Wireless Netw (2014) 20:2063–2080
DOI 10.1007/s11276-014-0727-9

Rethinking TCP flow control for smartphones and tablets

Shruti Sanadhya · Raghupathy Sivakumar

Published online: 6 May 2014

Abstract The focus of this work is to study the efficacy of TCP's flow control algorithm on mobile devices. Specifically, we identify the design limitations of the algorithm when operating in environments, such as smartphones and tablets, 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 *NS*2 simulations, that AFC can provide considerable performance benefits over classical TCP flow control.

Keywords TCP · Flow control · Smartphones · Tablets

1 Introduction

The flow control mechanism in classical TCP is simple. The receiver piggybacks on every ACK the available space in the receive buffer, and the sender never allows the number of outstanding packets to grow beyond the available buffer space. While the conservative strategy ensures that there is no overflow of data at the receive buffer, it does not directly track the application behavior at the receiver. 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* platforms (phones and tablets) 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 devices*,[1] *and propose a new flow control algorithm for such platforms.*. In this context, using a Samsung Galaxy S 4G phone on the T-mobile data network and Samsung Galaxy Tab 10.1 as representative mobile devices, we first show that the available processing power for a given TCP connection can fluctuate drastically even for simple user workloads, and such fluctuations invariably lead to the flow control algorithm dominating transmission decisions at the sender.

We then explore how a TCP connection in a flow control dominated regime performs using several example scenarios. We observe that the throughput performance of such a 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.

S. Sanadhya (✉) · R. Sivakumar
Georgia Institute of Technology, Atlanta, GA, USA
e-mail: shruti.sanadhya@cc.gatech.edu

R. Sivakumar
e-mail: siva@ece.gatech.edu

[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 and tablets.

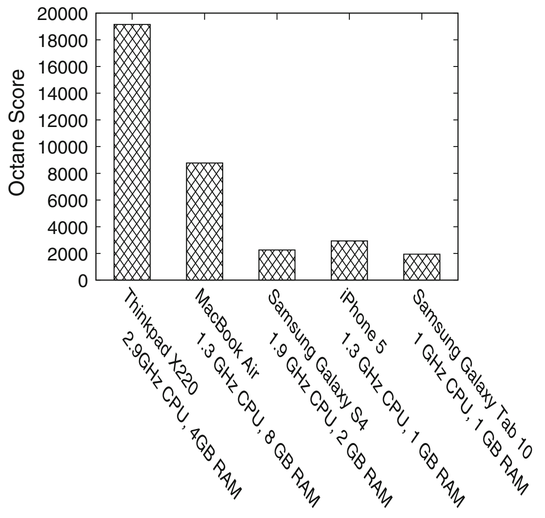


Fig. 1 Comparison of Javascript benchmark score across laptops, smartphones and tablets

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 network stacks with AFC enabled are still backward compatible to communicate with non AFC-enabled stacks. We evaluate AFC using *NS*2 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.

The rest of the paper is organized as follows: In Sect. 2 we discuss the different limitations of flow control in classical TCP and also show why a simple buffer provisioning solution is not desirable. In Sect. 3 we perform a control theoretic analysis of TCP and motivate the core design rationale for AFC. In Sect. 4 we present the solution details for AFC and in Sect. 5 we evaluate the performance of AFC. Finally, in Sect. 7 we describe related work and conclude in Sect. 8.

2 Background and motivation

2.1 Resource constraints on mobile devices

Even though smartphones and tablets have been growing in performance since their inception, these devices have not scaled up to the same performance as desktop and laptop computers. This is mainly because smartphones and tablets have to offer portability as the primary feature. Excess compute power comes at the cost of size, weight and battery life. To further motivate this gap in compute power on mobile devices and computers, we run a JavaScript benchmark, Octane [1], on the following devices:

- Laptop1: Lenovo Thinkpad X220 running Ubuntu 12.04 with 2.9 GHz Intel I7 processor and 4 GB RAM
- Laptop2: Apple MacBook Air running OS 10 with 1.3 GHz Haswell I5 processor and 8GB RAM
- Smartphone1: Samsung Galaxy S4 running Android 4.2.2 with 1.9 GHz quad-core Krait processor and 2 GB RAM
- Smartphone2: iPhone 5 running iOS 7 with dual-core 1.3 GHz Swift processor and 1GB RAM
- Tablet: Samsung Galaxy Tab 10 with dual-core 1 GHz Cortex-A9 processor and 1GB RAM

Octane is Google's benchmark suite to measure the performance of browser's JavaScript engine over 13 tests. The tests create representative workloads for the browser, such as regular expression matching, function calls, polymorphism, object creation/deletion, pdf reading, floating point math, etc. The test suite computes a score for each of the 13 tests and a combined score. A high score means high performance. Figure 1 shows Octane results for the five devices. We observe that the performance on laptop is an order of magnitude better than that on smartphones and tablets. It is particularly interesting to note that Apple MacBook Air with 1.3 GHz processor performs 3× better than iPhone 5 with a similar processor speed and 4× better than Samsung Galaxy S4 which has a 'faster' processor. These results show that even with significant technical advances in compute power, smartphones and tablets do not perform same as traditional desktops and laptops.

2.2 TCP flow control basics

TCP's flow control algorithm provides the receiver with the ability to control the rate at which the sender transmits [2]. Thus, if the data consumption rate at the receiver is lower than the rate at which the sender is transmitting, the receiver is able to influence the sending rate down to an appropriate level. While we discuss some variants later in the paper, the basic 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. 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.

Thus, if the available network rate is the bottleneck, *cwnd* is likely to be smaller than the *rwnd* and flow control does not influence the data rate of the TCP connection. On the other hand, 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. The sender also independently sends periodic one-octet *probes* when it is in the frozen zero window state hoping to elicit an open window from the receiver. This handles any reliability issues associated with open window losses.

Thus, some of the highlights of the flow control algorithm are as follows:

- Buffer occupancy: TCP's flow control is heavily buffer dependent. The sender will never allow the number of unacknowledged packets to grow larger than the receiver's buffer size. This property holds independent of whether such outstanding packets have in fact been drained out of the receive buffer as long as the acknowledgements for those packets have not reached the sender.
- Application read rate: The buffer occupancy in turn is heavily influenced by the application read rate at the receiver. The TCP receive buffer has no other influencers other than the input rate and the drain rate, as we discuss later in the section.
- Feedback latency: Since the sender explicitly relies on feedback from the receiver to adjust its notion of the receive buffer occupancy, the feedback latency for the flow control process is directly influenced by the round-trip time for the connection.

2.3 Problems with TCP flow control on mobile devices

2.3.1 Flow control bottlenecks occur more often

Mobile devices such as smartphones and tablets, 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 constraints and cost. Fig. 2(a)–(c) present comparative CPU allocation results for an FTP application running on a laptop (Dell Inspiron 1,525 with Ubuntu 10.10), a mobile phone (Samsung Galaxy S 4G with Android OS) and a tablet (Samsung Galaxy Tab 10.1 with Android OS) respectively. In all three cases, a large file ($\sim$2 GB) is downloaded over WiFi from an Internet server down to the client. To ensure that network is not the bottleneck, we choose high capacity channels supported by each device, i.e. 2.4 GHz 802.11 g channel on smartphone and 5 GHz 802.11a channel for tablet. We run the experiment on the laptop with both channel settings, but only present the 2.4 GHz result here for brevity. As each file download progresses, three workloads; email, web browsing and progressive video download—are introduced. The impact on the CPU allocation for the FTP process is measured using the *top* utility.

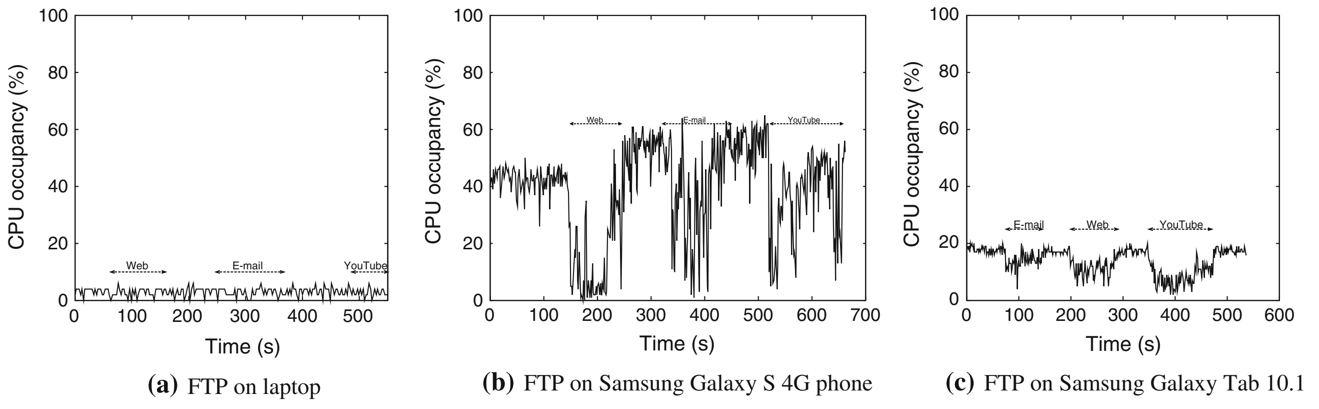
We observe that on the laptop the FTP client is relatively unaffected by the background processes and remains at around 5 % CPU allocation. However, for the FTP client on the mobile phone, the CPU occupancy fluctuates between 60 and 0 % during the download. The performance on tablet is closer to the mobile phone, the CPU occupancy fluctuates between 20 and 5 %. It is interesting to note that the tablet has a dual core processor but still the FTP application and the background workloads shared the same core leading to the observed fluctuations.

Investigating the individual FTP connections further, we observe that the instantaneous throughput degrades from 10 to 50 % on both the mobile devices in the presence of background workload while no such degradation is observed on the laptop. The individual results are shown in Fig. 3(a)–(c). In addition to this, there are no zero window events on the laptop and tablet but 5 zero window events are observed on the mobile. Note that the overall throughput on tablet is higher due to the higher capacity of the 802.11a channel. The throughput on laptop on 802.11a channel (not shown here) is also comparable.

The above result highlights the vulnerability of TCP connections on mobile platforms to fluctuations in processor allocations. These fluctuations in turn *impact the degree to which flow control influences the performance of the connections*. We study this impact next.

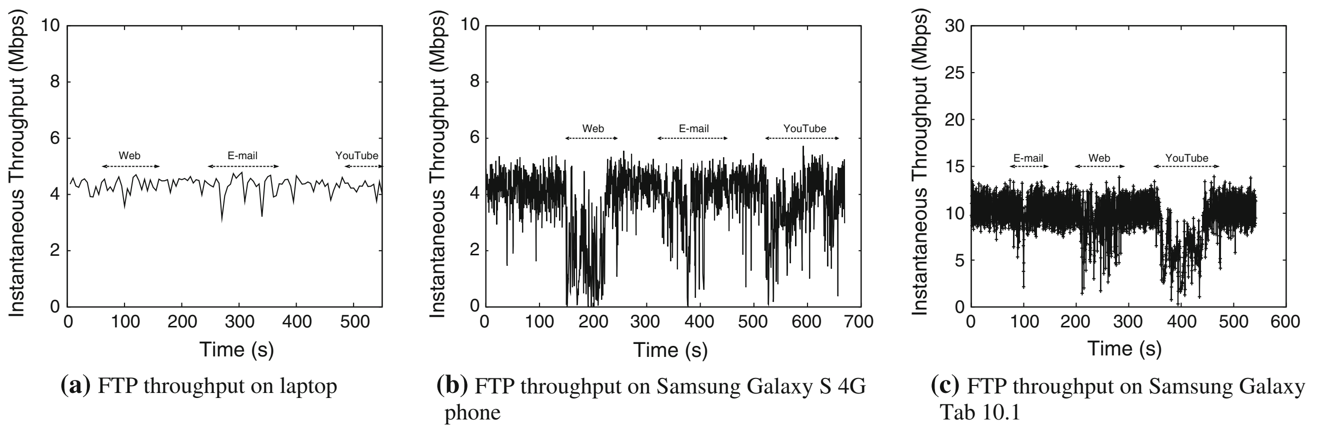
2.3.2 TCP flow control is inefficient

As discussed earlier, fluctuations in processing power allocated to an application directly impact the rate at which the application interacts with TCP, i.e. the rate at which it reads from the receive buffer. While TCP flow control is expected to converge to a throughput of *min*(network rate, application read rate), this turns out to be true only when



(a) FTP on laptop **(b)** FTP on Samsung Galaxy S 4G phone **(c)** FTP on Samsung Galaxy Tab 10.1

Fig. 2 Comparison of CPU occupancy of FTP connection on laptop and mobile devices



(a) FTP throughput on laptop **(b)** FTP throughput on Samsung Galaxy S 4G phone **(c)** FTP throughput on Samsung Galaxy Tab 10.1

Fig. 3 Comparison of instantaneous TCP throughput of FTP connection on laptop and mobile devices

both the network and application rates are steady. Fluctuations in the application read rate make it difficult for TCP to converge as expected.

To demonstrate this, we conduct simulations in NS2 with the following setup: (a) sender and receiver connected over a direct link; (b) RTT of 530 ms; (c) network rate of 15 Mbps; (d) average application read rate of 4 Mbps, with a fluctuation profile of $\langle 0, 6, 6 \rangle$ (period of 1 RTT); and (e) receive buffer size equal to the perceived bandwidth delay product (BDP) (*min*(network rate, average application rate) $\times$ RTT $=$ 256 KB). While we pick these values as an example (e.g. TCP connection over a WiFi last leg for an inter-continental 'USA/Aus' communication), we generalize the values for the parameters in the setup to a broader set both later in the section and in Sect. 5.

The observed throughput should ideally be equal to the minimum of the network and application read rates, which for the above setup is equal to 4 Mbps. *However, the aggregate throughput observed is only 1.45 Mbps, a degradation of 63 %* (Fig. 4). Note that given the high network rate assumed, there are no congestion artefacts influencing

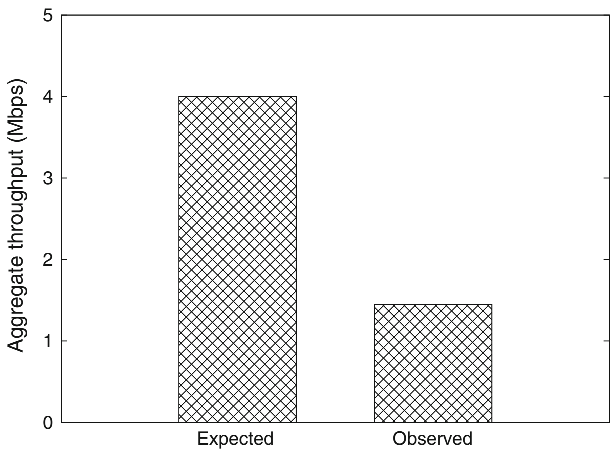


Fig. 4 Impact of application read rate fluctuations on TCP throughput

the performance, and hence this degradation is directly due to the flow control behavior of TCP.

There are several microscopic reasons for why this degradation in performance is attributable to the flow control behavior of TCP. We discuss these next.

2.4 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 530 ms. The read rate of the receiving application (AR) fluctuates in a pattern of $\langle$AR1, AR2 $\rangle$ or $\langle$0, AR, AR$\rangle$ with a time period of 1 RTT. If the pattern is $\langle$AR1, AR2$\rangle$, the application reads at AR1 for one RTT, then at AR2 for another RTT and back to AR1. If its $\langle$0, AR, AR$\rangle$, 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. In some scenarios, the network rate (NW) is also made to fluctuate in a pattern of $\langle$NW1, NW2, NW2$\rangle$ with a time-period of 1 RTT, i.e. the link bandwidth stays at NW1 for one RTT, then at NW2 for two RTTs and back to NW1, and so on. The scenarios we consider are the following.

2.4.1 *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 = 1 s; (b) Application profile: $\langle$2, 6$\rangle$ 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) $\times$ RTT = 500 KB = 4 Mb (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 ($\sim$3 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 4 Mbps. If the application is reading at 2 Mbps, every half RTT 1 Mb of data would be read by the application and 1 Mb stored in the buffer. At the end of the first half RTT, the advertised window is 3 Mb. At the end of 1 RTT, the application would have read another 1 Mb and stored 1 Mb in the buffer, the advertised window reduces to 2 Mb. In the next half RTT, the application reads at the rate of 6 Mbps, it reads the 2 Mb stored data in the buffer and also the 1 Mb received from the sender, which is (3 Mb (advertised window an RTT back) − 2 Mb (outstanding data)). The latest advertised window is now 4 Mb. In the next half RTT, the receiver receives another 1 Mb, which is 2 Mb (the advertised window an RTT back)−1 Mb (traffic outstanding in the last RTT). The receiving application reads the entire received 1 Mb and advertises a window of 4 Mb. The same sequence repeats from there on.

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

2.4.2 *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 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 = 530 ms; (b) Application profile of $\langle$0, 6, 6$\rangle$ (AAR = 4 Mbps); (c) NW = 15 Mbps; 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 63 % degradation), as shown in Fig. 4. 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 $\times$ 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 1,132 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

[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.

degradation (difference between the expected throughput and the observed throughput) increases.

2.4.3 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 [3], auto-tuning [4], etc. We use the following parameters for this scenario: (a) RTT = 530 ms; (b) Application profile: $\langle 0, 6, 6\rangle$ Mbps with the fluctuation interval = 1 RTT, AAR=4 Mbps; (c) Network profile: $\langle 2, 4, 4\rangle$ Mbps with the fluctuation interval = 1 RTT; and (d) buffer B set to 128/213 KB (perceived/ideal BDP).

In this scenario, the application is expected to enjoy a throughput of *min*(average network rate, average application rate), i.e. min (3.3, 4 Mbps). However, to achieve that performance, the receiver needs to make sure that the receive buffer is tuned to the network. Current buffer resizing solutions [3–5] 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 $N_p \sim 2$ Mbps is much lesser than the actual network rate $N_a = (2+4+4)/3 = 3.3$ Mbps. The observed throughput with a buffer size of 2 Mbps $\times$ 530 ms = 128 KB, is 0.67 Mbps, which is 20 % of the expected ideal. Even when the buffer is scaled up to 213 KB, i.e. 3.3 Mbps $\times$ 530 ms, the observed throughput is still only 1.45 Mbps.

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.

2.5 Trivial buffer-based approach

We now briefly argue for why a buffer provisioning based solution is not desirable to tackle the problems discussed thus far. We consider three categories of scenarios, as described in Table 1, in increasing order of complexity, and discuss requirements in a pure buffer provisioning solution. When necessary, we use NS2 based simulations to verify our arguments.

- *No application read-rate or network rate fluctuations:* This scenario is relatively well explored and the recommended buffer allocation when the application read-rate is greater than the network rate is as follows:

$$B_{req} = NR \times RTT \quad (1)$$

where, *NR* is the network rate and *RTT* is the round-trip time of the connection. However, if the application read-rate *AR* is less than the network rate and hence is the bottleneck, the buffer required is only proportional to the application read-rate. Hence, the buffer requirement under steady rates is as follows:

$$B_{req} = min(NR, AR) \times RTT \quad (2)$$

- *Only application read-rate fluctuations:* When the application read-rate fluctuates, the consequent zero-windows that occur will end up causing the connection to under-utilize the achievable performance. Specifically, consider *Scenario* 2 from Table 1. Assuming a buffer size based on Eq. (2) of 256 KB, the expected throughput is 4 Mbps ($min(NR, AAR)$), where *AAR* is the average application rate. However, the observed performance in the simulation study for the above parameters is only 1.45 Mbps. This degradation is directly explainable by the fact that two out of every three RTTs the application stays idle. Note that the performance observed is higher than the 1.33 Mbps based on the above argument as zero windows are not triggered precisely every third RTT. A straightforward solution to the above problem is to *provision the buffer such that the application does not find the buffer to be empty during the two RTTs recovering from a zero-window*. Hence, the buffer requirement can be arrived at as follows:

$$B_{req} = 3 \times AAR \times RTT \quad (3)$$

We do verify in simulations that the above buffer allocation increases the observed throughput to 3.86 Mbps. Now, the above scenario consisted of the *AAR* being less than the *NR*. If on the other hand the *AAR* is greater than the *NR*, the two idle RTTs can be fully utilized as long as buffer provisioning sustains the network rate. Hence, modifying Eq. (3), we get the following:

$$B_{req} = 3 \times min(AAR, NR) \times RTT \quad (4)$$

- *Both application read-rate and network rate fluctuations:* Finally, if both the network rate and application read-rate fluctuate, the scenario differs even further. Specifically, when both rates fluctuate, it is possible to

Table 1 Network and application scenarios

#	Application profile (Mbps)	Network profile (Mbps)	Fluctuation time	Round trip time (ms)	Receive buffer (KB)	Ideal throughput (Mbps)
1	⟨0,6,6⟩	2	per RTT	530	128	2
2	⟨0,6,6⟩	15	per RTT	530	256	4
3	⟨0,6,6⟩	⟨2,4,4⟩	per RTT	530	213	3.3
4	⟨0,6,6⟩	⟨3,6,6⟩	per RTT	530	256	4
5	⟨0,18,18⟩	⟨3,15,15⟩	per RTT	530	704	11

create a pathological scenario wherein the connection does not realize the higher network rate possible *because it is idle due to recovery from zero-windows when the network rate is high*. For example, consider *Scenario* 5, where the application rate fluctuates as $\langle 0, 18, 18 \rangle$ (period of one *RTT*), and the network rate fluctuates as $\langle 3, 15, 15 \rangle$ (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 15 Mbps was possible during that period. In our simulation study of the above scenario, we observe a throughput of 3 Mbps in contrast to the expected throughput of 11 Mbps. 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:

$$B_{req} = 3 \times AAR \times RTT \tag{5}$$

Taking into account Eqs. (2)–(5), the buffer required in a pure provisioning based strategy to cover all scenarios is $3 \times AAR \times RTT$. 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 a basic android phone, we were able to observe application read-rates as high as 100 Mbps (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 2 Mbps network rate scenario will ideally need only 125 KB of buffer allocation, whereas the provisioning based strategy will necessitate 18.75 MB 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 devices. Furthermore, even if such allocation can be achieved on the mobile devices, 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?

3 Theoretical analysis

3.1 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:

$$W = B_0 - B \tag{6}$$

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

$$dB/dt = TCP - AR \tag{7}$$

Differentiating (6) and using (7), we get

$$W' = dW/dt = AR - TCP \tag{8}$$

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

$$W = min\left(B_0, \int W'dt\right) \tag{9}$$

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:

$$W' = (AR - TCP) = err \tag{10}$$

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.

$$TCP = \alpha\ W \text{where} \alpha = 1/RTT \tag{11}$$

$$\text{using (9)},\ TCP = \alpha\ min\left(B_0, \int W'\ dt\right) \tag{12}$$

$$\text{using (10)},\ TCP = \alpha\ min\left(B_0, \int err\ dt\right) \tag{13}$$

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 Eqs. (10) and (11):

$$W' = AR - \alpha W \tag{14}$$

$$\text{on reorganizing},\ W' + \alpha W = AR \tag{15}$$

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:

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

multiplying (15) with integrating factor

$$e^{\alpha t}W' + \alpha e^{\alpha t}W = e^{\alpha t}AR \tag{16}$$

$$\text{on simplifying},\ \frac{d}{dt}(e^{\alpha t}W) = e^{\alpha t}AR \tag{17}$$

$$\text{on integrating},\ \int_{t=0}^{t} \frac{d}{dt}(e^{\alpha t}W) = \int_{t=0}^{t} (e^{\alpha t}AR)dt \tag{18}$$

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]

$$AR = A_0(1 + \sin \omega t), \text{where } \omega = 2\pi/T \tag{19}$$

using (19) in (18) and simplifying,

$$e^{\alpha t}W - B_0 = A_0 \int_{t=0}^{t} e^{\alpha t}dt + A_0 \int_{t=0}^{t} e^{\alpha t} \sin \omega t dt \tag{20}$$

on solving,

$$W = e^{-\alpha t}\left[B_0 - \frac{A_0}{\alpha} + \frac{A_0 \sin\theta}{\sqrt{\alpha^2+\omega^2}}\right] + \frac{A_0}{\alpha} + A_0\frac{\sin(\omega t - \theta)}{\sqrt{\alpha^2+\omega^2}}, \text{where } \theta = \tan^{-1}\left(\frac{\omega}{\alpha}\right) \tag{21}$$

The error *err* in *TCP* can thus be computed from (10) as:

$$err = W' \tag{22}$$

differentiating (21) and using in (22)

$$err = -\alpha e^{-\alpha t}\left[B_0 - \frac{A_0}{\alpha} + \frac{A_0 \sin\theta}{\sqrt{\alpha^2+\omega^2}}\right] + \frac{A_0\omega}{\sqrt{\alpha^2+\omega^2}}\cos(\omega t - \theta) \tag{23}$$

In steady state: $e^{-\alpha t} \to 0$, thus (23) becomes

$$err = \frac{A_0\omega}{\sqrt{\alpha^2+\omega^2}}(\cos(\omega t - \theta)) \tag{24}$$

$$\text{further},\ err = A_0 \sin\theta(\cos(\omega t - \theta)) \tag{25}$$

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 (11) and (21), *TCP* is:

$$TCP = \alpha e^{-\alpha t}\left[B_0 - \frac{A_0}{\alpha} + \frac{A_0 \sin\theta}{\sqrt{\alpha^2+\omega^2}}\right] + A_0\left[1 + \frac{\alpha \sin(\omega t - \theta)}{\sqrt{\alpha^2+\omega^2}}\right] \tag{26}$$

which in steady state becomes:

$$TCP = A_0\left[1 + \frac{\alpha}{\sqrt{\alpha^2+\omega^2}}\sin(\omega t - \theta)\right] \tag{27}$$

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,

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

i.e. time taken to converge to AR, for TCP. Equation (27) 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 (13), the actual TCP data rate is given by:

$$TCP = min\left(\alpha B_0, A_0\left[1+\frac{\alpha}{\sqrt{\alpha^2+\omega^2}}\sin(\omega t-\theta)\right]\right) \quad (28)$$

Depending on the relation between the two terms in (28), 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.

3.2 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 Eq. (27)), 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 Eq. (28)).*

A corrective term needs to be added in Eq. (11) 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 (11) thus takes the form of:

$$TCP = \alpha W + AR \quad (29)$$

working out equation (10)

$$W' = AR - TCP \quad (30)$$

$$\text{using (29)},\ W' = AR - \alpha W - AR \quad (31)$$

$$\text{i.e. } W' = -\alpha W \quad (32)$$

$$\text{on solving } W = B_0 e^{-\alpha t} \quad (33)$$

$$\text{differentiating (32) and using in (22)},\ err = -\alpha B_0 e^{-\alpha t} \quad (34)$$

Note that (34) presents a decaying error in TCP send rate. From Eqs. (29) and (33), TCP takes the form:

$$TCP = \alpha B_0 e^{-\alpha t} + AR \quad (35)$$

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.

4 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 detailed algorithm.

4.1 Key design elements

4.1.1 Using application read rate

The first design element in AFC follows directly from the theoretical analysis presented in Sect. 3. 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:

$$W_{fc} = B + AR \times RTT \quad (36)$$

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.

4.1.2 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. In an alternate approach, the receiver can explicitly notify the sender of the specific sequence numbers that have been dropped at the buffer. However, conveying explicit information about buffer losses would require going from one sequence number to two sequence numbers (one for congestion control and one for reliability/flow-control) similar to strategies adopted by WTCP [8], pTCP [9]. However, such a strategy would help only in the specific scenario of overlapping flow-control and congestion-control dominated periods for the connection. The downside of our simpler approach is that we will not react to congestion if it occurs during a flow control recovery period. However, if the congestion is persistent, the TCP sender will recognize it as soon as it comes out of flow control. As part of future work, we are planning to explore whether a more sophisticated scheme is warranted.

4.1.3 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.

4.1.4 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

[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.

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

Algorithm 1 Data packet delivered at TCP receiver

```
Input: $data$ = Data packet received by TCP
$seqno$ = Sequence number of the first octet in the data
Variables: $max\_seen$ = Maximum sequence number seen by the receiver, even out-of-order
receive_data()
 1: $bytes\_read \leftarrow 0$
 2: if $seqno > read\_nxt + bufsize$ then
 3:     Drop packet
 4: else if $seqno < read\_nxt$ then
 5:     $is\_dup \leftarrow$ true
 6: else if $seqno > max\_seen$ then
 7:     mark all buffer spaces from $max\_seen$ to $seqno$ as null
 8:     $buffer[seqno \bmod bufsize] \leftarrow data$
 9:     $buffer[(seqno + 1) \bmod bufsize] \leftarrow$ null
10:     $just\_marked \leftarrow$ true
11:     $max\_seen \leftarrow seqno$
12: end if
13: if $read\_nxt \leq seqno \leq max\_seen$ then
14:     if not $just\_marked$ and $buffer[seqno \bmod bufsize] \neq$ null then
15:         $is\_dup \leftarrow$ true
16:     end if
17:     $buffer[seqno \bmod bufsize] \leftarrow data$
18:     while (Application needs the $read\_nxt$ byte and $buffer[read\_nxt \bmod bufsize] \neq$ null ) do
19:         Pass $buffer[read\_nxt \bmod bufsize]$ to application
20:         $read\_nxt \leftarrow read\_nxt + 1$
21:         $bytes\_read \leftarrow bytes\_read + 1$
22:     end while
23:     $rcv\_nxt \leftarrow read\_nxt$
24:     while ($buffer[rcv\_nxt \bmod bufsize] \neq$ null and $rcv\_nxt \leq max\_seen$) do
25:         $rcv\_nxt \leftarrow rcv\_nxt + 1$
26:     end while
27: end if
28: if $bytes\_read \neq 0$ or $rcv\_nxt - read\_nxt > 0$ then
29:     $ar\_update(bytes\_read, now())$
30: end if
31: $window \leftarrow bufsize - (rcv\_nxt - read\_nxt)$
32: if $is\_dup \neq$ true then
33:     Generate $ack$ with $rcv\_nxt$ and SACK information
34:     $ack.win \leftarrow window$
35:     $ack.rx \leftarrow smooth\_rx$
36:     Send ACK
37: end if
```

4.2 AFC solution details

4.2.1 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.

4.2.2 AFC receiver (data) processing

Algorithm (1) details the data processing at AFC enabled TCP receiver. Table 2 describes the variables used in AFC pseudocode. A data packet delivered by the network at the receiver can encounter three actions; (i) enqueued in the receive buffer for the application, (ii) dropped by the receiver, or (iii) delivered instantly to a waiting application. For a newly arrived data packet with sequence number *seqno*, the receiver checks if it falls within *bufsize* of admissible sequence numbers beyond the oldest buffered packet *read_nxt* and drops it if it doesn't (lines 2 and 3). For a fresh packet lying within the window, the receiver saves it in the buffer and updates the *max_seen* count (lines 6–17). Duplicate packets are completely ignored (lines 5 and 15). The receiver also checks if the application has been waiting for data. If yes, it passes new data to the application above and updates *read_nxt* and *bytes_read* values (lines 18–22). The *rcv_nxt* pointer is also updated to the next in-order byte not present in the receive buffer (lines 23–26). The remaining data, both in-order and out-of-order, is queued at the receive buffer.

Table 2 List of state variables at the TCP receiver

bytes_read	Count of bytes read by application in this instance
read_nxt	Next in-sequence byte to be read from buffer
bufsize	Total size of the TCP receive buffer
buffer	Receiver buffer
rcv_nxt	Next in-sequence byte expected from the network by the TCP receiver
window	Number of bytes, starting from *rcv_nxt*, the receive buffer can accommodate
smooth_rx	Exponential average of application read rate
last_rx	Last value of *smooth_rx*

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

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 (line 29). The *ar_update* module (algorithm (2)) 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 *smooth_rx* of samples seen so far.

Algorithm 2 Computing application read rate

```
Input: $num\_bytes$ = Bytes read by application since last sample
$read\_time$ = Time when function was called
Variables: $history\_factor \in [0, 1]$ is the weight given to the old application rate estimate while computing the new one.
$last\_read$ = Time when this procedure was last called
ar_update()
1: $t\_elapsed \leftarrow read\_time - last\_read$
2: if $t\_elapsed$ then
3:     $last\_rx \leftarrow smooth\_rx$
4:     $smooth\_rx \leftarrow history\_factor * smooth\_rx + (1 - history\_factor) * num\_bytes/t\_elapsed$
5: end if
6: $last\_read \leftarrow read\_time$
```

Algorithm 3 Data packet read from buffer by application

```
read_buffer()
1: $bytes\_read \leftarrow 0$
2: while Application can read $buffer[read\_nxt \bmod bufsize]$ do
3:     Pass $buffer[read\_nxt \bmod bufsize]$ to application
4:     $read\_nxt \leftarrow read\_nxt + 1$
5:     $bytes\_read \leftarrow bytes\_read + 1$
6: end while
7: $window \leftarrow bufsize - (rcv\_nxt - read\_nxt)$
8: $ar\_update(bytes\_read, now())$
9: if $smooth\_rx > factor * last\_rx \;||\; smooth\_rx < factor * last\_rx$ then
10:     Generate an ACK for application update: $ack$
11:     $ack.win \leftarrow window$
12:     $ack.rx \leftarrow smooth\_rx$
13:     Send ACK
14: end if
```

The TCP receiver is also responsible for sending ACKs for every new segment *delivered* to it, even if it is dropped. Algorithm (1) also captures this. It computes receiver's *window*, i.e. the number of octets beyond *rcv_nxt* that the receive buffer can accept (line 31). The value of *window*, *rcv_nxt*, SACK [10] information and *smooth_rx* is fed back to the sender through the ACK packet (lines 33–36).

Furthermore, a sample of the application read rate is also taken whenever the application tries to independently read data from the buffer, as illustrated in algorithm (3). The *read_nxt* is updated as application reads bytes from the *buffer* (lines 2–6). Once it is done reading, the window size is updated (line 7 and *ar_update* is invoked to compute a new value of *smooth_rx* (line 8). A proactive acknowledgement is triggered if the new *smooth_rx* is greater/lesser than a factor times the last value *last_rx* (lines 9–14).

Algorithm 4 ACK processing at the TCP sender

```
Input: $ack$ = Acknowledgement from receiver
Relevant fields:
$seqno$ = the next in-sequence byte expected at the receiver
$win$ = number of bytes, starting from $seqno$, that receive buffer can accommodate
$rx$ = application read rate as computed by the receiver
$ts\_echo$ = timestamp of the packet which triggered this ACK
Variables:
$rtt$ = Round trip time
$highest\_ack$ = Highest sequence number acknowledged
$snd\_nxt$ = Sequence number of the next byte to transmit
$zw\_flag$ = Flag to monitor start/stop of zero window event
receive_ack()
1: $awnd \leftarrow ack.win$
2: $app\_rate \leftarrow ack.rx$
3: if $rtt > 0$ then
4:     $rwnd \leftarrow awnd + app\_rate * rtt$
5: end if
6: if $awnd = 0$ and $zw\_flag = 0$ then
7:     $zw\_flag \leftarrow 1$
8: else if $awnd > 0$ and $zw\_flag = 1$ then
9:     $zw\_flag \leftarrow 0$
10:     if not fast recovery phase then
11:         $ts\_recover \leftarrow now()$ {Store the current time}
12:         if $ack.seqno > highest\_ack$ then
13:             $highest\_ack \leftarrow ack.seqno - 1$
14:         end if
15:         $snd\_nxt \leftarrow highest\_ack + 1$
16:         Process SACK information
17:         Reset the retransmit timer
18:     end if
19: end if
20: if $zw\_flag$ then
21:     Send zero window probes to learn about open windows
22: else
23:     if not fast recovery phase then
24:         if $ack.seqno > highest\_ack$ then
25:             Process the packet like a new ACK
26:         else if $ack.seqno = highest\_ack$ then
27:             Process SACK information
28:             if $ack.ts\_echo > ts\_recover$ then
29:                 Process duplicate ACK
30:             end if
31:         end if
32:     end if
33: end if
34: Send data, if allowed by $min(cwnd, rwnd)$ and burst control
```

4.2.3 AFC sender (ACK) processing

To enable AFC at a TCP sender, new logic is introduced in processing the acknowledgement, as shown in algorithm (4). The TCP sender determines the adaptive flow window from the advertised window *win* and application reading rate *rx* (lines 1–5). It further distinguishes buffer losses from congestion losses, by tracking zero window event through a flag *zw_flag* (lines 6–9).

While zero windows are being received at the sender zero window probes are sent with increasing time-periods (line 21). 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 (line 11). Moreover, to recover from the losses after an open window is received, *highest_ack* and *snd_nxt* values are updated and retransmit timeout is reset (lines 12–17). Congestion indicators are only considered when *zw_flag* is not set (lines 23–32). If permitted by the sending window and AFC burst control, the sender can now send more data to the receiver (line 34).

5 Performance

5.1 Evaluation methodology

We evaluate our solution in NS2 (version 2.34). We use the NS2 TCP implementation, with classic flow control,[7] as the default TCP in all experiments. Further, we added the design principles described in Sect. 4 in NS2 TCP implementation. This Adaptive Flow Control(AFC) enabled TCP is referred to as AFC in future. We assume SACK [10] 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.

For the throughput and sensitivity analysis the network topology has a single sender node and receiver node connected by a link. The link characteristics are based on typical bandwidths and delays observed on mobile phones and tablets connecting over 3G or WiFi. The link delay we use is 265 ms. For fairness analysis, we consider a dumbbell shaped topology defined later in Sect. 5.3

[7] Basic flow control features such as a 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.

Table 3 Network and application scenarios

#	Application profile (Mbps)	Network profile (Mbps)	Receive buffer (KB)	Ideal throughput (Mbps)
1	⟨0,6,6⟩	2	128	2
2	⟨0,6,6⟩	15	256	4
3	⟨0,6,6⟩	⟨2,4,4⟩	213	3.3
4	⟨0,6,6⟩	⟨3,6,6⟩	256	4
5	⟨0,18,18⟩	⟨3,15,15⟩	704	11

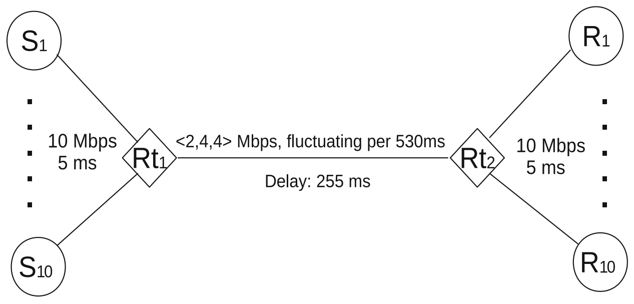


Fig. 5 Topology for fairness evaluation

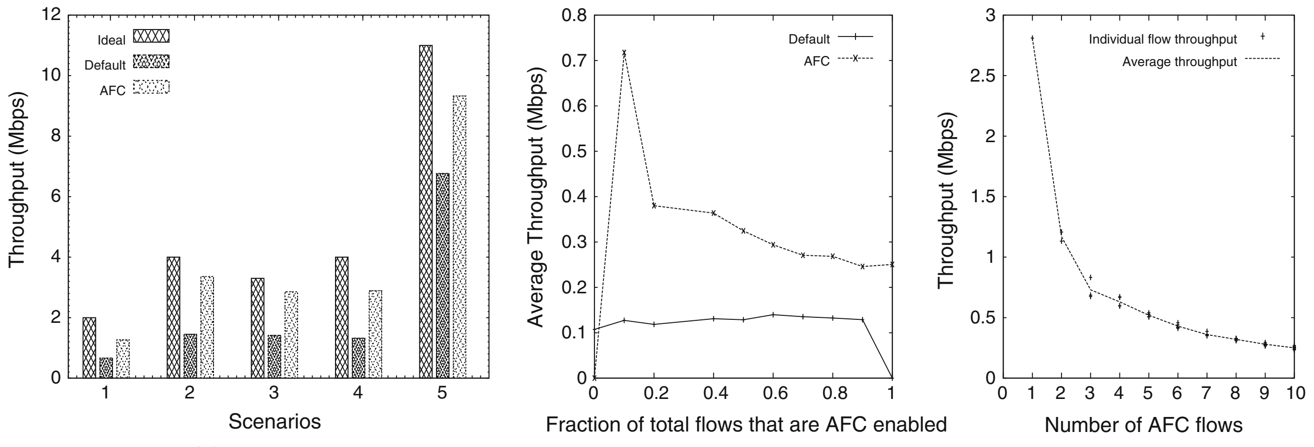
5.2 Throughput gain

For throughput analysis, we consider the scenarios mentioned in Table 3, for RTT = 530 ms. Present auto-tuning techniques [4] configure the receive buffer based on the perceived bandwidth-delay product, which is minimum(average network rate, average application rate) $\times$ 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 s.

Figure 6(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.

5.3 Fairness properties

To evaluate fairness between concurrent optimized and unoptimized connections we use a dumbbell topology with 10 TCP connections, as shown in Fig. 5. Senders $S_1 \ldots S_{10}$ are connected to router Rt_1 through individual links of



(a) Throughput **(b)** Fairness with classical TCP **(c)** Fairness with AFC

Fig. 6 Throughput gains and Fairness analysis of AFC

10 Mbps rate and 5 ms delay. Router Rt_1 is connected to another router Rt_2 with a network link of delay 255ms. The bandwidth of this link fluctuates in the pattern of $\langle 2, 4, 4\,\text{Mbps}\rangle$ with a time-period of 1 RTT, i.e. 530 ms. All the receivers $R_1 \ldots R_{10}$ are connected to router Rt_2 through individual links of 10 Mbps rate and 5 ms delay. Each receiver has an application running on it whose read rate fluctuates as $\langle 0, 6, 6\rangle$ Mbps with a time period of 1RTT. Considering fair distribution of link bandwidth, each connection gets an average network rate of 0.33 Mbps. The receive buffers are thus set to 0.33 Mbps $\times$ 530 ms = 22 KB. Each connection in the simulation runs for 600 s.

5.3.1 *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 Fig. 6(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 0.33 Mbps 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.*

5.3.2 *Fairness among AFC flows*

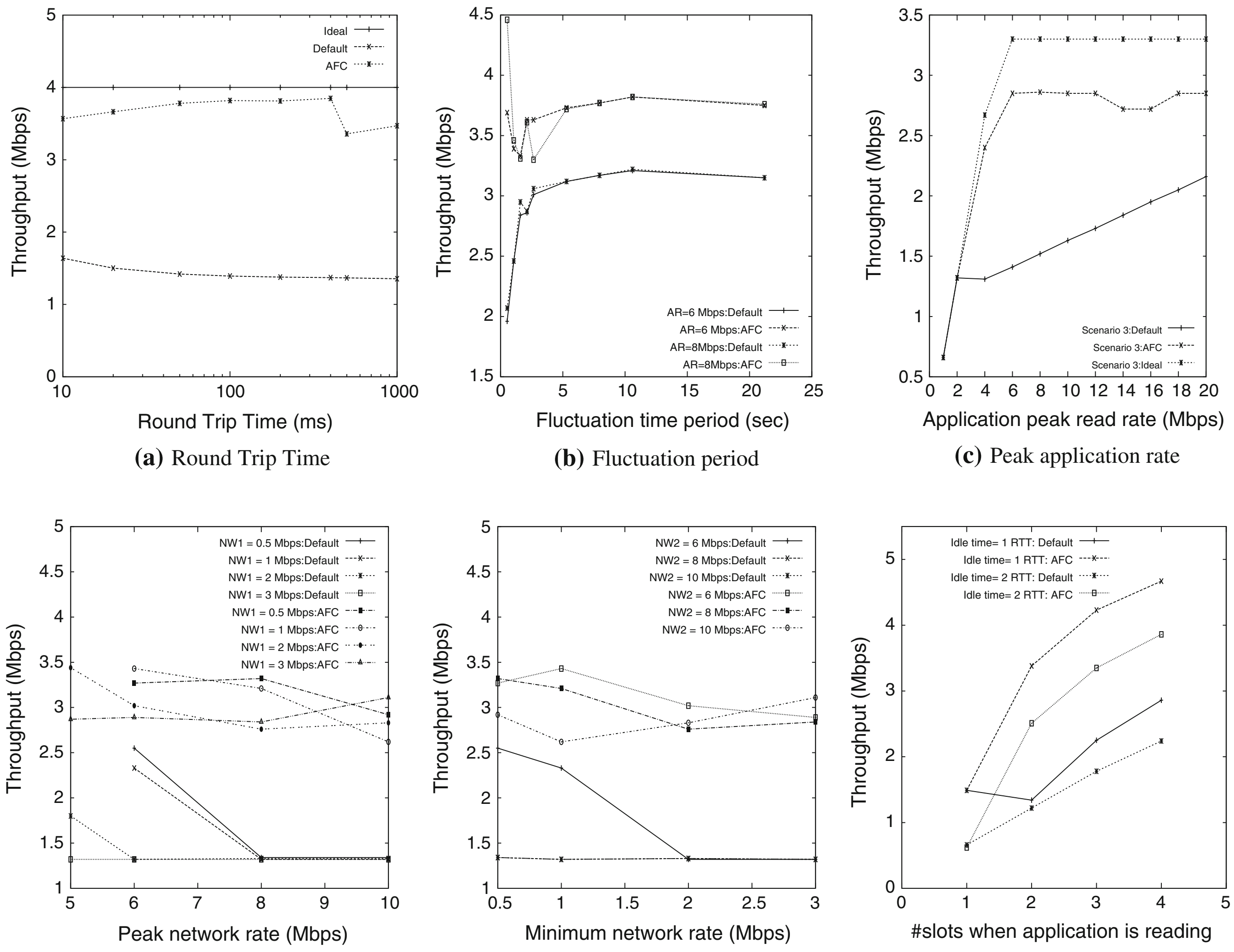
To demonstrate fairness amongst AFC flows we use the same dumbbell topology as above. However, this time we present results for increasing number of TCP connections. All the TCP connections use AFC as the flow control mechanism. The receive buffer size is adjusted down based on the number of connections (from 213 KB for one connection to 22 KB for ten connections). The average throughput enjoyed by connections is shown in Fig. 6(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.*

5.4 Sensitivity analysis

In this section we discuss how Adaptive Flow Control reacts to variations in the RTT, the time period of fluctuation, application read rate, network rate and the application fluctuation profile. We also present the performance of default TCP flow control for each case.

5.4.1 *Sensitivity to round trip time*

The NS2 simulation in Sect. 2 and the macroscopic results above consider a round trip time(RTT) of 530 ms. While we use this number as a representative of delays seen over 3G networks, the impact of flow control is equally significant in low delay scenarios as well. With the advent of 4G cellular technologies, round trip times have become smaller. In this section, we evaluate the performance of AFC over varying RTT. We consider the simulation scenario 2 from Table 3 for this analysis and vary the RTT from



(a) Round Trip Time **(b)** Fluctuation period **(c)** Peak application rate

(d) Peak network rate **(e)** Minimum network rate **(f)** Application fluctuation pattern

Fig. 7 Scenario description and sensitivity analysis of AFC

10 ms to 1 s. The receive buffer size is also changed in each case to comply with $min(AvgNW, AvgAR) \times RTT$. Figure 7(a) shows the ideal TCP throughput and the throughput observed with default flow control and AFC. The RTT is shown with a log scale for ease of presentation. We observe that AFC shows more than 100 % improvement over default TCP for all RTT values. Additionally, AFC throughput stays between 83 and 96 % of ideal throughput. The drop in throughput at 500 ms just reflects the impact of RTT on TCP performance as larger delay means slower rate of growth of congestion window.

5.4.2 *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 8 Mbps to simulate a network limited scenario. The throughput of default flow control and adaptive flow control are compared in Fig. 7(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 and tablet environments, as we have observed in previous sections, belong to the former set.

5.4.3 Sensitivity to peak application read-rate

In this evaluation, we vary the peak application read rate in the ⟨0, AR, AR⟩ profile in the setup of *Scenario* 3. The network is the bottleneck in this scenario, hence the ideal throughput remains 3.3 Mbps. Results are presented in Fig. 7(c). The receive buffer of 213 KB is more than sufficient when the read rate is less than 2 Mbps. Hence, the default throughput is optimal. However, as the application read rate grows current flow control grows linearly with the application read rate reaching 65 % of the ideal even at reading rates of 20 Mbps. Adaptive flow control, on the other hand, grows up to 86 % and more of the expected throughput in all cases. *We observe that AFC can scale with application read rate faster than classic flow control.*

5.4.4 Sensitivity to network rate

For *Scenario* 4, we modified the network profiles to study the change in throughput. Given the network profile of $\langle NW1, NW2, NW2\rangle$, we first keep $NW1$ constant and modify $NW2$, then keep $NW2$ constant and modify $NW1$. In all cases, the average application rate stays lesser than the network rate, hence the ideal throughput expected is 4 Mbps. Figure 7(d) shows the variation in throughput when the peak network rate is altered for the same minimum network rate. Figure 7(e) shows the variation in throughput when the minimum network rate is altered for the same peak network rate. *While default flow control shows a degradation of up to 50 % over a bandwidth variation of 2.5 Mbps, the maximum degradation of AFC is only 25 % over a bandwidth span of 4 Mbps.*

5.4.5 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.

From the application profile of $\langle 0, 6, 6\rangle$ 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 15 Mbps and the RTT is 530 ms. The results are shown in Fig. 7(f).

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.*

6 Related issues and discussion

- *Computational Overhead*: Adaptive flow control requires the receiver to monitor the rate at which the buffer is getting drained at the receiver. A sample of application read rate is computed whenever the receiver gets any new data or the application reads from the buffer. Both these computations can be piggy-backed on TCP receive module and the receive call from an application on a TCP socket, respectively. In order to avoid overshoots in calculation when a bunch of packets are read together, a single sample of application read rate is computed when the receive/read module is invoked. Two new state variables; *smooth_rx* and *last_rx*, are maintained to monitor application read rate at the receiver. If the application read rate changes beyond a factor of the last rate and no ACK is scheduled for a while, a proactive feedback is sent to the sender.

  The computation at the sender is done whenever an acknowledgement is received; the flow window is computed by adding the advertised window and RTT times application read rate. The window size and read rate are read from the TCP header and round trip time is pre-computed at the sender. The sender also records a timestamp; *ts_recover*, at every open window event to manage reliability at the sender.

All in all, AFC introduces one state variable at the data sender, two state variables on the data receiver and one TCP header options field into the existing TCP protocol. Constant time computations are added on data/ACK receive at receiver/sender, respectively. Thus, AFC introduces a constant magnitude overhead over classical TCP flow control.

As part of future work, we plan to build a prototype of AFC on smartphones and tablets to evaluate the computational overhead on real systems.

- *Application in PC environment*: We have motivated adaptive flow control in mobile platforms, as resource limitations make TCP flow control more vulnerable. We believe that adaptive flow control can also be applied to other flow control dominant computing environments, like servers and data centers. Though powerful processors, more memory and flow control solutions such as Linux auto-tuning prevent TCP flow control from becoming a bottleneck for application performance, adaptive flow control can reduce the buffer overheads per TCP connections.

7 Related work

A number of TCP optimizations have been presented for mobile hosts. Mobile TCP [11] does it through an asymmetric transport protocol which offloads IP processing to the base station instead of the mobile device. AFC, on the other hand tries to address the deficiencies of TCP flow control, which are magnified in mobile phone platforms.

In [12] and [13], the authors try to address the impact of mobility and handoffs on TCP congestion control. TCP Westwood [14] is another protocol optimization which aims to reduce the impact of random losses on TCP congestion control. These solutions optimize TCP congestion control. AFC is a complementary approach to these solutions as it aims to fix issues with flow control.

Several variants of TCP flow control have also been proposed in related work. Automatic Buffer Tuning [5] presents an algorithm to dynamically configure TCP sender buffer by comparing the congestion window size and the sender buffer size. They maintain the receiver buffer at the maximum allowed size. Dynamic Right Sizing [3] and Auto-tuning in Linux [4] implement receiver side solutions to grow the window sizes to match the available bandwidth. The Wed100 [15] 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 redefining the very concept of flow control window.

8 Conclusions

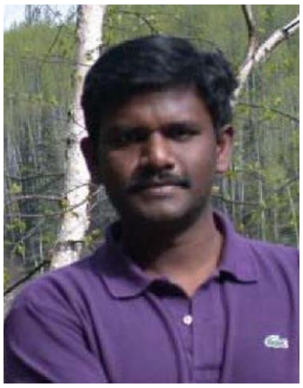
In this paper, we discuss the deficiencies in classical TCP flow control. These deficiencies are magnified on mobile platforms, due to the resource constraints. We demonstrate, both empirically and theoretically, that to address this problem, we need an Adaptive Flow Control(AFC) which makes a shift from an entirely buffer dependent flow control mechanism, to one that reacts to the application read rate. Through NS2 simulations we show that AFC performs better than classical TCP flow control, exhibits fairness and is robust to variations in network, application rate, fluctuation time and pattern.

References

1. Google Octane Benchmark. [Online]. Available: developers.google.com/octane.
2. I. S. Institute. (1981). *RFC 793*. [Online]. Available: rfc.sunsite.dk/rfc/rfc793.html.
3. Weigle E., & Chun Feng, W. (2001). Dynamic right-sizing: A simulation study. In *IEEE ICCCN*.
4. Linux Auto Tuning. [Online]. Available: www.kernel.org/.
5. Semke, J., Mahdavi, J., & Mathis, M. (1998). Automatic TCP buffer tuning. *Computer communication review*.
6. Franklin, G. F., Powell, D. J., & Emami-Naeini, A. (2001). *Feedback control of dynamic systems*. Upper Saddle River: Prentice Hall PTR.
7. Oppenheim, A. V., & Schafer, R. W. (1975). *Digital signal processing*. Upper Saddle River: Prentice-Hall.
8. Sinha, P., Nandagopal, T., Venkitaraman, N., Sivakumar, R., & Bharghavan, V. (2002). *Wtcp: A reliable transport protocol for wireless wide-area networks*. Wireless Networks, pp. 301–316.
9. Hsieh, H.-Y., & Sivakumar, R. (2002). ptcp: An end-to-end transport layer protocol for striped connections. In *IEEE ICNP*.
10. Mathis, M., Mahdavi, J., Floyd, S., & Romanow, A. (1996). *RFC 2018*. [Online]. Available: www.faqs.org/rfcs/rfc2018.html.
11. Haas, Z. J. (1997). Mobile-TCP: An asymmetric transport protocol design for mobile systems. In *IEEE international conference on communications*.
12. Bakre, A., & Badrinath, B. R. (1995). I-TCP: Indirect TCP for mobile hosts. In *International conference on distributed computing systems*.
13. Balakrishnan, H., Seshan, S., Katz, R. H., & Katz, Y. H. (1995). Improving reliable transport and handoff performance in cellular wireless networks. *Wireless networking*.
14. Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M. Y., & Wang, R. (2001). TCP westwood: Bandwidth estimation for enhanced

transport over wireless links. In *ACM conference on mobile computing and networking*.
15. Heffner, J. *High bandwidth TCP queuing*. [Online]. Available: www.psc.edu/jheffner/papers/senior_thesis.pdf.

Shruti Sanadhya is a 2013 Ph.D. graduate from the School of Computer Science at Georgia Institute of Technology, now working as a researcher at HP Labs. From 2008 to 2013, she was a member of the GNAN research group led by Prof Raghupathy Sivakumar. Her research interests are in developing algorithms and protocols to improve network performance on smartphones and tablets. She received her B.Tech. degree in Computer Science and Engineering from IIT Kanpur, India in 2008.



Raghupathy Sivakumar is a Professor in the School of Electrical and Computer Engineering at Georgia Tech. He leads the Georgia Tech Networking and Mobile Computing (GNAN) Research Group, where he and his students do research in the areas of wireless networking, mobile computing, and computer networks. He currently serves as the Co-Founder, Chairman and CTO for StarMobile, Inc., a next generation enterprise mobility company. Previously, he served as a technologist for EMC Corporation between 2011 and 2012, as the founder and CTO of Asankya, Inc. (now EMC), between 2004 and 2011, and as a technologist for EG Technology, Inc., between 2001 and 2004.