Characterization and Optimization of Resource Utilization For Cellular Networks

by

Feng Qian

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2012

Doctoral Committee:

Associate Professor Z. Morley Mao, Chair
Associate Professor Robert Dick
Associate Professor Jason N. Flinn
Technical Staff Oliver Spatscheck, AT&T Labs – Research

To my family.

ACKNOWLEDGEMENTS

Five years are not short. I would like to thank many people who helped me during this long journey of Ph.D. study. The completion of this dissertation is impossible without them.

I would like to thank my advisor, Professor Z. Morley Mao. As an expert in networking, systems, security, and mobile computing, Morley provided me with excellent guidance on conducting research in a professional manner. At the early stage of my Ph.D. study, she taught me tediously on every step of attacking a research problem. The training of this whole procedure will benefit me for my whole life. Morley also set a great example for me by her diligent, persistence, and enthusiasm. It was my great pleasure to work with her since our first greeting on August 16 2007, the second day after I came to Michigan.

I am deeply thankful to my mentors at AT&T labs - research where I have done four successful summer internships in 2008, 2009, 2010, and 2011. I am very fortunate to work with these knowledgeable and experienced researchers: Alexandre Gerber, Oliver Spatscheck, Subhabrata Sen, Jeffrey Erman, and Walter Willinger. At AT&T, I had a lot of enlightening discussions with them. Without their effort, our projects such as ARO would never be so successful and impactful.

I want to express my gratitude to Professor Robert Dick, Professor Jason Flinn, and Oliver Spatscheck for serving on my thesis committee. They provided valuable comments and suggestions for improving my dissertation. I thank them for coming to my oral defense.

I would like to thank my friends and colleagues at Michigan. Zhiyun Qian and I came to Michigan on the same day and we also finished our oral defenses on the same day, after

five years great time of working and playing together. Dr. Ying Zhang and Dr. Xu Chen are former members of our group. They helped me a lot and both set great models for me. I have worked closely with Yudong Gao, Junxian Huang, Kee Shen Quah, Zhaoguang Wang, Qiang Xu on various research projects. They are also my great friends at Michigan. Many thanks to other friends and colleagues, including, but not limited to: Zhe Chen, Lujun Tony Fang, Mark Gordon, Xiaoen Ju, Abhinav Pathak, Li Qian, Sanae Rosen, Yunjing Xu, Jie Yu, Caoxie Zhang, Jing Zhang, and Xinyu Zhang. My thank goes to Stephen Reger, the secretary of our software and systems lab.

I love the University of Michigan. "For today, Goodbye; For tomorrow, Good Luck; For a lifetime, GO BLUE!" Ann Arbor is such a great place to live. I will forever remember its picturesque fall.

Finally, I would like to express my deepest gratitude to my beloved parents, Jieyu Shi and Jian Qian, for their enduring love, support and encouragement throughout my life. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Characterization and Optimization of Resource Utilization For Cellular Networks

by

Feng Qian

Chair: Z. Morley Mao

Cellular data networks have experienced significant growth in the recent years particularly due to the emergence of smartphones. Despite its popularity, there remain two major challenges associated with cellular carriers and their customers: carriers operate under severe resource constraints, while many mobile applications are unaware of the cellular specific characteristics, leading to inefficient radio resource and handset energy utilization. My dissertation is dedicated to address both challenges, aiming at providing practical, effective, and efficient methods to monitor and to reduce the resource utilization and bandwidth consumption in cellular networks. Specifically, from carriers' perspective, we performed the first measurement study to understand the state-of-the-art of resource utilization for a commercial cellular network, and revealed that fundamental limitation of the current resource management policy is treating all traffic according to the same resource management policy globally configured for all users. On mobile applications' side, we developed a novel data analysis framework called ARO (mobile Application Resource Optimizer), the first tool that exposes the interaction between mobile applications and the radio resource management policy, to reveal inefficient resource usage due to a lack of transparency in the lower-layer protocol behavior. ARO revealed that many popular applications built by pro-

fessional developers have significant resource utilization inefficiencies that are previously unknown. Motivated by the observations from both sides, we further proposed a novel resource management framework that enables the cooperation between handsets and the network to allow adaptive resource release, therefore better balancing the key tradeoffs in cellular networks. We also investigated the problem of reducing the bandwidth consumption in cellular networks by performing the first network-wide study of HTTP caching on smartphones due to its popularity. Our findings suggest that for web caching, there exists a huge gap between the protocol specification and the protocol implementation on today's mobile devices, leading to significant amount of redundant network traffic.

CHAPTER I

Introduction

Cellular data networks have experienced significant growth in the recent years particularly due to the emergence of smartphones. As reported by a major U.S. carrier [1], its cellular data traffic has experienced a growth of 5000% over 3 years [1]. Despite its popularity, there remain two major challenges associated with cellular carriers and their customers: *carriers operate under severe resource constraints, while mobile applications often utilize radio channels and consume handset energy inefficiently.*

From the carriers' perspective, compared to the Wi-Fi and wired networks, cellular systems operate under more resource constraints. To keep up with the explosive increase of their cellular traffic, all U.S. carriers are expected to spend 40.3 billion dollars on cellular infrastructures in 2011 [2]. Cellular networks employ a unique resource control mechanism to manage the limited resources [3]. However, our research identified significant inefficiency in the current resource management policy. For example, by analyzing the data collected from a large commercial 3G carrier, we found that up to 45% of the high-speed transmission channel occupation time is wasted on idling, because the critical parameters controlling the release of radio resources are configured in a static and ad-hoc manner. Cellular carriers therefore urgently need methods to systematically characterize and optimize resource usage for their networks.

From the customers' perspective, there is a plethora of mobile applications devel-

oped by both enthusiastic amateurs and professional developers. As of October 2011, the Apple app store had more than 500K mobile apps with 18 billion downloads. Smartphone applications are different from their desktop counterparts. Unfortunately, mobile application developers often overlook the severe resource constraints of cellular networks, and are usually unaware of the cellular specific characteristics that incur complex interaction with the application behavior. This potentially results in smartphone apps that are not cellular-friendly, *i.e.,* their bandwidth usage, radio channel utilization and energy consumption are inefficient. For example, we found that by improving the data transfer scheduling mechanism of professionally developed popular mobile apps such as Facebook and Pandora, their radio energy consumption can be reduced by up to 30% [4].

My thesis is dedicated to address both challenges, aiming at *providing practical, effective, and efficient methods to monitor and to reduce the resource utilization and bandwidth consumption in cellular networks.* We leverage the implications of the underlying resource management mechanism to balance the key tradeoffs in cellular data networks, from perspectives of the carrier, the mobile applications, and their complex interactions. I elaborate the contributions of my dissertation in the following four sections.

1.1 *Measuring the State of the Art*: Characterizing Radio Resource Utilization for Cellular Networks

Understanding the current resource utilization for commercial cellular networks is the very first necessary step towards optimizing them. To achieve this goal, we collected cellular data of hundreds of thousands of 3G users from the core network of large cellular carrier in the U.S., then replayed the network traces against a novel RRC (Radio Resource Control) state machine simulator to obtain detailed statistics about radio resource utilization. To the best of my knowledge, our work is the first empirical study that investigates the optimality of cellular resource management policy using real cellular traces.

In a cellular system, a handset can be in one of several RRC states (*e.g.,* a high-power state, a low-power state, and an idle state), each with different amount of allocated radio resources. The state transitions also have significant impact on the cellular network and the handset energy consumption: state promotions (resource allocation) incur signaling load and state demotions (resource release) are controlled by critical inactivity timers.

The RRC state machine is the key for cellular resource management but it is hidden from the mobile applications. This motivated me to design algorithms to accurately infer it through a light-weight probing scheme, then systematically characterize the impact of operational RRC state machine settings. The key observation is that the radio resource utilization is surprisingly inefficient: up to 45% of the occupation time of the high-speed transmission channel is wasted on the idle time period matching the inactivity timer value, which is called *tail time*, before releasing radio resources [5]. We further explored the optimal state machine settings in terms of several critical timer values evaluated using real network traces. My findings revealed that the fundamental limitation of the current cellular resource management mechanism is its *static nature of treating all traffic according to the same RRC state machine*, making it difficult to balance tradeoffs among the radio resource usage efficiency, the signaling load, the handset radio energy consumption, and the performance. Such an important observation drove me to delve into the optimization of cellular resource utilization described below.

1.2 *Exposing the Visibility*: Profiling Smartphone Apps for Identifying Inefficient Resource Usage

From cellular customers' perspective, as mentioned before, there remain far more challenges associated with mobile applications compared to their desktop counterparts, leading to smartphone applications that are not cellular-friendly, *i.e.,* their radio channel utilization and handset energy consumption are inefficient because of a lack of transparency in the

lower-layer protocol behavior. To fill such a gap, we developed a novel data analysis and visualization framework called ARO (mobile **A**pplication **R**esource **O**ptimizer) [6]. ARO is the first tool that *exposes the cross-layer interaction* for layers ranging from higher layers such as user input and application semantics down to the lower protocol layers such as HTTP, transport, and very importantly radio resources. Correlating behaviors of all these layers helps reveal inefficient resource usage due to a lack of transparency in the lower-layer protocol behavior, leading to suggestions for improvement.

One key observation is that, from applications' perspective, given the aforementioned static nature of the current resource management policy, the low resource efficiency in cellular networks is fundamentally attributed to *short traffic bursts* carrying small amount of user data while interleaved with long idle periods during which a handset keeps the radio channel occupied. ARO employs a novel algorithm to identify them and to distinguish which factor triggers each such burst, *e.g.,* user input, TCP loss, or application delay, by synthesizing the cross-layer analysis results. Discovering such triggering factors is crucial for understanding the root cause of inefficient resource utilization.

ARO revealed that many popular applications (Pandora, Facebook, Fox News *etc.*) have significant resource utilization inefficiencies that are previously unknown. For example, for Pandora, a popular music streaming application on smartphones, due to the poor interaction between the RRC state machine and the application's data transfer scheduling mechanism, 46% of its radio energy is spent on periodic audience measurements that account for only 0.2% of received user data. Improving the data transfer scheduling mechanism can reduce its radio energy consumption by 30%.

1.3 *Enabling the Cooperation*: Optimizing Radio Resource Usage Through Adaptive Resource Release

We have investigated the resource optimization problem from perspectives of the network and customer applications, respectively. As mentioned before, analyses from both sides indicate the resource inefficiency origins from the *release* of radio resources controlled by static inactivity timers. The timeout value itself, known as the *tail time*, can last for more than 10 seconds, leading to significant waste in radio resources of the cellular network and battery energy of user handsets. Naively decreasing the timer is usually not an option because it may significantly increase the signaling load.

Therefore, to eliminate tail times, we need to change the way resources are released *from statically to adaptively*. This requires the cooperation between the network and handsets, since the latter have the best knowledge of application traffic patterns determining resource allocation and release. Towards this goal, we proposed **T**ail **O**ptimization **P**rotocol (TOP), a cooperative resource management protocol that eliminates tail times [7]. Intuitively, applications can often accurately predict a long idle time. Therefore a handset can notify the network on such an imminent tail, allowing the network to *immediately* release resources. However, doing so aggressively may incur unacceptably high signaling load. TOP employs a set of novel algorithms to address this key challenge by *(i)* letting individual applications predict tails and *(ii)* designing an efficient and effective scheduling algorithm that coordinates tail prediction of concurrent applications. The handset requests for immediate resource release only when the combined idle time prediction across all applications is long.

Interestingly, we found that the basic building block for realizing TOP is already supported by most cellular networks. It is a recent proposal of 3GPP specification called fast dormancy [8], a mechanism for a handset to request for an immediate RRC state demotion. TOP thus requires no change to the cellular infrastructure or the handset hardware given

that fast dormancy is widely deployed. The experimental results based on real traces of a commercial cellular network showed that with reasonable prediction accuracy, TOP saves the overall radio energy (17%) and radio resources (14%) by reducing tail times by up to 60%. For applications such as multimedia streaming, TOP can achieve more significant savings of radio energy (60%) and radio resources (50%).

1.4 *Reducing the Footprint*: Eliminate Redundant Data Transfers in Cellular Data Networks

Another important topic in cellular networks is to reduce the amount of data transferred without compromising the application semantics. Compared to wired and Wi-Fi networks, this issue is particularly critical in cellular networks. From carriers' perspective, cellular networks operate under severe resource constraints. Even a small reduction of the total traffic volume by 1% leads to savings of tens of millions of dollars for carriers [2]. The benefits are also significant from customers' perspective, as fewer network data transfers cut cellular bills, improve user experience, and reduce handset energy consumption.

There are multiple ways to achieve this goal, such as caching, compression, and offloading transfers to Wi-Fi. We have performed the first network-wide study of HTTP caching on smartphones, because HTTP traffic generated by mobile browsers and smartphone applications far exceeds any other type of traffic. Also caching on handsets (compared to caching in the network) is particularly important as it eliminates all network-related overheads.

Our study focuses on redundant transfers caused by *inefficient handset web caching implementation* [9]. We used a dataset collected from 3 million smartphone users of a commercial cellular network, as well as another five-month-long trace contributed by 20 smartphone users at the University of Michigan. Surprisingly, our findings suggest that redundant transfers contribute 18% and 20% of the total HTTP traffic volume in the two

datasets. Even at the scope of *all* cellular data traffic, they are responsible for 17% of the bytes and 9% of the radio resource utilization. As confirmed by our local experiments, most of such redundant transfers are caused by the smartphone web caching implementation that does not fully support or strictly follow the protocol specification, or by developers not fully utilizing the caching support provided by the libraries. Our finding suggested that improving the cache implementation on handsets will bring considerable reduction of network traffic volume, cellular resource consumption, handset energy consumption, and user-perceived latency, benefiting both cellular carriers and customers.

1.5 Thesis Organization

This dissertation is structured as follows. Chapter II provides sufficient background of resource allocation mechanism in cellular networks. In Chapter III, we describe four tools for cellular network analysis: the RRC state machine inference, the trace-driven RRC state inference, the radio power model, and the methodology for quantifying resource consumption. These analyses will be used in the rest of the dissertation as building blocks. The goal of Chapter IV is to understand the state-of-the-art of cellular resource utilization by performing network-wide measurement for a commercial cellular network. Then in Chapter V, we describe our approach of profiling mobile applications to reveal their inefficient resource usage due to potentially poor interactions among multiple layers. Motivated by the observations of Chapter IV and Chapter V, we propose a novel cellular resource management framework that enables adaptive resource release in Chapter VI. Next, in Chapter VII, we present the first network-wide study of HTTP caching on smartphones, to quantify the network data redundancy caused by disparities between protocol specification and implementation. We discuss related work in Chapter VIII before concluding the thesis in Chapter IX.

CHAPTER II

Background

This chapter provides sufficient background of resource allocation mechanism in cellular networks.

To efficiently utilize the limited resources, cellular networks employ a resource management policy distinguishing them from wired and Wi-Fi networks. In particular, there exists a radio resource control (RRC) state machine [5] that determines radio resource usage based on application traffic patterns, affecting device energy consumption and user experience. Similar RRC state machines exist in different types of cellular networks such as UMTS (Universal Mobile Telecommunications System) [5], EvDO (Evolution-Data Optimized) [10] and 4G LTE (3GPP Long Term Evolution) networks [11] although the detailed state transition models may differ. The description below focuses on the popular 3G UMTS networks (§2.1), which was the state-of-the-art cellular access technology during the course of my research (2008 to 2012). We briefly discuss the 4G LTE networks in §2.2. To highlight the uniqueness and the impact of the cellular resource management policy, we measure and compare the radio energy overhead of 3G and Wi-Fi for small data transfers in §2.3. We present the key tradeoffs of resource management in cellular networks in §2.4, and describe a new feature called fast dormancy, a mechanism allowing a handset to bypass the default resource management policy, in §2.5.

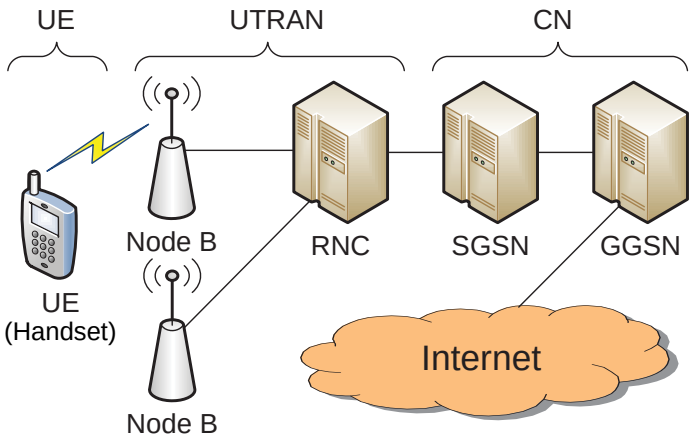


Figure 2.1: The UMTS architecture

2.1 Radio Resource Management in 3G UMTS Networks

We first give a brief overview of the 3G UMTS networks, then describe how radio resources are managed in UMTS networks.

2.1.1 The UMTS Network

As illustrated in Figure 2.1, the UMTS network consists of three subsystems: User Equipments (UE, or handsets), UMTS Terrestrial Radio Access Network (UTRAN), and the Core Network (CN). UEs are essentially mobile handsets carried by end users. We use the term "handset" instead of "UE" throughout this dissertation.

The UTRAN allows connectivity between a handset and the CN. It consists of two components: base stations, called Node-Bs, and Radio Network Controllers (RNC), which control multiple Node-Bs. Most UTRAN features such as packet scheduling, radio resource control, and handover control are implemented at the RNC. The centralized CN is the backbone of the cellular network. In particular the GGSN (Gateway GPRS Support Node) within the CN serves as a gateway hiding UMTS internal infrastructures from the external network.

2.1.2 The RRC States

In the context of UMTS, the *radio resource* refers to WCDMA codes that are potential bottleneck resources of the network. To efficiently utilize the limited radio resources, the UMTS radio resource control (RRC) protocol introduces a state machine associated with each handset. There are typically three RRC states as described below [12, 3].

IDLE. This is the default state when a handset is turned on. The handset has not yet established an RRC connection with the RNC, thus no radio resource is allocated, and the handset cannot transfer any user data (as opposed to control data). The only allowed message is control messages sent through a shared control channel for initializing the RRC connection. Some UMTS networks support a hibernating state called CELL_PCH. It is similar to IDLE but the state promotion delay from CELL_PCH is shorter.

CELL_DCH. The RRC connection is established and a handset is usually allocated dedicated transport channels in both downlink (DL, RNC to handset) and uplink (UL, handset to RNC) direction. This state allows a handset to fully utilize radio resources for user data transmission. We refer to CELL_DCH as DCH henceforth.

A handset can access HSDPA/HSUPA (High Speed Downlink/Uplink Packet Access) mode, if supported by the infrastructure, at DCH state. For HSDPA, the high speed transport channel is not dedicated, but shared by a limited number (*e.g.,* 32) of users [3]. Further, when a large number of handsets are in DCH state, the radio resources may be exhausted due to the lack of channelization codes in the cell. Then some handsets have to use low-speed shared channels although their RRC states are still DCH.

CELL_FACH. The RRC connection is established but there is no dedicated channel allocated to a handset. Instead, the handset can only transmit user data through shared low-speed channels that are typically less than 15kbps. We refer to CELL_FACH as FACH from this point on. FACH is designed for applications requiring very low data throughput rate. It consumes much less radio resources than DCH does.

RRC states impact a handset's energy consumption. A handset at IDLE consumes al-

most no energy from its radio interface. The radio power consumption for DCH is 50% to 100% higher than that for FACH (Table 3.1). While within the same state, the radio power is fairly stable regardless of the data throughput when the signal strength is stable. Further, the RRC state machine is maintained at both the handset and the RNC. The two peer entities are always synchronized via control channels except during transient and error situations. Also note that both the downlink (DL) and the uplink (UL) use the same state machine.

Some UMTS networks support a hibernating state called CELL_PCH. It is similar to IDLE but the state promotion delay from CELL_PCH is shorter [3].

2.1.3 State Transitions

In the RRC state machine, there are two types of state transitions. *State promotions*, including IDLE→FACH, IDLE→DCH, and FACH→DCH, switch from a state with lower radio resource and handset energy utilization to another state consuming more resources and handset energy. *State demotions*, consisting of DCH→FACH, FACH→IDLE, and DCH→IDLE, go in the reverse direction. Depending on the starting state, a state promotion is triggered by either any user data transmission activity, if the handset is at IDLE, or the per-handset queue size, called Radio Link Controller (RLC) buffer size, exceeding a threshold in either direction, if the handset is at FACH.

The state demotions are triggered by two inactivity timers maintained by the RNC. We denote the DCH→FACH timer as α, and the FACH→IDLE timer as β. At DCH, the RNC resets the α timer to T seconds, a fixed threshold, whenever it observes any UL/DL data frame. If there is no user data transmission activity for T seconds, the α timer times out and the state is demoted to FACH. A similar scheme is used for the β timer for the FACH→IDLE demotion. We use the notions of α and β throughout the dissertation. Note that such a timeout mechanism is widely used in computer and networking systems in order to save limited resources. Examples include suspending a system component (*e.g.,* disk) or

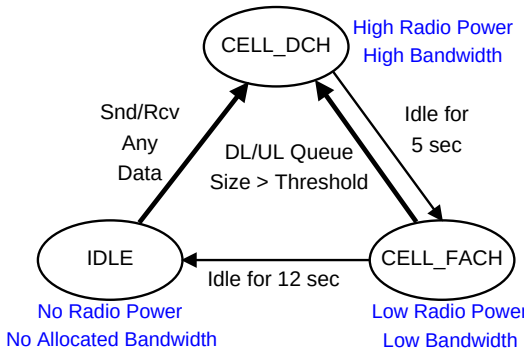


Figure 2.2: The RRC state machine for the 3G UMTS network of Carrier 1

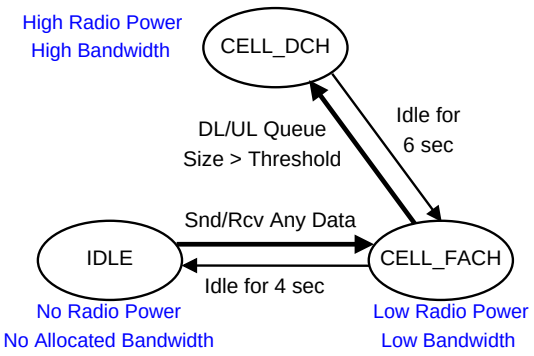


Figure 2.3: The RRC state machine for the 3G UMTS network of Carrier 2

closing a TCP connection after an idle time period. They however incur different tradeoffs that need to be analyzed and optimized separately.

Promotion Delays and Tail Times distinguish cellular networks from other types of access networks. An RRC state promotion incurs a long latency (up to several seconds) during which tens of control messages are exchanged between a handset and the RNC for resource allocation. A large number of state promotions incur high signaling overhead as they increase processing load at the RNC and worsen user experience [13]. In contrast, state demotions finish much faster, but they incur *tail times* that cause significant waste of resources [14, 5, 7]. A *tail* is the idle time period matching the inactivity timer value before a state demotion. During a tail time, a handset simply waits for the inactivity timer to expire, but it still occupies transmission channels and WCDMA codes, and its radio power consumption is kept at the corresponding level of the state. Due to the tail time, transmitting even small amount of data can cause significant radio energy and radio resource consumption.

Figures 2.2 and 2.3 depict the state machine models for two large UMTS carriers in the U.S., Carrier 1 and Carrier 2, respectively, based on our inference methodology described in §3.1. We will refer to these two carriers in later chapters. Their difference naturally introduces the problem of seeking the optimal state machine configuration to better balance radio resource utilization and performance. We compare both carriers in §4.4.4.

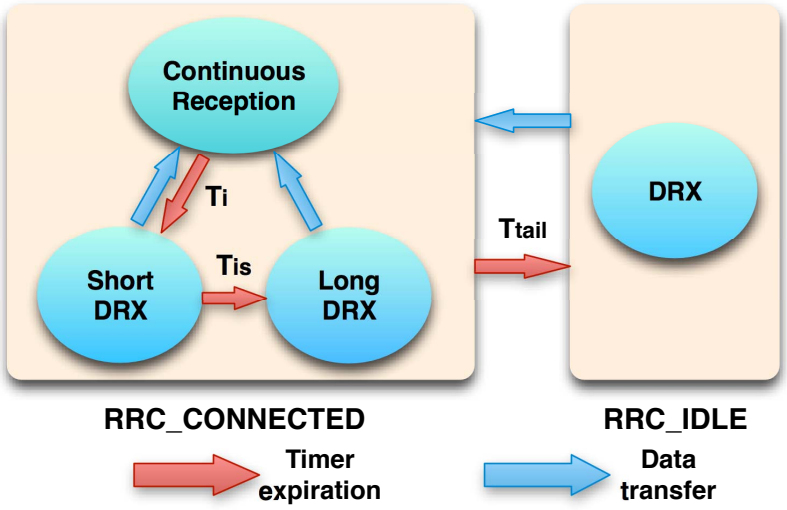


Figure 2.4: RRC state machine of LTE network

2.2 Radio Resource Management in 4G LTE Networks

We describe the RRC state machine in 4G LTE networks [15, 16]. As shown in Figure 2.4, it has a similar concept but differs from the 3G UMTS RRC state machine in two ways: *(i)* there exist only two RRC states: `RRC_CONNECTED` and `RRC_IDLE`, *(ii)* within `RRC_CONNECTED`, there are three microstates: Continuous Reception, Short DRX, and Long DRX. DRX stands for "discontinuous reception", a new feature that allows a handset to periodically wake up to check the downlink transmission channel, thus reducing its radio energy consumption. Specifically, as shown in Figure 2.5, the channel access in DRX mode consists of consecutive DRX cycles. Time slots in each DRX cycle belong to either an *on duration* or a *sleep duration* (unshaded slots in Figure 2.5). In an *on duration*, the handset monitors the Physical Downlink Control Channel (PDCCH) for any incoming data, while the handset simply hibernates in the much longer *sleep duration*, in order to save the energy consumed by the radio interface. The difference between Short DRX and Long DRX is the length of the *sleep duration* in each DRX cycle. Clearly, DRX incurs tradeoffs between the latency and the energy consumption. Increasing the *sleep duration* saves the radio energy but worsens the latency as the handset wakes up less frequently to check the

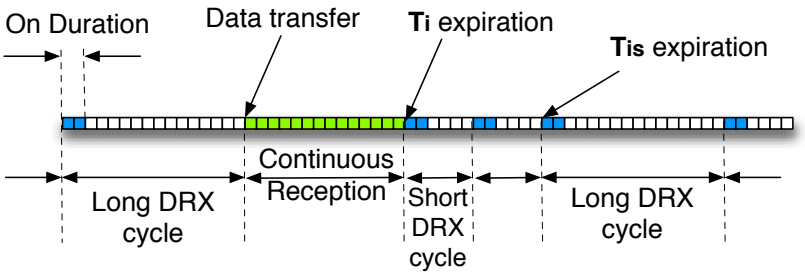


Figure 2.5: Illustration of the LTE DRX in **RRC_CONNECTED**

downlink channel. UMTS 3G does not use DRX at the DCH or FACH state so the handset has to continuously monitor the downlink channel. But DRX is used at the **RRC_IDLE** state for both 3G UMTS and 4G LTE.

For state transitions, the promotion from **RRC_IDLE** to **RRC_CONNECTED** is triggered by an uplink or downlink packet of any size, which also triggers transitions from the two DRX modes to the Continuous Reception mode within the three microstates. Demotions from **RRC_CONNECTED** to **RRC_IDLE**, as well as from Continuous Reception to Short DRX, then to the Long DRX mode are controlled by three inactivity timers. It is important to note that in 4G LTE networks, both the tail time caused by the inactivity timer from **RRC_IDLE** to **RRC_CONNECTED**, and the promotion overhead from **RRC_CONNECTED** to **RRC_IDLE** still exist. Both factors incur the fundamental tradeoffs among the radio resource utilization, the handset radio energy consumption, and the signaling overhead to be discussed in §2.4. Please refer to our work [11] for more detailed discussion of resource management in 4G LTE networks.

2.3 Comparing Resource Consumption: 3G vs. Wi-Fi

To highlight the uniqueness and the impact of the cellular resource management policy, we measure and compare the radio energy overhead of 3G and Wi-Fi for *small data transfers*, on which the resource impact of the RRC state machines is particularly high.

For 3G (we use Carrier 1's UMTS network, see Figure 2.2), we assume the handset is

on IDLE before transmitting a burst. Therefore the total radio energy consists of four parts: E_{Promo} (the IDLE→DCH promotion energy), $E_{\text{3G-Data}}$ (the energy for transferring the actual data), $E_{\text{DCH-Tail}}$ (the DCH tail energy), and $E_{\text{FACH-Tail}}$ (the FACH tail energy). For Wi-Fi, the radio energy consists of $E_{\text{WiFi-Data}}$ and $E_{\text{WiFi-Tail}}$, which are the energy for the data and the short tail, respectively. Our measurement on Google Nexus One using a power monitor [17] indicates that the Wi-Fi energy consumption for a data transfer is largely proportional to the transfer size when the signal is stable. The radio power for Wi-Fi is lower than that for 3G, and Wi-Fi incurs no observable promotion delay but a much shorter tail time around 250 ms. Similar observations were reported on a Nokia N95 phone in [14].

We measured radio energy consumption for small data transfers by performing controlled local experiments. We set up an HTTP server hosting two small objects A and B, whose sizes are 1KB and 9KB, respectively. Then we used a Google Nexus One phone to fetch both objects for 20 times, making sure no caching. Meanwhile, we recorded both packet traces and power traces (using a Monsoon power monitor [17]) so that the energy consumption of each component (promotion, data and tail) can be accurately computed by correlating power traces with packet traces. All experiments were performed when the signal strength was good and stable. To determine the radio interface power, we tried to keep other device components consuming constant power (*e.g.,* keeping the LCD at the same brightness level). Then the radio interface power, which contributes at least 50% of the total handset power [18], can be approximated by subtracting the constant power baseline (420 mW) from the overall power reported by the power monitor.

Figure 2.6 plots the energy breakdown. For transferring Object A (Object B), 97.0% (94.3%) of radio energy belongs to E_{Promo}, $E_{\text{FACH-Tail}}$, or $E_{\text{DCH-Tail}}$. As indicated by the three short bars on the right of Figure 2.6, the Wi-Fi energy consumption is significantly less than that for 3G, because *(i)* Wi-Fi has smaller RTT and higher data rate than 3G, thus the data transfer time for Wi-Fi is much shorter, *(ii)* the Wi-Fi radio power (300 mW) is also smaller than 3G (650 mW), and *(iii)* Wi-Fi has a much shorter tail time (250 ms) and

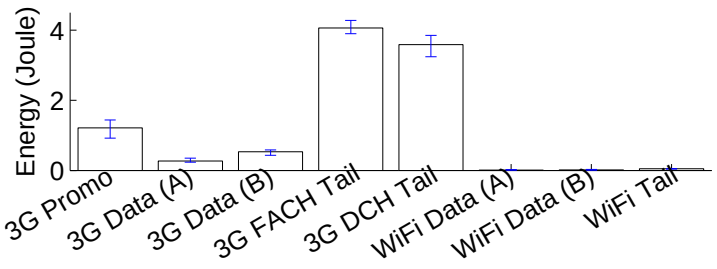


Figure 2.6: Radio energy breakdown for transmitting a small burst. A and B are two small HTTP objects of 1KB and 9KB, respectively.

negligible state promotion delay. In our experiments, $E_{\text{3G-Data}}$ is 22 (31) times of $E_{\text{WiFi-Data}}$ for transferring Object A (B). When promotion and tail energy are taken into account, the disparity can be even as high as 140 times. All three factors result in a 22x (31x) energy difference between 3G and Wi-Fi for transferring Object A (Object B). The disparity is as large as 140x if promotion and tail energy are taken into account. The measurement results indicate that the state promotion and tail time incur significant energy overhead for transmitting a small burst in cellular networks.

2.4 Tradeoffs in Optimizing Resource Allocation

The RRC state machine introduces tradeoffs among radio resource utilization, handset energy consumption, end user experience, and management overheads at the RNC. We need to quantify these factors to analyze the tradeoff. Given a cellular trace and a state machine configuration C, we compute three metrics to characterize the above factors. Previous work either consider only one factor [19, 20] or focus on other metrics (*e.g.,* dropping rate due to congestion [21] and web page response time [22]), using analytical models. We detail our methodology for computing the three metrics in §3.4.

- The DCH state occupation time, denoted by $D(C)$, quantifies the overall radio resources consumed by handsets on dedicated channels in DCH state. We ignore the relatively low radio resources allocated for shared low-speed channels on FACH.

Table 2.1: Optimize radio resources: the key tradeoff

Increase α or β timers	**Decrease** α or β timers
ΔD increases *Increase* tail time *Waste* radio resources	ΔD decreases *Decrease* tail time *Save* radio resources
ΔS decreases *Reduce* state promotions *Reduce* RNC overhead *Improve* user experiences	ΔS increases *Increase* state promotions *Increase* RNC overhead *Degrade* user experiences
ΔE increases *Waste* handset radio energy	ΔE decreases *Save* handset radio energy

- The signaling overhead, denoted by $S(C)$, is the total delay of IDLE→DCH, IDLE→FACH, and FACH→DCH promotions. $S(C)$ quantifies the overhead brought by state promotions that worsen user experience and increase the management overhead at the RNC. We ignore the state demotion overhead as it is significantly smaller compared with the state promotion overhead.

- The energy consumption, denoted by $E(C)$, is the energy consumed by the cellular radio interface whose radio power contributes 1/3 to 1/2 of the overall handset power during the normal workload [6].

We are interested in relative changes of D, S, E when we switch to a new state machine using the same trace. Let C be the default state machine used as the comparison baseline, and let C' be a new state machine configuration. The relative change of D, denoted as ΔD, is computed by $\Delta D(C') = (D(C') - D(C))/D(C)$. We have similar definitions for ΔS and ΔE.

As mentioned before, ΔE quantifies the relative change of the energy consumed by the handset radio interface. We are also interested in the relative change of the overall handset energy, denoted by ΔE_{all}. Note that ΔE_{all} is slightly different from ΔE when $\Delta S \neq 0$. In that case, the total duration of a trace changes by $|\Delta S|$ due to the increased (if $\Delta S > 0$) or decreased (if $\Delta S < 0$) promotion delay. Therefore the total handset energy consumption for the new trace should include (exclude) the energy consumed by

the non-radio components during such additional (removed) periods of state promotions whose total duration is ΔS (the radio energy consumed during ΔS is already included in ΔE). However, since in most cases ΔS is much shorter than the total trace duration, ΔE is a good estimation of ΔE_{all}.

As we shall see in Chapter IV, the key tradeoff expressed in our notations is that, for any state machine setting, increasing ΔS causes both ΔD and ΔE to decrease (there may exist exceptions when ΔS is too large). In other words, if more state promotions are allowed, then we can save more radio resources and handset energy. Ideally, we want to find a state machine configuration C' such that $\Delta D(C')$ and $\Delta E(C')$ are significantly negative, while $\Delta S(C')$ is reasonably small. This important tradeoff is summarized in Table 2.1.

2.5 Fast Dormancy

The fundamental reason why inactivity timers are necessary is that the network has no easy way of predicting the network idle time of a handset. Therefore the RNC conservatively appends a tail to *every* network usage period. This naturally gives rise to the idea of letting mobile applications determine the end of a network usage period since they can make use of application knowledge useful for predicting network activities. Once an imminent tail is predicted, a handset notifies the RNC, which then immediately releases allocated resources.

Based on this simple intuition, a feature called *fast dormancy* has been proposed to be included in 3GPP Release 7 [23] and Release 8 [24]. The handset sends an RRC message, which we call the $\boldsymbol{T}$ message, to the RNC through the control channel. Upon the reception of a $\boldsymbol{T}$ message, the RNC releases the RRC connection and lets the handset go to IDLE (or to a hibernating state that has lower but still non-trivial promotion delay). This feature is supported by several handsets [24]. To the best of our knowledge, no smartphone application uses fast dormancy in practice, partly due to a lack of the OS support that provides a simple programming interface.

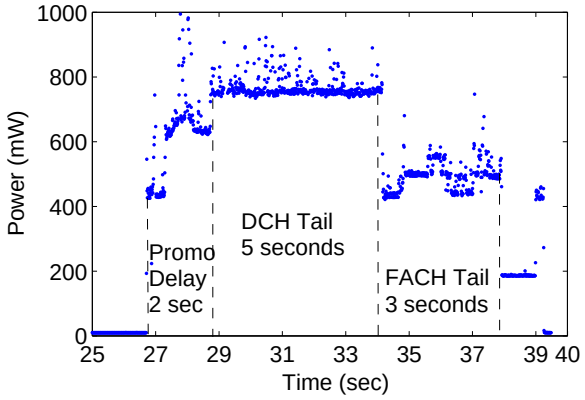


Figure 2.7: Shorter tail times of a Nexus One phone using fast dormancy. A single UDP packet is sent at $t = 26.8$ sec.

However, based on measuring the device power consumption, we do observe that a few phones adopt fast dormancy in an *application-agnostic* manner: the handset goes to IDLE (or to the hibernating CELL_PCH state with a lower promotion delay) faster than other phones do for the same carrier. In other words, they use a shorter inactivity timer controlled by the device in order to lengthen the battery life. The disadvantage of such an approach is well understood [5, 13]: the additionally incurred state promotions may introduce significant signaling overhead at the RNC and may worsen user experience.

For example, we investigated four handsets using Carrier 1's UMTS network (Table 3.1): HTC TyTN II, Sierra 3G Air card, and two Google Nexus One phones (A and B). For TyTn II, the air card, and Nexus One A, their state demotions are solely controlled by inactivity timers. For Nexus One B, the measured α and β timers are 5 sec and only 3 sec (shorter than the default 12-sec β timer), respectively. Such an observation is further validated by measuring the power of Nexus One B (Figure 2.7). It is highly likely that it employs fast dormancy to release radio resources earlier to improve its battery life. In contrast, Figure 3.7 shows the default tail time (of the same carrier) on an HTC TyTn II phone that does not use fast dormancy. Both Figure 2.7 and Figure 3.7 are measured using a power monitor [17]. We detail the measurement methodology in §3.1.3.

We believe that currently fast dormancy is controlled by the upgradable radio image

that distinguishes the two Nexus One phones (Nexus One A and B are identical except for the radio image version). Again, the incurred drawbacks of fast dormancy are extra state promotions causing additional signaling overhead and potentially worsen user experience [13, 5].

CHAPTER III

Tools for Cellular Network Analysis

In this chapter, we describe four tools for cellular network analysis: the RRC state machine inference (§3.1), the trace-driven RRC state inference (§3.2), the radio power model (§3.3), and the methodology for quantifying cellular resource consumption (§3.4). They are the necessary methodologies for carrying out the experimental work or the smartphone traffic analysis in the rest of the dissertation. Specifically, we have made the following contributions in this chapter.

- In §3.1, we propose a novel inference technique for the RRC state machine model purely based on probing from the user device. It systematically discovers the state transitions by strategically adjusting the packet dynamics. We apply our algorithm to two UMTS carriers and validated its accuracy by measuring the device power consumption.
- In §3.2, we present a methodology that accurately infers RRC states from packet traces collected on a handset. The inference technique is necessary due to lacking of an interface for accessing RRC states directly from the handset hardware.
- In §3.3, we describe a simple but robust power model to estimate the UMTS radio energy consumption.

- In §3.4, we design a novel methodology, which leverages the techniques proposed in §3.2 and §3.3, for quantifying the resource consumption for cellular traces.

3.1 RRC State Machine Inference

We propose an end-host based probing technique for inferring the cellular state machine, which we validate using power measurements. Accurate inference of the state machine and its parameters is the first necessary step towards characterizing and improving the RRC state machine. The probing technique is based on 3G UMTS network, but it can be easily generalized to other types cellular networks such as 4G LTE.

3.1.1 Inference Methodology

We present our methodology for inferring the RRC state machine and its parameters.

3.1.1.1 Basic Assumptions

We make the following assumptions for the inference algorithm. *(i)* There are at most three states: IDLE, FACH, and DCH. DCH is the state allowing high data rate transfer. *(ii)* The time granularity for inactivity timers is assumed to be seconds. Our algorithms can easily adapt to finer granularities. *(iii)* The state promotion delay is significantly longer than (at least two times as) a normal RTT for both DCH and FACH (less than 300 ms based on our measurements). This is reasonable due to the promotion overhead explained earlier. *(iv)* We roughly know the range of RLC buffer thresholds (64B–1KB) that trigger the FACH→DCH promotion. We detail our methodology below.

3.1.1.2 State Promotion and Demotion Inference

State promotion inference determines one of the two promotion procedures adopted by UMTS: $P1$: IDLE→FACH→DCH, or $P2$: IDLE→DCH. Algorithm 1 illustrates how

Algorithm 1 State promotion inference

1: Keep the handset on IDLE.
2: The handset sends min bytes. Server echoes min bytes.
3: The handset sends max bytes. Server echoes min bytes.
4: The handset records the RTT Δt for Step 3.
5: Report $P1$ iff $\Delta t \gg$ normal RTT. Otherwise report $P2$.

we distinguish between $P1$ and $P2$, where min and max denote RLC buffer sizes that does not trigger, and does trigger, the FACH→DCH promotion, respectively. Note that IDLE→DCH or IDLE→FACH always happens regardless of the RLC buffer size. The idea is to distinguish $P1$ and $P2$ by detecting the presence of the FACH→DCH promotion. We set min and max to 28 bytes (an empty UDP packet plus an IP header) and 1K bytes, respectively. If $P1$ holds, then the state is promoted to FACH after Step 2, and then further promoted to DCH at Step 3. Thus Δt includes an additional FACH→DCH promotion delay. Otherwise, for $P2$, Δt does not include the promotion delay since the state is already DCH after Step 2.

Algorithm 2 State demotion inference

1: **for** $n = 0$ to 30 **do**
2: The handset sends max bytes. Server echoes min bytes.
3: The handset sleeps for n sec.
4: The handset sends min bytes. Server echoes min bytes.
5: The handset records the RTT $\Delta t_1(i)$ for Step 4.
6: **end for**
7: **for** $n = 0$ to 30 **do**
8: The handset sends max bytes. Server echoes min bytes
9: The handset sleeps for n sec.
10: The handset sends max bytes. Server echoes min bytes.
11: The handset records the RTT $\Delta t_2(i)$ for Step 10.
12: **end for**
13: Report $D1$ iff $\Delta t_1(\cdot)$ and $\Delta t_2(\cdot)$ are similar, else report $D2$.

State demotion inference determines whether UMTS uses $D1$: DCH→IDLE or $D2$: DCH→FACH→IDLE. The inference method is shown in Algorithm 2, which consists of two experiments. The first experiment (Steps 1 to 6) comprises of 30 runs. In each run, the handset goes to DCH by sending max bytes (Step 2), sleeps for n seconds (Step 3), then

sends min bytes (Step 4). Recall that min and max denote RLC buffer sizes that does not trigger, and does trigger, the FACH→DCH promotion, respectively. By increasing n from 0 to 30, we fully exercise all the states experienced by the handset at the beginning of Step 4 due to the inactivity timer effects. The second experiment (Step 7 to 12) is similar to the first one except that after Step 10, the handset always promotes to DCH. In contrast, after Step 4, if there exists a DCH→FACH demotion, the handset will be in FACH. Therefore, for $D1$ the observed RTTs for two experiments, $\Delta t_1(0..30)$ and $\Delta t_2(0..30)$, will be similar. On the other hand, for $D2$, *i.e.,* the state is demoted to FACH (the α timer, see §2.1.3), then back to IDLE (the β timer), then for $\lfloor \alpha \rfloor < i \leq \lfloor \alpha + \beta \rfloor$, the difference between $\Delta t_1(i)$ and $\Delta t_2(i)$ is roughly the FACH→DCH promotion delay.

3.1.1.3 Parameter Inference

Given the process of inferring the state transitions, it is easy to infer related parameters. The inactivity timers can be directly obtained from $\Delta t_1(\cdot)$ and $\Delta t_2(\cdot)$ computed by Algorithm 2. For the case where the demotion is DCH→FACH→IDLE, we can deduce α and β from the fact that $\Delta t_1(0...\lfloor \alpha + \beta \rfloor)$ are smaller than $\Delta t_1(\lceil \alpha + \beta \rceil ...30)$, and $\Delta t_2(0...\lfloor \alpha \rfloor)$ are smaller than $\Delta t_2(\lceil \alpha \rceil ...30)$. This is because in the first experiment in Algorithm 2, a state promotion (IDLE→FACH or IDLE→DCH) will not happen until $n \geq \lceil \alpha + \beta \rceil$, while in the second experiment, a state promotion from IDLE or FACH happens when $n \geq \lceil \alpha \rceil$. Similarly, for the case where the demotion is DCH→IDLE, let the only inactivity timer be γ. Then we will observe that $\Delta t_1(0...\lfloor \gamma \rfloor)$ are much smaller than $\Delta t_1(\lceil \gamma \rceil ...30)$.

Promotion Delay. To infer the promotion delay X→Y, we measure the entire RTT including the promotion, then subtract from it the normal RTT (*i.e.,* the RTT not including the promotion) on state Y. The delays are not constant (the variation can be as high as 50%) because the control channel rate, and the workload of the RNC, which processes state promotions, may vary.

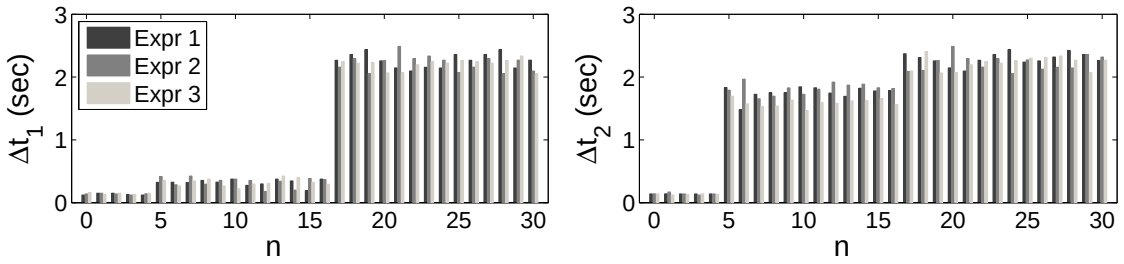
Usually a state promotion is triggered by an uplink packet. But it is worth mention-

ing that an IDLE→DCH/FACH promotion triggered by a downlink packet is usually longer than that triggered by an uplink packet. This is because when a downlink packet is to be received, it may get delayed due to *paging*. In fact, even at IDLE, a handset periodically wakes up to listen for incoming packets on the paging channel. If a downlink packet happens to arrive between two paging occasions, it will be delayed until the next paging occasion. In practice, we observe via power monitor that the paging cycle length is 2.56 sec or 1.28 sec, depending on the configuration of the carrier.

RLC Buffer Thresholds are essential in determining the promotions from FACH to DCH, as described in §2.1.3. We measure the RLC buffer thresholds for uplink (UL) and downlink (DL) separately, by performing binary search for the packet size that exactly triggers the FACH→DCH promotion, using the promotion delay as an indicator.

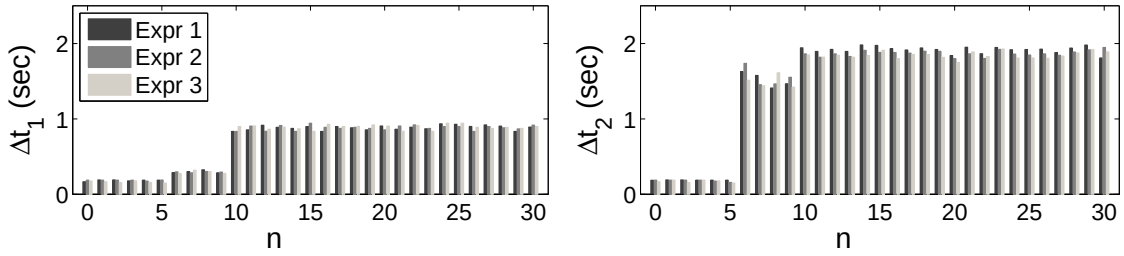
RLC Buffer Consumption Time quantifies how fast the RLC buffer is cleared after it is filled with data at FACH state. It depends on channel throughput at FACH since the RLC buffer is not emptied until all data in the buffer are transmitted [3]. Considering RLC buffer consumption time enables the state inference algorithm (§3.2) to perform more fine-grained simulation of RLC buffer dynamics to more precisely capture state promotions, thus improving the inference accuracy.

We infer the RLC buffer consumption time by sending two packets separated by some delay. First, we send a packet of x bytes at FACH with x smaller than the RLC buffer threshold so it never triggers a FACH→DCH promotion. After a delay for y milliseconds, another packet of z bytes is sent in the same direction. The value of z is chosen in a way that $z <$ the RLC buffer threshold $< x + z$. Then observing a FACH→DCH promotion suggests that the RLC buffer is not yet emptied when the second packet arrives at the buffer, causing the RLC buffer size to exceed the threshold. In other words, at FACH state, the RLC buffer consumption time of x bytes is longer than y milliseconds. On the other hand, not observing a FACH→DCH promotion implies the the RLC buffer consumption time is at most y milliseconds.



(a) Algorithm 2 ($\Delta t_1(\cdot)$) (b) Algorithm 2 ($\Delta t_2(\cdot)$)

Figure 3.1: State machine inference results for Carrier 1



(a) Algorithm 2 ($\Delta t_1(\cdot)$) (b) Algorithm 2 ($\Delta t_2(\cdot)$)

Figure 3.2: State machine inference results for Carrier 2

3.1.2 Results of State Machine Inference

We present the inference results for state machines used by two large UMTS carriers: Carrier 1 and 2 (introduced in §2.1.3). For each carrier, we repeat Algorithm 1 and Algorithm 2 for three times, ensuring that in each experiment *(i)* the server does not experience a timeout; *(ii)* in tcpdump trace, we never observe other user data transmission that may trigger a state transition; *(iii)* the 3G connection is never dropped. The entire experiment is discarded if any of these conditions is violated.

For the state promotion inference, the normal RTTs for Carrier 1 and 2 are less than 0.3 sec, and the measured Δt values in Algorithm 1 are 0.2 sec for Carrier 1, and 1.5 sec for Carrier 2, for all three trials. Based on Algorithm 1, we conclude that the promotion procedures for Carrier 1 and Carrier 2 are IDLE→DCH and IDLE→FACH→DCH, respectively. For the state demotion inference, we notice the qualitative difference between $\Delta t_1(5...16)$ in Figure 3.1(a) and $\Delta t_2(5...16)$ in Figure 3.1(b), indicating that the state demotion pro-

Table 3.1: Inferred parameters for two carriers.

Inactivity timer	Carrier 1	Carrier 2
α: DCH→FACH	5 sec	6 sec
β: FACH→IDLE	12 sec	4 sec
Average promotion delay (triggered by UL packet)	Carrier 1	Carrier 2
IDLE→FACH	N/A	0.6 sec
IDLE→DCH	2.0 sec	N/A
FACH→DCH	1.5 sec	1.3 sec
RLC Buffer threshold	Carrier 1	Carrier 2
FACH→DCH(UL)	540 B	151 B
FACH→DCH(DL)	475 B	119 B
Average State radio power*	Carrier 1	Carrier 2
DCH/FACH/IDLE	800/460/0 mW	600/400/0 mW

* Tested on an HTC TyTn II smartphone. See Table 3.3 for radio power measurement results for more devices.

cedure for Carrier 1 is DCH→FACH→IDLE. Similarly, Figure 3.2(a) and Figure 3.2(b) imply that Carrier 2 also uses DCH→FACH→IDLE, due to the obvious difference between $\Delta t_1(6..9)$ and $\Delta t_2(6..9)$.

We note that for Carrier 1, $\Delta t_1(17...30)$ and $\Delta t_2(17...30)$ are roughly the same, because in Algorithm 2, for $17 \leq n \leq 30$, either sending min bytes (Step 4) or sending max bytes (Step 10) triggers an IDLE→DCH promotion, which is the only promotion transition for Carrier 1. In contrast, for Carrier 2, $\Delta t_1(10...30)$ is smaller than $\Delta t_2(10...30)$. Carrier 2 may perform two types of promotions depending on the RLC buffer size, therefore for $10 \leq n \leq 30$ in Algorithm 2, sending min bytes in Step 4 and sending max bytes in Step 10 will trigger IDLE→FACH and IDLE→FACH→DCH, respectively, resulting in different promotion delays. This observation does not affect the inference results of Algorithm 2 for either carrier.

Given the $\Delta t_1(\cdot)$ and $\Delta t_2(\cdot)$ values computed by Algorithm 2, it is easy to infer α and β by following the logic described in §3.1.1. The inference results are $(\alpha, \beta) = (5sec, 12sec)$ for Carrier 1 and $(\alpha, \beta) = (6sec, 4sec)$ for Carrier 2. To infer the RLC buffer thresholds,

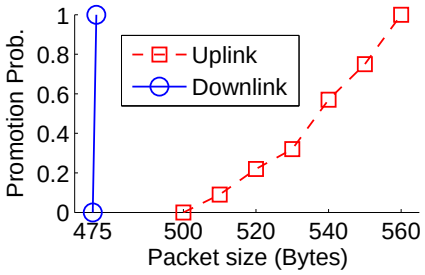


Figure 3.3: RLC buffer thresholds (UL/DL)

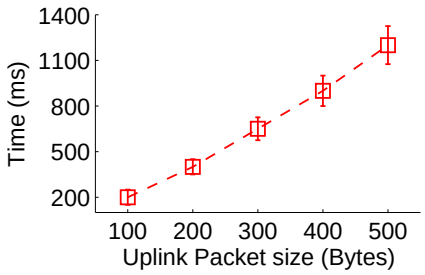


Figure 3.4: RLC buffer consumption time (UL)

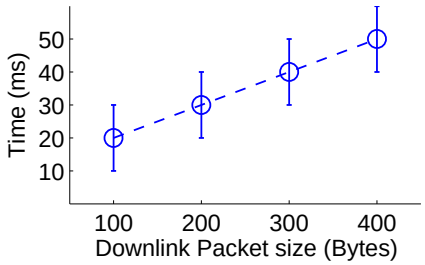


Figure 3.5: RLC buffer consumption time (DL)

we repeat the experiments 30 times and summarize the results in Table 3.1[1].

We further studied the following fine-grained characteristics of Carrier 1's RRC state machine.

1. **Variation of RLC buffer thresholds**. As shown in Figure 3.3 where the Y axis corresponds to the probability of observing a FACH→DCH promotion when a packet of x bytes is sent. We observe that for DL, the threshold is fixed to 475 bytes while the RLC buffer threshold for UL varies from 500 to 560 bytes. Such a difference is likely due to the disparity between the UL and DL transport channels used by the FACH state [25].

2. **RLC Buffer Consumption Time** is measured by using the method described in §3.1.1.3. We fix z at 500 bytes and 475 bytes for uplink and downlink, respectively, according to Figure 3.3. Figure 3.4 shows the RLC buffer consumption time for uplink. For each packet size x (X axis), we vary the delay y (Y axis) at a granularity of 25 ms,

[1]Results in Table 3.1 were measured in November 2009.

and perform aforementioned test for each pair of (x, y) for 20 times. The error bars in Figure 3.4 cover a range of delays (y values) for which we probabilistically observe a promotion. The results for downlink are shown in Figure 3.5. Our results confirm previous measurements that at FACH, the uplink transport channel is much slower than the downlink channel [25].

By considering RLC buffer consumption time, the trace-driven RRC state simulation algorithm (to be described in §3.2) performs more fine-grained simulation of RLC buffer dynamics (both uplink and downlink) to more precisely capture state promotions, because sometimes a FACH→DCH promotion is triggered by multiple small packets that incrementally fill up the RLC buffer, instead of a single large packet with its size exceeding the RLC buffer threshold.

3. **Low traffic volume not triggering timers to reset.** We also found that for Carrier 1, the DCH→FACH timer is not reset when a handset has very little data to transfer for both directions. Specifically, at DCH, a packet P does not reset the timer if both uplink and downlink have transferred no more than 320 bytes (including P) within the past 300 ms. We believe the intent of such a design, which is specific to Carrier 1 and is not documented by literature [3, 26, 25], is to save radio resources in DCH when there is small traffic demand by a handset. In the trace-driven simulation (§3.2), not considering this factor leads to overestimation of the DCH occupation time.

3.1.2.1 Inference Results on 2G Networks

For 2G (GPRS/EDGE) networks, there exists a similar RRC state machine model. The three RRC states are "IDLE", "CELL_SHARED", and "CELL_DEDICATED" [27], corresponding to IDLE, FACH, and DCH in the 3G case, respectively. We applied our inference methodology (using a smaller increment of n in Algorithm 2) on Carrier 2's 2G network, and show the inference results in Figure 3.6. We observe that the inactivity timers (1 sec) are much shorter, therefore resulting in better efficiency of radio re-

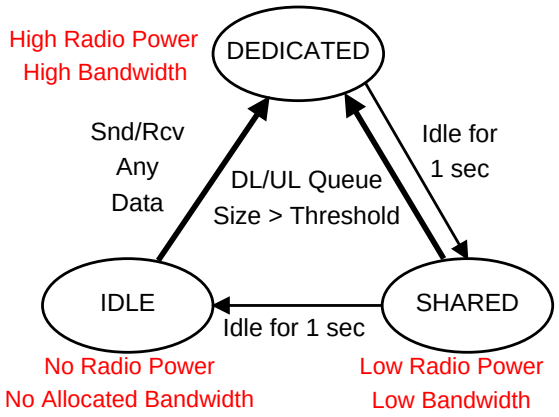


Figure 3.6: The RRC state machine for the 2G GPRS/EDGE network of Carrier 2

source utilization and handset energy consumption. The negative impact of short timers is more frequent state transitions. The state promotion delays of IDLE→DEDICATED and SHARED→DEDICATED are both 0.5 sec.

3.1.3 Validation using Energy Consumption

As described in §2.1.2, the handset radio energy consumption differs for each state, a property we may use to infer the state machine. However, accurately measuring energy consumption requires special monitoring equipments. So we use it as validation for our inference algorithms, which only require handset-based probing.

We set up experiments to confirm the inactivity timers and state promotion delays for Carrier 1 by monitoring the handset energy consumption as follows. The battery of an HTC TyTN II smartphone is attached to a hardware power meter [17], which is also connected via USB to a PC that records fine-grained power measurements by sampling the current drawn from the battery at a frequency of 5000 Hz. Figure 3.7 shows one representative experimental run of the validation. During probing, we keep the handset LCD at the same brightness level, turn off GPS and WiFi, and disable all network activities. After keeping the smartphone in this inactive state for 20 sec, we send a UDP packet at $t = 23.8s$ thus triggering an IDLE→DCH promotion that takes approximately 2 sec as inferred in §3.1.1. From $t = 26.1s$, the phone remains at the high-power DCH state for about 5 sec, then

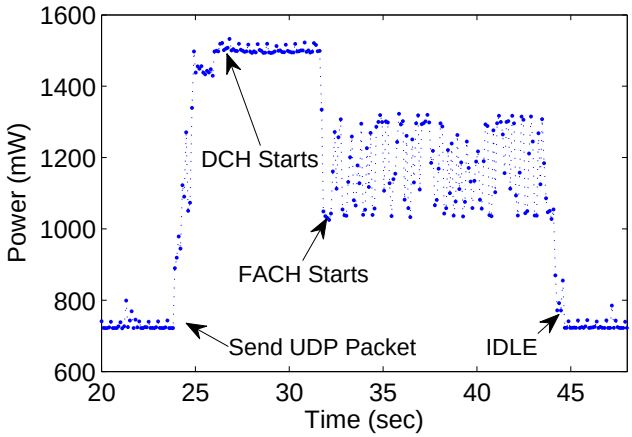


Figure 3.7: Validation using power measurements

switches to the low-power FACH state at $t = 31.5s$. Finally at $t = 44.1s$, the phone returns to the IDLE state. The measured inactivity timer values are longer than the inferred ones by about 10%, likely due to the synchronization overhead between the RNC and the handset. We similarly verified the FACH→DCH promotion delay, and validated Carrier 2's state machines for both 2G and 3G.

Using the IDLE power as baseline, we compute the power consumption of the 3G radio interface as shown in Table 3.1[2]. We also infer the RLC buffer thresholds for the FACH→DCH promotion by performing binary search. Instead of using the promotion delay as described in §3.1.1, we use energy as an indicator to search for the RLC buffer threshold that exactly triggers the promotion, for each direction. The validation results are consistent with our inference results.

3.2 Trace-driven RRC State Inference

We now describe our state inference algorithm, which takes a packet trace $P_1, ..., P_n$ as input where P_i is the i-th packet in a trace collected on a handset. The output is $S(t)$ denoting the RRC state or state transition at any given time t. $S(t)$ corresponds to one of the

[2]The power values in Table 3.1 were measured in good signal strength conditions. [28] shows that signal strength may have significant impact on the handset radio power consumption.

following: IDLE, FACH, DCH, IDLE→FACH, and FACH→DCH. We focus on describing the inference algorithm for Carrier 1 (Figure 2.2) while the technique is also applicable to other carriers using a different state machine through minor modification.

Clearly, this problem can be cleanly solved if the online data collector is able to read the RRC state from the handset hardware. However, we are not aware of any API or known workaround for directly accessing the RRC state information on any smartphone system. In other words, it is difficult to directly observe the low-level communication between a handset and the RNC.

3.2.1 Inference Methodology

The state inference algorithm follows a high-level idea of replaying the packet trace against an RRC state machine simulator, whose the state transition model and parameters can be inferred using the techniques described in §3.1.

The algorithm performs iterative packet-driven simulation. Let P_i and P_{i+1} be two consecutive packets whose arrival time are t_i and t_{i+1}, respectively. Intuitively, if $S(t_i)$ is known, then $\forall t_i < t \leq t_{i+1}$, $S(t)$ can be inferred in $O(1)$ based on three factors, by following the RRC state transition rules.

1. The inter-arrival time between P_i and P_{i+1}, depending on which a handset may experience tail times causing a state demotion then a possible state promotion when P_{i+1} arrives.

2. The packet size of P_{i+1}, which may trigger a FACH→DCH promotion if it fills up the RLC buffer. Considering RLC buffer consumption time (§3.1.1.3) enables the inference algorithm to perform more fine-grained simulation of RLC buffer dynamics to more precisely capture state promotions.

3. The direction of P_{i+1}. Depending on the location where the packet trace is collected, the algorithm behaves differently in terms of inferring a state promotion.

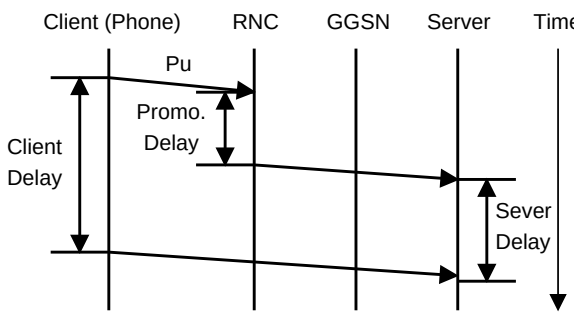


Figure 3.8: State promotion triggered by an UL packet P_u. The data collection point can be either on the phone or at the GGSN.

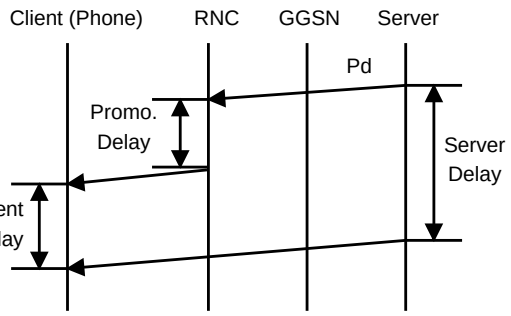


Figure 3.9: State promotion triggered by a DL packet P_d. The data collection point can be either on the phone or at the GGSN.

- The trace is collected at the handset, which is at the *downstream* from the RNC where state promotions take place. Assume that P_{i+1} triggers a promotion. If P_{i+1} is downlink (Figure 3.9), the promotion already finishes when the handset receives it. Therefore P_{i+1} triggers a promotion *before* its arrival. On the other hand, if P_{i+1} is uplink *i.e.,* the application just puts the packet into the uplink RLC buffer, then the state promotion has just begun (Figure 3.8). Therefore the promotion will happen *after* P_{i+1} is captured.
- The trace is collected at the core network (GGSN), which is at the *upstream* from the RNC where state promotions take place. That is just the opposite case. Assume that P_{i+1} triggers a promotion. If P_{i+1} is downlink (Figure 3.9), the promotion will happen after P_{i+1} is captured. Otherwise the promotion has already finished (Figure 3.8).

When $S(t_{i+1})$ is determined, $S(t_{i+2})$ can be iteratively computed based on $S(t_{i+1})$ and P_{i+2}, and so on.

3.2.2 Validation of State Inference

We evaluate our *simulation-based* state inference technique, described in §3.2.1, by comparing it with *handset-power-based* inference approach. We simultaneously collect

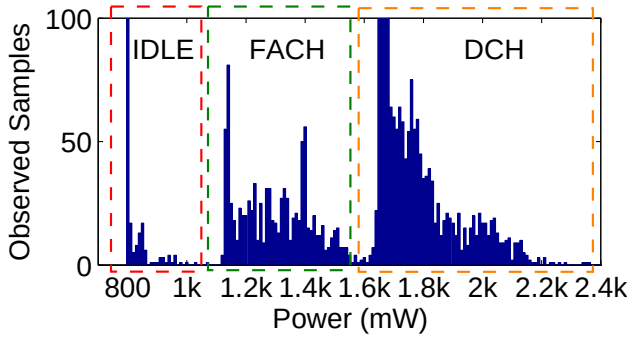


Figure 3.10: Histogram of measured handset power values for the `News1` trace collected on an HTC TyTn II phone

both power traces (using a hardware power monitor [17]) and packet traces for popular websites from an HTC TyTn II smartphone using Carrier 1's UMTS network. We infer the RRC states independently from each trace, and then compare their results. Note that the simulation-based state inference technique does not require any special monitoring equipment.

3.2.2.1 Power-based State Inference

Inferring RRC states from power traces requires special monitoring equipment and is non-trivial due to noise. We next describe a novel algorithm that infers RRC states from a handset's *overall* power consumption, since it is difficult to measure the radio interface power separately. Our basic assumptions are *(i)* a handset's 3G radio interface consumes a considerable fraction of the total handset power [18], and *(ii)* the power consumed at the three RRC states differs significantly (§2.1.2). Both assumptions are confirmed by our experiments (described later in Figure 3.10).

The input generated by the power meter is $P(t)$ describing the overall handset power at a granularity of 0.2 msec. Our power-based state inference algorithm distinguishes RRC states using fixed power thresholds and identifies state transitions by observing power changes. It consists of three steps. *(i)* Downsample $P(t)$ from 5kHz to 10Hz by averaging power values of every 500 power samples to reduce noise. *(ii)* Use two power thresholds μ

Table 3.2: Packet-based vs. power-based inference results

Trace name (Trace Length)	% Time Overlap	State Promotions		
		Agree	Pkt Err	Pwr Err
`News1` (180s)	93.4%	7	1	0
`News2` (180s)	96.5%	9	0	0
`News3` (190s)	96.3%	9	0	0
`News4` (250s)	95.5%	12	0	1
`Social1`(250s)	95.7%	13	0	0
`Social2`(180s)	91.3%	10	0	0
`Social3`(275s)	94.5%	18	1	0
`Email1` (250s)	94.4%	16	0	1
`Email2` (275s)	94.4%	17	0	0
`Stream1`(180s)	98.7%	3	0	0
`Stream2`(180s)	99.1%	2	0	0
Total (2390s)	95.3%	116	2	2

and ν to distinguish the three RRC states: $P(t) < \mu$ for IDLE, $\mu \leq P(t) < \nu$ for FACH, and $\nu \leq P(t)$ for DCH. *(iii)* Identify state transitions by examining power changes crossing the thresholds.

The values of μ and ν are determined from histograms of measured power values of each trace. A representative example for an HTC TyTn II phone is shown in Figure 3.10, from which we observe that the measured overall handset power values form three clusters corresponding to IDLE, FACH, and DCH. The variation within each cluster is mostly due to power change of other system components (*e.g.,* CPU). Figure 3.10 indicates that the 3G radio interface plays a vital role in determining the overall handset power consumption, as at DCH, the radio power (800 mW) contributes 1/3 to 1/2 of the total device power (1600 mW to 2400 mW[3]). All histograms for traces shown in Table 3.2 suggest that we set μ and ν to 1000 mW and 1600 mW, respectively, for TyTn II.

3.2.2.2 Validation Results

We obtained 11 traces listed in Table 3.2 by simultaneously collecting both packet traces and power traces. Then we employ both algorithms (packet-based and power-based) to

[3]As shown in Figure 3.10, the base power of TyTn II is 800 mW as long as the handset is on.

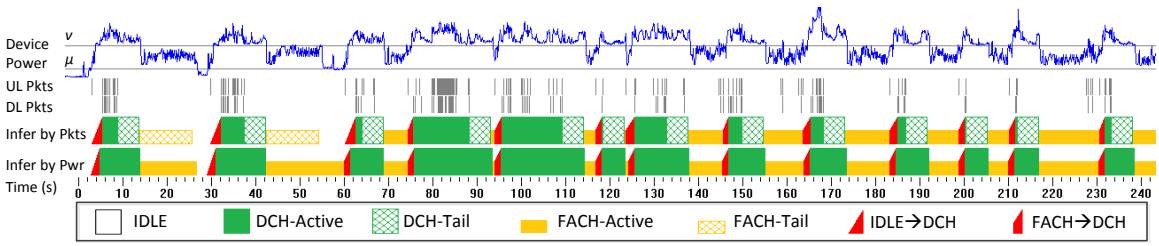


Figure 3.11: Comparing power-based and packet-based state inference results (the `Social1` trace).

infer the RRC states. Table 3.2 shows the comparison results. The "% Time Overlap" column computes the percentage of time periods during which both algorithms produce exactly the same RRC state or the same state promotion.

The right three columns of Table 3.2 compares inferred state promotions. "Agree" counts the number of pairs (X, Y), where promotion X and Y are inferred by the two methods respectively, such that X and Y have the same promotion type and their time periods overlap (it does not require that they match exactly). For a promotion inferred by either algorithm not belonging to such a pair, it is either an error of packet-based inference methodology (counted by the first number of "Pkt Err") or an error of the power-based approach (counted by "Pwr Err") due to noise, judged by our manual inspection. Table 3.2 indicates that both inference methods have comparably high accuracy. We observe that inaccuracies are mostly caused by noises for power-based approach, and variations of uplink RLC buffer thresholds and consumption time, for packet-based inference method.

Figure 3.11 visualizes inference results for the `Social1` trace. The figure consists of five bands (from up to down): measured power curve, uplink packets (each vertical line corresponds to one packet), downlink packets, RRC states inferred by packets (with tails inferred too), and states inferred by power. Different RRC states are visualized by blocks with different shapes, shades, and colors.

Table 3.3: Measured average radio power consumption

	TyTn II Carrier 1	NexusOne Carrier 1	ADP1 * T-Mobile
$P(\mathsf{IDLE})$	0	0	10mW
$P(\mathsf{FACH})$	460mW	450mW	401mW
$P(\mathsf{DCH})$	800mW	600mW	570mW
$P(\mathsf{FACH}\rightarrow\mathsf{DCH})$	700mW	550mW	N/A
$P(\mathsf{IDLE}\rightarrow\mathsf{DCH})$	550mW	530mW	N/A

* Reported by [18] on an Android HTC Dream phone

3.3 Radio Power Model

We use a simple but robust power model to estimate the UMTS radio energy consumption using the inferred RRC states introduced in §3.2. It is calculated by associating each RRC state or state promotion an average power value measured from a particular phone, assuming in 3G networks, these average values are representative [18, 5] (also see Figure 3.11) despite other factors such as signal strength that also directly impact power consumption [28]. Table 3.3 reports measured average radio power for an HTC TyTn II and a Google Nexus One for each RRC state and during each state promotion. The measurement methodology is described in §3.1.3. It is important to note that the 3G radio interface plays a vital role in determining the overall handset power consumption, as at DCH, the radio power contributes 1/3 to 1/2 of the overall device power during the normal workload, as measured in §3.2.2.1.

3.3.1 Radio Power Model of 4G LTE Networks

Although this dissertation focuses on 3G UMTS networks, in our work [11], we also investigated the 4G LTE networks by proposing an accurate LTE radio power model. It has two major differences from the aforementioned UMTS power model. *(i)* It considers the DRX mode in both the **RRC_CONNECTED** and the **RRC_IDLE** state. *(ii)* Since the data rate of LTE is much higher than that of 3G, it is not safe to assume the constant power level at the **RRC_CONNECTED** state regardless of the network throughput. So our LTE power model

further takes into account the impact of data rate on radio power consumption. Note that the UMTS power model used in our analyses from Chapter IV to Chapter VII can be easily replaced by the LTE power model so our proposed analysis methodologies can be applied to 4G traffic as well.

3.4 Quantifying Resource Consumption

Recall that in §2.4, we introduced three important metrics for quantifying resource consumption and revealed their incurred key tradeoffs. We revisit the three metrics below.

- ***E***: the handset radio energy consumption. It is the energy consumed by the cellular radio interface whose radio power contributes 1/3 to 1/2 of the overall handset power during the normal workload [6].
- ***S***: the signaling overhead.
- ***D***: the radio resource consumption.

By leveraging the trace-driven RRC state inference technique (§3.2) and the radio power model (§3.3), we can compute the three metrics from a packet trace collected from a handset or from the cellular core network, by following the steps below.

1. Perform trace-driven state inference (§3.2). The input is a packet trace $P_1, ..., P_n$. The output is $S(t)$ denoting the RRC state or state transition at any given time t. $S(t)$ corresponds to one of the following: IDLE, FACH, DCH, IDLE→FACH, and FACH→DCH.
2. E is calculated by associating each RRC state or state promotion an average power, using the power model described in §3.3. Given the state inference results $S(t)$ and the measured radio power values $P(\cdot)$ shown in Table 3.3, the radio energy consumed between t_1 and t_2 is computed as $\int_{t_1}^{t_2} P(S(t))dt$.

3. S is quantified by the total state promotion delay (including IDLE→DCH, IDLE→FACH, and FACH→DCH).

4. D is measured by the total DCH (the high-speed dedicated channel) occupation time (including the tail time).

3.5 Summary

We have described four important analysis techniques for cellular networks: the RRC state machine inference, the trace-driven RRC state inference, the radio power model, and the methodology for quantifying cellular resource consumption. They will be used as "building blocks" in the rest of the dissertation:

- We use the technique described in §3.1 to infer the RRC state transition models for Carrier 1 and 2 introduced in §2.1.3. The inferred RRC state machines and their parameters are shown in Figure 2.2 and Figure 2.3, respectively. We will refer to these two carriers in the rest of the dissertation.

- We use the methodology introduced in §3.4 to quantify the cellular resource consumption in Chapter IV to Chapter VII in various contexts. Note that the resource quantification technique leverages the trace-driven RRC state inference technique (§3.2) and the cellular radio power model (§3.3).

CHAPTER IV

Characterizing Radio Resource Utilization for Cellular Networks

4.1 Introduction

As described in §2.1, the current design of the RRC state machine appears to be ad-hoc with statically configured parameters. We now use real cellular traces from a large cellular ISP to systematically investigate its effect on important factors from both the network operator's perspectives, namely radio resource usage efficiency and management overhead, and from end-user's perspective, namely device energy consumption and application performance. We quantitatively analyze the tradeoffs incurred by the RRC state machine(§2.4) among these factors. In particular, we focus on settings of critical inactivity timer values that determine when to release radio resources after a period of inactivity.

As an example of the challenge in balancing the tradeoffs, using real UMTS cellular traces we observe that decreasing one inactivity timer by three seconds can reduce the overall radio resource usage by 40%, but increasing the number of state promotions by 31%. On the other hand, increasing the inactivity timer effectively enhances end user experience and reduces the management overhead, however at the expense of low efficiency in radio resource utilization and energy consumption. Intuitively, the optimal timer settings heavily depend on application traffic patterns and thus can benefit from *traffic awareness*

via trace-driven tuning.

In this chapter, we undertake a detailed exploration of the RRC state machine and its optimizations using real traces from a large cellular ISP. In particular, we make the following two contributions.

1. Characterization of state machine behaviors. The current RRC state machine parameters are either empirically configured in an ad-hoc manner [29], or determined using analytical traffic models [19, 20]. The latter approach, however, suffers from several limitations. *(i)* The expressiveness of an analytical model is quite limited and is unlikely to capture, using a statistical distribution with a few parameters, the characteristics of real-world traffic patterns generated by millions of cellular users. *(ii)* The existence of concurrent applications accessing the network further increases the difficulty of modeling the packet dynamics.

We systematically characterize the impact of existing operational state machines by analyzing traces collected from a commercial UMTS network. We found that short data transfers severely suffer from the state promotion delay, and significant portions (up to 45.3%) of the occupation time of the high-speed dedicated transmission channel is wasted on the tail time (§2.1.3). We explore the optimal timer values by replaying the trace against different state machine settings with varying parameters. Our findings suggest that the fundamental limitation of the current state machine design is its *static* nature of treating all packets according to the same inactivity timer, making it difficult to balance the aforementioned tradeoffs. We also observe that applications exhibit different sensitivities to the change of inactivity timers due to their different traffic patterns. To the best of our knowledge, our work is the first empirical study that employs real cellular traces to investigate the optimality of RRC state machine configurations.

2. Analysis of multimedia streaming strategies. We study the streaming approaches employed by Pandora audio [30] and YouTube video streaming, the two most popular smartphone multimedia streaming applications contributing large traffic volume. Our anal-

ysis demonstrates that traffic patterns impose significant impact on the radio resource and energy consumption, again due to the interplay between the mobile application and the RRC state machine. The current Pandora approach incurs long tail periods, which waste 50% of the dedicated channel time and 59% of the radio energy, while the YouTube strategy suffers from long dedicated channel occupation time due to bandwidth under-utilization. We propose a simple improvement that saves the YouTube streaming energy by 80% by leveraging the existing fast dormancy feature supported by 3GPP specifications (§2.5).

The remainder of the chapter is organized as follows. Using the cellular measurement data described in §4.2, we quantify the resource impact of the RRC state machine in §4.3 and investigate how to optimize the inactivity timer values in §4.4. In §4.5, we briefly describe two approaches that can potentially improve the current inactivity timer scheme, before concluding the chapter in §4.6. The two approaches described in §4.5 will be explored in more depth in Chapter V and Chapter VI, respectively.

4.2 The Measurement Data

This section discusses the traffic data used in our study (§4.3 and §4.4). We first describe the raw dataset in §4.2.1, then in §4.2.2, we describe the extraction procedure for application-specific data for further analysis.

4.2.1 The Raw Dataset

Our dataset is a large TCP header packet trace collected from Carrier 1 (introduced in §2.1.3) on January 29, 2010 in the normal course of operations. The collection point is one GGSN that primarily serves 3G UMTS users but also 2G GPRS users. Non-3G traffic are filtered by excluding SGSNs that exclusively serve 2G users.

Sampling was performed on a per GTP (GPRS Tunneling Protocol) session basis with a sampling rate of 1:16, so that all packets in both directions from a sampled GTP session were captured. Our trace contains 278 million TCP packets (162 GB data) of 3G traffic

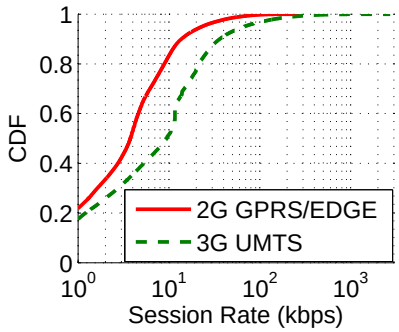


Figure 4.1: Session rate distribution over sessions longer than 100ms for 2G and 3G data

continuously captured in 3 hours. Due to concerns of large traffic volume and user privacy issues, we only recorded TCP/IP headers and a 64-bit timestamp for each packet, but no subscriber IDs or phone numbers.

Subsequently, we extract *sessions* (they are different from GTP sessions) from the trace, with each session consisting of all packets transferred by the same handset identified through the private client IP address present in the trace. It is known that cellular public IPs change very quickly [31]. But private IPs of Carrier 1 are much less dynamic, changing only at the interval of tens of minutes. Multiple TCP flows from concurrent applications may be mixed in the same session. We use a threshold of 60 sec of idle time to decide that a session has terminated. Changing this value to other values such as 45 or 75 sec does not qualitatively affect the analysis results. Besides, as shown in Figure 4.1, we do observe qualitative difference of session rate (total bytes divided by the duration of a session) between 3G and 2G. Finally, we extracted about 1.0 million 3G sessions from the raw trace.

We discuss limitations of our dataset. First, similar to previous measurements of wired network using passive trace [32, 33], the finite duration (3 hours) may influence distributions of state machine characteristics. Second, the dataset does not contain UDP traffic. A recent study [33] indicates that UDP accounts for less than 8% of the traffic volume for a wired VPN link. We expect the percentage of UDP traffic for cellular networks to be even smaller because currently UDP-based streaming and gaming applications on smartphones

Table 4.1: Cellular traces of five applications

Application	Sessions	Bytes	Description
Email-1	15K	1.4 GB	Email provider 1
Email-2	11K	536 MB	Email provider 2
Sync	4.9K	62 MB	Synchronization service
Stream	2.7K	395 MB	Pandora audio streaming [30]
Map	1.8K	60 MB	Interactive mapping

are not as popular as those on PCs.

4.2.2 Application-specific Data Extraction

From the preprocessed trace, we select five traffic types each corresponding to a particular application as shown in Table 4.1. For each application, we extract sessions in which at least 95% of the packets have either the source or the destination as one fixed server IP, thus eliminating coexistence of other applications as one session may contain multiple TCP flows of concurrent applications. For example, "Sync" consists of 4.9K sessions of a popular synchronization service. All its sessions access the same server that synchronizes emails, calendar events, contacts *etc.* between PCs and a handset using push-based notification mechanism. We use the five datasets for per-application analysis in §4.3.2 and §4.4.3 to study how application traffic patterns affect the tradeoff described in §2.4.

4.3 Resource Impact of the RRC State Machine

In this section, we study two main negative effects of the RRC state machine, and quantify them using our datasets provided by Carrier 1, whose RRC state machine is shown in Figure 2.2 (inferred by §3.1). As far as we know, this is the first study that uses real cellular traces to understand how the two factors of the RRC state machine, the state promotion overhead (§4.3.1) and the tail effects (§4.3.2), impact performance, energy efficiency, and radio resource utilization. These two factors pose key tradeoffs that we attempt to balance in this chapter. We investigate how multimedia streaming traffic pattern affects radio

resource utilization in §4.3.3.

Our overall analysis approach is as follows. We feed the packet trace into the trace-driven RRC state inference program (§3.2). It simulates the RRC states, based on which we compute various statistics of resource usage shown below.

4.3.1 State Promotion Overhead

As discussed in §2.1.3, the RRC state promotion may incur a long latency due to control message exchanges for resource allocation at the RNC. A large number of state promotions increase management overheads at the RNC and worsen user experience [13]. They have particularly negative performance impact on short sessions. For example, starting from the IDLE state, usually it takes less than 10 sec to transfer a 200KB file under normal signal strength conditions. In such a scenario, the constant overhead of 2 sec (the IDLE→DCH promotion delay) accounts for at least 20% of the total transfer time. In other UMTS networks with 3.4kbps SRB (Signalling Radio Bearer), such a promotion time for packet session setup may take even longer, up to 4 seconds [3]. It is also known that signaling DoS attacks that maliciously trigger frequent state transitions can potentially overload the control plane and detrimentally affect the 3G infrastructure [34].

We statistically quantify the state promotion overhead using our dataset. Given Σ, the set of sessions extracted in §4.2.1, we compute its *average promotion overhead* $R(\Sigma)$, defined as the fraction of the total promotion delay relative to the total duration of all sessions. The duration of a session is defined as the timestamp difference between the first and the last packet in that session.

$$R(\Sigma) = \frac{\sum_{s\in\Sigma}\{2.0N_{\text{Idle-DCH}}(s) + 1.5N_{\text{FACH-DCH}}(s)\}}{\sum_{s\in\Sigma}\{T(s) + 2.0N_{\text{Idle-DCH}}(s) + 1.5N_{\text{FACH-DCH}}(s)\}}$$

Here 1.5 sec and 2.0 sec are the two promotion delays described in Table 3.1, $N_{\text{Idle-DCH}}(s)$ and $N_{\text{FACH-DCH}}(s)$ denote for session s the number of IDLE→DCH and FACH→DCH promotions, respectively. And $T(s)$ is the session duration after preprocessing (excluding

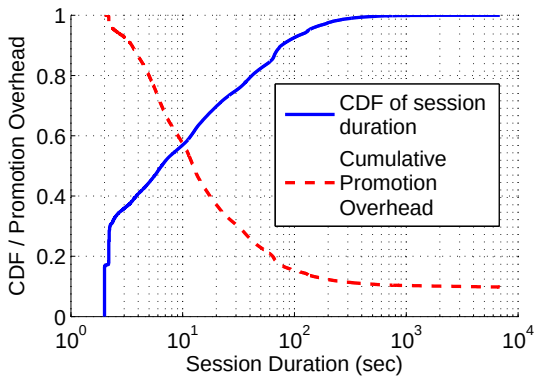


Figure 4.2: Cumulative promotion overhead

promotion delays). Then in Figure 4.2, we plot the CDF of total session duration (including promotion delays) and the cumulative promotion overhead function $CP(x)$, defined as $R(\Sigma')$ where Σ' contains all sessions whose session durations (including promotion delays) are less than x. For example, we have CDF$(10) = 0.57$ and $CP(10) = 0.57$. They indicate that within the dataset, 57% of the sessions are at most 10 sec, and their average promotion overhead $R(\Sigma')$ is 57%. Clearly, Figure 4.2 indicates that shorter sessions, which contribute to the vast majority of sessions observed, suffer more severely from the promotion delay, as $CP(x)$ is a monotonically decreasing function of x.

Our second observation is that, the RRC state promotion delay may be longer than application-specific timeout values, thus causing unnecessary timeouts leading to increased traffic volume and server overhead. For example, by default, Windows uses $t = 1, 2, 4, 8$ sec as timeout values for DNS queries [35]. Whenever a DNS query triggers an IDLE→DCH promotion that takes 2 seconds, the handset always experiences the first two timeouts. Therefore, two additional identical DNS queries are unnecessarily transmitted, and three identical responses simultaneously return to the handset, as shown in Table 4.2. This often happens in web browsing when a user clicks a link (triggering a DNS query) after an idle period. The problem can be addressed by using a large timeout value after a long idle period for UMTS 3G connection.

Table 4.2: Duplicated DNS queries and responses due to an IDLE→DCH promotion in Windows XP

Time (s)	Direction	Details of DNS query/response
0.000	Uplink	Std. query A www.eecs.umich.edu
0.989	Uplink	Std. query A www.eecs.umich.edu
1.989	Uplink	Std. query A www.eecs.umich.edu
2.111	Downlink	Std. query response A 141.212.113.110
2.112	Downlink	Std. query response A 141.212.113.110
2.121	Downlink	Std. query response A 141.212.113.110

4.3.2 The Tail Effects

One straightforward way to alleviate the state promotion overhead is to increase inactivity timer values. For example, consider two sessions that transfer data on DCH during time=0 to time=3 sec and from $t = 10s$ to $t = 14s$. We can eliminate the state promotion at $t = 10s$ by increasing α (the DCH→FACH timer, see §2.1.3) to at least 7 sec. However, this decreases the DCH utilization efficiency as the handset occupies the dedicated channel from $t = 3s$ to $t = 10s$ without any data transmission activity. Furthermore, this worsens the negative impact of *tail effects*, which waste radio resources and handset radio energy.

We define a *tail* as the idle time period matching the inactivity timer value before a state demotion [14]. It can never be used to transfer any data. In the above example, if $\alpha < 7sec$, then there exists a DCH tail from $t = 3$ to $t = 3 + \alpha$. Otherwise the period between $t = 3$ and $t = 10$ does not belong to a tail since there is no state demotion.

During a tail time, a handset still occupies transmission channels and WCDMA codes, and its radio power consumption is kept at the corresponding level of the state. In typical UMTS networks, each handset is allocated dedicated channels whose radio resources are completely wasted during the tail time. For HSDPA [3] described in §2.1.2, although the high speed transport channel is shared by a limited number of handsets, occupying it during the tail time can potentially prevent other handsets from using the high speed channel. More importantly, tail time wastes considerable amount of handset radio energy regardless of whether the channel is shared or dedicated. Reducing tail time incurs more frequent

Table 4.3: Breakdown of RRC state occupation /transition time and the tail ratios

Occupation/Trans. Time	
P_{DCH}	44.5%
P_{FACH}	48.0%
$P_{\text{IDLE}\rightarrow\text{DCH}}$	6.8%
$P_{\text{FACH}\rightarrow\text{DCH}}$	0.7%
Tail Ratios	
$P_{\text{DCH-Tail}}$	45.3%
$P_{\text{FACH-Tail}}$	86.1%

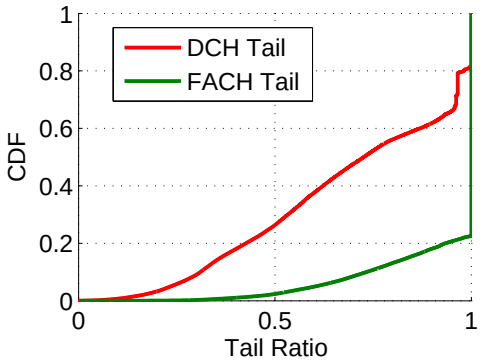


Figure 4.3: Distribution of $P_{\text{DCH-Tail}}$ and $P_{\text{FACH-Tail}}$ across all sessions

state transitions, a key tradeoff we explored in §2.4.

Overall measurement. Table 4.3 provides overall statistics for all sessions. The first four rows in Table 4.3 break down state occupation/transition time of all sessions into four categories (they add up to 100%): on DCH/FACH states, or in IDLE→DCH/ FACH→DCH promotions. In the other two rows, $P_{\text{DCH-Tail}}$ is the DCH tail ratio, defined as the total DCH tail time as a percentage of the total DCH time. We define $P_{\text{FACH-Tail}}$, the FACH tail ratio, in a similar way. Figure 4.3 plots the CDFs of $P_{\text{DCH-Tail}}$ and $P_{\text{FACH-Tail}}$ on a per-session basis. The results indicate that tail time wastes considerable radio resources and hence handset radio energy. It is surprising that the overall tail ratio for DCH and FACH are 45% and 86% respectively, and that more than 70% of the sessions have $P_{\text{DCH-Tail}}$ higher than 50%.

We also found that although the state occupation time of FACH is 48%, it transfers only 0.29% of the total traffic volume. The vast majority of the bytes (99.71%) are transferred in DCH. In other words, due to its low RLC buffer thresholds (§2.1.3), low throughput, and

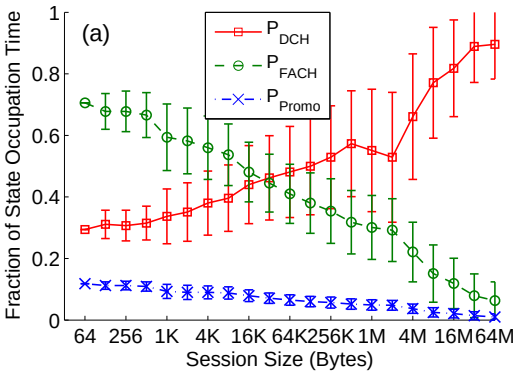


Figure 4.4: Session size vs. state occupation time

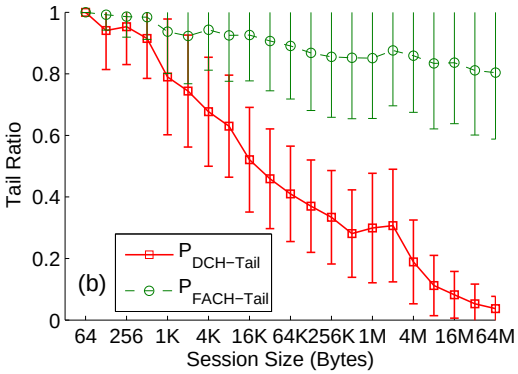


Figure 4.5: Session size vs. tail ratios

long tail, the efficiency of FACH is extremely low.

There exists strong correlations between session size (*i.e.,* the number of bytes of a session) and state machine characteristics. In particular, Figure 4.4 indicates that large sessions tend to have high fraction of DCH occupation time (P_{DCH}) as well as low P_{FACH} and $P_{\text{PROMO}} = P_{\text{IDLE}\rightarrow\text{DCH}} + P_{\text{FACH}\rightarrow\text{DCH}}$ values. Also as shown in Figure 4.5, as session size increases, both DCH and FACH tail ratios statistically decrease. In fact, small sessions are short. Their tail time periods, which are at least 5 sec and 12 sec for DCH and FACH, respectively, are comparable to or even longer than the total session duration, thus causing high $P_{\text{DCH-Tail}}$ and $P_{\text{FACH-Tail}}$ values. Another reason is that, we observe large sessions are more likely to perform continuous data transfers (*e.g.,* file downloading and multimedia streaming) that constantly occupy DCH, while small sessions tend to transfer data intermittently.

Per-application measurement. Figure 4.6 plots the per-session DCH tail distributions for the five applications shown in Table 4.1. The map application has low DCH tail ratios as its traffic patterns are non-consecutive data bursts interleaved with short pauses that are usually shorter than the α timer, so a handset always occupies DCH. On the other hand, the Sync application has inefficient DCH utilizations indicated by high $P_{\text{DCH-Tail}}$ values, since most of its sessions are very short. Usually a Sync session consists of a single data burst

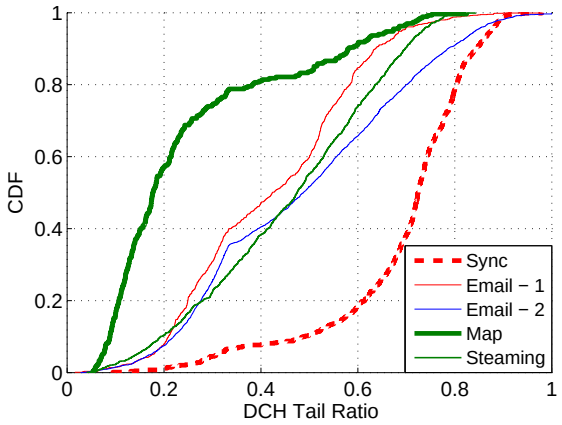


Figure 4.6: CDF of DCH tail ratios for different apps

incurring a 5-sec DCH tail and a 12-sec FACH tail.

4.3.3 Streaming Traffic Pattern and Tails

To investigate streaming traffic tails, we use an Android G2 phone of Carrier 2 to collect tcpdump traces for Pandora audio streaming [30], which is the "Streaming" application in Table 4.1, and YouTube video streaming[1]. They are the two most popular smartphone multimedia streaming applications. Both of them use TCP.

Pandora [30] is a music recommendation and Internet radio service. A user enter her favorite song or artist (called a "radio station"), then Pandora automatically streams similar music. We collected a 30-min Pandora trace by logging onto one author's Pandora account, selecting a pre-defined station, and then listening to seven tracks (songs). The traffic pattern is shown in Figure 4.7. Before a track is over, the content of the next track is buffered in one burst utilizing the maximal bandwidth (indicated by "Buffer Track x" in Figure 4.7). Then at the exact moment of switching to the next track, a small traffic burst is generated (indicated by "Play Track x" in Figure 4.7). This explains why Pandora has high tail ratios as each short burst incurs a tail. Based on our measurement, in the example shown in Figure 4.7, 50.1% of the DCH time and 59.2% of the radio energy are wasted on tails.

[1]The two applications' streaming behaviors may be different on other platforms (*e.g.,* iPhone).

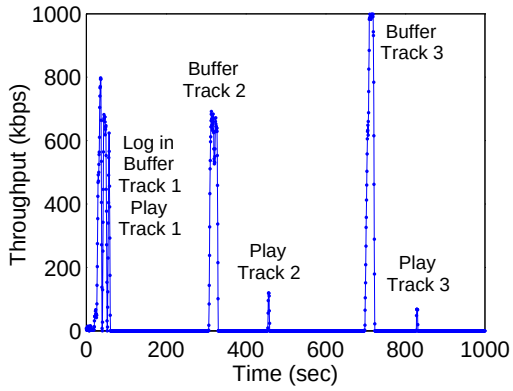


Figure 4.7: Pandora streaming (first 1k sec)

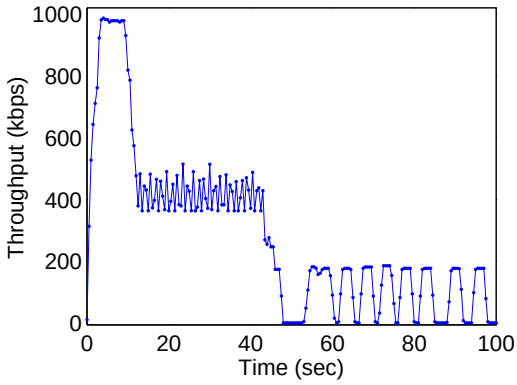


Figure 4.8: YouTube streaming (first 100 sec)

YouTube employs a different streaming procedure consisting of three phases shown in Figure 4.8. *(i)* For the first 10 sec, the maximal bandwidth is utilized for data transmission. *(ii)* The throughput is kept at around 400kbps in the next 30 sec. *(iii)* The remaining content is transmitted intermittently with the inter-burst time between 3 to 5 sec and the throughput for each burst of 200 kbps. Shorter video clips may only experience the first one or two phases. We found that YouTube traffic incurs almost no tail (except for one tail in the end) since nearly all packet inter-arrival times (IATs) are less than the α timer value. However, its drawback is under-utilization of network bandwidth causing its long DCH occupation time. Further, sending data slowly may significantly waste handset energy since on DCH a handset's radio power is relatively stable ($\pm$ 50mW as measured by a power meter) regardless of the bit rate when the signal strength is stable.

In summary, our analysis of Figure 4.7 and Figure 4.8 again implies that application traffic patterns can cause significant impact on radio resource and energy consumptions. The Pandora's approach incurs long tail periods while the YouTube streaming strategy suffers from long DCH occupation time and low energy efficiency due to bandwidth under-utilization. We propose a more energy-efficient approach for YouTube streaming in §4.5.

4.4 Tuning Inactivity Timers

Given the earlier observation that the inactivity timer values determine the key tradeoff (§2.4), we discuss how to optimize inactivity timers by trace-driven tuning. We describe our methodology and evaluation metrics in §4.4.1, followed by the results in §4.4.2 and for different applications in §4.4.3. We mainly focus on Carrier 1, with Carrier 2's findings briefly covered in §4.4.4.

4.4.1 Methodology and Evaluation Metrics

Given the moderate size of the search space for both inactivity timers, we exhaustively enumerate all combinations of the α (DCH→FACH) timer, and the β (FACH→IDLE) timer and evaluate each combination empirically by replaying all sessions for the corresponding RRC state machine. We revisit the three metrics used to characterize the tradeoff as described in §2.4 and §3.4.

- $\Delta E(\alpha, \beta) = (E(\alpha, \beta) - E(A, B))/E(A, B)$: the relative change in handset radio energy consumption relative to that of the default timer setting.
- $\Delta S(\alpha, \beta) = (S(\alpha, \beta) - S(A, B))/S(A, B)$: the relative change in the number of state promotions.
- $\Delta D(\alpha, \beta) = (D(\alpha, \beta) - D(A, B))/D(A, B)$: the relative change in the total DCH time.

Here $E(\alpha, \beta)$ corresponds to radio energy consumption under a new state machine configuration with different α and β timer values. The definitions of $S(\alpha, \beta)$ and $D(\alpha, \beta)$ are similar. $A = 5sec$ and $B = 12sec$ correspond to the default inactivity timer values for Carrier 1 (Table 3.1). We compute E, S, and D using the methodology described in §3.4.

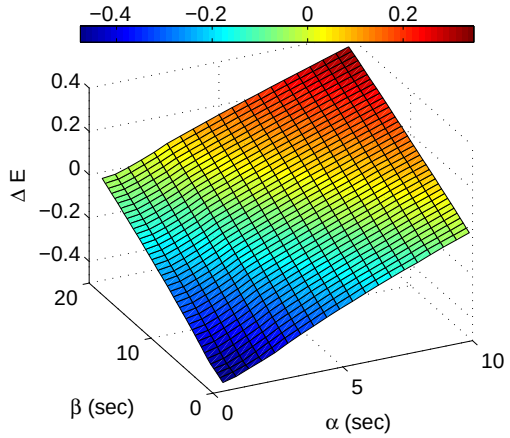


Figure 4.9: Impact of (α, β) on ΔE

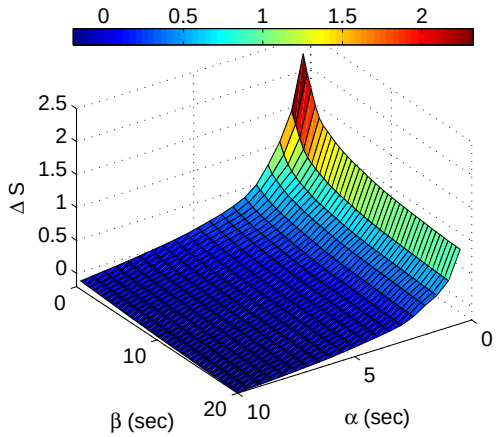


Figure 4.10: Impact of (α, β) on ΔS

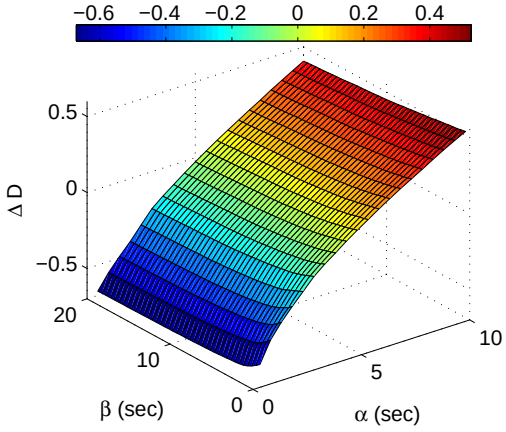


Figure 4.11: Impact of (α, β) on ΔD

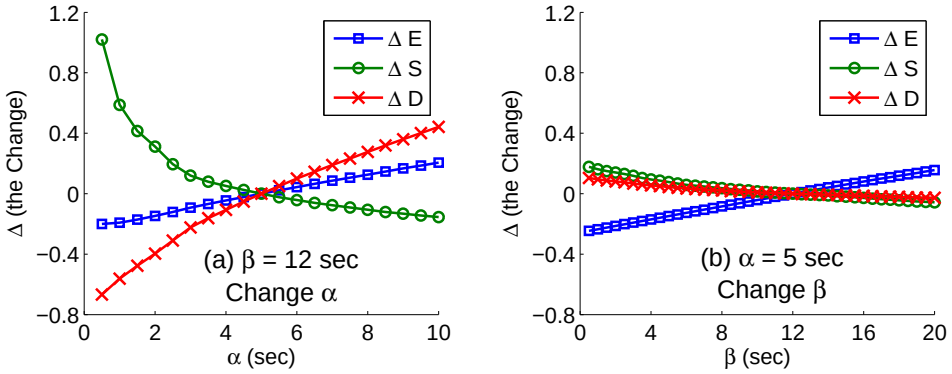


Figure 4.12: Impact of changing one timer (α or β). The other timer (β or α) is set to the default value

4.4.2 Overall Results

We visualize the distributions of ΔE, ΔS, ΔD in Figure 4.9, Figure 4.10, and Figure 4.11, respectively. Each plot contains $|\{0.5s, 1s, ..., 10s\} \times \{0.5s, 1s, ..., 20s\}| = 800$ samples of (α, β) pairs. We first fit them to quadratic functions for $\alpha, \beta \geq 1$ using multi-linear regression, with the residuals of less than 0.01, 0.08 (for $\alpha, \beta > 2sec$), and 0.03 for ΔE, ΔS, and ΔD, respectively. Clearly, the regression model is traffic-dependent.

$$\Delta E(\alpha, \beta) = -0.544 + 0.061\alpha + 0.026\beta - 0.001\alpha^2$$

$$\Delta S(\alpha, \beta) = 0.652 - 0.123\alpha - 0.023\beta + 0.005\alpha^2 + 0.001\alpha\beta$$

$$\Delta D(\alpha, \beta) = -0.618 + 0.171\alpha - 0.011\beta - 0.006\alpha^2$$

We observe that ΔE is close to a linear function of α and β. Clearly, decreasing inactivity timer values reduces tail time, thus cutting a handset's energy consumption. Usually a handset is at DCH after transferring some data. Assuming this, the handset will *first* experience the α and then the β tail. Therefore α contributes more than β does in determining ΔE. For ΔS and ΔD, they can also be approximately regarded as linear functions of α

and β when $\alpha, \beta > 2sec$, and the contribution of α is much larger than that of β as well. This is visualized in the 2D plots of Figure 4.12(a) and (b), where we fix one timer to the default value and change the other timer, then study its impact on the three metrics.

The linear fitting for ΔS does not hold when α is small ($< 2sec$) due to skewed distribution of packet inter-arrival times. Aggressively decreasing α causes excessive number of state promotions, which increase processing overheads at the RNC, worsen the application performance, and cause additional energy overhead since a promotion may consume as much as 87.5% of the power of the DCH state. We observe in Figure 4.12(a) that $|d\Delta S/d\alpha| > |d\Delta D/d\alpha|$ and $|d\Delta S/d\alpha| > |d\Delta E/d\alpha|$ for $\alpha < 5sec$, meaning that as we reduce α, the incurred state promotion overhead grows faster than the saved DCH time and energy do. This implies the fundamental limitation of current timeout scheme: it is difficult to well balance the tradeoff among ΔD, ΔE, and ΔS as timers are globally and statically set to constant values.

4.4.3 Per-application Results

Applications exhibit traffic patterns with application-specific packet dynamics, thus with different responses to changes in inactivity timer values. We next study the impact of the α timer on ΔE, ΔS, and ΔD for the five applications described in §4.2.2. We vary α and fix β at the default value of 12 sec given the relatively small impact of β.

Figures 4.13, 4.14, and 4.15 illustrate the effect of α on ΔE, ΔS, and ΔD, respectively, for the four applications (the curves for "Email-2" are very similar to those for "Email-1"). For the interactive map application, when α is small, decreasing α causes considerable increase of ΔS up to 470% (Figure 4.14). Accordingly, its ΔE increases as α decreases due to additional energy consumed by state promotions (Figure 4.13). As described in §4.3.2, the map traffic pattern consists of non-consecutive data bursts interleaved with short user-injected pauses. Therefore a small value of α comparable to or shorter than most pause durations will trigger a tremendous increase of FACH→DCH promotions. Figures 4.13

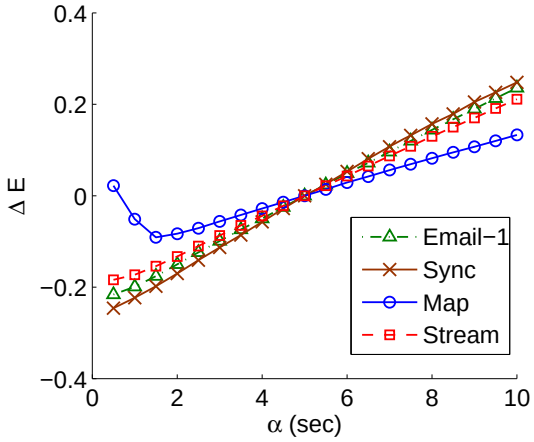


Figure 4.13: Impact of α on ΔE (β is set to the default) for four apps.

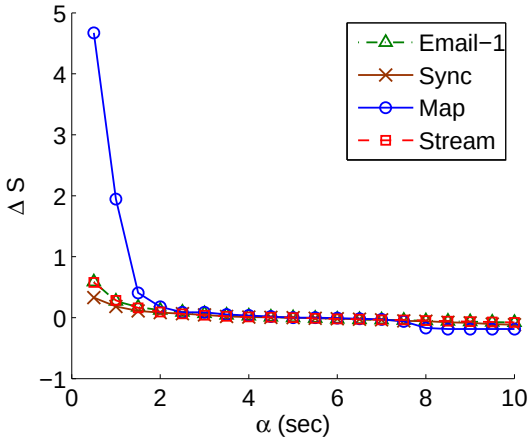


Figure 4.14: Impact of α on ΔS (β is set to the default) for four apps.

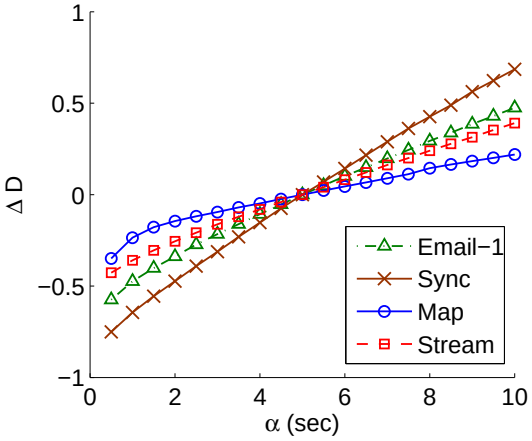


Figure 4.15: Impact of α on ΔD (β is set to the default) for four apps.

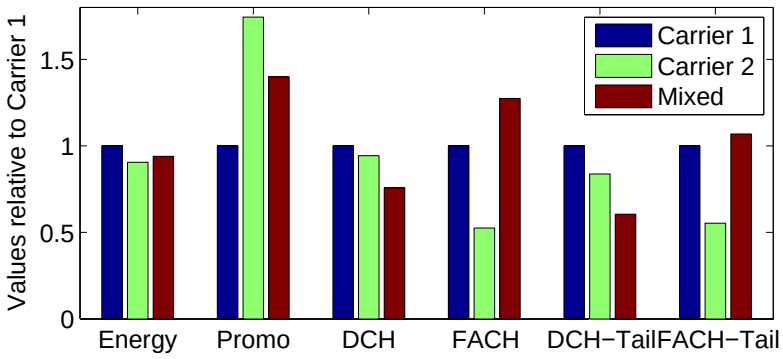


Figure 4.16: Compare state machines for two carriers

and 4.15 also indicate that short sessions with a single data burst (the Sync application) benefit more from small α values in terms of energy and radio resource savings. Our observations suggest that dynamically setting timers according to application traffic patterns may better balance the tradeoff.

4.4.4 Comparing to Carrier 2's State Machine

Recall that the major difference between the state machine for Carrier 1 (Figure 2.2) and that for Carrier 2 (Figure 2.3) is the state promotion from IDLE, from which upon any user triggered network activity, Carrier 1 directly enters into DCH while Carrier 2 may do either IDLE→FACH or IDLE→FACH→DCH, depending on the packet size. By employing the same simulation approach described in §4.4.1 for Carrier 2 we observe qualitatively similar trends in terms of the way ΔE, ΔS, and ΔD response to changes of α and β. Also α plays a more important role than β does in controlling the three metrics.

Figure 4.16 quantitatively compares both carriers. "Carrier 1" and "Carrier 2" are the real state machine settings adopted by both carriers as described in Table 3.1. "Mixed" corresponds to an artificial scheme using Carrier 2's state transition model but Carrier 1's parameters. We characterize each scheme using six metrics with Carrier 1 serving as the comparison baseline (its Y values are always 1).

The results in Figure 4.16 verify again the tradeoff discussed in §2.4. First, by compar-

Table 4.4: Optimal timer values under constraints of ΔS and different objective functions

Objective	$\min\{.75\Delta E + .25\Delta D\}$ $(\alpha, \beta, \Delta E, \Delta D)$	$\min\{.50\Delta E + .50\Delta D\}$ $(\alpha, \beta, \Delta E, \Delta D)$	$\min\{.25\Delta E + .75\Delta D\}$ $(\alpha, \beta, \Delta E, \Delta D)$
$\Delta S < -0.1$	$(8.0, 11.5, +0.12, +0.28)$	$(6.5, 18.0, +0.18, +0.12)$	$(6.5, 18.0, +0.18, +0.12)$
$\Delta S < 0.0$	$(6.0, 7.0, -0.06, +0.13)$	$(4.0, 19.0, +0.10, -0.13)$	$(4.0, 19.0, +0.10, -0.13)$
$\Delta S < 0.1$	$(4.0, 8.0, -0.13, -0.09)$	$(3.0, 14.5, -0.04, -0.23)$	$(3.0, 14.5, -0.04, -0.23)$
$\Delta S < 0.2$	$(3.5, 4.5, -0.23, -0.11)$	$(2.5, 11.5, -0.13, -0.31)$	$(2.5, 11.5, -0.13, -0.31)$
$\Delta S < 0.3$	$(3.0, 3.0, -0.30, -0.16)$	$(2.5, 6.0, -0.26, -0.28)$	$(2.0, 13.5, -0.11, -0.40)$
$\Delta S < 0.4$	$(2.5, 3.0, -0.33, -0.25)$	$(2.0, 7.5, -0.25, -0.38)$	$(1.5, 13.5, -0.14, -0.48)$
$\Delta S < 0.5$	$(2.0, 4.0, -0.34, -0.35)$	$(1.5, 8.0, -0.26, -0.46)$	$(1.5, 8.0, -0.26, -0.46)$

ing "Carrier 1" and "Mixed", we find that changing IDLE→DCH to IDLE→FACH→DCH decreases DCH and DCH tail time by 24% and 39%, respectively, but at the cost of increased number of state promotions by 40%, since for Carrier 2, when a handset at IDLE has non-trivial amount of data (greater than the RLC buffer threshold) to transfer, it always experiences two state promotions to DCH. Our second observation is derived by comparing "Carrier 2" and "Mixed". Their FACH→IDLE timers are significantly different (4 sec for "Carrier 2" and 12 sec for "Mix"), resulting in considerable disparities of their FACH (and FACH tail) times.

4.4.5 Summary

We tune the two inactivity timer values for the RRC state machine of Carrier 1. Table 4.4 summarizes our derived timer values under different constraints of ΔS. Column 2 to 4 correspond to three optimization objectives: energy-saving biased (minimize $0.75\Delta E + 0.25\Delta D$), no bias (minimize $0.5\Delta E + 0.5\Delta D$), and radio-resource-saving biased (minimize $0.25\Delta E + 0.75\Delta D$). The coefficients are empirically chosen to weight the energy and/or the radio resource consumption.

We highlight our findings in this section as follows.

- ΔE, ΔS, and ΔD are approximately linear functions of α and β when they are not very small. The α timer imposes much higher impact on the three metrics than the β timer does.

- Very small α timer values ($< 2sec$) cause significant increase of the state promotion overhead.

- Applications have different sensitivities to changes of inactivity timers due to their different traffic patterns.

- It is difficult to well balance the tradeoff among ΔD, ΔE, and ΔS since the state promotion overhead grows faster than saved DCH time and energy do when we reduce the timers. The fundamental reason is that timers are globally and statically set to constant values.

4.5 Improve The Current Inactivity Timer Scheme

We explore approaches that improve the current inactivity timer scheme whose limitations are revealed in §4.4.

4.5.1 Shaping Traffic Patterns

Handset applications alter traffic patterns based on the state machine behavior in order to reduce the tail time. We describe two such approaches.

Batching and Prefetching. In [14], the authors discuss two traffic shaping techniques: batching and prefetching. For delay-tolerant applications such as Email and RSS feeds, their transfers can be *batched* to reduce the tail time. In the scenario of web searching, a handset can avoid tails caused by a user's idle time with high probability by *prefetching* the top search results. [14] proposes an algorithm called *TailEnder* that schedules transfers to minimize the energy consumption while meeting user-specified deadlines by batching or prefetching. Their simulation indicates that TailEnder can transfer 60% more RSS feed updates and download search results for more than 50% of web queries, compared to using the default scheme. Similar schemes for delay-tolerant applications in cellular environment

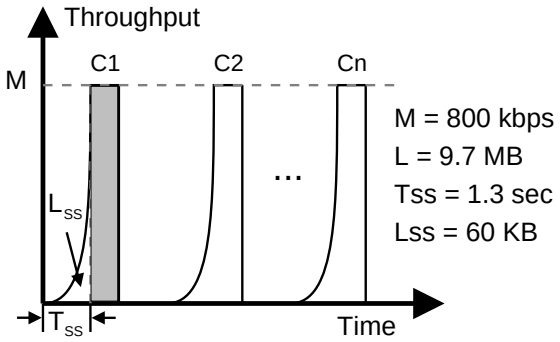


Figure 4.17: Streaming in chunk mode

were proposed for, for example, offloading 3G data transfers to WiFi [36] and scheduling communication during periods of strong signal strength [28].

Proposed Traffic shaping scheme for YouTube. Recall that in §4.3.3, we pinpoint the energy inefficiency of YouTube traffic caused by under-utilizing the available bandwidth (Figure 4.8).

To overcome such inefficiency, we reshape the traffic using a hypothetical streaming scheme called *chunk mode*, as illustrated in Figure 4.17. The video content is split into n chunks $C_1, ..., C_n$, each transmitted at the highest bit rate. We model the traffic pattern of chunk mode transfer as follows. Let $L = 9.7MB$ be the size of a 10-minute video and let $M = 800kbps$ be the maximal throughput. For each chunk, it takes T_{SS} seconds for the TCP slow start[2] to ramp up to the throughput of M. By considering delayed ACKs and letting the RTT be 250ms (as measured by the ping latency to YouTube), we compute T_{SS} at 1.3 sec during which $L_{SS} = 60KB$ of data is transferred. The total transfer time for the n chunks (excluding tails) is $T = (T_{SS} + (\frac{L}{n} - L_{SS})/M)n$, and the DCH tail and FACH tail time are $n\alpha$ and $n\beta$, respectively. The whole transfer incurs n IDLE→FACH promotions and n FACH→DCH promotions for Carrier 2's UMTS network.

Based on the above parameters, we compute ΔD and ΔE and plot them in Figure 4.18(a) and (b), respectively. Note that the energy E consists of three components: the state pro-

[2]A slow start is necessary since the interval between consecutive chunks is longer than one TCP retransmission timeout [37]. Using TCP keep alive can avoid slow starts but it consumes more energy due to the tail effect.

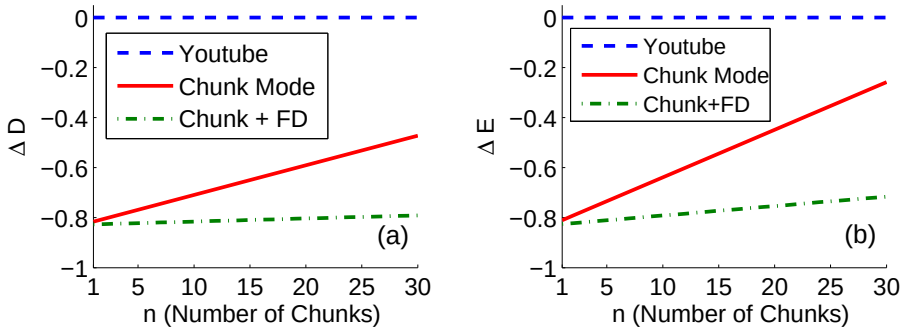


Figure 4.18: The evaluations of chunk mode streaming on (a) ΔD and (b) ΔE

motion energy, the DCH non-tail energy, and the tail energy of DCH/FACH. In each plot, the two curves "YouTube" and "Chunk Mode" correspond to streaming schemes of current YouTube and the chunk mode, respectively. We describe the "Chunk + FD" curve in §4.5.2.

As indicated by Figure 4.18, compared to current YouTube streaming strategy, the chunk mode saves DCH time and energy by up to 80%. Transferring the whole video in one chunk ($n = 1$) is the most effective. However, measurement results show that users often do not watch the entire video [38]. Therefore when n is small, it may cause unnecessary data transfers if a user only watches part of the video. This problem can be addressed by increasing n and transferring data chunks according to the playing progress of the video. However, resources saved by the chunk mode decrease as n increases due to the tail effect.

To summarize, for some applications, shaping their traffic patterns based on the prior knowledge of the RRC state machine brings significant savings of radio resources and handset energy. Motivated by such an observation, in Chapter V, we design a tool that profiles smartphone applications and identifies their inefficient resource usage due to their traffic patterns poorly interacting with the RRC state machine.

4.5.2 Dynamic Timers and Fast Dormancy

Dynamic Timer Scheme. Our per-application study in §4.3.2 and §4.4.3 suggests that *dynamically* changing timers can potentially better balance the tradeoff. Ideally, this can be achieved by the RNC that adjusts timers at a per-handset basis according to observed

traffic patterns. However, such an approach requires significant changes to the RNC, which currently does not recognize IP and its upper layers. The computational overhead is also a concern as the RNC has to identify traffic patterns and compute the appropriate timer values for all connected handsets. The third challenge is that for each handset, the traffic observed by the RNC may originate from multiple applications that concurrently access the network, thus making identifying traffic patterns even harder.

Fast Dormancy. Another potential approach for mitigating the tail effect is fast dormancy described in §2.5. Recall that in the fast dormancy scheme, a handset can proactively request that the RRC state be immediately demoted to IDLE based on its prediction of an imminent tail, thus reducing the tail time.

We use the YouTube example to demonstrate a typical scenario where fast dormancy can be applied. Recall the chunk mode streaming scheme shown in Figure 4.17. In order to eliminate tails, the YouTube application would invoke fast dormancy to demote the state to IDLE immediately after a chunk is received (assuming no concurrent network activity exists). This corresponds to the "Chunk + FD" curve in Figure 4.18(a) and (b), which indicate that by eliminating the tails, fast dormancy can keep ΔD and ΔE almost constant regardless of n, the number of chunks. Recall that a large n prevents unnecessary data transfers in common cases [38] where a user watches part of the video.

As described in §2.5, we observe that a few handsets adopt fast dormancy in an application-agnostic manner. To the best of our knowledge, however, no individual smartphone application today can invoke fast dormancy based on its traffic pattern, partly due to two reasons. First, for user-interactive applications (*e.g.,* web browsing), accurately predicting a long idle period is challenging as user behavior injects randomness to the packet timing. Second, there lacks OS support that provides a simple programming interface for invoking fast dormancy. In particular, the concurrency problem presents a challenge. It is not feasible that applications independently predict the tail and invoke fast dormancy since state transitions are determined by the aggregated traffic of all applications. The OS should schedule

concurrent applications and invoke fast dormancy only if the combined idle period predicted by all applications is long enough. In Chapter VI, we propose a novel resource management framework called TOP that bridges the gap between the application and the fast dormancy support.

4.6 Summary

In this chapter, we undertook a detailed exploration of the RRC state machine, which guides the radio resource allocation policy in 3G UMTS network, by analyzing real cellular traces and measuring from real smartphones. We found that the RRC state machine may cause considerable performance inefficiency due to the state promotion overhead, as well as cause significant radio resource and handset energy inefficiency due to the tail effects. These two factors form the key tradeoff that is difficult to balance by the current inactivity timer designs. The fundamental reason is that the timers are globally and statically set to constant values that cannot adapt to the diversity of traffic patterns generated by different applications. We believe that addressing this problem requires the knowledge of mobile applications, which can proactively alter traffic patterns based on the state machine behavior, or cooperate with the radio access network in allocating radio resources (*e.g.,* the fast dormancy approach). We will explore both approaches in Chapter V and Chapter VI, respectively.

CHAPTER V

Profiling Smartphone Apps for Identifying Inefficient Resource Usage

Our analysis described in the previous chapter has shown that the traffic patterns for many mobile applications can be improved based on the prior knowledge of the cellular resource management policy. Doing so can potentially bring significant savings of radio resources and handset energy. Motivated by such an observation, we focus on improving the efficiency of smartphone applications in this chapter.

5.1 Introduction

Increasingly ubiquitous cellular data network coverage gives an enormous impetus to the growth of diverse smartphone applications. Despite a plethora of such mobile applications developed by both the active user community and professional developers, there remain far more challenges associated with mobile applications compared to their desktop counterparts. In particular, application developers are usually unaware of cellular specific characteristics that incur potentially complex interaction with the application behavior. Even for professional developers, they often do not have visibility into the resource-constrained mobile execution environment. Such situations potentially result in smartphone applications that are not cellular-friendly, *i.e.,* their radio channel utilization or device en-

ergy consumption are inefficient because of a lack of transparency in the lower-layer protocol behavior. For example, we discovered that for Pandora, a popular music streaming application on smartphones, due to the poor interaction between the radio resource control policy and the application's data transfer scheduling mechanism, 46% of its radio energy is spent on periodic audience measurements that account for only 0.2% of received user data (§5.5.2.1).

In this chapter, we address the aforementioned challenge by developing a tool called ARO (mobile **A**pplication **R**esource **O**ptimizer). To the best of our knowledge, ARO is the first tool that exposes the *cross-layer interaction* for layers ranging from higher layers such as user input and application behavior down to the lower protocol layers such as HTTP, transport, and very importantly radio resources. In particular, so far little focus has been placed on the interaction between applications and the radio access network (RAN) in the research community. Such cross-layer information encompassing device-specific and network-specific information helps capture the tradeoffs across important dimensions such as energy efficiency, performance, and functionality, making such tradeoffs explicit rather than arbitrary as it is often the case today. By performing various analyses for RRC layer, TCP layer, HTTP layer, user interactions, followed by their cross-layer interactions, ARO therefore helps reveal inefficient resource usage (*e.g.,* high resource overhead of periodic audience measurements for Pandora) due to a lack of transparency in the lower-layer protocol behavior, leading to suggestions for improvement.

ARO consists of an online lightweight data collector and an offline analysis module. To profile an application, an ARO user simply starts the data collector, which incurs less than 15% of runtime overhead, and then runs the application for a desired duration as a normal application user. The collector captures packet traces, system and user input events, which are subsequently processed by the analysis module on a commodity PC. The proposed ARO framework (§5.2) also applies to other types of cellular networks such as GPRS/EDGE (§3.1.2.1) [27], EvDO [10], and 4G LTE (§2.2) [11] that involve similar tradeoffs to those

in UMTS. We highlight our contributions as follows.

1. **Root cause analysis for short traffic bursts (§5.3.2).** Low efficiency of radio resource and energy usage are fundamentally attributed to *short traffic bursts* carrying small amount of user data while having long idle periods, during which a device keeps the radio channel occupied, injected before and after the bursts [14, 5]. We develop a novel algorithm to identify them and to distinguish which factor triggers each such burst, *e.g.,* user input, TCP loss, or application delay, by synthesizing analysis results of the TCP, HTTP, and user input layer. ARO also employs a robust algorithm (§5.3.2.1) to identify periodic data transfers that in many cases incur high resource overhead. Discovering such triggering factors is crucial for understanding the root cause of inefficient resource utilization. Previous work [39, 5] also investigate the impact of traffic patterns on radio power management policy and propose suggestions. In contrast, ARO is essential in providing more specific diagnosis by breaking down resource consumption into each burst with its triggering factor accurately inferred. For example, for the Fox News application (§5.5.2.2), by correlating application-layer behaviors (*e.g.,* transferring image thumbnails), user input (*e.g.,* scrolling the screen), and RRC states, ARO reveals it is user's scrolling behavior that triggers scattered traffic (*i.e.,* short bursts) for downloading image thumbnails in news headlines (*i.e.,* images are transferred only when they are displayed as a user scrolls down the screen), and quantifies its resource impact. Analyzing data collected at one single layer does not provide such insight due to incomplete information (Table 5.3).

2. **Quantifying resource impact of traffic bursts (§5.3.3).** In order to *quantitatively* analyze resource bottlenecks, ARO addresses a new problem of quantifying resource consumption of traffic bursts due to a certain triggering factor. It is achieved by computing the difference between the resource consumption in two scenarios where bursts of interest are kept and removed, respectively. The latter scenario requires changing the traffic pattern. To address such a challenge of modifying a cellular

packet trace while having its RRC states updated accordingly, ARO strategically decouples the RRC state machine impact from application traffic patterns, modifies the trace, and then faithfully reconstructs the RRC states.

3. **Identification of resource inefficiencies of real Android applications (§5.5).** We apply ARO to six real Android applications each with at least 250,000 downloads from the Android market as of Dec 2010. ARO reveals that many of these very popular applications (Fox News, Pandora, Mobclix ad platform, BBC News *etc.*) have significant resource utilization inefficiencies that are previously unknown. We provide suggestions on improving them. In particular, we are starting to contact developers of popular applications such as Pandora. The feedback has been encouragingly positive as the provided technique greatly helps developers identify resource usage inefficiencies and improve their applications [40].

The rest of the chapter is organized as follows. We outline the ARO system in §5.2. In §5.3, we detail the analyses at higher layers (TCP, HTTP, burst analysis) as well as the cross-layer synthesis. We briefly describe how we implement the ARO prototype in §5.4, then present case studies of six Android applications in §5.5 to demonstrate typical usage of ARO before concluding the chapter in §5.6.

5.2 ARO Overview

This section outlines the ARO system, which consists of two main components: the data collector and the analyzers. The data collector runs efficiently on a handset to capture information essential for understanding resource usage, user activity, and application performance. Our current implementation collects network packet traces and user input events. But other information such as application activities (*e.g.,* API calls) and system information (*e.g.,* CPU usage) can also be collected for more fine-grained analysis. The collected traces are subsequently fed into the analyzers, which run on a PC, for offline

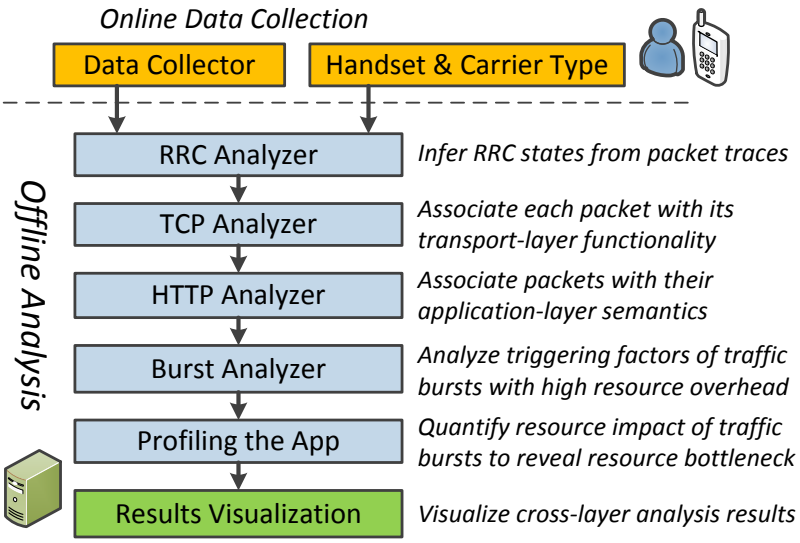


Figure 5.1: The ARO System

analysis. Our design focuses on modularity to enable independent analysis of individual layers whose results can be subsequently correlated for joint cross-layer analysis. The proposed framework is easily extensible to other analyzers of new application protocols. We describe the workflow of ARO as outlined in Figure 5.1.

1. The ARO user invokes on her handset the data collector, which subsequently collects relevant data, *i.e.,* all packets in both directions and user input (*e.g.,* tapping or scrolling the screen). Unlike other smartphone data collection efforts [41, 42], our ability to collect user interaction events and packet-level traces enables us to perform fine-grained correlation across layers. ARO also identifies the packet-to-application correspondence. This information is used to distinguish the *target application*, *i.e.,* the application to be profiled, from other applications simultaneously accessing the network. Note that ARO collects all packets since RRC state transitions are determined by the aggregated traffic of all applications running on a handset.

2. The ARO user launches the target application and uses the application as an end user. Factors such as user behavior randomness and radio link quality influence the collected data and thus the analysis results. Therefore, to obtain a representative

understanding of the application studied, ARO can be used across multiple runs or by multiple users to obtain a comprehensive exploration of different usage scenarios of the target application, as exemplified in our case studies (§5.5.1). The target application might also be explored in several usage scenarios, covering diverse functionalities, as well as execution modes (*e.g.,* foreground and background).

3. The ARO user loads the ARO analysis component with the collected traces. ARO then configures the RRC analyzer with handset and carrier specific parameters, which influence the model used for RRC analysis (§3.2). The TCP, HTTP, and burst analyzers are generally applicable.

4. ARO then performs a series of analyses across several layers. In particular, the RRC state machine analysis (§3.2) accurately infers the RRC states from packet traces so that ARO has a complete view of radio resource and radio energy utilization during the entire data collection period. ARO also performs transport protocol and application protocol analysis (§5.3.1) to associate each packet with its transport-layer functionality (*e.g.,* TCP retransmission) and its application-layer semantics (*e.g.,* an HTTP request). Our main focus is on TCP and HTTP, as the vast majority of smartphone applications use HTTP over TCP to transfer application-layer data [43, 44]. ARO next performs burst analysis (§5.3.2), which utilizes aforementioned cross-layer analysis results, to understand the triggering factor of each short traffic burst, which is the key reason of low efficiency of resource utilization [5].

5. ARO profiles the application by computing for each burst (with its inferred triggering factor) its radio resource and radio energy consumption (§5.3.3) in order to identify and quantify the resource bottleneck for the application of interest. Finally, ARO summarizes and visualizes the results. Visualizing cross-layer correlation results helps understand the time series of bursts that are triggered due to different reasons, as later demonstrated in our case studies (§5.5).

Table 5.1: TCP analysis: transport-layer properties of packets

Category	Label	Description
TCP connection management	ESTABLISH	A packet containing the SYN flag
	CLOSE	A packet containing the FIN flag
	RESET	A packet containing the RST flag
Normal data transfer	DATA	A normal data packet with payload
	ACK	A normal ACK packet without payload
TCP congestion, loss, and recovery	DATA_DUP	A duplicate data packet
	DATA_RECOVER	A data pkt echoing a duplicate ACK
	ACK_DUP	A duplicate ACK packet
	ACK_RECOVER	An ACK echoing a duplicate data pkt
Others	TCP_OTHER	Other special TCP packets

5.3 Profiling Mobile Applications

This section details analyses at higher layers, in particular the transport layer and the application layer, using TCP and HTTP as examples due to their popularity. We further describe how ARO uses cross-layer analysis results to profile resource efficiency of smartphone applications. Note that the RRC state machine analysis performed by ARO has been described in §3.2.

5.3.1 TCP and HTTP Analysis

TCP and HTTP analysis serve as prerequisites for understanding traffic patterns created by the transport layer and the application layer. Our main focus is on TCP and HTTP, as the vast majority of smartphone applications use HTTP over TCP to transfer application-layer data [43]. A recent large-scale measurement study [44] using datasets from two separate campus wireless networks (3 days of traffic for 32,278 unique devices) indicates that 97% of handheld traffic is HTTP.

We first describe the TCP analysis. ARO extracts TCP flows, defined by tuples of {srcIP, srcPort, dstIP, dstPort} from the raw packet trace, and then infers the transport-layer property for each packet in each TCP flow. In particular, each TCP packet is assigned to one of the labels listed in Table 5.1. The labels can be classified into four categories

covering the TCP protocol behavior: *(i)* connection management, *(ii)* normal data transfer, *(iii)* TCP congestion, loss, and recovery, and *(iv)* other special packets (*e.g.,* TCP keep alive and zero-window notification).

In the third category, `DATA_DUP` is usually caused by a retransmission timeout or fast retransmission, and `ACK_DUP` is triggered by an out-of-order or duplicate data packet. Duplicate packets indicate packet loss, congestion, or packet reordering that may degrade TCP performance. A `DATA_RECOVER` packet has its sequence number matching the ack number of previous duplicate ACK packets in the reverse direction, indicating the attempt of a handset to transmit a possibly lost uplink packet or a downlink lost packet finally arriving from the server. Similarly, the ack number of an `ACK_RECOVER` packet equals to the sequence number of some duplicate data packets plus one, indicating the recipient of a possibly lost data packet.

ARO subsequently performs HTTP analysis by reassembling TCP flows then following the HTTP protocol to parse the TCP flow data. HTTP analysis provides ARO with the precise knowledge of mappings between packets and HTTP requests or responses.

5.3.2 Burst Analysis

As described earlier, low efficiencies of radio resource and energy utilization are attributed to short traffic bursts carrying small amount of data. ARO employs novel algorithms to identify them and to infer which factor triggers each such burst by synthesizing analysis results of the RRC, TCP, HTTP, and user input layer. Such triggering factors, which to our knowledge are not explored by previous effort, are crucial for understanding the root cause of inefficient resource utilization.

ARO defines a *burst* as consecutive packets whose inter-arrival time is less than a threshold δ. We set δ to 1.5 seconds since it is longer than commonly observed cellular round trip times [45]. Since state promotion delays are usually greater than δ, all state promotions detected in the trace-driven RRC state inference (§3.2) are removed before bursts

Table 5.2: Burst Analysis: triggering factors of bursts

Label	The burst is triggered by ...
USER_INPUT	User interaction
LARGE_BURST	(The large burst is resource efficient)
TCP_CONTROL	TCP control packets (*e.g.,* FIN and RST)
SVR_NET_DELAY	Server or network delay
TCP_LOSS_RECOVER	TCP congestion / loss control
NON_TARGET	Other applications not to be profiled
APP	The application itself
APP_PERIOD	Periodic data transfers (One special type of APP)

are identified. Each bar in the "Bursts" band in Figure 3.11 is a burst.

A burst can be triggered by various factors. Understanding them benefits application developers who can then customize optimization strategies for each factor, *e.g.,* to eliminate a burst, to batch multiple bursts, or to make certain bursts appear less frequently. Some bursts are found to be inherent to the application behavior. We next describe ARO's burst analysis algorithm that assigns to each burst a triggering factor shown in Table 5.2 by correlating TCP analysis results and user input events.

The algorithm listed in Figure 5.2 consists of seven tests each identifying a triggering factor by examining burst size (duration), user input events, payload size, packet direction, and TCP properties (§5.3.1) associated with a burst. We explain each test as follows. A burst can be generated by a non-target application not profiled by ARO (Test 1). For Test 2, if a burst is large and long enough (determined by two thresholds th_s and th_d), it is assigned a LARGE_BURST label so ARO considers it as a resource-efficient burst. If a burst only contains TCP control packets without user payload (Lines 06 to 08), then it is a TCP_CONTROL burst as determined by Test 3. To reveal delays caused by server, network, congestion or loss, the algorithm then considers properties of the first packet in the burst in Test 4 and 5. For Test 6, if any user input activity is captured within a time window of ω seconds before a burst starts, then the burst is assigned a USER_INPUT label, if it contains user payload. For bursts whose triggering factors are not identified by the above tests, they are considered to be issued by the application itself (APP in Test 7). Most such bursts turn

```
01 Burst_Analysis (Burst b) {
02   Remove packets of non-target apps;
03   if (no packet left) {return NON_TARGET;} Test 1
04   if (b.payload > th_s && b.duration > th_d) Test 2
05     {return LARGE_BURST;}
06   if (b.payload == 0) { Test 3
07     if (b contains any of ESTABLISH, CLOSE, RESET,
08     TCP_OTHER packets)
09       {return TCP_CONTROL;}
10   }
11   d_0 ← direction of the first packet of b;
12   i_0 ← TCP label of the first packet of b;
13   if (d_0 == DL && (i_0 == DATA || i_0 == ACK)) Test 4
14     {return SVR_NET_DELAY;}
15   if (i_0 == ACK_DUP || i_0 == ACK_RECOVER ||
16   i_0 == DATA_DUP || i_0 == DATA_RECOVER) Test 5
17     {return TCP_LOSS_RECOVER;}
18   if (b.payload > 0 && find user input before b) Test 6
19     {return USER_INPUT;}
20   if (b.payload > 0) {return APP;} Test 7
21   else {return UNKNOWN;}
22 }
```

Figure 5.2: The burst analysis algorithm

out (and are validated) to be periodic transfers (APP_PERIOD) triggered by an application using a software timer. We devise a separate algorithm to detect them (§5.3.2.1). In practice it is rare that a short burst satisfies multiple tests.

The burst analysis algorithm involves three parameters: th_s and th_d that quantitatively determine a large burst (Test 2), and the time window ω (Test 6). We set th_s = 100 KB, th_d = 5 sec, and ω = 1 sec. We empirically found that varying their values by $\pm 25\%$ (and $\pm 50\%$ for ω) does not qualitatively affect the analysis results presented in §5.5.

Within aforementioned seven tests, Test 1 to 3 are trivial. We validate Test 4 and 5 by setting up a web server and intentionally injecting server delay and packet losses. Evaluation for Test 6 and Test 7, which is more challenging due to a lack of ground truth, is done by manually inspecting our collected traces used for case studies (§5.5).

5.3.2.1 Identifying Periodic Transfers

We design a separate algorithm to spot APP_PERIOD bursts (Table 5.2), which are data transfers periodically issued by a handset application using a software timer. Such transfers are important because their impact on resource utilization can be significant although they may carry very little actual user data (*e.g.,* the Pandora application described in §5.5.2.1).

ARO focuses on detecting three types of commonly observed periodic transfers, though not mutually exclusive. They constitute the most simple forms of periodic transfers a mobile application can do using HTTP: *(i)* periodically fetching the same HTTP object, *(ii)* periodically connecting to the same IP address, and *(iii)* periodically fetching an HTTP object from the same host. Detecting other periodic activities can be trivially added to the proposed detection framework shown in Figure 5.3. Also we found that existing approaches for periodicity or clock detection (*e.g.,* DFT-based [33] and autocorrelation-based [46]) do not work well in our scenario where the number of samples is much fewer.

The algorithm, shown in Figure 5.3, takes as input a time series $t_1, ..., t_n$, and outputs the detected periodicity (*i.e.,* the cycle duration) if it exists. It enumerates all $n(n-1)/2$

```
01 Detect_Periodic_Transfers (t_1, t_2, ..., t_n) {
02   C ← {(d, t_i, t_j) | d = t_j − t_i  ∀j > i};
03   Find the longest sequence
04     D = (d_1, x_1, y_1), ..., (d_m, x_m, y_m) in C s.t.
05     (1) y_1 = x_2, y_2 = x_3, ..., y_{m−1} = x_m, and
06     (2) max(d_i) − min(d_i) < p;
07   if m ≥ q return mean(d_1, ..., d_m);
08   else return "no periodic transfer found";
09 }
```

Figure 5.3: Algorithm for detecting periodic transfers

possible intervals between t_i and t_j where $1 \leq i < j \leq n$ (Line 2), from which the longest sequence of intervals is computed by dynamic programming (Lines 3-6). Such intervals should be consecutive (Line 5) and have similar values whose differences are bounded by parameter p (Line 6). If the sequence length is long enough, larger than the threshold parameter q, then the average interval is reported as the cycle duration (Line 7). We empirically set p=1 sec and q=3 based on evaluating the algorithm on *(i)* randomly generated test data (periodic time series mixed with noise), and *(ii)* real traces studied in §5.5.

5.3.3 Profiling Applications

We describe how ARO profiles mobile applications using cross-layer analysis. First, leveraging RRC state inference and burst analysis results, ARO computes for each burst (with its triggering factor known) its radio resource and radio energy consumption. Then the TCP and HTTP analysis described in §5.3.1 allow ARO to associate each burst with the transport-layer or the application-layer behavior so that an ARO user can learn quantitatively what causes the resource bottleneck for the application of interest.

We describe two methodologies for quantifying the resource consumption of one or more bursts of interest: computing the *upperbound* and the *lowerbound*. Their key difference is whether or not they consider non-interested bursts whose tails help reduce the resource consumption of those interested bursts.

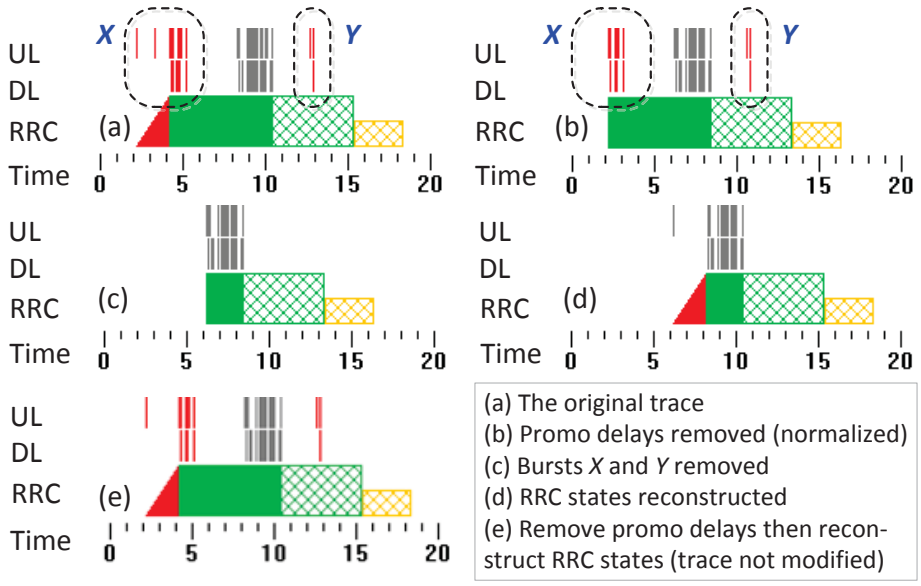


Figure 5.4: An example of modifying cellular traces (X and Y are the bursts of interest to be removed)

Method 1: Compute the upperbound of resource consumption. The radio energy consumed by burst B_i is computed as $\int_{t_1}^{t_2} P(S(t))dt$ where $S(t)$ is the inferred RRC state at time t and $P(\cdot)$ is the power function (Table 3.3). t_1 is the time when Burst B_i starts consuming radio resources. Usually t_1 equals to the timestamp of the first packet of B_i. However, if B_i begins with a downlink packet triggering a state promotion, t_1 should be shifted backward by the promotion delay since radio resources are allocated during the state promotion before the first packet arrives (§3.2.1). t_2 is the timestamp of the first packet of the next burst B_{i+1}, as tail times incurred by B_i need to be considered (there may exist IDLE periods before t_2, but they do not consume any resource). Similarly, t_2 is shifted backward by the promotion delay if necessary. The radio resources consumed by B_i are quantified as the DCH occupation time between t_1 and t_2. We ignore radio resources allocated for shared low-speed FACH channels.

Method 2: Compute the lowerbound. One problem with Method 1 is that it may *overestimate* a burst's resource consumption, which may already be covered by the tail of a previous burst. For example, consider burst Y in Figure 5.4(a). Its resource utilization lasts

from t=12.5 sec to t=18.3 sec according to Method 1. However, such an interval is already covered by the tail of the previous burst. In other words, the overall resource consumption is not reduced even if in the absence of burst Y.

To address this issue, we propose another way to quantify the resource impact of one or more bursts by computing the *difference* between the resource consumption of two scenarios where the bursts of interest are kept and removed, respectively. For example, in Figure 5.4, let X and Y be the bursts of interest. Trace (a) and (d) correspond to the original trace and a modified trace where X and Y are removed. Then their energy impact is computed as $E_a - E_d$ where E_a and E_d correspond to the radio energy consumption of trace (a) and (d), respectively. The consumed resource computed by this method does not exceed that computed by Method 1.

5.3.3.1 Modifying Cellular Traces

The aforementioned Method 2 is intuitive, while the challenge here is to construct a trace with some packets removed. In particular, RRC state promotion delays affect the packet timing. Therefore, removing packets directly from the original trace causes inaccuracies as it is difficult to transform the original promotion delays to promotion delays in the modified trace with different state transitions. To address such a challenge, we propose a novel technique for modifying cellular traces. The high-level idea is to first *decouple* state promotion delays from application traffic patterns before modifying the trace, then reconstruct the RRC states for the modified trace.

The whole procedure is illustrated in Figure 5.4a-d (assuming we want to remove bursts X are Y). First, the original trace (Figure 5.4a) is *normalized* by removing all promotion delays (Figure 5.4b). This essentially decouples the impact of state promotions from the real application traffic patterns [5]. Then the bursts of interest are removed from the normalized trace (Figure 5.4c). Next, ARO runs the state inference algorithm again to reconstruct the RRC states with state promotions injected using the average promotion delay values

shown in Table 3.1 (Figure 5.4d). As expected, the first packet in Figure 5.4d triggers a promotion that does not exist in the original trace (a).

Validation. We demonstrate the validity of the proposed cellular trace modification technique as follows. For each of the 40 traces listed in Table 5.3, we compare the state inference results for *(i)* the original trace (*e.g.,* Figure 5.4a) and *(ii)* a trace with promotion delays removed then RRC states reconstructed, but without any packet removed (*e.g.,* Figure 5.4e). Ideally their RRC inference results should be the same. Our comparison results show that for each of the 40 traces, both inference results are almost identical as their time overlap (defined in §3.2.2) is at least 99%, and their total radio energy consumption values differ by no more than 1%. The small error stems from the difference between original promotion delays and injected new promotion delays using fixed average values. This demonstrates that the algorithm faithfully reconstructs the RRC states. In §5.5, we show resource consumption computed by both Method 1 and Method 2.

5.4 Implementation

We briefly describe how we implemented ARO. We built the data collector on Android 2.2 by adding two new features (1K LoC) to tcpdump: logging user inputs and finding packet-to-application correspondence (§5.2). ARO reads `/dev/input/event*` that captures all user input events such as touching the screen, pressing buttons, and manipulating the tracking ball.

Finding the packet-to-application correspondence is more challenging. The ARO data collector realizes this using information from three sources in Android OS: `/proc/PID/fd` containing mappings from process ID (PID) to inode of each TCP/UDP socket, `/proc/net/tcp(udp)` maintaining socket to inode mappings, and `/proc/PID/cmdline` that has the process name of each PID. Therefore socket to process name mappings, to be identified by the data collector, can be obtained by correlating the above three pieces of information. Doing so once for all sockets takes about 15 ms on Nexus One, but it is performed only

when the data collector observes a packet belonging to a newly created socket or the last query times out (we use 30 seconds).

The runtime overhead of the data collector mainly comes from capturing and storing the packet trace. When the throughput is as high as 600 kbps, the CPU utilization of the data collector can reach 15% on Nexus One although the overhead is much lower when the throughput is low. There is no noticeable degradation of user experience when the data collector is running.

The analyzers were implemented in C++ on Windows 7 (7.5K LoC). The analysis time for the entire workflow shown in Figure 5.1 is usually less than 5 seconds for a 10-minute trace. As mentioned in §5.2, ARO configures the RRC analyzer with handset and carrier specific parameters. Currently our ARO prototype supports one carrier (Carrier 1 in Table 3.1) and two types of handsets (HTC TyTn II and Nexus One in Table 3.3). The RRC analyzer for other carriers can be designed in a way following §3.2. Differences among handsets mainly lie in radio power consumption and the fast dormancy behavior (§2.5) that are easy to measure. Also note that TCP, HTTP, and burst analyzers are independent of specific handset or carrier.

5.5 ARO Use Case Studies

To demonstrate typical usage scenarios of ARO, we present case studies of six real Android applications listed in Table 5.3 and describe their resource inefficiencies identified by ARO. All applications described in this section are in the "Top Free" section of Android Market and have been downloaded at least 250,000 times as of December 2010. The handset used for experiments is a Google Nexus One phone with fast dormancy (its α and β timers are 5 sec and 3 sec, respectively as shown in Figure 2.7). For identified inefficiencies, their resource waste is even higher if fast dormancy is not used. All experiments were performed between September 2010 and November 2010 using Carrier 1's UMTS network whose RRC state machine is depicted in Figure 2.2. As of early 2012, some application

Table 5.3: Case studies of six popular Android applications

App name	Mode*	Traces	Description (Recommendations)	Similar apps	Layers
Pandora §5.5.2.1	B	3	High resource overhead of periodic audience measurements (Delay transfers and batch them with delay-sensitive transfers)	Fox News, Tune-in Radio	RRC App
Fox News §5.5.2.2	F	5	Scattered bursts due to scrolling (Transfer them in one burst)	USA Today	RRC App User
			Transferring duplicated contents (Use the "Expires" HTTP header)	NY Times	App
BBC News §5.5.2.3	F	10	Inefficient content prefetching (Use HTTP pipelining to transfer multiple small objects for networks with high bw-delay product)	NY Times	TCP App
			Scattered bursts of delayed FIN/RST packets (Close a TCP connection immediately if possible, or within the tail time)	CBS News, Google Shopper	RRC TCP App
Google Search §5.5.2.4	F	15	High resource overhead of query suggestions and instant search (Balance between functionality and resource when battery is low)	Bing Search, Yahoo Search	RRC App User
Tune-in Radio §5.5.2.5	F	5	Low DCH utilization due to constant-bitrate streaming (Buffer data and periodically stream data in one burst)	NPR Radio, Iheartradio	RRC App
Mobclix §5.5.2.6	F	2	Aggressive ad refresh rate making a handset persistently occupy FACH or DCH (Decrease the refresh rate, piggyback or batch ad updates)	Apps with Mobclix ads embedded	RRC App

+ The analyses were performed between September 2010 and November 2010.

* Mode B = Background, Mode F = Foreground.

developers (*e.g.,* Pandora) were actively improving their applications based on findings brought by the ARO tool.

5.5.1 Experimental Methodology

Our experimental methodology is straightforward: for each application studied, we collected a trace by running the application for at least 5 minutes (except for Google Search), then used ARO to analyze the trace. Several factors including user behavior randomness, traffic of non-target applications, and radio link quality, may affect the data collected and henceforth the analysis results. Clearly, the discovered traffic patterns should be *stable* in that they are inherent to the application logic and thus are not affected by user behavior randomness in common application usage scenarios. To ensure this, for each application, we analyzed at least 5 traces collected by at least 3 students who used the application as normal users (except for Pandora and Mobclix that do not involve user interaction). A case listed in Table 5.3 was reported only if the same symptom was observed in *all* collected traces so that we had high confidence that the observed traffic patterns stemmed intrinsically from the application logic (although it is still useful to learn uncommon problems from individual traces).

To minimize the impact of non-target applications that concurrently access the network, we discarded a trace if any of the transferred bytes, the radio energy, or the DCH time caused by `NON_TARGET` bursts (Table 5.2) was greater than 5% of the total bytes, the total radio energy, or the total DCH time, respectively. We did similar filtering by examining `TCP_LOSS_RECOVER` bursts. Further, to minimize the impact of poor radio link quality, we collected all traces at reasonable signal strength conditions.

5.5.2 Results

We now describe case studies for the six applications. As shown in Table 5.3, for each application, we also found other popular applications with the same problem identified

Table 5.4: Pandora profiling results (Trace len: 1.45 hours)

Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
LARGE_BURST	96.4%	35.6%	35.9%	42.4%	42.5%
APP_PERIOD	0.2%	45.9%	46.7%	40.4%	40.9%
APP	3.2%	12.8%	13.4%	12.4%	12.8%
TCP_CONTROL	0.0%	1.2%	1.6%	1.1%	1.5%
TCP_LOSS_RECOVER	0.2%	0.2%	0.6%	0.3%	0.7%
NON_TARGET	0.0%	1.8%	1.8%	1.7%	1.7%
Total	23.6 MB	846 J		895 sec	

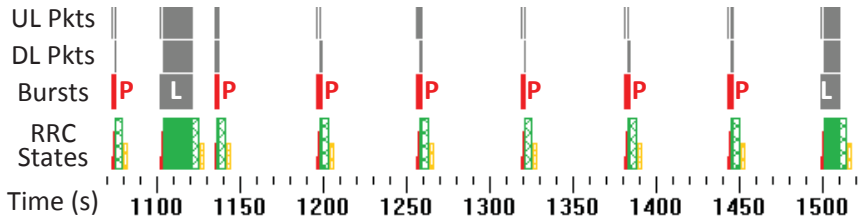


Figure 5.5: Pandora visualization results. "L" (grey) and "P" (red) bursts are LARGE_BURST and APP_PERIOD bursts, respectively.

by ARO. The last column of Table 5.3 shows the layers that are related to the identified inefficiency. *RRC*, *TCP*, *App*, and *User* correspond to the RRC layer, the transport layer, the application layer, and the user input layer, respectively.

5.5.2.1 Pandora Streaming

Pandora is a popular music streaming application. We collected three Pandora traces by simply listening to the music for at least 1 hour for each trace. Table 5.4 shows the profiling results of one trace, and the results for other traces are qualitatively similar. The "UB" (upperbound) and "LB" (lowerbound) columns in Table 5.4 refer to the resource consumption computed by Method 1 and 2 described in §5.3.3, respectively. Numbers in the "LB" column do not necessarily add up to 100%.

High resource overhead of periodic audience measurements. The profiling results in Table 5.4 indicate that periodic data transfers (APP_PERIOD bursts), which carry only 0.2% of total bytes, account for 46% of total radio energy consumption and 40% of radio

Table 5.5: Fox News profiling results (Trace len: 10 mins)

Prefetching Phase					
Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
USER_INPUT(Click)	91.0%	56.7%	67.6%	60.2%	70.4%
USER_INPUT(Scroll)	5.9%	15.2%	17.9%	14.7%	16.7%
APP_PERIOD	1.5%	5.2%	7.5%	6.1%	7.4%
TCP_CONTROL	0	0.7%	3.7%	0.0%	2.3%
TCP_LOSS_RECOVER	1.5%	0.7%	2.5%	1.9%	3.2%
SVR_NET_DELAY	0.1%	0.4%	0.8%	0.0%	0.0%
Total	1.0 MB	276 J		284 sec	

resource usage. The detection algorithm in Figure 5.3 further pinpoints that for every 62.5 seconds, Pandora connects to `lt.andomedia.com`, which provides various real-time audience measurement services (*e.g.,* monitor online listeners' favorite radio stations), and downloads hundreds of bytes. Each such a burst, however, triggers an IDLE→DCH promotion and subsequently two tails of 8 seconds in total. On the other hand, Pandora usually takes less than 30 seconds to download a song (a `LARGE_BURST` burst) that can be played for several minutes. As illustrated in Figure 5.5, the total DCH occupation time for periodic data transfers is similar to or even longer than the music streaming time, although the former carries much fewer and much less important user data than the latter. The `APP` bursts shown in Table 5.4 correspond to non-periodic transfers of album images and other metadata. Here one straightforward fix is to increase the periodicity. A more intelligent approach is to batch such delay-tolerant transfers with delay-sensitive transfers to reduce the overall tail time [14].

5.5.2.2 Fox News

Fox News is a popular news application. We obtained five traces from three users who browsed the news headlines or articles as they like for at least 5 minutes. Table 5.5 exemplifies profiling results of one representative trace. The results indicate that ARO is essential in quantitatively breaking down resource consumption into bursts with their

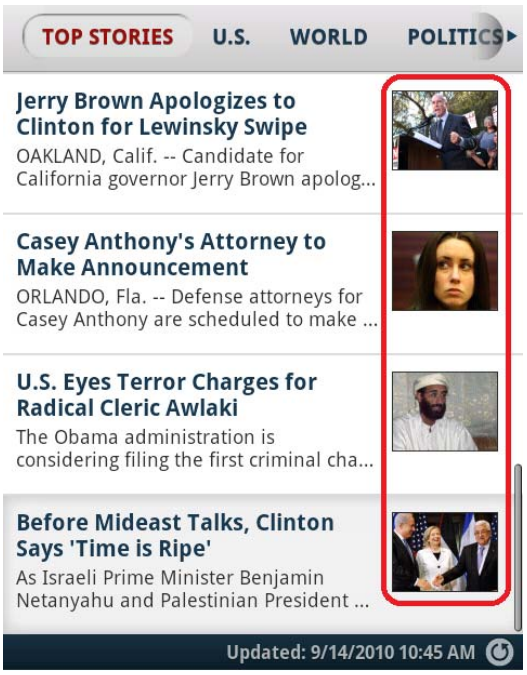


Figure 5.6: Headlines of the Fox News application. The thumbnail images (highlighted by the red box) are transferred only when they are displayed as a user scrolls down the screen.

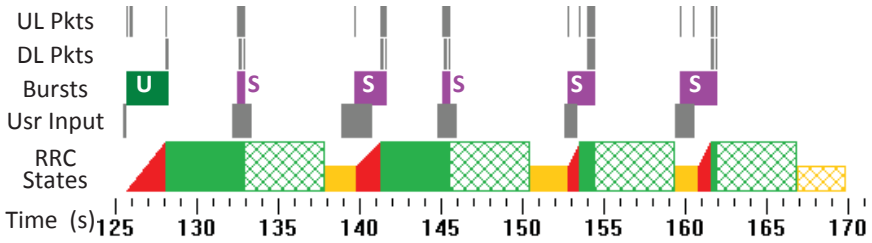


Figure 5.7: The Fox News results. "U" (green) and "S" (purple) bursts are triggered by tapping and scrolling the screen, respectively.

triggering factors inferred.

Scattered bursts due to scrolling. Table 5.5 indicates that the majority of resources are spent on bursts initiated by user interactions. Among them, about 15%∼18% of radio energy is responsible for bursts generated when a user scrolls the screen. By examining HTTP responses associated with such bursts, we discover that thumbnail images embedded in headlines (Figure 5.6) are transferred only when they are displayed as a user scrolls down the screen. Thus as illustrated in Figure 5.7, when a user browses the headlines, the handset always occupies the DCH state due to such on-demand transfers of thumbnails. On the other

hand, each thumbnail has very small size (less than 5KB each). A suggested improvement is to download all thumbnails (usually less than 15) in one burst. Doing so significantly shortens the overall DCH occupation time for headline browsing with negligible bandwidth overhead incurred. We observe this problem for other news applications (*e.g.,* USA Today) that use the same application framework.

Transferring duplicate contents. The HTTP analyzer (§5.3.1) extracts HTTP objects from the trace. We discovered that often, the same content is repeatedly transferred, leading to waste of bandwidth. For example, Fox News fetches the same object `http://foxnews.com/weather/feed/getWeatherXml` whenever a news article is loaded, and the response from the server (45 KB) is identical unless the weather information, updated hourly, changes. The problem can be fixed by letting the server put an "Expires" header in an HTTP response to explicitly tell the client how long the content can be cached [47].

5.5.2.3 BBC News

BBC News is another news application. Unlike Fox News, which fetches an article only when a user wants to read it, the network usage of BBC News consists of two phases: prefetching and user-triggered data fetching.

Inefficient content prefetching. Prefetching happens when a news category (*e.g.,* Sports), which is not yet cached or is out-of-date, is selected by a user. In the prefetching phase, the application downloads the headline page with thumbnails, and more aggressively, contents of all articles of the selected news category in a single large burst. While it is arguable whether aggressive prefetching, which efficiently utilizes radio resources but wastes network bandwidth as some contents may not be consumed by end users, is a good strategy, the prefetching of BBC News is performed inefficiently. It takes up to two minutes for BBC News to prefetch all articles (*e.g.,* 60 articles in one trace) of a news category. The HTTP analyzer reveals that the application issues one single HTTP GET for each article,

Table 5.6: BBC News profiling results

Prefetching Phase (1.4 mins)					
Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
LARGE_BURST	100%	100%	100%	100%	100%
Total	1.1 MB	60.1 J		82.8 sec	

User-triggered Fetching Phase (8 mins)					
Burst type	Payloads	Energy		DCH	
		LB	UB	LB	UB
TCP_CONTROL	0	11.3%	24.2%	0.0%	5.7%
USER_INPUT	98.7%	42.5%	73.1%	37.9%	90.0%
SVR_NET_DELAY	1%	0.0%	2.7%	0.0%	5.2%
Total	162 KB	145 J		120 sec	

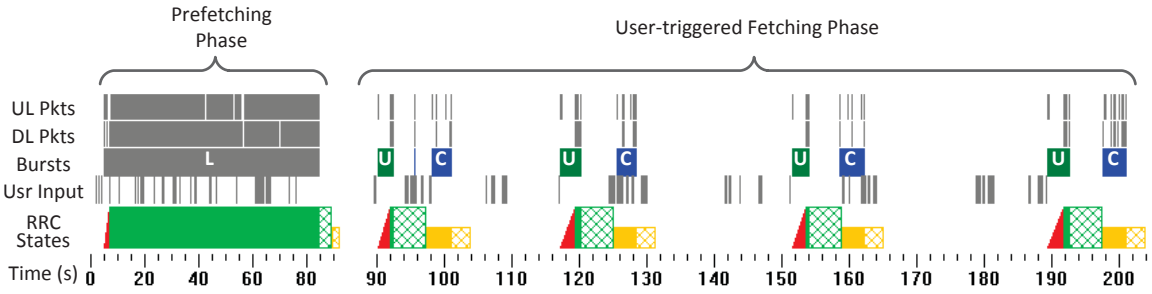


Figure 5.8: BBC News results: prefetching followed by 4 user-triggered transfers. "U" (green), "C" (blue), and "L" (grey) bursts are USER_INPUT, TCP_CONTROL, and LARGE_BURST bursts, respectively.

then waits for the response before issuing the next HTTP GET. A more efficient approach is HTTP pipelining, *i.e.,* the application sends all 60 URLs in a single HTTP GET and hence the server transfers all articles without interruption. Given the scenario where many small objects are transferred in a network of high bandwidth-delay product, HTTP pipelining, which is widely supported by modern web servers, dramatically improves the throughput by eliminating unnecessary round trips and allowing more outstanding (*i.e.,* in-flight) data packets with almost no head-of-line blocking overhead [48].

Scattered bursts due to delayed FIN/RST. After prefetching, clicking on an article triggers very little traffic. However, as shown in Table 5.6, TCP_CONTROL bursts, which do not carry any user payload, consume 11%∼24% of the radio energy. Such TCP_CONTROL

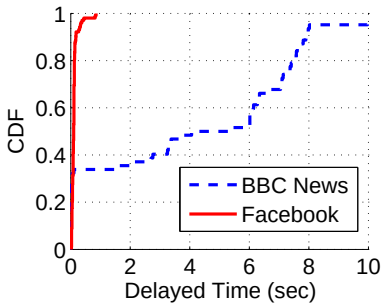


Figure 5.9: Distribution of delayed time for FIN or RST packets for BBC News and Facebook applications

bursts are FIN or RST packets, *i.e.,* the application delays closing TCP connections. As shown in Figure 5.8, they waste radio energy by causing additional FACH occupation time.

Delayed FIN or RST packets are caused by connection timeout maintained by either an HTTP client or server that uses persistent HTTP connections. Different applications may use different timeout values since the HTTP 1.1 protocol places no requirements on how to set the value [49]. We observe that some applications (*e.g.,* Facebook and Amazon Shopper) always immediately shut down a connection, while BBC News may delay closing a connection by up to 15 seconds after the last HTTP response is transmitted. In our traces, 50% of its FIN/RST are delayed by at least 5 seconds, which is the α timer value, potentially triggering a FACH$\rightarrow$DCH promotion.

Figure 5.9 plots distributions of delayed time for FIN or RST packets for two application traces. Facebook always immediately shuts down a connection, while BBC News may delay closing a connection by up to 15 seconds after the last HTTP response is transmitted. We observe from traces that most FIN and RST packets are initiated by a handset instead of by a server.

Eliminating delayed FIN/RST packets saves resources, but closing a connection too early may prevent it from being reused, thus incurring additional overhead for establishing new connections. A compromise is to close the connection before the α timer expires to avoid a state promotion triggered by delayed FIN/RST. For Carrier 1, doing so further benefits handset battery life, as usually FIN and RST packets do not reset the α timer

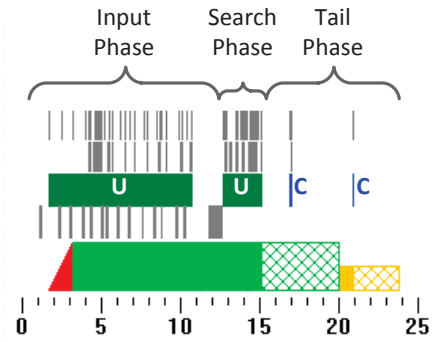


Figure 5.10: ARO visualization results for Google search

due to their small sizes (§3.1.2). Smartphone OS can help applications properly close TCP connections by collecting hints from applications and employing different connection timeout values depending on the carrier type.

5.5.2.4 Google Search

Search is among the most popular browsing activities on smartphones [42]. Almost all search engines provide real-time query suggestions as a user types keywords in the search box. We show that such a feature consumes significant radio energy (up to 78%) and radio resources (up to 76%) by conducting a user study.

Five student users participated in our user study. Each student searched three keywords in mobile version of Google using Nexus One: "university of michigan", "ann arbor", and "android 2.2". A trial is abandoned if any typing mistake was made (typing mistakes worsen the resource efficiency). The participants were asked to use the query suggestion whenever possible. Browser caches were cleared before each trial. We believe these keywords are representative although the length and popularity of keywords may affect the results.

High resource overhead of real-time query suggestions and instant search. We obtained 15 traces (3 keywords searched by 5 users) which were further analyzed by ARO. We broke down each trace into three phases: *(i)* Input Phase, *i.e.,* a user is typing a keyword. *(ii)* Search Phase, *i.e.,* after a user submits the keyword, and before the last byte

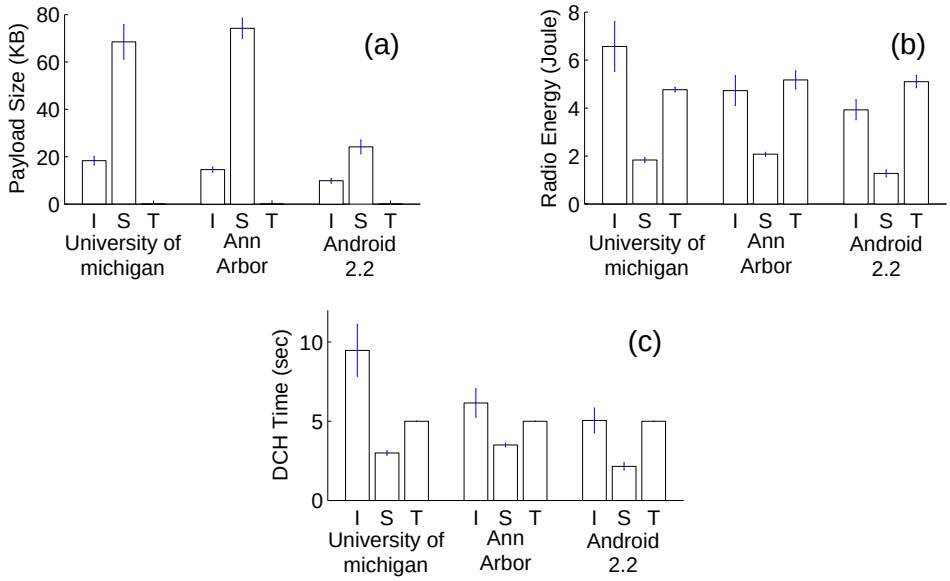


Figure 5.11: Breakdown of (a) transferred payload size (b) radio energy (c) DCH occupation time for searching three keywords in Google. "I", "S", "T" correspond to Input Phase, Searching Phase, and Tail Phase, respectively.

of the search results is received. *(iii)* Tail Phase, *i.e.,* the remaining time until the RRC state is demoted to IDLE. An example for searching "university of michigan" is shown in Figure 5.10. Subsequently, ARO computes transferred payload bytes (Figure 5.11-a), radio energy consumption (Figure 5.11-b), and DCH time (Figure 5.11-c) for each phase. Each plot of Figure 5.11 consists of results of the three keywords. For each keyword, "I", "S", and "T" correspond to Input Phase, Search Phase, and Tail Phase, respectively. Figures 5.10 and 5.11 clearly show that while a user is typing a keyword, real-time query suggestions keep the handset at DCH, consuming 2.3 to 3.5 times of radio energy and 1.8 to 3.2 times of DCH time, compared to those consumed by Search Phase. We note that a similar problem occurs for Google instant search (results appear instantly as a user types a keyword) that is available for Android since Nov 2010 [50].

Query suggestions and instant search improve user experience. However, realizing their high resource impact in cellular network, the application can balance between functionality and resource when the latter becomes a bottleneck (*e.g.,* the battery is critically low). For example, using historical keywords and a local dictionary to suggest search hints is an

Table 5.7: Constant bitrate vs. bursty streaming

Name	Server	bitrate	Radio Power
NPR News	SHOUTcast	32 kbps^{+}	36 J/min
Tune-in	Icecast	119 kbps	36 J/min
Iheartradio	QTSS	32 kbps	36 J/min
Pandora	Apache	bursty	11.2 J/min
Pandora w/o mes*	Apache	bursty	4.8 J/min
Slacker	Apache	bursty	10.9 J/min

*A hypothetical case where all periodic audience measurement data transfers are removed.
$^{+}$NPR News also uses a higher bitrate of 128 kbps for some content.

alternative but with worse functionality.

5.5.2.5 Tune-in Radio (and Other Streaming Apps)

The Tune-in Radio application delivers live streams of hundreds of FM/AM radio stations. Table 5.7 further lists NPR News and Iheartradio, two popular live radio streaming applications similar to Tune-in Radio. All three applications employ existing radio streaming schemes that work well on wired networks and WiFi: the server streams data at a constant bitrate (*e.g.,* 32 kbps) to a client without any pause.

Low DCH utilization due to constant-bitrate streaming. In cellular networks, however, continuously streaming at a constant low bitrate causes considerable inefficiencies on resource utilization, as a handset is always using the DCH channel, whose available bandwidth is significantly under-utilized, whenever a user is listening to the radio. Table 5.7 compares constant-bitrate streaming to the bursty streaming strategy employed by Pandora and Slacker Radio where a program is buffered in one burst utilizing the maximum available bandwidth then the application does not access the network while playing the program. The last column of Table 5.7 indicates that for the two streaming strategies, their energy efficiency, *i.e.,* the average radio energy consumption for listening to the radio for 1 minute, differs by up to 7.5 times. For radio programs whose real-time is not strictly required (*e.g.,* their delivery can be delayed by one minute), a live streaming server can also perform

Table 5.8: Comparing three mobile ad platforms

Name	Default Refresh Rate	Avg Update Size	Radio Power	
			w/ FD	w/o FD
Google Mobile Ad	180.0 sec	6.0 KB	2.5 J/min	3.6 J/min
AdMob	62.5 sec	6.8 KB	5.7 J/min	8.8 J/min
Mobclix	15.0 sec	1.4 KB	23.2 J/min	29.6 J/min

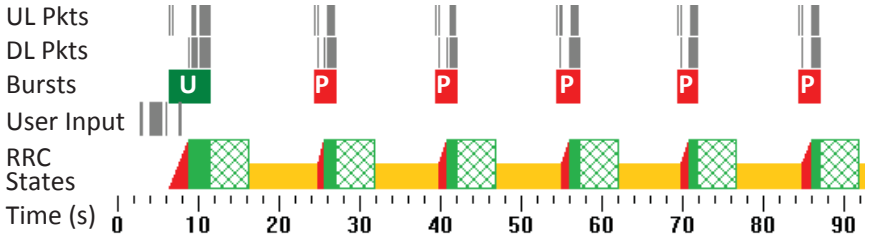


Figure 5.12: Results for Mobclix (w/o FD). "U" (green) and "P" (red) bursts are `USER_INPUT` and `APP_PERIOD` bursts, respectively.

similar bursty streaming to save handset energy and radio resources by buffering data and periodically streaming data in one burst.

5.5.2.6 Mobclix (and Other Mobile Ad Platforms)

We investigate advertisement dissemination strategies for three popular mobile ad platforms: Google Mobile Ad, AdMob, and Mobclix. All platforms allow developers to easily display ads in their applications by providing simple SDKs. We thus built three toy Android applications each using one ad platform with its default configuration. Then we employed ARO to profile each ad-embedded application for five hours, using two Nexus One phones (Nexus One A and B described in §2.5) where one has a shorter FACH→IDLE timer (3 sec) due to fast dormancy (the "w/ FD" column in Table 5.8) and the other uses default timers of Carrier 1 without fast dormancy (the "w/o FD" column). Also note that our toy applications themselves do not have any network activity so the network traffic is solely generated by ad modules.

Aggressive ad refresh rate. We discuss the profiling results. As identified by ARO, all ad platforms by default employ a fixed refresh rate for updating the ads. For example,

an application using AdMob pings `r.admob.com` for every 62.5 sec. Then the server may either push a new ad or let the application display the existing ad. Surprisingly, as summarized in Table 5.8, the three platforms use considerably different refresh rates, leading to remarkable disparity of their radio power consumption, especially for applications without network activities (*e.g.,* games). In particular, as illustrated in Figure 5.12, Mobclix employs an aggressive refresh rate of 15 sec that is even shorter than the default tail time of Carrier 1 (17 sec when fast dormancy is not used), making the handset persistently occupying DCH or FACH whenever the application is running.

5.6 Summary

We have presented ARO, the first tool that exposes the cross-layer interaction for layers ranging from radio resource control to application layer. We have demonstrated using popular mobile applications that ARO helps reveal several previously unknown, general categories of inefficient resource usage, affecting distinct classes of mobile applications due to a lack of transparency in the lower-layer protocol behaviors. In particular, we are starting to contact developers of popular applications such as Pandora. The feedback has been encouragingly positive as the provided technique greatly helps developers identify resource usage inefficiencies and improve their applications [40]. The ARO prototype [51] has been productized by AT&T and is now available to developers [52].

Our work opens and enables new research opportunities for *(i)* analyzing other cross-layer information (*e.g.,* cellular handoff events, OS events, and application activities) for more fine-grained diagnosis of performance and resource utilization inefficiencies, and *(ii)* providing automated mitigation solutions to identified inefficiencies. We plan to explore them in our future work.

CHAPTER VI

Optimizing Radio Resource Usage Through Adaptive Resource Release

6.1 Introduction

Chapter IV indicates that the tail effect (§2.1.3) causes significant resource inefficiency in cellular networks due to the fundamental tradeoff in resource allocation (§2.4). The ARO tool introduced in Chapter V mitigates this by identifying inefficient traffic bursts so they can be potentially optimized to reduce the tail time (*e.g.,* grouping several bursts into a single large burst). But it does not work for all types of traffic patterns. For example, it is difficult to use ARO to optimize web browsing traffic consisting of delay-sensitive bursts initiated by users.

In this chapter, we address the problem of mitigating the tail effect by leveraging the fast dormancy feature described in §2.5. It is complementary to the handset-based traffic shaping approach, with more general applicability. Before introducing our proposed solution, let us first revisit existing approaches for tail removal, which can be classified into three categories.

Tuning inactivity timers. Previous work [19, 20] propose tuning inactivity timers using analytical models by considering radio resource utilization, handset energy consumption, service quality, and processing overheads of the radio access network. However,

mitigating the tail effect requires reducing the inactivity timers. Doing so inevitably causes the number of state transitions to increase. Based on our measurements using real traces collected from a large UMTS carrier, we found that aggressively reducing the most critical timer from 5s to 0.5s reduces the tail time by 50%, but increases the state promotion delay by about 300%. This introduces significant processing overhead for the radio access network [13], and increased delay for users.

Handset-based approach. The handset alters traffic patterns based on the prior knowledge of the RRC state machine. For delay-tolerant applications such as Email and RSS feeds, data transfers can be delayed and batched to reduce the tail time [14]. However, as mentioned before, such an approach is not suitable for more interactive applications such as Web browsing, otherwise users may suffer from delayed processing of their requests.

Cooperation between the handset and the network. The handset applications may be able to predict the end of a data transfer based on the application logic. If an idle time period that lasts at least as long as as the inactivity timer value is predicted, the handset sends a message to notify the network, which then immediately releases allocated resources. This approach can thus completely eliminate the tail if the prediction is accurate, without incurring additional promotion delays. As mentioned in §2.5, a feature called *Fast Dormancy* has been proposed to be included in 3GPP [23] to help realize this approach. Note that although this is a standard already adopted by several handsets [24], to the best of our knowledge, no smartphone application to date uses fast dormancy, partly due to a lack of OS support, as discussed in §4.5.2.

In this chapter, we propose Tail Optimization Protocol (TOP), an application-layer protocol that bridges the gap between the application and the fast dormancy support provided by the network. Some of the key challenges we address include the required changes to the OS, applications, and the implication of multiple concurrent connections using fast dormancy. In particular, TOP addresses three key issues associated with allowing smartphone applications to benefit from this support.

First, our work is the first to propose a simple interface for different applications to leverage the fast dormancy feature. In our framework, applications define their logical *transfers* and perform predictions of inter-transfer times. The prediction can be easy accomplished for applications having limited or no user interaction (*e.g.,* video streaming), but it is more challenging for user-interactive applications such as Web browsing. The prediction methodology is not our focus in this work. The application invokes a tail removal API provided by TOP that automatically coordinates concurrent traffic of multiple applications, as state transitions are determined by the aggregated traffic of all applications. Our design minimizes applications' implementation overhead for tail removal. Note that our proposed framework is also applicable to the 3G EvDO (Evolution-Data Optimized) and the 4G LTE (Long Term Evolution) cellular networks that also use inactivity timers for releasing radio resources and therefore have the tail effect [10, 53].

Second, by using cellular traces collected from a large UMTS carrier, we are the first to quantify the tail effect for nearly a million user sessions. We found that for the two RRC states, 34.4% and 76.8% of the time is spent on the tail. By using the traces, we also empirically derive critical parameters used by TOP to properly balance the tradeoff between the resource saving and the state transition overhead.

Third, we demonstrate the benefits of TOP using real traces collected from a UMTS carrier and from our Android smartphones. With a reasonable prediction accuracy, TOP saves the overall radio energy (up to 17%) and radio resources (up to 14%) by reducing up to 60% of the tail time. For some applications such as multimedia streaming, TOP can achieve even more significant savings of radio energy (up to 60%) and radio resources (up to 50%).

The rest of this chapter is organized as follows. §6.2 provides an overview of our proposal. §6.3 describes the cellular measurement data used in this chapter. After empirically quantifying the tail effect using our dataset in §6.4, we describe our proposed Tail Optimization Protocol (TOP) in §6.5 and evaluate it in §6.6 before concluding the chapter

in §6.7.

6.2 Overview

The high-level idea behind our proposed Tail Optimization Protocol (TOP) is straightforward. It involves invoking the fast dormancy support (§2.5) that directly triggers a DCH→IDLE or a FACH→IDLE demotion without experiencing timeout periods, in order to save radio energy and radio resources. However, doing so aggressively may incur unacceptably long delay of state promotions, worsening user experience and increasing processing overheads at the RNC. TOP employs a set of novel techniques to address this key challenge by letting individual applications predict tails and coordinating tail prediction of concurrent applications for invoking fast dormancy. Our design requires no changes at a handset's firmware/hardware given that fast dormancy is widely deployed, and is transparent to the radio access network.

- TOP leverages the knowledge of applications that *predict* the idle period after each data transfer. The definition of a data transfer depends on the application. Fast dormancy is not invoked if the predicted idle period is smaller than a predefined threshold called *tail threshold* to prevent unnecessary state promotions (§6.5.1).

- As described in §6.5.3, we carefully tune the value of the tail threshold and other parameters used by TOP by empirically measuring traces collected from a large UMTS carrier (§6.3), in order to well balance the tradeoff (§6.4.2) between the incurred state promotion overhead and resource savings.

- The RRC state transitions are determined by the *aggregated* traffic of all applications running on a handset. TOP introduces a novel coordination algorithm to handle concurrent network activities. TOP also handles tail optimization for legacy applications that are themselves unaware of TOP. We detail the coordination algorithm design in §6.5.

6.3 The Measurement Data

This section describes the data used in our study. Our dataset is a large TCP header packet trace collected from Carrier 1 on April 13, 2009 in the normal course of operations. The collection point is at the core network (CN) that primarily serves UMTS users but also 2G GPRS users. Our trace contains 265 million TCP packets (169 GB data) continuously captured in 1.3 hours without any sampling in either direction. Due to concerns of the large traffic volume and user privacy issues, we only recorded TCP/IP headers and a 64-bit timestamp for each packet, but no subscriber IDs or phone numbers.

We subsequently extract *sessions* from the trace, each consisting of all packets transferred by the same handset identified through the private client IP address in the trace. Multiple TCP flows from concurrent applications may be mixed in the same session. We use a threshold of 60 sec of idle time to detect session termination. A different threshold value, *e.g.,* 45 or 75 sec, does not qualitatively affect the analysis results.

We will use this dataset in §6.4.1, §6.5.3, and §6.6.1. Our common methodology is to replay sessions against a program simulating the RRC state machine with desired settings, and a tail removal algorithm to be studied, *e.g.,* TOP, to obtain statistics about the state machine's behavior. Before that, timestamps of the original trace were first *calibrated* to eliminate promotion delays caused by the existing state machine of Carrier 1. We detail the calibration methodology in our measurement work [5]. Then the calibrated trace, whose promotion delays are zero, can be applied to a different state machine, and new promotion delays are injected separately by the simulator. The calibration procedure also detects sessions (about 17%) that violate the RRC state machine. Such sessions are mostly caused by non-UMTS traffic mixed in the trace. They are not used in our subsequent data analysis.

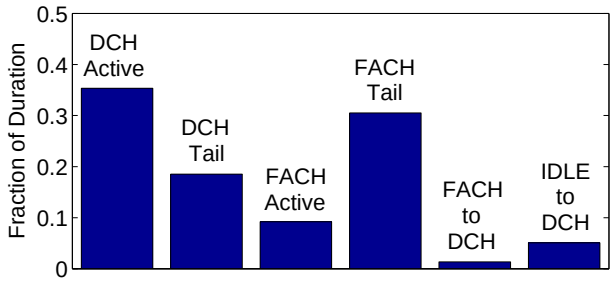


Figure 6.1: Breakdown of the duration of all sessions for Carrier 1

6.4 The Tail Effect

We briefly revisit the cellular tail effect described in §2.1.3. At DCH or FACH, when there is no user data transmission in either direction for at least T seconds, *i.e.,* the inactivity timer value, the RRC state will be demoted to save radio resources and handset energy. However, during the wait time of T seconds, a handset still occupies the transmission channel and WCDMA codes, and its radio power consumption is kept at the corresponding level of the state. We define a *tail* as the idle time period matching the inactivity timer value before a state demotion. We also refer to any non-tail time as *active*.

In typical UMTS networks, each handset is allocated dedicated channels whose radio resources are completely wasted during the tail time. For HSDPA [3], which is a UMTS extension with higher downlink speed described in §2.1.2, although the high speed transport channel is shared by a limited number of handsets (*e.g.,* 32), occupying it during the tail time can potentially prevent other handsets from using the high speed channel. Furthermore, tail time wastes handset radio energy, which contributes to up to half of the total battery energy consumption of a handset based on our measurements using a power meter in §3.2.2.1.

6.4.1 Measuring the Tail Time

We quantify the tail time by studying the trace described in §6.3. We feed the 0.82 million calibrated sessions into a simulator (§3.2) for Carrier 1's state machine whose state

machine transitions and parameters are shown in Figure 2.2 and Table 3.1. We decompose the total duration of all sessions into six components: DCH active time, DCH tail time, FACH active time, FACH tail time, the FACH→DCH promotion delay, and the IDLE→DCH promotion delay, and then plot the fraction of each component in Figure 6.1, which clearly shows that considerable amount of time on DCH and FACH is wasted by the tail effect. On DCH, 34.4% of the time belongs to the tail, and on FACH, 76.8% of the time is spent on the tail, which is even longer than the FACH active time, as the β (FACH→IDLE) timer is set to be as long as 12 seconds. For Carrier 2, 35.3% and 72.0% of the DCH and FACH time belong to the tail, respectively.

6.4.2 Tradeoff Considerations to Optimize Radio Resources

From the carrier's perspective, the most naïve way to mitigate the tail effect is to reduce the inactivity timer values. However, doing so inevitably increases the signaling load. This is the key tradeoff discussed at the beginning of the dissertation (§2.4). Similar to our analysis for changing the inactivity timer values (§4.4), we quantify the tradeoff using the resource consumption D, the signaling overhead S, and the handset radio energy E. Besides them, we use D^T to denote radio resources wasted on DCH tails, so D consists of D^T and radio resources spent on non-tail DCH times. We compute D^T, D, S and E using the simulation-based approach described in §3.4.

When changing inactivity timer values or using a new technique for tail removal, we are interested in relative changes of D, S, E, and D^T compared to the default setting where we use the default state machine parameters (Table 3.1) without removing tails. We capture this using ΔD, ΔS, ΔE, and ΔD^T introduced in §2.4. For example, Let D_1^T and D_0^T be the DCH tail time in the new setting and in the default setting, respectively. The relative change of D^T, denoted as ΔD^T, is computed by $\Delta D^T = (D_1^T - D_0^T)/D_0^T$. We have similar definitions for ΔD, ΔS and ΔE, which will be revisited in §6.5.3 and §6.6.

6.5 Tail Optimization Protocol

In this section, we describe our proposed Tail Optimization Protocol (TOP), an application-layer protocol that leverages the support of fast dormancy to remove tails. In TOP, applications define *data transfers* and predict the inter-transfer time at the end of each data transfer (§6.5.1) using a simple interface described in §6.5.2. If the predicted inter-transfer time is greater than a *tail threshold*, the application informs the RNC to initiate fast dormancy. We discuss how to set the tail threshold in §6.5.3 and describe how TOP handles concurrent network activities in §6.5.4.

Our design of TOP requires small changes at handset applications (and optionally server applications, as a server may provide a handset with hints about predicting a tail) and the handset OS, but no change at a handset's firmware/hardware given that fast dormancy is widely deployed. Also TOP is transparent to the UTRAN and CN. Therefore TOP is incrementally deployable. Note that the $\boldsymbol{T}$ message (*i.e.,* the fast dormancy request, see §2.5) is already supported by the RNC [23].

6.5.1 Feasibility of Tail Prediction

From applications' perspective, tail eliminations are performed for each *data transfer*, defined by applications to capture a network usage period. For example, a data transfer can correspond to all packets belonging to the same HTML page. To use TOP, an application only needs to do two things.

1. Ensure that the current data transfer has ended.

2. Provide TOP with its predicted delay between the current and the next data transfer, denoted as the *inter-transfer time* (ITT), via a simple interface described in §6.5.2. ITT is essentially the packet inter-arrival time between the last packet of a transfer and the first packet of the next transfer. Note that downlink (DL) and uplink (UL) packets are not differentiated as both use the same state machine.

We first consider the most simple scenario with no concurrent network activities. TOP sends a $\boldsymbol{T}$ message (*i.e.,* invoking fast dormancy) to eliminate the tail if the predicted ITT is longer than a threshold called *Tail Threshold* (TT). A large value of TT limits the radio resource and energy savings achieved by TOP while a small TT incurs extra state promotions. We justify how we choose TT in §6.5.3.

Clearly, the ITT prediction is application specific. It is easier to predict for applications with regular traffic patterns, with limited or no user interaction (*e.g.,* video streaming), but it is more difficult for user-interactive applications such as Web browsing and Google Map, as user behaviors inject randomness to the packet timing. For example, in Web browsing, each transfer corresponds to downloading one HTML page with all embedded objects. The browser knows exactly when the page has been fully downloaded. However, the timing gap between two consecutive transfers may be shorter than the tail threshold (*e.g.,* a user can quickly navigate between pages). Thus the browser should *selectively* invoke TOP. The second example is multimedia streaming. A streaming transfer consists of a single burst of packets of video/audio content (§6.6.2.1). The application usually can predict termination of a streaming burst. TOP can be applied if the timing gap between two consecutive bursts (usually known by the application) is longer than TT. As another example, interactive map applications involve continuous user interactions, thus TOP may not be applicable as it is very hard to define a transfer.

There are two issues related to ITT prediction. First, applications may not predict ITT accurately: misprediction can lead to increased promotion overhead due to predicting a short ITT less than TT to be a long ITT greater than TT, or lead to missing opportunities for tail removal due to predicting a long ITT to be short. A comprehensive study of prediction methodologies for interactive applications such as Web browsing is beyond the scope of this chapter and is our ongoing work. Here we assume ITTs are predicted with a reasonable accuracy (*e.g.,* 80% to 90%).

The second issue is that, the existence of concurrently running applications and independent components of the same application (*e.g.,* a streaming application with an advertisement bar embedded) further complicates tail prediction. Clearly applications cannot predict other applications' concurrent network activities that affect state transitions. But there is a need to look across applications when deciding whether to invoke fast dormancy. TOP is responsible for handling the concurrency as will be described in §6.5.4.

We also considered an alternative approach where the prediction is performed by the network instead of on the phone. In spite of the advantages of reducing the prediction overhead on handsets (the overhead is assumed to be small though), doing so has several limitations:

- Compared to mobile applications, the network has much less knowledge about the application logic, which in many cases determines the traffic patterns, leading to lower prediction accuracy of ITT.
- Compared to the data that can be collected on a handset, the information available to the network is much more limited. For example, user input and system events are potentially useful features for ITT prediction, but it is impossible for the network to obtain them.
- It is very difficult for the network to perform per-flow based or per-application based prediction, resulting in lower prediction accuracy when concurrency exists (§6.5.4).
- The handset-based prediction is more scalable than the network-based approach.

6.5.2 The Interface for Tail Removal

Unlike applications, TOP is unaware of the way applications define their transfers. TOP instead schedules tail removal requests at the *connection* level. A connection is defined as usual by five tuples: srcIP, dstIP, srcPort, dstPort, and protocol (TCP/UDP). Note that it is

possible that either one connection contains multiple transfers or one transfer involves multiple connections. At the end of a transfer, after the last packet is transmitted, an application informs TOP via a simple API

`TerminateTail(`c**`,` δ**`)`**

that the predicted ITT of connection c is δ. In other words, the next UL/DL packet of connection c belongs to the next transfer and will arrive after δ time units.

When user interactions are involved, it may be difficult for applications to predict the exact value of ITT. An application can then performs *binary* prediction *i.e.,* whether $ITT \leq TT$ or $ITT > TT$. The API is only called in the latter case: if ITT is predicted to be greater than TT, then `TerminateTail(`c`,` δ`)` is invoked with δ set to a fixed large value (*e.g.,* 60 sec). In fact, when no concurrent network activities exist, the exact prediction of ITT is not necessary at all as long as the binary prediction is correct.

On the other hand, when concurrency exists, the predicted ITT value may affect how fast dormancy is invoked. Let the real ITT be δ_0 and the predicted ITT be δ. Underestimating δ_0 ($\delta < \delta_0$) may prevent other concurrent applications from invoking fast dormancy and overestimating δ_0 ($\delta > \delta_0$) may incur additional state promotions. However, based on our empirical evaluation using real cellular traces in §6.6, we found that as long as the binary prediction is correct, the actual prediction value of ITT is much less important.

Calling `TerminateTail(`c`,` δ`)` indicates that the next transfer belongs to an *established* connection c. Also an application may start the next transfer by establishing a *new* connection that does not exist when `TerminateTail` is called. For example, a Web browser may use a new TCP connection for fetching a new page. Suppose that at the end of a connection c, the application makes a prediction that the next transfer initiates a new connection and the ITT is δ. In that case, the application needs to make two consecutive calls:

1. `TerminateTail(`$null$`,` δ`)`, indicating that a new connection will be established after δ time. The first parameter is $null$ as the application does not know about the

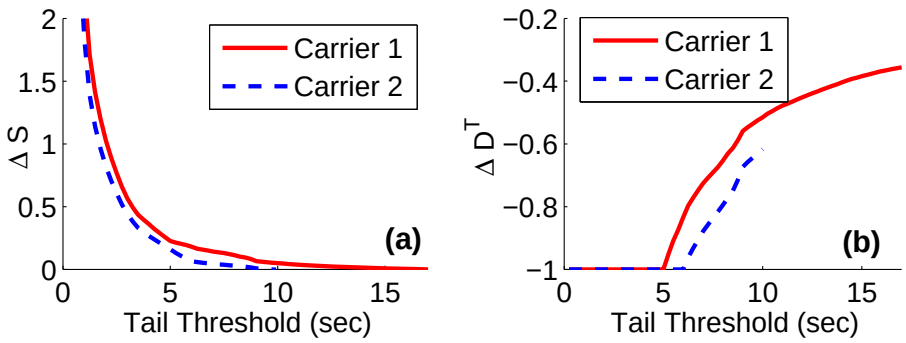


Figure 6.2: Impact of Tail Threshold (TT) on (a) ΔS (b) ΔD^T

future connection.

2. `TerminateTail(`c`, ` ∞`)`, indicating the termination of c and the termination of the current transfer. Note that the two calls trigger at most one $\boldsymbol{T}$ message. We explain why the second call is necessary in §6.5.4.

6.5.3 Determining the Tail Threshold Value

A *tail threshold (TT)* is used by TOP to determine whether to send a $\boldsymbol{T}$ message when `TerminateTail` is called. When no concurrency exists, TOP sends a $\boldsymbol{T}$ message if and only if the predicted ITT is greater than TT. A large value of TT may limit radio resource and energy savings, and a small TT may incur extra state promotions.

Usually a handset is at DCH when a transfer ends. Assuming this, a $\boldsymbol{T}$ message triggers a DCH→IDLE demotion. However, in the current state machine setting, a handset will experience two state demotions: DCH→FACH with duration of α, and FACH→IDLE with duration of β. Therefore, the default value of TT should be $\alpha + \beta$ to match the original behavior. In other words, assuming the predicted ITT is δ, to ensure no additional promotion occurs (if predictions are correct), TOP should not send a $\boldsymbol{T}$ message unless $\delta > \alpha + \beta = TT$. However, such a default value of TT is large (17 sec for Carrier 1 and 10 sec for Carrier 2), limiting the effectiveness of TOP.

The key observation here is based on our empirical measurement shown in Figure 6.2(a),

which is generated as follows. We replay calibrated sessions (§6.3) against both carriers' state machines with different TT values, assuming that a $\boldsymbol{T}$ message is sent whenever the packet inter-arrival time is greater than TT. We measure the change of the total duration of state promotions ΔS (§6.4.2). As expected, ΔS monotonically decreases as TT increases, and reaches zero when $TT = \alpha + \beta$. Also we empirically observe that reducing TT to the value of α incurs limited promotion overhead: 22% and 7% for Carrier 1 and Carrier 2, respectively. Note that given a fixed TT, Carrier 1 has a higher promotion overhead because it has a much longer β timer. On the other hand, as shown in Figure 6.2(b), which plots the relationship between TT and the change of the total DCH tail time ΔD^T, setting TT to $\alpha + \beta$ saves only 35% and 61% of the DCH tail time for the two carriers, respectively, but reducing TT to α eliminates all DCH tails (*i.e.,* $\Delta D^T = -1$). Therefore we set TT to α since it better balances the tradeoff described in §6.4.2.

6.5.4 Handling concurrent network activities

Tail elimination is performed for each data transfer determined by the application. As described in §6.5.1, an application only ensures that the delay between consecutive transfers is longer than the tail threshold, without considering other applications. However, RRC state transitions are determined by the *aggregated* traffic of all applications. Therefore, allowing every application to send $\boldsymbol{T}$ messages independently causes problems. For example, at time t_1, TOP sends a $\boldsymbol{T}$ message for Application 1 to cut its tail. But a packet transmitted by Application 2 at t_2 will trigger an unnecessary promotion if $t_2 - t_1 < TT$.

Ideally, if all connections can precisely predict the *packet* inter-arrival time, then there exists an optimal algorithm to determine whether to send the $\boldsymbol{T}$ message. The algorithm aggregates prediction across connections by effectively treating all network activities as part of the same connection, so that fast dormancy is triggered only when the combined ITT exceeds TT. At a given time t, let $a_1, ..., a_n$ be the predicted arrival time of the next packet for each connection, then TOP should send a $\boldsymbol{T}$ packet if $\min\{a_i\} - t > TT$.

In practice, however, TOP faces two challenges. First, as mentioned in §6.5.1, applications perform predictions at *transfer* level. Therefore no prediction information is available except for the last packet of a transfer. This may incur additional promotions if, for example, Connection c_1 invokes fast dormancy when Connection c_2 is in the middle of a transfer. Second, legacy applications are unaware of TOP and some applications may not use TOP due to their particular traffic patterns. To handle both issues, we design a simple and robust coordination algorithm described below.

The algorithm considers two cases to determine whether to send a ***T*** message for fast dormancy. First, for all connections *with* ITT prediction information, that is, `TerminateTail` is called but the next packet has not yet arrived, fast dormancy is triggered only when the combined ITT exceeds the tail threshold. Second, we apply a simple heuristic to handle connections *without* ITT being predicted, because either those connections are not at the end of a transfer (`TerminateTail` is not called after transmitting a packet in the middle of a transfer) or they do not use TOP (`TerminateTail` is never called). If such connections exist, a ***T*** message is not sent if any of them has recent packet transmission activity within the past p seconds where p is a predefined parameter, as for an active connection, a recent packet transmission usually indicates another packet will be transmitted in the near future. Not sending a ***T*** message at such a case reduces additional promotions. We set p to α based on our empirical measurement similar to the one described in §6.5.3.

We now describe the coordination algorithm in detail by referring to the pseudo code listed in Figure 6.3. The algorithm maintains three states for each connection. ts and $predict$ correspond to the timestamp of the last observed packet, and the predicted arrival time of the next packet, respectively (Line 2-3). We explain the $dummy$ state shortly. Whenever an incoming or outgoing packet of connection c arrives, $c.ts$ is updated to ts_{cur}, the current timestamp, and $c.predict$ is set to $null$, indicating that no prediction information is currently available for connection c (Line 27-30). At the end of a transfer, after the last packet is transmitted, an application calls `TerminateTail(`c`,` δ`)`. Then TOP updates

```
01 struct CONNECTION {//per-conn. states maintained by TOP
02   TIME_STAMP predict;
03   TIME_STAMP ts;
04   BOOLEAN dummy; //false for any existing connection
05 };
06 TerminateTail(CONNECTION c, ITT δ) {
07   foreach conn in Connections{ //handle out-of-date predictions
08    if (conn.predict < ts_cur) { //ts_cur is the current timestamp
09     if (conn.dummy = true)
10      {Connections.remove(conn);}
11     else {conn.predict ← null;}
12    }
13   }
14   if (c = null) { //create a dummy connection established soon
15    c ← new CONNECTION;
16    c.dummy ← true;
17    Connections.add(c);
18   }
19   c.predict ← ts_cur + δ; //update the prediction
20   foreach c' in Connections{ //check the two constraints
21    if ((c'.predict ≠ null && c'.predict < ts_cur + α)
22     || (c'.predict = null && c'.ts > ts_cur − α))
23    {return;} //fast dormancy is not invoked
24   }
25   send T message;
26 }
27 NewPacketArrival(CONNECTION c) {
28   c.ts = ts_cur;
29   c.predict ← null;
30 }
```

Figure 6.3: The coordination algorithm of TOP

$c.predict$ to $ts_{cur}+\delta$ (Line 19) and sends a $\boldsymbol{T}$ message if both conditions hold (Line 20-25).

$$\min_{c'}\{c'.predict \neq null\} > ts_{cur} + TT \tag{6.1}$$

$$\forall c' : c'.predict = null \rightarrow c'.ts < ts_{cur} - p \tag{6.2}$$

where c' goes over all connections and "$\rightarrow$" denotes implication. Equation (6.1) and (6.2) represent two aforementioned cases where connections are with and without prediction information, respectively. Note that both the tail threshold TT in Equation (6.1) (Line 21) and the p value in Equation (6.2) (Line 22) are both empirically set to α.

Recall that in §6.5.2, when the next transfer starts in a new connection, an application calls `TerminateTail(`$null$`, `δ`)` then `TerminateTail(`c`, `∞`)` at the end of connection c, which is also the end of current transfer. TOP handles the first call by creating a dummy connection c_d (Line 14-18) with $c_d.predict = ts_{cur} + \delta$, and c_d is considered in Equation (6.1). The dummy connection c_d is removed when its prediction is out-of-date *i.e.,* $ts_{cur} > c_d.predict$ (Line 9-10). For an established (*i.e.,* not dummy) connection c, $c.predict$ is set to $null$ (no prediction information) when it is out-of-date (Line 11), and c is removed *i.e.,* not considered by Equation (6.1) or (6.2), when c is closed.

An application may call `TerminateTail(`$null$`, `δ`)` at ts_{cur}, *immediately* after the last packet of connection c is transmitted. However, it is possible that at ts_{cur}, c is not yet removed by TOP although no packet of c will appear. In this case, $c.ts$, the timestamp of the last packet of c, is very close to ts_{cur}, making Equation (6.2) not hold. Thus a $\boldsymbol{T}$ message will never be sent. The problem is addressed by the second call `TerminateTail(`c`, `∞`)` that sets $c.predict = \infty$. Therefore making two calls guarantees that a $\boldsymbol{T}$ message is properly sent even if c is not timely removed.

An application abusing fast dormancy can make a handset send a large amount of $\boldsymbol{T}$ messages, each of which may cause a state demotion to IDLE followed by a promotion triggered by a packet, in a short period. To prevent such a pathological case, TOP sends at

most one $\boldsymbol{T}$ message for every t seconds even if multiple $\boldsymbol{T}$ messages are allowed by the constraints of Equation (6.1) and (6.2)(not shown in the pseudo code). This guarantees that repeatedly calling `TerminateTail` is harmless, and that the frequency of the additional state promotions caused by TOP is no more than one per t seconds. We empirically found that setting t to 6 to 10 seconds has negligible adverse impact on resource savings for normal usage of TOP.

We notice that the major runtime overhead of TOP is to intercept packets and to record their timestamps (Line 27-30). We implemented a kernel module for that task on an Android G2 smartphone. By measuring the additional CPU utilization, we found that the runtime overhead is negligible regardless of the network throughput.

6.6 Evaluation

We use real traces to demonstrate radio resource and energy savings brought by TOP, focusing on evaluating how well TOP handles concurrent network activities. In §6.6.1, we use the passive trace described in §6.3 to study the impact of TOP on a large number of users. In §6.6.2, we perform case studies of two applications using traces locally collected by Tcpdump from an Android G2 phone.

We use ΔD, ΔD^T, ΔE, and ΔS defined in §6.4.2 as evaluation metrics. They are computed using the simulation-based approach described in §6.3. The comparison baseline is the default state machine configuration without using a tail removal technique for the same carrier. For TOP, we set both TT (§6.5.3) and p (§6.5.4) to the α timer value. TOP sends at most one $\boldsymbol{T}$ message for every $t = 10$ seconds.

6.6.1 Evaluation using passive traces

The evaluation is performed at a per-session basis using the calibrated passive trace described in §6.3. For each session, we extract connections (defined by 5-tuples) and classify them into four types by the port number of the TCP peer of a handset, since only TCP head-

ers are available: Web (80, 443, 8080), Email (993, 995, 143, 110, 25), Sync (a popular synchronization service of Carrier 1 using a special port number), and Other (all other port numbers). They contribute to 78.8%, 15.1%, 0.2%, and 5.9% of the total traffic volume, respectively. We use a threshold of 10 sec of idle time to decide that a connection has terminated. Changing this value does not qualitatively affect the simulation results.

For simplicity, we assume that there are four applications, each involving one traffic type, running on smartphones. For Web, Email, and Sync applications, a transfer is defined as consecutive connections of the same traffic type whose inter-connection time (the interval between the last packet of one connection and the first packet of the next connection) is less than 1 sec. Note that a transfer may consist of multiple connections and connections may overlap (*e.g.,* concurrent connections supported by smartphone browsers). At the end of each transfer, each application independently calls `TerminateTail` with probability of z, which quantifies the applicability of TOP, to perform binary predictions (whether ITT is greater than TT, see §6.5.2) with accuracy of w. In other words, the probabilities of a correct prediction, an incorrect prediction, and no prediction (`TerminateTail` is not invoked) are zw, $z(1-w)$, and $1-z$, respectively. Since binary predictions are performed, each application uses an ITT of 60 sec if the ITT is predicted to be greater than TT. Varying this from 30 sec to infinity, or using the exact prediction value of ITT changes the results in Figure 6.4 by no more than 0.01. We assume that the "Other" application is unaware of TOP.

Figure 6.4 plots the impact of TOP on ΔE, ΔS, ΔD^T, and ΔD by varying w and z for Carrier 1. In each plot, the $z = 0$ curve is a horizontal line at $y = 0$ corresponding to the comparison baseline *i.e.,* the default case where TOP or fast dormancy is not used. Figure 6.4 clearly shows that, increasing z, the applicability of TOP, brings more savings at the cost of increasing the state promotion delay. On the other hand, increasing w, the prediction accuracy, not only benefits resource savings but also reduces the state promotion overhead. Under the case where $z = 0.8$ and $w = 90\%$, TOP saves the overall radio

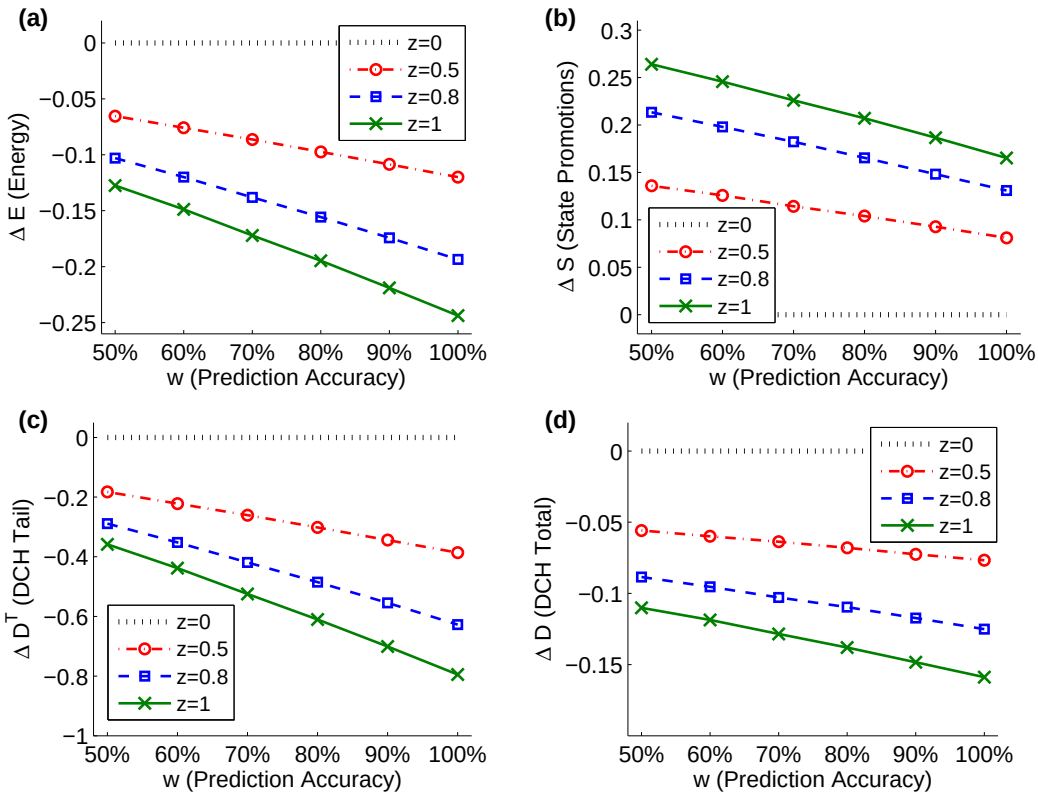


Figure 6.4: Evaluation of TOP using the passive trace from Carrier 1: (a)ΔE (b)ΔS (c)ΔD^T (d)ΔD

energy E, the DCH tail time D^T, and the total DCH time D by 17.4%, 55.5%, and 11.7%, respectively, with the state promotion delay S increasing by 14.8%. The results for Carrier 2 show similar trends. Under the condition of $z = 0.8$ and $w = 90\%$, TOP can save E, D^T, and D by 14.9%, 60.1%, and 14.3%, respectively with S increasing by 9.0%.

We compare TOP with other schemes for saving the tail time. In each plot of Figure 6.5, the X axis is the state promotion delay ΔS, and the Y axis corresponds to saved resources (ΔE, ΔD^T, or ΔD) for Carrier 2. A more downward or leftward curve indicates a better saving scheme since given a fixed ΔS, we prefer a more negative value of ΔE, ΔD^T, or ΔD indicating higher resource savings. Each plot of Figure 6.5 contains four curves. The "TOP" curve corresponds to using TOP with $w = 80\%$ and z being varied from 0.5 to 1.0. The "FD" (fast dormancy) curve is generated using the same parameters, but in the "FD" scheme, applications use fast dormancy without being scheduled by TOP. In other words, an application (Web, Email, or Sync) sends a $\boldsymbol{T}$ message whenever its predicted ITT is greater than TT. The "timer" curve corresponds to a strategy of proportionally decreasing α and β timers that affect all sessions in the trace.

The "TE" curve denotes employing TailEnder [14] to save energy and radio resources. As described in §6.1, for delay-tolerant applications, their data transfers can be delayed and batched to reduce the tail time. TailEnder is a scheduling algorithm that schedules transfers to minimize the energy consumption while meeting user-specified deadlines by delaying transfers and transmitting them together. The TailEnder algorithm was implemented in our simulator using the default parameters described in [14]. We apply TailEnder on all Email and Sync transfers and vary the deadline (the maximally tolerated delay) from 0 to 5 minutes. A longer deadline can potentially save more resources but a user has to wait for longer time.

We discuss the results in Figure 6.5. TOP outperforms fast-dormancy (FD), whose curve lies on the right of the "TOP" curve. To achieve the same savings in D, E, and D^T, the state promotion delay of TOP is always less than that of FD by 10% of the overall pro-

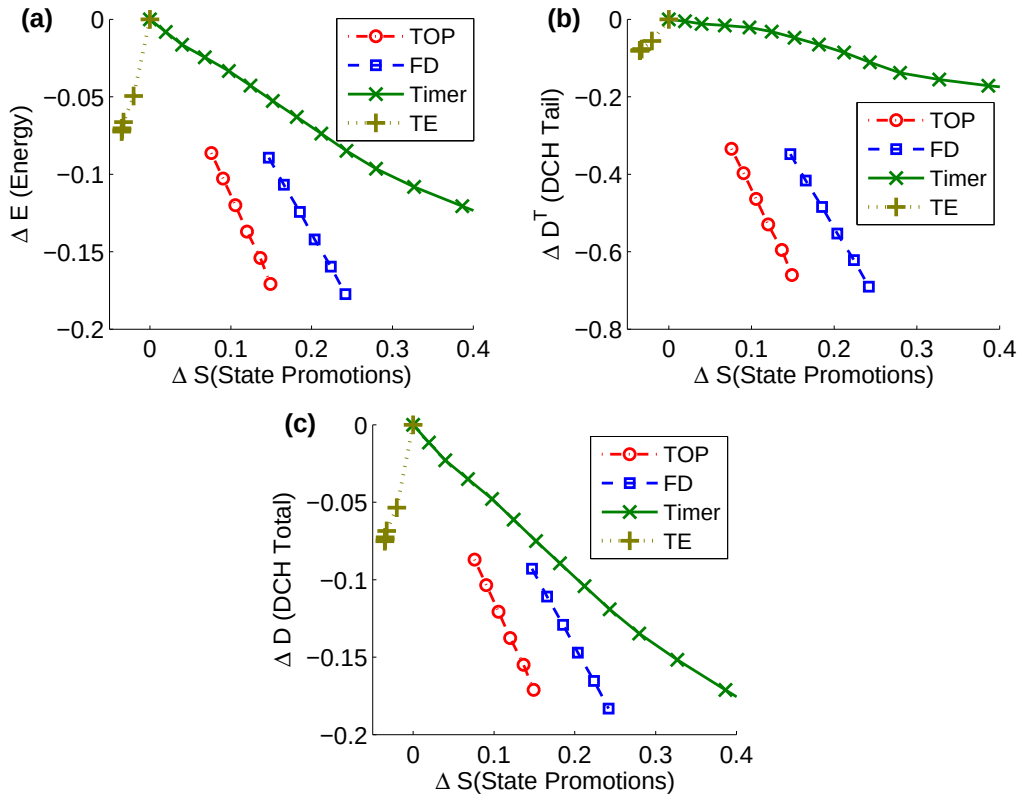


Figure 6.5: Comparison of four schemes of saving tail time for Carrier 2: (a)ΔS vs. ΔE (b)ΔS vs. ΔD^T (c)ΔS vs. ΔD

motion delay in the default scheme. Further, reducing inactivity timers incurs additional state promotions, overwhelming the savings of D and E. The fundamental reason for this is the static nature of the inactivity timer paradigm where all packets experience the same timeout period. We also notice that TailEnder can reduce the overall state promotion delay (as indicated by the negative ΔS values) due to its batching strategy. However, its applicability is very limited, yielding much less savings, and it incurs additional waiting time for users. The comparison results for Carrier 1 is qualitatively similar, implying that invoking fast dormancy with a reasonable prediction accuracy (around 80%) surpasses the traditional approach of tuning inactivity timers in balancing the tradeoff, and TOP's coordination algorithm effectively reduces the state promotion overhead caused by concurrent network activities.

6.6.2 Evaluation using locally collected traces

We perform case studies of two applications (Pandora streaming and Web browsing) using traces locally collected from an Android G2 phone using Carrier 2's UMTS network. We investigate each application separately without injecting concurrent traffic, then apply the coordination algorithm on the aggregated traffic of both applications.

6.6.2.1 Pandora radio streaming

Pandora [30] is an Internet radio application. We collected a 30-min trace using Tcpdump by logging onto one author's Pandora account, selecting a pre-defined radio station, then listening to seven tracks (songs). By analyzing the trace, we found that the Pandora traffic consists of two components: the audio/control traffic and the advertisement traffic. Before a track is over, the content of the next track is transferred in one burst utilizing the maximal bandwidth. Then at the exact moment of switching to the next track, a small traffic burst of control messages is generated. The second component is periodical advertisement traffic from an Amazon EC2 server for every one minute. Each such burst can trigger an

Table 6.1: Impact of TOP on Pandora

ΔD^T	-100%
ΔD	-50.1%
ΔS	3%
ΔE	-59.2%

Table 6.2: Impact of TOP on traces of three websites

Website	m.cnn.com	amazon.com	m.facebook.com
% long ITT	91.2±4.8%	91.4±6.5%	38.1±5.9%
ΔD^T	-90.7%	-88.7%	-36.7%
ΔD	-51.4%	-48.0%	-22.2%
ΔE	-61.1%	-57.4%	-21.6%
ΔS	+8.4%	+9.8%	+51.5%

IDLE→DCH promotion.

We apply TOP on the trace by regarding each data/control burst and each advertisement burst as a transfer. The results in Table 6.1 indicate that TOP achieves remarkably good resource savings for traffic patterns consisting of bursts separated by long timing gaps. TOP eliminates all tails that contribute to about half of the total DCH time with the state promotion delay increasing by 3% regardless of using binary or exact prediction of ITT. The radio energy usage decreases by 59%.

6.6.2.2 Web Browsing

We show the applicability of TOP to Web browsing. As described in §6.5.1, here a transfer can be naturally defined as packets belonging to the same Web page (including embedded objects) downloaded in a burst. Usually a browser can precisely know the termination of a transfer, and the challenging part is to predict inter-transfer times (ITTs) that involve user interactions, as a page download is mostly triggered by clicking a link, and ITTs correspond to a user's reading or thinking time. A comprehensive study of the prediction methodology is beyond the scope of this chapter. Here we describe our preliminary study showing that even very simple heuristics can lead to good prediction results for some popular websites.

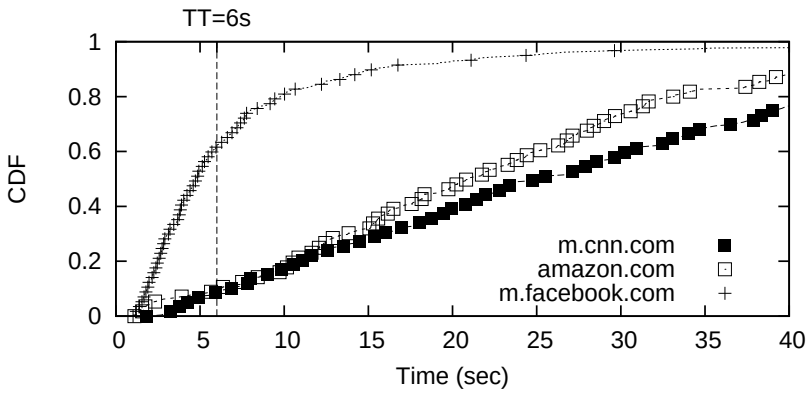


Figure 6.6: CDF of inter-transfer time (ITT) for three websites

We simultaneously collected user input event traces (*e.g.,* tapping the screen) and packet traces from eight users while they visited the three websites listed in Table 6.2. The traces we collected have a total duration of 298 minutes with each between 7 and 20 minutes long. Then we extract individual transfers by examining HTTP headers and by correlating packet traces with user input events. We found that all transfers were triggered by users *i.e.,* a user input is observed within 0.5s (to tolerate the processing delay) before a transfer starts.

Figure 6.6 plots the CDF of ITTs for the three websites. Each curve consists of ITTs of all eight users. We call an ITT whose value is greater than TT, which is 6 sec, a long ITT. Otherwise it is a short ITT. Ideally a $\boldsymbol{T}$ message should only be sent for a long ITT. Figure 6.6 clearly indicates the wide disparity of traffic patterns among websites due to their different contents. For `cnn` and `amazon`, 92% and 91% of ITTs are long. In contrast, `facebook` has much less long ITTs (only 38%). The second row of Table 6.2 shows the average ratio of long ITTs for each user. The small standard deviations indicate that the long-ITT ratio is relatively stable among the eight users.

Figure 6.6 suggests that a browser can adopt a simple approach where for websites with historically observed high long-ITT ratios, the browser always predicts an ITT to be long by setting ITT to a large value (we use 60 sec, assuming browsers cannot predict the exact values of ITT and do only binary predictions). The simulation results are shown in Table 6.2. For `cnn` and `amazon`, TOP saves about half of the total DCH time and 60% of

Table 6.3: Impact of TOP on Mixed Traces (Pandora and CNN)

Pandora+CNN	ΔD^T	ΔD	ΔS	ΔE
Default	0	0	0	0
TOP	-88±3%	-41±2%	+17±3%	-48±2%
FD	-100±0%	-50±2%	+66±9%	-58±2%

the radio energy with less than 10% of increase on the state promotion overhead. However, for `facebook`, the savings are much less and the promotion overhead becomes high due to its small long-ITT ratio. The browser can thus use a fixed long-ITT ratio threshold for deciding whether or not to apply TOP for each website.

6.6.2.3 Evaluation of mixed traces

To evaluate the coordination algorithm, for `cnn` and `amazon`, we concatenate traces of 2 to 4 randomly selected users, then mix the concatenated trace (roughly 30 min) with the 30-min Pandora trace. We assume that three application components concurrently use TOP: Pandora audio (exact prediction of ITT), Pandora advertisement (exact prediction), and the browser application (always predict ITT to be long). For each website, we generate 100 mixed traces and feed each of them into the simulator for the three schemes listed in Table 6.3: "Default" (the comparison baseline where TOP or fast dormancy is not used), "TOP" (using TOP), and "FD" (only using fast dormancy). Table 6.3 (`Pandora` + `cnn`) clearly shows the benefits of TOP, which significantly decreases ΔS from 66% to 17% by reasonably sacrificing savings of other three dimensions. The results for `Pandora` + `amazon` are very similar.

6.7 Summary

By leveraging fast dormancy, TOP enables applications to actively inform the network of a tail that can be eliminated via a simple interface. Our design of TOP enables significantly better radio resource usage and substantial energy savings for cellular networks.

More importantly, our work opens new research opportunities for designing effective tail prediction algorithms for smartphone applications (especially for applications involving user interactions), which is the major part of our ongoing work. In addition, we are seeking ways to build a real implementation of TOP on Android phones.

CHAPTER VII

Web Caching on Smartphones: Ideal vs. Reality

We have studied several complementary approaches for reducing the resource consumption in cellular networks: tuning the RRC state machine parameters (Chapter IV), changing application traffic patterns (Chapter V), and releasing radio resources in an adaptive manner (Chapter VI). Another approach not explored by us is simply to reduce the bandwidth consumption by making applications transfer less data.

In fact, mobile data traffic is experiencing unprecedented growth. Cisco predicted that from 2011 to 2016, global mobile data traffic will increase by 1800%, reaching 10.8 exabytes per month. This prediction also projects the growth of smartphone traffic to be 5000% greater in 2016 than it is now [54], far exceeding the deployment of cellular infrastructures: in 2011, the cellular infrastructure spending was expected to be only a 6.7% rise over 2010 [2]. From customers' perspective, as most carriers have moved to a volume-based charging model, reducing the bandwidth consumption effectively cuts cellular bills.

To achieve the goal of bandwidth reduction, network Redundancy Elimination (RE) plays a crucial role by preventing duplicate data transfers and making the transferred data more compact [55]. There exist a wide range of RE techniques such as caching and compression. In this chapter, we investigate a particular RE technique of *web caching on smartphones*, due to *(i)* the dominance of HTTP in today's smartphone traffic, *(ii)* the popularity and the potential benefits of caching, and *(iii)* the complexity of the caching protocol.

7.1 Introduction

Anand:PowerManage:2005 Caching plays a vital role in HTTP that dominates the Internet traffic usage [56]. Web caching can effectively decrease network traffic volume, lessen server workload, and reduce the latency perceived by end users.

Web caching in cellular networks is even more critical. HTTP traffic generated by mobile browsers and smartphone applications far exceeds any other type of traffic. HTTP accounts for 82% of the overall downstream traffic based on a recent study of a large cellular ISP [1]. From network carriers' perspective, cellular networks operate under severe resource constraints [5] due to the explosive growth of cellular data traffic [54]. Therefore even a small reduction of the total traffic volume by 1% leads to savings of tens of millions of dollars for carriers which are expected to spend $40.3 billion on cellular infrastructures in 2011 [2] (for all U.S. carriers).

The benefits of caching are also significant from customers' perspective. *(i)* Fewer network data transfers cut cellular bills since most carriers impose a monthly limit on data plan usage. *(ii)* The user experience improvement brought by caching is more notable in cellular networks whose latency is usually higher than those in Wi-Fi and wired networks. *(iii)* Transferring less data also improves handset battery life, as the power consumed by the cellular radio interface contributes 1/3 to 1/2 of the total device power during the normal workload [6].

The location of a cache can be at one or more places on a path from a handset (*i.e.,* a mobile device) to a web server. Although most of prior works focus on caching proxies placed in the network (*e.g.,* hierarchical caching proxies [57], caching proxy placement [56], and caching within 3G networks [1]), we emphasize that in cellular networks, *caching on handsets* is particularly important because it does not incur any network-related overhead. In particular, it eliminates data transmitted over the last-mile, *i.e.,* the radio access network, which is known as the performance and resource bottleneck of a cellular network [5]. In contrast, a network cache does not bring the aforementioned benefits but

can cut the wide-area latency and enable sharing across users. In this study we focus on handset caches, which can coexist with network caches.

The rules of web caching are defined in HTTP 1.1 protocol (RFC 2616 [49]), although there exist numerous caching proposals [58, 59, 60, 61] that were never widely deployed. As specified in RFC 2616, HTTP employs *expiration* and *revalidation* to ensure cache consistency: the server sets for each cacheable file an expiration time before which the client should safely assume the freshness of the cached file. After the cache entry expires, the client must send a small revalidation message to the server to query the freshness of the cache entry. We detail the procedure in §7.2.

Redundant network transfers due to cache implementation. Let an *ideal* handset cache be a cache that has unlimited size and strictly follows the aforementioned caching rules. Any practically implemented handset cache is not ideal due to one or more reasons below: *(i)* it has limited size, *(ii)* the implemented caching logic does not satisfy RFC 2616, and *(iii)* the cache is not persistent (*i.e.,* it does not survive a process restart or a device reboot). A non-ideal cache potentially incurs *redundant transfers* that do not occur if an ideal cache were used. Also, redundant transfers can be caused by developers not utilizing the caching support even if it is provided by the HTTP library.

In this chapter, we present to our knowledge the first network-wide study of the redundant transfers (defined above) by investigating the following important characteristics:

- The prevalence of redundant transfers within today's smartphone traffic, in terms of both the traffic volume contribution and the resource impact;
- The root cause for the redundant transfers;
- The handset caching logic, identified to be the main reason of redundant transfers, of HTTP libraries and browsers on popular smartphone systems.

We summarize our main findings as follows.

- We leveraged a large dataset containing 695M HTTP transaction records collected from 2.9M customers of a commercial cellular network in the U.S. To complement this data set which has a short duration of 24 hours, we collected another five-month trace from 20 smartphone users using smartphones instrumented by us. We measured the prevalence of redundant transfers for both datasets using an accurate caching simulation algorithm, assuming an ideal cache. To our surprise, redundant transfers due to caching implementation issues contribute 18% and 20% of the total HTTP traffic volume, for both datasets, respectively. The fraction slightly decreases to 17% at the scope of *all* traffic for the user study trace. Almost all redundant transfers are caused by the handsets instead of the server. Further, the redundancy ratio varies among applications. Some popular smartphone apps have unacceptably high fractions (93% to 100%) of redundant transfers.

- By strategically changing the simulation algorithm and analyzing the cache access patterns, we found the impact of limited cache size and non-persistent cache on the redundancy to be limited. For example, the fraction of redundant bytes is at least 13% (compared to 18% for an ideal cache) within all HTTP traffic even when our simulation uses a small cache of 4 MB. This implies the problematic caching logic is the main reason for redundant transfers.

- We investigated the resource overhead (handset radio energy, radio resources, and signaling load) incurred by redundant transfers. The results indicate that the impact on resource consumption is smaller than their impact on the traffic volume (yet still significant, *i.e.,* 7% for radio energy, 9% for radio resources, and 6% for signaling load, based on our analysis of the user study trace), due to reasons explained in §7.5.

- We performed comprehensive tests of ten state-of-the-art HTTP libraries and mobile browsers, many of which were found to not fully support or strictly follow the HTTP 1.1 caching specifications. In particular, among the eight HTTP libraries, four (three

for Android and one for iOS) do not support caching at all. Smartphone apps using these libraries thus cannot benefit from caching. By exposing the shortcomings of existing implementations, our work helps encourage library and platform developers to improve the state of the art, and helps application developers choose the right libraries to use for better performance.

Overall, our findings suggest that for web caching, *there exists a huge gap between the protocol specification and the protocol implementation on today's mobile devices*, leading to significant amount of redundant network traffic, most of which could be eliminated if the caching logic of HTTP libraries and browsers fully supports and strictly follows the specification, and developers fully utilize the caching feature provided by the libraries. Fixing this cache implementation issue can bring considerable reduction of network traffic volume, cellular resource consumption, handset energy consumption, and user-perceived latency, benefiting both cellular carriers and customers.

In the rest of this chapter, §7.2 provides background of HTTP caching. §7.3 describes the measurement goal, data, and methodology. Then we measure the traffic volume impact and the resource impact of redundant transfers in §7.4 and §7.5, respectively. In §7.6 we perform caching tests for popular HTTP libraries and mobile browsers. We discuss the future work and conclude the chapter in §7.7.

7.2 Background: Caching in HTTP

This section provides background of web caching defined in HTTP 1.1 [49]. As described in §7.1, our study focuses on caching on handsets instead of within the network. In the remainder of the chapter, a *cache* refers to an HTTP cache on a handset unless otherwise specified. We refer to a web object (*e.g.,* an HTML document or an image) carried by an HTTP response as a *file*.

Caching consistency (*i.e.,* keeping cached copies fresh) is the key issue. To realize

that, the server sets the expiration time for each file by specifying either **`Expires`** or **`Cache-Control:max-age`** header directive. Then before a cache entry expires, a handset should safely assume the freshness of the file by serving the request using the cached copy without generating any network traffic. After a cache entry expires, a handset should perform cache *revalidation*, *i.e.,* asking the origin server whether the file has changed, by issuing conditional requests using **`If-Modified-Since:<time>`** or **`If-None-Match:<eTag>`** directive (or both). The former directive instructs the server to send a new copy if the file has changed since a specified date, which is usually the last modified time indicated by the **`Last-Modified`** directive in the previous response. The latter allows the server to determine the freshness using a file "version identifier" called eTag (entity tag), which is a quoted string attached to the file in the previous response. An eTag might be implemented using a version name or a checksum of the file content. In either case, if the file has changed, the server sends a new copy to the handset. Otherwise, a small **`304 Not Modified`** response is returned to the handset, without a document body, for efficiency.

A handset determines a cache entry has expired if and only if $t_{\text{arrive\_age}} + t_{\text{cache\_age}} \geq t_{\text{fresh\_life}}$. $t_{\text{arrive\_age}}$ is the age of the file when it arrives at the cache. It is computed from the **`Date`** or **`Age`** directive of a response, plus an estimated round-trip time compensating for the network delay. $t_{\text{cache\_age}}$, which can be trivially calculated, indicates how long the file has been in the cache. $t_{\text{fresh\_life}}$ is the cached copy's freshness lifetime (similar to the shelf life of food in grocery stores) derived from the **`Expires`** or **`Cache-Control:max-age`** directive where the latter one overrides the former one if both exist. $t_{\text{fresh\_life}}$ is usually specified by the server while a handset can also specify **`Cache-Control`** request directives (**`max-age`**, **`max-stale`**, and **`min-fresh`**) to tighten or loosen expiration constraints although they are rarely used in practice.

Non-storable and must-revalidate files. A *non-storable* file (marked by **`Cache-Control:no-store`**) forbids a cache from storing (*i.e.,* caching) the file. A *must-revalidate* file (marked by **`Cache-Control:must-revalidate`**, **`Cache-Control:no-cache`**, or

`Pragma:no-cache`) can be stored in a cache. However, the cache must bypass the freshness calculation mechanism and always revalidate with the origin server before serving it ("**`no-cache`**" is misleading because the file actually can be cached). A handset can also explicitly set the above caching directives in a request, indicating it will not store or will always revalidate the file.

Clearly, well-behaved caching logic requires correct implementation at both the handset and the server side. Our analysis described in §7.3 helps identify inefficient caching behaviors and which side is responsible for any of them.

7.3 Measurement Goal, Data, and Methodology

This section highlights our measurement goal (§7.3.1), describes the measurement data (§7.3.2), and then details our analysis approach for redundant data transfers (§7.3.3).

7.3.1 The Measurement Goal

We define *inefficient caching* as caching behaviors that *(i)* lead to redundant data transfers, whose negative impact on performance, resource consumption, and billing is particularly high for mobile users, and *(ii)* relate to the *implementation* (as opposed to the *semantics*, as described below) of the HTTP caching mechanism. Inefficient caching can be caused by multiple reasons including the following factors covering the most important aspects of cache implementation (we discuss less significant factors in §7.3.3.2).

- **Problematic caching logic** due to any of the following reasons. *(i)* The handset does not fully support or strictly follow the protocol specification[1]. *(ii)* The server does not properly follow the caching specification. *(iii)* In order to leverage the caching support provided by an HTTP library, a developer still needs to configure it. Many

[1]Not following the specification can also cause consumption of expired contents. But this was never observed in our tests in §7.6.

developers may skip that for simplicity, or simply be unaware of it, therefore missing the opportunity of caching even if it is supported.

- **A limited cache size** causing the same content to be fetched twice if the first cached copy is evicted from the cache.

- **A *non-persistent* cache** whose cached data does not survive a process restart or a device reboot (unlike a *persistent* cache that survives both).

As highlighted in §7.1, we aim at understanding the impact of redundant transfers caused by the above inefficient caching factors, for commercial cellular networks (§7.4 and §7.5), as well as how the caching logic of popular HTTP libraries and mobile browsers deviates from the specification (§7.6).

Caching implementation vs. semantics. All the factors listed above relate to caching *implementation*. Redundant transfers may also be attributed to the misconfiguration of caching parameters, which ideally should be properly set by a server according to the *semantics* of files. For example, for a news website, conservatively marking all news articles and images as non-storable or setting for them very short freshness lifetime values may lead to redundant transfers while bringing negligible benefits given that the article contents rarely change. A thorough study of the file semantics and caching parameter settings is our on-going work, although we show in this chapter examples where caching parameter settings are obviously too conservative (§7.7).

From this point on, unless otherwise specified, *redundant transfers* refer to redundant transfers caused by caching implementation.

7.3.2 The Smartphone Measurement Data

We collected two diverse datasets summarized in Table 7.1 to measure redundant transfers caused by inefficient caching.

Table 7.1: Our measurement datasets.

Dataset	ISP	UMICH
Data collection period	May 20 2011 0:00 GMT ∼ May 20 2011 23:59 GMT	May 12 2011 ∼ Oct 12 2011
Collection point	Commercial cellular core network	Directly on users' handsets
Number of users	2.92 million (estimated)	20
Dataset size	271 GB	119 GB
Traffic volume	24.3 TB	118 GB
Platforms	Multiple (mainly iOS and Android)	Android 2.2
Data format	695 million records of HTTP transactions	Full packet trace (including payload) of all traffic

7.3.2.1 The ISP Dataset

The ISP dataset was collected from a large U.S. based cellular carrier (Carrier 1, see §2.1.3) at a national data center on May 20, 2011, on the interface between the GGSN (Gateway GPRS Support Node) and SGSNs (Serving GPRS Support Node) without any sampling. Each record in the dataset corresponds to one HTTP transaction, containing three pieces of information: *(i)* a 64-bit timestamp, *(ii)* summaries of header fields in the request and the response, and *(iii)* the actual amount of data transferred based on the TCP data associated to each HTTP transaction. To preserve subscribers' privacy, the URLs were anonymized using a 128-bit hash function and also all cookie information was removed from the HTTP headers[2].

Subscriber identification. Since our cache simulation is performed at a per-user basis (§7.3.3), we need to identify the subscriber ID for each record. Instead of using MSISDN (the phone number) or IMEI (the device ID) information that was not collected due to privacy concern, we used anonymized session-level information to correlate multiple HTTP transaction records with a single subscriber. One disadvantage of our approach is that one real subscriber may be identified with multiple subscriber IDs (but one subscriber ID never maps to multiple real subscribers). This leads to an underestimation of the amount of redundant data due to increased cold start cache misses [63] (explained in §7.3.3.2).

[2]In HTTP, caching and cookies are decoupled, and a server is responsible for explicitly disabling caching when appropriate [62].

7.3.2.2 The UMICH Dataset

The ISP dataset is representative due to its large user base, however it is limited in terms of the trace duration and recorded content, as only summarized HTTP transaction records were captured. This is complemented by our second dataset called UMICH, collected from 20 smartphone users for five months, allowing us to keep detailed track of each individual user's web cache for a much longer period. These participants consisted of students from 8 departments at University of Michigan[3]. The 20 participants were given Motorola Atrix (11 of them) or Samsung Galaxy S smartphones (9 of them) with unlimited voice, text and data plans of the same cellular carrier from which we obtained the ISP dataset. All smartphones use Android 2.2. The participants were encouraged to take advantage of all the features and services of the phones. We kept collected data and users' identities strictly confidential.

We developed custom data collection software and deployed it on the 20 smartphones. It continuously runs in the background and collects two types of data: *(i)* full packet traces in `tcpdump` format including both headers and payload, and *(ii)* the process name responsible for sending or receiving each packet, using the method described in [6] by efficiently correlating the socket, the inode, and the process ID in Android OS in realtime. Both cellular and Wi-Fi traces were collected without any sampling performed. The data collector incurs no more than 15% of CPU overhead although the overhead is much lower when the throughput is low (*e.g.,* less than 200 kbps).

We also built a data uploader that uploads the captured data (stored on the SD card) to our server when the phone is idle. The data collection is paused when the data is being uploaded so the uploading traffic is not recorded. Also the upload is suspended (and the data collection is resumed) by any detected network activity of user applications. The entire data collection and uploading process is transparent to the users, although we do advise the users to keep their phones powered on as often as possible.

[3]This user study has been approved by the University of Michigan IRB-HSBS #HUM00044666.

7.3.3 Analyzing Redundant Transfers

We explain our data analysis approach. We feed each user's HTTP transactions in the order of their arrival time to a web cache simulator developed by us. The simulator behaves like a cache that strictly follows the HTTP 1.1 caching mechanism specified in RFC 2616 (§7.2). Redundant transfers can be identified through the simulation process. Our approach differs from previous trace-driven cache simulations [58, 63, 56, 1] in two ways. *(i)* Our simulation is performed at a *per-user* basis to capture redundant transfers for each handset, while previous ones consider aggregated HTTP transfers of all users to compute, for example, the cache hit ratio, for a network cache. *(ii)* Ours is more fine-grained in that it distinguishes various causes of the redundancy (and also non-redundancy).

In the remainder of the chapter, a *handset cache* and the *simulated cache* refer to the cache on a real handset and the cache maintained by our simulator, respectively.

The basic simulation algorithm is illustrated in Figure 7.1. It assigns to each HTTP transaction a label indicating its caching status. A file can be **`NOT_STORABLE`** due to its **`Cache-Control:no-store`** directive or causes **`CACHE_ENTRY_NOT_EXIST`** because it has not yet been cached. **`NOT_EXPIRED_DUP`** is an undesired case where a handset issues a request for a file cached in the simulator *before* it expires, resulting in redundant transfers ("**`DUP`**" means duplication).

Then Line 10 to 17 deal with the scenario where a cached file has expired. If the file has changed, then the entire file needs to be transferred again (**`FILE_CHANGED`**). If the file remains unchanged, the ideal way to handle it is that the handset performs cache revalidation and the server sends back an **`HTTP_304`** response. Otherwise, the problem either comes from the handset side, which does not issue a conditional request (**`EXPIRED_DUP`**), or from the server side, which does not recognize a conditional request (**`EXPIRED_DUP_SVR`**). Among the aforementioned labels, **`NOT_EXPIRED_DUP`**, **`EXPIRED_DUP`** and **`EXPIRED_DUP_SVR`** correspond to redundant transfers.

Is our simulated cache complete? Let us first assume our simulated cache is persistent

```
01 foreach HTTP transaction r
02    if (file is not storable) then
          //the file contains "Cache-Control: no-store"
03        assign_label(r, NOT_STORABLE);
04        continue;
05    else if (cache entry not exists) then
          //cache entry not found
06        assign_label(r, CACHE_ENTRY_NOT_EXIST);
07    else if (cache entry not expired) then
          //a request is issued before the file expires
08        assign_label(r, NOT_EXPIRED_DUP);
          //the response is ignored by the simulator because the
          //request should not have been generated
09        continue;
10    else if (file changed) then
          //the file has changed after the cache entry expires
11        assign_label(r, FILE_CHANGED);
12    else if (HTTP 304 used) then
          //the file has not changed after the cache entry expires,
          //and a cache revalidation is properly performed
13        assign_label(r, HTTP_304);
14    else if (revalidation not performed) then
          //the file has not changed after the cache entry expires,
          //but the handset does not perform cache revalidation
15        assign_label(r, EXPIRED_DUP);
16    else
          //the file has not changed after the cache entry expires,
          //but the server does not recognize the cache revalidation
17        assign_label(r, EXPIRED_DUP_SVR);
18    update_cache_entry(r);//update the simulated cache
19 endfor
```

Figure 7.1: The basic caching simulation algorithm.

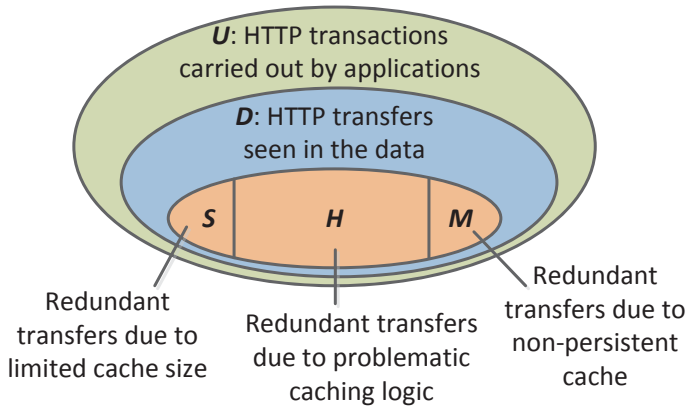


Figure 7.2: A Venn diagram showing HTTP transactions observed by applications and by our simulator, as well as the redundant transfers.

with unlimited size (*i.e.,* an ideal cache defined in §7.1). Consider the Venn diagram shown in Figure 7.2. Let U be all HTTP transactions carried out by applications. What we observe in the data, D, is a subset of U since requests of $U \setminus D$ (the relative complement of D in U) are already served by the handset cache so $U \setminus D$ does not appear on the network. However, remember that in order for a file f to be cached, it must be transferred over the network (*i.e.,* $f \in D$) at least once. Therefore our simulated cache will not miss any file that is in a handset cache, if the simulated cache is ideal (we discuss the only exception in §7.3.3.2). More importantly, as explained in §7.3.1, our goal is to study the redundant data transfers that always belong to D instead of $U \setminus D$.

On the other hand, a file in our simulated cache may be missing in a handset cache because of its problematic caching logic (the set H in Figure 7.2), the limited size (the set S), or a lack of persistent storage (the set M) of the handset cache. Ideally we want to distinguish the three cases. However, the redundant transfers identified by our simulator are in fact $H \cup S \cup M$ where H, S, and M are indistinguishable, given that the simulated cache is ideal.

7.3.3.1 Algorithm Details

Figure 7.1 sketches the basic simulation algorithm. We provide details of the algorithm below.

The key of a cache entry (our simulated cache was implemented using a hash map) consists of three parts: *(i)* the host name indicated by the **`Host`** request directive, *(ii)* the file name followed by the **`GET`** command, and *(iii)* the eTag. Part *(i)* and *(ii)* must exist otherwise the HTTP transaction is assigned a special label "**`OTHER`**" (Table 7.3) while the eTag part is optional. Also the file name includes the entire **`GET`** string containing query strings so **`/a.php?para=1`** and **`/a.php?para=2`** have different cache entries. Empty or error responses (*e.g.,* **`404 Not Found`**) are also counted as **`OTHER`**, which only accounts for 0.5% of all HTTP traffic in both datasets.

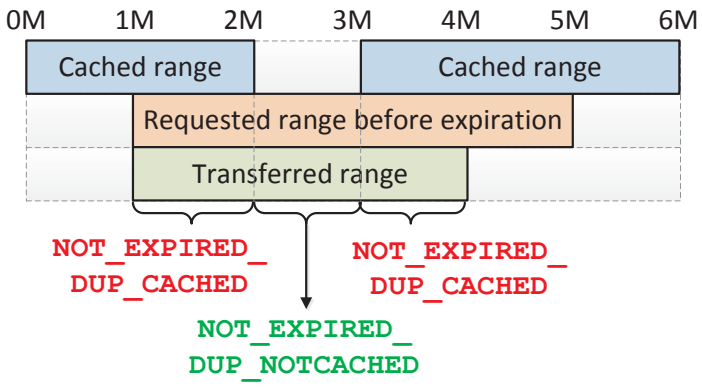


Figure 7.3: A partially cached file and a partial cache hit.

The change of a file is identified through a different **`Last-Modified`**, **`Content-Length`** or **`Content-MD5`** value for the same cache entry.

A heuristic freshness lifetime can be used when neither **`Cache-Control:max-age`** nor **`Expires`** exists in a response, according to RFC 2616. We use a heuristic lifetime of 24 hours and later we show our analysis results are not sensitive to this value (§7.4.1).

A partially cached file is caused either by a byte-range request (using the **`Range`** directive) for a subrange of the origin file, or by a prematurely broken connection. Our simulator supports partial caching by allowing a cache entry to contain one or more subranges of a file. We implemented the following logic according to RFC 2616. Assume one or more subranges of a file have been cached, and an incoming response transfers another subrange R. The new subrange R is then combined with the existing range(s) if both the existing and the new range have the same eTag value (the eTag value must exist). Otherwise, all previously cached range(s) are removed before R is put into the cache entry.

If a file is partially cached, then a single transfer of the whole or a part of the file may contain both redundant and non-redundant bytes. To handle such a case, for each of the **`*EXPIRED_DUP*`** labels, we use two new labels, **`*EXPIRED_DUP*_CACHED`**, and **`*EXPIRED_DUP*_UNCACHED`** (not shown in Figure 7.1), to distinguish the redundant and the non-redundant ranges, respectively. Consider a file of 6 MB shown in Figure 7.3. Assume two ranges [0, 2M) and [3M, 6M) are already cached by the simulator. The handset

makes a byte-range request of [1M, 5M) before the cache entry expires. However the user cancels the transfer in the middle so only [1M, 4M) is actually transferred, as observed in our dataset. In this example, ideally the HTTP library should only request for [2M, 3M), which is not in the cache, using the **`Range`** and the **`If-Range`** directives[4]. We therefore label [2M, 3M) as **`NOT_EXPIRED_DUP_NOTCACHED`**, the non-redundant range, and label [1M, 2M) and [3M, 4M) as **`NOT_EXPIRED_DUP_CACHED`**, the redundant ranges. The two labels substitute for the original **`NOT_EXPIRED_DUP`** label which is not used for partially cached files. We introduce similar labels for **`EXPIRED_DUP_SVR`** and **`EXPIRED_DUP`**. Table 7.3 summarizes all labels.

7.3.3.2 Limitations

We discuss five limitations of our simulation approach.

- Inefficient caching and hence redundant transfers exist in both unencrypted HTTP and encrypted HTTPS traffic. However, the ISP trace contains only HTTP records. For the UMICH trace, the simulator cannot parse the HTTPS traffic that was collected by **`tcpdump`** running below the SSL library. HTTPS accounts for only 11.2% of the total traffic volume, compared to 85.4% for HTTP transfers.

- As mentioned before, the simulator cannot precisely distinguish S, H, and M shown in Figure 7.2. We qualitatively address such indistinguishability in §7.4.2 based on robust heuristics.

- A file may already be cached in a handset cache before the data collection started but the simulator does not know that. As an inherent problem (called cold start cache miss [63]) of any trace-driven cache simulation algorithm, it leads to an underestimation of the cache hit ratio (or the redundancy ratio in our case). But both our traces (in

[4]The **`Range`** directive is often used together with an **`If-Range:<eTag>`** conditional request. It means "if the file is unchanged, send me the range that I am missing; otherwise, send me the entire new file".

Table 7.2: Statistics of file cacheability.

Count by	Dataset	Normally Cacheable	Must-revalidate	Non-storable	Other
Bytes	ISP	69.8%	14.3%	15.4%	0.5%
	UMICH	78.2%	1.6%	19.7%	0.5%
Files	ISP	72.4%	12.4%	14.9%	0.4%
	UMICH	65.6%	6.8%	25.4%	2.1%

particular the UMICH dataset) are sufficiently long so the impact of cold start cache miss is expected to be small.

- Redundant transfers can also be caused by users explicitly reloading a file (*e.g.,* refreshing a web page). In that case, the application may override the default caching behavior by, for example, requesting for a file before its cached copy expires, but the simulator has no way to identify such manually triggered redundant transfers.

- Similarly, our simulator cannot identify redundant transfers caused by users manually clearing the cache after browsing sessions. The amount of such redundant data is expected to be small due to the observed strong temporal locality of accessing the same cache entry (described in §7.4.2).

7.4 The Traffic Volume Impact

In this section, we investigate the *traffic volume* impact of redundant transfers caused by inefficient caching behaviors.

7.4.1 Basic Characterization

We first assume our simulated cache is ideal (*i.e.,* it is persistent with unlimited cache size). Thus all redundant transfers caused by the three factors described in §7.3.1 can be identified.

File cacheability. Table 7.2 breaks down all HTTP bytes (files) transferred over the net-

Table 7.3: Detailed breakdown of caching entry status.

Label	Cache hit or miss?	Fully or partially cached?	Redundant?	% HTTP bytes		
				ISP	UMICH	
				(GC)*	(GC)	(PC)*
1. **NOT_STORABLE**	-	-	-	15.4%	19.7%	19.7%
2. **CACHE_ENTRY_NOT_EXIST**	Miss	-	-	47.5%	42.0%	42.3%
3. **FILE_CHANGED**	Miss	-	-	1.9%	0.5%	0.5%
4. **HTTP_304**	Hit	Either	-	0.1%	0.0%	0.0%
5. **NOT_EXPIRED_DUP**	Hit	Full	Yes	13.6%	15.0%	14.7%
6. **NOT_EXPIRED_DUP_CACHED**	Hit	Partial	Yes	2.3%	1.3%	1.3%
7. **NOT_EXPIRED_DUP_NOTCACHED**	Miss	Partial	-	16.0%	17.0%	17.0%
8. **EXPIRED_DUP**	Hit	Full	Yes	1.7%	4.0%	4.0%
9. **EXPIRED_DUP_CACHED**	Hit	Partial	Yes	0.1%	0.0%	0.0%
10. **EXPIRED_DUP_NOTCACHED**	Miss	Partial	-	0.9%	0.0%	0.0%
11. **EXPIRED_DUP_SVR**	Hit	Full	-	0.0%	0.0%	0.0%
12. **EXPIRED_DUP_SVR_CACHED**	Hit	Partial	-	0.0%	0.0%	0.0%
13. **EXPIRED_DUP_SVR_NOTCACHED**	Miss	Partial	-	0.0%	0.0%	0.0%
14. **OTHER**	-	-	-	0.5%	0.5%	0.5%

* GC: all processes on a handset share a global cache; PC: each process has its own cache.

work into four categories: normally cacheable (*i.e.,* following the standard expiration and freshness calculation mechanism), must-revalidate (§7.2), non-storable, and other HTTP transfers (§7.3.3.1). For both datasets, most bytes (70% to 78%) and most files (66% to 72%) are normally cacheable, indicating the potential benefits of caching if handled properly by a handset.

A detailed breakdown of caching entry status is shown in Table 7.3, which lists the 14 labels described in §7.3.3. We show two simulation scenarios for the UMICH dataset: *(i)* all processes on a handset share one single global cache, and *(ii)* each process has its own cache. They correspond to "GC" and "PC" in Table 7.3, respectively. The per-process cache simulation is feasible because the UMICH trace contains packet-process correspondence for each packet. We summarize our findings as follows.

- **NOT_EXPIRED_DUP** contributes most bytes (77% for ISP and 74% for UMICH) among the four labels (5, 6, 8, 9) incurring redundant transfers. In other words, redundant transfers are usually caused by a handset issuing unnecessary requests *before* re-

ceived files expire. For the ISP (UMICH) trace, 14% (31%) of all HTTP transactions (not shown), corresponding to 14% (15%) of all HTTP bytes (Row 5 in Table 7.3), are unnecessary because if handsets properly cache previous responses, no request needs to be sent out over the network and the responses can be served from local caches. On the other hand, the contribution of **EXPIRED_DUP** is much less. For the ISP (UMICH) trace, for 4.5% (10.2%) of all HTTP transactions (not shown), conditional requests to check the freshness of the cached data need to be sent, so that their responses, corresponding to 1.7% (4.0%) of all HTTP bytes (Row 8 in Table 7.3) will end up being served from local caches (if handsets properly cache previous responses) because they do not change.

- When **HTTP_304** is not used, it is almost always attributed to the handset instead of the server which properly handles cache revalidation, as indicated by the negligible contribution of **EXPIRED_DUP_SVR***.

- Partially cached files incur limited redundant transfers in that the traffic volume contribution of ***_DUP_CACHED** is small. In contrast, ***_DUP_NOTCACHED** (illustrated in Figure 7.3) account for considerable amount of non-redundant bytes. We found most (95%)[5] of ***_DUP_NOTCACHED** bytes originate from servers using byte-range responses for streaming large multimedia files. Note that a recent measurement study [64] showed that 98% of multimedia streaming traffic for a commercial cellular network is delivered over HTTP.

- The results of UMICH (GC) and UMICH (PC) are almost identical, indicating small overlap among files requested by different applications.

- Cellular and Wi-Fi traffic exhibit similar breakdown (not shown in Table 7.3), indicating the caching strategies on both the server and the handset side are independent of the network interface.

[5]Identified by their **User-Agent** strings. See §7.4.3 for details.

Table 7.4: The overall traffic volume impact of redundant transfers when different heuristic freshness lifetime values are used.

Dataset	% of redundant bytes of all HTTP traffic [% of redundant bytes of all traffic (HTTP and non-HTTP)] under different values of heuristic freshness lifetime				
	1 hour	6 hours	1 day*	1 week	1 month
ISP (GC)	17.74% [16%]**	17.74% [16%]	17.74% [16%]	17.74% [16%]	17.74% [16%]
UMICH (GC)	20.24% [17.29%]	20.25% [17.30%]	20.28% [17.33%]	20.32% [17.36%]	20.34% [17.38%]
UMICH (PC)	19.94% [17.03%]	19.95% [17.04%]	19.98% [17.07%]	20.02% [17.10%]	20.04% [17.12%]

* 1 day is the heuristic freshness lifetime used for other results in this chapter.
** Assume the fraction of HTTP traffic is 90%, based on a recent large-scale measurement study for a commercial cellular data network [65].

The overall traffic volume impact is summarized in Table 7.4. We highlight key observations below.

- The first number in each grid of Table 7.4 is the fraction of redundant bytes within all HTTP traffic. Recall the redundant bytes come from label 5, 6, 8, 9 in Table 7.3, and they correspond to $H \cup S \cup M$ in Figure 7.2. By eliminating redundant transfers, the reduction of HTTP traffic is as high as 17.7% and 20.3% for ISP and UMICH, respectively.

- Even at the scope of *all* traffic (HTTP and non-HTTP), the redundancy ratio is also significant (17.3% for UMICH) as indicated by the second number. Note that this is an underestimation because we did not consider the redundancy of HTTPS traffic accounting for 11.2% of the total traffic volume of UMICH.

  We do not have the number for the ISP dataset that only contains HTTP records. As reported by a recent measurement study [65], HTTP accounts for at least 90%[6] of the total traffic volume of an aggregated one-week dataset involving 600K cellular

[6] See Figure 1(a) of the paper [65]. The following categories use HTTP: **`web_browsing`**, **`smartphone_apps`**, **`market`**, and **`streaming`**.

subscribers collected in August 2010. If we assume that fraction is representative and apply it to our ISP dataset, then its overall redundancy ratio at the scope of all traffic is at least 16%.

- Recall that our simulator introduced a heuristic freshness lifetime when a response contains no expiration information. Table 7.4 shows this parameter has negligible impact on the amount of redundant data.

- Although the duration of the ISP trace is much shorter, its redundancy ratio is only marginally smaller than that of UMICH, implying the usage duration has limited impact on the redundancy ratio. This is partly explained by the strong temporal locality of accessing the same cache entry (Figure 7.5 in §7.4.2).

- For the UMICH dataset, the difference between redundancy ratios of per-process caches (PC) and a single global cache (GC) is as small as 0.3%.

7.4.2 The Impact of the Cache Size

Now we discard the assumption of unlimited cache size and consider a *finite* cache for the simulator. This helps quantify the impact of limited cache size on redundant transfers. We implemented LRU (Least Recently Used) algorithm for our simulator since all HTTP libraries and browsers we tested in §7.6 use LRU as the replacement algorithm, if they support caching. LRU discards the least recently accessed files first when the cache is full. With a finite cache size, the simulated cache may not capture all redundant transfers in the trace. We refer to those captured ones as *detected* redundant transfers.

Consider Figure 7.2 again. Let us decrease the simulated cache size. Then we see fewer detected redundant transfers (each of $|S|$, $|H|$, and $|M|$ decreases) since more previously detected redundant transfers are classified as **`CACHE_ENTRY_NOT_EXIST`** due to cache misses as the simulated cache becomes smaller. In particular, when the simulated cache size is smaller than the handset cache size, redundant transfers due to limited size of the hand-

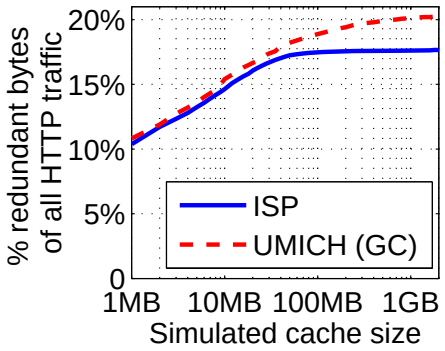


Figure 7.4: Relationship between the simulated cache size and the fraction of detected HTTP redundant bytes.

set cache will be eliminated (*i.e.,* $|S|$ becomes 0) because if a cache entry is evicted from the handset cache, it must have been evicted from the simulated cache (assuming both use LRU). Therefore, if the detected redundant bytes decrease to $x\%$ when the simulated cache size goes below the handset cache size, then the traffic volume impact of $H \cup M$ is at least $x\%$. Note this is a very loose lower bound in that $|H \cup M|$ also decreases as the simulated cache becomes smaller.

Our measurement results are shown in Figure 7.4 where we vary the simulated cache size from 1 MB to 2 GB for both datasets (note that the X-axis is in log scale). Figure 7.4 shows that even when the cache has a very small size of, for example, 4 MB[7], the detected redundant bytes is still as high as 12.8% and 13.2% (compared to 17.7% and 20.3% when the simulated cache has unlimited size), which are the aforementioned (loose) lower bounds of $|H \cup M|$, for ISP and UMICH, respectively. We therefore conclude that the problematic caching logic (instead of the limited cache size) takes the major responsibility for redundant transfers.

Figure 7.4 also suggests how to set the cache size, which can be small enough to entirely fit into today's smartphone memory with very limited additional cache misses incurred. For example, reducing the simulated cache size from infinity to 50 MB causes additional cache misses for only 2.0% and 0.4% of HTTP bytes (they are the loose upper bounds of the

[7]In comparison, our caching tests in Table 7.10 (§7.6.2) show that the cache sizes for the Android 2.2 browser and the Safari browser of iOS 4.3.4 / iPhone 4 are 8 MB and 100 MB, respectively.

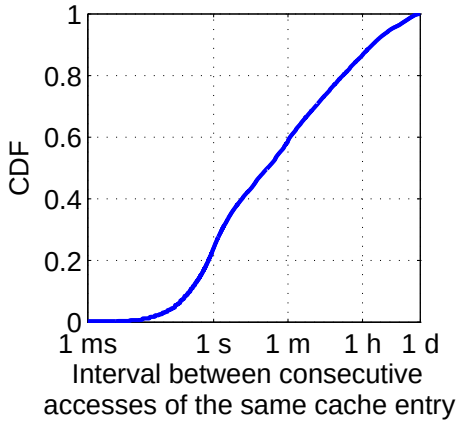


Figure 7.5: Distribution of intervals between consecutive accesses of the same simulated cache entry (for the ISP trace).

reduction of $|S|$) for UMICH and ISP, respectively, as indicated by the decrease of the detected redundant bytes shown in Figure 7.4.

Persistent vs. non-persistent cache. As described in §7.3.1, a non-persistent cache does not survive a process restart or a device reboot while a persistent cache does survive both. Based on our caching tests of 6 HTTP libraries and mobile browsers that support caching (§7.6.2), only one library for iOS (`NSURLRequest`) uses a non-persistent cache (Table 7.10). All Android libraries as well as both iPhone and Android browsers use persistent caches.

Our simulation assumes a persistent cache, which is consistent with the UMICH trace involving only Android handsets. For the ISP trace involving iOS devices, redundant transfers can also be caused by the non-persistent cache, which cannot be simulated since we do not know when a user restarts a process or reboots a handset. However, we expect the fraction of redundant transfers caused by the non-persistent cache is small due to two reasons. *(i)* Restarting a process happens infrequently in iOS [66]. On iPhone and iPad, pressing the "home" button simply puts an application to background. *(ii)* More quantitatively, Figure 7.5 plots the CDF of the intervals between consecutive accesses of the same simulated cache entry for the ISP trace. It is generated during the cache simulation. As shown in Figure 7.5, 59% of the intervals are less than 1 minute and 87% are less than 1 hour. We

expect such strong temporal locality of cache entry access makes a non-persistent cache comparable to a persistent cache in terms of efficiency.

7.4.3 Diversity Among Applications

This subsection investigates the caching efficiency of individual smartphone applications.

Identifying smartphone applications is trivial for the UMICH trace, which contains the process name for each packet and hence for each HTTP transaction. 741 unique processes were observed from the UMICH dataset.

For the ISP dataset whose application identification is less straightforward, we used the **`User-Agent`** field in HTTP requests to distinguish different applications. First, from the 48,214 unique **`User-Agent`** strings appeared in the dataset, we picked the top 500 strings with the highest HTTP traffic coverage, yielding an overall traffic coverage ratio of 95.5%. We found many **`User-Agent`** strings belong to the same application. They are only slightly different in OS versions, hardware specifications, and languages, *etc.* For example, Apple iTunes has the following **`User-Agent`** format: **`iTunes-`***Device*`/`*iOS version* **`(`***device version*`;` *memory size*`)`, such as **`iTunes-iPhone/4.3.3(4;16GB)`** or **`iTunes-iPhone/4.2.1 (2;8GB)`**. To avoid duplicated applications, we generated 95 regular expressions, each corresponding to an app for a specific device and/or OS, that cover all 500 **`User-Agent`** strings. All regular expressions follow a simple pattern of *app_name*`*`*device/OS_name*`*`, such as **`iTunes*iPhone*`** and **`Pandora*iOS*`**, by ignoring other less significant fields such as the OS version number.

Redundant transfers for top applications are measured in Table 7.5 (assuming an ideal cache for simulation). For the UMICH (ISP) trace, we show the top 12 process names (**`User-Agent`** regular expressions), their contributions of HTTP traffic, and their fractions of redundant bytes (their names have been anonymized). Due to the heavy-tail distribution of the smartphone application usage [65], these top apps are responsible for more than 83%

Table 7.5: Measuring redundant transfers for top applications in both datasets.

The UMICH dataset (HTTP Bytes: 101 GB)		
Android process*	% HTTP bytes	% redundant HTTP bytes
Streaming Service 1	29.8%	8.8%
Streaming Service 2	12.4%	0.5%
Web Browser 1	11.5%	20.4%
Entertainment	6.6%	12.0%
News and Weather	6.3%	55.3%
Lifestyle	3.9%	99.4%
Music and Audio 1	3.4%	0.0%
Music and Audio 2	2.9%	0.1%
Web Browser 2	1.8%	8.6%
Social Network Manager	1.7%	99.3%
Media and Video	1.7%	0.8%
Web Browser 3	1.4%	18.3%
(Total or average)	83.4%	18.3%

The ISP dataset (HTTP Bytes: 24.3 TB)		
`User-Agent` regex (device/OS name)*	% HTTP bytes	% redundant HTTP bytes
Streaming Service 1 (A)**	37.8%	20.1%
Internet Radio (A)	11.6%	1.9%
Web Browser (A)	11.3%	14.6%
Media player (A)	8.9%	3.1%
Map (A)	3.1%	0.0%
HTTP Library (B)	2.4%	86.6%
Web Browser (C)	2.1%	1.6%
Weather (A)	2.0%	93.0%
Social Networking (A)	1.6%	7.3%
Streaming Service 2 (D)	1.5%	19.1%
Web Browser (B)	1.4%	13.5%
Ad library (B)	1.0%	100.0%
(Total or average)	84.7%	18.2%

* The process names, **`User-Agent`** regular expressions, and device/OS names have been anonymized.

** For the ISP dataset, A, B, C, D refer to four different device/OS names after anonymization.

of all HTTP traffic. We found that while some apps incur small fractions of redundant data, some have unacceptably high redundancy ratios.

To validate the cache simulation results, we further studied the four apps with high redundancy ratios greater than 90% as shown in Table 7.5. We used them locally by exploring their common application usage scenarios, whose packet traces were simultaneously collected by `tcpdump` running on our handsets. By analyzing the traces, we found that all four apps use HTTP as the application-layer protocol but none of them performs caching. For example, for the "Weather (A)" app, HTTP responses of the weather information contain the `Expires` and the `Cache-Control:max-age` directives both specifying a freshness lifetime of 5 minutes. But when we checked the weather for the same location again (we verified from the trace that the URLs were identical), the handset always issued a non-conditional request regardless of the freshness of the downloaded file.

Inefficiency caused by HTTP POST. Table 7.5 indicates that the "Map (A)" app incurs negligible redundant transfers, because almost all its bytes are not cacheable (not shown). Specifically, we found that instead of employing HTTP GET, the application heavily uses HTTP POST requests that point to a single static file name by including the parameters in the body of a POST request, making it impossible for HTTP to cache the responses. A more caching-friendly approach is to attach query strings to the URLs to make them cacheable.

We summarize important findings regarding to the traffic volume impact of redundant transfers in Table 7.6.

7.5 The Resource Impact

§7.4 reveals the traffic volume impact of redundant transfers due to inefficient caching. In cellular networks, resources such as handset battery life, radio resources, and signaling load could also become critical bottlenecks. We now focus on understanding the impact of redundant transfers on cellular resource consumption: the handset radio energy consumption E, the signaling overhead S, and the radio resource consumption D. We have defined

Table 7.6: Findings regarding to the traffic volume impact of redundant transfers.

Question	Our finding based on the two datasets	§
File cacheability	Most bytes (70% to 78%) and most files (66% to 72%) are cacheable.	7.4.1
Traffic volume impact of redundant transfers	They account for 18% to 20% of HTTP traffic, and 17% of all traffic for the UMICH trace.	7.4.1
Main reason for redundant transfers	Problematic caching logic of the handsets (instead of the sever).	7.4.2
Impact of cache size on redundant transfers	Limited. The detected redundant bytes are at least 13% even for a simulated cache of as small as 4 MB.	7.4.2
Suggested handset cache size	Reducing the cache size from infinity to 50 (100) MB causes cache misses for at most 2.0% (1.4%) of HTTP bytes.	7.4.2
Difference between persistent cache and non-persistent cache	Very small. Both have similar caching efficiency due to strong temporal locality of accessing the same cache entry.	7.4.2
Caching efficiency of individual mobile apps	Some popular apps have unacceptably high fractions (93% to 100%) of redundant transfers.	7.4.3

these metrics in §2.4 and have described how to compute them in §3.4. We use Carrier 1, whose UMTS network was used by the 20 users contributing the UMICH dataset, to evaluate the impact. Its RRC state machine is depicted in Figure 2.2.

7.5.1 Computing the Resource Impact

We only studied the UMICH trace, because accurate RRC state reconstruction, a prerequisite for computing the above metrics, requires for each incoming and outgoing packet its precise timing and size, which is not available in the ISP trace. Also, we only considered the 3G traffic within the UMICH trace since the resource management policy of Wi-Fi is much more efficient due to its short-range nature[8].

To quantify the resource impact of redundant transfers, we use the methodology described in §2.4 by *taking the difference* of the computed metrics for the original trace and the modified trace with all redundant transfers removed. Similar analysis is performed

[8]Wi-Fi has very short tail time and negligible promotion delay[14].

in §4.4 and §6.4.2 to measure the impact of the inactivity timer and different tail optimization techniques, respectively. Specifically, the radio energy impact is computed as $\Delta E = (E_R - E_0)/E_0$ where E_0 and E_R correspond to the radio energy consumed by the original and the redundant-transfer-free trace, respectively. ΔE is negative as removing redundant transfers reduces energy consumption. The radio resource impact ΔD and the signaling impact ΔS are computed in similar ways.

Two factors may lead to underestimation of the resource impact. *(i)* We conservatively consider all HTTPS traffic non-redundant. *(ii)* In the above approach, by removing redundant transfers (or any transfers of our interest), we assume that any of the remaining traffic is *unaffected* in terms of its schedule and occurrence. This may not be true in reality: if some redundant transfer is eliminated, the subsequent transfer may happen sooner, or some other traffic may not occur at all (*e.g.,* a DNS lookup). Ignoring such dependency leads to an underestimation of the resource impact because in reality the resultant redundant-transfer-free trace has shorter duration and/or less traffic than our simulated one. Although it is difficult to handle all such cases, we do address a common case where a DNS lookup would not occur if its corresponding HTTP transfer were eliminated. Specifically, when removing a redundant transfer X, we also try to eliminate a DNS lookup D right before X, using a time window δ tolerating the handset processing delay. We empirically choose δ=300 ms but varying it from 100 to 500 ms has negligible impact on the results.

7.5.2 Measurement Results

Table 7.7 measures the resource impact of redundant transfers in two scenarios: *(i)* consider only 3G HTTP traffic and exclude 3G non-HTTP traffic, and *(ii)* consider all 3G traffic in the UMICH trace. The "ΔE (HTC)" and "ΔE (Nexus)" columns refer to the radio energy impact using power parameters of an HTC TyTn II smartphone and a Google Nexus One smartphone [6], respectively. The presented results also assume an ideal cache. We found that similar to our findings in §7.4.2, as long as the simulated cache size is reasonably

Table 7.7: Resource impact of redundant transfers (the UMICH trace), under the scopes of HTTP traffic and all traffic.

	ΔS	ΔE (HTC)	ΔE (Nexus)	ΔD
HTTP only	-26.9%	-26.1%	-25.9%	-27.1%
All traffic	-6.1%	-7.0%	-6.7%	-9.0%

large (*e.g.,* greater than 10 MB), its impact on the measured resource impact of redundant transfers is small (results not shown).

As shown in Table 7.7, by considering non-HTTP traffic, the resource impact of redundant transfers decreases sharply from more than 25% to less than 10%, although non-HTTP traffic accounts for only 13% of the total 3G traffic volume. This is attributed to two reasons explained below.

First, the resource impact of non-HTTP traffic can be significant although their traffic volume contribution is small. Due to the tail effect explained in §2.1.3, intermittently transmitting very small amount of data may utilize much more resources than transmitting large amount of data in one burst [6]. One representative example of such an inefficient traffic pattern identified in the UMICH trace is Android push notification (identified by TCP port 5228) and XMPP (Extensible Messaging and Presence Protocol, a popular instant messaging protocol using TCP port 5222 [67]) traffic. They account for 1% of the total 3G traffic volume while their resource impact in terms of E (HTC) is 18%[9]. On the other hand, for all HTTP traffic dominating the overall 3G traffic volume (87%), the resource impact in terms of E (HTC) is only 20%. Note that the traffic patterns of push notifications (and in general, delay-tolerant transfers) can be optimized to be more resource efficient [14, 68], resulting in higher resource impact of redundant transfers.

The second reason is *resource sharing*. Unlike the traffic volume measured in §7.4, resources in cellular networks can be shared by multiple HTTP sessions, or be shared by HTTP and non-HTTP transfers. Recall that in §2.1.3, the "radio-on" period of a transfer

[9]It is computed using the method described in §7.5.1 by taking the difference of the radio energy for the entire trace and the modified trace where push notification and XMPP transfers are removed.

consists of a state promotion, its data transmission period and the following tail. If the radio-on periods of two transfers are fully or partially overlapped, then D and hence E are shared during the overlapped period. The signaling load S is also shared in that only one state promotion is triggered by the two transfers. Resource sharing significantly reduces the resource impact of redundant transfers, many of which do not incur additional resource overhead because their channel occupation periods overlap with those of other transfers. For S, E (HTC), E (Nexus), and D, the fractions of resource reduction due to resource sharing among redundant transfers and other transfers are 70%, 61%, 63%, and 50%. respectively[10].

7.6 Finding the Root Cause

We learn from §7.4.2 that the main reason for redundant transfers is the problematic caching logic. We verify this by performing comprehensive caching tests for state-of-the-art HTTP libraries and browsers of Android and iOS.

Previously, professional developers also spent efforts investigating HTTP cache implementation issues leading to poor performance, focusing on mobile browsers [69, 70, 71, 72, 73]. Our tests go beyond them in two aspects. *(i)* Our tests are much more complete, covering all important aspects of caching implementation. To our knowledge, only three (Test 7, 10, 11) out of the thirteen tests described in §7.6.1 were performed before. *(ii)* Prior efforts only investigated browsers but we further examined HTTP libraries that are heavily used by today's smartphone applications.

7.6.1 Test Methodology

We examined eight HTTP libraries listed in Table 7.8. To the best of our knowledge, they cover all publicly available HTTP libraries for Android and iOS. To test them, we

[10]It is computed as $1 - (U(T) - U(T_2))/U(T_1)$ where $U(\cdot)$ computes the resource consumption for a certain trace. Trace T is the original trace of all traffic. T_1 consists of only redundant transfers. T_2 corresponds to T with T_1 removed.

Table 7.8: Our tested HTTP libraries and smartphone browsers.

Name	HTTP library or browser	Platform	Handset*
UC	**`java.net.URLConnection`**	Android 2.3	GS
HUC	**`java.net.HttpURLConnection`**	Android 2.3	GS
HC	**`org.apache.http.client.HttpClient`**	Android 2.3	GS
WV	**`android.webkit.WebView`**	Android 2.3	GS
HRC	**`android.net.http.HttpResponseCache`**	Android 4.0.2	GN
T20	**`Three20`** (Version 1.0.6.2)	iOS 4.3.4	iP4
NSUR	**`NSURLRequest`**	iOS 5.0.1	iP4S
ASIHR	**`ASIHTTPRequest`** (Version 1.8.1)	iOS 4.3.4	iP4
AB	The Android web browser	Android 2.3	GS
SB	The Safari web browser on iPhone	iOS 4.3.4 iOS 5.0.1	iP4 iP4S

* Handset: GS = Samsung Galaxy S, GN = Samsung Galaxy Nexus, iP4 = iPhone 4, iP4S = iPhone 4S.

wrote small applications using these libraries as HTTP clients. We also investigated the default browsers on Android and iPhone, using strategically generated HTML pages embedding multiple web objects to perform tests involving multiple files (for Test 11 and 12).

We performed all tests on real handset devices: Samsung Galaxy S with Android 2.3, Samsung Galaxy Nexus with Android 4.0.2, iPhone 4 with iOS 4.3.4, and iPhone 4S with iOS 5.0.1. Each handset has non-volatile storage of at least 10 GB. In each test, a client only requested files, whose caching directives were properly configured, from our controlled HTTP server running Apache 2.2. We ran **`tcpdump`** on the server to monitor incoming HTTP requests to tell whether a request we made was served by the handset cache or by the server.

We took the following measures to further eliminate external factors that may affect the accuracy. *(i)* Before launching each test, the handset cache (if existed) was always cleared either manually (for browsers) or by calling the corresponding APIs (for libraries). *(ii)* We verified that the caching behaviors of the server in all tests were correct by analyzing the traces collected at the server[11]. The clocks of both the server and the handset were

[11]The server was also tested by `http://redbot.org`, an online tool for checking HTTP caching implementation of web servers.

synchronized as well. *(iii)* We also ran **`tcpdump`** on the handset and compared HTTP requests and responses observed at the handset and at the server. We found both to be identical in all tests, implying that the cellular middleboxes did not change any caching directives. This is further confirmed by the fact that using cellular and Wi-Fi yielded the same testing results.

We designed 13 black-box tests to understand how caching was implemented in the state-of-the-art HTTP libraries and mobile browsers. Test 1 to Test 7 verify whether key features (*e.g.,* revalidation) were supported. Failure to support any of them may lead to redundant transfers. Test 8 to Test 13 determine important attributes of a cache (*e.g.,* its size) whose implications on redundant transfers cannot be overlooked either. Note the testing methodology is generally applicable to HTTP libraries and browsers on any platform. We detail the tests below.

Test 1 (Basic caching)**.** The handset requests for a small cacheable file f. The server transfers f with a proper **`Expires`** directive. Then the client requests for f again before it expires. If the basic caching is supported, the second request should not incur any network traffic.

Test 2 (Revalidation)**.** It is similar to Test 1 except that the client requests for f after it expires. For the second request, the client should issue a conditional request with an **`If-Modified-Since`** directive. Then the server will respond with an HTTP 304 indicating the file has not changed.

Test 3 (Non-caching directives)**.** It tests whether the following directives are correctly followed: **`Cache-Control:no-store`**, **`Pragma:no-cache`**, and **`Cache-Control:no-cache`**. Their caching logic is described in §7.2.

Test 4 (Expiration directives)**.** It is similar to Test 1 except that the server uses other expiration directives: *(i)* **`Cache-control:max-age`**, *(ii)* **`Expires`** and **`max-age`**, and *(iii)*

max-age and **age**. In *(ii)*, **max-age** should override **Expires**. In *(iii)*, the file should always expire if **age** is greater than **max-age**.

Test 5 (URL with query string)**.** It is similar to Test 1 except that the URL contains a query string (*e.g.,* **query.php?p=123**).

Test 6 (Partial caching)**.** The handset performs a byte-range request $[p_1, q_1]$ for a small cacheable file f. The server transfers the corresponding range of f with a proper **Expires** directive and an eTag. Before the response expires, the client performs another byte-range request $[p_2, q_2]$ for f. If $p_2 \geq p_1$ and $q_2 \leq q_1$, then we should not observe the second request over the network.

Test 7 (Redirection)**.** We test whether a handset caches two types of responses: **301 Moved Permanently** and **302 Found**. They are common ways of performing a permanent and a temporary redirection, respectively. A 301 response is always cacheable unless indicated otherwise (*e.g.,* by **Cache-Control:no-store**). A 302 response is only cacheable if indicated by a **Cache-Control** or **Expires** header field. The testing procedure is similar to Test 1 except that the response of the server is HTTP 301 or 302, redirecting the request to another small file.

Test 8 (Shared or non-shared cache)**.** We run two applications A and B using the same library to be tested on the same phone. A requests for a small cacheable file f. Then B requests for f again before f expires. If the cache is shared by both applications, the second request should not incur any network traffic. Otherwise we will see two requests over the network. We did not find any publicly available API that allows one to read or modify cache entries of the default Android or iPhone browser, whose caches are thus assumed to be non-shared.

Test 9 (Persistent or non-persistent cache)**.** We perform a test similar to Test 1 except that we reboot the phone after receiving the first response. A persistent cache must survive a device reboot (and therefore a process restart).

Test 10 (Cache entry size limit)**.** A large file may not be cached by a given cache implementation with a cache entry size limit. We perform binary search for this limit by fetching cacheable files of varying sizes from the server. For a file of size s_i, we perform Test 1 after clearing the cache to see whether s_i is above the cache entry size limit. Previous measurement [71] reported that HTML pages and external web objects (*e.g.,* JavaScript and CSS) may have different cache entry size limits, which are measured separately by us.

Test 11 (Total cache size)**.** We perform binary search for the total cache size. To test whether it exceeds a particular value z, we clear the cache and download n cacheable files f_1, ..., f_n each with a size of z/n (smaller than the cache entry size limit inferred by Test 10). We then request for the n files again. z exceeds the total cache size if and only if any file is transferred over the network in the second pass.

Test 12 (Replacement policy)**.** We test for five cache replacement algorithms known to be commonly implemented [74]: *(i)* LRU (Least Recently Used), *(ii)* LFU (Least Frequently Used), *(iii)* evicting the oldest cache entry, *(iv)* evicting the cache entry with the nearest expiration time, and *(v)* evicting the cache entry of the largest size.

To test whether the replacement policy is LRU, we first fill up the cache using n cacheable files f_1, ..., f_n such that $\sum_{i=1}^{n} f_i < z$ where z is the total cache size inferred by Test 11. Next, we randomly generate an n-permutation p_1, p_2,..., p_n, and then request for f_{p_1}, ..., f_{p_n} again. Subsequently the handset downloads a new file f_{n+1} such that $\sum_{i=1}^{n+1} f_i > z$, thus triggering a cache entry eviction. If f_{p_1} is evicted (based on Test 1), then we know LRU is the replacement policy since f_{p_1} is the least recently accessed file. Other replacement algorithms are tested in similar ways.

Test 13 (Heuristic freshness lifetime)**.** The HTTP server is configured in a way that it puts neither **`Cache-Control:max-age`** nor **`Expires`** in a response. Then we test whether a small file can be cached by performing Test 1 in which the two requests are sent back to back. If it can, we do binary search for the heuristic freshness lifetime (§7.3.3.1) by varying the interval between the two requests.

7.6.2 Test Results

Table 7.9 and Table 7.10 summarize the results (refer to Table 7.8 for acronyms of the libraries and browsers). Each feature in Test 1 to Test 7 can be fully supported (indicated by a "●" symbol), not supported at all ("○") or partially supported ("◑" with the reason explained). For each of the attribute tests (Test 8 to Test 13), a "✕" symbol means the test was not performed since the corresponding API does not support HTTP caching. We highlight key findings as follows.

- To our surprise, among the eight HTTP libraries, four (three for Android and one for iOS) do not support caching at all. Smartphone apps using these libraries thus cannot gain any benefit from caching. It is most likely that the library developers skip implementing the caching feature for simplicity. In fact, for **`java.net.HttpURLConnection`** and **`org.apache.http.client.HttpClient`**, they leave the implementation of the caching logic to library users by providing caching interfaces through several abstract classes. Other (less likely) concerns of using caching relate to its memory and storage overhead. We however show in §7.4.2 that the cache size does not need to be very large: reducing the cache size from infinity to 50 (100) MB causes cache misses for at most 2.0% (1.4%) of HTTP bytes. Another issue of a persistent cache is its performance. A recent study [75] has shown that the random write performance of flash storage (the persistent storage used by most handsets today) can be extremely low (0.02MB/s or even less), affecting caches that frequently perform synchronous random writes. For example, WebView on Android writes metadata into a SQLite database using synchronous random writes. The cache performance can be significantly improved by putting the cache in the RAM.

- For libraries and browsers that do support caching, they may not strictly follow RFC 2616, as detailed by footnotes d, h, j, k in Table 7.9 and Table 7.10. To our knowledge, only observations h and j were reported by previous measurements [73, 72]

Table 7.9: Testing results for smartphone HTTP libraries and browsers (Part 1). ●: fully supported ○: not supported ◑: partially supported ×: not applicable. Refer to Table 7.8 for acronyms of the libraries and browsers.

Test Name	UC	HUC	HC	WV	HRC
1. Basic caching	○	○[b]	○[a]	●	●
2. Revalidation	○	○	○	●	●
3. Non-caching directives	○	○	○	●	●
4. Expiration directives	○	○	○	●	●
5. URL with query string	○	○	○	●	●
6. Partial caching	○	○	○	○	○
7. Redirection	○	○	○	○	○
8. Shared or non-shared cache	×	×	×	Non-shared	Non-shared
9. Persistent or non-persistent cache	×	×	×	Persistent	Persistent
10. Cache entry size limit	×	×	×	2 MB	No limit
11. Total cache size	×	×	×	Storage Capacity	Specified by Developers[c]
12. Replacement policy	×	×	×	LRU	LRU
13. Heuristic fresh lifetime	×	×	×	17.5 hrs	30 mins

[a] Including subclasses **`AbstractHttpClient`**, **`AndroidHttpClient`**, and **`DefaultHttpClient`**. None supports basic caching.

[b] The class provides caching interfaces through the abstract classes **`ResponseCache`**, **`CacheRequest`**, and **`CacheResponse`**. But developers need to implement them by themselves.

[c] The cache size must be specified by developers.

Table 7.10: Testing results for smartphone HTTP libraries and browsers (Part 2).

Test Name	T20	NSUR	ASIHR	AB	SB
1. Basic caching	○	◑[j]	●	◑[j]	◑[j]
2. Revalidation	○	●	●	●	●
3. Non-caching directives	○	●	◑[d]	●	●
4. Expiration directives	○	●	●	●	●
5. URL with query string	○	●	●	●	●
6. Partial caching	○	○	○	○	○
7. Redirection	○	◑[h]	○	◑[h]	◑[h]
8. Shared or non-shared cache	×	Shared	Shared by default[e]	Non-shared	Non-shared
9. Persistent or non-persistent cache	×	Hybrid[l]	Persistent	Persistent	Persistent
10. Cache entry size limit	×	NP: 50 KB P: 2 MB[l]	No limit / 512 KB[i]	2 MB / 4 MB[i]	250 KB[g] / 4 MB[i]
11. Total cache size	×	NP: 1 MB P: 40 MB[f]	Storage Capacity	8 MB	100 MB
12. Replacement policy	×	LRU	LRU	LRU	LRU
13. Heuristic fresh lifetime	×	48 hrs	0[k]	48 hrs	48 hrs

[d] The class does not cache responses with **`Pragma:no-cache`** or **`Cache-Control:no-cache`**.

[e] Developers can make it non-shared by specifying a private cache storage path.

[f] The default sizes are 1 MB for the non-persistent cache and 40 MB for the persistent cache. But they are also configurable by developers.

[g] Safari on iOS 5 has a larger cache entry size limit of 2 MB for an HTML page.

[h] They do not cache a cacheable HTTP 302 response.

[i] The first and the second numbers are the cache entry size limits for an HTML page and an external web object (*e.g.,* JavaScript), respectively.

[j] When loading the same URL back-to-back, the second load is treated as a reload without using a cached copy or issuing a conditional request.

[k] Revalidation is always performed when neither **`Cache-Control:max-age`** nor **`Expires`** exists in a response.

[l] Both a persistent and a non-persistent cache are used. Given a file whose size is s, it is stored in the non-persistent cache if s <50KB, or stored in the persistent cache if 50KB$\leq s <$2 MB, or not stored if $s \geq$ 2MB.

(for browsers only). All such cases of non-compliance potentially incur redundant transfers.

- The Android browser uses a small cache of 8 MB. As described in §7.4.2, increasing the cache size brings non-trivial reduction of cache misses.

- No library or browser supports partial caching, although its impact on redundant transfers is limited (§7.4.1).

- We found that for all libraries that support caching, in order to leverage the caching support, a developer still needs to configure the library. However, developers can easily skip that for simplicity, or they can simply be unaware of it, therefore missing the opportunity of caching and incurring redundant transfers.

By exposing the shortcomings of existing implementations, our work helps encourage library and platform developers to improve the state of the art, and helps application developers choose the right libraries for better performance.

7.7 Discussion and Conclusion

Web caching in mobile networks is critical due to the unprecedented cellular traffic growth that far exceeds the deployment of cellular infrastructures. Caching on handsets is particularly important as it eliminates all network-related overheads. We have performed the first network-wide study of the redundant transfers caused by inefficient web caching on handsets. We found that redundant transfers account for 17% and 20% of the HTTP traffic, for the two large datasets, respectively. Further analysis on the UMICH trace suggests that redundant transfers are responsible for 17% of the bytes, 6% of the signaling load, 7% of the radio energy consumption, and 9% of the radio resource utilization of *all* cellular data traffic. Most of the redundancy can be eliminated by making the caching logic fully

support and strictly follow the protocol specification, and making developers fully utilize the caching support provided by the HTTP libraries.

For further optimizing web caching for mobile networks, we discuss three potential directions which are not explored in this Chapter.

The offline application cache is a new feature in HTML5, the latest version of the HTML standard [76]. It differs from HTTP caching in two ways. *(i)* Caching information of all objects embedded in an HTML page is specified in a small *cache manifest* file associated with the HTML page. *(ii)* There is no explicit expiration, which is instead indicated by a new version of the manifest file that is always downloaded whenever the HTML page is fetched over the network. Although the usage of HTML5 caching is very unpopular in our datasets (only one app in the UMICH trace used it), analysts envision it will eventually be widely used as almost all smartphones are expected to support HTML5 by 2013 [77]. We expect strategically employing this coarse-grained caching mechanism with the traditional per-file-based HTTP caching can achieve more reduction of revalidation traffic causing non-trivial resource consumption despite their small sizes [78].

Optimizing caching parameter settings based on the file semantics is not addressed in this chapter, as described in §7.3.1. However we do observe from our datasets examples where caching parameter settings are obviously too conservative. For example, in the UMICH trace, for the built-in weather app of Motorola Atrix, 95% of its bytes are marked by server as non-storable. We plan to conduct a more in-depth investigation on optimizing caching parameter configurations.

Previous caching proposals such as delta encoding [61] and piggyback cache validation [59] may provide additional benefits in cellular networks. For example, batching multiple validation requests into a single message [59] potentially reduces the resource overhead as otherwise each individual validation request may incur a separate tail. We plan to revisit both studies in our future work.

CHAPTER VIII

Related Work

8.1 RRC state machine: Inference, Measurement, and Optimization

Inference. Accurate inference of the RRC state machine is the first necessary step towards characterizing and improving the resource management policy in cellular networks. Previous work [79, 25] introduced 3G Transition Triggering Tool to infer RRC state machine parameters and to measure one-way delays in different RRC states. But their approach is based on a fixed state transition model and only infers the corresponding parameters (*e.g.,* inactivity timers). Beyond their work, in §3.1, we also considered and inferred different state transition models configured by two commercial UMTS carriers.

Measurement. To the best of our knowledge, our work described in Chapter IV is the first comprehensive study that characterizes the RRC state machine and and investigate the optimality of state machine configurations using realistic traffic patterns. The key difference between our work and previous measurement studies of cellular networks (*e.g.,* 3gTest [45], LiveLab [41], and a study of the diversity of smartphone usage [42]) is that, previous ones focus on characterization at only IP and higher layers while ours puts special emphasis on the radio resource control layer and its interaction with the upper layers.

Optimization. The problem of choosing the optimal inactivity timer values has also been studied in several previous work. Among them, Chuah *et al.* [80] studied the impact of inactivity timers on UMTS network capacity by simulating the performance of web

browsing. Some [19, 20] proposed analytical models to measure the energy consumption of user devices under different timer values. Previous studies [21, 22] also discussed the influence of different timeout values on both service quality and energy consumption. In addition, several other projects also studied network resource management [81, 82]. However, all this prior work was evaluated based on simulation using particular traffic models. In fact, real traffic patterns depend highly on user behavior among other factors and are not easily captured using analytic models. We therefore use real traffic traces for evaluation to ensure the applicability of our work to real network settings.

Researchers also proposed ideas on setting timeout values dynamically. In [83] timeouts for the inactivity timers are decided dynamically for radio resources and computation resources. However, they only addressed the problem from the perspective of network capacity as to reducing the call blocking and dropping rate. In their scenario, the same timer values are applied *globally* to all handsets at any given time. The fast dormancy scheme described in §2.5 and §4.5.2 is essentially setting the timer dynamically. Its goal is to save radio resources and handset's energy. Therefore the dynamic timer (*i.e.,* the invocation of fast dormancy) is customized for each handset. Using fast dormancy as the building block, in Chapter VI, we propose Tail Optimization Protocol (TOP), an application-layer protocol that bridges the gap between the application and the fast dormancy support provided by the network. Some of the key challenges we address include the required changes to the OS, applications, and the implication of multiple concurrent connections using fast dormancy. In particular, TOP addresses three key issues associated with allowing smartphone applications to benefit from this support.

8.2 Profiling and Optimizing Mobile Applications

In Chapter §IV, we systematically characterize the impact of the RRC state machine on radio resources and energy by analyzing traces collected from a commercial UMTS network. As mentioned in §8.1, similar measurements have been done by previous studies

such as [20] and [21], using analytical models. Recent work [39] also investigates impact of traffic patterns on radio power management policy and proposes suggestions such as reducing the tail time to save handset energy. These studies did examine the interplay between smartphone applications and the state machine behavior, while our work described in Chapter §V makes a significant further step by introducing a novel tool, the mobile Application Resource Optimizer (ARO) that systematically correlates information at multiple layers to reveal the low efficiency of resource utilization to application developers.

Many work propose techniques to optimize smartphone application performance and energy efficiency by strategically scheduling applications' data transfers or changing applications' traffic patterns. Prior work [14] introduced TailEnder, which delays transfers of delay-tolerant traffic and transfers them with normal traffic, so that the overall tail time incurred by delay-tolerant traffic could be reduced. Also prefetching could be used to reduce tail time. Similar scheduling strategies for cellular networks are presented in [36], which delays transfers to offload data transfers from 3G to WiFi, when delaying reduces 3G usage and the transfers can be completed within the application's tolerance threshold. Given that cellular radios consume more power and suffer reduced data rate when the signal is weak, the Bartendr system [28] allows applications to preferentially communicate when the signal is strong, either by deferring non-urgent communication or by advancing anticipated communication to coincide with periods of strong signal, in order to reduce the energy consumption. Other cellular data scheduling approaches include piggyback [68], batching [68], Time Alignment [84], and Intentional Networking [85]. In contrast, our ARO tool provides application developers with more opportunities to optimize short bursts, whose resource impact is particularly high due to the tail effect (§2.1.3), that can be triggered by multiple factors.

The ARO tool provides a surprising observation (§5.5.2.1) that periodic transfers, where a handset periodically exchanges small amount of data with a remote server, can be very resource-inefficient in cellular networks, we recently performed the first network-wide,

large-scale investigation of cellular periodic transfers [68]. Using a large packet trace collected from a commercial cellular carrier, we found that periodic transfers are very prevalent in today's smartphone traffic. However, they are extremely resource-inefficient for both the network and end-user devices even though they predominantly generate very little traffic. This is a direct consequence of the tail effect (§2.1.3) in cellular networks. For example, for Facebook, periodic transfers account for only 1.7% of the overall traffic volume but contribute to 30% of the total handset radio energy consumption. We found periodic transfers are generated for various reasons such as keep-alive, polling, and user behavior measurements. We further investigate the potential of various traffic shaping and resource control algorithms, including piggyback, batching, TailEnder [14], and fast dormancy (§2.5). Depending on their traffic patterns, applications exhibit disparate responses to optimization strategies. Jointly using several strategies with moderate aggressiveness can eliminate almost all energy impact of periodic transfers for popular applications such as Facebook and Pandora.

Complementary to our ARO tool are profiling tools that focus on power modeling. For example, PowerTutor [18] is an online power estimation tool to determine system-level power consumption for smartphone devices. Based on the deployed inactivity timers in current commercial networks, studies [86, 87, 88] carried out measurement studies to examine the energy consumption and network performance on smartphones. Prior work [89] attempted to optimize network performance and increase energy efficiency by proposing a self-tuning power management (STPM) that adapts its behavior to the access patterns and intent of applications, the characteristics of the network interface, and the energy usage of the platform, for Wi-Fi networks. The Cool-Tether architecture [90] harnesses the cellular radio links of one or more mobile smartphones in the vicinity, builds a Wi-Fi hotspot on-the-fly, and provides energy-efficient, affordable connectivity.

8.3 Caching and Redundancy Elimination

To the best of our knowledge, our study described in Chapter VII is the first comprehensive investigation of HTTP cache implementation on mobile devices. We describe its related work in four categories below.

Extensive research on web caching has been done since the World Wide Web was in its nascent state. We summarize the important topics. *(i)* Web server workload modeling and characterization, focusing on the implication on caching [91, 62]. *(ii)* Efficient cache replacement algorithms [58, 74]. *(iii)* Efficient cache validation and invalidation techniques for strong consistency [59, 60]. Note a *validation* is initiated by a client which verifies the validity of its cached files (as used in HTTP), while an *invalidation* is performed by the origin server which notifies clients which of its cached files have been modified. *(iv)* Cooperative proxy caching [57, 63] where individual proxies share their cached files with each other's clients. *(v)* Caching-friendly content representation such as delta encoding [61].

Caching in mobile networks. Recent study [1] explored the potential benefits of HTTP caching in 3G cellular networks by analyzing traffic traces collected from a large 3G cellular carrier. They found that the cache hit ratio is 33% when caching at the Internet gateway. Another study [64] investigated the potential for caching video content in cellular networks, indicating that 24% of the bytes for progressive video download can be served from a network cache located at the Internet gateway. By comparison, our study investigates caching efficiencies from the perspective of individual handsets.

Another recent study [78] examined three client-only solutions: caching, prefetching, and speculative loading, using web usage data collected from 24 iPhone users over one year. The authors focus on improving the smartphone browsing speed instead of saving the bandwidth. They found that 40% of resource requests can be served by a local browser cache of 6 MB, implying the necessity of HTTP caching. However, its effectiveness of reducing the latency is found to be not as good as that of reducing the traffic volume, mainly because revalidation cannot hide network RTT, which is an important factor affecting mo-

bile browser performance.

HTTP cache implementation on browsers. Professional developers spent efforts investigating HTTP cache implementation issues leading to poor performance, focusing on mobile browsers. The following measurement studies were reported on various technical blogs. Early measurement [69] in 2008 shows that the iPhone 3G browser has a non-persistent cache with an entry size limit of 25 KB (for HTML files) and a total size of 500 KB, implying potential performance issue for large web pages. The experiments were revisited in 2010 [70], and larger cache sizes of iOS 4 on iPhone 4 and Android 2.1 were observed. Similar tests of caching sizes were performed in [66]. Blog entry [71] further pinpoints that for iPhone and Android browsers, the cache entry size limit differs depending on the file type (we considered this in our tests). Blog entry [72] revealed an implementation bug of Safari on iOS 4 shown in footnote j in Table 7.10. We confirmed this and found this problem also exists in the Android 2.3 browser and the `NSURLRequest` library. Blog entry [73] discovered that most desktop and mobile browsers do not cache HTTP redirections properly. Our caching tests cover all aforementioned aspects, and are much more complete as described at the beginning of §7.6.

Data compression. Besides caching, another orthogonal approach for redundancy elimination is data compression, which can be performed at each single object (*e.g.,* gzip [92]), across multiple objects (*e.g.,* shared dictionary compression over HTTP [93]), or for packet streams (*e.g.,* MODP [94]). Compression can be jointly applied with caching to further save the network bandwidth.

CHAPTER IX

Conclusion and Future Work

My dissertation is dedicated to address two major challenges associated with cellular carriers and their customers: carriers operate under severe resource constraints, while mobile applications often utilize radio channels and consume handset energy inefficiently. From carriers' perspective, we performed the first measurement study to understand the state-of-the-art of resource utilization for a commercial cellular network, and revealed that fundamental limitation of the current resource management policy is treating all traffic according to the same resource management policy globally configured for all users. From mobile applications' perspective, we developed a novel data analysis framework called ARO (mobile Application Resource Optimizer), the first tool that exposes the interaction between mobile applications and the radio resource management policy, to reveal inefficient resource usage due to a lack of transparency in the lower-layer protocol behavior. ARO revealed that many popular applications built by professional developers have significant resource utilization inefficiencies that are previously unknown. Motivated by the observations from both sides, we proposed a novel resource management framework that enables cooperation between handsets and the network to allow adaptive resource release, therefore better balancing the key tradeoffs in cellular networks. We also investigated the problem of reducing the bandwidth consumption in cellular networks by performing the first network-wide study of HTTP caching on smartphones due to its popularity. Our find-

ings suggest that for web caching, there exists a huge gap between the protocol specification and the protocol implementation on today's mobile devices, leading to significant amount of redundant network traffic. In summary, my dissertation indicates the importance of all of the following:

- Understanding the underlying radio resource control mechanism;
- Properly handling the interaction between mobile applications and lower layer behavior;
- Leveraging the knowledge of handsets to facilitate resource management;
- Ensuring the consistency between protocol specification and implementation.

In my opinion, it is good to adhere to the following general principles in the course of system and networking research.

- **The measurement observations are representative.** We collaborated with a commercial cellular ISP in the U.S., and collected cellular data of hundreds of thousands users from its cellular core network, to ensure the representativeness of our observations drawn from the data.
- **The solution is general** in that it attacks the fundamental limitations of cellular networks. My proposed methodologies are directly applicable to any type of cellular networks including 2G GPRS/EDGE, 3G UMTS/HSPA, and 4G LTE networks that employ similar core resource management policies.
- **The solution is practically deployable.** The ARO [6] and TOP [7] systems do not require any change to the cellular infrastructure. In particular, our ARO prototype [51] has been productized by AT&T and is now available to developers [52]. For resource inefficiencies found in popular smartphone applications such as Pandora and Facebook [6, 4], we have contacted the corresponding developers, and the responses were encouragingly positive [95].

- **The underlying concept has even longer-term impact.** My proposed frameworks of cross-layer analysis [6] and cooperative resource management [7] provide insightful guidelines for analyzing and optimizing general wireless network systems.

9.1 Future Work

In the course of my research, I have noticed that understanding the underlying radio resource control mechanism and its implications helps balance the key tradeoffs in cellular data networks and improve the resource efficiency for mobile applications. In the near future, I am interested in further leveraging this guideline to make wireless systems more resource efficient, as well as identifying the new challenges of cellular network and smartphone applications.

The network: from 3G to 4G. Currently 3G (UMTS, EvDO, and HSPA) is the mainstream cellular access technology. In 2009, 4G LTE (Long Term Evolution) started entering the commercial markets and is available now in more than 10 countries with a fast-growing user base. Besides the higher bit rate and lower latency that significantly outperform those of 3G, LTE employs a more complex RRC state machine with DRX (Discontinuous Reception where the handset periodically wakes up to check paging messages and sleeps for the remaining time) enabled even when a handset is occupying the high-speed transmission channel, in order to save the energy. We have done preliminary analysis on understanding the RRC policy, the power model, and the impact of DRX on application performance in 4G LTE networks [11]. A more in-depth exploration is an important part of my future work.

The apps: from individuals to the full spectrum. Given the extreme popularity of smartphone applications, a critical missing piece of information needed by carriers is their efficiencies of resource usage (radio resource utilization, signaling load, and handset energy consumption), which may have little correlation with their bandwidth consumption. Based on my experience of developing ARO, I plan to build a real-time monitoring system that

can be leveraged by cellular carriers, which already have infrastructures to capture packet data in their core networks, to monitor resource efficiencies for a wide range of applications used by millions of customers. For those resource-inefficient applications detected by the system, the carrier can contact their developers for improvement.

We face three challenges towards building such a monitoring system. First, unlike ARO that can obtain various types of data directly from handsets, the only type of data available to the monitoring system is the network packet traces, whose payload needs to be carefully mined to extract useful information. Second, since the data collection is completely passive, multiple applications can be simultaneously running on a handset. However, the RRC state transitions are determined by the aggregated traffic of all applications. Therefore we need to separate the resource impact of each of the concurrent applications running on a handset. Third, as a real-time monitoring system, it should be well scalable with small computation and storage overhead.

The optimization techniques: from uniform to diverse. Besides handset-based HTTP caching, I plan to pursue other directions for reducing the amount of data transferred in cellular networks, such as efficient compression, delta encoding [61], and the offline application cache provided by HTML5, which is expected to be supported by almost all smartphones by 2013 [77]. There remain three main challenges: *(i)* how to select the most effective technique for a particular content type or content provider, *(ii)* how to handle the complex interplay among multiple techniques when they are used together (if this brings additional benefits), and *(iii)* how to make the entire mechanism as transparent as possible from the application developers' perspective.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Erman, J., Gerber, A., Hajiaghayi, M., Pei, D., Sen, S., and Spatscheck, O., "To Cache or not to Cache: The 3G case," *IEEE Internet Computing*, 2011.

[2] "Invest in Cell Phone Infrastructure for Growth in 2010," `http://pennysleuth.com/invest-in-cell-phone-infrastructure-for-growth-in-2010/`.

[3] Holma, H. and Toskala, A., "HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications," John Wiley and Sons, Inc., 2006.

[4] Qian, F., Wang, Z., Gao, Y., Huang, J., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization," *WWW*, 2012.

[5] Qian, F., Wang, Z., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "Characterizing Radio Resource Allocation for 3G Networks," *IMC*, 2010.

[6] Qian, F., Wang, Z., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "Profiling Resource Usage for Mobile Applications: a Cross-layer Approach," *Mobisys*, 2011.

[7] Qian, F., Wang, Z., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "TOP: Tail Optimization Protocol for Cellular Radio Resource Allocation," *ICNP*, 2010.

[8] "Fast Dormancy: a way forward," 3GPP discussion and decision notes R2-085134, 2008.

[9] Qian, F., Quah, K. S., Huang, J., Erman, J., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "Web Caching on Smartphones: Ideal vs. Reality," *Mobisys*, 2012.

[10] Chatterjee, M. and Das, S. K., "Optimal MAC State Switching for CDMA2000 Networks," *INFOCOM*, 2002.

[11] Huang, J., Qian, F., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "A Close Examination of Performance and Power Characteristics of 4G LTE Networks," *Mobisys*, 2012.

[12] Perez-Romero, J., Sallent, O., Agusti, R., and Diaz-Guerra, M., "Radio resource management strategies in UMTS," John Wiley and Sons, Inc., 2005.

[13] "System Impact of Poor Proprietary Fast Dormancy," 3GPP discussion and decision notes RP-090941, 2009.

[14] Balasubramanian, N., Balasubramanian, A., and Venkataramani, A., "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications," *IMC*, 2009.

[15] "3GPP TR 25.813: Radio interface protocol aspects (V7.1.0)," 2006.

[16] "3GPP TS 36.331: Radio Resource Control (RRC) (V10.3.0)," 2011.

[17] "Monsoon Power Monitor," `http://www.msoon.com/`.

[18] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R., Mao, Z. M., and Yang, L., "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," *CODES+ISSS*, 2010.

[19] Lee, C.-C., Yeh, J.-H., and Chen, J.-C., "Impact of inactivity timer on energy consumption in WCDMA and cdma2000," *Wireless Telecommunications Symposium*, 2004.

[20] Yeh, J.-H., Chen, J.-C., and Lee, C.-C., "Comparative Analysis of Energy-Saving Techniques in 3GPP and 3GPP2 Systems," *IEEE transactions on vehicular technology*, Vol. 58, No. 1, 2009.

[21] Liers, F., Burkhardt, C., and Mitschele-Thiel, A., "Static RRC Timeouts for Various Traffic Scenarios," *PIMRC*, 2007.

[22] Talukdar, A. and Cudak, M., "Radio resource control protocol configuration for optimum Web browsing," *IEEE VTC*, 2002.

[23] "UE "Fast Dormancy" behavior," 3GPP discussion and decision notes R2-075251, 2007.

[24] "Configuration of Fast Dormancy in Release 8," 3GPP discussion and decision notes RP-090960, 2009.

[25] Perala, P., Barbuzzi, A., Boggia, G., and Pentikousis, K., "Theory and Practice of RRC State Transitions in UMTS Networks," *Proc. of IEEE Broadband Wireless Access Workshop*, 2009.

[26] Holma, H. and Toskala, A., "WCDMA for UMTS: HSPA Evolution and LTE," John Wiley and Sons, Inc., 2007.

[27] "GERAN RRC State Mchine," 3GPP GAHW-000027, 2000.

[28] Schulman, A., Navda, V., Ramjee, R., Spring, N., Deshpande, P., Grunewald, C., Jain, K., and Padmanabhan, V., "Bartendr: A Practical Approach to Energy-aware Cellular Data Scheduling," *Mobicom*, 2010.

[29] "System Parameter Recommendations to Optimize PS Data User Experience and UE Battery Life," Engineering Services Group, Qualcomm, 2007.

[30] "Pandora Radio," http://www.pandora.com.

[31] Balakrishnan, M., Mohomed, I., and Ramasubramanian, V., "Where's That Phone?: Geolocating IP Addresses on 3G Networks," *IMC*, 2009.

[32] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S., "On the Characteristics and Origins of Internet Flow Rates," *SIGCOMM*, 2002.

[33] Qian, F., Gerber, A., Mao, Z. M., Sen, S., Spatscheck, O., and Willinger, W., "TCP Revisited: A Fresh Look at TCP in the Wild," *IMC*, 2009.

[34] Lee, P. P. C., Bu, T., and Woo, T., "On the Detection of Signaling DoS Attacks on 3G Wireless Networks," *INFOCOM*, 2007.

[35] "DNSQueryTimeouts," http://technet.microsoft.com/en-us/library/cc977482.aspx.

[36] Balasubramanian, A., Mahajan, R., and Venkataramani, A., "Augmenting Mobile 3G Using WiFi," *Mobisys*, 2010.

[37] Allman, M., Paxson, V., and Stevens, W. R., "TCP Congestion Control," RFC 2581, 1999.

[38] Guo, L., Tan, E., Chen, S., Xiao, Z., Spatscheck, O., and Zhang, X., "Delving into Internet Streaming Media Delivery: A Quality and Resource Utilization Perspective," *IMC*, 2006.

[39] Falaki, H., Lymberopoulos, D., Mahajan, R., Kandula, S., and Estrin, D., "A First Look at Traffic on Smartphones," *IMC*, 2010.

[40] "A Call for More Energy-Efficient Apps," http://www.research.att.com/articles/featured_stories/2011_03/201102_Energy_efficient.

[41] Shepard, C., Rahmati, A., Tossell, C., Zhong, L., and Kortum, P., "LiveLab: Measuring Wireless Networks and Smartphone Users in the Field," *HotMetrics*, 2010.

[42] Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., and Estrin, R. G. D., "Diversity in Smartphone Usage," *Mobisys*, 2010.

[43] Maier, G., Schneider, F., , and Feldmann, A., "A First Look at Mobile Hand-held Device Traffic," *PAM*, 2010.

[44] Gember, A., Anand, A., and Akella, A., "A Comparative Study of Handheld and Non-Handheld Traffic in Campus WiFi Networks," *PAM*, 2011.

[45] Huang, J., Xu, Q., Tiwana, B., Mao, Z. M., Zhang, M., and Bahl, P., "Anatomizing Application Performance Differences on Smartphones," *Mobisys*, 2010.

[46] Veal, B., Li, K., and Lowenthal, D., "New Methods for Passive Estimation of TCP Round-Trip Times," *PAM*, 2005.

[47] "Add an Expires or a Cache-Control Header," `http://developer.yahoo.com/performance/rules.html#expires`.

[48] Chakravorty, R. and Pratt, I., "WWW Performance over GPRS," *IEEE MWCN*, 2002.

[49] Fielding, R., Gettys, J., Mogul, J., Masinter, H. F. L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol - HTTP/1.1 ," RFC 2616, 1999.

[50] "Google Instant search now available globally for iOS4 and Android 2.2+," `http://www.mobileburn.com/news.jsp?Id=12012`.

[51] Qian, F., Wang, Z., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "Mobile Application Resource Optimizer (ARO)," *Mobisys (System Demo)*, 2011.

[52] "AT&T Application Resource Optimizer (ARO)," `http://www.research.att.com/articles/featured_stories/2012_01/201201_ARO_Release`.

[53] Sesia, S., Toufik, I., and Baker, M., "LTE: The UMTS Long Term Evolution From Theory to Practice," John Wiley and Sons, Inc., 2009.

[54] "Cisco Visual Networking Index Forecast Projects 18-Fold Growth in Global Mobile Internet Data Traffic From 2011 to 2016," `http://newsroom.cisco.com/press-release-content?type=webcontent&articleId=668380`, 2012.

[55] Ashok Anand, Chitra Muthukrishnan, A. A. and Ramjee, R., "Redundancy in Network Traffic: Findings and Implications," *SIGMETRICS*, 2009.

[56] Erman, J., Gerber, A., Hajiaghayi, M., Pei, D., and Spatscheck, O., "Network-Aware Forward Caching," *WWW*, 2009.

[57] Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M., and Worrell, K., "A hierarchical internet object cache," *USENIX ATC*, 1996.

[58] Cao, P. and Irani, S., "Cost-aware WWW proxy caching algorithms," *USITS*, 1997.

[59] Krishnamurthy, B. and Wills, C., "Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web," *USITS*, 1997.

[60] Liu, C. and Cao, P., "Maintaining Strong Cache Consistency in the World-Wide Web," *ICDCS*, 1997.

[61] Mogul, J., Douglis, F., Feldmann, A., and Krishnamurthy, B., "Potential benefits of delta encoding and data compression for HTTP," *SIGCOMM*, 1997.

[62] Caceres, R., Douglis, F., Feldmann, A., Glass, G., and Rabinovich, M., "Web proxy caching: the devil is in the details," *SIGMETRICS Perf. Eval. Rev.*, Vol. 26, No. 3, 1998.

[63] Wolman, A., Voelker, G., Sharma, N., Cardwell, N., Karlin, A., and Levy, H., "On the scale and performance of cooperative Web proxy caching," *SOSP*, 1999.

[64] Erman, J., Gerber, A., Ramakrishnan, K., Sen, S., and Spatscheck, O., "Over The Top Video: the Gorilla in Cellular Networks," *IMC*, 2011.

[65] Xu, Q., Erman, J., Gerber, A., Mao, Z. M., Pang, J., and Venkataraman, S., "Identifying Diverse Usage Behaviors of Smartphone Apps," *IMC*, 2011.

[66] "Understanding Mobile Cache Sizes," `http://www.blaze.io/mobile/understanding-mobile-cache-sizes/`.

[67] "Extensible Messaging and Presence Protocol," `http://xmpp.org/xmpp-protocols/`.

[68] Qian, F., Wang, Z., Gao, Y., Huang, J., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O., "Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization," *WWW*, 2012.

[69] "iPhone Cacheability - Making it Stick," `http://www.yuiblog.com/blog/2008/02/06/iphone-cacheability/`.

[70] "Mobile Browser Cache Limits: Android, iOS, and webOS," `http://www.yuiblog.com/blog/2010/06/28/mobile-browser-cache-limits/`.

[71] "Mobile cache file sizes," `http://www.stevesouders.com/blog/2010/07/12/mobile-cache-file-sizes/`.

[72] "(lack of) Caching for iPhone Home Screen Apps," `http://www.stevesouders.com/blog/2011/06/28/lack-of-caching-for-iphone-home-screen-apps/`.

[73] "Redirect caching deep dive," `http://www.stevesouders.com/blog/2010/07/23/redirect-caching-deep-dive/`.

[74] Wong, K.-Y., "Web Cache Replacement Policies: a Pragmatic Approach," *IEEE Network*, Vol. January/February, 2006.

[75] Hyojun Kim, Nitin Agrawal, C. U., "Revisiting Storage for Smartphones," *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[76] "HTML5 (W3C working draft)," `http://www.w3.org/TR/html5/`.

[77] "HTML5-enabled phones to hit 1 billion in sales in 2013," `http://news.cnet.com/8301-1023_3-57339156-93/html5-enabled-phones-to-hit-1-billion-in-sales-in-2013/`.

[78] Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M., "How Far Can Client-Only Solutions Go for Mobile Browser Speed?" *WWW*, 2012.

[79] Barbuzzi, A., Ricciato, F., and Boggia, G., "Discovering Parameter Setting in 3G Networks via Active Measurements," *Communications Letters, IEEE*, Vol. 12, No. 10, 2008.

[80] Chuah, M., Luo, W., and Zhang, X., "Impacts of Inactivity Timer Values on UMTS System Capacity," *Wireless Communications and Networking Conference*, 2002.

[81] Ghaderi, M., Sridharan, A., Zang, H., Towsley, D., and Cruz, R., "TCP-Aware Resource Allocation in CDMA Networks," *Proceedings of ACM MOBICOM*, Los Angeles, CA, USA, September 2006.

[82] Sridharan, A., Subbaraman, R., and Guerin, R., "Distributed Uplink Scheduling in CDMA Networks," *Proceedings of IFIP-Networking 2007*, May 2007.

[83] Liers, F. and Mitschele-Thiel, A., "UMTS data capacity improvements employing dynamic RRC timeouts," *PIMRC*, 2005.

[84] Kononen, V. and Paakkonen, P., "Optimizing Power Consumption of Always-On Applications Based on Timer Alignment," *COMSNETS*, 2011.

[85] Higgins, B., Reda, A., Alperovich, T., Flinn, J., Giuli, T., Noble, B., and Watson, D., "Intentional Networking: Opportunistic Exploitation of Mobile Network Diversity," *Mobicom*, 2010.

[86] Tan, W. L. and Yue, O., "Measurement-based Performance Model of IP Traffic over 3G Networks," *TENCON 2005 2005 IEEE Region 10*, November 2005, pp. 1–5.

[87] Haverinen, H., Siren, J., and Eronen, P., "Energy Consumption of Always-On Applications in WCDMA Networks," *IEEE VTC*, 2007.

[88] Liu, X., Sridharan, A., Machiraju, S., Seshadri, M., and Zang, H., "Experiences in a 3G network: interplay between the wireless channel and applications," *Mobicom*, 2008.

[89] Anand, M., Nightingale, E. B., and Flinn, J., "Self-Tuning Wireless Network Power Management," *Wireless Networks*, Vol. 11, No. 4, 2005.

[90] Sharma, A., Navda, V., Ramjee, R., Padmanabhan, V., and Belding, E., "Cool-Tether: Energy Efficient On-the-fly WiFi Hot-spots using Mobile Phones," *CoNEXT*, 2009.

[91] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S., "Web Caching and Zipf-like Distributions: Evidence and Implications," *INFOCOM*, 1999.

[92] "The gzip home page," `http://www.gzip.org/`.

[93] Butler, J., Lee, W.-H., McQuade, B., and Mixter, K., "A Proposal for Shared Dictionary Compression over HTTP," `http://groups.google.com/group/SDCH`.

[94] Lumezanu, C., Guo, K., Spring, N., and Bhattacharjee, B., "The Effect of Packet Loss on Redundancy Elimination in Cellular Wireless Networks," *IMC*, 2010.

[95] "AT&T API Platform Tools used in the Pandora Mobile App," `http://www.youtube.com/watch?v=3WuCDwnQyfM`.