ACK timer. Essentially, whenever a *proactive* ACK has to be sent by the receiver, the ACK is delayed for a constant amount of time. If a *reactive* ACK (an ACK in response to data arrival) is triggered within the aforementioned constant amount of time, the proactive ACK can be discarded. This allows for curtailing the number of such proactive ACKs sent when there are reactive ACKs sent naturally.

4) ***Burst control:*** Classical TCP is self-clocked. Hence, whether or not new segments are transmitted and how many new segments are transmitted are both determined by the receipt of ACKs at the sender and the consequent adjustment to the windows. In a congestion control dominated regime, such self-clocking works very well. However, in a flow control dominated regime, large transmission bursts can occur. Consider a scenario where the application read rate is low and hence the buffer begins to fill up. Let the connection reach a state where the sender has only one outstanding segment left in the network because its flow control window is reduced, but its congestion control window is much larger. Now, if the application read rate rapidly increases and drains the receive buffer *before the outstanding segment reaches the receiver*, the ACK sent on receipt of the new segment will advertise a full buffer. When the sender receives this ACK it is no longer flow control limited, and *will transmit an entire congestion control window of segments*[5] instantaneously as a single burst. Such bursty behavior is not desirable as the bursts will increase the likelihood of overflows of buffers along the path of the connection. The overflows will be interpreted as congestion losses and hence impact the throughput performance of the connection adversely.

Thus, one of the design elements in AFC is to explicitly control any bursts in transmissions at the sender. The occurrence of a burst is detected by the difference in the allowed range of outstanding packets, which is oldest unacknowledged packet snd_una plus $min(cwnd, rwnd)$, and the next packet

[4]Note that the TCP ACK sequence number reflects the next expected sequence number.

[5]Assuming the congestion control window is smaller than the receive buffer size. Otherwise, the sender will transmit an entire flow control window of segments.

to be sent(snd_nxt). If this difference is above a threshold, every packet is delayed by $RTT/sender's\ window$.

B. AFC Solution Details

1) Protocol headers: AFC introduces new feedback from the data receiver to sender. At the same time, an AFC enabled network stack must be able to communicate with a default stack. Thus, we propose AFC specific information to be exchanged using a new TCP header option. At the time of connection set-up, an AFC enabled receiver will advertise an AFC-permitted flag in a 2 byte option field[6]. If both ends of the connection agree to use AFC as the flow control mechanism, another variable length option field is used to convey the application read rate to the sender. The first two octets convey the type and length of the option, the later octets carry the application read rate in Kbps.

2) AFC Receiver (Data) Processing: A data packet delivered by the network at the receiver can encounter three actions; enqueued in the receive buffer for the application, dropped by the receiver, or delivered instantly to a waiting application. For a newly arrived data packet with sequence number $seqno$, the receiver checks if it falls within the $window$ of admissible sequence numbers beyond the oldest buffered packet $read\_nxt$. If not, it is dropped. For a packet lying within the window, the receiver checks if it is the next expected in-order packet rcv_nxt and advances rcv_nxt if it is. In case the sequence number is $> rcv\_nxt$, the max_seen count is manipulated, depending on where $seqno$ lies. If any of this data is being waited upon by the application, it is passed on to the application, and $read\_nxt$ is advanced. The remaining data, both in-order and out-of-order, is queued at the receive buffer. As this is an interface between the TCP receiver and the application, AFC takes a sample of the application read rate by invoking the ar_update module. The ar_update module computes the instantaneous application read rate from the bytes read in this instance and time elapsed since last sample. It then computes an exponential moving average of samples seen so far.

The TCP receiver is also responsible for sending ACKs for every segment *delivered* to it, even if it is dropped. It computes the receiver window, i.e., the number of octets beyond rcv_nxt that the receive buffer can accept. This value of the receive window, rcv_nxt, SACK[8] information and the smoothed average of the application read rate $smooth\_rx$ is fed back to the sender through the ACK packet.

Furthermore, a sample of the application read rate is also taken whenever the application tries to independently read data from the buffer. The $read\_nxt$ is updated as application reads bytes from the $buffer$. Once it is done reading, the window size is updated and ar_update is invoked to compute a new value of $smooth\_rx$. A proactive acknowledgement is triggered if the new $smooth\_rx$ is greater/lesser than a factor times the last value $last\_rx$.

[6]One byte for the type of option and one for the length.

3) AFC Sender (ACK) Processing: To enable AFC at a TCP sender, new logic is introduced in processing the acknowledgement. The TCP sender determines the adaptive flow window from the advertised window win and application reading rate rx. It further distinguishes buffer losses from congestion losses, by tracking zero window event through a flag zw_flag. While zero windows are being received at the sender, all congestion indicators are suppressed and zero window probes are sent with increasing time-periods. Once an open window advertisement is received the time is recorded in $ts\_recover$ to ignore congestion indications for out-of-window packets that were dropped. Moreover, to recover from the losses after an open window is received for sequence number $open\_seq$, the snd_nxt is set to $open\_seq$. The retransmit timeout is also reset. If permitted by the sending window and AFC burst control, the sender can now send more data to the receiver.

V. Performance

A. Evaluation methodology

We evaluate our solution in NS2 (version 2.34). We use the NS2 TCP implementation, with classic flow control, as the default TCP in all experiments. Further, we added the design principles described in section IV in NS2 TCP implementation. This Adaptive Flow Control(AFC) enabled TCP is referred to as AFC in future. We assume SACK [8] to be enabled in all scenarios. The history factor for exponential moving average in AFC is taken as 0.5, i.e. equal weight is accorded to the history and the current sample. In the following sections, we evaluate AFC with respect to default TCP. We compare the throughput gains of each; fairness of both approaches in concurrent connections and sensitivity of our solutions to different parameters. In all experiments, the throughput is measured at the application level.

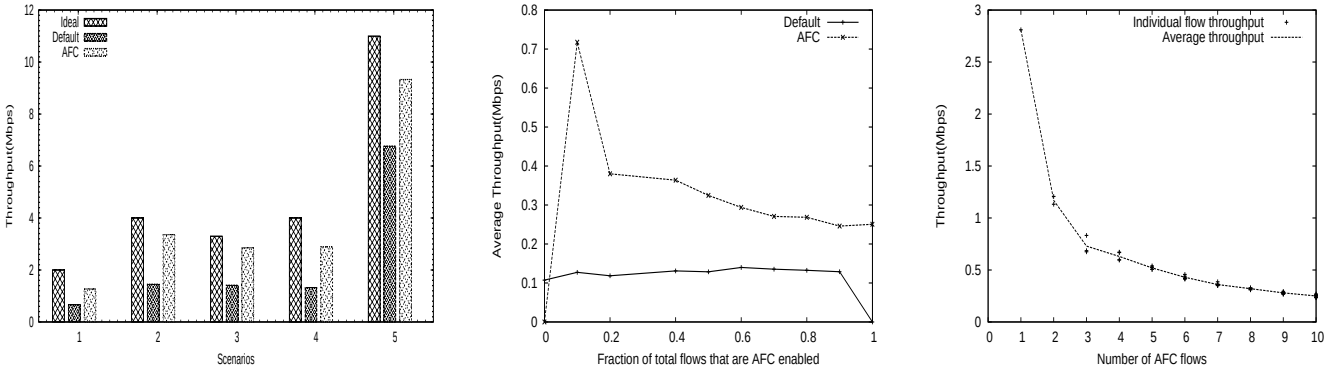
B. Throughput Gain

For throughput analysis, we consider the scenarios mentioned in table 3(a), for RTT = 530ms. Present auto-tuning techniques [3] configure the receive buffer based on the perceived bandwidth-delay product, which is minimum(average network rate, average application rate)*RTT. We use this estimate in configuring the receive buffer size. The ideal TCP throughput in all scenarios is min(average network rate, average application rate). Each simulation runs for 600 seconds.

Figure 2(a) shows the ideal, default and optimized throughput in all scenarios. We observe that AFC shows an improvement ranging from 50%, in $Scenario\ 5$, to 100% and more in the remaining scenarios. In addition to this, it scales up to 85% of the ideal throughput, while the default flow control can only achieve up to 60% of the ideal performance.

C. Fairness Properties

To evaluate fairness between concurrent optimized and unoptimized connections we use a dumbbell topology with 10 TCP connections. Senders $S_1....S_{10}$ are connected to router Rt_1 through individual links of 10Mbps rate and 5ms delay. Router Rt_1 is connected to another router Rt_2 with a network



(a) Throughput (b) Fairness between AFC and Default TCP (c) Fairness amongst AFC flows

Fig. 2. Throughput gains and Fairness analysis of AFC

link of delay 255ms. The bandwidth of this link fluctuates in the pattern of $< 2Mbps, 4Mbps, 4Mbps >$ with a time-period of 1 RTT, i.e. 530ms. All the receivers $R_1....R_{10}$ are connected to router Rt_2 through individual links of 10Mbps rate and 5ms delay. Each receiver has an application running on it whose read rate fluctuates as $< 0, 6, 6 >$ Mbps with a time period of 1RTT. Considering fair distribution of link bandwidth, each connection gets an average network rate of 0.33Mbps. The receive buffers are thus set to 0.33Mbps*530ms = 22KB. Each connection in the simulation runs for 600 seconds.

1) Fairness between AFC and Default Flows: We evaluate fairness of AFC towards classic flow control by increasing the number of optimized connections from 0 to 10, i.e., all connections using default flow control to all connections using AFC. In each case, we calculate the average throughput achieved by connections running default TCP and that achieved by connections using AFC. The results are shown in Figure 2(b). We observe that the average throughput of default TCP connections stays unchanged in the presence of Adaptive Flow Control. The average throughput of the AFC enabled flows shows a peak when there is one optimized connection and converges to the expected 0.33Mbps as the flows increase. This happens because an optimized flow tries to scale up to the available bandwidth, left unused by the default TCP flows. In the case of one optimized flow, all this bandwidth gets utilized by a single connection and is fairly shared, later on, by the increasing number of optimized connections. Thus,*AFC remains fair with classical flow control.*

2) Fairness amongst AFC Flows: To demonstrate fairness amongst AFC flows we use the same dumbbell topology as above. However, this time we present results for increasing number of TCP connections. All the TCP connections use AFC as the flow control mechanism. The receive buffer size is adjusted down based on the number of connections (from 213KB for one connection to 22KB for ten connections). The average throughput enjoyed by connections is shown in Figure 2(c). For each data point we also show the individual connection throughputs. It can be observed that the individual throughputs are heavily clustered around the average establishing the fairness amongst AFC flows.*Thus, AFC is fair with itself.*

D. Sensitivity Analysis

In this section we discuss how Adaptive Flow Control reacts to variations in the time period of fluctuation and the application fluctuation profile. We also present the performance of default TCP flow control.

1) Sensitivity to Fluctuation Period: Note that in all the scenarios discussed above we have considered that the application and the network always fluctuate with a period of 1 RTT. However, the adverse affect of flow control is not tied to this unique case. We run further simulations where the fluctuation period is increased from 1 RTT to 40 RTTs for $Scenario\ 4$. As this scenario is application rate dominated we also consider a modified version of $Scenario\ 4$ with peak application reading rate of 8Mbps to simulate a network limited scenario. The throughput of default flow control and adaptive flow control are compared in Figure 3(b).

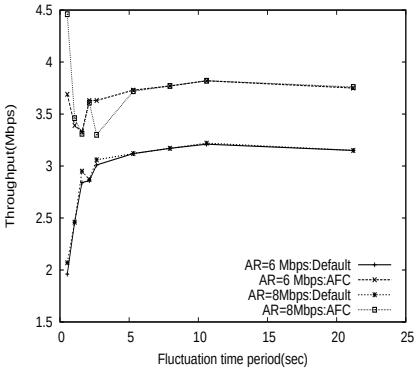
The throughput achieved by default flow control increases with fluctuation time-period as TCP gets more time to settle after every disturbance, making the connection more steady. The throughput observed by AFC shows an immediate dip when fluctuation time period increases from 1 RTT to 2 RTTs. This is because, while in former case AFC can avoid the sender from stalling completely, in the later cases, sender stalls are inevitable. Even then, AFC constantly performs better than default flow control.

AFC provides a gain of 100% over default flow control in highly fluctuating network and application environments and 20% in steady environments. Mobile phone environments, as we have observed in previous sections, belong to the former set.

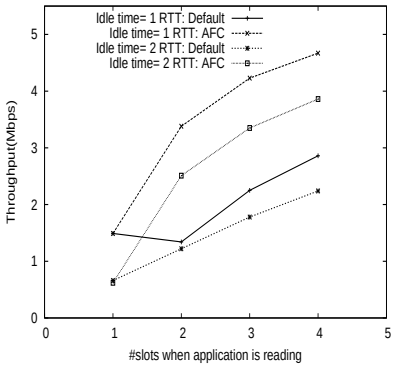
2) Sensitivity to Fluctuation-pattern: We now evaluate the performance of default flow control and AFC for other fluctuation patterns of application read rate. We consider repeated fluctuations throughout the connection. Each period of 1RTT is considered as a slot and we vary the number of consecutive slots for which the application is reading at AR and 0. The network rate is constant and greater than the average application read rate, for simplicity.

#	Application profile (Mbps)	Network profile (Mbps)	Receive buffer (KB)
1	$(0, 6, 6)$	2	128
2	$(0, 6, 6)$	15	256
3	$(0, 6, 6)$	$(2, 4, 4)$	213
4	$(0, 6, 6)$	$(3, 6, 6)$	256
5	$(0, 18, 18)$	$(3, 15, 15)$	704

(a) Network and application scenarios



(b) Fluctuation period



(c) Application fluctuation pattern

Fig. 3. Scenario description and sensitivity analysis of AFC

From the application profile of $< 0, 6, 6 >$ Mbps that we have considered so far, we create two sets of scenarios: application idle for 1 slot per fluctuation and application idle for 2 slots per fluctuation. In each of these sets, we further vary the number of reading slots of application from 1 to 4. All in all, there are 8 scenarios. The network rate is 15Mbps and the RTT is 530ms. The results are shown in figure 3(c).

The aggregate throughput intuitively decreases with increase in idle slots and increases with increase in reading slots. A pathological scenario arises when the application reads for exactly one slot before becoming idle. This is because TCP has an inherent delay of half RTT. Even with AFC, the sender learns about the increased receiving rate half an RTT late. By the time new data reaches the receiver, it has gone idle. Thus, in every 2 slots, the receiver can successfully accommodate exactly one buffer size of data. The throughput is thus buffer limited and same for both default and optimized cases. In other scenarios, AFC is able to improve throughput by at least 63% in all scenarios up to a maximum of 150%. We also observe that with increase in number of reading slots per fluctuation, the difference in the throughput of classic flow control and AFC starts to reduce. This is expected behavior, as increasing number of reading slots indicate a steadier network/application environment. *Thus, for a variety of application fluctuation patterns, AFC provides significant gain(more than 60%) over classic flow control.*

VI. Related Work

Several variants of TCP flow control have been proposed in related work. Automatic Buffer Tuning [4] presents an algorithm to dynamically configure TCP sender buffer. Dynamic Right Sizing [2] and Auto-tuning in Linux [3] implement receiver side solutions to grow the window sizes to match the available bandwidth. The Wed100 [9] project has presented approaches to decouple the re-assembly queue and the receive buffer, to hide out-of-order delays from the sender. All these approaches advocate a buffer-based approach to resolve flow control incompetencies. But they all rely on *perceived* BDP for their estimation, which, as we demonstrate, can be affected by flow control problems. AFC addresses these issues, without over-provisioning the buffer, by re-defining the very concept of flow control window.

VII. Conclusions

In this paper we show that classical TCP flow control performs poorly for flow control bottlenecked connections such as those terminating on mobile phones. We identify the reasons for the poor performance and propose an alternative flow control strategy called adaptive flow control (AFC). We have established through simulations that AFC can considerably improve throughput performance. We plan to explore several extensions to AFC as part of future work including the curtailment of unnecessary retransmissions when recovering from zero window advertisements and handling interplay between congestion control and flow control for connections that are bottlenecked intermittently by both.

References

[1] I. S. Institute, "RFC 793," 1981. [Online]. Available: http://rfc.sunsite.dk/rfc/rfc793.html

[2] E. Weigle and W. chun Feng, "Dynamic right-sizing: A simulation study," in *IEEE International Conference on Computer, Communication and Networking*, 2001.

[3] "Linux auto tuning." [Online]. Available: http://www.kernel.org/

[4] J. Semke, J. Mahdavi, and M. Mathis, "Automatic TCP buffer tuning," *Computer Communication Review*, 1998.

[5] S. Sanadhya and R. Sivakumar, "AFC: Technical report," 2010. [Online]. Available: http://www.ece.gatech.edu/research/GNAN/archive/tr-afc.pdf

[6] G. F. Franklin, D. J. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Prentice Hall PTR, 2001.

[7] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Prentice–Hall, 1975.

[8] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "RFC 2018," 1996. [Online]. Available: http://www.faqs.org/rfcs/rfc2018.html

[9] J. Heffner, "High bandwidth TCP queuing." [Online]. Available: $http://www.psc.edu/\ jheffner/papers/senior\_thesis.pdf$