ULTRA-MOBILE COMPUTING: ADAPTING NETWORK PROTOCOLS AND ALGORITHMS FOR SMARTPHONES AND TABLETS

A Dissertation
Presented to
The Academic Faculty

by

Shruti Sanadhya

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology
December 2013

ULTRA-MOBILE COMPUTING: ADAPTING NETWORK PROTOCOLS AND ALGORITHMS FOR SMARTPHONES AND TABLETS

Approved by:

Professor Raghupathy Sivakumar,
Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Kishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Professor Mostafa Ammar
School of Computer Science
Georgia Institute of Technology

Dr Kyu-Han Kim
Networking and Communications Lab
Hewlett Packard Laboratories

Professor Ellen Zegura
School of Computer Science
Georgia Institute of Technology

Dr Jatinder Pal Singh
Mobile Innovation Strategy
Palo Alto Research Center

Date Approved:

To Amma and Papa, without whom nothing would have been possible.

ACKNOWLEDGEMENTS

This dissertation culminates a very formative period of my life. In the process of pursuing my PhD, I have met several intellectual and kind-hearted individuals. I am grateful to the support and encouragement offered by each of them.

First and foremost I would like to extend my sincere gratitude to my advisor, Prof. Raghupathy Sivakumar. This dissertation would not have been possible without his vision for mobile computing and, more importantly, his faith in me. I started in the GNAN research group with close to nil experience in research and here I learned how to find new research problems and solve them with perseverance. Siva constantly pushed me to give my best. His emphasis on quality of research and working on problems which can have a real impact have led to some good publications during my PhD. He inspired me to believe in my work and persistently strive towards a solution, despite failures, such as paper rejections. His constant advice on telling a story in each paper and presentation has enhanced my communication skills. Our one-on-one brainstorming sessions have taught me how to approach complex problems by solving small components while not losing sight of the overall goal. These learnings will go a long way with me, not just on the professional front but also on the personal front.

In addition to my dissertation, I am also thankful to Siva for exposing me to other aspects of being a researcher, such as reviewing work of other students, writing proposals, mentoring other students, taking lectures, collaborating with people in industry. These diverse experiences have served as diverse lenses to look at my own knowledge and has made me more confident of my approach to research. It has also opened my mind towards the rapid pace of technology and constant demand to innovate. I believe that I have come a long way since I started at Georgia Tech and

I thank him for mentoring me.

I have also been fortunate in finding a prolific set of peers in my research group. Working on a research problem I would often get stuck trying to get some results and not being able to get them. At such moments, I would often turn to fellow GNAN students: Zhenyun, Sandeep, Sriram, Cheng-Lin, Chao-Fang, Jiechao, Bhuvana, Nishith and Uma. Brain-storming sessions with each of them helped me multiple times to rethink my approach and some times to even rethink the problem I was trying to solve. I would like to thank Zhenyun, Sandeep, Sriram and Cheng-Lin for continuing their role as mentors for me even after graduating. At the same time, I would like to extend a warm thanks to Chao-Fang, Bhuvana and Uma for showing immense patience in listening to numerous practice talks for my conference presentations, thesis proposal, job interviews and thesis defense, and always providing relevant feedback.

Outside GNAN and Georgia Tech, I owe thanks to my external collaborator, Kyu-Han, who has been a constant support for the last three years. His constructive feedback on my data analysis, paper writing and presentations, both as a collaborator and as a thesis committee member, have added to my growth. I look forward to start my career as a researcher at HP Labs under his leadership. For the *Asymmetric caching* component of my dissertation, I would like to thank Mr. Nitin Agarwal at the University of Illinois for early discussions and pointers on techniques to identify changes in stationarity in a time-series.

As I thank all the people who directly influenced this dissertation, I would also like to acknowledge those who prepared me to embark on this journey and gave me the strength to keep going. My parents have been the most integral part of my success. They have supported me at every step in my life. *Amma* (my mom) taught me to always work hard and be self-dependent and *Papa* taught me to stay focused on my goal and be humble in the pursuit. I dedicate my dissertation to both of them. I can never thank them enough.

I also owe my deepest gratitude to my grandparents, *Nani* and *Nana*, and my grand-aunt *Bua*. Their compassion, words of encouragement and advice has played a major part in shaping my childhood. I would also like to thank Swati, my elder sister, for being my honest critic and a compassionate advocate. Through the last five years, two dear friends, Bhanu and Shatam, have supported me every step of the way. I thank both of them from the bottom of my heart.

Atlanta is the first city I have called home after my home town Kanpur in India. This would not have been possible without the love of my extended family: my uncle (Atul), my aunt (Kanchan), my cousin sister (Maansi) and my cousin brother (Kushal). I thank them for being there. The friends I made while at Georgia Tech have given me good company in my fun-filled graduate life. I thank Surabhi, Neha, Pooja, Luxmi, Ashish, Ramik, Karthik, Harsha, Chao-Fang, Bhuvana, Uma, Anshul and Vartika for making the last five plus years very memorable for me. I will always cherish our friendship.

Lastly, this dissertation would not have been complete without the rigorous review from my committee members: Prof. Mostafa Ammar, Dr. Kyu-Han Kim, Prof. Kishore Ramachandran, Dr. Jatinder Pal Singh and Prof. Ellen Zegura. I thank each one of them for devoting their time to review my work and providing constructive feedback.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Smartphones and tablets have been growing in popularity. These ultra mobile devices bring in new challenges for efficient network operations because of their mobility, resource constraints and richness of features. There is thus an increasing need to adapt network protocols to these devices and the traffic demands on wireless service providers. This dissertation focuses on identifying design limitations in existing network protocols when operating in ultra mobile environments and developing algorithmic solutions for the same.

Our work comprises of three components. The first component identifies the shortcomings of TCP flow control algorithm when operating on resource constrained smartphones and tablets. 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.

The second component of this work looks at network deduplication for mobile devices. With traditional network deduplication (dedup), the dedup source uses only the portion of the cache at the dedup destination that it is aware of. We argue in this work that in a mobile environment, the dedup destination (say the mobile)could have accumulated a much larger cache than what the current dedup source is aware of. In this context, we propose *Asymmetric caching*, a solution which allows the dedup destination to selectively feedback appropriate portions of its cache to the dedup source with the intent of improving the redundancy elimination efficiency.

The third and final component focuses on leveraging network heterogeneity for prefetching on mobile devices. Our analysis of browser history of 24 iPhone users show that URLs do not repeat exactly. Users do show a lot of repetition in the

domains they visit but not the particular URL. Additionally, mobile users access web content over diverse network technologies: WiFi and cellular (3G/4G). While data is unlimited over WiFi, users typically have monthly limits on data over the cellular network. In this context, we propose *Precog*, an action-based prefetching solution to reduce cellular data footprint on smartphones and tablets.

CHAPTER I

INTRODUCTION

The adoption of mobile devices such as smartphones has reached a significant threshold with the number of such devices shipped now surpassing the number of PCs shipped [1]. Nearly 40% of Internet time is now attributed to mobile devices such as smartphones and tablets [2]. Not only are consumers adopting mobile devices at a blistering pace, but such adoption is being witnessed within the traditionally conservative enterprise sector as well. 71% of enterprises are currently deploying or planning the deployment of mobile applications [3]. Such adoption amongst enterprises is driven by a clear return-on-investment from mobility in the form of higher employee productivity, reduced paper work, and increased revenue [4].

These trends have become possible because of multiple reasons: developed operating systems, web browsers with support for full websites, WiFi connectivity and rich set of application programming interfaces (APIs). Smartphones today are much more capable than their predecessors. Additionally, the applications have also adapted to mainstream consumers, making them even more popular. The simplicity, portability and always-on connectivity of smartphones and tablets makes them constant companions of users. Advancements in mobile broadband technologies, from 2G to 3G to 4G, has further accelerated this trend. In this work, we argue that smartphones and tablets have introduced a new paradigm in mobile computing, which we refer to as *ultra-mobile computing*.

Specifically, *ultra-mobile computing* can be characterized by the following distinguishing features:

- Ubiquitous wireless networks: Past decade has seen a surge in development

of cellular networks. With 3G/4G technologies, cellular networks can provide broadband connectivity to users wherever they go. Cellular infrastructure has grown at an exponential rate to create a congenial ecosystem for smartphones and tablets. In addition to this, the prevalence of WiFi technology in indoor environments has also boosted the demand for smartphones and tablets.

- Highly portable compute devices: While traditional mobile computing focused on laptops, personal digital assistants(PDAs) and feature phones, smartphones and tablets are *ultra-mobile devices*, which combine the rich features of laptops with the mobility of PDAs and feature phones.

- Consumer adaptation: In addition to the above changes, smartphones and tablets appeal to a wider consumer base than laptops. Users do not need as much technical expertise to operate smartphones and tablets as for laptops. Application developers are now making rich content available to end-users through simple and intuitive mobile applications, increasing their mainstream acceptance.

Cisco Visual Networking Index(VNI)[5] report shows that by the end of 2012, the number of mobile-connected devices will exceed the number of people on earth. The report also shows that the average amount of traffic per smartphone in 2011 was 150 MB per month, up from 55 MB per month in 2010. It is predicted that due to increased usage on smartphones, handsets will exceed 50 percent of mobile data traffic in 2014. In addition to this, the number of mobile-connected tablets has tripled to 34 million in 2011, and it is estimated that these tablets will generate almost as much traffic in 2016 as the entire global mobile network in 2012.

These global trends force us to think whether network protocols and algorithms need to evolve with *ultra-mobile computing*. As the applications on smartphones

and tablets keep getting richer, the hardware growth is still constrained to maintain their compactness and mobility. In addition to that, the heterogeneous wireless networks to which these devices connect introduce a cost-performance trade-off in efficient network operations. There is thus a need to examine how efficient current network protocols perform with ubiquitous connectivity, highly portable yet resource constrained devices, exploding consumer base and heterogeneity of wireless networks. Identifying and resolving the impact of these conflicting characteristics of *ultra-mobile computing* on existing network protocols and algorithms forms the core of this thesis.

Specifically, we look at three distinct problems:

- *Rethinking transport layer protocols for ultra-mobile devices*: Traditionally, transport layer optimizations have focused on congestion control approaches as flow control was never considered a dominating factor. However, as smartphones and tablets gain popularity, the resource constraints on these devices increase the significance of flow control. Our experiments on HTC G1 phone, Samsung Galaxy S 4G phone and Samsung Galaxy Tablet show that TCP throughput on these devices can degrade from 10% to 50% as workload increases. No such degradation is observed if the same experiment is conducted on a laptop. During each experiment, we ensure that network is not the bottleneck and congestion control does not come into play. Thus flow control is a governing component of transport layer operations on ultra-mobile devices.

  The first component of this thesis identifies the limitations of flow control on mobile devices. In particular, we observe that the existing TCP flow control[6] does not react efficiently to fluctuating application read rates, is inefficient in recovering from zero window events and cannot reap the benefits of buffer autotuning. We then propose an *adaptive flow control (AFC)*[7] 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.

- *Improving network deduplication for ultra-mobile devices*: Network deduplication (dedup) is an attractive approach to improve network performance for mobile devices. With traditional dedup[8, 9, 10, 11], the *dedup source* uses only the portion of the cache at the *dedup destination* that it is aware of. We argue in this work that in a mobile environment, the *dedup destination* (say the mobile) could have accumulated a much larger cache than what the current *dedup source* is aware of. This can occur because of several reasons ranging from the mobile consuming content through heterogeneous wireless technologies, to the mobile moving across different wireless networks.

  In this context, we propose *asymmetric caching*[12], a solution that is overlaid on baseline network deduplication, but which allows the *dedup destination* to selectively feedback appropriate portions of its cache to the *dedup source* with the intent of improving the redundancy elimination efficiency. We show using traffic traces collected from 30 mobile users, that with asymmetric caching, over 89% of the achievable redundancy can be identified and eliminated *even when the dedup source* has less than *one hundredth of the cache size as the dedup destination*. Further, we show that the ratio of bytes saved from transmission at the *dedup source* because of asymmetric caching is over 6× that of the number of bytes sent as feedback. Finally, with a prototype implementation of asymmetric caching on both a Linux laptop and an Android smartphone, we demonstrate that the solution is deployable with reasonable CPU and memory overheads.

- *Smart prefetching solutions for ultra-mobile devices*: Prefetching is predictive fetching of content which is likely to be accessed by a user in the future. Traditional prefetching approaches[13, 14, 15] have focused on reducing the access

latency of webpages, identified through the uniform resource locators(URLs). Prefetching solutions in prior works calculate the popularity of each URL by counting the number of occurrences of each URL in user's web history. Popular URLs are prefetched if the URLs accessed right before them are accessed again or if it is the hour of the day during which a URL is mostly accessed.

All existing prefetching solutions perform *n*ame based prefetching. They focus on prefetching the exact URL which was requested before. However, our analysis of network traces of five Android users and 24 iPhone users show that URLs do not repeat exactly. Users do show a lot of repetition in the domains they visit but not the particular URL. The main reason for this is that web content is very dynamic. If a user reads the headline news on *nytimes.com* everyday, the URL of the headline news page changes everyday, or every few hours. Prior works on prefetching does not apply to these scenarios. Additionally, mobile users access web content over diverse network technologies: WiFi and cellular (3G/4G). While data is unlimited over WiFi, users typically have monthly limits on data they can download over the cellular network. This creates an *u*nbalanced cost problem and the question we try to answer here is "How can time-shifted cheaper network access (WiFi) be leveraged to offset costs on the more expensive network (cellular)?".

The third component of this dissertation,*Precog*, is a name-independent network-aware prefetching solution for smartphones and tablets.

While these three problems do not exhaust the network performance issues on smartphones and tablets, we consider these as three directions where we can reap significant benefits by adapting existing network protocols. All the three approaches mentioned above focus on improving the network performance with respect to mobile devices. Adaptive flow control increases the capacity of the network, dedup reduces the load on the network while network-aware prefetching time shifts the load from

overly congested cellular networks to high capacity WiFi networks.

The rest of this dissertation is organized as follows: we discuss prior related research in chapter 2, chapter 3 gives detailed description of *adaptive flow control*, chapter 4 presents *asymmetric caching*, chapter 5 discusses *precog* and chapter 6 describes integrated operations of the three solutions. Finally, chapter 7 discusses directions of future research and chapter 8 concludes our findings.

CHAPTER II

RELATED WORK

Several prior works have looked at transport layer optimizations, network deduplication and prefetching in isolation. However, these solutions were not driven by ultra-mobile computing and hence do not apply directly to such environments. Here we present the individual differences between related research and the components of this thesis.

2.1 Transport layer optimizations

A number of TCP optimizations have been presented for mobile hosts. Mobile TCP [16] 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 [17] and [18], the authors try to address the impact of mobility and handoffs on TCP congestion control. TCP Westwood [19] 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 [20] 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 [21] and Auto-tuning in Linux [22] implement receiver side solutions to grow the

window sizes to match the available bandwidth. The Wed100 [23] 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.

2.2 Deduplication solutions

Deduplication of network traffic has been considered at multiple granularities. The prior research in this direction can be categorized as:

- **Network dedup approaches:** The notion of network dedup was first presented in [11], where packets are decomposed into segments using the Rabin fingerprinting algorithm so that partial-packet redundancy can be exploited. This approach was developed further in value-based web caching [10], where the data is cached on its value rather than its name. The idea of using packet caches on routers was introduced in [24]. In [25], the authors perform an experimental study of redundancy across 12 different enterprise networks. Both [24] and [25] identify that significant bandwidth savings can be achieved by using packet level dedup approaches, thereby motivating the current work. Similarly, EndRE [8] is an end-to-end solution for network dedup, which presents a new fingerprinting scheme called SampleByte. Recently, [9] proposed overhearing content in wireless networks to dedup across wireless users. Celleration [26] is another sender-driven dedup solution that leverages inter-user redundancy for a single point of attachment. All the above works are designed for static scenarios, do not work across points of attachment and do not support IP address changes due to mobility. Asymmetric caching is complementary to the above

works and specifically optimizes wireless traffic, without requiring modification to Internet servers.

- **Application layer dedup:** Application layer works include caching HTTP objects on browsers [27] and on proxy servers [28], delta encoding, file differencing (e.g., VCDIFF) [29], techniques for detecting duplicate transfers of the same file [30] and techniques such as base-instant caching [31], template caching [32] for enhanced cacheability of dynamic objects. More recent developments include content-delivery networks [33] and peer-to-peer caching solutions [34]. All these solutions operate at the granularity of files or application-objects and hence do not provide fine-grained redundancy elimination. Further, they are application layer solutions and have to be realized independently for every single application. Most importantly, using proxy or other intermediate caches while reducing the load on servers does not reduce the traffic on the wireless link. Asymmetric caching operates agnostic to different applications.

- **Transport layer dedup:** Recently, Zohar *et* al [35] propose an end-to-end receiver driven dedup solution that extends TCP options. The receiver matches TCP stream chunks with its cache and sends predictions for future chunks in the ongoing flow to the sender. The dedup solution in [35] is similar to asymmetric caching in that both use receiver driven feedback to improve dedup performance. However, there are fundamental differences. The solution in [35] is closely tied to the TCP protocol and operates at a coarse data-granularity. Asymmetric caching on the other hand is transport protocol agnostic and operates at sub-packet level granularity. The solution in [35] is an end-to-end solution, whereas asymmetric caching is a last hop solution. This is important as wireless service providers who have the motivation to utilize their spectrum better can deploy asymmetric caching without any dependencies on the content provider. Also,

the solution in [35] does not partition old connections into flowlets and hence maintains connections in their entirety. However, asymmetric caching partitions content into flowlets depending on their stationarity and this helps when the composition of connections changes in terms of a few objects or in terms of the ordering of the objects. Finally, the solution in [35] does not address how feedback might be chosen when chunks experience hits with multiple old connections. The feedback selection algorithm in asymmetric caching explicitly tackles this problem by choosing from multiple flowlets. This capability is especially important when operating at fine data granularities.

2.3 Prefetching approaches

Several prefetching solutions have been proposed for wired domains and wireless domains. These approaches can be split into two prominent categories:

- Server-driven prefetching approaches [13, 14], and proxy-driven approaches [36] determine content popularity by collecting inter-user statistics. A dependency graph is constructed at the server, which predicts the URI B most likely to be requested in the future, given URI A is requested. These solutions use content-based triggers for prefetching. Given the diversity of smartphone and tablet users[37], a server-driven approach is not desirable. Each user has its unique web access profile which cannot be applied to other.

- Client-based prefetching solutions, such as [15], uses the web history of an individual user to determine what URL is likely to be prefetched and when should it be prefetched. The prefetching engine considers the web history of an individual client and assigns a rank to each URI. Next, the prefetching engine considers the last visit time of the high ranked URLs to decide when to prefetch each URI. This approach uses time-based triggers for prefetching.

The solution in [38] takes a hybrid approach for prefetching in mobile environments. The URLs to be prefetched can be decided on either the client or the server. For a mobile client, a prefetching metric is calculated every time the mobile connects to a new network. If the available bandwidth and congestion levels permit, the popular URLs are prefetched. All the above solutions consider that the exact URI a user will access will be from the web history. This argument may not hold for dynamic web content as we show later in Chapter 5.

Kroeger *et al*[39] and Marquez *et al*[40] evaluate different caching and prefetching approaches over different network environments to study their latency benefits. They do not propose any new prefetching mechanism but give a comparative analysis of prior works. Recently, "Informed mobile prefetching"[41] proposed a prefetching solution for mobile devices where the overlying application predicts what content to prefetch and the prefetching API decides whether to prefetch or not based on the duration, energy usage and cellular data footprint of the download. *Precog* is different from all the above solutions as it aims to predict the *new* URI a user is going to access, by learning patterns from the web history.

Another aspect of *precog* is network awareness, i.e. preferring WiFi over 3G/4G for prefetching. Recently, Breadcrumbs[42] studied network connectivity predictions, where a Markov Model is built from user's WiFi connectivity history and used to predict WiFi network bandwidth and latency based on location. Unlike Breadcrumbs, smart prefetching considers heterogeneous networks and also tries to predict the content a user is going to access. Network unaware prefetching has also been studied for data storage solutions. Wherestore[43] proposed a location-aware data replication system for mobile devices. It provides a mechanism for application to specify which data should be cached on mobile device based on user's current and future location. It does not consider network usage while making prefetching/caching decisions, unlike smart prefetching.

CHAPTER III

ADAPTIVE FLOW CONTROL FOR TCP ON SMARTPHONES AND TABLETS

3.1 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

[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 work to only mobile phones and tablets.*

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.

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 $NS2$ 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.

3.2 Background and Motivation

3.2.1 Resource Constraints on Mobile Devices

Even though smartphones and tablets have been growing in performance since their inception, the 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 ultra-mobile devices and computers, we run a JavaScript benchmark, Octane[44], on the following devices:

- Laptop1: Lenovo Thinkpad X220 running Ubuntu 12.04 the 2.9 GHz Intel I7 processor and 4GB 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 2GB 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. Table 1 shows Octane results for the five devices. We observer 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 better 3x better than iPhone 5 with a similar processor speed and 4x better than Samsung

Table 1: Octane benchmark comparison

Device Id	Device	Octane Score
Laptop1	Thinkpad X220	19154
Laptop2	MacBook Air	8769
Smartphone1	Samsung Galaxy S4	2261
Smartphone2	iPhone 5	2941
Tablet	Samsung Galaxy Tab 10	1942

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.

3.2.2 TCP Flow Control Basics

TCP's flow control algorithm provides the receiver with the ability to control the rate at which the sender transmits [6]. 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 chapter, 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.

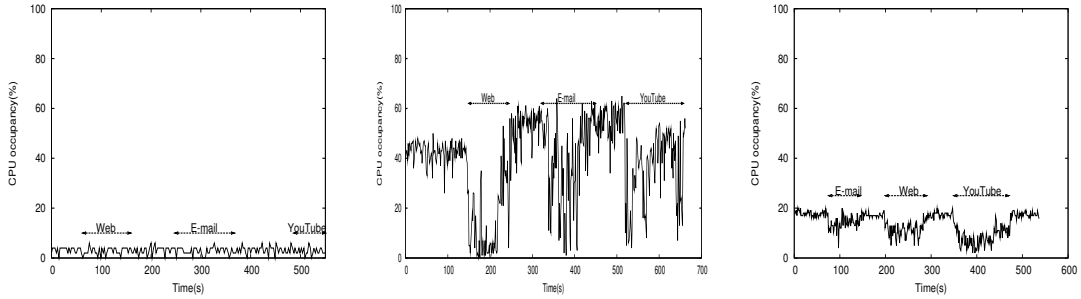
- Application read rate: The buffer occupancy in turn is heavily influenced by the application read rate at the receiver. The TCP receive buffer has no other influencers other than the input rate and the drain rate, as we discuss later in the section.

- 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.

3.2.3 Problems with TCP Flow Control on Mobile Devices

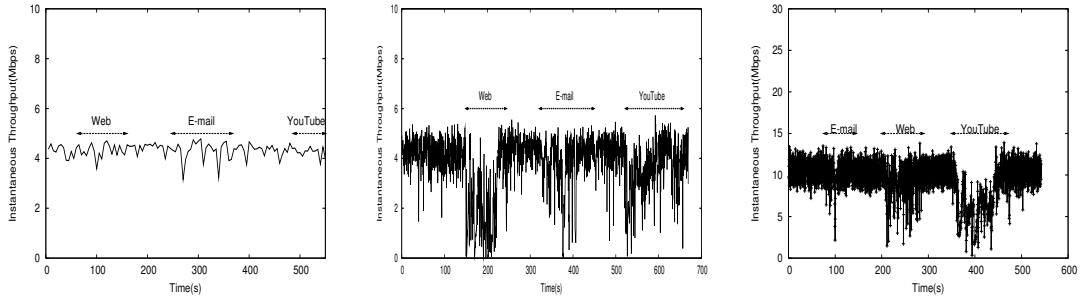
3.2.3.1 Flow control bottlenecks occur more often

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



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

Figure 1: 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

Figure 2: Comparison of instantaneous TCP throughput of FTP connection on laptop and mobile devices

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), 1(b) and 1(c) present comparative CPU allocation results for an FTP application running on a laptop (Dell Inspiron 1525 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$2GB) is downloaded from an Internet server down to the client. As the 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% 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 Figures 2(a), 2(b) and 2(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. 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.

3.2.3.2 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 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 530ms; (c) network rate of 15 Mbps; (d) average application read rate of 4 Mbps, with a fluctuation profile of $< 0, 6, 6 >$ (period of 1 RTT); and (e) receive buffer size equal to the

perceived BDP (min(NW, AAR)*RTT = 256KB). 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 Section 3.5.

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 4Mbps. *However, the aggregate throughput observed is only 1.45Mbps, a degradation of 63%(Figure 3).* Note that given the high network rate assumed, there are no congestion artefacts influencing the performance, and hence this degradation is directly due to the flow control behavior of TCP.

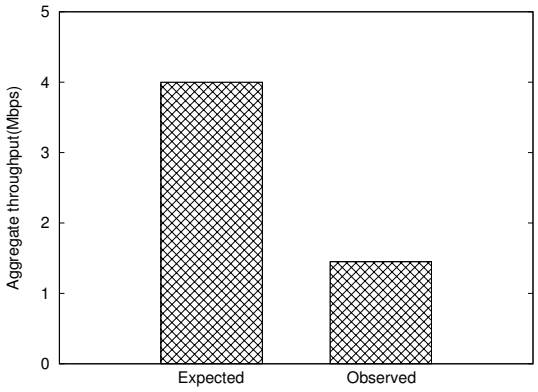


Figure 3: Impact of application read rate fluctuations on TCP throughput

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

3.2.4 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

[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

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. In some scenarios, the network rate is also made to fluctuate in a pattern of <NW1, NW2, NW2> 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:

3.2.4.1 ***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 = 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(∼3Mbps), 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

these currently. A configurable application read rate parameter was also added to simulate different application patterns.

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.

3.2.4.2 ***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 = 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 63% degradation), as shown in Figure 3. 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.

3.2.4.3 ***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[21], auto-tuning[22], 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 BDP).

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

Table 2: Network and application scenarios

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

achieve that performance, the receiver needs to make sure that the receive buffer is tuned to the network. Current buffer resizing solutions[21, 22, 20] 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 expected ideal. 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.

3.2.5 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 2, 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:

$$B_{req} = NR * RTT \tag{1}$$

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:

$$B_{req} = min(NR, AR) * RTT \tag{2}$$

- *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 2. Assuming a buffer size based on Equation (2) of 256KB, the expected throughput is 4Mbps ($min(NR, AAR)$), where AAR is the average application rate. However, the observed performance in the simulation study for the above parameters is only 1.45Mbps. 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 1.33Mbps 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:

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

We do verify in simulations that the above buffer allocation increases the observed throughput to 3.86Mbps. 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 Equation (3), we get the following:

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

- *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 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 $(0, 18, 18)$ (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:

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

Taking into account equations (2)- (5), the buffer required in a pure provisioning based strategy to cover all scenarios is $3 * AAR * RTT$. 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 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 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 this chapter is that if the application read-rate is already being monitored, could a better solution be derived to achieve the expected performance?

3.3 Theoretical Analysis

3.3.1 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 $0 \leq B \leq B_0$ and $0 \leq W \leq B_0$. Thus,

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

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.

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

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

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

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 [45]. In the following analysis, we look at some characteristics of this system and its implication on TCP's performance.

Eliminating TCP from the equations (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:

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

$$\text{multiplying (15) with integrating factor}$$

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

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

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

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 ,

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

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

[3]Note that any other periodic application profile can be represented as a sum of sine/cosine functions[46].

(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, 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:

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

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.

3.3.2 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 equation (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 equation* (28)*).*

A corrective term needs to be added in equation (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:

$$TCP = \alpha W + AR \tag{29}$$

working out equation (10)

$$W' = AR - TCP \tag{30}$$

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

$$\text{i.e., } W' = -\alpha W \tag{32}$$

$$\text{on solving, } W = B_0 e^{-\alpha t} \tag{33}$$

differentiating (33) and using in (22),

$$err = -\alpha B_0 e^{-\alpha t} \tag{34}$$

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

$$TCP = \alpha B_0 e^{-\alpha t} + AR \tag{35}$$

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.

3.4 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.

3.4.1 Key Design Elements

3.4.1.1 ***Using Application Read Rate***

The first design element in AFC follows directly from the theoretical analysis presented in Section 3.3. 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:

$$W_{fc} = B + AR * RTT \tag{36}$$

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.

3.4.1.2 ***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[47], pTCP[48]. 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

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

are planning to explore whether a more sophisticated scheme is warranted.

3.4.1.3 ***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.

3.4.1.4 ***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 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 $RTT/sender's\ window$.

3.4.2 AFC Solution Details

3.4.2.1 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

[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.

Algorithm 1 Data packet delivered at TCP receiver

Input: $data$ = Data packet received by TCP
$data.seqno$ = Sequence number of the first octet in the data
Variables: max_seen = Maximum sequence number seen by the receiver, even out-of-order

```
1: procedure ((r)eceive_data)
2:     bytes_read ← 0
3:     if data.seqno > read_nxt + bufsize then
4:         Drop packet
5:     else if data.seqno < read_nxt then
6:         is_dup ← True
7:     else if data.seqno > max_seen then
8:         mark all buffer spaces from max_seen to data.seqno as null
9:         buffer[data.seqno mod bufsize] ← data
10:        buffer[(data.seqno + 1) mod bufsize] ← null
11:        just_marked ← True
12:        max_seen ← data.seqno
13:    end if
14:    if read_nxt ≤ data.seqno ≤ max_seen then
15:        if not just_marked and buffer[data.seqno mod bufsize] ≠ null then
16:            is_dup ← True
17:        end if
18:        buffer[data.seqno mod bufsize] ← data
19:        while (Application needs the read_nxt byte and buffer[read_nxt mod
   bufsize] ≠ null ) do
20:            Pass buffer[read_nxt mod bufsize] to application
21:            read_nxt ← read_nxt + 1
22:            bytes_read ← bytes_read + 1
23:        end while
24:        rcv_nxt ← read_nxt
25:        while (buffer[rcv_nxt mod bufsize] ≠ null and rcv_nxt ≤ max_seen) do
26:            rcv_nxt ← rcv_nxt + 1
27:        end while
28:    end if
29:    if bytes_read ≠ 0 or rcv_nxt − read_nxt > 0 then
30:        ar_update(bytes_read, now())
31:    end if
32:    window ← bufsize − (rcv_nxt − read_nxt)
33:    if is_dup ≠ True then
34:        Generate ACK for packet: ack
35:        ack.win ← window
36:        ack.rx ← smooth_rx
37:        Send ACK
38:    end if
39: end procedure
```

Table 3: 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 $smooth\_rx$

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.

3.4.2.2 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. Algorithm (1) captures this logic. 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 $read\_nxt$. If not, it is dropped (line 3 and 4). 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 greater than 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

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

sample of the application read rate by invoking the *ar_update* module(algorithm (2)). 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 $smooth\_rx$ 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: $history\_factor \epsilon [0, 1]$ is the weight given to the old application rate estimate while computing the new one.
$last\_read$ = Time when this procedure was last called

```
1: procedure ((a)r_update)
2:     t_elapsed ← read_time − last_read
3:     if t_elapsed then
4:         last_rx ← smooth_rx
5:         smooth_rx ← history_factor ∗ smooth_rx + (1 − history_factor) ∗ num_bytes/t_elapsed
6:     end if
7:     last_read ← read_time
8: end procedure
```

Algorithm 3 Data packet read from buffer by application

```
1: procedure ((r)ead_buffer)
2:     bytes_read ← 0
3:     while Application can read buffer[read_nxt mod bufsize] do
4:         Pass buffer[read_nxt mod bufsize] to application
5:         read_nxt ← read_nxt + 1
6:         bytes_read ← bytes_read + 1
7:     end while
8:     ar_update(bytes_read, now())
9:     window ← bufsize − (rcv_nxt − read_nxt)
10:    if smooth_rx > factor ∗ last_rx || smooth_rx < factor ∗ last_rx then
11:        Generate an ACK for application update: ack
12:        ack.win ← window
13:        ack.rx ← smooth_rx
14:        Send ACK
15:    end if
16: end procedure
```

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[49] information and $smooth\_rx$ 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, as illustrated in algorithm (3). 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 $last\_rx$.

3.4.2.3 AFC Sender (ACK) Processing

To enable AFC at a TCP sender, new logic is introduced in processing the acknowledgement. Algorithm (4) captures this logic. 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.

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
$flow\_window$ = Flow window
$ts\_recover$ = Timestamp to ignore ACKs for packets sent during zero window event
$highest\_ack$ = 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

```
1: procedure ((r)eceive_ack)
2:     awnd ← ack.win
3:     app_rate ← ack.rx
4:     if rtt > 0 then
5:         flow_window ← awnd + app_rate * rtt
6:     end if
7:     if awnd = 0 and zw_flag = 0 then
8:         zw_flag ← 1
9:     else if awnd > 0 and zw_flag = 1 then
10:        zw_flag ← 0
11:        if not fast recovery phase then
12:            ts_recover ← now()                    ▷ Store the current time
13:            if ack.seqno > highest_ack then
14:                highest_ack ← ack.seqno
15:            end if
16:            snd_nxt ← highest_ack + 1
17:            Process SACK information
18:            Reset the retransmit timer
19:        end if
20:    end if
21:    if flow_window then
22:        if not fast recovery phase then
23:            if ack.seqno > highest_ack then
24:                Process the packet like a new ACK
25:            else if ack.seqno = highest_ack then
26:                Process SACK information
27:                if ack.ts_echo > ts_recover then
28:                    Process duplicate ACK
29:                end if
30:            end if
31:        end if
32:    else
33:        Send zero window probes to learn about open windows
34:    end if
35:    Send data, if allowed by min(cwnd, rwnd) and burst control
36: end procedure
```

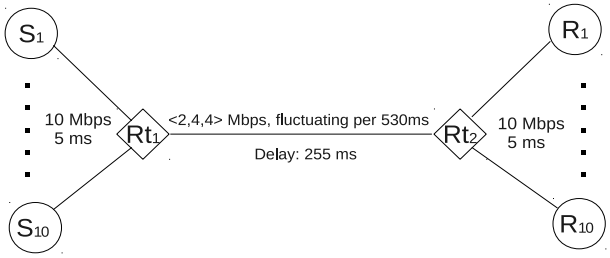


Figure 4: Topology for fairness evaluation

Table 4: Network and application scenarios

#	Application profile	Network profile	Receive buffer	Ideal throughput
1	$< 0, 6, 6 >$Mbps	2 Mbps	128KB	2Mbps
2	$< 0, 6, 6 >$Mbps	15 Mbps	256KB	4Mbps
3	$< 0, 6, 6 >$Mbps	$< 2, 4, 4 >$Mbps	213KB	3.3Mbps
4	$< 0, 6, 6 >$Mbps	$< 3, 6, 6 >$Mbps	256KB	4Mbps
5	$< 0, 18, 18 >$Mbps	$< 3, 15, 15 >$Mbps	704KB	11Mbps

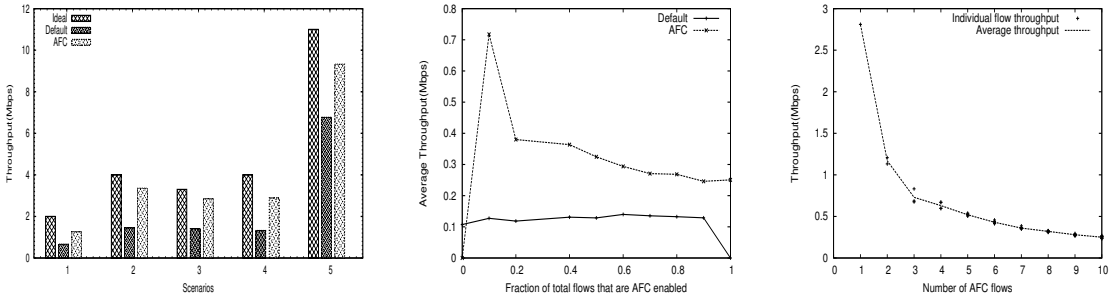
3.5 Performance

3.5.1 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 section 3.4 in NS2 TCP implementation. This Adaptive Flow Control(AFC) enabled TCP is referred to as AFC in future. We assume SACK [49] 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

[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



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

Figure 5: Throughput gains and Fairness analysis of AFC

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 265ms. For fairness analysis, we consider a dumbbell shaped topology defined later in section 3.5.3

3.5.2 Throughput Gain

For throughput analysis, we consider the scenarios mentioned in Table 4, for RTT = 530ms. Present auto-tuning techniques [22] 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 5(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.

3.5.3 Fairness Properties

To evaluate fairness between concurrent optimized and unoptimized connections we use a dumbbell topology with 10 TCP connections, as shown in Figure 4. 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 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.

3.5.3.1 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 Figure 5(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.*

3.5.3.2 Fairness among AFC Flows

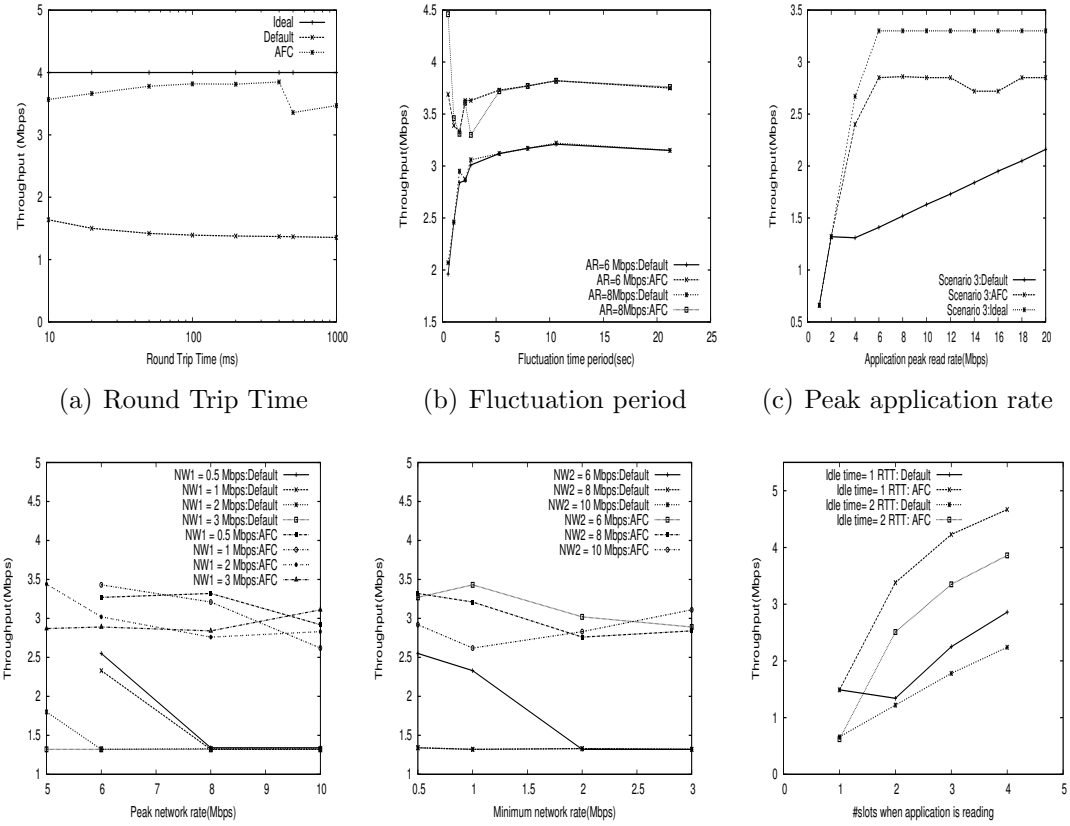
To demonstrate fairness amongst AFC flows we use the same dumbbell topology as above. However, this time we present results for increasing number of TCP connections. All the TCP connections use AFC as the flow control mechanism. The receive buffer size is adjusted down based on the number of connections (from 213KB for one connection to 22KB for ten connections). The average throughput enjoyed by connections is shown in Figure 5(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.*

3.5.4 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.

3.5.4.1 Sensitivity to Round Trip Time

The NS2 simulation in Section 3.2 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 4 for this analysis and vary the RTT from 10ms to 1s. The receive buffer size is also changed in each case to comply with $min(AvgNW, AvgAR) * RTT$. Figure 6(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%



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

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

Figure 6: Scenario description and sensitivity analysis of AFC

improvement over default TCP for all RTT values. Additionally, AFC throughput stays between 83% to 96% of ideal throughput. The drop in throughput at 500ms just reflects the impact of RTT on TCP performance as larger delay means slower rate of growth of congestion window.

3.5.4.2 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 8Mbps to simulate a network limited scenario. The throughput of default flow control and adaptive flow control are compared in Figure 6(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.

3.5.4.3 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 3.3Mbps. Results are presented in Figure 6(c). The receive buffer of 213KB is more than sufficient when the read rate is less than 2Mbps. 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 20Mbps. 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.*

3.5.4.4 Sensitivity to Network Rate

For *Scenario* 4, we modified the network profiles to study the change in throughput. Given the network profile of $< NW1, NW2, NW2 >$, 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 6(d) shows the variation in throughput when the peak network rate is altered for the same minimum network rate. Figure 6(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 2.5Mbps, the maximum degradation of AFC is only 25% over a bandwidth span of 4Mbps.*

3.5.4.5 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 $< 0, 6, 6 >$ 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 6(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.*

3.6 Issues

- ***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 precomputed 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.

- ***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.

CHAPTER IV

IMPROVED NETWORK DEDUPLICATION FOR MOBILE DEVICES

4.1 Introduction

Several recent efforts have established that there are considerable redundancies in network traffic [10, 25, 8] that are not fully leveraged by application layer approaches [30, 31, 32]. Network deduplication (dedup[1]) is a class of solutions that exploits such redundancies to improve network performance [10, 8, 11, 50]. Briefly, a dedup source (`dd-src`) intercepts traffic coming from the sender; segments the traffic into chunks of byte-sequences; and sends across only the hash of a byte-sequence if it is a repeating sequence. The dedup destination (`dd-dst`) then inflates any hashes back to the original byte-sequences and forwards the traffic to the eventual receiver. The reduction in the number of bytes sent between the `dd-src` and `dd-dst` results in improved network performance by allowing the network to sustain a greater traffic load and by reducing congestion levels for a given volume of traffic.

The application of dedup to wireless environments is an attractive proposition due to the ever-prevalent pressures on wireless capacities. Wireless service providers continually attempt to support more users and higher traffic loads without having to use additional spectrum, and dedup is a viable low-cost solution to do so. A straightforward deployment of dedup in a wireless environment would involve the `dd-src` residing in the wireless service provider's network (e.g., the SGSN in 3G) and the `dd-dst` residing on the mobile.

We argue in this work that dedup faces a unique challenge when used in mobile

[1]For brevity we refer to network deduplication as dedup in rest of this description.

environments. In static wireline environments, the `dd-src` is typically aware of the complete cache at the `dd-dst` by virtue of being on the data path to the destination. However, there exist several mobile scenarios where the `dd-src` is likely to have knowledge of only a small subset of the cache at the mobile. We elaborate on such scenarios in [12]. However, if such an asymmetry exists between the caches at the `dd-src` and at the `dd-dst`, the efficiency of traditional dedup techniques is a restricted function of the smaller cache.

In this context, we consider the following question: *How can all of the past cached information at the mobile be successfully leveraged for dedup by any given* `dd-src`*?* In answering the question we categorize traditional dedup techniques as *symmetric caching* techniques where the `dd-src` and `dd-dst` maintain identical caches. We then introduce a new approach for dedup in mobile environments called *asymmetric caching*.

Fundamentally, asymmetric caching allows for the cache at the `dd-dst` to be larger than that at the `dd-src`. However, it enables the `dd-dst` to send feedback about portions of its cache to the `dd-src`. The feedback, sent in real-time, is selected to be pertinent to the ongoing traffic flow. The `dd-src` thus performs its operations not just based on its regular cache, but also based on the feedback received. The feedback selection problem is non-trivial because not only is the goal to increase redundancy elimination, but also to achieve a high feedback efficiency (ratio of bytes saved from transmission to bytes sent as feedback). Thus, the core of the asymmetric caching solution consists of an *application agnostic mechanism that can partition past and current traffic into contiguous byte sequences called* **flowlets** based on stationarity when the underlying byte stream is considered as a time series. Subsequently, feedback selection occurs by matching an arriving flowlet with a past flowlet, and then choosing content appropriately from the past flowlet to send as feedback. We elaborate on these mechanisms in Section 4.3.

Asymmetric caching achieves considerably better redundancy elimination by virtue of exploiting a much larger cache at the `dd-dst`. We show later with trace driven evaluations that asymmetric caching can increase redundancy elimination by over 100%, and provides such improvement even when the cache size at the `dd-src` is a fraction of that on the `dd-dst`. Furthermore, asymmetric caching achieves a feedback efficiency (ratio of bytes saved from transmission using feedback to bytes sent as feedback) of over $6X$. In other words, for every byte of feedback sent upstream, 6 *bytes* of downstream data are saved. In terms of adoption, asymmetric caching can be incrementally and independently deployed. A wireless service provider can thus deploy asymmetric caching to gain from all the past cached content accumulated on the mobile *without requiring any cooperation from other service providers the mobile might utilize*. Also, using prototype implementations of asymmetric caching on a laptop (Linux) and a smartphone (Android), we demonstrate that the CPU and memory overhead are quite reasonable.

In the rest of this chapter we introduce the concept of asymmetric caching for dedup in mobile environments. We also answer several questions that arise including the following: How is the feedback chosen to make dedup perform better? Are the dedup benefits with asymmetric caching significant enough to justify the cost of the feedback? How much does asymmetric caching improve performance over a symmetric caching solution in a mobile environment? What are the overheads of implementing asymmetric caching on standard mobile platforms?

4.2 Scope and Motivation

4.2.1 Scope

The focus of this work is to enable better wireless network performance through the use of improved network deduplication for mobile devices. The technical contributions of the work broadly apply to a variety of mobile devices and networks. Nevertheless,

we restrict the scope of the proposed work as follows:

- We specifically focus on **laptops** and **smartphones** as the mobile devices of interest, and **3G** and **WiFi** as the wireless technologies used at such devices.

- With regard to the wireless environment, since spectrum is inarguably more expensive in 3G environments, we primarily focus on 3G networks as the target environment for the proposed solution, but consider devices that consume content through both 3G and WiFi. However, our proposed solutions can be deployed in WiFi environments as well if required.

- While dedup can be deployed in an end-to-end fashion, we focus on a **last-hop layer 2.5 deployment** model for this work. In such a model, the dedup functionality is realized at entities on either side of the wireless link. While the mobile device is the only candidate deployment location for the `dedup-dst`, the `dedup-src` deployment location is likely to be a node such as the Serving GPRS Support Node (SGSN).[2] *In the rest of this dissertation, for brevity, we generically refer to the upstream dedup node as the* `dd-src`*, and the downstream dedup node as* `dd-dst`*.* Where the deployment location is relevant, we assume the SGSN as the deployment location for the `dedup-src`. However, the solution may be deployed in possibly other nodes (e.g., a dedicated dedup server) inside the wireless service provider's network.

- Finally, we restrict our focus in this work to dedup on the downstream and on only unencrypted traffic. Wireless traffic remains dominantly downstream and a significant portion of the traffic is not end-to-end encrypted. Hence, we believe that the contributions of this work will have significant impact in spite

[2]We consider the SGSN as the point of upstream deployment as opposed to the Gateway GPRS Support Node (GGSN) since it already performs per-user functions such as encryption. The `dedup-src` can be Packet Data Service Node (PDSN) in CDMA networks, or the Access Point for WiFi networks.

of these restrictions. We leave for future work the extensions of the proposed strategies for upstream and end-to-end encrypted traffic.

4.2.2 Motivational Scenarios

Traditional network deduplication solutions require the `dd-src` to rely only upon portions of the `dd-dst`'s cache that it is aware of. Such knowledge at the `dd-src` is implicitly accumulated when the corresponding data traffic flows through the `dd-src` en-route to the `dd-dst`. For static wireline hosts, such an arrangement is quite sufficient as the `dd-src` is always likely to be along the data-path to the `dd-dst`.

The basic premise of this work, however, is that for mobile devices using wireless connections, the `dd-src` is likely to be aware of only a fraction of the cache at the `dd-dst`. Thus, the `dd-src` is unable to perform deduplication to the fullest extent possible.

We now provide three scenarios in which the above *disconnect between the contents of the* `dd-dst` *cache and the* `dd-src` *cache* manifests itself.

- Multi-homed Devices: Most mobile devices today consume content through heterogeneous interfaces. WiFi is the preferred access technology when available due to its low cost and high data rate properties. However, 3G is the access technology used when users are not at locations with WiFi access. Recent studies of wireless data usage have profiled how both technologies are heavily used by mobile devices [51]. Moreover, cellular data offloading to WiFi is observed uniformly across both laptops and smartphones, and across different smartphone platforms [52].

  With traditional dedup, such data access offloaded to WiFi cannot be leveraged for redundancy elimination when the 3G interface is used, because the `dd-src` is different.

- Resource Pooling: Cellular providers have increasingly started to perform IP

core resource pooling that is part of the 3GPP standard. SGSN pooling is an example of this trend. In traditional GPRS networks, each SGSN, for example, is wholly responsible for its own service area. However, with SGSN pooling in 3G networks, all the SGSNs in the network work together, and the capacity load between them is distributed by the base station controllers (BSCs) and radio network controllers (RNCs). All BSCs and RNCs are connected to all SGSNs. Any mobile attached to the network is dynamically routed to the SGSN as per the current load distribution [53].

Thus, the specific SGSN that serves a mobile at a certain point in time does not need to be the same SGSN that serves the mobile at a different time. If traditional dedup were to be used, the `dd-src` at a subsequent SGSN will be unable to use the entire data cache at the mobile because it has no knowledge of the cache entries accumulated through a different SGSN.[3]

- Memory Scalability: With traditional dedup, the `dd-src` dynamically creates a complete date cache for each associated `dd-dst`. A single SGSN typically serves 100,000-1,000,000 simultaneously attached users [54]. Even if SGSN pooling were not to be performed, requiring the SGSN to maintain persistent state across the different attachment sessions for a mobile is thus quite prohibitive. Thus, if the cache state per user is maintained only during the lifetime of that attachment session, then all data accumulated through past sessions will go unused.

All of the above scenarios point to the need for an approach that allows the `dd-src` to leverage the full extent of the cache at the mobile device, even if it might not have prior knowledge of the entire cache. In the rest of this section, we outline any additional goals we want such a solution to satisfy.

[3]SGSN pooling does not involve state transfer between SGSNs.

4.2.3 Goals

One approach to address the above-discussed problem is to enable the `dd-src` to fully leverage the cache at the mobile device, and therein increase the dedup efficiency for the downstream communication. However, the following additional goals are critical for the design of such a solution:

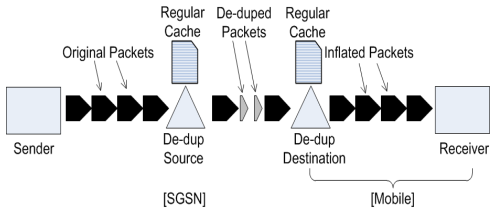
- Overall efficiency: While increasing dedup efficiency has the implicit result of helping in spectrum conservation by decreasing network load downstream, any solution has to be explicitly successful in using the overall spectrum (including both upstream and downstream) more efficiently.
- Application agnostic: Network deduplication is a generic technology that is application agnostic. Therefore, any solution to improve dedup has to remain application unaware, and hence applicable to any application used on the mobile device.
- Limited overheads: Both ends of the dedup solution (the SGSN or WiFi access point upstream, and the mobile device downstream) are resource constrained environments. Hence, any solution to improve dedup performance has to have deployable computational and memory complexities.

4.2.4 Background: Baseline Dedup

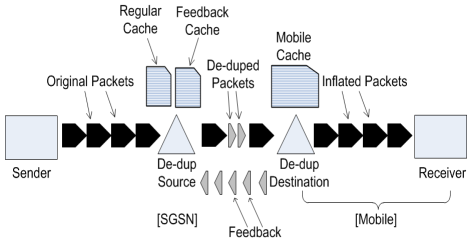
Network deduplication has been widely studied in related work, and in this work we use the well known byte-sequence caching as the baseline dedup technique. [10] and [11] first introduced the concept of network deduplication, while [25] and [24] studied characteristics of real network traffic to establish that there indeed exists considerable amounts of redundancy that can be eliminated. More recently, [8] and [9] have proposed techniques to leverage redundancies for traffic reduction in an end-to-end fashion and by overhearing content in wireless networks, respectively. The

byte-sequence based dedup is commonly used in both commercial WAN and storage optimization products (e.g., [50, 55]) and related research (e.g., [10, 25, 8, 11, 24, 9]).

Figure 7(a) depicts the operations of the byte-sequence caching. As shown in the figure, the byte-sequence caching algorithm essentially optimizes downlink traffic by replacing previously transmitted byte-sequences or segments of a packet with shorter hashes. Once the base-station receives a downstream packet destined to a mobile, it decomposes the packet into segments using Rabin Fingerprinting [56]. For each of the k segments of a packet, the hashes $[H_1, H_2, ...H_k]$ are computed using a known hashing algorithm such as Jenkins. If any of the hashes H_i is found in the cache, the corresponding segment in the packet is replaced by its hash. The resulting packet that includes both hashes and previously unsent data segments is then transmitted to the mobile. By virtue of the hashes being shorter, the load on the wireless link is reduced. At the mobile, the hashes for the data segments are computed and added to the hash table. Hashed segments are replaced with the corresponding original data before the packet is passed on to higher layers at the mobile.



(a) Baseline dedup (byte-sequence caching)



(b) Asymmetric caching

Figure 7: Baseline dedup vs. Asymmetric caching: Asymmetric caching is an overlay on baseline dedup.

4.3 Asymmetric Caching

In the rest of this chapter, we present *asymmetric caching*, a solution that satisfies the goals identified earlier. Asymmetric caching performs dedup on the unencrypted downlink network traffic from the `dd-src` (at the SGSN) to the `dd-dst` (at the mobile) as shown in Figure 7(b). It is built atop a baseline dedup algorithm such as the one described in Section 4.2.4. At a high level, asymmetric caching enables the `dd-dst` to send timely feedback to the `dd-src` about selected portions of its cache. The feedback should be such that the redundancy elimination efficiency, when the `dd-src` uses both its regular cache and the feedback, approaches that of a scenario where the `dd-src` has complete knowledge of the `dd-dst`'s cache. While we present the details of the approach in [12], we describe the key design elements in the rest of this section.

4.3.1 When is the feedback sent?

Asymmetric caching uses a *reactive strategy* for sending feedback. Feedback is sent upstream only when data traffic is flowing to the destination. The matching of arriving content with past data in the cache is explicitly used for the selection of feedback that is likely to be most useful. Such a reactive strategy for sending feedback improves the redundancy elimination efficiency in the downstream *while maximizing the feedback efficiency*. An alternative proactive strategy that sends feedback even during idle downstream periods might have a better redundancy elimination performance by virtue of being able to send a larger volume of feedback. However, such an approach will not fare well in terms of feedback efficiency.

4.3.2 Where from is the feedback chosen?

The `dd-dst` cache, in asymmetric caching, is partitioned into *flowlets*. Each flowlet is a contiguous subset of a byte stream. The currently arriving traffic is partitioned into flowlets, and any arriving flowlet, say $flowlet_{arr}$ is matched with one of the past flowlets in cache. The feedback is then selected from that past flowlet.

The concept of flowlets is motivated by the fact that most content arriving at the destination is a composition of a collection of objects. For example, an HTTP connection carries different objects such as JPEG images, HTML files, CSS scripts, etc.; an SMB connection carries independent blocks of data as per the scope and sequence of requests; and a peer-to-peer application connection carries different chunks of data, not necessarily contiguous, as requested by the receiver. Thus, a desirable approach for matching arriving traffic to past content would be to match the currently arriving object to an object in the past, and then select feedback from that past object. This would enable the selection of relevant feedback and hence will be favorable to feedback efficiency. While objects may be identified easily if application knowledge is used, such an approach would violate the goal to remain application agnostic. Instead, in asymmetric caching, flowlets are considered as approximations of underlying objects in the data traffic, but are extracted using purely application unaware techniques. We elaborate on the approach next.

4.3.3 How are flowlets extracted?

In keeping with the application agnostic goal of the design, asymmetric caching employs statistical segmentation to break each downstream flow[4] into *flowlets* without any knowledge of the application. Asymmetric caching relies on the stationarity of the content of individual objects when considered as a time series to perform the segmentation. Thus, changes in the statistical distribution of the underlying byte stream is used to identify flowlet boundaries. Each flowlet is then stored as a sequence of hashes at the `dd-dst` cache. For any $flowlet_{arr}$, the `dd-dst` selects hashes from the past flowlet that matches the most.

The flowlet segmentation used in asymmetric caching is a variant of the strategy originally introduced in [57]. The segmentation strategy in [57] segments a piecewise

[4]A flow consists of contiguous bytes/packets with same (source IP, destination IP, source port, destination port) tuple. A connection consists of an upstream and downstream flow.

stationary time series $\{X_1, X_2, ..., X_N\}$ into several separate time series $\{X_i, ..., X_j\}$, which are individually stationary. In asymmetric caching, the sequence of bytes in a flow is considered to be a time series, but the algorithm is simplified to grossly approximate position of the boundaries between the time series in return for a lowered computational complexity that is better suited for a resource-constrained mobile environment.

The approach takes a parameter l that is the minimum number of observations required to estimate reliable statistics of a series. At any given location $s(> l)$ in the series, three segments of the series are considered: segment from X_0 to X_s, segment from X_s to X_{s+l} and the aggregated segment from X_0 to X_{s+l}. An autoregressive (AR) model of order p is attempted to be fit on each segment, i.e.

$$X_i = \sum_{j=1}^{j=p} a_i X_{i-j} + \sigma\epsilon \tag{37}$$

on each segment, where ϵ is a white noise (error term).

The gain ($\sigma^2_{a:b}$) of the white noise for the best fitting model on segment $\{X_a, ...X_b\}$ is computed from the sample covariance matrix of that segment. Next, a distance value $d_{0:s:s+l}$ is computed as:

$$d_{0:s:s+l} = (s+l)\log\sigma^2_{0:s+l} - s\log\sigma^2_{0:s} - l\log\sigma^2_{s:s+l} \tag{38}$$

This is intuitively the extra power of the white noise (error) if the two segments are considered in one model as opposed to being in separate AR models. If this distance is more than a given threshold d_{thresh}, a boundary is said to exist between s and $s+l$. If a boundary is not detected, the next considered boundary is after l bytes. An empirical evaluation of the above solution, discussed later in Section 4.4.3, shows that when presented with a mixed-source traffic consisting of different object types such as JPEG, TXT, XML, etc., the object byte boundaries are indeed approximately identified.

Finally, the flowlet in cache that has maximum number of matching hashes with the arriving traffic is identified to be $flowlet_{match}$, the flowlet from where feedback is selected.

4.3.4 How is the feedback selected?

Once $flowlet_{match}$ is identified, the specific feedback to be sent is selected based on two parameters: the location of the last matching hash between the $flowlet_{arr}$ and $flowlet_{match}$ and the latency for feedback on the upstream. Specifically, the location of the last matching hash in $flowlet_{match}$ offset by δ hashes, where δ is the number of segments that the `dd-src` is likely to have transmitted before the feedback reaches, is used as the start point for the feedback. δ depends on the average segment size, upstream data rate, and downstream data rate; all parameters computable at the `dd-dst`. γ hashes are selected from the start point, aggregated into a single packet and transmitted. Note that to ensure that the feedback is always new, the mobile keeps track of all the hashes that have occurred in past downstream packets from this `dd-src` or have been sent upstream in the past. Such hashes are explicitly removed from any feedback.

4.3.5 How is the feedback used?

Finally, the `dd-src` maintains a *feedback cache* in addition to its regular cache. Structurally, the feedback cache is identical to the regular cache and consists of a list of hashes available at the `dd-dst`. However, the feedback cache is populated only with hashes received through explicit feedback from the `dd-dst`. When data arrives at the `dd-src`, each of its hashes is first looked up in the regular cache, and if there is no hit, the hash is added to the regular cache, but the same hash is then looked up in the feedback cache. If either of the cache lookups results in a hit, the hash is sent to the `dd-dst`. Otherwise, the original data segment is sent as-is. When a hash encounters a hit in the feedback cache, the hash is deleted after its first use since a corresponding

entry would have been made into the regular cache.

4.4 Solution Details

This section presents the details of the asymmetric caching solution. We first describe its operations at the `dd-src` and the `dd-dst` respectively, and then discuss system details for the solution including packet formats and software architecture.

Algorithm 5 Operations at the `dd-src`

Input: *in_packet* = Packet received
Variables:
regular_cache = Regular cache at `dd-src`
feedback_cache = Feedback cache at `dd-src`
out_packet = Packet to be sent
pkt_chunks = List of chunks
pkt_hashes = List of hashes
seg_hash = Hash of a single chunk
on_flow = Flow to which *in_packet* belongs
shim_hdr = 16-bit header: first bit tells if the following sequence is a hash or original text, next 15 bits are used to specify the length of the sequence
Functions:
hash(*chunk*) = Compute hash of *chunk*
rabinFingerprints(*string*) = Return value based chunks of *string*
dd-srcDedup(in_packet)

```
1: if in_packet is going to dd-dst then
2:     pkt_chunks ← rabinFingerprints(packet)
3:     for each chunk in pkt_chunks do
4:         if hash(chunk) in regular_cache or hash(chunk) in feedback_cache then
5:             out_packet ← out_packet + shim_hdr + hash(chunk)
6:         else
7:             out_packet ← out_packet + shim_hdr + chunk
8:             add hash(chunk) to regular_cache
9:         end if
10:    end for
11:    send out_packet to dd-dst
12: else[packet is coming from dd-dst]
13:    pkt_hashes ← hashes in in_packet
14:    for each seg_hash in pkt_hashes do
15:        add seg_hash to feedback_cache
16:    end for
17:    out_packet ← IP and TCP headers of in_packet
18:    send out_packet to upstream node
19: end if
```

4.4.1 Operations at the `dd-src` (SGSN)

The operations of the asymmetric caching at the `dd-src` can be explained in two parts: downstream and upstream (see Algorithm 5). For every downstream packet, asymmetric caching first divides the packet into value based chunks using Rabin Fingerprinting (line 2). These chunks are then deduplicated using the *regular_cache* and *feedback_cache* (lines 3 to 10). Specifically, if a matching hash is found in either of the caches, the original chunk is replaced with a shim header and hash of the chunk. Otherwise, it is replaced with a shim header and the original chunk. For every new chunk, its hash is added to the *regular_cache* for future use. The new packet is then sent to the `dd-dst`.

Next, for an upstream packet, if it carries feedback from the `dd-dst`, asymmetric caching extracts all the hashes and inserts them into the *feedback_cache* (line 14 to 16). If the upstream packet was a piggybacked packet, the packet stripped of the feedback is forwarded upstream.

4.4.2 Operations at the `dd-dst` (mobile)

The asymmetric caching algorithm at the `dd-dst` can be explained in terms of three functions: cache maintenance, cache organization, and feedback selection.

First, for cache maintenance, asymmetric caching at the `dd-dst` (`dd-dstDedup`) maintains a local cache, called `dd-dst`*cache*, that keeps chunks indexed by their hashes and all the flowlets seen thus far. When a packet is received from the `dd-src`, the `dd-dst` first reconstructs the original packet (line 7 to 14). The incoming packet is parsed into hashes and clear content by looking at the shim headers. The hashes are replaced by their corresponding chunks found in the `dd-dst`*cache*, while clear content is copied as it is, without any shim headers. The reconstructed packet is sent up the network stack. The `dd-dst` then hashes all the Rabin chunks of the packet to create a list *seg_hashes* (lines 16 and 17).

Algorithm 6 Operations at the `dd-dst`

Input: in_packet = Packet received

Variables:

d_thresh = Distance threshold to determine start of new flowlet

p = Order of AR() model to fit on the series

ar = Estimated coefficients of the p order AR model

win_factor = Region to be searched in the $past\_flowlet$

`dd-dst`$\_cache$ = Extensive cache at `dd-dst`

out_packet = Packet to be sent

on_flow = Flow to which in_packet belongs

id_count = Number of flowlets seen so far, used as flowlet id

$current\_flowlet[on\_flow]$ = Latest flowlet being created from on_flow

$hit\_count[flowlet][past\_flowlet]$ = Redundant bytes between current flowlet and an old flowlet in `dd-dst`$\_cache$

$parsed\_pkt$ = Mixed list of chunks and hashes in dedup packet

seg_hashes = List of hashes of chunks

adv_hashes = List of hashes to advertise

$last\_match[flowlet][past\_flowlet]$ = Pointer to last matching hash in $past\_flowlet$ for $flowlet$

$last\_adv[flowlet][past\_flowlet]$ = Pointer to last hash advertised from $past\_flowlet$ for $flowlet$

δ = Temporal offset to account for network delays

Functions:

$hash(chunk)$: Compute hash of $chunk$

$unhash(element)$: Fetch $chunk$ (from cache) whose hash is $element$

$rabinFingerprints(string)$ = Return value based chunks of $string$

gain(bseries)

1: $N \leftarrow len(bseries)$
2: covariance matrix $C \leftarrow [C_{i,j}]$, for $0 \leq i, j \leq p$, where $C_{i,j} \leftarrow \frac{1}{N-p} \sum_{k=p}^{N} bseries[k-i] * bseries[k-j]$
3: matrix $D \leftarrow [D_{i,j}]$, for $0 \leq i, j \leq p-1$, where $D_{i,j} \leftarrow C_{i+1,j+1}$
4: column vector $b \leftarrow [b_{i,0}]$, for $0 \leq i \leq p-1$, where $b_{i,0} \leftarrow C_{i+1,0}$
5: $\alpha \leftarrow D^{-1} * b$ ▷ estimate AR coefficients
6: vector $ar \leftarrow [1, \alpha_0, ..., \alpha_{p-1}]$
7: **return** $N * log(ar * C * ar^T)$

updateFlowlets(on_flow,packet)

1: **if** $series[on\_flow]$ is $null$ **then**
2: $series[on\_flow] \leftarrow packet$ ▷ First packet in the flow
3: $current\_flowlet[on\_flow] \leftarrow (++id\_count)$
4: **return**
5: **end if**
6: $d_{packet} \leftarrow gain(series[on\_flow] + packet) - gain(packet) - gain(series[on\_flow])$
7: **if** $d_{packet} > d_{thresh}$ **then**
8: $last\_flowlet \leftarrow current\_flowlet[on\_flow]$
9: $current\_flowlet[on\_flow] \leftarrow (++id\_count)$
10: link $last\_flowlet$ to $current\_flowlet[on\_flow]$
11: **end if**
12: $series[on\_flow] \leftarrow packet$
13: **return**

Algorithm 7 Operations at the `dd-dst`(continued)

subsequenceMatch(seg_hashes, flowlet, past_flowlet)

14: $num\_hashes \leftarrow len(seg\_hashes)$
15: **if** $last\_match[flowlet][past\_flowlet]$ is *null* **then**
16: $last\_match[flowlet][past\_flowlet] \leftarrow$ first hash in $past\_flowlet$
17: **end if**
18: $max\_seq \leftarrow$ hashes in seg_hashes found among $win\_factor * num\_hashes$ hashes after $last\_match[flowlet][past\_flowlet]$
19: $last\_match[flowlet][past\_flowlet] \leftarrow$ last hash in max_seq
20: **return** length of max_seq

bestMatchedFlowlet(flowlet,seg_hashes,old_flowlet_list)

1: **for** each $past\_flowlet$ in $old\_flowlet\_list$ **do**
2: Add $subsequenceMatch(seg\_hashes, flowlet, past\_flowlet)$ to $hit\_count[flowlet][past\_flowlet]$
3: **end for**
4: $best\_past \leftarrow past\_flowlet$ with maximum $hit\_count[flowlet][past\_flowlet]$
5: **if** $best\_past$ is $flowlet$ **then** ▷ Ongoing flowlet matches most
6: $top2 \leftarrow past\_flowlet$ with 2^{nd} most hits
7: **if** $hit\_count[flowlet][top2] \geq K\%$ of $hit\_count[flowlet][best\_past]$ **then**
8: $best\_past \leftarrow top2$
9: **end if**
10: **end if**
11: **return** $best\_past$

selectAdvertisement(flowlet,matched_flowlet)

1: $anchor \leftarrow$ later of $last\_match[flowlet][matched\_flowlet]$ and $last\_adv[flowlet][matched\_flowlet]$
2: $anchor \leftarrow \delta + 1$ hash after $anchor$
3: $adv\_hashes \leftarrow MTU/(2 * hash\_length)$ hashes after $anchor$ in $matched\_flowlet$
4: Remove all hashes from adv_hashes which have been seen downstream or sent upstream before **return** adv_hashes

Next, for cache organization, asymmetric caching uses the list of hashes, created and maintained by the above mechanism, to select relevant feedback for the ongoing flow. Specifically, the `dd-dst` first checks if a new flowlet has started in the current flow. This is done by the *updateFlowlets* module, which uses the segmentation approach presented in Section 4.3.3. It checks if a statistical boundary has occurred right before the current packet (line 6 and 7). If yes, a new flowlet is added in the current flow and the last flowlet of the flow is linked to it (line 8 to 11). The *gain* method is used to compute the power of white noise error term of any given series (i.e., sequence of bytes). The flow byte series is modified to remember the content of

Algorithm 8 Operations at the `dd-dst`(continued)

dd-dstDedup(in_packet)

```
 5: if in_packet is coming from dd-src then
 6:     on_flow ← (src ip, dest ip, src port, dest port) of packet
 7:     parsed_pkt ← Parse in_packet into chunks and hashes
 8:     for each element in parsed_pkt do
 9:         if element is a hash then
10:             out_packet ← out_packet + unhash(element)
11:         else
12:             out_packet ← out_packet + element
13:         end if
14:     end for
15:     send out_packet to the application above
16:     pkt_chunks ← rabinFingerprints(out_packet)
17:     seg_hashes ← list of hashes of pkt_chunks
18:     updateFlowlets(on_flow, out_packet)
19:     flowlet ← current_flowlet[on_flow]
20:     old_flowlet_list ← all the past_flowlet in dd-dst_cache in which any of seg_hashes
        have appeared
21:     Insert seg_hashes with flowlet in dd-dst_cache
22:     if old_flowlet_list is null then
23:         adv_hashes ← null
24:     else
25:         best_flowlet ← bestMatchedFlowlet(flowlet, seg_hashes, old_flowlet_list)
26:         adv_hashes ← selectAdvertisement(flowlet, best_flowlet)
27:     end if
28: else                                          ▷ packet is going to dd-src
29:     if adv_hashes is not null then
30:         out_packet ← in_packet + adv_hashes
31:     end if
32:     send out_packet to the dd-src
33: end if
```

this packet (line 12). In case this is the first packet, a new flowlet is created for the ongoing flow and the flow byte series is set to be the packet's content (line 1 to 5)

Once the `dd-dst` has updated the current flowlet in the flow, the `dd-dst` can select and advertise feedback (*feedback selection*) using *seg_hashes*. To reduce the complexity of searching in the cache, the `dd-dst`_cache maintains a mapping from each hash to a list of past flowlets in which it had appeared. Only the past flowlets that have seen any of the hashes in the current flowlet are considered for feedback. After extracting these, *seg_hashes* are inserted in the `dd-dst` cache along with the

current flowlet ID (line 21). Next, the *bestMatchedFlowlet* module takes the hashes of the packet and determines the number of hits seen in any past flowlet in the cache. The hit count for each past flowlet with the current flowlet is updated using the *subsequenceMatch* method. This method searches for the *seg_hashes* among $(win\_factor \times num\_hashes)$ hashes in the past flowlet after the last matched hash in that past flowlet (line 18). The algorithm remembers where the current flowlet has last matched with the old flowlet. After these updates, the *bestMatchedFlowlet* selects the old flowlet with overall maximum hits for feedback. In case the best matching flowlet is the current flowlet itself, the module selects the second best matching flowlet that has $K\%$ of the maximum hits (line 5 to 9). Feedback is then selected from this old flowlet by the method *selectAdvertisement*.

The *selectAdvertisement* method keeps track of the last hash that matched between current flowlet and any old flowlet and also the last advertised hash for the pair. For the best matching flowlet, it first determines the later of these two pointers and then jumps δ hashes after that. This δ is the temporal offset to account for the feedback delay incurred by the underlying network. The larger the feedback delay, the more the offset must be for the feedback to be relevant when it reaches the `dd-src`. In our implementation we derive δ from the uplink and downlink data rates seen at the `dd-dst`. After including the temporal offset, the `dd-dst` selects $MTU/(2 * hash\_length)$ hashes to advertise to the `dd-src`. We choose these many hashes as the minimum chunk size created by our *rabinFingerprints* method is $2 * hash\_length$, so $MTU/(2 * hash\_length)$ is the maximum number of chunks expected in a downstream packet.

This feedback is further optimized by removing from it all hashes that have occurred in the downstream or have been advertised upstream by the `dd-dst`. Note that if there are no hits in this packet, no feedback is generated (line 22 of `dd-dstDedup`).

The feedback to the `dd-src` is opportunistically piggybacked on upstream data packets, e.g., TCP ACKs. If the `dd-dst` has some feedback to send in the form of hashes, it inserts these hashes into the payloads of upstream packets (line 30 in `dd-dstDedup`) and sends the packet upstream. If upstream data packets are not pending to be transmitted, a custom packet is constructed and transmitted.

4.4.3 Related Issues

Although the detailed algorithms presented above have established key insights for asymmetric caching, there are several issues associated with the design choices of the algorithms.

- *Flowlet segmentation*: We evaluate the performance of the flowlet segmentation algorithm in isolation. We take a set of 25 HTTP connections from one user, each containing one or more objects. The absolute object boundaries are identified through manual analysis of each connection in Wireshark by looking at HTTP GET requests and responses. This gives the actual number of objects; a total of 62 objects across 25 connections. Next we run the segmentation algorithm on each connection and record (i) number of flowlet boundaries detected and (ii) number of flowlet boundaries matching real object boundaries. For the 62 objects, our flowlet segmentation detected 56 objects. Out of these, 36 flowlet boundaries were found at the same location as the object boundary. Thus 60% of the flowlets were accurately detected. The algorithm does over-segment in some connections, here $56 > 36$. But over-segmentation does not affect the performance of *asymmetric caching* as when the best matched flowlet is not long enough to fill a packet of feedback, more feedback is selected from the following flowlet, which is the same object in the case of over-segmentation.

  Additionally, the algorithm is unable to detect a flowlet boundary in two cases. The first case is when there is exactly one object in a flow, which does not

Table 5: Performance of hash algorithms in collision handling

Hash algorithm	Digest size	Data set size	Collisions	TCP checksum detection
SHA1	20B	5GB	0	N/A
MD5	16B	5GB	0	N/A
Jenkins-8B	8B	5GB	0	N/A
Jenkins-4B	4B	5GB	0.02%	100%

impact performance. The second case occurs when all the objects in the flow belong to the same statistical source, e.g., all text files or all GIF images. Among the 62 objects being evaluated here, 15 object boundaries were missed due to this case. However, our macroscopic results show that this does not impact the overall results. Improving the segmentation further is part of our future work.

- *Hash function selection*: We use Bob Jenkins hash algorithm to create 8B hashes of the packet chunks on the `dd-src` and `dd-dst` [58]. The choice of hash is based on two conflicting goals: desire for more bandwidth savings by reducing the number of bytes sent on the network and minimum collision rate so that packets are not corrupted during dedup. Popular hash functions such as SHA1 and MD5 are typically computationally heavy and the digest size is large. We compare the aforementioned hash algorithms and the two versions of Jenkins hash (a 4B digest size and an 8B digest size) to hash the Rabin fingerprints generated over $5GB$ of traces. As shown in table 5, we observe no collisions with SHA-1, MD5 and Jenkins 8B hash, but 0.02% collision rate with Jenkins 4B hashing. Jenkins 8B provides a good trade-off between bandwidth savings and collisions, making it our choice for the implementation. Jenkins hash has also been used in prior works on network dedup [8, 9]

- *Handling hash collisions*: We use TCP checksum to detect hash collisions. A TCP checksum is computed on the header and the payload of a packet. As we do

not mangle TCP headers, the checksum is sent unchanged. After reconstructing the original packet from a deduped packet the `dd-dst` device checks to see if the checksum matches. If not, it sends a control message upstream to the `dd-src`, requesting it to delete all the hashes that it had sent in the corresponding packet. The `dd-dst` also includes the hashes in the upstream packet. In our hash algorithm evaluation, we also apply this detection test to all cases. While the test was not invoked with SHA1, MD5 or Jenkins 8B, it was invoked by the Jenkins 4B hash. Table 5 shows that TCP checksum was able to detect 100% of the collision events.

- *Cache management*: Both the `dd-src` and `dd-dst` have finite cache space. Thus, asymmetric caching uses an LRU cache eviction policy. At the `dd-src`, LRU runs independently on the regular cache and the feedback cache, and evicts the least recently used hashes in each cache. As the `dd-dst` cache is organized in flowlets, the LRU on the `dd-dst` evicts least recently used flowlets at every run. All the state maintained for that flowlet is removed. The hashes (and chunks) seen in that flowlet are also removed unless they have also appeared in some other flowlet, which is still in the cache. A hash (and the original chunk) is evicted once the last flowlet referencing it is removed from the cache.

4.4.4 System Architecture

In the rest of this section we present a system architecture for asymmetric caching that is detailed in Figure 8. As shown in the figure, asymmetric caching (AC) works at layer 2.5 on the `dd-src` and `dd-dst`.

- *Dedup source:* This module (shown in the dotted-line box in Figure 8 (a)) captures the downstream packets at the base-station. The captured packet is then broken into chunks using Rabin Fingerprinting and hashes of each chunk

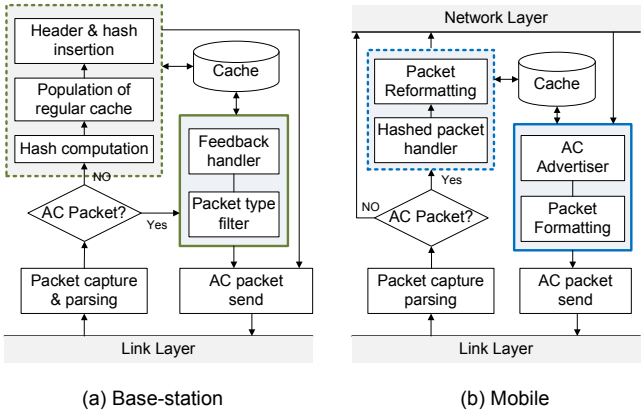


Figure 8: Software prototype of asymmetric caching: It consists of `dd-src` at 2.5 layer of the base station and `dd-dst` at 2.5 layer of the mobile.

are searched in the regular and feedback cache. If a matching hash is found, that chunk is replaced with its hash in the packet. If any packet is modified from its original, the IP options value is changed to reflect the same. We consider transport layer payload for dedup. Inside the transport layer payload, 2B[5] shim headers are inserted before every individual chunk and hash to demarcate original content from hashed content. Shim headers are not added if the entire packet is to be sent as original. This modified packet is then inserted back in the stack to be routed out.

- *Dedup receiver:* At the mobile, the dedup packet is received by the receiver module (the dotted-line box in Figure 8 (b)). This module takes care of inflating the packet into its original form, updates flowlets in the caches, and selects the hashes to be advertised. It then passes the reconstructed packet to the higher layer (i.e., network layer).

- *Feedback source:* This module on the mobile device (the thick solid line box in Figure 8 (b)) is responsible for getting the hashes chosen by the dedup receiver

[5]The first bit is set if it is a hash value and the rest of the bits indicate the length of the following chunk/hash

and piggybacking the hashes on the next upstream packet. This module captures an outgoing packet, using Netfilter[59], modifies the IP options field and adds feedback to the packet. This packet is then inserted into the stack to be routed out.

- *Feedback receiver:* At the base-station, the feedback receiver (the thick solid line box in Figure 8 (a)) captures upstream IP packets with header options set. It then strips off the feedback from the payload, restores the original header and forwards the new packet to upstream nodes. The extracted hashes are inserted into the feedback cache at the base-station and used for further network deduplication.

4.5 Performance Evaluation

We evaluate asymmetric caching via trace-based analysis of real network traces. We first explain the trace collection environment, and then describe the trace analysis methodology. Finally, we present trace-based evaluation results as well as prototype-based experimental results.

4.5.1 Collecting Network Traffic

We use real network traffic collected from 30 different mobile users (volunteers), 5 of whom are smartphone users and the rest are laptop users. We use these traces for performing the trace-based evaluation. Below are the details of the trace collection process.

- *Connectivity:* The laptop users relied only on WiFi connectivity for their network access. The smartphone users relied on both WiFi and 3G connectivity. The data collection spanned a period of 3 months and yielded over 26 *Gigabytes* of unsecured downlink data. Since we do not require any change in the user

access pattern for the trace collection, users accessed the Internet as per their normal behavior.

- *Devices and tools:* The laptop users ran Windows 7 and Linux operating systems and used Wireshark to collect their traces. The smartphone users used the Samsung Vibrant Galaxy phone and the HTC G2 phone, both running the Android 2.1 operating system on the T-Mobile network and relied on Tcpdump for trace collection. Users were able to parse trace files and remove any sensitive information before submitting them for analysis. Only unencrypted traffic was used in the analysis.

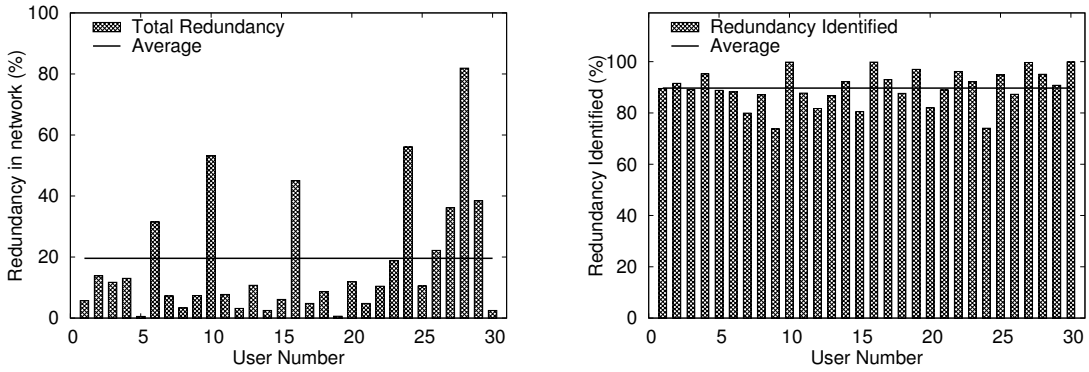
- *User demographics:* The volunteers included full-time employees at an industry research lab and an enterprise, as well as graduate students at a large university campus. The users span the age group of 21 to 50 and were spread over two different geographic regions.

4.5.2 Analysis Methodology

We use a custom trace analyzer to operate on the above traces. The analyzer models components of asymmetric caching presented in Section 4.3 and is configured and used for analysis, as follows:

Caches: The analyzer maintains three caches in the form of hash tables (i) `dd-src` regular cache (ii) `dd-src` feedback cache and (iii) `dd-dst` cache. It also maintains additional data structures required by the asymmetric caching algorithm at the `dd-dst`. We set the default `dd-src` cache size (i.e., regular + feedback) to 1MB and the default `dd-dst` cache size to 250MB. We explicitly study the sensitivity of the solution to cache sizes later in the section.

Past and Present Trace: To emulate the temporal history of the traces, the packet trace for each user is split equally into a past trace and a present trace (e.g., a 40MB trace was split into a 20MB *past* and a 20MB *present*. Then, the past trace is used



(a) Total network redundancy

(b) Redundancy identified

Figure 9: Identifying network redundancy: (a) shows there exist network redundancy (avg., 19.6%) and (b) shows asymmetric caching finds most of network redundancy (avg., 89.7%).

as an input to the analyzer to populate the initial cache at the `dd-dst`. This is the memory collected at the `dd-dst` without the knowledge of the current `dd-src`. Next, in the present trace, a set of 30 connections is randomly selected and used for the second set of inputs to the analyzer.[6] We use a minimum threshold of 5KB for the size of connections under consideration to avoid very small (redundant) connections and to filter out insignificant connections from the analysis.

Metrics: Given the above set-up, we monitor three values for each user: (1) redundant bytes found in the `dd-src`'s regular cache, (2) redundant bytes found in the `dd-src`'s feedback cache and (3) total number of unique hashes sent as feedback from `dd-dst` to the `dd-src`.

Comparison: We also implement and run the byte-sequence caching algorithm (with average segment size of 128B) on a merged trace of the *p*ast and the 30 connections from the present for each user. This represents the scenario where the `dd-src` has all the hashes that the `dd-dst` has ever seen and thus gives a measure of 'ideally' achievable dedup.

[6]We have observed that the trend of redundancy is similar even when considering the entire present trace for each user.

4.5.3 Evaluation Results

4.5.3.1 Identifying Network Redundancy

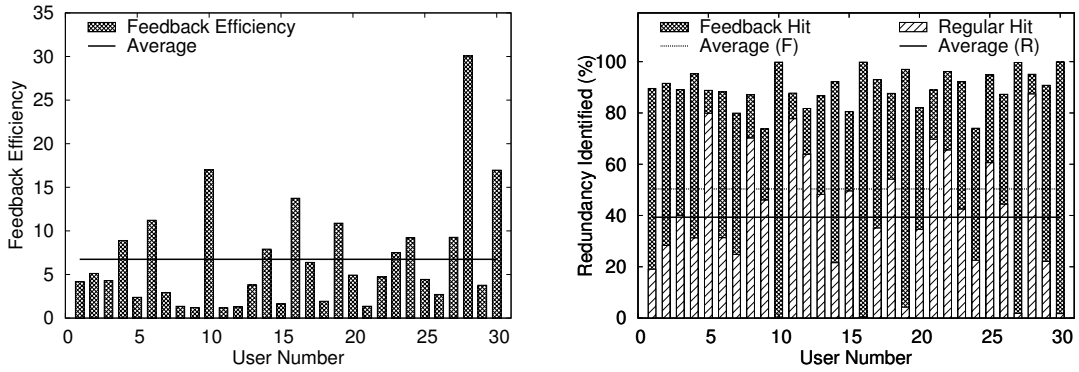
We first show how much network redundancy exists in the collected traces and how much of that redundancy can be identified by asymmetric caching. Figure 9(a) plots total network redundancy that exists in the 30 user traces. Here, the total network redundancy is defined as the number of total cache hits at `dd-dst` for each chunk of a received packet, if the chunk exists in local cache (cache hit), we count the chunk as redundant bytes. As shown in the figure, there is indeed network redundancy of 19.6% on average.

Next, Figure 9(b) shows the percentage of the redundant bytes identified by asymmetric caching (found in Figure 9(a)). Specifically, we measure the number of cache hits at a `dd-src` based on both regular and feedback caches and use it for the redundant bytes identified. As shown in the figure, the asymmetric caching is able to identify on average 89.7% of the total redundancy, a considerable fraction of which is attributable to the feedback (see next subsection). Also note that its variance is small across 30 different users, owing to our fine-grained and adaptive advertisement scheme.

4.5.3.2 Feedback Efficiency

In this section, we present the feedback efficiency of asymmetric caching. Figure 10(a) plots the ratio (λ) of the redundant bytes identified to every feedback byte. If λ is greater than 1, asymmetric caching's feedback is effective in finding redundancy, and vice versa. As shown in the figure, the average λ value over 30 users is 6.74. One interesting observation is that the higher a user shows mobility (i.e., smartphone users including user 10, 28, 30), the higher λ is.

Next, we further study how much of the redundancy elimination is attributable to the feedback cache versus the regular cache. Recall that the redundant bytes are



(a) Redundancy to feedback

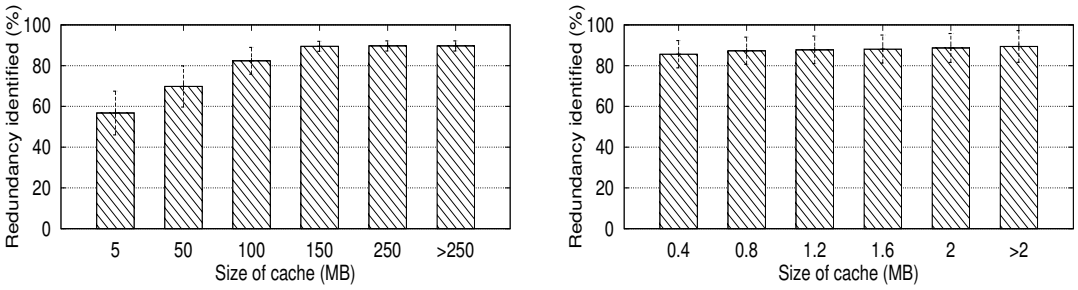
(b) Feedback hits vs Regular hits

Figure 10: Feedback Efficiency: (a) shows the ratio of the total redundancy identified to the feedback bytes (λ), and (b) shows how much each cache (feedback and regular) contributes to the redundancy detection.

identified by searching its regular cache and then, if not found there, the feedback cache. For this, we analyze the redundancy found using only the feedback cache (F) and the redundancy found using only the regular cache (R). Figure 10(b) shows relative contribution from both R and F for each user. As shown in the figure, feedback cache largely contributes to 50.35% of the redundancy elimination, whereas the regular cache contributes to 39.3% of the redundancy elimination. Note that the hits in the regular cache can be considered as an indication of the performance of conventional dedup that relies only on symmetric caching. Hence, this result shows that asymmetric caching can improve the performance of dedup significantly.

4.5.3.3 Sensitivity to Cache (Memory) Size

In this section we measure the sensitivity of asymmetric caching to cache size. In this experiment, we first fix the cache size of the `dd-src` to 2MB. Then, while we increase the cache size of the `dd-dst` (from 5MB to >250MB), we analyze the percentage of redundancy identified by the asymmetric caching. We also analyze the opposite setting to study the sensitivity of `dd-src`'s cache size on its performance (250MB of `dd-dst` cache and from 0.4MB to >2MB of `dd-src` cache).



(a) Sensitivity to cache size at `dd-dst`

(b) Sensitivity to cache size at `dd-src`

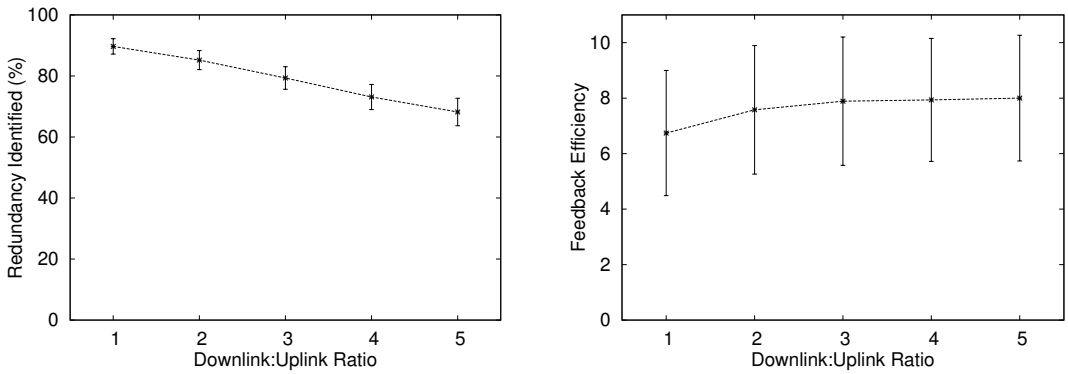
Figure 11: Sensitivity to cache size: (a) shows that asymmetric caching can identify 89% of redundancy by using 150MB on the mobile device. (b) shows that the asymmetric caching requires only a small cache size at the `dd-src` (e.g., ∼1MB) to achieve 85% of the redundancy detection.

Figure 11 shows the results of the both experiments. First, as shown in Figure 11(a), given the constant size of cache at the `dd-src`, the redundancy identified by asymmetric caching increases with the increase in `dd-dst` cache size. This trend is a result of fewer cache evictions when using larger caches. In addition, with only 150MB of cache size at `dd-dst`, asymmetric caching is able to identify 89% of the redundancy.

Next, as shown in Figure 11(b), even with a small cache (e.g., ∼1MB) at `dd-src`, asymmetric caching is able to identify more than 85% of redundancy. Finally, looking at the both figures in Figure 11, we can observe that for a 1:100 ratio in the cache sizes (e.g., ∼1MB at the `dd-src` and 100MB at the `dd-dst`), asymmetric caching effectively detects and leverages redundancy. Furthermore, this ratio supports the design goal of asymmetric caching—the use of large cache at the `dd-dst` with a small cache requirement at the `dd-src`.

4.5.3.4 Performance under varying data-rates

So far we have assumed that the uplink and downlink data-rates are same, i.e. for every byte downstream, the mobile sends a byte of feedback upstream. In this section, we analyze the performance of asymmetric caching when the downlink and uplink



(a) Redundancy leveraged

(b) Feedback efficiency

Figure 12: Performance with varying data-rates: (a) shows the % redundancy identified for increasing downlink to uplink data-rate ratio, and (b) shows the feedback efficiency in each case.

data-rates are asymmetric. We vary the ratio of downlink to uplink data-rates from 1 through 5 and monitor the percentage of redundancy leveraged and feedback efficiency in each case. The results are shown in Figure 12.

We observe in Figure 12(a) that as downlink rate grows to 5× the uplink rate, the redundancy identified by asymmetric caching goes down to 68%. Interestingly, for the same ratio, the feedback efficiency grows to 8×, as shown in Figure 12(b). This shows that when asymmetric caching is restricted to send less upstream feedback, the redundancy elimination performance drops modestly and the feedback is more efficient.

4.5.3.5 Application-agnostic dedup

Finally, we measure how effectively asymmetric caching works for different types of application traffic. Recall that one of the design principles of asymmetric caching is to remain application agnostic— not require application information— but provide dedup performance regardless of application types. To this end, we analyze the trace of 30 users to calculate the percentage of redundancy identified by asymmetric caching over total network redundancy for each application. We classify the applications based

on the port numbers used for the connections.

Table 6: Application-agnostic redundancy identification

Applications	Port Numbers	Redundancy Identified (%)
HTTP	80	98.76
ICSLAP	2869	80.48
Android Market	5228	53.10
McAfee, HP, SAP	5555	82.13
P2P and others	Ephemeral	97.70

Table 6 shows the performance results of asymmetric caching under different types of applications. As shown in the table, the asymmetric caching is able to identify 53% to 97.76% of the network redundancy and it does not have radical performance penalty for specific types of application. This clearly supports our design goal of application-agnostic feature.

CHAPTER V

PRECOG: SMART PREFETCHING FOR MOBILE DEVICES

5.1 Introduction

The adoption of mobile devices such as smartphones has reached a significant threshold with the number of such devices shipped now surpassing the number of PCs shipped [1]. Nearly 40% of Internet time is now attributed to mobile devices such as smartphones and tablets [2]. Not only are consumers adopting mobile devices at a blistering pace, but such adoption is being witnessed within the traditionally conservative enterprise sector as well. 71% of enterprises are currently deploying or planning the deployment of mobile applications [3]. Such adoption amongst enterprises is driven by a clear return-on-investment from mobility in the form of higher employee productivity, reduced paper work, and increased revenue [4]. *An ever-increasing pressure on wireless network performance and scalability, however, accompanies such exciting trends.*

Cellular technologies such as 3G and 4G [1], in tandem with WiFi, serve as the fundamental access mechanisms for mobile users. Recent studies show that the availability-cost trade-offs of the aforementioned technologies results in users relying heavily on both for data-access [52]. While WiFi operates in the ISM band, and hence is free to deploy, cellular providers purchase cellular wireless spectrum at FCC auctions and pay billions of dollars for licenses. The recent FCC auctions for spectrum in the 700MHz band resulted in approximately 50MHz of spectrum purchased

[1] All our discussions apply to both 3G and 4G networks. For brevity we refer to both these technologies as *cellular* in the rest of this chapter.

for close to $20 Billion [60]. Thus, wireless service providers are heavily motivated to continually improve efficiencies of how such spectrum is used.

One approach to achieve this is to shift the load from expensive cellular network to much cheaper WiFi networks. Mobile devices today are almost always equipped with heterogeneous interfaces with each interface having distinct cost-bandwidth-connectivity trade-offs. WiFi is low-cost and high bandwidth, but is not always available for use; whereas cellular is high-cost and low bandwidth, but is almost always available. Mobile users have access to unlimited data over WiFi but monthly limits are imposed on data over cellular network. Over WiFi, every gigabyte of data costs around twelve cents[61], while over cellular it can be over ten dollars[62]. The costs have an order of two difference. We call this the *unbalanced cost problem*. Previous works have considered WiFi offloading to shift ongoing cellular traffic on to WiFi[63, 64]. But these solutions rely on concurrent availability of both cellular and WiFi networks. The more prevalent scenario is when user connects to WiFi in some locations, e.g., home, office, coffee shops and cellular in other places, e.g., transit, client offices, stores. In this context, we try to answer the question: How can time-shifted access over the cheaper (WiFi) network be leveraged more effectively to reduce cost of the more expensive (cellular) network access?

We answer this question through *Precog*, *a name-independent network-aware prefetching* solution for smartphones and tablets. Prior prefetching solutions proposed for wired scenarios [13, 14, 15] are (i)*network agnostic*, i.e. prefetching is done irrespective of the network in use, (ii)*name-based*, i.e., they remember the exact webpage, identified through its uniform resource locator(URL), accessed in the path and only prefetch a subset of those. However, with the advent of multi-homed devices and the web becoming increasingly more dynamic, such solutions severely under perform. *Precog* addresses these deficiencies through *action-based prefetching*. Briefly, *Precog* remembers not the exact URL accessed in the past, but the actions performed on a

particular website. We argue that even though web content keeps changing frequently, there is consistency in the layout of websites and the actions of any given user on these websites. We leverage this consistency in designing *Precog*. Additionally, *Precog* enables the mobile device to cache over WiFi the content user is most likely to access over cellular network in the future. We evaluate *Precog* over real web content fetched through synthetic user and network traces and show that *Precog* can give 47% byte savings over cellular network while using 26% of the bytes prefetched over WiFi.

5.2 Motivation

5.2.1 Limitations in Existing Prefetching Solutions

Prefetching is predictive fetching of content that a user is likely to access in the future. Traditionally, prefetching has focused on reducing the latency of web access. However, with the unbalanced cost of network access on mobile devices, prefetching can be used to offload traffic from cellular to WiFi network. Specifically, a prefetching module can predict what the user will access over cellular network and fetch it in advance while the user is on WiFi.

Prior works on prefetching[13, 14, 15] propose *name-based prefetching*, which relies on the names, in the form of Universal Resource Locators(URLs), of web content to decide what to prefetch. Specifically, *name-based prefetching* remembers users' browser history in the form of URLs visited in the past. It then selects the URLs most likely to be accessed again, based on: (i) content triggered prefetching [13, 14], where the URL fetched most, subsequent to the current URL, in the past, is fetched again, or (ii) time triggered prefetching [15], where most accessed URL at given time of day is prefetched every day. All these solutions are based on the assumption that users access the exact same URL repeatedly and content on Internet does not change very often. We argue that name-based prefetching, as proposed in prior works, cannot be applied as-is to offload data from cellular network to WiFi. In this section we list

the limitations in name-based prefetching:

- *Network unawareness*: The first and foremost limitation in name-based prefetching is its network unawareness. Prefetching is triggered by content [13, 14] or time [15], but not by change in network, as these solutions were focused on wired environments where network heterogeneity does not exist. With the advent of ultra-mobile devices, network heterogeneity has become more pronounced. Additionally, name-based prefetching is not *time-shifted*. Content is prefetched for short-term consumption and not for an hour later.

- *Dynamic Web*: The web is becoming increasingly more dynamic. Websites keep updating content to incorporate latest trends and preferences. Content management systems[65, 66, 67, 68] for the web provide a scalable approach for web management, but entail dynamic naming strategies. Simple name-based prefetching will be ineffective on dynamic web as it relies solely on object names for prefetching.

  End users also access similar but not the exactly same content, e.g., a user might read headline on *www.nytimes.com* everyday. While the headline is the main news everyday, the exact URL for the headline may change. A traditional prefetching solution will try to fetch the same URL everyday but the headline page keeps changing everyday, or even every hour. In such scenarios, the name (URL) based prefetching solutions are ineffective.

  To support our observation we analyzed the number of URLs changing over time for nine popular mobile websites[69], for a week. Using *wget* we downloaded all the HTML files on each website, up to second level of reference, every six hours, giving four snapshots per website per day. The twenty-eight snapshots are labeled $S0, S1, ..., S27$. The $user - agent$ parameter of *wget* was set so that only mobile versions of the websites are fetched and a cookie file is also supplied

for accessing websites which require logging in, such as Facebook, Twitter, LinkedIn and Pandora. All the downloaded HTML files are parsed to extract the URLs referred within *href* tags. We then compute the number of new URLs added in a snapshot as a percentage of URLs in the previous snapshot. The results for twenty-eight snapshots for all the nine websites are shown in Figure 14(a). We observe that there is high variance in the changes per snapshot. On an average 8% to 88% of the URLs referred to on a webpage changed every six hours over the week.

While the previous analysis showed that websites undergo changes, we also did an analysis of user accesses across different websites. We consider the LiveLab dataset [70], which is collected from 24 volunteers in Rice University using iPhone 3GS from Feb 2010 to Feb 2011. The LiveLab software records several components of a user's mobile activity such as phone state, list of running applications, web browser history, etc. We only consider the web browsing data set from this collection. This data set contains web browser history collected from Safari browser every night. Each record is a timestamp and a hashed URL. The URLs are hashed piecewise so that the domain names can be separated from the file paths on the servers. For example, a URL given as:

```
http://username:password@www.example.com:80/over/there/index.dtb;
parameters?type=animal&name=narwhal#nose
```

is hashed to:

```
http://hash(username):hash(password)@www.hash(example).com:80/
hash(over)/hash(there)/hash(index.dtb);hash(parameters)?hash(type=
animal&name=narwhal)#hash(nose)
```

For each user, we compute the percentage of URLs which occur more than once in the trace. Additionally, we also compute the percentage of domain names,

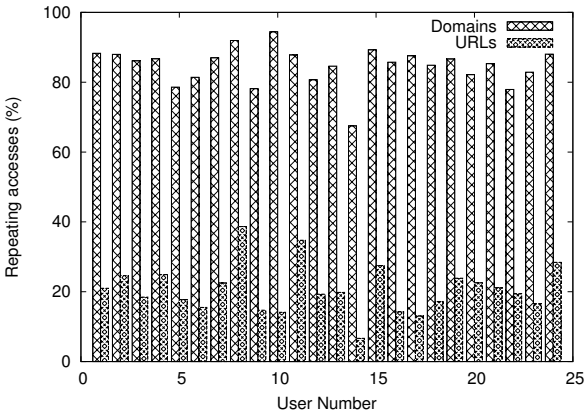
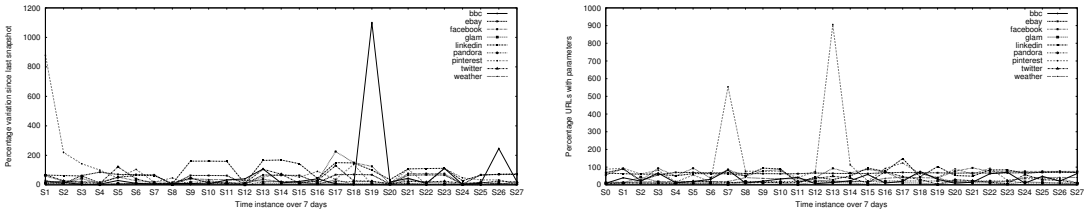


Figure 13: Domains vs URLs repeatedly accessed by LiveLab users.



(a) Percentage change in URLs referenced on a website

(b) Percentage change in URLs with parameters on a website

Figure 14: (a) Dynamism observed on nine popular mobile websites. (b) URLs including parameters on popular mobile websites.

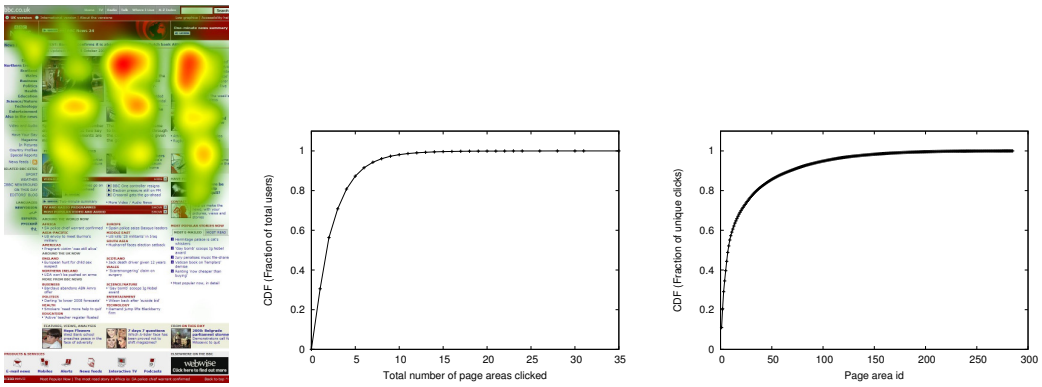
i.e. `www.hash(example).com`, that occur more than once in the entire trace. Figure 13 shows the results for all 24 users. We observe that, on an average, 81% of domain names are repetitive, while only 21% of the URLs repeat in the year long trace. This further strengthens our claim that users access similar but not exactly the same content and a pure name-based prefetching will not be efficient in such scenarios. Hence the need for a $name-independent$ prefetching approach.

- *Client-side logic*: Over the past decade, applications have evolved such that logic is driven not just by server, but also by client-side technologies such as AJAX[71], Flash[72], JavaScript[73], etc. Web has become more interactive and response time is significant in defining performance. Thus technologies like

AJAX, which allow for asynchronous updates of webpages, have been adopted in web applications. Additionally, richer applications need non-standard technology to perform complex tasks, e.g., Flash, Java-applet[74], ActiveX[75] which have become popular in the past decade. Today's collaborative web uses complex user data objects which are sent to servers, further necessitating sophisticated client-side logic. A prefetching solution which runs a simple HTTP GET request for a single URL cannot incorporate newer client-side logic governed by new technologies.

To further motivate this, we further analyze the website snapshots discussed in Figure 14(a). Among all the URLs referenced in each snapshot we identify the URLs which contain some parameters by looking at the occurrence of the delimiter '?'. We compute the number of such URLs as a percentage of the total URLs seen in that snapshot. Figure 14(b) shows the results for all the snapshots. It is interesting to note that on an average 11% to 80% of URLs on a page contained some parameters. This serves as one example of how client-side parameters are used to generate content in today's web.

- *Personalized access*: In addition to web dynamics and newer technologies, content is becoming increasingly personalized on the web. One form of personalization is authentication. Several websites, such as Facebook, YouTube, etc, require username and password information to provide per user customized content. Name-based prefetching is inefficient in such cases. The prefetching algorithm needs to remember the state in which the user accessed a URL and recreate that state on the web server to get access to the new URL. This can be through (i) form based authentication, where user explicitly enters the username and password into a form on a login page or (ii) complex authentication, where the user automatically logs into a website given she is already logged into another website, e.g., opening Google calendar in a new tab while being logged



(a) Heat map of BBC homepage[76] (b) CDF of areas per user on microsoft.com (c) CDF of clicks per area on microsoft.com

Figure 15: (a) Heat map of user eye movement on BBC homepage (b) Number of page areas clicked by users over a week (b) Number of clicks on each page area

into Gmail in another tab or visiting LivingSocial while logged into Facebook. A stateless prefetching approach cannot prefetch such content.

Traditional prefetching approaches fail to perform in the above four setups as they try to leverage consistency in URLs accessed across users. While this consistency does not exist in the web today, consistency still exists on other fronts. We next look at which other consistencies can be leveraged for efficient prefetching in the above mentioned scenarios.

5.2.2 Case for an Action-Based Prefetching Solution

The above results show that there is very little consistency in URLs. Thus, *Precog* proposes to leverage consistency on a different front to perform prefetching. *Precog* builds on the fact that despite the change in URLs, website layouts stay consistent. By the usability principle of consistency[77], web site layouts should follow same design templates, despite changes in the content, to ensure high usability of the website. Figure 15(a) shows a heat map generated by tracking users' eye motion over BBC home page[76]. It shows that users focus more on certain areas (red patches) than other areas (green patches) of the page. Website designers follow these guidelines to organize their content. This shows that even when the exact content of a web domain

changes, it is presented in a consistent fashion.

In addition to consistency in web layout, users are also consistent in their actions. They visit same domains looking for similar content repeatedly. To motivate this, we study click patterns on *www.microsoft.com* for 38000 users over a span of one week. The data set has been provided for public use by Microsoft[78]. This data set monitors the area of the webpage where each user clicks during the week. Each area is identified by the name of the section on the website, e.g., 'Free Downloads', etc. Figure 15(b) presents a cumulative density function (cdf) for the total number of areas clicked per week by each user. The graph shows that *over 80% users clicked on less than five areas* on the webpage and *98% users viewed less than ten areas of the page*. Another dimension we analyze on this dataset is the popularity of different areas of the page. From all the unique clicks in the dataset, we compute the cdf of the number of clicks per page area, which are shown in Figure 15(c). We notice that among 294 page areas on the website, less than 50 page areas get 80% of the clicks. Thus 17% of the website components get 80% of the clicks.

This shows that users are consistent in their activity on a website. At the same time, the LiveLab results shown in Figure 13 show these actions do not always lead to same URLs. This shows that while there is limited consistency in URL names, there is consistency in the user activity. Thus, *consistent user actions can be learned from the past and applied to consistent content layouts to predict dynamic future content.* This forms the core idea of *Precog*.

5.3 Precog

In this section we propose *Precog*, an action-based approach to prefetching. At a high level, *precog* records user actions on web domains and replays them to identify what content to prefetch. *Precog* motivates from the fact that users access dynamic content and hence name based prefetching is not effective. Further, given the consistency in

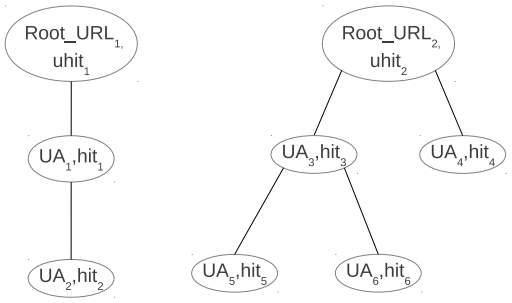


Figure 16: Forest of session trees to organize user action history

content layouts and user actions, consistency can be leveraged at the granularity of actions. It thus tries to learn user actions on a web domain, while naturally accessing it both over cellular. It then uses this knowledge, while the user is on WiFi, to predict and prefetch what the user will access in future over cellular. The key components of *precog* are best described as answers to the following questions:

5.3.1 When to record user actions?

Due to *the unbalanced cost problem*, user access pattern over WiFi and cellular networks are different. As *Precog* records user actions to improve prediction of future content access over cellular, it only records user actions while the device is connected to cellular network.

5.3.2 How to record user actions?

In order to record user actions, *Precog* leverages the HTML Document Object Model(DOM)[79]. The layout of a webpage is structured using a DOM tree of HTML tag nodes, with the $< html >$ tag node as the root. Even when content on the webpage changes, the consistency in layout of content is maintained through the DOM tree.

Each user action UA_i on the webpage can thus be identified as a $(DOM\ locator, Event)$ tuple, where *DOM locator* is used to locate a node in the DOM tree and the *Event*

is any interaction with the element, such as click, checkbox selection or form submission. *Precog* only considers user actions that trigger HTTP GET requests and stops recording actions if a HTTP POST is triggered. Handling POST messages is discussed further in the future work section.

The *DOM locator* is described using DOM attributes. While some nodes can be easily identified through an *id* attribute associated to them, not all nodes have an *id* attribute. Such nodes are identified using a $(start\_id, path, target)$ tuple, where *start_id* is the nearest DOM parent node which has an *id* attribute, *path* is the relative tree path from *start_id* to the target node and *target* is the attribute of the node on which action was performed. Thus, each user action UA_i is defined as:

$$UA_i = (DOM\ locator, Event) \tag{39}$$

5.3.3 How to group action sequences?

While the $(DOM\ locator, Event)$ tuple identifies an individual action of a user, the user may perform a sequence of such actions, separated by non-DOM actions. Non-DOM actions are actions which do not involve interaction with the web page DOM tree, e.g., opening a browser, opening a tab, typing a URL in the address bar, clicking on a bookmark, etc. In such sequence of actions, result of one action depends on the result of previous actions. Thus, individual DOM actions need to be grouped into sessions of user access, delimited by non-DOM actions.

Precog records action sequences in the form of session trees, as shown in Figure 16. The static URL which is accessed at the start of the session becomes the root of the tree ($Root\_URL_1$ and $Root\_URL_2$). This can be the first URL user typed when the browsing session started or when the user clicked a bookmark. These URLs are recorded as is in the *precog* tree. Any actions performed on the root URL becomes a child of the root node. Performing any action UA_i at time t_j can trigger HTTP requests for webpage W_{ij}. For example, in Figure 16, performing user action UA_1

on $Root\_URL_1$ at time t_a, t_b and t_c results in an HTTP request for URL W_{1a}, W_{1b} or W_{1c}, respectively. Any DOM actions performed on webpage W_{ij} adds children nodes to action UA_i. In Figure 16, action UA_2 was performed on W_{1j}, adding a child node to user action UA_1. Thus, each node in the tree represents the URL reached by visiting the root URL and following all the actions in the path from the root to the node. A separate session tree is maintained for each static URL visited by the user. Thus *precog* maintains a forest of session trees to remember all action sessions for a user.

5.3.4 How to rank?

Given the structure of session tree described above, *precog* needs to determine which nodes on the tree should be prefetched. In order to do that, each node in the session tree also maintains a counter for the number of times a user has accessed it(hit_i and $uhit_i$). Every time user repeats a set of actions, i.e., traverses a preexisting path in the session tree, *while on cellular network*, the counter on each node on that path is incremented. When the prefetch decision is to be made, all nodes in the tree are ranked according to their hit counter and top k nodes in the tree are prefetched. Note that the hit counter of a parent node is always higher than its child. During every prefetch session over WiFi the top ranked nodes are prefetched periodically.

5.3.5 How to prefetch?

Based on the ranking determined in the previous steps, the *precog* module fetches the highest ranked content when the mobile device is connected to WiFi. However, *precog* does not intrude on user's WiFi experience. The *precog* module runs using a *headless browser* such as PhantomJS[80]. A *headless browser* is a computationally light-weight web browser which does not include a fully featured graphical user interface like regular browsers, but has full capabilities to perform HTTP and DOM processing. The headless browser runs in the background allowing the user to continue using the

mobile device.

Additionally, *precog* also checks if the available disk space, CPU and battery life are above a certain threshold before it triggers the prefetching activity. As long as the system resources permit prefetching is performed periodically while the device is connected to the WiFi network. This ensures that the latest content is cached on the device when the user moves from WiFi to cellular network.

If all above constraints are met, the selected top nodes in each session tree are prefetched as per the action sequence in each tree. Only the root node of each session tree has a fixed URL which can be directly requested through a browser. For all interior nodes, the user actions are *replayed* to request the content created as a result of those actions. Thus, to prefetch the child node of $Root\_URL_1$, first $Root\_URL_1$ is opened in the browser and then action UA_1 is performed. The resulting HTTP GET request, say for URL W_1', fetches the latest content corresponding to that node in the tree. Further, if the child node of UA_1 is also marked to be prefetched, the action UA_2 is performed on the newly fetched W_1'.

Each UA_i is preplayed using DOM methods and attributes. For example, if the *DOM locator* of UA_1 is $(sid_1, index_1, attr_1)$ then the target element of the action is located as $document.getElementById(sid_1).childNodes[index_1].attr_1$, where:

- Method $getElementById(sid_1)$ finds the node in the *document* tree with id sid_1
- Attribute $childNodes[i]$ selects the i^{th} child of the node found above
- Attribute $attr_1$, which can be $href$, $text$, etc., selects the attribute of the child node which was acted upon

Once the target node has been determined, the recorded action is preplayed on it. The result of each preplay is used to perform further related actions.

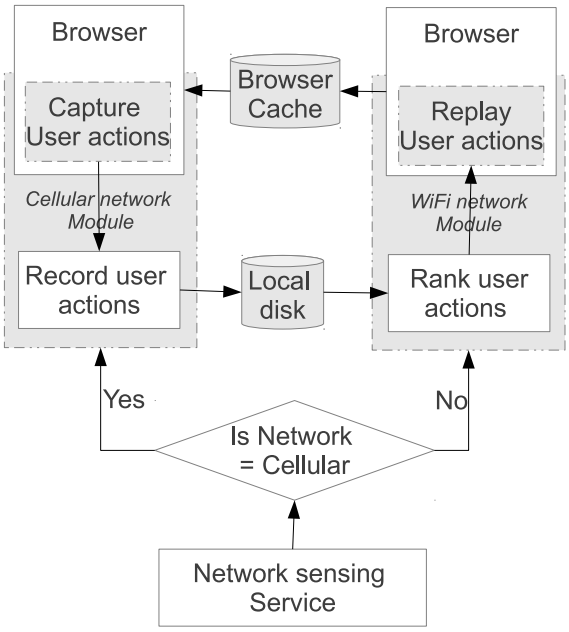


Figure 17: System architecture of *Precog*

5.3.6 How to use the prefetched content?

The prefetched content populates the browser cache on the mobile device. For any future access, if the content being fetched can be found in the browser cache, it is rendered from the cache. If it is not in the cache, it is fetched from the end server.

Precog also adds the prefetched content to the dedup cache so that bytes transferred on cellular network can be reduced even for content not found in the browser cache and requested from the end server. Thus, even when content keeps changing at the object level, the sub-object level redundancy can be leveraged to reduce cost of cellular access.

5.4 System Architecture

In this section we discuss the system architecture of *precog*. The solution consists of a network sensing service, which detects when the mobile device switches from cellular to WiFi, and vice versa. Android Java API exports a Connectivity Service[81] to

detect the active network interface. If the device is on cellular network, the *Cellular Network Module* is triggered which enables a JavaScript extension inside the browser to record user actions and header information of HTTP requests. The actions are stored on the local disk for later use.

When the device moves to WiFi network, the *WiFi network module*, reads the recorded actions from the local disk. It then ranks the actions, according to *precog* algorithm and starts a new browser instance in the background, using PhantomJS[80] to preplay the best ranked user actions. All content accessed during this browser session is stored in the browser cache. The next time mobile device uses the browser while on cellular network, the browser first checks this populated cache to address HTTP requests locally. If the requested object is already cache, no bytes need to be downloaded over the expensive cellular network, reducing both cost and latency.

While we discuss a simplistic system architecture for *precog* here, a more realistic extension of this solution can be a proxy based solution where the WiFi module of *precog* is offloaded to a proxy server. The task of ranking user actions and replaying them is performed on the proxy, which then pushes the prefetched content on the mobile device, while it is connected to WiFi. We discuss more details of this deployment strategy later in Chapter 6.

5.5 Evaluation

5.5.1 Methodology

We evaluate *precog* using synthetic user and network traces. Each user trace lists the different action paths followed by a user per hour of the day. Each action path is defined as $(Root\_URL_i, UA_a, UA_b, UA_c, ...)$. The trace creation has the following components:

- *Creating the universal set of action paths*: We first create a large set of possible paths that a user may traverse by crawling fourteen popular sites listed in

cbsinteractive.com	nbcuni.com	walmart.com
vevo.com	apple.com	nytimes.com
about.com	weather.com	epsn.go.com
google.com	disney.com	bbcnews.com
yelp.com	amazon.com	

Table 7: List of websites crawled

Table 7. The websites are selected from a list of top 50 digital properties published by ComScore[69]. The digital properties are ranked based on number of unique adult visitors/viewers across US, using iOS or Android platforms. From these top 50 digital properties we select a mix of video, news, sports, kids, health and shopping sites for our study.

We use WebDriver, a component of Selenium[82] browser automation tool. WebDriver provides a Python API to automate actions on different web browsers, in our case Firefox. Using WebDriver, our Python module clicks on all possible elements on each website, traversing links up to a reference depth of ten. We do not use any authentication based websites in this evaluation and the only event performed is clicks on $< href >$ tags on each page. This exercise generates a set of over 470,000 paths of length ten.

- *Creating individual user traces*: Next, we extract user traces from this universal set of action paths. To generate a week long user trace, we consider that user accesses network for sixteen hours per day and seven days a week. So each user trace contains 112 hours of network access. For each hour long session, we consider a parameter MAX_ACCESS, which describes the maximum number of action paths in any session. Every hour, the Python module selects a random number between one and MAX_ACCESS and selects that many paths from the universal set for that session. For each of the path selected for a session, the module further selects a random depth from one to ten and truncates each path up to that depth. This is done to create diversity in the length of action

paths. These truncated paths are then tagged with the hour value and added to the user trace. The parameter MAX_ACCESS controls the redundancy in user actions per trace as users with more paths per hour have a wider variety of paths and lower redundancy. We vary MAX_ACCESS count as 100, 1000 and 5000, and refer to them as $High$, $Medium$ and Low action redundancy. For each MAX_ACCESS value we generate ten user traces.

- *Creating network traces*: In order to evaluate each user over multiple network profiles, we also create ten network profiles. Again, we assume that network is accessed sixteen hours a day for seven days a week. We also assume that network stays same for at least an hour. For each of the 112 hours, we randomly select either WiFi or cellular connectivity with equal probability. One exception to this random selection is that the start of the day, i.e. hour 1, 17, 33, 49, 65, 81 and 97 are all on WiFi, which means the user starts the day at home, where there is WiFi connectivity.

We next run the *Precog* algorithm on each combination of user and network traces. In each run, the first six days, i.e. first 96 hours, of the traces are used to build the forest of session trees. Every hour the algorithm checks the network connectivity and if it is cellular, all the paths in that hour are used to grow the forest of session trees. Accesses over WiFi connectivity are ignored. No network traffic is downloaded for the first 96 hours. From the 97^{th} hour, prefetching is done for every hour when the user is on WiFi. At the beginning of the prefetch session, the top 25% nodes in the forest, based on the hit count, are prefetched. Again, WebDriver is used to replay the actions on Firefox and the *real* network traffic is recorded using Tcpdump. When the user switches to cellular network, all the actions for that hour are also replayed using WebDriver and the session trees are also populated. The *real* network traffic generated from these actions is also recorded using Tcpdump. These traces are then used to evaluate the performance of *precog*.

5.5.2 Performance Results

5.5.2.1 Macroscopic Results

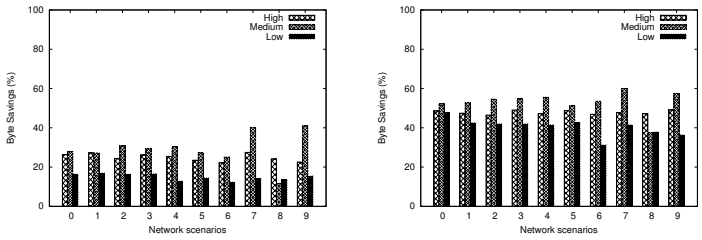
The metrics we consider for macroscopic analysis are (i) byte savings and (ii) prefetch efficiency. We compare the bytes downloaded in the prefetch session and the cellular session after a training period of 96 hours. We restrict the trace collection to one prefetch session and one cellular session on the seventh day. From the Tcpdump traces we count the following:

- Total cellular bytes: Number of bytes downloaded during cellular session.
- Total WiFi bytes: Number of bytes prefetched during WiFi session.
- Inter redundant object bytes: Total bytes *in all the objects* accessed over cellular network on day 7 which were prefetched in the latest WiFi session.
- Inter redundant bytes: Total bytes accessed over cellular network on day 7 which were prefetched in the latest WiFi session.

Byte Savings

Here we measure the number of bytes saved over cellular network as a result of prefetching over WiFi. We compute byte savings with prefetched browser cache as:

$$Byte\ savings_{browser\ cache} = \frac{Inter\ redundant\ object\ bytes}{Total\ cellular\ bytes} * 100 \qquad (40)$$



(a) Prefetched browser cache (b) Prefetched dedup cache

Figure 18: Cellular bytes saved using bytes prefetched over WiFi

We also compute a byte level redundancy in the prefetched bytes, i.e. byte savings with prefetched dedup cache as:

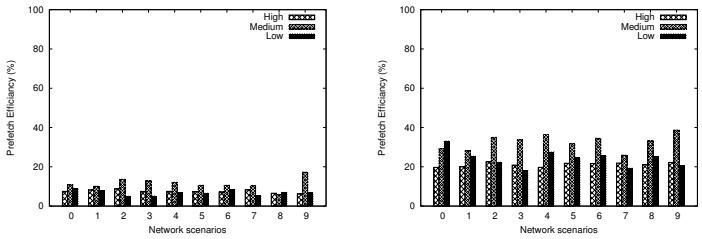
$$Byte\ savings_{dedup\ cache} = \frac{Inter\ redundant\ bytes}{Total\ cellular\ bytes} * 100 \tag{41}$$

Figure 18 shows the average byte savings observed across multiple users for each network trace. We observe that just using the prefetched browser cache saves 23% bytes over cellular. This metric more than doubles if dedup is also used, leading to 47% byte savings.

Prefetch Efficiency

In addition to the bytes saved over cellular network, we also measure the efficiency of prefetching, i.e. how many bytes downloaded over WiFi are actually used over cellular. We compute prefetch efficiency with browser cache as:

$$Prefetch\ efficiency_{browser\ cache} = \frac{Inter\ redundant\ object\ bytes}{Total\ WiFi\ bytes} * 100 \tag{42}$$



(a) Prefetched browser cache (b) Prefetched dedup cache

Figure 19: Prefetch efficiency of Precog

We also compute prefetch efficiency with dedup cache as:

$$Prefetch\ efficiency_{dedup\ cache} = \frac{Inter\ redundant\ bytes}{Total\ WiFi\ bytes} * 100 \tag{43}$$

Figure 19 shows the average prefetch efficiency observed across multiple users for each network trace. We observe that the prefetched browser cache gives a 9% prefetch efficiency. The prefetched dedup cache gives much better prefetch efficiency of 26%. This means that for every 100 bytes downloaded over WiFi, 26 bytes are saved over

cellular network using prefetched dedup cache. Note that the cost of data access over cellular is around *83 times* of WiFi access. Even a 26% prefetched efficiency, gives a 95% cost reduction over cellular.

5.5.2.2 Microscopic results

Note that the performance of *precog* is governed by the frequency with which content changes on the server and the accuracy of *precog*'s ranking logic. Here we evaluate the performance of *precog*'s ranking logic. Specifically, we compute the action-level redundancy in user traces for different network traces. Based on the user activity learned from the first 96 hours in each user-network trace combination, we enable *precog* in the last sixteen hours of the trace. For these last sixteen hours of each run, as *precog* algorithm runs on the traces, we count the following:

- Total actions: Number of actions performed while on cellular network on day 7.
- Intra redundant actions: Number of actions on cellular network on day 7 which were performed in any previous cellular session.
- Inter redundant actions: Number of actions on cellular network on day 7 which were prefetched in the latest WiFi session.

Given these three numbers, we compute *Ideal action redundancy* as:

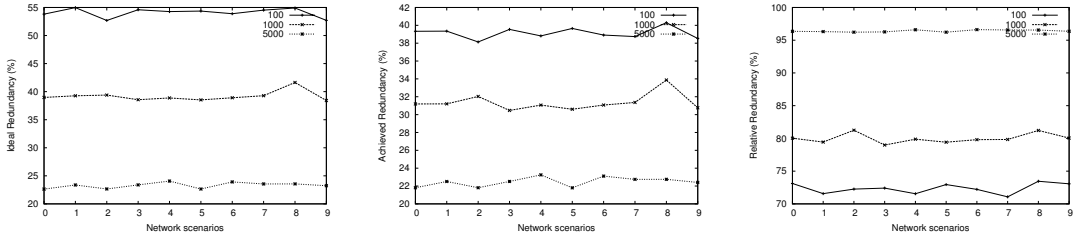
$$Ideal\ action\ redundancy = \frac{Intra\ redundant\ actions}{Total\ actions} * 100 \tag{44}$$

and we compute *Observed action redundancy* as:

$$Observed\ action\ redundancy = \frac{Inter\ redundant\ actions}{Total\ actions} * 100 \tag{45}$$

We also compute *Relative redundancy* as:

$$Relative\ redundancy = \frac{Observed\ action\ redundancy}{Ideal\ action\ redundancy} * 100 \tag{46}$$



(a) Ideal redundancy (b) Achieved redundancy (c) Relative redundancy

Figure 20: (a)Ideal redundancy for varying network traces (b) Redundancy achieved by *precog* in different network scenarios (c) Redundancy achieved by *precog* as a percentage of ideal redundancy for different network setups

Figure 20 presents the trends in action redundancy with change in network traces. Each point on the graph is an average over ten users. We observe in Figure 20(a) that the ideal or best achievable redundancy varies from average 23% with 5000 paths per session, to average 54% with only 100 paths per session. Across these different levels of redundancy, *precog* can leverage an action level redundancy of average 22.5% for 5000 MAX_ACCESS, up to average 39% with 100 MAX_ACCESS. The relative redundancy of *precog* with respect to ideal redundancy averages from 72% for 5000 MAX_ACCESS to over 96% for 100 MAX_ACCESS. This shows that user's actions can be learned from their past and efficiently used for predicting future accesses.

5.6 Related Issues

- *Robust replay*: *Precog* is based on the observation that content layout stays consistent. However, it could happen that the layout of a page changes, though not quite often. In such a scenario, an action learned from the past layout cannot be preplayed on the latest layout. If *precog* finds a scenario where no node can be found for the recorded DOM locator, e.g., child index is out of range, then the preplay is aborted. No further actions in the current action sequence are executed.

- *Impact on battery*: While the main focus of *precog* is to reduce the cost of cellular access, it can also serve to reduce the energy consumption of cellular network access. Prior analysis in MAUI[83] showed that WiFi gives 102 KB/Joule efficiency while 3G networks give only 36 KB/Joule. Thus prefetching over WiFi can reduce the overall energy consumption on mobile devices. The trace analysis in Section 5.5 shows that *precog* gives 26% efficiency, thus around 4× bytes needs to be prefetched over WiFi to save a given number of bytes over cellular. While this results in more energy consumption than in accessing all the bytes over cellular, *precog* prediction can be improved further to increase the prefetch efficiency and perform overall energy savings.

- *Handling video traffic*: Video traffic is a dominant part of Internet traffic. It is thus relevant to study the impact of *precog* on video traffic. There are multiple granularities at which prefetching can offset the cost of cellular video access: (i) The entire video can be prefetched to serve future requests from the user, (ii) some popular sections of the video can be prefetched based on a popularity metric or (iii) the video advertisements, which appear during a video, can be prefetched. In this work we have evaluated the performance of *precog* with first case and plan to explore alternate strategies as part of future work.

- *Privacy concerns*: While *precog* adds to name-based prefetching by handling authentication, it also creates privacy concerns with prefetching. Accessing authentication protected content can be intrusive to users. *Precog* addresses this by exposing to the user whether *precog* is enabled for that a domain or not. Additionally, *precog* does not preplay any actions, except sign-in, which trigger an HTTP POST request as POST requests are non-idempotent. They can lead to change of state at the server which is irreversible.

CHAPTER VI

INTEGRATED OPERATIONS

We have discussed thus far three complementary approaches of adapting network protocols and algorithms to *ultra-mobile computing*: *Adaptive Flow Control, Asymmetric Caching and Precog*. *AFC*, *Asymmetric caching* and *Precog* are fully complementary as they work on different layers of the network stack; while *Asymmetric caching* operates at layer 3.5 right above IP layer, *AFC* functions on transport layer and *Precog* interacts with the applications on layer 5. We now discuss how these three approaches can work synergistically.

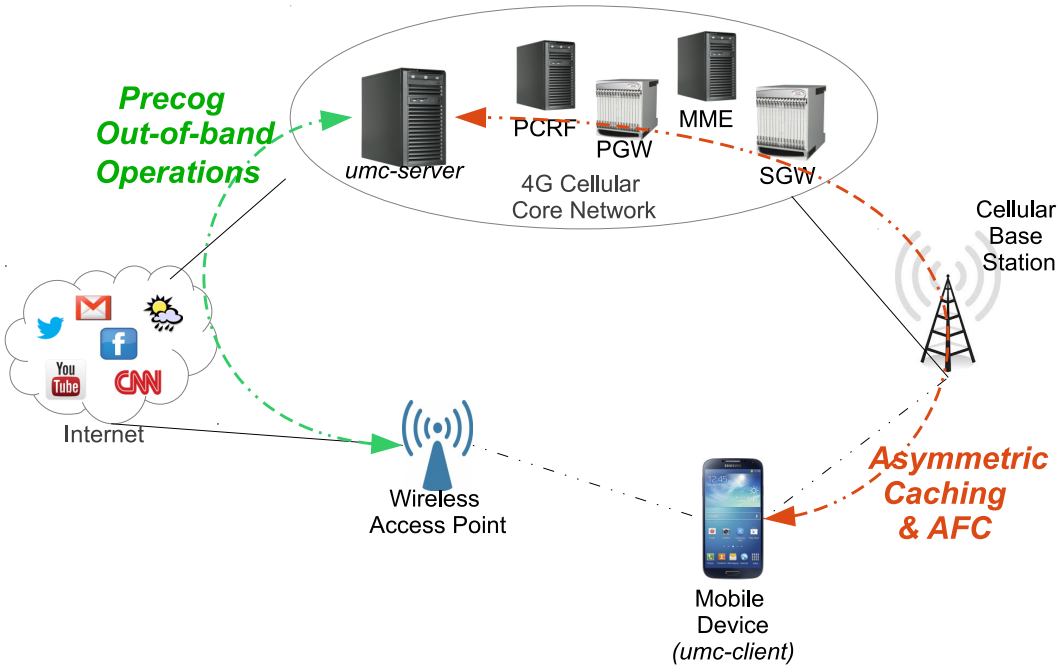


Figure 21: System architecture for integrated operations of *AFC*, *Asymmetric caching* and *Precog*

We propose a two-ended cloud proxy based solution, as shown in Figure 21, to

integrate all the three approaches. One end of the solution, which we call the *umc-server*, is a Linux server, which acts as an HTTP/TCP proxy, deployed between the core cellular network and the Internet. The other end is a software module running on the mobile device, which we call the *umc-client*. Similar proxy based solutions, which integrate TCP optimization, caching and application acceleration have been deployed in real networks [84], establishing precedence.

The individual components of *umc-server* are:

- At the IP layer, *umc-server* incorporates the `dd-src` component of *Asymmetric caching* which breaks every downstream packet into value based chunks, using Rabin Fingerprinting, stores the hashes of the chunks and replaces any repeating chunk in the packet with its corresponding hash.

- At the TCP layer, *umc-server* breaks the end-to-end TCP connection between the content server and mobile client, into two connections. It acts as a receiver for the connection with the content server and uses default TCP flow control for this connection. As application read rate does not fluctuate on the resource rich server, default TCP flow control does not under-perform here. The *umc-server* also maintains a downstream TCP connection with the mobile client, which is *AFC* enabled. The TCP sender on this connection listens to application read rate advertisement from the mobile device and computes the flow control window from these values. It also handles zero window advertisements on the downstream TCP connection as per *AFC* algorithm.

- At the application layer, *umc-server* also runs an HTTP proxy which performs out-of-band prefetching using *precog* algorithm. While the device is connected over WiFi the proxy establishes an out-of-band connection with it over the Internet. It then receives the recorded actions from the *precog* module running on the mobile, ranks them and replays the top actions using a headless browser,

such as PhantomJS[80]. The prefetched content is then pushed on the mobile device to be used later when the mobile device uses cellular network access. This two-ended deployment relieves the computational cost of *precog* on mobile device.

The *umc-client*, located on the mobile device, contains the following modules:

- At the IP layer, `dd-dst` component of *umc-client* runs the flowlet extraction, feedback selection and feedback advertisement module of *asymmetric caching*. It also inflates any deduplicated packets received downstream by using the hash to chunk map stored in the local cache. This inflated packet is sent to the upper layer, in this case the *adaptive flow control* receiver module.

- The *adaptive flow control* receiver module forms the transport layer component of *umc-client*. It monitors the TCP receive buffer to estimate the current application read rate and advertises it in any outgoing packet.

- The *precog* module runs on the application layer in the *umc-client*. This module interacts with the browser on the mobile device to capture user actions during web browsing sessions over cellular network. These actions are stored and later uploaded to the *umc-server* while the user is on WiFi network. While on WiFi network, the *precog* module also downloads prefetched content from the *umc-server* and populates *both the browser cache as well as the* `dd-dst` *cache*. During any subsequent cellular access, if the content being requested can be found in the browser cache, it is rendered from the browser cache. If it is not in the cache, it is fetched from the end server. During the download of the new content, the cached information in `dd-dst` is leveraged to perform dedup.

We have described a proxy based deployment to integrate the operations of *AFC*, *Asymmetric caching* and *Precog* to improve cellular network access for smartphones

and tablets. Alternate approaches to integrate these solutions can be a cloud based service for users which can leverage redundancy in content and user actions across WiFi and cellular access, and accelerate TCP performance. Another possible deployment can be a dedicated proxy within an enterprise network, e.g., Georgia Tech, where Georgia Tech deploys an HTTP/TCP proxy at the edge of the network. This proxy can improve TCP performance for smartphones and tablets connecting to the Internet from within the campus network, reduce congestion in the campus WiFi network through dedup and reduce the traffic load on WiFi by prefetching content when users are connected to less loaded WiFi access points.

CHAPTER VII

FUTURE WORK

7.1 Rethinking transport layer protocols for ultra-mobile devices

As part of this dissertation, we have developed an *adaptive flow control(AFC)* for TCP on mobile devices. We have investigated and addressed some of the key deficiencies in classical flow control. And as we undertook this research, we realized some avenues for future work:

- *Control theoretic model of AFC*: While we established in Chapter 3 that classical TCP flow control models an *Integral controller*, AFC does not fit into any classical system template. Modelling AFC as a control theoretic model thus remains an open issue.

- *Impact of network losses on AFC*: AFC includes loss detection and recovery mechanisms to take care of buffer losses. Further research can be done in studying the interactions of network losses and AFC and how should loss recovery happen when congestion and buffer losses occur simultaneously.

- *Including delay in control theory model of TCP*: TCP has an inherent delay of half an RTT. An ACK packet received at the sender was sent half an RTT before by the receiver. Hence, the state information, such as receive window, application read rate, etc, is a snapshot of the receiver half an RTT back. An open research area is to include this into the control model and study the impact of this delay on TCP performance.

- *Impact of base buffer size on AFC*: Another area of further study is: *how does AFC compare with classical flow control in sensitivity to base buffer size.* While in some scenarios just doubling the buffer would improve the performance of classical flow control, in others it may not. An open issue remains to determine the buffer growth pattern of Auto-Tuning and compare AFC with it.

7.2 Improving network deduplication for ultra-mobile devices

Asymmetric caching proposes an improved network deduplication for ultra-mobile devices. The promising results seen with *asymmetric caching* motivates further directions of research such as:

- *Performance*: One of the critical factors that will influence viability of the *asymmetric caching* solution will be the performance at both the server and client ends. At the server side, whether the server platform can scale to 10Gbps speeds will be of importance. The primary constraint that will have to be overcome will be the disk access that *asymmetric caching* will rely on for its caches. Disk accesses in general are expensive operations, especially when they occur in the data-path. Intelligent prefetching techniques will have to be developed so that any data-path cache lookups happen only in main memory, and the transfer from secondary storage to main memory happens in the background. At the client side, the limited CPU and memory resources available could dampen the performance of the *asymmetric caching* client. Code optimization and intelligent short-circuiting of operations will have to be explored to overcome the aforementioned constraints.

- *Encrypted traffic*: While the concept of *feedback* can apply to both encrypted and unencrypted traffic, the current system architecture of *asymmetric caching* does not allow deduplication of encrypted packets as the `dd-src` has no way

of learning the actual content in an encrypted packet. As more and more authentication based applications gain popularity over smartphones and tablets, *asymmetric caching* needs to be adapted to an end-to-end solution which can leverage redundancy within encrypted traffic also.

- *Upstream traffic*: Smartphones and tablets are not only content consumption devices but also content creation tools. Users use their smartphones and tablets to create documents, click pictures, shoot videos, create video mash-ups or photo effects. All this content is further uploaded on the Internet for cloud storage or publishing on the web. It will be interesting to study the scope of dedup on such upstream traffic.

- *Evaluating the energy impact of* asymmetric caching: Battery life is an important factor contributing to the success of any solution for ultra-mobile devices. To leverage all past information on ultra-mobile devices, *asymmetric caching* performs several complex tasks such as Rabin fingerprinting and flowlet extraction. While our preliminary prototype showed that *asymmetric caching* has deployable CPU and memory overheads, a more rigorous analysis of battery consumption by *asymmetric caching* can be a future study.

7.3 Smart prefetching solutions for ultra-mobile devices

Precog proposes a novel action-based approach to prefetching on smartphones and tablets. While *precog* addresses the major limitation in existing prefetching solutions, i.e. name-dependence, we identify the following directions of future work:

- *Incorporate usefulness and opportunity*: *Precog* uses the hit count of each user action over past cellular sessions to determine which actions are most likely to repeat in the future. To ensure that user always gets the latest content for a particular action, prefetching is performed repeatedly over WiFi access. The

algorithm can be further tuned to incorporate a $usefulness$ and *opportunity* parameter. The $usefulness$ of any content is a measure of the freshness of prefetched content. If the content on a server changes every hour, irrespective of user actions, the content prefetched right before the user accesses it is more $useful$ than content fetched hours ago. Similarly, while *precog* does an aggressive prefetch over each WiFi session, the *opportunity* factor will try to estimate which WiFi session is best suited to prefetch content.

- *Extend to authenticated accesses and single sign-on services*: *Precog* considers only HTTP GET requests to build user action session trees and preplays HTTP POST messages only for authentication such as logging into a site. Another future area of research can be to record and replay more complex HTTP POST messages for prefetching. This introduces new challenges as POST messages are non-idempotent, i.e. they can result in state change at the server and client which will destroy the transparency of *precog*.

- *Workloads*: Caching solutions in general are heavily influenced by the nature of the workload that they are applied to. Hence, synthetic workloads are risky to use because of the possibility of them being not representative of real-life workloads. Hence, part of future work will involve the collection of real-life workloads from different sources (users and content-servers) that can then be used for testing the prototypes.

- *Prototype*: Yet another area of future research is to verify the viability of *precog* through a real prototype with the user action recording module implemented as an extension in the mobile browser and another headless browser implementation to select best user actions and prefetch content without significant CPU, memory, battery and graphical overhead.

CHAPTER VIII

CONCLUSIONS

The adoption of mobile devices such as smartphones has reached a significant threshold with the number of such devices shipped now surpassing the number of PCs shipped [1]. Nearly 40% of Internet time is now attributed to mobile devices such as smartphones and tablets [2]. *An ever-increasing pressure on wireless network performance and scalability, however, accompanies such exciting trends.* While these ultra-mobile devices try to catch up to the performance of traditional personal computers, they are bound by the promise of portability and compactness. They have introduced a new paradigm in mobile computing, which we refer to as *ultra-mobile computing*.

The focus of this work is identifying and resolving the impact of the conflicting characteristics of *ultra-mobile computing* on existing network protocols and algorithms. While there are a number of problems to address in this direction, we have examined three directions where significant improvement can be observed by adapting existing solutions for smartphones and tablets.

In *Adaptive Flow Control*, 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 and exhibits fairness.

In *Asymmetric caching*, we proposed an improvement to baseline network deduplication that allows the *dedup destination* to selectively feedback appropriate portions of its cache to the *dedup source* with the intent of improving the redundancy elimination efficiency. We show using traffic traces collected from 30 mobile users, that with asymmetric caching, over 89% of the achievable redundancy can be identified and eliminated *even when the dedup source* has less than *one hundredth of the cache size as the dedup destination*[12]. Further, we show that the number of bytes saved from transmission at the *dedup source* because of asymmetric caching is over 6× that of the number of bytes sent as feedback.

In *Precog*, we motivate the need to rethink prefetching approaches for smartphones and tablets, given the dynamism in current web content and the heterogeneity in network access on these devices. We then propose a name-independent network-aware prefetching solution for these devices. Through a synthetic trace analysis with real web content we show that for different levels of user level redundancy, *precog* achieves 47% byte savings on average, with a prefetch efficiency of 26%.

Finally in Chapter 6, we also show that *Adaptive Flow Control*, *Asymmetric caching* and *Precog* are complementary solutions which can come together in a proxy based deployment to provide significant improvement in network performance and reduction in network load for smartphones and tablets. Future directions of research are discussed in Chapter 7

REFERENCES

[1] "Smartphone sales exceed those of PCs for first time." Applesmashesrecordwww.digitaltrends.com/mobile/smartphone-sales-exceed-those-of-pcs-for-first-time-apple-smashes-record/.

[2] "Nearly 40 Percent Of Internet Time Now On Mobile Devices." marketingland.com/report-nearly-40-percent-of-internet-time-now-on-mobile-devices-34639.

[3] "Symantec Survey Feb 22." 2012,www.symantec.com/about/news/release/article.jsp?prid=20120221_02.

[4] "Infographic: Mobile Apps in the Enterprise Are the Future." www.zendesk.com/blog/mobile-apps-in-the-enterprise-are-the-future.

[5] "Cisco Visual Networking Index, February 2011." http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html.

[6] I. S. Institute, "RFC 793." rfc.sunsite.dk/rfc/rfc793.html, 1981.

[7] S. Sanadhya and R. Sivakumar, "Adaptive flow control for tcp on mobile phones," in *INFOCOM, 2011 Proceedings IEEE*, 2011.

[8] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "Endre: an end-system redundancy elimination service for enterprises," in *NSDI*, 2010.

[9] A. A. Shan-Hsiang Shen, Aaron Gember and A. Akella, "Refactor-ing content overhearing to improve wireless performance," in *ACM MobiCom*, 2011.

[10] S. C. Rhea and K. Liang, "Value-based web caching," in *The 12th Int. World Wide Web Conference*, 2003.

[11] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *ACM SIGCOMM*, 2000.

[12] S. Sanadhya, R. Sivakumar, K.-H. Kim, P. Congdon, S. Lakshmanan, and J. P. Singh, "Asymmetric caching: improved network deduplication for mobile devices," in *ACM MobiCom*, 2012.

[13] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *SIGCOMM Comput. Commun. Rev.*, 1996.

[14] T. S. Loon and V. Bharghavan, "Alleviating the latency and bandwidth problems in www browsing," in *Proc. of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, pp. 20–20, 1997.

[15] K. Lau and Y.-K. Ng, "A client-based web prefetching management system based on detection theory," in *Web Content Caching and Distribution*, vol. 3293 of *Lecture Notes in Computer Science*, pp. 129–143, Springer Berlin Heidelberg, 2004.

[16] Z. J. Haas, "Mobile-TCP: An asymmetric transport protocol design for mobile systems," in *IEEE International Conference on Communications*, 1997.

[17] A. Bakre and B. R. Badrinath, "I-TCP: Indirect TCP for mobile hosts," in *International Conference on Distributed Computing Systems*, 1995.

[18] H. Balakrishnan, S. Seshan, R. H. Katz, and Y. H. Katz, "Improving reliable transport and handoff performance in cellular wireless networks," *Wireless Networking*, 1995.

[19] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *ACM Conference on Mobile Computing and Networking*, 2001.

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

[21] E. Weigle and W. chun Feng, "Dynamic right-sizing: A simulation study," in *IEEE ICCCN*, 2001.

[22] "Linux Auto Tuning." `www.kernel.org`.

[23] J. Heffner, "High bandwidth TCP queuing." `www.psc.edu/~jheffner/papers/senior_thesis.pdf`.

[24] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: the implications of universal redundant traffic elimination," in *ACM SIGCOMM*, 2008.

[25] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: findings and implications," in *ACM SIGMETRICS*, 2009.

[26] E. Zohar, I. Cidon, and O. O. Mokryn, "Celleration: loss-resilient traffic redundancy elimination for cellular data," in *ACM HotMobile*, 2012.

[27] "RFC 2616: Hypertext Transfer Protocol – HTTP/1.1." `www.w3.org/Protocols/rfc2616/rfc2616.html`.

[28] "The apache HTTP Proxy." `httpd.apache.org`.

[29] "RFC 3284: The VCDIFF Generic Differencing and Compression Data Format." `www.faqs.org/rfcs/rfc3284.html`.

[30] Y. M. Chan, T. Kelly, and J. C. Mogul, "Design, implementation, and evaluation of duplicate transfer detection in http," in *NSDI*, 2004.

[31] B. C. Housel and D. B. Lindquist, "Webexpress: a system for optimizing web browsing in a wireless environment," in *ACM MobiCom*, 1996.

[32] F. Douglis, M. Rabinovich, and A. Haro, "Hpp: Html macro-preprocessing to support dynamic document caching," in *USENIX Symposium on Internet Technologies and Systems*, 1997.

[33] "The Akamai Content Delivery Network." `www.akamai.com`.

[34] O. Saleh and M. Hefeeda, "Modeling and caching of peer-to-peer traffic," in *IEEE ICNP*, 2006.

[35] E. Zohar, I. Cidon, and O. O. Mokryn, "The power of prediction: cloud bandwidth and cost reduction," in *Proc. of the ACM SIGCOMM*, 2011.

[36] L. Fan, P. Cao, and Q. Jacobson, "Web prefetching between low-bandwidth clients and proxies: Potential and performance," in *Proc. of ACM SIGMETRICS*, 1999.

[37] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pp. 179–194, 2010.

[38] Z. Jiang and L. Kleinrock, "Web prefetching in a mobile environment," *IEEE Personal Communications*, vol. 5, 1998.

[39] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploring the bounds of web latency reduction from caching and prefetching," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, 1997.

[40] J. Mrquez, J. Domenech, J. Gil, and A. Pont, "Exploring the benefits of caching and prefetching in the mobile web," in *Second IFIP Symposium on Wireless Communications and Information Technology for Developing Countries*, 2008.

[41] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in *ACM MobiSys*, 2012.

[42] A. J. Nicholson and B. D. Noble, "Breadcrumbs: forecasting mobile connectivity," in *ACM MobiCom*, 2008.

[43] P. Stuedi, I. Mohomed, and D. Terry, "Wherestore: location-based data storage for mobile devices interacting with the cloud," in *ACM Workshop on Mobile Cloud Computing Services: Social Networks and Beyond*, 2010.

[44] "Google Octane Benchmark." `developers.google.com/octane`.

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

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

[47] P. Sinha, T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan, "Wtcp: A reliable transport protocol for wireless wide-area networks.," *Wireless Networks*, pp. 301–316, 2002.

[48] H.-Y. Hsieh and R. Sivakumar, "ptcp: An end-to-end transport layer protocol for striped connections," in *IEEE ICNP*, 2002.

[49] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "RFC 2018." `www.faqs.org/rfcs/rfc2018.html`, 1996.

[50] "Riverbed." `www.riverbed.com/us/solutions/wan\_optimization/`.

[51] "Comscore Report: Digital Omnivore, October 2011." `www.comscore.com/Press_Events/Presentations_Whitepapers/2011/Digital_Omnivores`.

[52] "WeFi Analytics Report." `August2010,mobilemarketingmagazine.com/sites/\\default/files/WeFi\%20Wi-fi\%20Data\%20Report\%20Q1\%202010.pdf`.

[53] "IP Core Pooling Tutorial, AUgust 2006." `archive.ericsson.net/service/internet/picov/get?DocNo=1/28701-FGB101256`.

[54] "Ericsson SGSN-MME." `www.ericsson.com/ourportfolio/products/sgsn-mme?nav=fgb_101_256`.

[55] "EMC Data Domain." `www.emc.com/backup-and-recovery/data-domain/data-domain.htm`.

[56] M. O. Rabin, "Fingerprinting by random polynomials," in *Technical Report TR 15-81, Department of Computer Science, Harvard University*, 1981.

[57] U. Appel and A. V. Brandt, "Adaptive sequential segmentation of piecewise stationary time series.," *Information Sciences*, pp. 27–56, 1983.

[58] "Bob Jenkins hash functions." `burtleburtle.net/bob/`.

[59] "Netfilter." `www.netfilter.org/projects/libnetfilter\_queue/`.

[60] "FCC." www.fcc.gov.

[61] "Comcast." wwwb.comcast.com.

[62] "AT&T." www.att.com.

[63] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, "Mobile data offloading: how much can wifi deliver?," in *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pp. 425–426, 2010.

[64] S. Dimatteo, P. Hui, B. Han, and V. Li, "Cellular traffic offloading through wifi networks," in *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pp. 192 –201, oct. 2011.

[65] "WordPress." www.wordpress.org.

[66] "Joomla!." www.joomla.org.

[67] "Drupal." www.drupal.org.

[68] "MediaWiki." www.mediawiki.org.

[69] "ComScore Media Metrix Multi-Platform Feb 2013." www.comscore.com/Insights/Press_Releases/2013/3/comScore_Announces_US_Launch_of_Media_Metrix_Multi-Platform?utm_source=Triggermail&utm_medium=email&utm_term=Mobile%20Insights&utm_campaign=Post%20Blast%20%28sai%29%3A%20BII%20MOBILE%20INSIGHTS%3A%20How%20Mobile%20Startups%20Are%20Taking%20Revenue%20From%20Carriers.

[70] "The Livelab Project." livelab.recg.rice.edu/index.html.

[71] "AJAX: Asynchronous JavaScript and XML." www.w3schools.com/ajax/.

[72] "Adobe Flash runtimes." www.adobe.com/products/flashruntimes.html.

[73] "JavaScript." www.w3schools.com/js/.

[74] "Java Applets." docs.oracle.com/javase/tutorial/deployment/applet/.

[75] "Introduction to ActiveX Controls." msdn.microsoft.com/en-us/library/aa751972(VS.85).aspx.

[76] "Does movement attract attention on Web pages?." wel.cs.manchester.ac.uk/studies/saswat/ticker.php.

[77] William Lidwell, Kritina Holden, Jill Butler, *Universal Prinicples of Design*. Rockport Publishers, 2003.

[78] "Anonymous Microsoft Web Data Data Set." archive.ics.uci.edu/ml/datasets/Anonymous+Microsoft+Web+Data.

[79] "HTML DOM Specification." www.w3.org/TR/DOM-Level-2-HTML.

[80] "PhantomJS." phantomjs.org/.

[81] "Detecting and Monitoring the Connectivity Status." developer.android.com/training/monitoring-device-state/connectivity-monitoring.html.

[82] "Selenium." docs.seleniumhq.org.

[83] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *ACM MobiSys*, 2010.

[84] "Bytemobile." www.bytemobile.com.