Improving Mobile Network Performance Through Measurement-Driven System Design Approaches

by

Sanae Rosen

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
2017

Doctoral Committee:

Professor Z. Morley Mao, Chair
Professor Jason N. Flinn
Assistant Professor Harsha V. Madhyastha
Associate Professor Vijay G. Subramanian

To my parents and my brother

ACKNOWLEDGEMENTS

First of all, I would like to thank Professor Morley Mao for her excellent advising over the years, for always believing in my ability to succeed and for encouraging me to always strive to be a better researcher. The depth and breadth of her knowledge and expertise have been an invaluable help, and I have learned an incredible amount working with her. I have been truly lucky to have her as my advisor .

I would also like to thank my thesis committee: Professor Jason Flinn, Professor Harsha Madhyastha and Professor Vijay Subramanian. Their expertise, advice and comments on my work have been invaluable in shaping my thesis, and in particular in leading my research in a strong direction for the last few projects. I am very glad I have had the opportunity to work with them.

My internships have also given me the opportunity to work with some truly excellent researchers who were instrumental to my grad student career: in particular, Professor S.J. Lee, J. K. Lee, Vijay Gopalakrishnan, Professor K. K. Ramakrishnan, Jeff Erman and Jeff Pang have all been wonderful to work with, and I am very lucky to have has the opportunity to work with such a large number of prominent researchers and benefit from their expertise. Outside of internships, I've collaborated with many great people at T-Mobile and AT&T more generally, including Shuai Hao, Bo Han, Shubho Sen, and Professor David Choffnes, all of whom I have greatly enjoyed working with.

I've also received a great deal of support and mentorship from grad students and former grad students in my lab. Ever since he was a grad student, Professor Feng Qian has been of great help and has acted as a mentor over the years, and Professor Zhiyun Qian did

an excellent job of helping introduce me to research when I was starting out and he was finishing up his PhD. I've had many other excellent grad student collaborators as well, including Haokun Luo, Ashkan Nikravesh, Qi Alfred Chen, and Yihua Guo. I've greatly enjoyed working with them and learning from them.

I've also received a great deal of support from many people over the years. My family, first and foremost, have been a source of endless encouragement: my parents, who have always believed in me, and my brother, who has been one of my greatest sources of support, especially when I've started to have doubts about completing my PhD. I've also had many wonderful friends and colleagues: too many to name, but especially Lauren Hinkle, Lizzie Mamantov, and Rob Goeddel for helping me through some difficult times; and Erik Brinkman, James Kirk, Lynn Garrett, Elaine Wah, Nilmini Abeyratne, Zach Musgrave, Sai Gouravajhala, Ameer Rahmati, Eric Crockett and many others who have all provided a great deal of support and advice over the years.

I haven't had the opportunity to work with everyone in my lab, but I've had many great conversations over the years with many of them, and I've enjoyed working with them even if I haven't formally collaborated with them. In addition to the people I've worked with, Mehrdad Moradi, Jeremy Erickson, Yikai Lin, and Tracy Zhou among many others have all been a great help over the years. The people of SRG and SECRIT have helped me learn an incredible amount over the years, and the people of ECSEL have been a great source of support. Finally, I've gotten to know many great professors here over the years who I haven't gotten a chance to collaborate with directly — in particular, I've enjoyed working with Professors Alex Halderman and Peter Honeyman in my last semester as a GSI. Finally, I'd like to thank the other staff of 388, including David Adrian and all the wonderful TAs, for helping me survive my final semester.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Improving Mobile Network Performance Through Measurement-Driven System Design Approaches

by

Sanae Rosen

Chair: Professor Z. Morley Mao

Mobile networks are complex, dynamic, and often perform poorly. Many factors affect network performance and energy consumption: examples include highly varying network latencies and loss rates, diurnal user movement patterns in cellular networks that impact network congestion, and how radio energy states interacts with application traffic. *Because mobile devices experience uniquely dynamic and complex network conditions and resource tradeoffs, incorporating ongoing, continuous measurements of network performance, resource usage and user and app behavior into mobile systems is essential in addressing the pervasive performance problems in these systems.*

This dissertation examines five different approaches to this problem. First, we discuss three measurement studies which help us understand mobile systems and how to improve them. The first examines how RRC state performance impacts network performance in the wild and argues carriers should measure RRC state performance from the user's perspective when managing their networks. The second looks at trends in applications' background network energy consumption, and shows that more systematic approaches are needed to

manage app behavior. The third examines how Server Push, a new feature of HTTP/2, can in certain cases improve mobile performance, but shows that it is necessary to use measurements to determine if Server Push will be helpful or harmful. Two other projects show how measurements can be incorporated directly into systems that predict and manage network traffic. One project examines how a carrier can support prefetching over time spans of hours by predicting the network loads a user will see in the future and scheduling highly delay-tolerant traffic accordingly. The other examines how the network requests of mobile apps can be predicted, a first step towards an automated and general app prefetching system. Overall, measurements of network performance and app and user behavior are powerful tools in building better mobile systems.

CHAPTER I

Introduction

Thesis statement: *Because mobile devices experience uniquely dynamic and complex network conditions and resource tradeoffs, incorporating ongoing, continuous measurements of network performance, resource usage and user and app behavior into mobile systems is essential in addressing the pervasive performance problems in these systems.* Mobile devices differ from traditional, stationary computers in many respects. In particular, they are more resource-constrained, and rely on lower-performance networks such as cellular networks rather than wired networks. Furthermore, network conditions shift rapidly, in terms of network quality, the network used, user location, and whether there is any connectivity at all. To complicate the situation further, internal phone states (such as the radio state or the app process state) change to compensate for limited energy resources. These often happen in an opaque way, and also modify the power and performance tradeoffs of network activity. We show through five projects how a measurement-oriented approach can be used to address a variety of these sorts of problems on mobile devices.

The projects discussed in this dissertation are summarized in Table 1.1. Each addresses network performance problems from a different perspective, but they are complementary, each addressing how to improve a different aspect of network performance using different types of measurements The first project, *Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks* [103], measures the impact of radio

Main measurements	Main contribution	Performance problem addressed
Improving systems through measurements		
Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks		
Latency around radio state transitions worldwide	Client-oriented measurements → understand RRC states in the wild	Detect new RRC-related latency problems
Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild		
App energy usage, user behavior and other context info	Long-term user study measurements → understand energy problems and how to solve them	Reduce excessive background energy consumption
Push or Request: An Investigation of HTTP/2 Server Push for Improving Mobile Performance		
Server Push performance, overhead, web page network request trends	Improved understanding of Server Push and when/if to use it	Reduce page load time
Time-shifting		
CellShift: A System to Efficiently Time-shift Data on the Cellular Network		
Use network load measurements to schedule delay-tolerant traffic	A system to time-shift data over hours to smooth network loads on the city scale	Avoid network congestion and associated costs
Predicting App Network Traffic to Facilitate Prefetching		
Network traffic patterns: infer what to prefetch	A method of predicting network traffic and an evaluation of the challenges of predicting traffic for prefetching	Prefetch for reduced network latency

Table 1.1: Summary of work supporting thesis statement and main research contributions

resource states in the wild to understand how these states and state transitions affect user-perceived performance. To conserve power while ensuring good performance on resource-constrained mobile devices, devices transition between different *Radio Resource Control* (RRC) states in response to network traffic and according to parameters specific to network operators. As RRC states significantly affect application power consumption and performance, it is important to understand how RRC states and network traffic interact.

In this project, we show that the impact of RRC states on performance is significantly more complex and diverse than found in previous work. We introduce an open-source tool for measuring RRC states as they affect users, in terms of their impact on network and application performance. We deploy the app in 23 countries around the world and collect data on a broad range of unmodified user devices and cellular network technologies. By examining what the end user devices experience, we detect previously unknown performance problems. These problems create network latencies of up to several seconds, and for LTE can increase packet losses by an order of magnitude. Examining these transitions through cross-layer analysis, we determine that the highly complex state transitions of certain carriers, and in particular poor interactions between state demotions and network traffic, can lead to substantial, unexpected latencies. Overall, our client-oriented measurements allow us to gain a better understanding of the complexity of RRC states. By examining RRC state transitions from the client perspective for the first time, we demonstrate that the interactions between client traffic and radio resource states is far more complex than previously described. Through ongoing performance measurements, we were able to monitor and observe previously unknown radio performance problems. We recommend that carriers make use of a similar system when managing their networks.

The second project, *Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild* [104], also evaluates the impact of RRC states and RRC state transitions (among other things), but in the context of how application traffic interacts with these transitions and the way in which these interactions impact how much energy is consumed. This project

is a two year user study of app energy usage on real user devices where client-based measurements were used to understand the behavior of users and applications and how the resulting network traffic interacts with RRC states to consume energy.

Energy consumption due to network traffic on mobile devices continues to be a significant concern, especially background traffic, which is responsible for about 84% of network energy in our study. Through our client-based measurements, we discover a new energy consumption problem where foreground network traffic persists after switching from the foreground to the background, potentially leading to unnecessary energy and data drain. Furthermore, while we find some apps have taken steps to improve the energy impact of periodic background traffic over the last few years, energy consumption differences of up to an order of magnitude exist between apps with very similar functionality. Finally, by examining how apps are used in the wild, we find that some apps continue to generate unneeded traffic for days when the app is not being used, and in some cases this wasted traffic is responsible for a majority of the app's network energy overhead. We propose that these persistent, widespread and varied sources of excessive energy consumption in popular apps should be addressed through new app management tools that tailor network activity to user interaction patterns and that make use of active measurements of application behavior.

Next, we look at another approach to improving performance on mobile devices by measuring performance trends, this time focusing on web browsing rather than applications, and latency rather than energy. *Push or Request: An Investigation of Server Push as a Means to Improve Mobile Performance*[1], investigates the performance impact of HTTP/2's Server Push. Server Push is a new technique that is supposed to reduce network latency when loading web pages by pushing additional content in response to the first request for the initial HTML page before the additional content is explicitly requested. However, the performance benefits of this technique were previously not known. This study demonstrates that Server Push performs best when there is moderately high latency or packet loss

[1] In submission to WWW '17

rates, motivating its use on mobile networks. Furthermore, there are slight energy benefits to using Server Push, especially on LTE. However, in many cases Server Push can actually degrade performance. Based on these findings, we recommend the cautious use of Server Push for mobile websites, but only after carefully testing the performance of Server Push on the site in question. This study motivates the use of ongoing measurements in conjunction with Server Push deployments to ensure Server Push gives the expected performance benefits.

We then continue to look at how shifting when we transmit traffic can address performance problems in the cellular network, focusing on shifting traffic on larger time scales. Because mobile networks are heterogeneous and change significantly throughout the day, we show that prefetching and time-shifting could be used to take advantage of changes in network connectivity over periods of hours. In *CellShift: A System to Efficiently Time-shift Data on the Cellular Network* [102], we examine how a long-term time-shifting system can schedule large volumes of highly delay-tolerant data in a changing network with minimal additional load on the client. Recent work [113] has shown that users are interested in time-shifting certain types of data by several hours in exchange for discounted data. Such a system could enable new, innovative, data-heavy and delay-tolerant services, such as a prefetching system that can load entire TV shows onto user's devices in advance of them being watched.

To examine how this approach would work on real cellular networks, we evaluate network load patterns from a major ISP in a major metropolitan area, demonstrating that there are substantial variations over time and between base stations that can be leveraged through time-shifting. Furthermore, we show these variations can be predicted hours in advance, even for individual, highly mobile users. We present *CellShift*, a scalable time-shifting framework that leverages these forecasts to efficiently schedule requests for millions of users. Through a city-scale simulation using real load data, we demonstrate CellShift can reduce the impact of a variety of data-heavy, highly delay tolerant traffic loads on peak base

station loads by 50-76% in most cases, and allow today's cellular networks to support an increase in demand by 40% or more even in heavily congested cities. A prototype implementation shows CellShift is scalable and efficient, in particular adding no more than a 2% battery overhead to the device.

Finally, we continue looking at how to address performance problems through prefetching, in particular looking at how to make prefetching to mask performance problems in mobile devices a reality by addressing the problem of predicting what to prefetch. For the final project, *Predicting App Network Traffic to Facilitate Prefetching*, we examine how effectively the requests made by an app can be predicted, with the goal of building an automated prefetching system where a proxy makes prefetching decisions for unmodified apps. We start by showing that app *Activity* transitions (transitions between different UI screens) can be predicted easily and accurately, as usually only a small set of transitions occur with any regularity. We then look at apps structured around populating relatively static *Activities*, such as social media apps or news apps, and find that about 60% of URLs for these apps can be predicted by a prefetching system that models how apps generate traffic requests. We then examine several other challenges, such as the data overhead, bandwidth requirements, and the time to download content to be prefetched, and determine that several challenges must be overcome to build such a prefetching system. While fully automated prefetching remains an open research project, our prediction system based around observing network traffic trends is a major step in the direction of developing such a system.

Overall, this thesis shows there are many ways in which measurements of mobile systems can improve network performance, and in this thesis we envision a network where from the web server to the device, measurements are used to make more informed decisions, as shown in Figure 1.1. In addition to being complementary elements of a measurement-driven mobile network, there are several other themes among these projects tying them together. Three of the projects focus more specifically on client-oriented measurements: *RRC State Inference*, *Network Energy Efficiency*, and *Server Push*. Measuring

the performance experienced by the client allows the impact of existing systems on power and performance to be understood, and can inform how we build systems in the future. The remaining two projects, *Cellshift* and *App Traffic Prediction*, focus on determining how to build systems that make use of measurements directly and actively. The first set of projects also motivate the need for dynamic systems that make use of the measurements we use, though. Taking this more active approach would be the next step for that first set of projects. Another common theme is using measurements to change when content is sent to mitigate performance problems, in *Server Push*, *Cellshift* and *App Traffic prediction*.

Overall, these projects are complementary. The *RRC State Inference* project allows networks to be configured optimally and can work in conjunction with any of the other projects, and *Network Energy Efficiency* 's insights about background power consumption informs the designs of other systems that use periodic measurements, such as the *RRC State Inference* project and *Cellshift*. *Cellshift* and *App Traffic Prediction* target time-shifting different types of traffic and are thus complementary, and could even be incorporated into one cohesive carrier-facilitated prefetching system. *Server Push*, which shows Server Push would benefit from being run inside a proxy, could also be incorporated into such a comprehensive prefetching system. Overall, these five projects can be viewed as pieces of a new paradigm for managing network traffic.

1.1 Common Research Challenges

All of these research projects face similar challenges in terms of dealing with unique aspects of how mobile apps, devices and networks function, but these challenges also introduce opportunities to leverage measurements in order to build more intelligent systems that can improve performance. There are three broad categories of these challenges which we address for these projects: the dynamic nature of network, device and user behavior; the fact that devices are resource-constrained and the overhead of measurements can be significant; and the problem of collecting accurate mobile measurements.

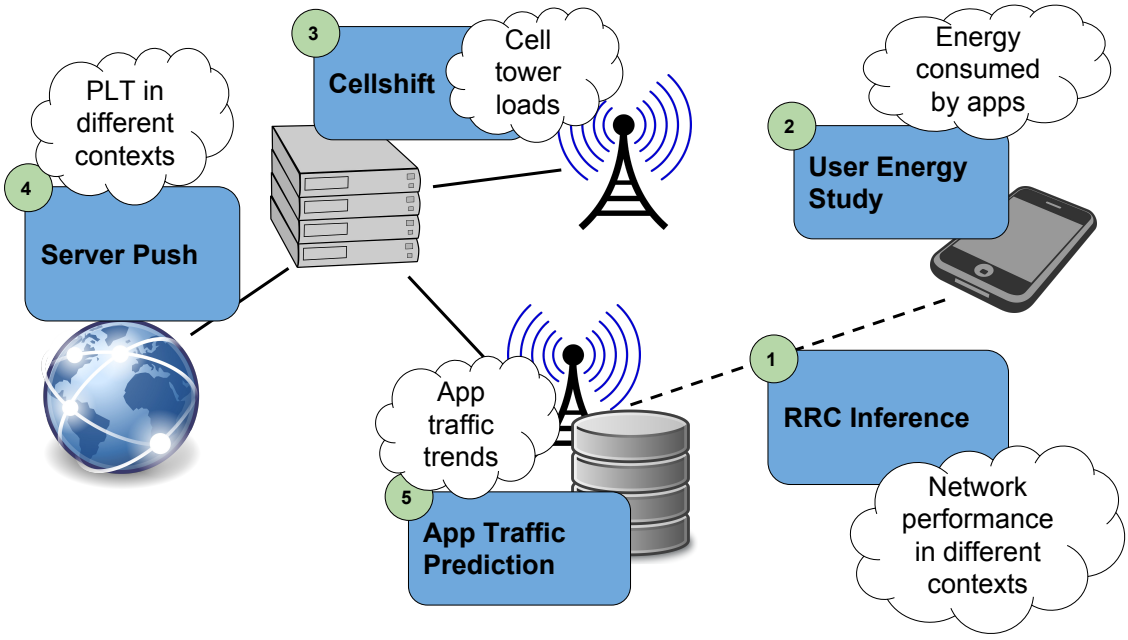


Figure 1.1: Overview of Research Projects

Dynamic mobile requests: Dealing with the dynamic nature of mobile devices and networks is one of the main challenges for all of these projects. There are three main aspects of dynamic networks that posed challenges. First, the RRC state machine means that the energy consumption and latency of cellular network traffic can be different at different points in time. Second, user mobility and dynamic trends in user traffic in particular complicate the *Cellshift* project. Finally, app traffic patterns introduce challenges and opportunities for building better mobile systems: the trend for apps to make requests in the background complicates the problem of managing application energy, but the relatively well-structured nature of certain types of app traffic is something which we are able to leverage.

Contending with the RRC state machine is the main focus of the *RRC State Inference* project, evidently, but also the *Network Energy Efficiency* project. The cellular network is unique in that devices shift between a variety of states with different power and performance tradeoffs, moving to high-power, high-performance states in response to network traffic, and lower-power, lower-performance states after timers set by the carrier expire.

We give more background in §2.1.1. As we find in the *RRC State Inference* project, these state changes can also introduce significant performance problems. For this project, this complex series of state transitions and their performance tradeoffs are the main target of our investigation.

For the *Network Energy Efficiency* project, the RRC state machine is also a major focus of our analysis. Traffic patterns on mobile devices, and how they interact with the RRC state machine, have a major impact on energy consumption. Network traffic that occurs periodically tends to have a high cost, as the radio must be woken up for several seconds whenever traffic is sent. Batching traffic, conversely, tends to lead to better energy consumption. It turns out that how traffic interacts with the RRC state machine is one of the main factors impacting network energy consumption.

The second challenge related to dynamic networks is user mobility and the dynamic nature of demand for network resources, which is of particular concern to the *Cellshift* project. *Cellshift* primarily leveraged variations in network traffic over time and between locations. These variations represent inefficiencies in how the cellular network is utilized, as the network must support the peak load at each location even if that peak load occurs only for an hour a day, since network capacity must be built for that peak load. The goal of this project is to address these inefficiencies by shifting traffic to other times and locations. However, these variations reflect the fact that users tend to be mobile, and building a system that can adapt to changes in network load when scheduling traffic, as well as adapt to user mobility, were major challenges of this project.

Finally, the unique nature of network requests on mobile devices also offers opportunities and challenges. The *Network Energy Efficiency* project is based on the fact that apps tend to have a significant amount of background traffic, something which is not a concern in the same way for desktops where power is less of a concern. Detecting and understanding this background traffic was a major focus of the study. For the *App Traffic Prediction* project, we are able to leverage the fact that mobile apps have unique approaches to net-

work traffic: a large class of apps have well-organized approaches to loading content based on well-structured files that specify what future content to load, allowing us to model and predict what to load.

Resource-constrained devices and measurement overhead: A major challenge in working with mobile devices is that they are quite resource-constrained. We have to worry about power consumption as well as data usage when designing measurements.

One major problem when collecting measurements is that periodic network traffic, as we discuss in §2.1.2, is a harmful pattern for network traffic on the cellular network. But in these projects, particularly the *RRC State Inference* project and the *Network Energy Efficiency* project, ongoing measurements collected on the phone needed to be transmitted to a central server. We decided to reduce how up to date the information collected would be to save energy, and uploaded data once a day on WiFi. For the *RRC State Inference* study, we started off collecting two measurements a day and reduced to one measurement a day. As a result, it took about a year to get enough data from enough users for us to properly analyze the results (also due to the slowness of advertising the app to users), but we could be sure we weren't negatively impacting the user.

This also informs the design of the entire *Cellshift* project. Ideally, we would have the cellular network constantly communicating with the device to inform it when to transmit data, but the energy overhead would be prohibitive. As a result, determining how to built a system that can schedule data in an intermittent way was a major challenge for this project.

Another concern with the *Cellshift* project and other projects is the overhead at the server. While not unique to mobile measurements, when collecting measurements at a large scale, reducing the storage needs and networking needs at the server is also important. For *Cellshift* , limiting the amount of data that would need to be stored, as well as making sure that most decisions could be made in a relatively decentralized manner, are important design constraints. For *App Traffic Prediction* , we examine the limitations of storing and moving data from cloudlet to cloudlet and how that would impact the feasibil-

ity of a cloudlet prefetching deployment. For *RRC State Inference* , limiting the amount of data that would be collected in a global measurement study is an important consideration.

Accurate mobile measurements: There are various factors that make it challenging to collect accurate mobile measurements, and with any measurement study, there are challenges in ensuring that high-quality, useful measurements are collected.

For the RRC state measurement system, a major challenge is how to accurately measure RRC states in the wild. It is necessary to first collect data on the phone about whether the device is being used at the time and about the current type of network. Measurements can be scheduled accordingly, and discarded if needed if we detect that other packets were sent while the tests are performed, as background traffic affects the accuracy of the results. While less of a fundamental challenge for the user study project, it is also necessary to collect a significant amount of context data for this project, such as on the running state of each application, to properly interpret network traffic and energy measurements and understand how applications behave.

This is also a challenge for *Server Push* . It was not possible to collect measurements on real servers, as there are not enough servers running Server Push. It is thus necessary to mirror content as well as to construct artificial websites for controlled experiments to explore how various factors impact Server Push performance. For the *Cellshift* and *App Traffic Prediction* projects, the challenges are more in making use of the measurements effectively. For the *Cellshift* project, determining how to make use of measurements of network load is one of the major challenges, especially in predicting load for users as they move around. For the *App Traffic Prediction* project, the challenge is in making use of network log data in order to accurately predict network load.

1.2 Contributions

Next, I will summarize the research contributions in this thesis, organized by research project.

1.2.1 Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks

In this project, we examine how client-based measurements that can help us understand RRC states. In particular, our contributions are:

- We develop a measurement approach that allows RRC state machine dynamics to be measured and observed on uncontrolled devices in the wild.
- Using Mobiperf, an app into which we incorporated this technique, we create a large database of RRC state performance worldwide, the only such dataset to date as far as we know.
- We uncover and examine some previously unknown, severe latency problems that can increase round trip times by seconds.
- We demonstrate the degree to which RRC state performance, including the performance problems we uncovered, impact application-layer delays and app QoE, including through crowdsourced measurements.

Overall, our client-based measurements allow for a deeper understanding of complex RRC states and their impact on performance, and we argue that carriers should use our approach to monitor and manage their networks, as these client-based measurements give a different perspective on performance than the information exposed by device manufacturers..

1.2.2 Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild

In this project, we also focus on measurements, but this time instead of measuring RRC states themselves, we examine how apps interact with them. Our main contribution was a better understanding of how apps consume energy due to background traffic that keeps the

RRC radio active, gathered through a measurement study over several years. In particular, our key findings are:

- We discover a new form of excessive energy consumption, where an app continues sending likely unintended background traffic after the app is moved to the background.
- We show there is high variability in how much energy is consumed even by very similar apps, and that app energy usage has evolved differently among apps over time, demonstrating that the adoption of best practices is still not universal.
- We discover that many apps that send background traffic are not used for days. At the time, App Standby and Doze did not exist, and we proposed that there should be some mechanism to suppress traffic from apps that are not currently being used.

Overall, our long-term measurement study helps us understand how apps consume energy and propose concrete guidelines to app developers as to how to build more energy-efficient systems. We argue that active measurements of app energy consumption and behavior are needed to manage these problems.

1.2.3 Push or Request: An Investigation of Server Push as a Means to Improve Mobile Performance

In this project, we measure the impact of Server Push, demonstrating it is likely most suitable to mobile networks. The main contributions of this paper are:

- Determining that Server Push is more effective on high loss or high latency networks, motivating its use on WiFi and cellular networks.
- Determining that pushing all content on a web page is more effective than pushing a few key objects.

- Determining that Server Push can sometimes decrease performance: various web page factors can influence if it's successful, but ultimately web page developers should check if Server Push results in performance benefits through measurement before deploying it.

- Server push leads to power savings on LTE of about 9%, another way it is particularly suitable for mobile devices.

Overall, through controlled, in-lab experiments, we are able to gain a greater understanding of how Server Push can effectively be used to improve performance, particularly for mobile devices, and demonstrate that website developers should incorporate performance measurements when deploying Server Push.

1.2.4 CellShift: A System to Efficiently Time-shift Data on the Cellular Network

In this project, we leverage measurements of network load in order to determine how to schedule delay-tolerant data over time spans of hours. Key contributions include:

- A city-scale examination of eNodeB usage trends over several months, determining that there is a significant amount of underused capacity on today's cellular networks.

- A method of predicting network loads on a per-eNodeB basis 15 minutes in advance (with an accuracy of 2% of the eNodeB's total capacity), and predicting loads a user will experience in a location-agnostic manner up to a day in advance (with an accuracy of 8% of an eNodeB's capacity).

- The design and evaluation, in simulation, of a highly scalable system that schedules data over time scales of hours on resource-constrained mobile devices.

1.2.5 Predicting App Network Traffic to Facilitate Prefetching

In this project, we determine how to effectively predict network traffic from mobile applications, using cloudlet-based prefetching as a motivating example. The key contribu-

tions are:

- A method for predicting URLs and their parameters in advance for a large class of applications, by observing prior network traffic and leveraging common app patterns.
- An evaluation of the accuracy and overhead of predicting URLs to facilitate prefetching: in particular, that about 60% of traffic can be predicted, but with a overhead of about 150% in terms of unneeded requests.
- The identification and examination of the challenges in making automated prefetching based on this approach feasible, including examining the storage overhead, the cost of migrating application state and the time to download content.
- The finding that application entry points and Activity transitions are highly predictable: for most apps, 50% of Activity transitions from one Activity go to the same next Activity.

Overall, we show how, by examining network traffic, we can build the prediction framework that would be needed for a cloudlet prefetching system that could reduce application latency.

CHAPTER II

Background

2.1 The cellular network

The cellular network is unique in many ways. The cellular network architecture reflects the fact that users are highly mobile. Also, there are unique ways that network radio energy management features inform the design of any work involving cellular devices. Users move around without switching networks, and the network allocates resources to users dynamically, while seamlessly transferring users between cell towers. I discuss two network types in this dissertation: 3G, and 4G LTE, the latter being more the more recent cellular network technology. I will first summarize the LTE network, drawing from "An Introduction to LTE" [24]. I then focus in more depth on RRC states and their impact on energy consumption.

An overview of the cellular network is shown in Figure 2.1. Users connect to the network with mobile devices, commonly known as UE (User Equipment). They connect to the cell tower (base station or eNodeB), which conveys both data and control plane messages to and from the device. In general, latency over the channel between the UE and eNodeB is a concern, and transmitting data over this channel has substantial energy costs, described in the section on RRC states below.

Furthermore, each eNodeB can only support a certain amount of traffic. Resources are allocated to devices in the form of *Physical Resource Blocks*, or PRBs. The amount of data

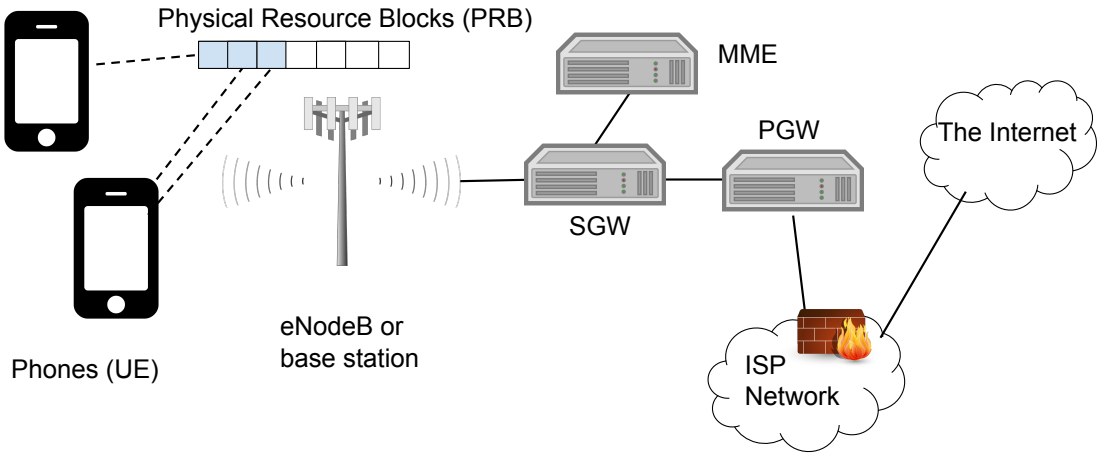


Figure 2.1: (Simplified) overview of the cellular network.

that can be sent for a given number of allocated PRBs is a function of the signal quality. The cost of sending data over this last hop is often a major overhead for the carrier. In particular, the cellular network in a given place has a fixed amount of capacity, and the cost of supporting network traffic at a location is related to the *peak load* at that location, rather than the average load. This fact is a major motivator for the *Cellshift* project.

The remainder of the network is less critical for understanding this dissertation. There are a variety of middleboxes: P-gateways, which connect the cellular network to the rest of the carrier's network; the S-gateway, which forwards data from the base station to the P-gateway and the Mobility Management Entity (MME) that manages the mobile devices. More importantly, the carrier also introduces a large number of middleboxes after the P-gateway and before the traffic reaches the outside world, such as NATs and firewalls [130], giving the opportunity to introduce other sorts of proxies as well.

2.1.1 RRC states

For cellular networks, there is a tradeoff between latency and battery consumption. Mobile devices do not maintain a constant, active network connection due to their limited battery life. Switching to an active connection, however, introduces delays, so mobile

devices switch between network states based on when network traffic is sent. Figure 2.2A gives a conceptual overview of these states as well as two common implementations, for LTE and 3G. When data needs to be sent, devices switch to a high-power, active state. This transition incurs additional delays, so to avoid making this transition more often than needed, since network traffic is often comes in bursts, the device remains in this state for several seconds. That way, only the first few packets in a burst are delayed by the state promotion. A timer determines how long the device waits for additional traffic before falling back to a lower power state. This timer, the *demotion timer*, is usually fixed, and set by the carrier. There may also be an intermediate state where small amounts of data can be transmitted without the high power consumption of the fully active state.

These are known as *RRC States*, and are defined by 3GPP specifications [4, 5]. Each carrier configures their RRC state machine timers, subject to the constraints of the protocol specification. For 3G network technologies [4], there are two to three main states. The first is DCH, which is high-power and high-bandwidth. FACH, an optional state, is lower power and can only transmit small amounts of data before needing to switch to DCH. Finally, in PCH, no transmission is possible and a state promotion is needed before sending data, but very little energy is consumed. An example of a 3G state machine is shown in Figure 2.2B. There can be slight variations in the possible state transitions depending on the carrier. For 4G LTE, as shown in Figure 2.2C, there are two main states: CONNECTED, a higher-power state, and IDLE, a lower-power state where no data is transmitted. The former may be broken down further into smaller sub-states. For these sub-states, the connection is active at regular intervals of tens or hundreds of milliseconds. This is known as *Discontinuous reception*, or DRX.

2.1.2 Application traffic and RRC states

It has been shown that these RRC timers can have a substantial impact on application performance and power consumption [36, 35, 63]. In particular, periodic messages may

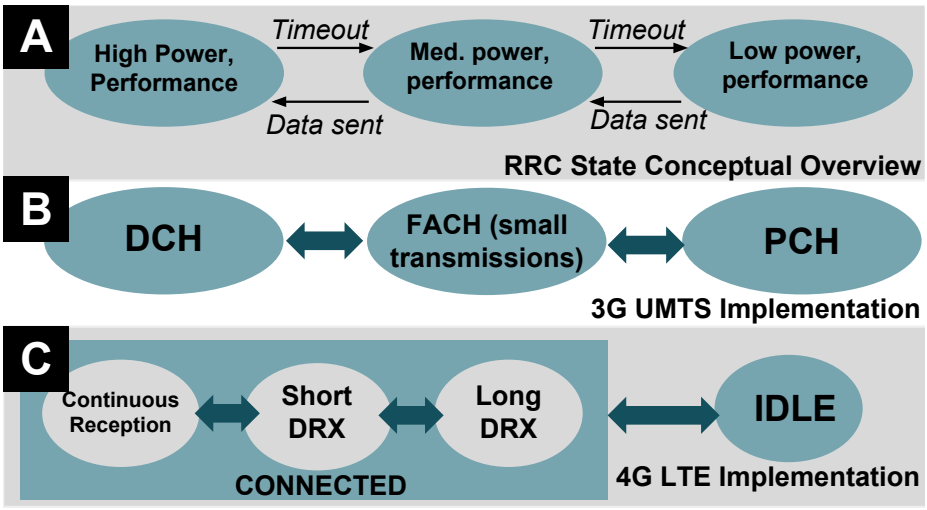


Figure 2.2: A: Overview of RRC state machine design. B-C: possible 3G and 4G state machines.

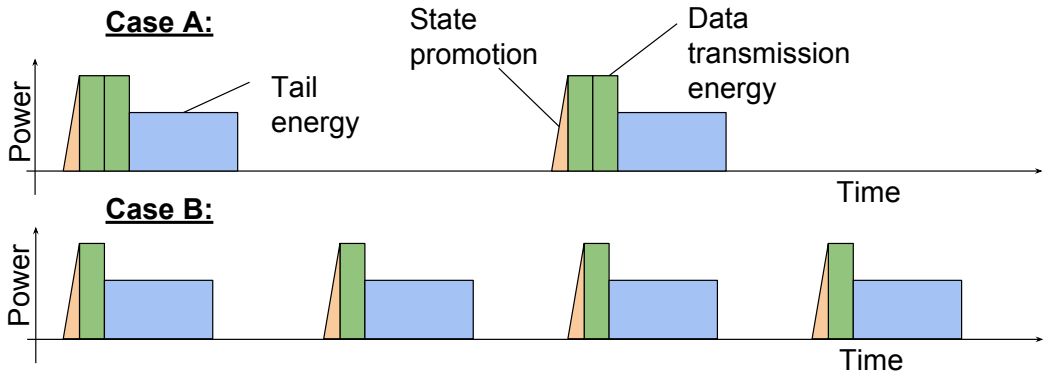


Figure 2.3: Comparison of energy consumed for the same amount of data with different traffic patterns.

be affected by long promotion latencies as well as lead to the device being in a high-power state longer than needed. Essentially, every state promotion incurs a performance penalty due to the state promotion, and later incurs an energy penalty due to the tail timer, regardless of how much data is sent, as shown in Figure 2.3. For some number of seconds after traffic has been sent (depending on the carrier), the device continues to consume energy, keeping the radio in a high-powered state. Since network transmissions can often be of a similar length to the tail timer, or potentially even shorter, having many network

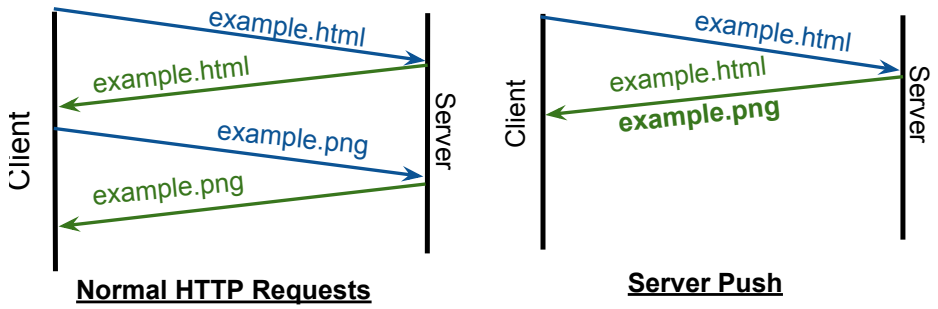


Figure 2.4: A simplified view of how Server Push results in performance benefits.

requests sent separately can cause a substantial increase in the energy consumed. In that figure, substantially less energy is consumed in case A, even though case A sends just as much traffic. By batching traffic and halving the number of bursts, the tail energy is halved as well.

This problem can be addressed in part through *fast dormancy* [45], where the device transitions to a low-power state early when no additional data transmissions are expected. In Chapter IV, we find that fast dormancy is rarely enabled in practice, perhaps due to the added complexity of implementing such a system, and problems with certain implementations to date [84, 10].

2.2 HTTP/2

HTTP/2 has been recently proposed as a replacement to HTTP/1.1, promising to offer better performance, and to address some of the performance limitations of HTTP/1.1 [54, 14]. It builds on SPDY, a protocol developed by Google [95].

HTTP/2 differs from HTTP/1.1 in many ways. One feature is that HTTP/2 connections now consist of a number of *streams* in a single HTTP/2 connection. Multiple streams can be open concurrently, and the *frames* of the streams, which are units of data, can be interleaved. Streams can be open or closed by either endpoint. Notably, this means fewer TCP connections are required for a HTTP/2 session, which has performance advantages

and disadvantages [127]. Other features include giving each stream a priority, indicating that more resources should be allocated to some strings; explicitly indicating dependencies between streams; and mandatory encryption.

One of the more interesting new features in HTTP/2 is Server Push. We focus on this feature primarily in Chapter VI. As shown in Figure 2.4, in HTTP/1.1, even if the server knows what content the user will need, the content needs to be requested by the user. First, the user fetches the initial HTML page, then parses it, and looks for additional content that will be needed. Then, it requests that content. It is possible that due to additional HTML files or javascript that further round trips would be needed for complex pages. Overall, this means there can be substantial delays to fetch all the images on a page.

Server Push assumes that in many cases, as the server generates the content the user will see, the server knows what the user should fetch. HTTP/2's stream-based construction supports streams being opened from either direction, and thus the server can initiate a connection where data the server things the user will need is sent to the user. The step of having the user parse the initial HTML page and request further content can thus be eliminated, which should lead to performance improvements. We examine this in Chapter VI.

CHAPTER III

Related Work

The projects described in this thesis build on a substantial amount of prior work, which has been organized into seven categories. We start by discussing past work on measuring browsing performance generally, followed by discussing some systems that aim to improve browsing and app performance. Next, we focus on RRC states, background traffic and the cellular network, including both measurement studies and proposals of how to better work with RRC states.

Then, we discuss more specific themes that pertain to specific chapters. As background to the *Network Energy Efficiency* project, we discuss prior work on understanding background traffic and dealing with energy consumption due to that traffic. As background to the *Cellshift* project, we discuss work on traffic forecasting and characterizing how cellular networks are used on a large scale, followed by a section on long-term prefetching. For Server Push, we examine other research on next-generation network protocols. Finally, we examine how cloudlets, which are used as a motivating example to the *App Traffic Prediction* project, have been discussed in the past.

3.1 Measuring Factors that Impact Browsing Performance

First, we discuss general related work on browsing performance, before focusing on more specific aspects of performance in later sections. Work by Qian et al [97] examines

how existing websites are designed and the impact on performance and resource usage. They develop a tool to measure website performance in the lab through cross-layer analysis and find that a wide range of factors impact power and performance, such as the use of Javascript causing the radio to be on for too long. Work by Wang et al [129] examines sources of delays in mobile web browsers by instrumenting the browser directly. They find that at the time the paper was written, object loading times are still the bottleneck, but that device CPU power still has a major impact due to the overhead of networking operations and other OS operations. Work by Butkiewicz et al [15] examines how website complexity impacts performance, developing a series of metrics to do so, and evaluate how they impact loading time. They find that the number of objects to load is a good predictor of network loading time, more than the size of the object.

Other work examines longitudinal trends in browsing performance: work by Nikravesh et al [88] examines longitudinal trends in mobile network performance generally, and find that in addition to temporal and geographic trends in performance, that there is a significant degree of instability in performance that cannot be attributed to these factors. Work by Imh et al [60] characterizes websites, looking at five years of network data from 70,000 users around the world and examining how network traffic has evolved. They discover Javascript and video are increasingly important factors, that half of traffic is not due to the initial page load, and that content-based caching rather than object-based caching would be valuable. We make use of a longitudinal approach in our measurement studies, focusing on different problems.

Thiagarajan et al [122] examine what types of browsing content leads to higher energy consumption. They find complex Javascript and CSS, as well as certain image types, are a problem, and propose a series of concrete guidelines to building better websites. Gember et al [42] discuss the challenges of accurately measuring cellular networks from actively used devices. They find that it is best to measure network performance at times when the user uses the device, which unfortunately was not possible in *RRC State Inference* but which

motivates measuring app energy consumption in the context of a user study. Work by Xu et al [133] measures country-wide trends in app usage, including the types of apps used, the times of day, and the context in which they are used. They find a substantial number of apps are only used locally, that use of certain apps is correlated, that many apps have clear diurnal traffic patterns, and that some apps are more likely to be used while mobile, all findings that could inform how carriers manage traffic. Unlike these projects, we examine different factors that impact browsing performance: we focus on the impact of Server Push on browsing, or propose systems such as *App Traffic Prediction* to improve performance.

Other work looks at networking performance. Work by Sundaresan et al [119] investigates residential broadband, and determines that the round trip time is still a bottleneck, motivating the *Server Push* study. Their solution is to place a cache in home routers, a similar concept to the cloudlets discussed in the *App Traffic Prediction* project. Work by Zaki et al [135] examines factors affecting network performance in a developing country, and find that a lack of CDN and caching infrastructure is a major factor in poor performance. In particular, relevant to the *Server Push* project, they find that SPDY[1] works particularly well in this context. Work by Halepovic et al [29] presents a method of passively measuring HTTP time to first byte from within the packet core, whereas previously expensive active probing is needed. This tool could help carriers better manage their network. Work by Narayanan et al [81] observes that content is often poorly distributed among CDNs. They also observe that a significant percentage of content is served from CDNs, which may have implications for Server Push deployment by third parties. These papers are complementary to our examination of Server Push and RRC states, addressing different aspects of network performance

User studies and app based measurement studies, such as those in Chapter IV and Chapter V are approaches that also have a substantial amount of related work. The Livelab project [18] makes use of users running a measurement app on their phones. A wide range

[1]Not an acronym — a proposed replacement to HTTP

of findings on how users interact with mobile devices have been published based on this project, including measuring web usage in the wild [17]. In that study, they find substantial differences between mobile web usage and desktop web usage, with less usage of the browser, and a higher dependence on search, which might inform how browsers on phones are designed. JamLogger [90] is an ongoing project to collect general performance and user activity on mobile devices — to date, there are no related publictions. "Diversity in Smartphone Usage" [37] examines how users use the phone, such as for how long or how much data they use, and find a large amount of diversity in all of these characteristics, with differences of orders of magnitude between users. They argue that as a result mechanisms to improve performance for users should be tailored to individual users, and demonstrate that energy prediction can be done in such a manner. These studies highlight the importance of user studies in understanding performance in the wild. Unlike these papers, we focus specifically on using user studies and measurement studies to address two problems in this thesis: network energy consumption and RRC state performance.

3.2 Building systems to improve performance

Next, we discuss approaches to building systems to improve browsing performance. WProf [126] finds performance dependencies in browsers through an in-browser tool, and finds that computation is a major part of the critical path and that caching is not necessarily helpful, and that SPDY is not helpful with low RTTs. Follow-up work [82] extends this analysis to mobile phones. This allows for a better understanding of the factors impacting mobile performance: they find that a major cause of slow mobile browsing is the computation overhead, that elements on the critical path may differ between the mobile and browser version of the page, and that mobile performance only really suffers for large pages. WebProphet [75] predicts page load times from object dependencies, and infers object dependencies using a novel technique based on timing perturbations. As a result, it is able to recommend simple approaches to optimizing web page loading performance. Po-

laris [83] makes computation more efficient by better detecting dependencies and scheduling requests through a client-side scheduler in a way that minimizes round trips. This approach is able to reduce page load times by 34% in the median case. The projects focused on system building in this thesis are complementary to these papers and use different techniques: long-term time-shifting for *Cellshift* and prediction of network traffic for *App Traffic Prediction*.

Using proxies to improve network performance is another common theme. Parcel [117] splits browsing functionality between a proxy and mobile browser to improve performance. The proxy downloads objects and pushes them to the browser, and interactive functionality happens entirely on the client where possible. As a result the page load time and energy consumption of loading a site is substantially reduced. Flywheel [6] is a compression proxy for mobile devices used in the wild which halves the size of mobile pages. It also applies other optimizations such as SPDY, and the paper discusses the engineering challenges of implementing a complex proxy at scale, concerns that would need to be addressed for projects such as the *App Traffic Prediction* project. They find overall that the performance benefits of this approach are mixed, and work better for larger pages. Flexiweb [116] finds that compression proxies can sometimes harm performance depending on network performance, and dynamically adapts proxy optimizations based on network conditions and website characteristics. By adapting to network conditions, they are able to greatly improve the performance of a compression proxy. A similar method of dynamically applying optimizations would be highly valuable to Server Push, as we show in this thesis.

Klotski [16] examines automatically detecting dependencies and scheduling object downloads using a proxy to meet arbitrary priorities. A back-end proxy analyzes the dependencies of a page and a front-end prioritizes the content, doubling the amount of high-utility content delivered early. We apply some similar methods to generalizing dynamic URLs in the *App Traffic Prediction* project. Shandian [128] optimizes the order and manner in which content is loaded at the granularity of HTML or CSS elements, using a proxy server. They

focus in particular on eliminating unneeded HTML and CSS content during the initial load, and find significant performance benefits of 50 to 60% on mobile devices. We propose the use of proxies in different contexts: for *Server Push*, to facilitate Server Push, and for *App Traffic Prediction*, to predict network traffic to facilitate prefetching.

3.3 RRC States and the Cellular network

Next, I discuss related work that gives background information on the cellular network and examines the impact of RRC states on browsing specifically. This is most relevant to the RRC Inference and Energy User Study projects, but most of the projects I discuss deal with the cellular network. Work by Huang et al [57] examines LTE in depth, with a focus on poor interactions between the cellular network and TCP flows. They argue that transport control mechanisms that are LTE-friendly are needed, as currently TCP often operates at below half of the maximum bandwidth and spends a lot of time in slow start. Sani et al [108] examine trends in data consumption by app. They find substantial differences between even very similar apps — similarly, we find major differences in energy consumption in the *Network Energy Efficiency* project. They suggest that users could use these findings to select an appropriate data plan based on the data they use. Work by Huang et al [58] measures the performance of 3G and its difference between devices using crowdsourcing, as well as controlled analysis on a variety of devices. They find substantial differences between the performance of carriers and devices — the main contribution of this paper is a comprehensive evaluation of the performance of 3G in a range of circumstances. This work is complementary to this thesis, as we focus on measuring other factors of performance such as the impact of RRC states or the background energy consumption of apps.

Other work has focused specifically on RRC states, a common feature in much of the work in this thesis, especially the *RRC State Inference* chapter. Previous work examines power and performance characteristics of RRC state machines in both 3G [35] and 4G LTE networks [63, 70] in controlled environments, as well as specific features of those

networks such as DRX [65]. The work on 3G [35] introduced a method of probing RRC state machine parameters which we adapted in the *RRC State Inference* project, and through trace-based analysis determine that state promotions cause substantial delays and that RRC state timers are suboptimal for many applications, motivating the use of variable RRC state timers. The work on 4G LTE networks [63] also characterizes the power and performance characteristics of LTE. It first generates a power model for RRC states and analyzes the power efficiency of LTE, and by examining the performance of applications, determines that with LTE computation is less of a bottleneck. Work by Zhou et al [70] examines DRX specifically, building a model of DRX performance and power consumption, and showing that different parameters for the length of time in DRX and the DRX frequency are suitable for different types of applications.

A web page by Souders [106] estimates RRC state machine performance through a web app, at a coarser granularity than we do and without accounting for background network activity on phones. This demonstrates that there is interest in RRC state machines outside the academic community, though. RILAnalyzer [123] monitors 3G RRC state transition events and measures directly how applications cause excessive RRC state promotions by leveraging chipset-specific functionality. They find RRC state transitions in the wild are more diverse than have been measured in the lab, and find that applications in the wild often interact poorly with RRC state transitions. Examining lower-layer control messages specifically, a Qualcomm whitepaper [77] explains how control plane messages in different network technologies are expected to lead to different promotion latencies. In the *RRC State Inference* chapter, we focus on RRC state transition dynamics: measuring and understanding how RRC state transitions vary and cause different performance trends on different carriers, and in particular non-ideal transition behavior.

3.4 Background Traffic

Prior work also examines various aspects of background network activity, including how it interacts interacts with RRC states. Aucinas et al. examines smartphone energy efficiency through in-lab experiments with a number of major apps which maintain a continuous online presence [11]. They find that these apps have a disproportionate energy impact due to interactions with the RRC state machine, and propose the use of push notifications instead. Earlier work by Qian et al [98] identify that periodic traffic is a general problem, and through an analysis of 1.5 billion packets, finds that despite being less than 2% of traffic overall, make up 30% of the radio energy. They find that by flexibly scheduling this periodic traffic, almost all of its energy impact can be eliminated. Addressing the problem of periodic but likely delay-tolerant traffic, Tailender [79] designs a scheduler that prefetches and batches delay-tolerant traffic and shifts the traffic to WiFi where possible in order to reduce network energy consumption. Tamer [76] demonstrates it is possible to modify the energy impact of app wakeups by interposing on wakeup events to better manage them by interposing on wakeup events, and show they can address many energy bugs with their system. Our work is complementary to these papers, examining a broad set of apps and focusing on their behavior in the wild over a long time period, which enables us to uncover new energy drain problems as well as understand how old problems have evolved.

More specifically related to the proposals in the *Network Energy Efficiency* project, there has also been a great deal of interest in the impact of background traffic in concurrently developed work. In particular, Google announced Android M after the submission of the *Network Energy Efficiency* paper, which introduces Doze and App Standby, which should decrease the energy impact of the excessive background traffic we uncovered in Chapter IV [53, 28]. Other concurrent work includes ZapDroid [115], which automatically isolates or disables infrequently-used apps, while allowing them to be easily restored when needed. They also examine the impact of these apps, finding that they can be responsible for as much as 20% of the network energy consumed in a day. Our findings in the *Network*

Energy Efficiency project suggest this approach would be highly valuable. Work by Chen et al [21] presents a large-scale user study of 1520 users of how users use their phones and how that interacts with app battery consumption. They find that almost half of energy is consumed when the screen is off, and that cellular energy is a major factor, among other things, comprehensively breaking down power consumption according to various factors such as the device types. These findings could potentially inform app design and carrier decision making in the future. Their study is complementary, covering all sources of energy consumption and trends across categories of devices, whereas we focus on examining the role of background network transfers specifically in depth and exploring the root causes of this excessive background consumption.

There has been a great deal of interest in understanding how applications can improve performance by accounting for RRC state timers, especially based on the observation of temporal clustering of network traffic. ARO [36] presents a tool for optimizing application performance, through cross-layer analysis, which accounts for poor interactions between applications and RRC state machines. Using this tool, they are able to provide concrete recommendations to six popular apps as to how to improve their energy consumption by addressing problems such as loading scattered content while scrolling. TailTheft [50] prefetches or delays traffic to reduce the amount of traffic sent in high-latency states by scheduling this content during the tail timer without pushing the tail timer back, reducing energy consumption by over 20%. Work by Lagar-Cavilla et al. [71] also proposes transmitting delay-tolerant traffic immediately after other data transmissions, and providing an API in order to allow apps to indicate what is delay-tolerant.

Work by Evensen et al [34], conversely, focuses on the latency impact of RRC states and provides a mechanism by which applications can request a state promotion in advance, thus avoiding the promotion latency. This approach allows almost all promotion-inducing requests to take place at least two seconds faster. RadioJockey [94] uses network traces to predict network traffic patterns and optimize when to trigger fast dormancy to both avoid

keeping the radio active unnecessarily and to avoid excessive signalling from unneeded state promotions. It is able to reduce energy consumption by up to 40% while minimizing added promotion overheads. Work by Deng et al. [105] propose a method of scheduling data transfers to minimize energy consumption without impacting user-perceived performance. They use statistical analysis of network traffic to determine when to make the cellular radio idle and when to delay a networking session by a few seconds to allow for traffic to be batched. All of this work motivates the need for an approach like *RRC State Inference* to allow us to understand the performance impact of RRC timers, and motivates *Network Energy Efficiency* examining if applications are in fact behaving well with respect to the interaction between RRC state timers and energy consumption.

3.5 Long-term Prefetching and Characterizing Carriers

Motivating *CellShift*, prior work [47, 22, 113] shows that having carriers offer discounts to users based on network load is beneficial and desirable for users, allowing them to use more data with less cost. TUBE [47] demonstrates that users are willing to time-shift traffic given the correct economic incentives, and that these incentives can significantly improve the distribution of delay-tolerant loads even if users increase data usage in response to lower prices. Mercado [22] presents a model of a system where some traffic can be designated as delay-tolerant and thus scheduled at a more desirable time by the carrier, and show that through "what-if" analysis that this approach can lead to improved network utilization. Through a user study, they find that software updates, large videos and cloud syncing are good candidates for time-shifting. Work by Sen et al [112] test out a small time-dependent pricing deployment, and find that this is likely a beneficial approach for users and carriers. Furthermore, they show that many less technically oriented users would even be willing to time-shift email or social networking, showing there is a great deal of diversity in how users use cellular networks. However, these papers focus on delay-tolerant loads in isolation for a small number of users, without examining large-scale cellular network load patterns, as

in Chapter VII.

There has also been work on operator support for such a prefetching system. A recent survey paper [113] shows that various forms of "smart pricing" have been implemented in real cellular networks and are becoming prevalent worldwide. While most popular for voice networks, especially in emerging markets, there has been recent interest in time-dependent pricing for mobile data in Europe as well. This work motivates the design of *Cellshift*.

Delay-tolerant apps are needed to build a system such as *CellShift*, and there has been a great deal of interest in designing these apps, especially to offload content to WiFi. Cedos [78] adapts existing apps to be delay-tolerant by generating a TCP-like API that handles disruptions and delays. They find that transparently offloading traffic can shift a significant amount of video and podcast traffic to WiFi and, through a user study, find that users would be willing to use such a system to delay requests by hours. A recent report from Cisco [23] predicts that the share of data used for video, audio and file sharing — data types Cedos identifies as being suited to delay-tolerant approaches — will grow dramatically over the next few years. Domain-specific approaches to making other apps more delay-tolerant has also been an area of recent interest. Cameo [69], is a comprehensive ad framework that improves mobile advertisement delivery in a variety of ways. Most relevant to prefetching, they find they can reduce the energy and data overhead of delivering advertisements by prefetching them by examining past user history to predict what ads will likely be shown to them in the future. O^2SM allows for the prefetching and offline viewing of social media content, dividing content up into streams and prefetching streams that are more likely to be viewed. They find they are able to significantly improve loading times with only a small increase in energy consumption. Furthermore, several major apps, including a podcast app [89], magazine-style app [40] and video app [134] already include delay-tolerant design approaches.

There has also been substantial prior work on shorter-term time-shifting. Informed Mobile Prefetching [51] decreases latency on mobile devices while prefetching while meeting

data and power budgets, through a system that provides cost-benefit analysis as to when to prefetch. Wiffler [13] focuses on offloading data from 3G to WiFi, delaying traffic by 30-60 seconds. It leverages a model of the evironment to predict 3G connectivity and is able to quickly switch back to 3G when needed, and is able to reduce the use of 3G by 30%. Work by Han el al [49] propose dealing with the increasing load on cellular networks by having devices collaborately share data when possible, falling back to normal cellular communication where needed. By carefully selecting sets of users to share data, they show that the majority of data can be offloaded in this way. Conversely, Procrastinator [100] argues that prefetching is frequently the wrong choice, given the cost of data on mobile devices, and provides a system to selectively prefetch data only when the user has specific data by automatically identifying what content is on the user's screen, resulting in data savings of up to 4x.. The prefetching we examine in this thesis is either on a much longer time scale, as in *Cellshift* and thus targets different types of content, or in the case of *App Traffic Prediction* proposes a new mechanism for short-term time-shifting.

More recently, CoAST [114] demonstrates that peak utilization can be reduced, by up to 50%, by scheduling traffic over time scales of seconds. This is based on the observation that there are many small load spikes on the order of seconds in the network, and that even content like streaming can be time-shifted on the scale of seconds. CellShift's approach is complementary: determining how best to distribute data within a one-minute time window does not preclude determining which one-minute time window to target. CellShift also has different challenges to address, such as forecasting and accounting for time-sensitive data, user movement, and the overhead of continuously coordinating traffic over long time scales. This sort of time-shifting could also be facilitated by predicting what content will be required in advance, such as in the cloudlet project.

There has also been theoretical work on how to optimally schedule content given some amount of future information. Work by Spencer et al [118] examines how the amount of information about the future during periods of high demand has a significant impact

on the queue length. Work by Xu et al [132] examines how to use a limited amount of future knowledge to bound the amount of delay possible when admissions can be rejected. Work on making use of future information to schedule optimally is complementary to our work, which focused on determining whether we could make use of imperfect network load predictions.

More specifically, there has been a series of papers by Tadrous et al on modeling the potential benefits of time-shifting data. First, in "Proactive resource allocation: harnessing the diversity and multicast gains" [68], they identify the problem of the disparity between peak and off-peak times, and build a model of the performance benefits of prefetching this predictable traffic. Like CellShift, they assume user load is highly predictable. They show in particular they can reduce the chance of requests being lost. They later elaborated on their model by examining the impact of imperfect predictability of demand and measured the cost of data delivery, showing that it is possible to use their system to reduce that cost [121]. Next, they extend their model to the multi-user case and show that prefetching can reduce the cost to the carrier and that this method continues to be beneficial as the number of users grows [120]. Finally, they elaborate on the statistical model by incorporating the network channel quality into the model [64]. They evaluate the impact of uncertain channel quality and determine an optimal strategy for scheduling data anyway. Overall, Tadrous et al have shown that from an information theoretic point of view, prefetching can provably provide significant performance benefits.

Unlike these more theoretical papers, we focus on dealing with limitations specific to long-term time-shifting on cellular networks, including determining if we can forecast network load, and dealing with the scheduling constraints on resource-constrained mobile devices, leaving determining an optimal scheduling method to future work.

Other work related to *Cellshift* looks at how cellular network performance varies on the carrier scale. Laner et al. [72] investigates cell loading patterns in a large European city and develop a model of network traffic and user behavior for future researchers to use. They

demonstrate that there are global network congestion trends, but that individual cell towers may differ substantially from this standard behavior. Xiong et al. [131] examine using collective trends in user movement patterns to predict future user locations. They are able to do so with some degree of accuracy 6 hours into the future, but with rapidly diminishing precision — it is not yet at the point where user location predictions are accurate enough for the scheduling done in the *Cellshift* project. Finally, Jin et al. [67] examine trends in data usage across users, finding that a small number of users and apps are responsible for a disproportionate amount of the content.. In particular, they show that heavy users tend to be clustered in a few cell towers. These carrier-scale variations in cellular network load motivate our work in the *Cellshift* project.

User location prediction is a challenging problem. Prior work has developed sophisticated statistical models for predicting the location of the next user given information, such as by using data from other users to help build a model when data for a specific user is insufficient [66]. In *Cellshift* however, we need to predict location on longer time scales and thus build a system that doesn't require such predictions.

3.6 Understanding New Application-Layer Protocols

In *Server Push* , I focus on Server Push in HTTP/2, but other work has looked at other aspects of next-generation protocols, and recent work has found the performance benefits of these protocols are mixed. Varvello et al [124] finds that the prevalence of HTTP/2 is small but rapidly growing, driven by a few key players, and that it offers some performance benefits in the wild, although that in general websites are not optimized for HTTP/2. "How Speedy is SPDY?" [127] performs a comprehensive evaluation of SPDY's performance under a variety of factors, finding that network quality and website design play a major role. They briefly look at Server Push, finding it to give performance improvements with high RTT. We examine Server Push specifically in more depth. The impact of SPDY on mobile devices specifically has also been examined: work by Erman et al [33] find that SPDY does

not consistently give performance benefits on cellular networks, suggesting it is worthwhile to examine the performance benefits of Server Push on mobile specifically. Work by Carlucci et al [19] examines the performance of QUIC, a new network protocol Google has proposed to replace SPDY, and finds mixed results. QUIC, being based on UDP, does mitigate the problem where SPDY's single TCP connection performs poorly under high loss rates. A study of SPDY performance by Elkhatib et al [30] examines the performance of real SPDY pages under a variety of network conditions and finds that SPDY's performance as a whole is impacted by network performance and web infrastructure. Recent work by Zarifis et al [136] builds and evaluates a model that can predict HTTP/2 performance from HTTP/1, and briefly examines the impact of Server Push, showing that it generally improves performance. Unlike this prior work, although some prior work has briefly touched on Server Push, we are the first to study Server Push specifically and in depth.

3.7 Cloudlets

Cloudlets are small computers near access points that are able to run computation or serve content for mobile devices with less latency than the cloud. In Chapter VIII, we use cloudlets as one motivating example for our work. Cloudlets have been extensively discussed in the past, including in a published lecture which summarizes much of the work in this area [38]. Work by Satyanarayanan et al [109] examine many of the challenges of making cloudlets work, in particular proposing a cloudlet architecture, exploring the VM requirements of cloudlets and the time to build a full VM. They find the overhead to be substantial, on the order of a minute. For our application having full isolated VMs per user that migrate their entire state are likely unnecessary, but we find migration even of the minimal amount of state to be a challenge. Work by Ha et al [46] focuses on the problem of creating VMs. They are able to create them in 10 sections for a specific application, by applying changes to a generic base VM. Work by Rajesh et al [12] opportunistically discovers servers and uses their resources with a focus on offloading computation. We

build on the concept of cloudlets in order to enable a system that automatically predicts user traffic for the purpose of prefetching it.

Using cloudlets to improve network performance has also been extensively explored. Infostations [59] proposes a new networking paradigm: high-speed, intermittent network connections are established with one user each to a nearby combined access point and server, called an infostation. The infostations can prefetch content locally and make intelligent decisions about fetching content and networking generally, leveraging information about current network conditions, user mobility, and the location of data. Work by Flinn et al [39] prefetches content to untrusted cloudlets, called surrogates, and uses encryption and the assistance of a trusted machine such as a home desktop to allow these untrusted devices to be used. A cloudlet-like approach has also been used in the real world [101], where internet connectivity is provided in a developing country by having users send an SMS message to a kiosk and then having the kiosk prefetch the data for when the user arrives. We build on this prior work on prefetching to cloudlets in the *App Traffic Prediction* project, by developing a system that could facilitate this prefetching.

CHAPTER IV

Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks

4.1 Introduction

Unlike traditional wired networks, or even WiFi, the high energy and resource cost of keeping a cellular network connection active along with the high overhead of establishing a connection capable of transmitting data has lead to a series of mechanisms of balancing these resource tradeoffs in response to data being sent. These *RRC states* (Radio Resource Control states) have different performance and energy consumption characteristics, and different latencies when transitioning to a high-power state. By using high-power RRC states only when necessary, and leveraging the temporal locality of network transmissions to avoid state promotion latencies, users can experience good network performance on resource-constrained mobile devices. Although the RRC states are largely defined by a set of specifications [4, 5], many aspects of the RRC state machine, such as timers for transitioning between states, are configured by the carrier.

The *complex network conditions and resource tradeoffs* we focus on in this chapter [103] are those related to the network states, known as *RRC (Radio Resource Control)* states, through which devices transition in order to balance power and performance. To better understand these states, this chapter introduces a technique to perform *ongoing, con-*

tinous measurements of network performance. As a result of our findings, we propose new ways in which *mobile systems can incorporate the results of these measurements* to address performance problems on these networks: specifically, that carriers should adopt a similar measurement framework to monitor their impact on user performance. In this way, this chapter supports the main thesis: because mobile devices experience uniquely dynamic and complex network conditions and resource tradeoffs, incorporating ongoing, continuous measurements of network performance, resource usage and user behavior into mobile systems is essential in addressing the pervasive performance problems in these systems.

Previous work [35, 63, 70, 65, 123] has focused on measuring RRC state configurations in the lab. It has been assumed that RRC state timers are static and RRC state transitions add a fairly constant and predictable amount of overhead. However, these static RRC timers do not provide the full picture of cellular network dynamics, and measurements performed in the lab on a limited number of devices. In this chapter, we focus on developing an in-depth understanding of the dynamics of RRC state transitions, which have a substantial impact on performance and have been under-explored. We first perform a global survey of the impact of RRC states and state transitions on performance in 28 countries on 3G and 4G LTE networks worldwide. In doing so, we determine that the impact of variable state transition latencies is substantial. We also discover a previously unknown cause of performance problems: *RRC state demotions*. When no data has been transmitted for *several seconds*, devices then enter a lower-power state. For many carriers, this is a long, complex process, and data sent during that time is often lost or delayed by as much as several seconds, in addition to the time to perform a subsequent state promotion. In previous work, these delays have been assumed to non-critical, but we discover for many carriers state demotion delays can often be the determining factor in overall packet latencies.

To understand and verify the existence of the performance problems discovered, we perform an in-depth, cross-layer examination of the root causes and application impact of state transitions. We start by examining the impact of layer 2 messages on RRC state tran-

sition latencies for several carriers. We discover major differences in the implementation of low-layer state changes and cell tower communication. In particular, overly complex state transition mechanisms, poorly-timed network connection configuration operations, and interfering control-plane operations cause substantial latencies during RRC state transitions for devices in use today. In particular, the use of the optional FACH state for 3G networks as a performance optimization in many cases actually leads to significant performance problems.

Additionally, we examine the impact of RRC states on higher-layer network protocols and Android applications. In our global deployment, we measure the impact of RRC states on HTTP requests and DNS lookups. We also develop an application for in-lab testing of Android applications in order to systematically measure the impact of RRC states on user-perceived performance in major applications. In doing so, we demonstrate that RRC transition latencies — including the previously unknown demotion latencies — can have a substantial impact on user-perceived performance.

Understanding the complex factors that cause RRC states to degrade application performance is valuable to many parties. Cellular network operators are interested in determining how devices on their networks perform and how both performance and signaling overhead can be improved. Major app developers have been interested in understanding how RRC state behavior can impact application performance [36, 74]. Finally, there has been interest recently by "power users" in understanding how issues such as RRC state implementations, as they differ among carriers, can affect performance [106, 44].

We summarize our contributions as follows:

- We provide an open-source RRC inference framework, requiring no special privileges, device-specific functionality, or network technology, for measuring the impact of RRC state transitions on performance (rather than just inferring the underlying timers.)
- We survey how RRC states impact performance in carriers worldwide and provide

an open data set of the results. Unlike previous work, we focus on measuring user-perceived latency rather than inferring configuration parameters. Through repeated measurements over time we can detect variations in latencies and transition delays. We also uncover variable timer configurations such as those due to Fast Dormancy, which will become more important as dynamic RRC timers become more common.

- We uncover previously unknown, severe latency problems that exist in many cellular network technologies and carriers around the world. These problems increase packet round-trip times by seconds (on top of normal transmission delays) and in LTE, can increase packet loss rates by at least an order of magnitude.
- We investigate the root causes of transition latency issues using cross-layer analysis. The most significant causes include complex state transitions in certain carriers, and non-RRC state control plane messages that coincide with state transitions, thus adding additional delays of hundreds of milliseconds, or even seconds, to already high-latency state transitions.
- We measure the impact of RRC states on application latencies as a whole, demonstrating that RRC states, especially transitions, have a significant impact on application-level latency as well as individual packets. In one case, we saw round-trip times greater than five seconds during a state demotion! To do so, we developed a controller application for systematically measuring the impact of RRC state transitions on requests initiated by user input.

Overall, we find that the impact of the RRC state machine on user-perceived performance is more complex than what has been described in previous work. In addition to having implications on designing network state aware apps for better performance and power consumption, our findings have implications on carrier network configurations as well. We propose that carriers should use a measurement system such as ours to monitor how their RRC states impact client performance.

Table 4.1: Summary of results in figures and tables.

Section	Name	Key finding
§4.2 (Methodology)	Table 4.2, Fig. 4.2	Validation of inferred RRC timers
§4.3 (Results)	Fig 4.4, 4.5, 4.6, 4.7	State transition delays for all network types can be substantial.
§4.4 (Root causes)	4.8	Causes of LTE transition delays
§4.4 (Root causes)	Fig 4.9	Causes of 3G transition delays
§4.5 (App impact)	Fig 4.10, 4.12, 4.11	RRC states affect a variety of application and transport protocols.

We start by describing our measurement methodology (§4.2), including our inference approach for the global deployment and the approach used for cross-layer local experiments. We then discuss our global results (§4.3) followed by an in-depth examination and confirmation of results from several carriers (§4.4). Finally, we examine the impact on application performance (§4.5) and discuss the implications of our finding and future work (§4.6). We summarize our findings in Table 4.1.

4.2 Measurement methodology

To understand RRC state performance, particularly the impact of RRC state transitions, we use several tools to develop a cross-layer understanding of RRC performance problems, their root causes, and their impacts on application performance.

It is known that state *promotions*—moving from a lower-power state to a higher-power state—involve additional latencies. We refer to these additional latencies as *promotion latencies*. One contribution of this chapter is a cross-layer, experimental examination of variations in these latencies across carriers, and how implementation differences among carriers lead to these variations. We also discover that the impact of *demotions* on latency can also be quite substantial, *i.e.,* moving from a high-power to a low-power state in some cases involves up to several seconds of network reconfiguration and measurement, In addi-

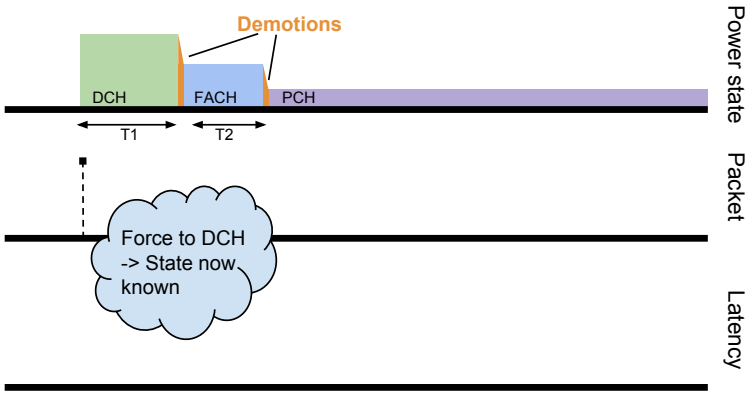
tion to adding delays of several seconds, for LTE it significantly increases the packet loss rate. We call the resulting delay the *demotion latency*. Previous work treats promotion latencies as constant and disregards demotion latencies, focusing on demotion timers. We demonstrate that variable promotion and demotion delays, differing by carrier and varying over time, have a substantial impact on performance.

To globally survey the impact of RRC state transitions on performance, we collected data from a wide range of carriers using an open-source cellular network testing tool for Android (§4.2.1). This tool adapted standard RRC inference techniques [35] to run automatically on end-user devices, and was designed to require no special privileges or device-specific functionality, allowing it to run on arbitrary user devices. It also accounts for interfering data which might otherwise result in incorrect measurements. We also measure the impact of RRC states and state transitions on HTTP requests and DNS lookups.

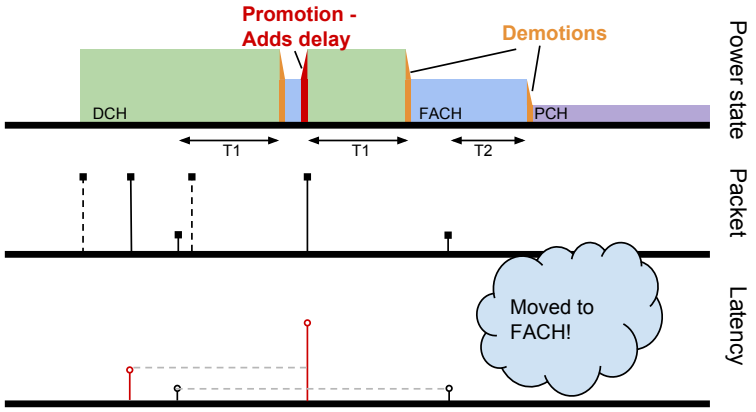
We also used local, controlled experiments to understand the performance impact of RRC states, especially RRC state transmissions, using a cross-layer approach. Starting at the RLC (Radio Link Control) layer, we examine control and data messages directly using a tool called QxDM (Qualcomm eXtensible diagnostic monitor) [99], in order to understand the root causes of the observed transition delays (§4.2.2). We built an application controller tool in order to test the impact of RRC states on user-perceived performance at the application layer.

4.2.1 Automated RRC Performance Measurement

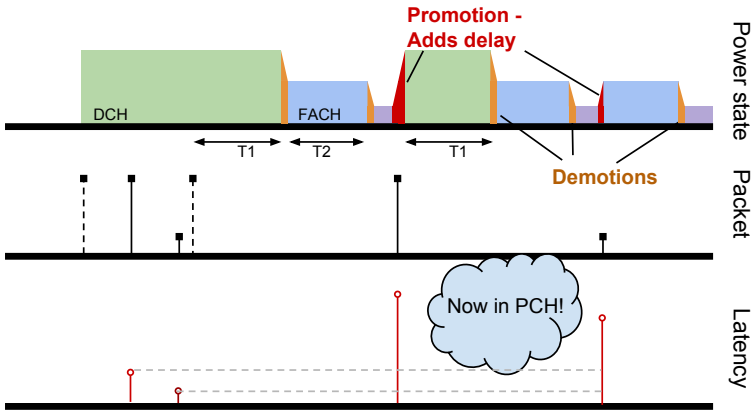
Inferring RRC state timers by observing how packet latencies change as the time between packets increases has been examined in previous work [35, 63]. The standard technique used is as follows (summarized in Figure 4.1): first, a UDP packet is sent to ensure the device is in a high power state. Next, the device is left idle for a period of time before another UDP packet is sent and echoed back by the target server. The latency of the second packet can then be compared to the latency of the first packet; if there is a substantial



(a) Initial packets sent.



(b) With a longer interval between the packets sent, we enter FACH.



(c) Finally, with an even longer inter-packet interval, we enter DCH.

Figure 4.1: Impact of sending packets with varying interpacket intervals and how one can use those to infer RRC states.

increase, it implies that a state promotion has occurred between them, adding latency. By examining a range of inter-packet intervals, the time at which a state transition occurs after a packet is sent can be determined. To distinguish between FACH, where a transition only occurs for sufficiently large packets, and PCH, we perform this test with empty packets and 1 KB packets.

Although this technique has been applied in the past in a controlled setting to measure RRC state timers as set by carriers, we focus on measuring how RRC states and state transitions impact *user-perceived performance*, by providing an implementation suitable for end-user devices. This allows us to deploy our system broadly, to cover global carrier-dependent effects, as well as gather data over a long time period, to capture probabilistically occurring problems (whose presence we then confirm through controlled experiments). Unlike methods of measuring RRC state transitions which focus on RRC state configuration, we are able to capture the effect of RRC states as they impact performance in practice, in particular performance issues caused by RRC state transitions.

To measure RRC states on a large scale, we implemented this method in an open-source network measurement tool for Android devices, and released it to the public. We are able to survey a large number of carriers to gather a representative picture of how RRC states on carriers around the world affect performance, and discovered performance problems not found in previous work.

A challenge of a public deployment is that control over all traffic on the device is needed. To address this problem, we use a Linux utility (*/proc/net/dev*) to monitor background traffic that might interfere with the current measurement, and discarded and rescheduled tests when this interference occurs. This RRC test runs automatically in the background to allow ongoing monitoring with no user involvement, and sends results back to a server, along with information such as the carrier and signal strength at the time the measurements were taken. User identifiable data is anonymized.

We also added tests to observe the impact of varying packet sizes on RRC state transi-

tions, by adjusting the size of the second packet sent. Finally, to observe the impact of RRC states on HTTP and DNS lookups, we replaced the second UDP packet with the request in question. Furthermore, as we measure RRC states repeatedly over time, we are able to observe dynamically configured timer values varying from test to test, such as those resulting from fast dormancy.

We vary the inter-packet timings at the granularity of half-seconds, which is not sufficient to identify timers for switching between various DRX timers in LTE's CONNECTED state. We determined these timers can be measured by performing a set of tests locally with several carriers to measure performance changes when inter-packet intervals are varied by 50 ms. For these timers, there were no unexpected transition delays. As measuring timers at such a fine granularity requires an order of magnitude more packets to be sent, thus consuming a lot of cellular data, and as we found no surprising results in our local experiments, we did not deploy this test globally.

4.2.2 Root Cause Analysis with QxDM

QxDM [99] is a debugging tool that can view all network data and signaling messages in the form of a pcap-like trace. Using this tool, we can map IP packets to RLC (Radio Link Control) PDUs (Packet Data Units), which are layer 2 data-plane messages. We are also able to view the control-plane messages associated with RRC state changes. We can then determine how RLC PDU delays and control-plane messages affect latency to understand the impact of RRC state changes on user-visible performance. We combine pcap traces with QxDM logs to determine what RLC events surround IP packet transmissions, using timestamps to match events at both layers. For 3G, we are able to map individual PDUs to IP packets as the contents of PDUs are logged.

To perform this analysis, we use a simplified RRC state testing application which repeatedly cycles through inter-packet intervals in order to induce RRC state transitions. By analyzing the resulting trace, we can determine which control messages related to each

	Demotion type	App.	QxDM
3G	C1 DCH⇒FACH	3 ± 0.5 s	3.1 ± 0.1 s
3G	C1 FACH⇒PCH	6.5 ± 0.5 s	6.2 ± 0.8 s [1]
3G	C2 DCH⇒Disconn.	10 ± 0.5 s	10.3 ± 0.1 s
	— fast dormancy	3 ± 0.5 s	3.2 ± 0.1 s
LTE	C1 Conn.⇒Idle	10 ± 0.5 s	10.5 ± 0.1 s
LTE	C2 Conn.⇒Idle	10 ± 0.5 s	10.2 ± 0.1 s

Table 4.2: Comparison of ground truth demotion timers from QxDM with values measured through the application.

RRC state and RRC state transition result in substantial delays. We also determine if any non-RRC state transition related processes are interrupting state transitions. We analyzed traces from three different carriers with different RRC state implementations and different transition delay behavior. In §4.4, we break down the causes of various RRC state delays, and compare carriers with differing delay behavior in order to both validate the presence of the observed differences and previously unknown delays, as well as understand their causes.

A limitation of this approach, unlike the app-based approach, is that it cannot be performed on actively-used devices in the wild, as proprietary software and some external equipment is needed, as well as specially configured devices. It is complementary to the app-based approach, which allows for a broad survey of RRC state performance to be performed, covering many carriers, locations, and device types. Conversely, this approach is more suitable for in-depth examination of specific performance problems. For this reason, this approach we use to understand RRC state behavior is likely of most use to carriers who, having detected a performance problem, are interested in understanding how best to address it.

Finally, in order to validate the application-based RRC state measurement methodology, we use QxDM to determine a ground truth for RRC state timers (*i.e.,* the time after a packet is sent where a state demotion occurs). After determining timers from two carriers for RRC states, we then verify the values by comparing the inferred RRC timers with the ground

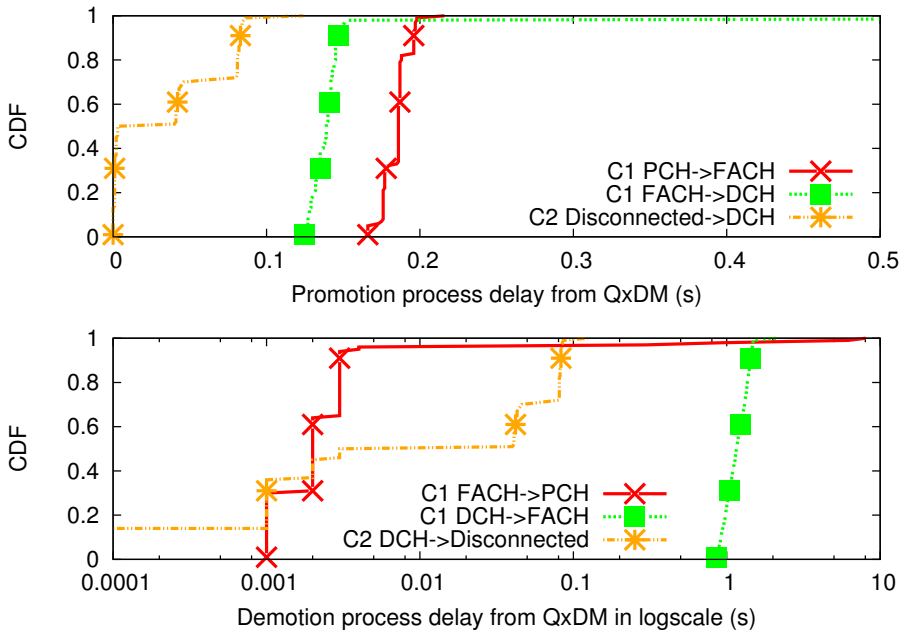


Figure 4.2: Measurement of demotion and promotion times for two carriers in QxDM.

truth values from QxDM, shown in Table 4.2. As we infer the RRC timers set by the carriers from changes in the measured performance, elevated and variable latencies during long RRC demotions mean that sometimes these values can only be inferred to within about a second. This limitation applies only to inferring the *demotion timers*, not to be confused with the *demotion or promotion latency*, which we measure at the millisecond granularity. Changes in the promotion latency are used to infer the demotion timers.

We also confirm the presence of the long delays during state transitions observed in our application tests. We examine three major carriers with over a hundred million subscribers each, which we refer to as C1, C2 and C3 throughout the chapter. The carrier names are anonymized to avoid appearing to endorse specific carriers, as we have uncovered performance and configuration issues in these carriers that they may want to address. In Figure 4.2, C1 has a substantially longer RRC demotion process delay than C2, which prevents packets from being sent during that time and results in significantly longer transmission delays at the application layer. We evaluate this in §4.5.

To systematically evaluate the impact of RRC state transition delays on user-perceived application performance in §4.5 through a cross-layer analysis framework, we develop an application controller which simulates normal user behavior over real, major Android applications such as Facebook. Built upon the Android Test Case framework [1], this controller programmatically triggers Android UI events such as clicking buttons and entering text, and enables performing common application UI operations such as "pull-to-update" on the Facebook news feed list. To measure the user-perceived UI latency, the controller also logs UI events, such as the start and end time of the news feed loading. As measured by Android DDMS [2], an application performance profiling tool, our controller incurs a computational overhead of less than 2% and thus has minimal impact on user-perceived latency measurement.

4.3 Global performance measurements

Our approach to measuring RRC states allows any Android device on any cellular network to measure the impact of RRC state and state transitions on user-perceived performance. We deploy our measurement tool as part of a mobile network testing suite to survey RRC performance around the world, to ensure results are representative of global user experience. This tool measures network performance automatically and in the background on Android devices, allowing users to effortlessly monitor performance trends. The amount of data consumed is configurable by users, and all data sent to the server is anonymized to protect the user's privacy.

Due to lost packets, interrupted measurements and unrelated network delays, a single set of tests was usually insufficient. For our final results, we consider carriers with more than 5 complete tests only, so that transient network delays, unrelated to RRC, will not affect our results. We also excluded six carriers where network noise was so high that all RRC states were indistinguishable.

In Figure 4.3, we give an example of measurements from one device type and car-

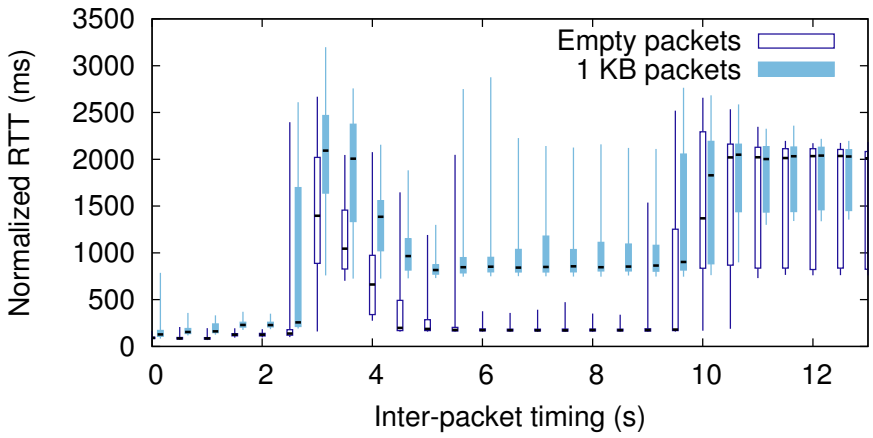


Figure 4.3: Observed round-trip times while transitioning through RRC states. Median, quartile and 5%/95% values shown.

rier (sizes do not include headers). The round-trip time for large packets is higher for inter-packet frequencies between 4–9 seconds, and higher for both packets from around 10 seconds onwards. In FACH, we expect round-trip times for large packets to be substantially higher than in DCH by design, and round-trip times for small packets to be similar to DCH. In PCH, while all packet sizes experience high round-trip times, we found that large packets still have a larger round-trip time than small packets due to network delays.

Unlike RRC measurement tests in previous work [35, 63, 70, 65, 123], we focus on measuring the impact of RRC states on user performance rather than just measuring RRC state configuration parameters. In particular, we were able to observe behavior inconsistent with the ideal model of RRC state transitions. In Figure 4.3, there is a period of several seconds after a packet is sent, from about 2.5 to 4 seconds, when the next packet experiences unexpectedly high round-trip times. We refer to this delay as the *promotion delay*, and the period of time where it occurs as the *promotion period*. Where there are no promotion delays, we instead identify the interpacket interval at which the promotion occurs to be the promotion period. For example, in Figure 2, the demotion can be seen between 9.5 and 10 seconds.

Carrier and device characteristics: After filtering out carriers with insufficient data

for our analysis, we analyze 44 carriers in 28 countries covering every continent. Data on 69 distinct device model types and seven distinct network types was collected, including 2G, 3G and 4G technologies. In this chapter we focus on 3G and 4G, which have been adopted by most carriers. 7 carriers use LTE, 23 use HSPA+, 16 use HSPA, 25 use HSPDA, and 6 use EVDO_A, with many carriers supporting more than one technology. In particular, most carriers with LTE also provide 3G.

Almost all carriers with LTE have a demotion timer to CONNECTED of 10 seconds, but with 3G technologies, the timers vary greatly, from 2 to 10 seconds, although the total time to enter PCH is generally less than 10 seconds. Carriers providing multiple 3G technologies generally use the same timers for each. About 2/3 of carriers with HSPA, HSPDA or HSPA+ have no FACH state, or at least no FACH state with measurable performance impact. We only saw definitive evidence of fast dormancy — a demotion timer varying substantially from test to test — with one carrier. Two more carriers exhibited variations of about a second. As fast dormancy and other dynamic RRC state timer approaches become more prevalent, the ability to measure these variations will become increasingly valuable.

For 3G, we also examined the impact of packet sizes on RRC state transitions, by varying the packet payload size from 0 to 1000 bytes by increments of 200 bytes. Dramatic increases in latency, indicating the size threshold for a promotion from FACH, all occurred between 0 and 200 bytes. Given the small threshold for a promotion from FACH and the high associated demotion overheads, FACH may not provide much benefit. We also observed that RTTs increase steadily with packet size by as much as a few hundred milliseconds in all states.

Transmission delays: Ideally, the overhead of acquiring radio resources to use the radio channel should be fairly constant, independent of the idle time of the device. We observed that when network transmissions are sent when the demotion timer expires and the device undergoes a state demotion, there is an unexpected and undesirable increase in the delay to promote back to a high power state. This problem occurs for a large number

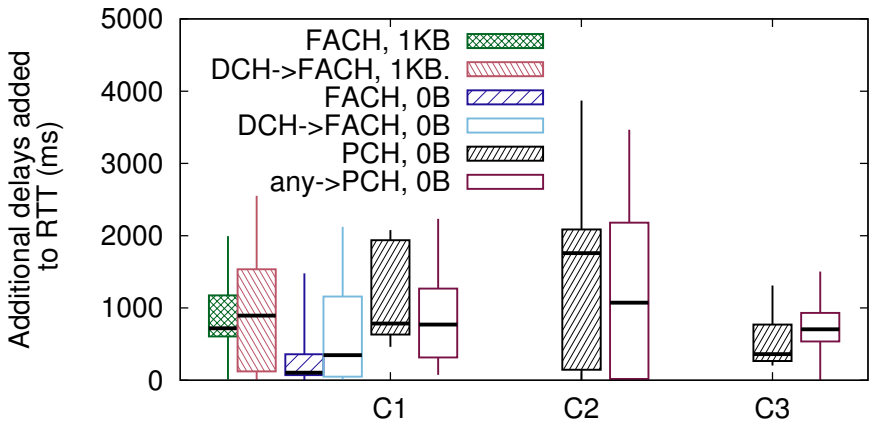


Figure 4.4: Variations in delays due to 3G states and state transitions, normalized against the DCH RTT. Median, quartile and 5th/95th% values shown.

of carriers.

We start by describing detailed results from three major carriers, illustrative of three different observed behavior patterns, and then summarize results globally. Values are normalized by the average latency in the absence of any RRC state change. We show values during state transitions separately, as described above.

In Figure 4.4 we show results from all 3G technologies for each carrier. C1 has implemented FACH, which has a substantial difference in latency depending on packet size. During demotions to FACH, latency becomes higher, especially for smaller packets, as expected. Round-trip times are substantially more variable during this time period, leading to poor tail performance. The demotion to DCH does not lead to substantial latencies. C3 is a CDMA network and thus does not implement FACH, but still experiences demotion delays. C2 does not implement FACH and does not experience observable demotion delays globally, although we find in some areas (including in local experiments), where performance is poor, demotion delays for this carrier appear. Performance sending data from low-power states is substantially worse for this carrier, although network performance for this carrier was generally poor.

In Figure 4.5, we show LTE performance for three carriers, comparing CONNECTED

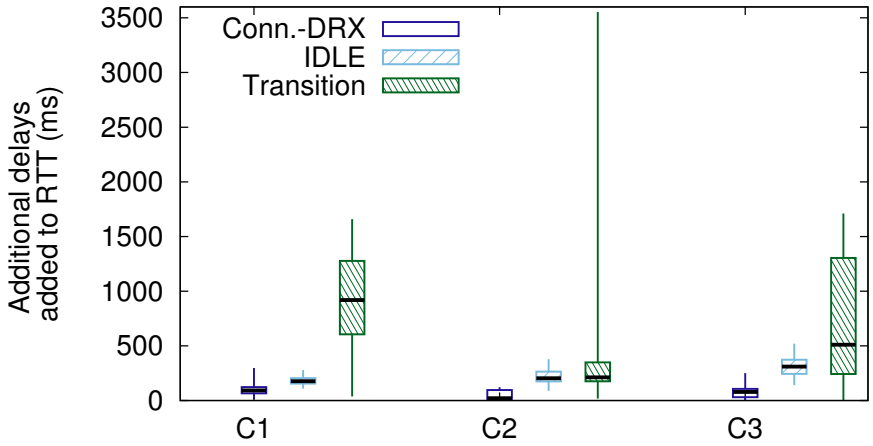


Figure 4.5: Delays due to LTE states and state demotions, normalized against the RTT of an empty packet sent in CONNECTED with no DRX.

against IDLE and the demotion between the two. LTE is supposed to perform better than 3G, but this is not necessarily true during demotions. For all three carriers the tail latency is substantially higher during state transitions, lasting potentially up to several seconds. For C1 and C3, the median values are substantially higher as well. In §4.4, we discover this difference is due to differences in how state transitions in these carriers are affected by control-plane activity.

For LTE only, we also found packet loss during state transitions are higher than average, by up to an order of magnitude. To measure loss rates, we sent out ten empty packets simultaneously and counted how many were echoed back. C1, C2 and C3 experienced packet loss of 26%, 63% and 68% respectively, with normal loss of 1–3% depending on the network state (aside from C3 which experienced loss of up to 30% of packets).

To examine trends across all carriers in RRC state impact on performance, we start by examining the impact of state *promotions* in Figure 4.6, which have been examined in previous work only for a small number of carriers. Promotions from FACH generally take several seconds, and even for empty packets, state promotions are sometimes triggered, contrary to expectations. Promotions from DCH are also long, and the performance impact

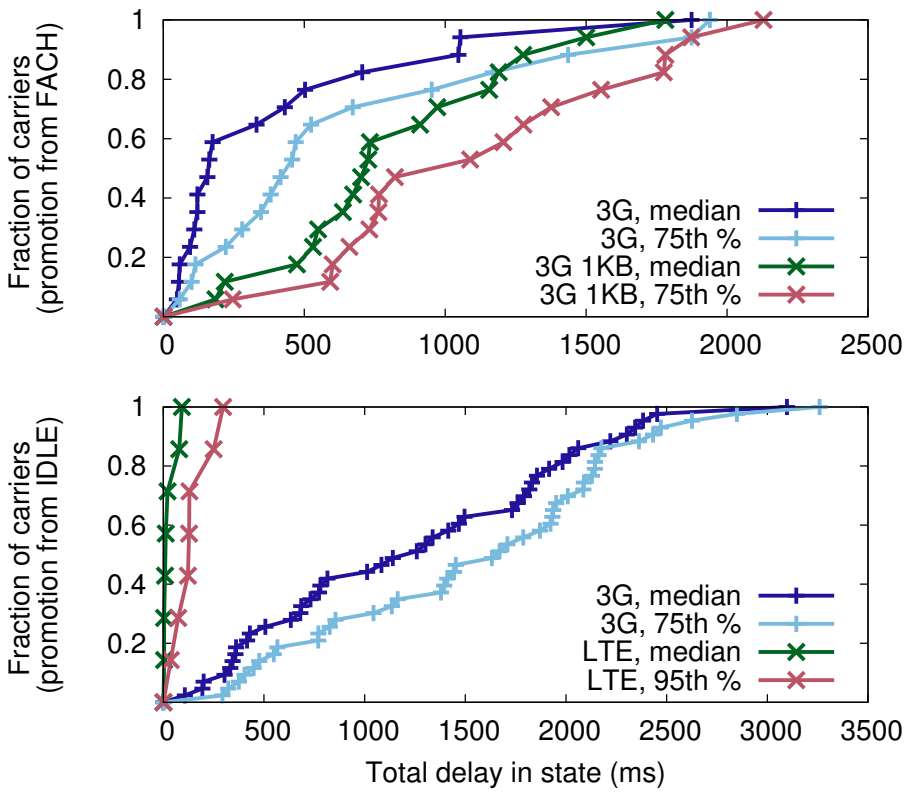


Figure 4.6: CDFs of distribution of latencies during state promotions over all carriers: in the top graph, going from FACH and in the bottom from IDLE.

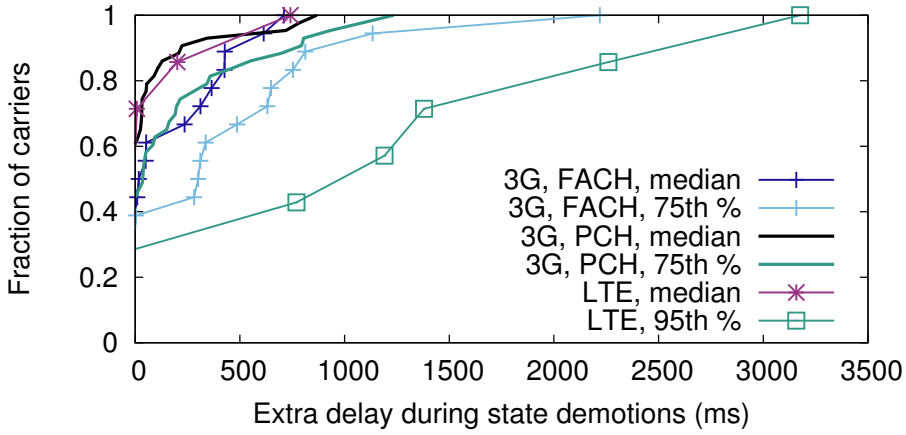


Figure 4.7: CDF over all carriers of additional latencies caused by transmissions during state demotions (minus promotion transition times in the new RRC state).

varies dramatically from carrier to carrier. LTE state promotion delays are generally no more than a few hundred milliseconds.

In Figure 4.7, we show the *additional* latencies added by attempting to send data near a state demotion, on top of the state promotion delays. We compare median values during state demotions with median values for state promotions only, and likewise for 75th and 95th percentile values. Ideally, no additional latency should be incurred, if the demotion is aborted, allowing the device to simply remain in the high-power state. This is not the case, especially for demotions to FACH. Eliminating FACH (as many carriers have) would likely reduce, though not completely eliminate, the prevalence of this demotion delay problem.

For LTE, median latencies are generally not affected by state promotions. It seems our local carriers explored above have somewhat non-typical behavior, underscoring the need for a broad survey of network performance. However, tail latencies, as we found earlier, are frequently affected. Note that we show 95th percentile latencies and not 75th percentile latencies. As we saw earlier, these tail latencies are substantially higher during demotions than any other time. Given the low network delays in LTE generally, these delays can have a major impact on user-perceived performance. Given that major web services go to great lengths to reduce tail latencies for 0.01% of users due to the potential revenue impacts [27], these latencies can be quite significant.

Summary: State demotion delays are common worldwide, though not experienced by every carrier, and occur in both 3G and LTE. Where they occur, they can have a critical effect on performance, in some cases causing delays of several seconds. State demotion delays (and to a lesser extent, state promotion delays) also vary greatly between requests, adding additional latencies of up to several seconds on top of the normal state promotion latency. Additionally, in LTE, state demotions are associated with high packet loss rates. More generally, we have shown that running RRC state performance tests on user devices is an effective way of monitoring global RRC state performance trends.

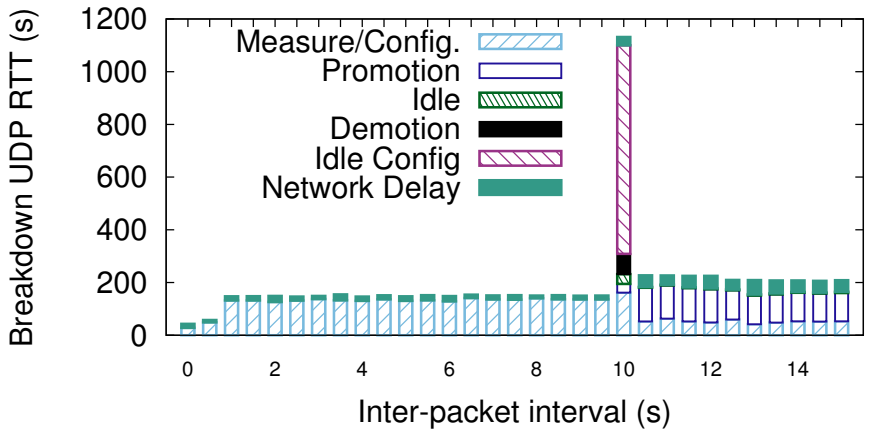


Figure 4.8: Median values of sources of transmission delays for C1 , labeled based on the messages observed in QxDM. C2 is similar but lacks "Idle Config.", which seems to have to do with network configuration.

4.4 Root Cause Analysis

Through controlled, in-lab testing of RRC latencies and measurements with QxDM, which provides detailed visibility of RRC state transition control and data messages, we examine the events that contribute to RRC state delays. Although delays during state transitions occur in nearly all cellular network technologies, the causes of these delays (as well as their magnitudes) differ as shown in §4.3.

Causes of promotion delays in LTE: First, we examine state promotions that do not occur in the vicinity of state demotions. Although previous work has identified that state promotions cause delays, the root causes of and variations in these delays have yet to be examined [35, 63]. We summarize the median delays for different inter-packet intervals in Figure 4.8. The promotion delay, unsurprisingly, adds a highly varying delay to the overall latency. This delay includes the effects of Discontinuous Reception (DRX), where devices must wait a few hundred milliseconds before sending data. This promotion process starts with a request to switch to a higher-bandwidth, more reliable channel, on an unreliable network channel. Delays during this process due to poor network conditions sometimes add substantial additional latencies. This contributes significantly to the high variation in

worst-case or tail network latency seen in Figure 4.5. A detailed description of some of the messages involved in state promotions (though not demotions) for LTE and 3G can be found in a white paper by Mohan et al. [77].

Causes of demotion delays in LTE: In Figure 4.8, it can be seen that latencies are often significantly worse during state demotions. These involve more measurement and configuration messages, although the demotion process itself is quite short. The promotion delay in this period is also very short, since an immediate promotion means that the DRX delay will be minimal.

We have isolated one set of message delays in particular that can add several seconds of delays, and labeled them "Idle config." These messages appear to be related to transmission synchronizations with the base station, although they are not well-documented. If an IP packet is sent before this message appears, then the entire process completes before the state transition process begins. However, if an IP packet is sent after it appears, then this process is aborted and a state transition begins right away, so this delay only appears for a narrow window of inter-packet timings. This illustrates the dependencies of control messages on the data packet timing. These messages do not appear for C2, which explains the lower median latencies during demotions seen in Figure 4.5. Additionally, for all carriers the device occasionally momentarily disconnects from the network before selecting a new cell tower, causing long delays. This appears to be responsible for the long 95th percentile latencies seen in Figure 4.5. This is likely unavoidable as user movement or poor network performance may necessitate this switch.

Causes of promotion delays in 3G: We summarize the breakdown of latency causes in Figure 4.9 as they vary by inter-packet interval. For 3G, promotion times are often longer — roughly 1200 ms on average where they occur. After a state promotion, additional control plane messages are sent, such as messages to measure channel conditions. These messages take up significantly more time than the state promotions themselves, and the messages seen can vary. One series of system information messages adds additional delays

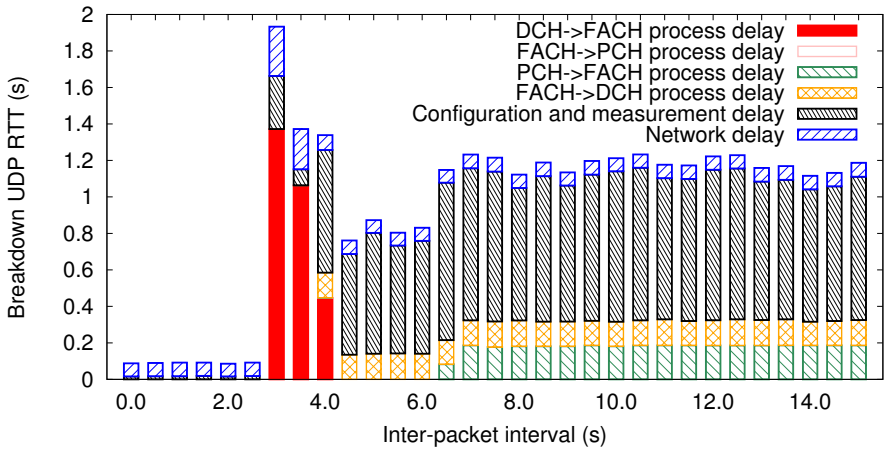


Figure 4.9: Breakdown of RTTs for varying inter-packet intervals, including a demotion to FACH at 3s and a demotion to PCH at 7s. The impact of the FACH⇒ PCH demotion is essentially nonexistent in this case.

of hundreds of milliseconds where it occurs, leading to high latency variations. These messages occur periodically, every few hundred milliseconds, not just during state transitions. Overall, state promotions are more complex and involve more messages being exchanged for 3G than for LTE. State promotions are already known to be slower in 3G for this reason [77].

Causes of demotion delays in 3G: In Figure 4.9, it can be seen that state demotions have a substantial impact when the inter-packet interval is between 3 and 4 seconds. Unlike with LTE, it is simply the demotion process itself which is slow, rather than other control plane messages which cause unexpected delays. This makes promotion latencies more common as well as affect a larger range of inter-packet intervals. As we showed in §4.3, for 3G, promotion delays affect more requests. It is also interesting to note that when a state demotion is interrupted in this manner, there is often no subsequent promotion delay.

Interestingly, several carriers appear to lack these demotion delays altogether. We were able to examine one such carrier in depth using QxDM. Our first observation was that this carrier omits the FACH state. However, we found in our global study of 36 carriers that not all carriers which omit FACH lack significant transition delays. The main difference

is that this carrier's demotion process is substantially simpler, consisting of sending one message to the base station followed by a small amount of additional delay due to device configuration operations. As this adds a median time delay of 175 ms, it did not have a statistically significant effect on the user-experienced latency. It is also interesting to note that this carrier was also one of the few carriers in our study to have implemented Fast Dormancy, a performance optimization which appears to have yet to be widely deployed. It also has fewer LTE demotion delays.

Summary: We have determined that, for both LTE and 3G, carrier-specific, RRC state related messaging and configuration delays can interact poorly with certain network state patterns. At least one carrier has been able to reduce these delays greatly. In general, however, LTE's state transition procedure ensures much better average performance than 3G's, largely due to a lower amount of control-plane signaling needed in order to transmit data or change RRC states. Delays in LTE are primarily due to poor interactions between certain control-plane messages and the state demotion process, affecting only a subset of requests (although it can add delays of several seconds). Delays in 3G, however, are generally due to issues with state demotion implementations. Additionally, while it was already known that state promotions can cause network delays, we experimentally quantify which components of the state promotion process lead to promotion delays. The overall observation is RRC state transitions contribute significantly to tail latencies on mobile devices.

4.5 Application impact

We next explore how upper-layer protocols are affected by RRC state transitions, using both our globally deployed RRC state measurement tool and in-lab controlled experiments. We find that HTTP connections, DNS lookups, and mobile applications are all impacted by poor RRC state transition performance.

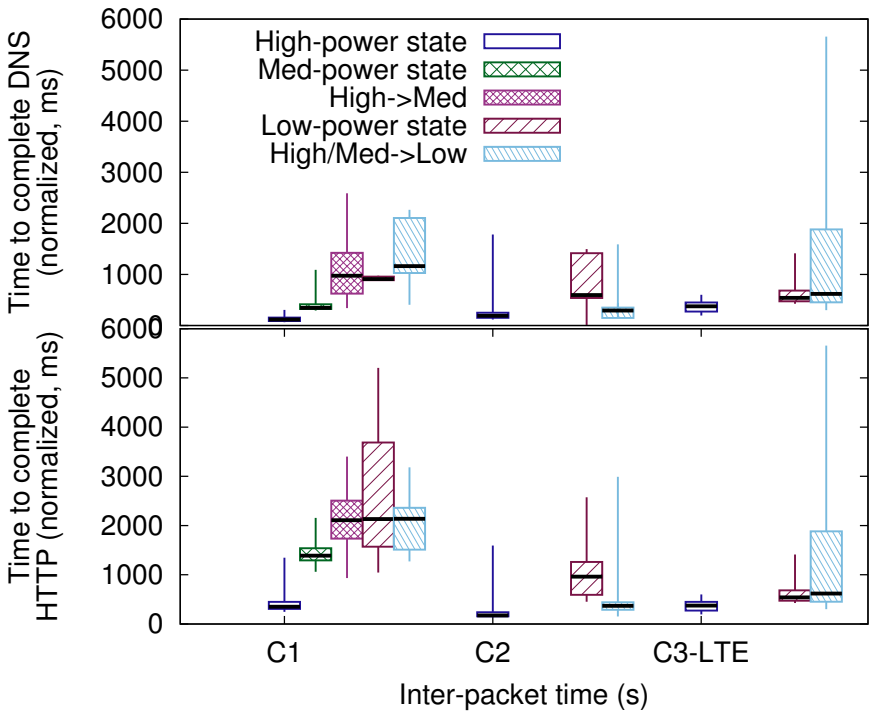


Figure 4.10: Performance of different carriers with different inter-packet timings, for DNS lookups and HTTP connections to a small website.

4.5.1 HTTP and DNS Results from Global Deployment

In our public deployment, we measured the impact of RRC states on DNS and HTTP requests, which are more representative of real network traffic than individual UDP packets. Testing with UDP packets allows us to understand the impact of RRC states without being affected by network protocol features, but UDP is not representative of most network traffic. We show the impact of RRC states on a DNS lookup and on the loading of a small web page in Figure 4.10. We show results from the three carriers discussed in-depth before: C1 in 3G, which experiences substantial demotion delays in our UDP-based testing; C2 in 3G, which did not experience these delays, and C3 in LTE.

For individual packets in 3G, C1 was found to experience promotion delays, whereas in 3G C2 did not. This trend also impacts the performance for DNS lookups and HTTP requests. For C1, FACH performs worse than DCH, and data sent during the promotion to FACH performs even worse, comparable to the performance in PCH. Unfortunately, for

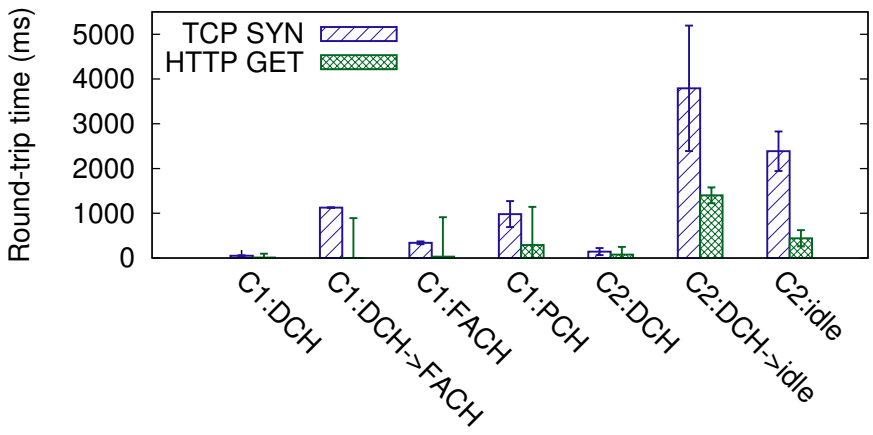


Figure 4.11: Effect of RRC states on TCP SYN RTTs and HTTP GET latencies.

this carrier we have limited data in PCH, but as we found before, demotions to PCH for this carrier do not add substantial extra delays. Like with our UDP tests, we also found that for C2 there were no state demotion delays. For LTE, while the median performance during state demotions is not substantially worse than in IDLE, the tail results are substantially worse, in one case lasting more than *five seconds* for a DNS lookup! Overall, we can see that these RRC state performance problems have a real impact on users.

4.5.2 Controlled Web Browsing Experiments

To verify our findings, we also examined RRC state delays in different circumstances in controlled, in-lab experiments. We evaluated the page loading time in a browser for 10 major websites, including search, social networking e-commerce, news, sports and finance websites. We varied the inter-request time from 1s to 11s, with a granularity of 0.1s. In total, we generated 3000 HTTP requests for both C1 and C2 over an entire day. In Figure 4.11, we compare the TCP SYN RTTs and the HTTP GET request RTTs from TCP flows. In addition to the expected promotion delays, the demotion delay increases the SYN RTT substantially. As the HTTP GET request starts with a SYN request, it suffers from the same transition delays.

We also evaluated the user experienced latency starting from the SYN packet until

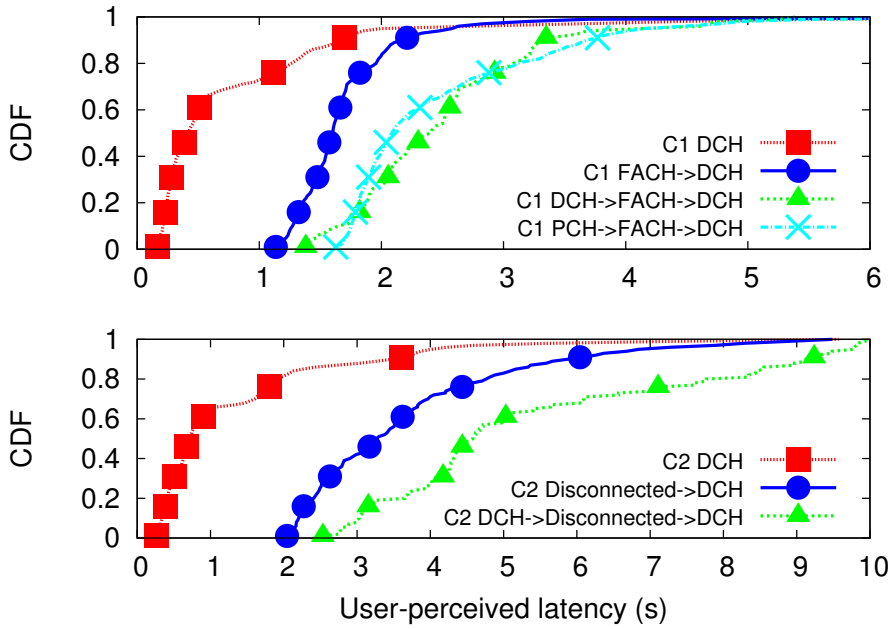


Figure 4.12: Effect of RRC states and transitions on user-perceived latency in web browsing experiments.

the last HTTP-related packet is received, focusing on network latency (i.e., disregarding Android system latencies). In Figure 4.12, we show the distribution of user-experienced latency when browsing in various RRC states as well as during state transitions. Starting in a low-power state has a substantial performance impact, adding 0.5–3s of user-experienced latency. C2's throughput happens to be significantly worse than C1's at our location, causing the atypical performance differences between the two.

Since C2 lacks an intermediate FACH state, unlike C1, there are two demotion types shown for C2. As a result, there is a higher chance that users of C1 transmit data during a state demotion. In our controlled experiments, we found that for C2, 2.4% of HTTP GET requests experienced demotion delays, and for C1, 4.25% of requests were affected.

4.5.3 Case Study: Facebook Application

In order to demonstrate the impact of RRC state transitions on a major app other than a web browser, we examine a common operation in the Facebook application. Facebook is

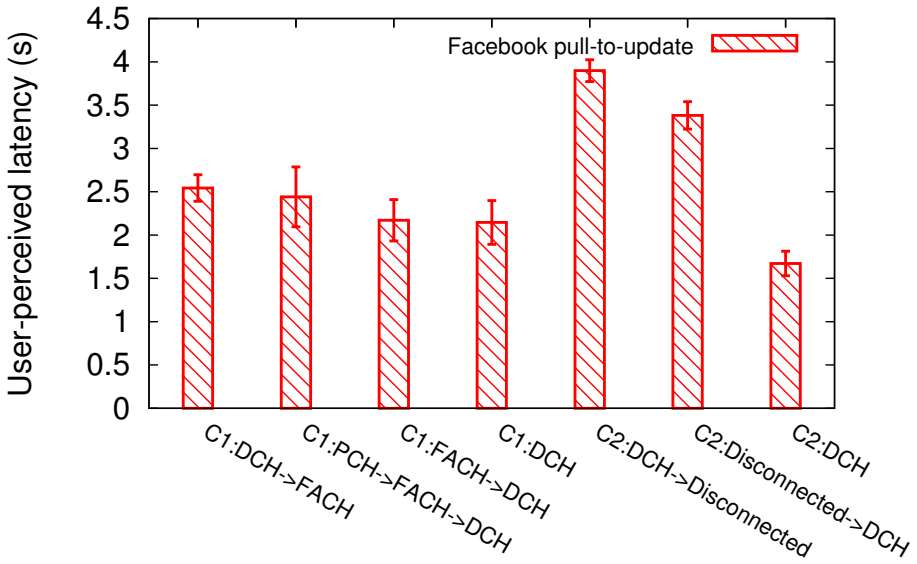


Figure 4.13: Impact of additional RRC state transition delays on Facebook's pull-to-update action.

one of the most popular social networking services, with 945 million active mobile users during the month of December 2013 [96]. Limiting network latency is critical to ensuring good user experience. One of the most important features of Facebook is its news feed [92]. We examine the time to fetch new news feed content over the network by pulling down on the list (the "pull-to-update" action). To systematically and repeatedly measure the latency associated with app operations, we created a controller application that repeatedly initiates the pull-to-update action and logs the timestamp of this action and the resulting news feed load completion event.

We performed the experiment on two Android 4.2.2 Samsung Galaxy S3 devices. We created two Facebook accounts, A and B, which are friends with one another. One device with account A repeatedly uploads two photos to generate news feed data. The other account performs a pull-to-update operation, varying the time intervals between each action. As shown in Figure 4.13, the DCH⇒FACH demotion process increases the user perceived latency by 398 ms for C1, and the DCH⇒Disconnected process introduces an additional 2.225s delay for C2. The results are consistent with the web browsing experiments. As

with those experiments, the RRC state transition delay is worse for C2 due to exceptionally poor network performance where we performed the experiment.

Summary: We show that the problems we observed are not just limited to affecting individual packet latencies. RRC state transitions, especially RRC state demotions, can greatly impact network and application layer protocols, as well as the performance of web browsing and network applications directly, adding delays of up to five seconds in the worst case. As we saw with the impact of RRC state transitions on latency for individual packets, the degree to which transitions impact performance varies significantly by carrier. Most importantly, for all carriers, in the worst case, delays of over a second can be added.

4.6 Discussion

We have shown that only determining RRC state timers set by the carrier is insufficient for understanding the impact of RRC state on performance. The model of how RRC state transitions impact user-perceived latency to date is incomplete, as variable state transmissions, in particular state demotions, have a critical impact on performance. Being able to monitor the impact of RRC states on performance in the real world is valuable for uncovering network performance issues.

Cellular carriers are one party for whom our findings and measurement approach are likely to be of interest. Implementation details which differ between carriers can lead to highly varying delays. They have a particularly substantial impact on user performance in 3G, which is still widely used by carriers around the world, although these problems persist in LTE as well. These delays are caused in part by poor interactions between unrelated control messages and RRC state transitions, and have been substantially reduced in one carrier, implying it is likely possible to reduce the prevalence of these transition delays. We have also found that the use of DRX in CONNECTED has a significantly lower impact on user-perceived performance due to the lack of demotion delays, and is a particularly effective way of reducing power consumption without significantly impacting

user-perceived performance. More generally, the use of our user performance measurement app would allow carriers to understand the impact of RRC states on user-perceived performance, breaking down trends by region or network type, in order to detect unexpected RRC state performance problems and then address them. We recommend that, overall, carriers use ongoing measurements of the impact of RRC states on the user experience to inform how they configure their networks.

Furthermore, as some of these problems are at least partially inherent in how RRC state transitions work, and may not be easy to fix, application developers would likely be interested in understanding the impact of these states. Recent work has been done on allowing applications to account for RRC state in order to reduce latency and save power on network transmissions [36, 50, 71, 34]. This benefits greatly from a more accurate understanding of RRC states as they impact application performance in the wild. In particular, they should account for the fact that different carriers are impacted differently by state transition delays, and therefore the performance tradeoffs for transmitting at different points in time will differ by carrier. For instance, for some 3G networks, transmitting during long FACH promotions can lead to even higher latencies than transmitting in PCH, making FACH an ineffective tradeoff between power consumption and performance. As the causes of these delays are complex and carrier-dependent, further development of libraries and frameworks to allow app developers to easily account for underlying network artifacts that affect performance and power consumption would be highly valuable. This is another way in which measurement-oriented systems can lead to better performance for end users.

4.6.1 Limitations of methodology

Fundamentally, the methodology we use assumes RRC state machine parameters are static. There has been some discussion of using techniques such as fast dormancy [94] or other techniques to adaptively change the RRC state machine's behavior. In particular, because we always run experiments when there is no background traffic, it's possible any fast

dormancy behavior triggered by certain traffic types would be missed. To our knowledge, based on working with a number of carriers, carriers have not yet start using fast dormancy, but of course there are hundreds of carriers which we do not work with which may have.

Similarly, it's possible carriers might change RRC state timers based on load, for instance more aggressively transitioning to a lower state when network performance is bad. This is also something we might be less likely to see due to how we schedule measurements.

We also haven't looked at whether RRC state timers vary by network conditions. We run other measurement tests around the same time we run the RRC tests, so if we saw substantial variations in RRC state machine behavior, that is something we could examine.

4.7 Conclusion

In this chapter, we examined the impact of RRC states on user-perceived performance in depth. We uncovered several previously unknown implementation artifacts that can lead to delays of up to several seconds, and have demonstrated the impact on latency and packet loss that RRC states have on network protocols and applications. We have investigated the root causes of these performance problems by examining RLC-layer messages in order to determine what configuration events and messages cause the delays observed. In doing so, we confirm the presence of these unexpected delays, and determine that, while they are partially unavoidable, they are exacerbated by complex, multi-stage state transitions and unexpected negative interactions with other control-plane configuration events. Furthermore, we discovered that some carriers have configured their RRC state machines to avoid many of these pitfalls, suggesting these problems are fixable.

In addition to identifying specific and previously unknown performance problems in networks around the world, this chapter also motivates the need for continuous, long-term and global monitoring of cellular network configurations and the impact on performance, with a emphasis on uncovering unexpected and non-ideal behavior, as these problems have

not been discovered in prior work. As applications increasingly account for underlying cellular network implementation details to avoid excessive power consumption, data usage or latency, properly understanding how the underlying cellular network affects application performance in practice is crucial. It is clear that the reality of the impact of RRC states on application performance is more complex than previously thought.

CHAPTER V

Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild

5.1 Introduction

Fueled by powerful mobile devices and ubiquitous cellular data network access, smartphone applications (*apps*) have become an indispensable part of modern life. There have been more than 100 billion mobile app downloads from the Apple App Store as of June 2015 [61]. However, battery life remains a scarce resource. Over the past 15 years, CPU performance has improved 250x while Li-Ion battery capacity has only doubled [26]. It is known that inefficient app design can lead to excessive battery drain. In particular, certain app traffic patterns, like periodic requests, interact poorly with the power-hungry cellular interface [36, 11, 98]. Despite these known problems, however, apps continue to drain user batteries.

Clearly, app behavior and how it impacts network energy[1] consumption remains a significant problem, due largely to the dynamic nature of network energy consumption which is dependent on the current phone and radio state. Through a long-term study, we measure the prevalence of network energy consumption, and these ongoing meausrements allow us to gain a better understanding of app behavior trends over time. Furthermore, we find that

[1] By network energy, we refer to the energy consumed on the device due to network traffic.

ongoing monitoring of app behavior by the phone OS, along with the management of background traffic, would greatly improve energy consumption. This supports our overall thesis that *because mobile devices experience uniquely dynamic and complex network conditions and resource tradeoffs, incorporating ongoing, continuous measurements of network performance, resource usage and user and app behavior into mobile systems is essential in addressing the pervasive performance problems in these systems.*

More specifically, in this chapter [104], we measure the prevalence of excessive mobile app network energy consumption by analyzing data collected from 20 real smartphone users and 342 unique apps over a period of 22 months. This unique long-term dataset allows us to examine the smartphone and app ecosystem in the wild. We focus on the impact of background traffic — traffic sent when the app has no UI element visible — which makes up 84% of the total network energy consumed across all users. Periodic background traffic is often power-hungry [98], but apps have flexibility in scheduling background traffic due to the absence of real-time user interaction, and can use strategies such as bundling traffic or reducing update frequencies to reduce energy consumption. We examine global trends across all apps and determine that energy overconsumption remains a pervasive problem, despite many apps taking steps to reduce their energy overhead. Furthermore, some of this traffic is likely unintentional and not useful to the end user.

Our key findings are as follows:

- We identify a significant new source of excessive background energy consumption (§5.3.1), where network traffic persists after an app transitions from the foreground to the background, sometimes for as long as a day. 30% of background traffic from one major browser is caused by this phenomenon. Over 80% of apps transmit more than 80% of their background data in the first minute after the app is sent to a background state, in total across all user devices in our study.

- We show that there is high variation in the energy overhead of apps that rely on frequent background traffic, even between apps with similar background functionality (§5.3.2). By

examining case studies of apps that require background updates, we find that the energy consumed by similar apps can vary by up to an order of magnitude. Furthermore, we find that apps studied in previous work have often improved their energy overheads but that other new apps continue to make the same mistakes. There is substantial room for improvement by adopting energy-efficient design approaches, such as batching background updates.

- By examining apps as they are used in the wild, we find that many apps are frequently not used for days, including apps with substantial background traffic. We demonstrate that the network energy overhead of these apps can be reduced by up to a half in some cases if the OS were to proactively terminate long-running apps after three days of inactivity (§5.4). More generally, we emphasize the need for apps to be aware of their foreground/background state when scheduling network requests, and our findings suggest that new suggestions for managing background traffic are likely to be highly valuable.

5.2 Data Collection and Overview

We first summarize our measurement dataset. We recruited 20 students[2] at the University of Michigan and provided each of them with a Samsung Galaxy S III smartphone with an unlimited LTE data plan. We pre-installed custom data collection software on each phone that transparently collects complete network traces. These traces include packet payloads (note we are unable to decrypt SSL traffic), user input events, and packet-process mappings. All collected data was kept strictly confidential. The data was collected over a period of 623 days (December 2012 to November 2014) with an overall raw data size of 125 GB, including cellular and WiFi packet traces and user input and context data. We focus primarily on cellular traffic in this study as it consumes far more energy than WiFi. Processes are labeled with names derived from the app package name, allowing us to straightforwardly map packets to the originating apps. In a few cases, requests are del-

[2]This user trial was approved by University of Michigan IRB-HSBS #HUM00044666.

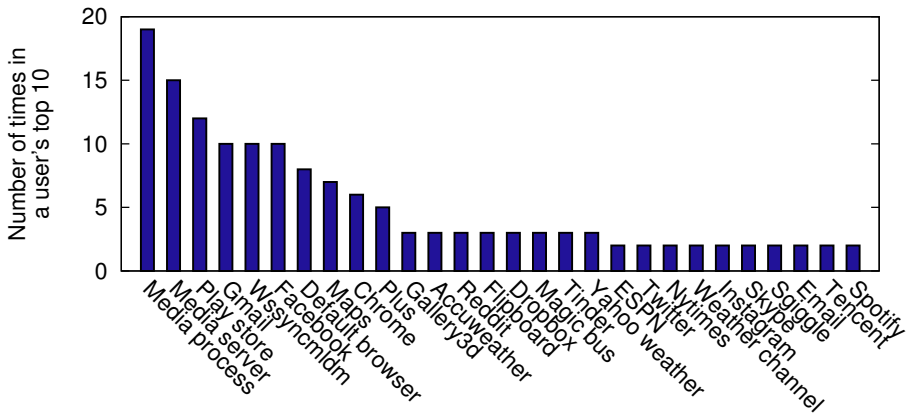


Figure 5.1: Number of times each app appears in a user's top 10 apps, ranked by total data consumption

egated to some system services such as the Media Server. We label this traffic according to the service from which it originated rather than the app which triggered it, as it is not straightforward to map back to the original app.

When calculating network traffic, we used a standard model of RRC state energy consumption [63] and allocate tail energies to the last packet that sent the traffic. We don't use the built-in Android energy manager because its way of calculating energy consumption is very approximate [25].

5.2.1 Measurement Data Overview

We next give an overview of this 22 month dataset before focusing on specific apps.

App Popularity and Diversity. Users differ greatly in the apps they use. Figure 5.1 shows apps that appear in at least two users' top-10 lists (by total data consumption). While a handful of apps are popular among all users (*e.g.,* the built-in media player, Facebook, and Google Play), users' top-ten lists otherwise exhibit significant diversity. Similar diversity of app usage was observed in previous work [37, 133].

Data- and Energy-Hungry apps. First, we examine trends in applications that con-

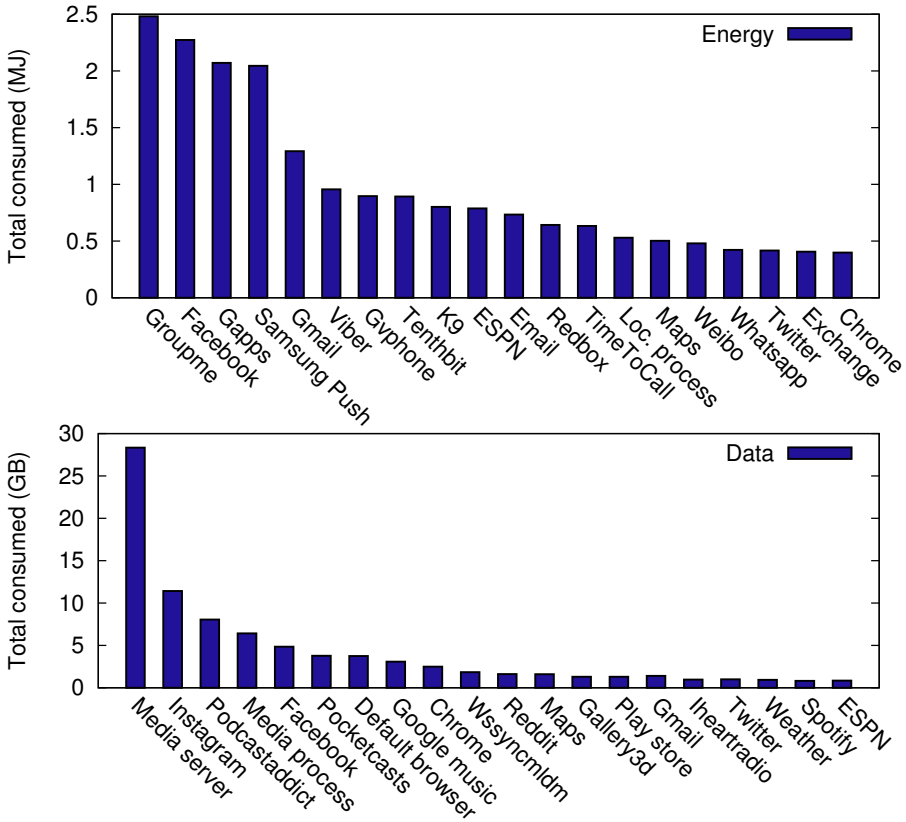


Figure 5.2: Highest cellular data and network energy usage by app across all users

sume a large amount of energy or data[3].

We summarize the top energy and data consumers in Figure 5.2. Note the top energy consumers and the top data consumers are not necessarily the same. For example, the default email app consumes network energy disproportionate to its data usage, whereas the built-in media server consumes significantly less energy per byte. As mentioned before, in cellular networks, the radio remains active for several seconds after a data transmission, consuming additional energy called *tail energy*. Since tail energy consumption is dependent on the number of traffic bursts and the time between them rather than the amount of data sent, apps sending data intermittently incur a disproportionate amount of energy.

As we evaluate the impact of each app in the wild, rather than the impact of apps in isolation, we assign any tail energy to the last packet sent during the tail period to avoid

[3]In the rest of the chapter, "energy" refers to the network energy unless otherwise noted.

double-counting energy when there are multiple concurrent flows. In this way, the total cellular network energy consumed by each device is the sum of the energy assigned to each app. This may potentially introduce some bias, however, if, for instance, network traffic from one app happens to frequently precede another. We expect this to happen rarely, though.

Longitudinal Trends. We examined trends in network usage and energy consumption over time. However, the diversity of apps, the smaller user set, users' propensity to change the apps they use over time, and changes in user behavior, obscured the overall impact of app design changes over time or any trends towards more energy efficiency. Background energy fluctuated by up to 60% from week to week throughout the study. Examining specific apps, we did determine that some apps have become more energy-efficient due to adjusting the inter-packet intervals of background traffic, which we discuss in more detail in §5.3.2.

5.3 Background Energy Consumption

Energy consumption in the background makes up 84% of the total network energy, and is thus the focus of this study. An app running in the background may run until either the user kills it manually or Android does (such as when more memory is needed). Many apps sync with a server, receive push notifications, or run updates in the background. Since no user interaction is present, these processes have much more freedom to determine when they transmit data than when running in the foreground, where they may be subject to time constraints to meet user expectations. Furthermore, there is often a tradeoff between ensuring updates are timely and avoiding wasted background updates the user never looks at. For this reason, apps vary greatly in the amount of energy that they consume in the background, even when providing similar functionality. In this section, we analyze the resource efficiency of app background network activities through detailed case studies, identifying large disparities between similar apps due to diverse design approaches. We

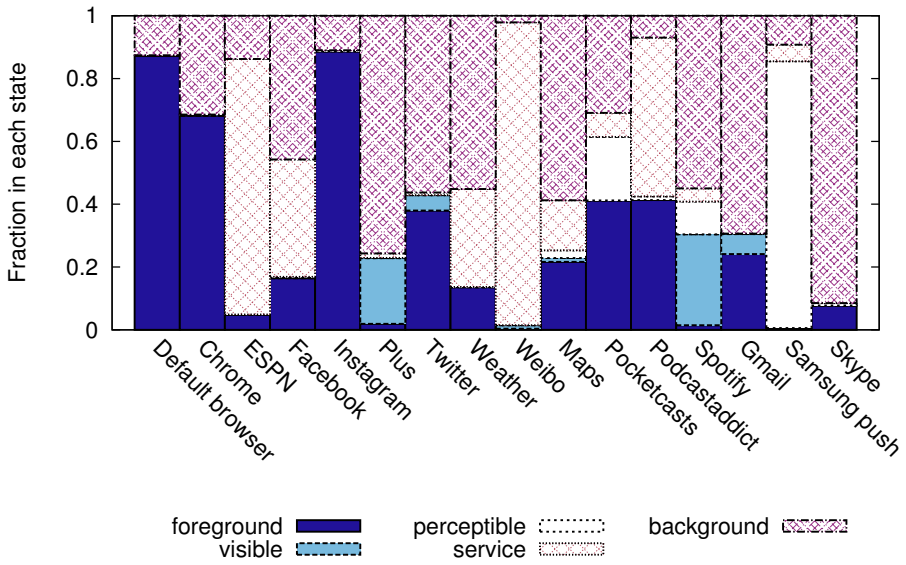


Figure 5.3: Fraction of energy in each foreground/background state, based on process codes assigned by the Android operating system

also identify several cases where large numbers of network requests are sent unnecessarily, verified through in-lab testing.

Our definition of "background" traffic is based on five main process states defined by Android [7]: *foreground*, where the process is responsible for the main UI; *visible*, where the process is responsible for a secondary UI element; *perceptible*, where a process not visible to the user may still affect the user experience (e.g. playing music); *service*, where a background process should not be terminated if possible; and *background*, where the OS will kill the app if system memory is low. We summarize the cellular energy in each of these five states for twelve data- or energy-hungry apps in Figure 5.3. We refer to the first two categories as "foreground" processes and the last three as "background" processes for the remainder of the chapter. Note that for all but three of these apps, background energy of some sort contributes more than half of the overall network energy consumed by the app. Across all apps, 84% of cellular network energy is consumed in a background state. This included only 8% of energy consumed by "perceptible traffic", as only a few users used streaming services and it is apparent from Figure 5.3 that not all apps made use of this

feature when expected. 32% was consumed by "service" traffic.

We focus on two main categories of background transfers. In §5.3.1, we examine background traffic that occurs when an app switches from the foreground to the background, and network traffic either continues after this switch or is triggered by this switch. In §5.3.2 we investigate traffic initiated automatically in the background, such as that for periodic updates, push notifications, or music streaming. We supplement our longitudinal traces with in-lab measurements to validate our findings and determine the context and purpose of the traffic in our traces.

5.3.1 Foreground Traffic not Terminated

While it is expected that some apps will transmit data in the background, such as when checking email, updating a social networking app or streaming music, other apps such as browsers are expected to mainly transmit data when the app is in the foreground. However, we find some such apps appear to inadvertently transmit data in the background. As shown in Figure 5.3, about 30% of the Chrome browser's network energy is consumed while the app is running in the background. To understand why, we examine a representative trace from the user study dataset in Figure 5.4. We have highlighted the time period after Chrome switches to the background in grey. During this time packets continue to be sent for several minutes: note that some websites also generate periodic background requests.

To validate our hypothesis that Chrome allows web pages to continue sending periodic traffic after the app is minimized, we first created a custom web page that only sends XMLHttpRequest asynchronously to a server every second. We found that the Chrome app allows this web page to transfer data when tabs are not selected and thus invisible to the user; when the screen is off; and even when the app has been sent to the background. To further confirm this problem exists in the wild, we also opened several web pages, minimized Chrome, and recorded the resulting network traces. In general, any web page which automatically refreshes content has this problem, including some ad and analytics content.

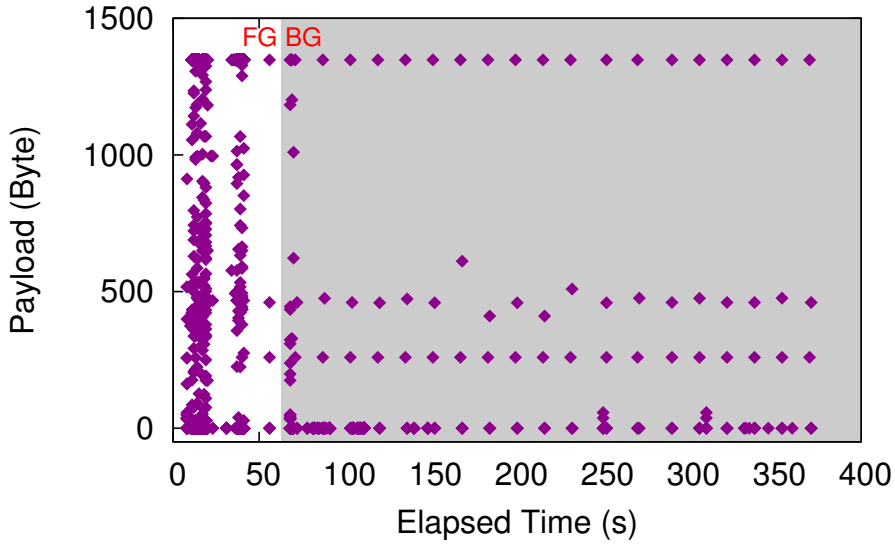


Figure 5.4: Chrome allows webpages to continue sending and receiving data in the background

In one particularly egregious case, a popular local transit information webpage sends background requests roughly every 2 seconds, indefinitely, keeping the cellular radio alive and draining the battery until the app is killed or the tab is closed.

To quantify the severity of the problem on a larger scale, we plot the distribution of the length of time during which Chrome continues to transmit data after being sent to the background in Figure 5.5. This includes flows that were started when Chrome was in the foreground but persist after Chrome is sent to the background. Each data point represents one instance of the app being minimized. In some cases background traffic flows persist for more than a day! While updating pages in the background may have some benefit to users who then revisit that page, in most cases continuing to send data for so long is likely not intended or useful behavior. Note that our data points do not include cases where the app remains in the foreground but a tab other than the one being viewed is sending data, and so the scope of the problem is likely even bigger.

We compared this behavior against that exhibited by Firefox and the default Android browser. Neither allow data to be sent when the app is in the background or the screen is

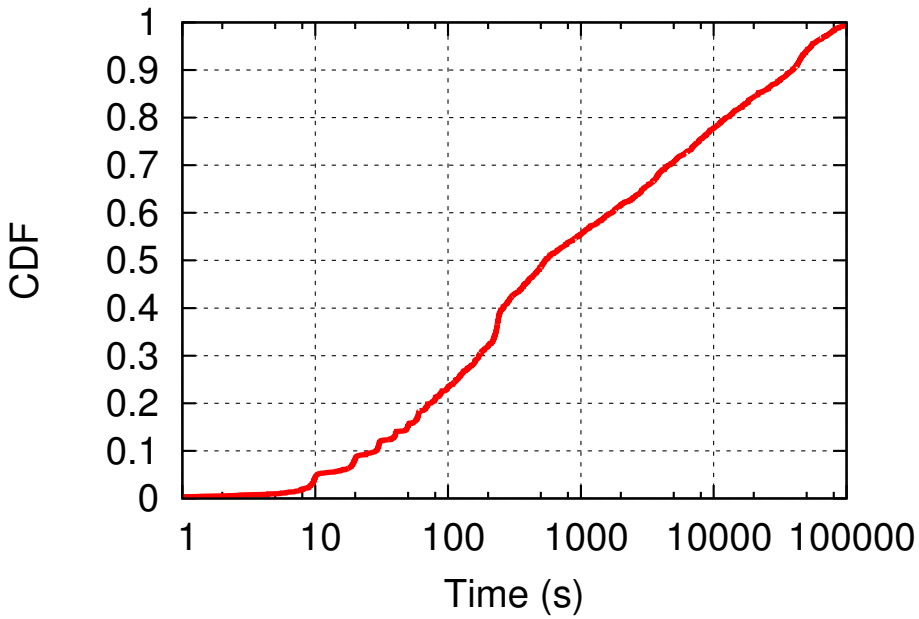


Figure 5.5: Duration for which traffic continues to be sent/received after the app is sent to the background. Each data point represents one transition to the background

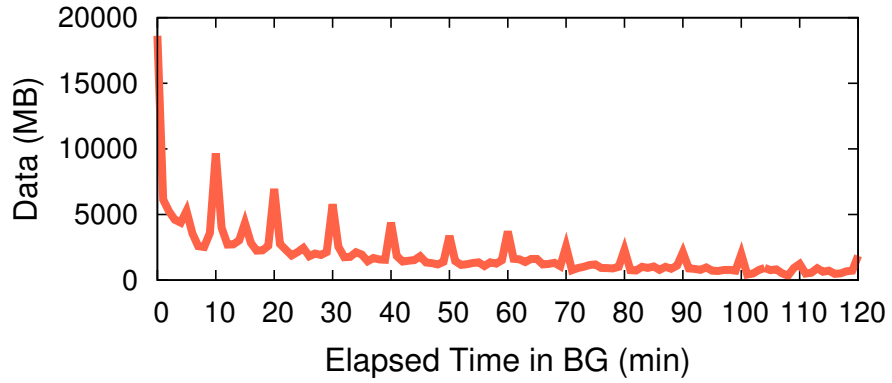


Figure 5.6: Total background data sent by all apps, as a function of the time since switching from a foreground state. Note the periodic spikes at 5 and 10 minute intervals, the large amount of traffic in the first minute, and the long tail of persisting, continuous flows

off, and Firefox blocks data from being sent by inactive tabs. To estimate the prevalence of this problem among other apps, we examine the data sent by apps in the background as a function of the time since the app was last in the foreground. As we show in Figure 5.6, the more recently the app was sent to the background, the more traffic is sent, with substantially

more traffic being sent in the first few minutes than any other time. Some of this traffic is periodic: there are peaks at 10 minute and 5 minute intervals, which are common time intervals for intentional periodic background traffic. However, there is also a non-periodic pattern, where the overall volume of background traffic falls off rapidly in the first few minutes. To estimate the prevalence of this problem, we look for apps where 80% of the background traffic is sent within 60 seconds of the app going to the background than any other time. 84% of apps meet this criteria.

There are some apps, like Dropbox, which may have valid reasons to upload content immediately after the app is closed, but in many other cases transmitting a large amount of traffic after the app is closed is undesirable. Developers of apps that send a large amount of data immediately after sending the app to the background should determine if this is expected or necessary behavior.

5.3.2 Transfers Initiated in the Background

We next evaluate data transfers that intentionally occur in the background. Even though these transfers may be beneficial to the user, depending on the frequency of user interaction with the app, the overhead of these transfers can be disproportionate. We examine a number of energy-hungry apps in depth, as well as some energy-conserving apps with similar functionality. Finally, we study a number of apps examined in previous work to evaluate how background update energy efficiency has improved over time. The energy efficiency of background transfers are primarily determined by their frequency, with small updates incurring a disproportionate amount of tail energy. Large transfers are known to be more efficient, as they make better use of available bandwidth [57]. As a result, apps with similar functionality can have very different overheads depending on the traffic pattern used.

We summarize key findings in Table 5.3.2, focusing on five classes of apps that are responsible for a substantial fraction of background updates in the user study: social media apps, widgets, music streaming, podcasts and services that provide background updates.

	MJ/day	MJ/flov	MB/flov	Avg.J/B	Update frequency	Notes
Social media						
Weibo	3500	57	0.3	190	5-10 min	Frequent, nearly-empty requests
Twitter	220	11	17	0.65	1h	
Facebook	930	14	7.7	1.8	5 min ⇒ 1 h	Previously every 20-60s [98] in 2012
Plus	180	11	8.9	1.2	1h	Rarely actively used but installed by default
Periodic update services						
Samsung Push	1500	140	2.2	64	15 min to 15h	
Urbanairship	2000	310	1.9	163	5-30 min	Library; period varies by app
Maps	190	21	55	0.38	20-30 min	Decreased to a few hours near the end
Gmail	410	20	10	2	30 min ⇒ varying	30 min in 2012 [98]; updates appear to become discontinuous.
Widgets						
Go Weather	220	2.8	5.6	0.5	5 min ⇒ 40 min	Switched push notification approaches
- widget	300	12	1.6	7.5	5 min	
Accuweather	1500	51	3.2	16	7 min but high variation	
- widget	33	1.7	18	0.094	˜3h	More efficient than the app
Streaming						
Spotify	310	50	220	0.23	5 min ⇒ 40 min	
Pandora	35	3.9	45	0.087	1 min ⇒ 2h	Previously every 1 min [98] in 2012
Podcasts						
Pocketcasts	36	4.3	2200	0.002	˜2h average	0.4 mJ per minute running.
Podcastaddict	92	2.9	750	0.004	12 min average	3.7 mJ per minute running.

Table 5.1: Case studies. Energy per flow and per day are averages over time, and one flow may not correspond to one periodic update. These can vary as apps change over time or as background apps are forced to close, and energy consumption values vary by device and carrier

We break down the energy overhead into average per-day energy consumption and average per-flow energy consumption. Note that it is not always the case that there is only one flow per periodic update, nor that for periodic updates that the updates necessarily continue for the entire day, as background applications may be forced to close for a variety of reasons.

First, we examine social media apps. These apps generally ask for updates from a central server periodically, regardless of user activity, and can thus potentially consume a large amount of energy. Apps with small, periodic background traffic (such as Weibo) have very high energy overhead and send little data, whereas apps with similar functionality (such as Twitter) have a much smaller footprint. Facebook, which had previously been identified as a heavy energy user, improved its energy efficiency over the course of our study by decreasing its background update frequency from 5 minutes to 1 hour, which is much longer than the 1-minute periodicity measured in 2012 [98]. To summarize, social media apps can vary substantially in how they manage background traffic both between apps and over time.

Applications oriented towards providing periodic background updates, such as certain push notification services, may consume a lot of energy compared to the amount of data they send. In an in-lab experiment, one third-party library transmitted nearly empty HTTP requests every five minutes for hours, but only provided one user-visible notification during this time. Another example is Google Maps, which by default provides a background location service that continuously collects anonymous location data. This service consumed up to 90% of the app's total energy usage at the beginning of the study, but the frequency decreased to once every few hours by the end. GMail also leverages periodic updates using push notifications: it has actually increased its inter-update intervals during times when it is active, but updates appear to no longer be periodic, arriving only on demand, leading to an overall low degree of energy consumption.

Widgets are a class of apps that appear on the home screen and have little or no direct user interaction. In many cases their functionality revolves around periodic background

updates (such as to keep the user updated on the latest weather). There is a tradeoff between timeliness of information and energy consumption. However, even just examining weather widgets, the difference in update frequency between two apps (and the resulting energy overhead) varies by an order of magnitude. Note also that the Accuweather app is far less efficient than the corresponding widget, as the widget updates itself less frequently, somewhat counterintuitively. Widgets and apps made by the same developers may have very different behavior.

We also examined several multimedia streaming apps. Music streaming apps were not as popular in our dataset as in prior work, but their update frequency was generally much lower than before [98], having apparently moved away from a continuous streaming model to larger batch downloads, although particularly long update frequencies may reflect users who only intermittently use these apps. Podcasts were far more popular, and we compare two popular apps. Podcastaddict consumed more energy overall, as Pocketcasts downloads an entire podcast in one chunk whereas Podcastaddict downloads smaller chunks as needed. While the latter approach may reduce data consumption if users don't finish listening to a podcast, it consumes more energy.

5.4 What-if Analysis: Preemptively Killing Idle Background Apps

In §5.3 we determined that background traffic has a substantial impact on energy consumption, and in some cases much of this traffic is from apps users are not frequently using. We propose having the OS kill background apps that have remained in the background for several days. A new permission or whitelist could address corner cases where apps (such as widgets) have a legitimate need to run in the background for an extended period of time, and OS feedback on background energy consumption could disincentivize unnecessary use of this functionality. In fact, shortly after completing this work, this functionality was added to Android M [53, 28]. We have identified a number of apps where this type of functionality has the potential to greatly reduce background traffic, although we do not

evaluate Doze itself in this chapter.

To evaluate the effectiveness of this approach, we simulate restricting background traffic after three days, and highlight six apps in Table 5.4. In row A we show the fraction of days where we see only background traffic from the app, and in row B we show the maximum number of such days that we see occurring consecutively, considering only time periods where there is foreground traffic at the beginning and end of the time period. These apps are rarely used by certain users, creating energy savings opportunities if the apps were to be preemptively killed. Row C summarizes the average savings per user of killing the app after three consecutive days. Note in particular that Weibo, which we showed was very energy-hungry, can have its network energy consumption more than halved this way.

Due to the large number of apps users in our study had installed on their phone, the impact of each app individually on a user's total network consumption was small. Thus, this would have resulted in total network energy savings of less than 1% on average overall. However, we found that for the users running Weibo, disabling Weibo alone after just three days of inactivity could have reduced their total network energy consumption by 16% on those days. Overall, how much users benefit from this functionality depends greatly on the set of apps involved and on user behavior, so it is hard to draw definite conclusions on the average benefits of our proposed system for or other systems such as Doze, but such an approach seems especially promising in protecting users from poorly optimized or buggy apps, and reducing the worst-case energy consumption generally.

5.5 Recommendations and Conclusion

Excessive energy consumption by mobile apps has long been known to be a significant problem, and background traffic continues to be a major battery drain. We have examined a significant but previously unstudied phenomenon where network traffic initiated in the foreground persists unnecessarily when the app is sent to the background. Furthermore, we have shown that improvements for known inefficiencies have not been universal, even

	Plus	Weibo	Maps	ESPN	Accuweather	Skype
A: % days with only background traffic	42%	83%	70%	13%	43%	62%
B: Max consecutive background days	40	24	84	10	18	49
C: Disable after 3 days: avg.% energy reduction	14%	54%	39%	6.2%	22%	45%

Table 5.2: Example trends in background traffic when apps are infrequently used, and simulated energy savings from suppressing this traffic

for professionally developed apps with a large user base. While we recommend that app developers continue to carefully consider the cost of the traffic they send, more is needed to improve the situation, especially for background traffic. Systems within Android and other mobile OSs that actively monitor and manage app behavior could play a major role in reducing the impact of background traffic

CHAPTER VI

Investigating using HTTP/2 Server Push for Improving Mobile Performance

6.1 Introduction

Recently, HTTP/2, which promises improvements over HTTP/1.1 in browsing performance due to new features such as Server Push, has been standardized. In Server Push, the server uses its knowledge of the website's content to push objects before the client requests them. In this chapter[1], we explore whether, and to what degree, Server Push leads to performance benefits, focusing on mobile networks where network conditions are dynamic and challenging.

Recent work has shown that improving web browsing performance is a complex problem. For instance, the performance benefits of SPDY have not been as great as expected [127], and are dependent on factors such as network performance. As most sites do not use Server Push, we take snapshots of 50 popular websites and test them locally on a server that supports Server Push. With this dataset, we find that Server Push offers far higher performance benefits on WiFi and LTE networks than Ethernet networks, and in fact on a high-speed Ethernet network, Server Push is not particularly useful. This motivates focusing on mobile devices.

[1] In submission to WWW 2017. Authors: Sanae Rosen, Bo Han, Shuai Hao, Z. Morley Mao, Feng Qian.

To understand the impact of network characteristics, we perform experiments on mobile phones, as well as controlled experiments with artificially limited network conditions on wired networks where it is easier to explore the impact of limited network performance in a systematic way. We find that Server Push performance is highly dependent on network features such as the loss rate and latency, and in some cases Server Push can even be detrimental. Individual websites also vary greatly in how they are impacted by the network, due to differences in loading patterns and the impact of rendering and computation. In absolute terms, savings are typically around a few hundred milliseconds, with some pages seeing benefits of seconds[2]. In this chapter, we explore these factors in depth and provide recommendations as to when Server Push would be most useful.

We examine how other factors can impact Server Push performance as well. Pushing the entire website, rather than a few Javascript or CSS files, is necessary to see substantial performance improvements. Websites split among different domains are a challenge for Server Push. We also find that the limited processing power of mobile phones can limit the benefits of Server Push, and that more computationally powerful devices would likely benefit more. Finally, we find that Server Push offers modest energy reductions of 9% on one LTE network on average.

Supporting the main thesis, the *complex network conditions* we consider are those that impact Server Push's performance on mobile devices. We perform a measurement study of the impact of these measurements, and suggest that web pages should use *measurements of network performance* and of how network conditions and website attributes impact the website's performance with Server Push. Thus, mobile pages can *incorporate the results of these measurements* in order to ensure that people on mobile devices can benefit from Server Push.

Our main contribution is the first study focused on understanding Server Push performance (particularly on mobile devices) and how and when Server Push should be used.

[2] 100 ms is often cited as having a financial impact [52]

More specifically, our findings are as follows:

- Server Push is more effective on high-loss or high-latency networks, such as cellular and WiFi networks, as compared to typical wired networks, motivating its use on mobile devices.
- Pushing all content on a website is on average significantly more effective than pushing a handful of Javascript or CSS files.
- Domain sharding and content otherwise split among multiple servers is a significant impediment to Server Push's effectiveness.
- Server Push works best with high latencies and loss rates, offering a median 16% improvement in PLT with a 2% loss rate and a 14% improvement with a round trip time of 100 ms, but excessively poor network performance hurts Server Push.
- Server Push doesn't improve performance on every website, and so should be used judiciously: measurements will likely play a major role when determining how to deploy Server Push.
- Server Push reduces LTE power consumption on mobile devices by about 9% and has no significant impact on WiFi power consumption.

6.2 Dataset and Methodology

Since so few sites make use of Server Push, we conducted controlled experiments using mirrored sites hosted locally, to test the impact of network performance and run other experiments to understand Server Push under a variety of circumstances.

We mirrored a total of 50 sites from the Alexa top 500 starting with the most popular, with essentially identical websites omitted. Sites were copied using the Firefox Scrapbook extension [111], including all Javascript, and where necessary manually edited to remove

Table 6.1: Summary of web page characteristics.

	Min value	Median value	Max value
Num. objects	6	64.5	440
- Images	0	25.0	435
- Javascript	0	7.0	51
- CSS	1	1.0	7
Page size	46 Kb	1.86 Mb	10 Mb

popups and redirects that interfered with automated analysis, and to ensure where possible content is loaded locally. Sites that could not be hosted locally without substantial modifications were skipped. The mobile version of the page was used in our analysis. We summarize some statistics on the pages in Table 6.1.

These sites were hosted on a server using nghttp2 [85], which has a complete implementation of Server Push. Its library is now used by other servers, in particular Apache [9]. We collected at least 5 measurements for each experiment, randomizing the order in which the sites were visited. Except for when exploring how much content to push, all content was pushed as we found that to be the most effective approach, as we will show in the next section. Nghttp2 prioritizes the HTML page first, then the CSS files, then the Javascript files, then images and other content, according to their documentation[3]. We leave exploring other prioritization approaches to future work.

On the client side, all experiments were run in an up-to-date Chrome browser. Our first set of experiments were carried out on two mobile devices: a Samsung S5 and (to compare with an older phone) a Samsung S3. Page load time measurements were collected using Chrome's debug interface accessible by plugging the phone into a computer and going to `chrome://inspect#devices`. The page load times are those listed by Chrome. All experiments were conducted with caching disabled, and the values of three measurements were averaged, rather than five, due to the manual effort involved.

[3]https://nghttp2.org/blog/2014/04/27/how-dependency-based-prioritization-works/

Controlled experiments, where latency, loss rates and bandwidth were varied, were carried out over Ethernet on a desktop (except for the bandwidth experiments, which were performed over Ethernet with a laptop). This allowed us to be able to precisely control each of these variables. However, in most cases we are examining the sorts of network conditions that are more typical for mobile devices than for desktops or laptops on modern Ethernet connections. For these, page load time information was collected by a script which connects to Chrome's remote debug interface through a JSON API[4]. Unfortunately, this API was not available on mobile devices, but the page load times indicated in the GUI-based debug interface used for the mobile phone experiments is equivalent. Latency and loss rates were varied using `tc`, a standard Linux utility that allows different network conditions to be emulated. Bandwidth was varied using the built-in Mac OS network emulation tools on a laptop. We also collected performance data for WiFi and tethering on the same laptop.

To calculate the energy overhead of Server Push, we made use of the power model in a recent paper [87], using the same parameters as in that paper. We collected tcpdump traces from our page loading experiments and calculated the power impact of Server Push with both the WiFi and cellular network.

6.3 Web Performance

There are a number of factors that impact Server Push performance that we examine. We first look at the impact of pushing differing amounts of content, and demonstrate that the current trend of distributing a web page's content across many domains introduces significant challenges for Server Push. We then examine how various network conditions can impact Server Push performance, demonstrating Server Push is mainly helpful for networks likely to experience high latency or loss rates, such as mobile networks. We then examine differences between websites that do well or poorly with Server Push. We summarize the

[4]https://developer.chrome.com/devtools/docs/debugger-protocol

Table 6.2: Summary of findings.

What to push	
Push as much content as possible, not just a few small files.	Fig. 6.1
Content divided across domains is a major problem (proxies are a possible solution).	§6.3.1
Server Push increases the loading time for the first object and so can harm performance.	Fig. 6.4
Network factors	
Server Push works best on WiFi and LTE for a given device; mobile devices are limited by their processing power.	Fig. 6.2
Server Push works well with latencies around 100 ms.	Fig. 6.5
Performs well with loss rates between 0.5% and 2%; better with high uplink losses.	Fig. 6.6
High losses and latencies combined don't do well with Server Push.	Fig. 6.7
Performs slightly better with low bandwidth.	Fig. 6.8
Energy impact (§6.5	
Server Push improves LTE energy consumption slightly (by about 9)%.	Fig 6.13
Server Push has almost no impact on WiFi energy consumption.	Fig 6.13

findings in this chapter in Table 6.2.

6.3.1 Impact of Content Pushed

First, we examine existing sites that use Server Push and what they do, then determine a good strategy for determining what content to push.

Server Push is still rarely used in practice. We used the nghttp2 client to crawl the top 10,000 sites according to Alexa in September 2016. Our client attempted to connect using HTTP/2, and recorded whenever a PUSH_PROMISE header is seen which indicates the start of an object being pushed. We found five sites which used Server Push (plus one more that logged a Server Push request in our automated testing, but not when we examined it manually a few days later).

We looked at each of the five websites using Server Push in Google Chrome and manually examined what was pushed. They took a variety of approaches: one site pushed every-

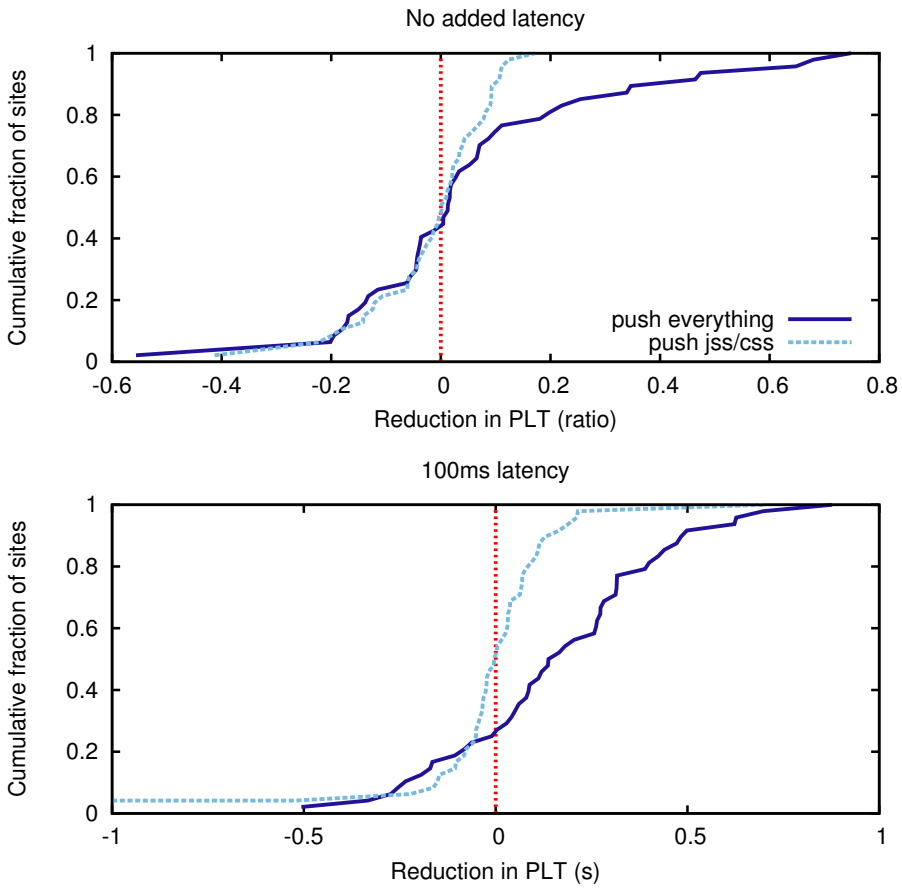


Figure 6.1: Pushing all content versus pushing only Javascript and CSS files. PLT stands for Page Load Time.

thing except dynamic content (`www.neobux.com`); other sites pushed just one Javascript file (`www.cloudflare.com`; `www.yoob.com`); another pushed its Javascript and CSS files (`www.kroger.com`) and one pushed a selection of Javascript and image files, but not all (`www.namepros.com`).

We next examine two cases: pushing only CSS and Javascript files, and pushing everything. For this experiment, we looked at our locally mirrored websites, as unfortunately we cannot set the push policy for real sites in the wild. We emulated a high-latency network (such as a cellular network) by adding 100 ms to the latency on an Ethernet connection, in addition to testing on a low latency network (1 ms ping, about 30ms total for a small object to load). The results are shown in Figure 6.1. Clearly, Server Push performs a lot better

when we push everything, and *we recommend pushing more content where possible*. In particular, pushing everything seems to matter more on higher latency networks: pushing just Javascript and CSS give similar results on the two networks, but pushing everything shows very different results.. Using nghttp2's hard-coded priorities, HTML was pushed first, then CSS, then Javascript, then all other content.

There is one major complication in pushing everything. In practice, content is often split over several domains, whether due to third-party advertisement, domain sharding, or other reasons. To examine the impact of this problem, we went through each page and manually determined whether each object came from the same domain or a different domain. We then moved that content to another server, and didn't push that content. In the median case for the 50 sites, we moved more than 95% of the objects to another server. Most major websites host images and other content separately, although a few websites were mostly unchanged. We found almost no benefit from Server Push when we split the content like this: Only 25% showed any measurable benefit, and less than 15% showed more than a 10% performance improvement. Clearly, the way in which websites are architected today are a major problem for the deployment of Server Push, and which should be addressed in future work.

HTTP/2 promises to make domain sharding unnecessary [55], and it is generally recommended not to use domain sharding in HTTP/2 [86]. Recent work has also discussed that content on a website served from outside a CDN can cause substantial performance degradation [43], motivating further keeping content on one server wherever possible. However, in the short term it's unlikely that websites will be drastically re-architected.

Other approaches, such as a mobile-specific proxy similar to Flywheel [6], which would enable Server Push as well, are probably more realistic deployment scenarios in the short term. At least one HTTP/2 proxy exists that supports Server Push, which is intended to provide HTTP/2 for servers using other protocols [85]. Other methods of supporting third-party and remote content through Server Push should also be explored. It might be possible

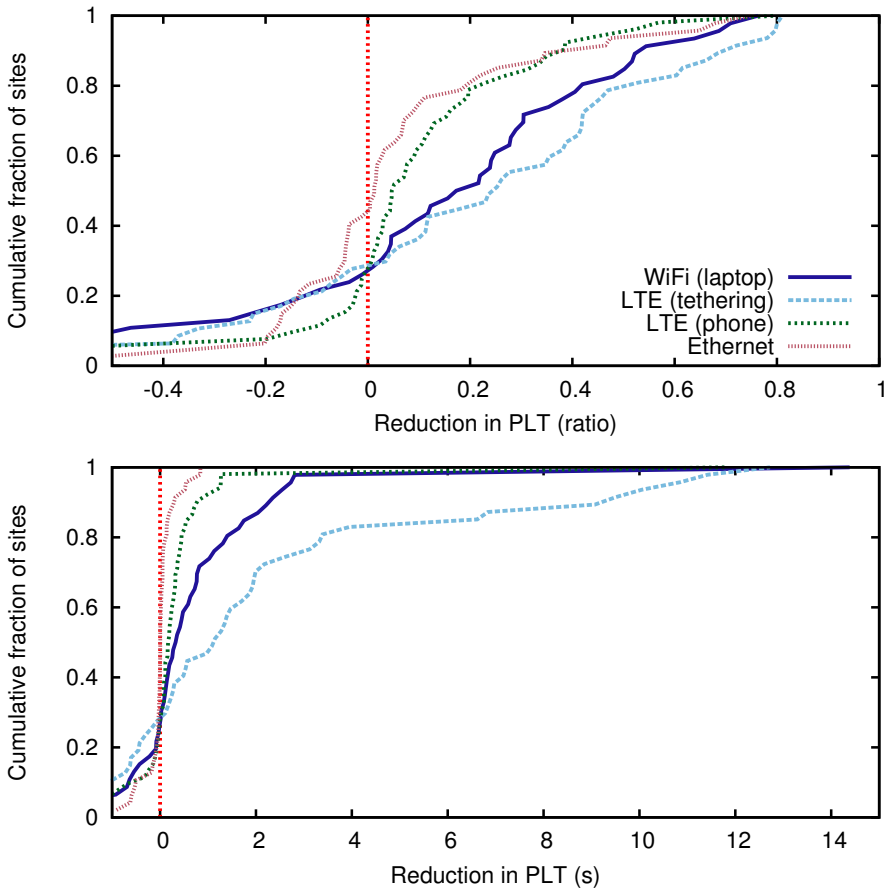


Figure 6.2: Server Push PLT savings for mobile websites on a variety of networks. Negative values cut off at -0.5.

to alert third-party servers as to what to push. For instance, perhaps a small object near the top of the HTML page could be loaded to alert the third-party server that it should push content. We leave developing a solution to dealing with third-party content to future work.

6.3.2 Impact of the Network

To understand how network conditions impact Server Push, we vary network performance parameters in a controlled manner by adding latency, loss and limited bandwidth to an Ethernet connection.

First of all, what networks should we focus on, when deploying Server Push? We first examine several typical connections: the local LTE network, a home WiFi network, and

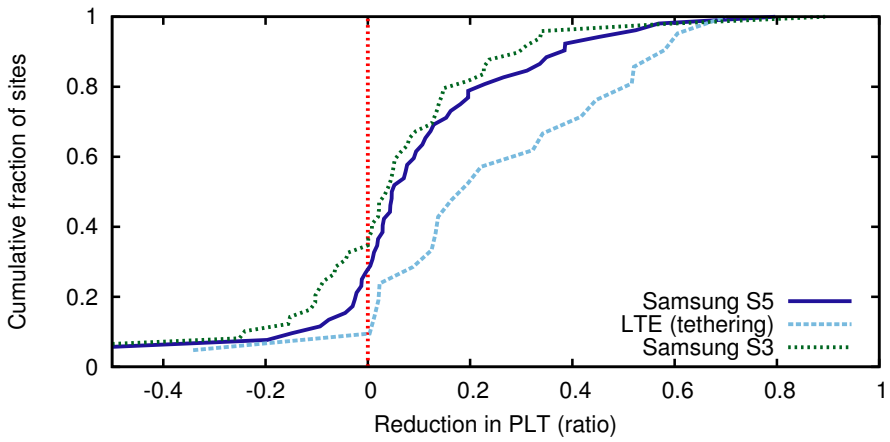


Figure 6.3: Impact of device processing power on Server Push.

the Ethernet connection in the lab. Typically, the LTE network experiences latencies of roughly 90 ms, the WiFi network of about 25 ms, and latencies of about 1 ms for the Ethernet network (the server hosting the pages was on campus). The time to first byte for the first object with a 1 ms latency was around 30ms for a small object. As we show later in this section, even more typical latencies of up to 50 ms perform similarly over Ethernet. As we show in Fig. 6.2, Server Push can greatly improve performance — reducing page load time (PLT) by up to 80% in the best case — but in many other cases, Server Push does not help. LTE and WiFi benefit more from Server Push than Ethernet does. For this reason, we believe it makes sense to focus on mobile devices when determining when and how to use Server Push. However, on an actual mobile device on LTE, savings as a percent of the original PLT shrink, although they remain higher than a laptop using Ethernet. Recent work has found the lower processing power of mobile devices makes the loading time less network-dependent [82]. We still see performance improvements, though. Furthermore, despite this limitation, there are savings of hundreds of milliseconds on average, and in some cases seconds. Even saving 100 ms can have a high benefit [52].

Next, we tested a second mobile phone (a Samsung S3) on a different carrier (still LTE), to make sure our results are not specific to a particular device or network. We show the results in Figure 6.3. This older phone performs slightly worse, as expected, but the

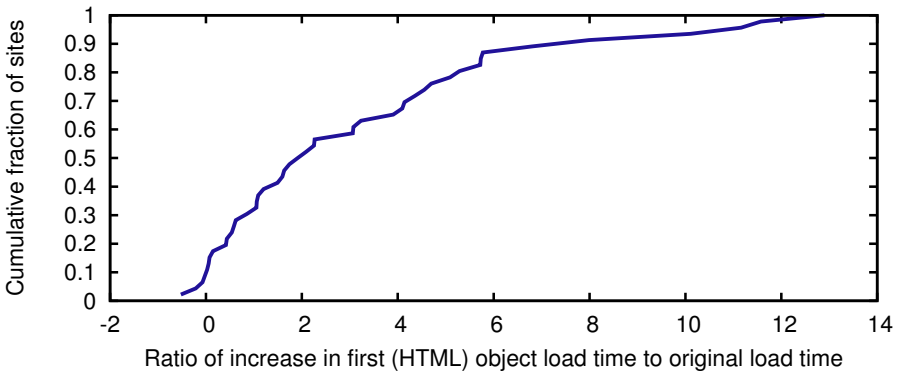


Figure 6.4: Relative increase in the loading time of the initial HTML page with Server Push.

overall shape of the curves are very similar, suggesting our results on the phone are at least somewhat representative. The laptop with tethering does far better though; substantial improvements in processing power would be needed to see the full benefit of Server Push on mobile phones.

It is also apparent that the benefits of Server Push can be lower than zero (although we cut off the negative values at -0.5 in the graphs to focus on the positive values, as there is a very long tail on the negative values, and we assume websites wouldn't use Server Push if it performs badly for them.) We examine why Server Push does not show positive performance benefits in every case. Server Push tends to make the initial HTML file slower to load, sometimes quite substantially, as shown in Figure 6.4. This is the case even though nghttp2 sends HTML traffic with a higher priority — we seem to still see interference from other requests. For Server Push to be beneficial, it has to offer savings greater than this cost.

Next, we examine individual network performance factors, varying the bandwidth, latency and loss rate of an Ethernet connection while holding the other variables constant. We use Ethernet to measure the impact of each variable in a controlled manner, as WiFi or LTE are likely to show a wider range of latencies and losses while we run these experiments, making it harder to draw a firm conclusion.

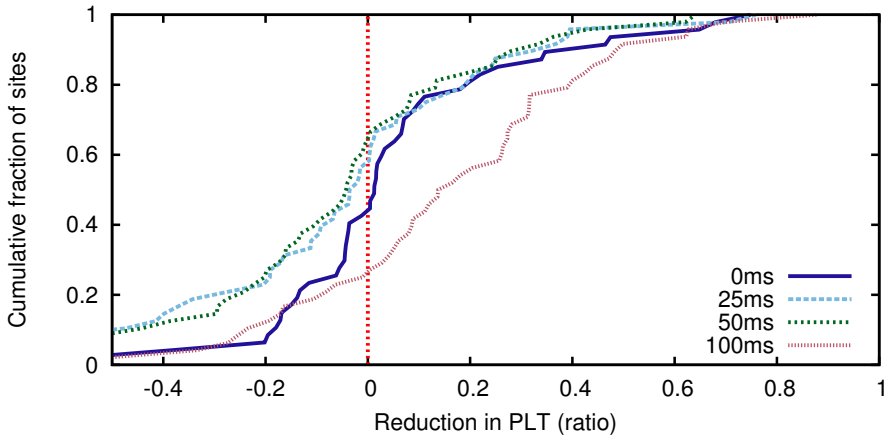


Figure 6.5: Impact of network latency on Server Push. Latencies shown are from ping; at 0ms, a small object takes about 30ms to load including server processing etc.

Latency: Fig. 6.5 shows the results of varying the latency. There is a sudden jump in performance between 50ms and 100ms, where web pages are likely becoming more network-bound. At higher latencies, fetching content earlier has a bigger impact.

Packet loss: The impact of packet loss is fairly substantial as well, as shown in Figure 6.6. As loss increases, the performance of Server Push for the better performing websites increases for loss rates up to 2%, and after that it decreases sharply. At about a 0.5% loss rate, websites almost consistently do better than with no loss. Also, sites that perform badly tend to perform worse at high loss rates. In the top subfigure, a 1% loss rate means a 1% loss rate for uplink packets and a 1% loss rate for downlink packets.

We then look at uplink and downlink losses individually in the bottom part of the graph, and we see that uplink losses benefit Server Push much more than downlink losses do. With Server Push, far fewer requests are made on the uplink and so there are fewer opportunities for these requests to get delayed when pushing content rather than waiting for requests to the client. With 3% loss rates, both start to see the PLT savings go down again.

If we look at high loss rates and latencies combined, as in Figure 6.7, Server Push also doesn't perform well. In fact, we see decreasing performance with increasing latency when there's a substantial loss rate, rather than the other way around. Note how the distributions

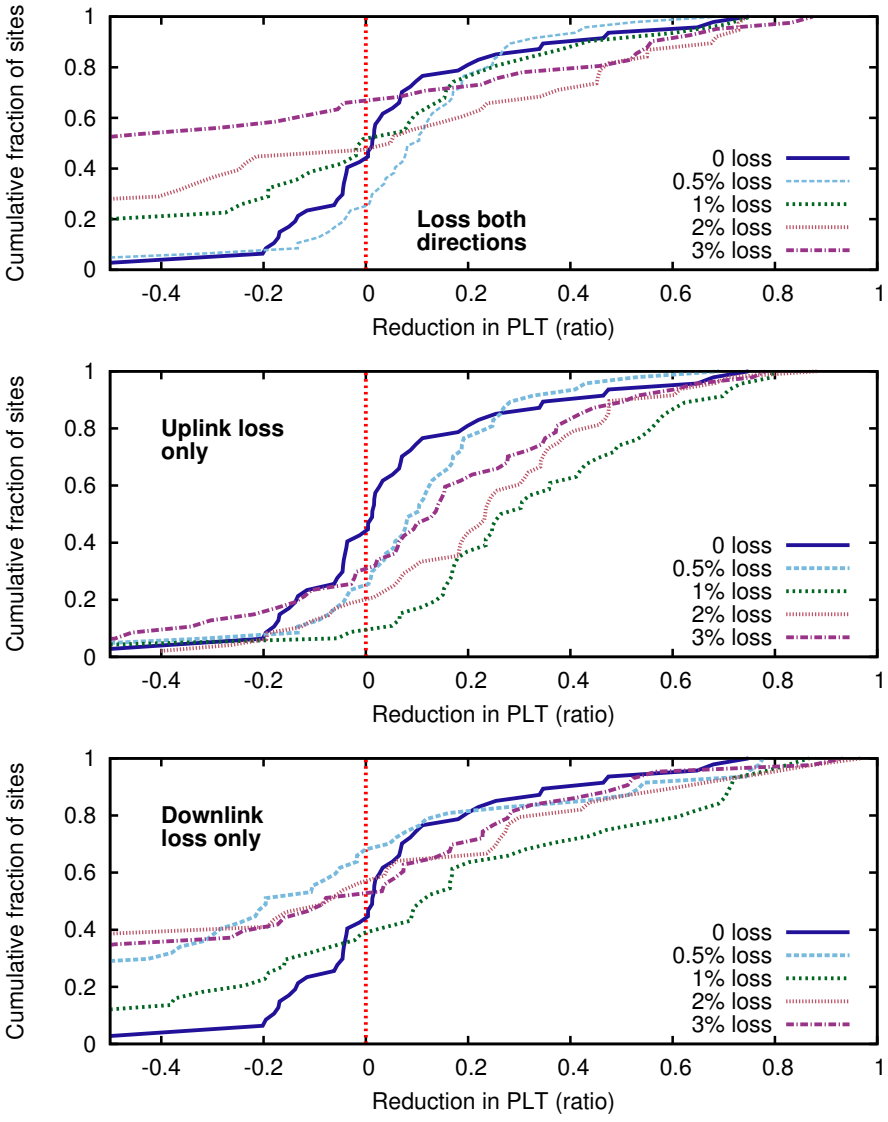


Figure 6.6: Impact of network packet loss on Server Push.

get much wider at high loss rates. In general, high latencies are more common on mobile networks than high loss rates.

Bandwidth: The amount of available bandwidth has a smaller impact, as can be seen in Figure 6.8, even with an added delay of 30ms which made differences more pronounced. If bandwidth is the limiting factor, pushing more content at a time generally will not have much of an impact on this limitation.

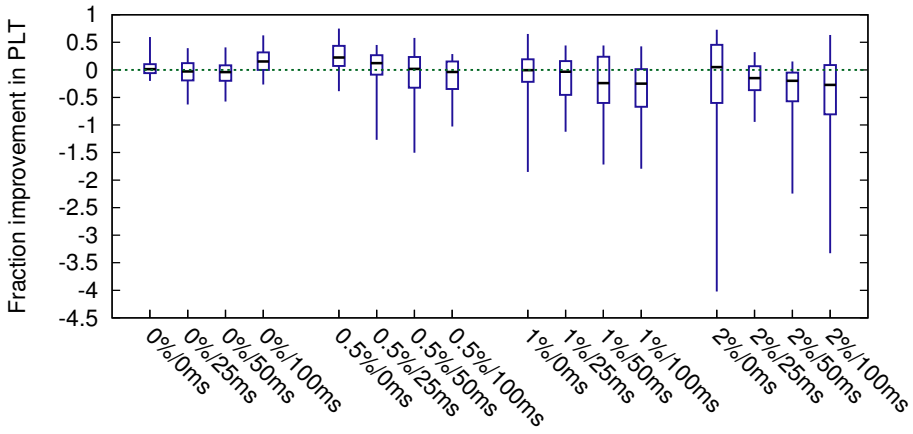


Figure 6.7: Impact of combined high latencies and loss rates. The loss rate is listed first, then the latency, as a percent and number of milliseconds, respectively.

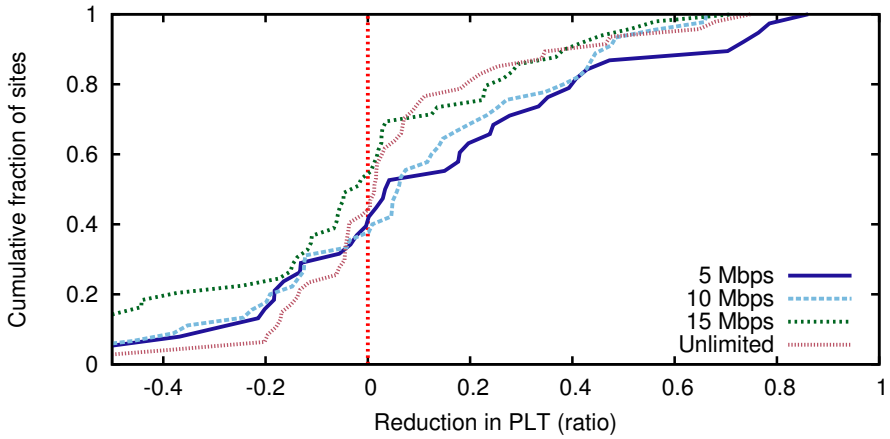


Figure 6.8: Impact of bandwidth at 30ms latency.

6.3.3 Impact of the Web Page

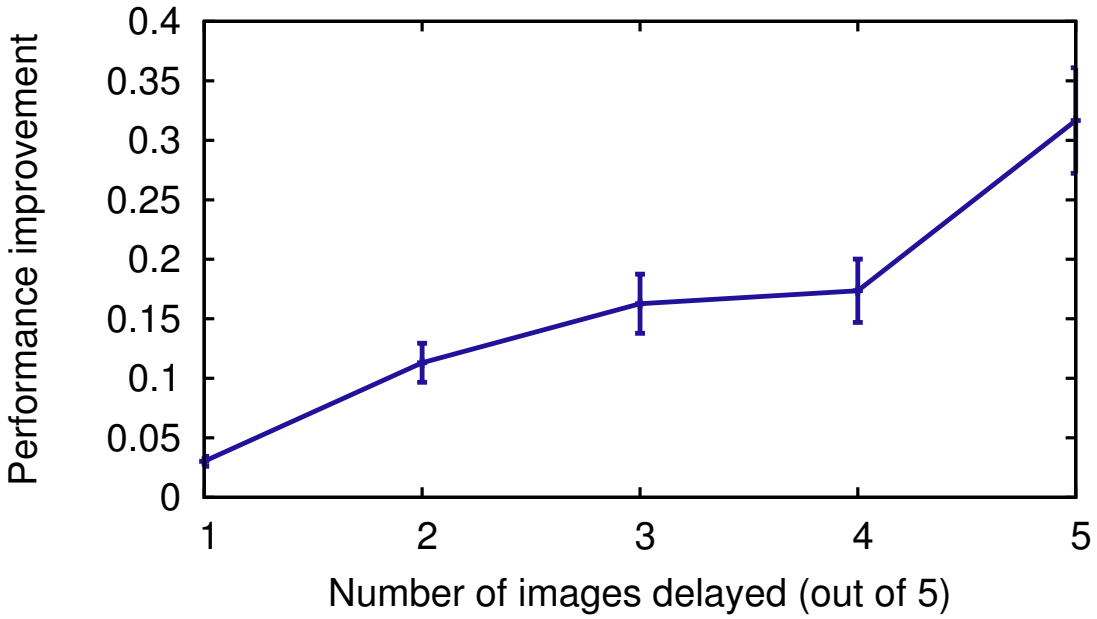
Overall, it is difficult to identify any one factor that lead to some websites performing better than others. Websites are complex and it appears a number of factors have an impact. To understand what factors might impact website performance, we performed a number of simple controlled experiments. The first was based on our observation that websites that benefited from Server Push often had content that loaded late. We created a page with a significant amount of Javascript computation, and with 5 images to load. We then varied

the number of figures that loaded after the Javascript, and show the results in Figure 6.9(a). Content that loads after Javascript or substantial page rendering can be fetched early by Server Push, resulting in higher performance savings as the download time can be hidden behind the compute time. However, this network traffic has to have a substantial impact on the overall loading time to matter.

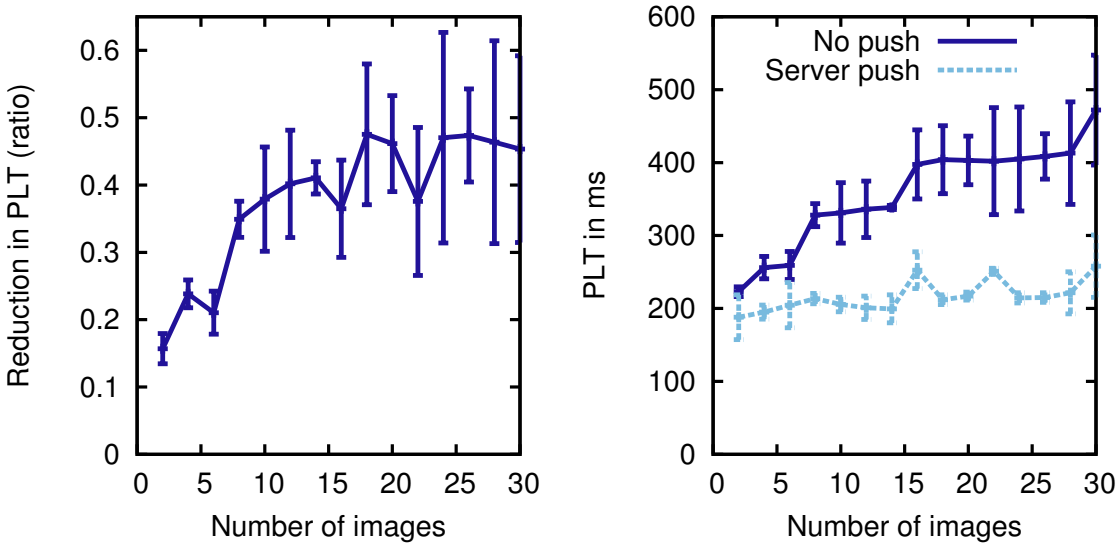
We examined, through controlled experiments on simple websites we made ourselves, a number of other website characteristics. The main one which appeared to also have an impact was the website size, shown in Figure 6.9(b). As we increase the number of images, the benefits of Server Push increase as it is able to reduce the resulting loading time increase. However, we did not find the size to consistently lead to better Server Push performance on real websites: larger websites were more complex and perhaps thus more compuation-dependent.

6.3.4 Summary

Overall, we've found that Server Push works best with mobile devices, although the performance limitations of these devices impact Server Push's performance. Even so, the performance benefits over wireless networks are higher than over Ethernet, at least for a reasonably high-speed network. Results are similar across devices and networks. However, Server Push introduces a delay to the first HTML object, despite prioritizing it, and so not all websites benefit from Server Push. Looking deeper into the network characteristics that lead to good Server Push performance, high latencies benefit from Server Push, as do high loss rates — but only to a point. Bandwidth plays a smaller role. In terms of web page structure, the main factor that predicts good Server Push performance appears to be how long it takes the second object to load.



(a) Pushing content for sites with a significant amount of computation, with 1-5 of 5 figures loading after the Javascript computation and the rest loading before



(b) Impact of web page size on Server Push with a 100 ms delay and 10 Mbps bandwidth

Figure 6.9: Examining, through controlled experiments, the impact of web page structure.

6.4 Case Studies

We next examine examples of real sites and how they load, in a variety of circumstances, to better understand how Server Push affects performance. These experiments were carried

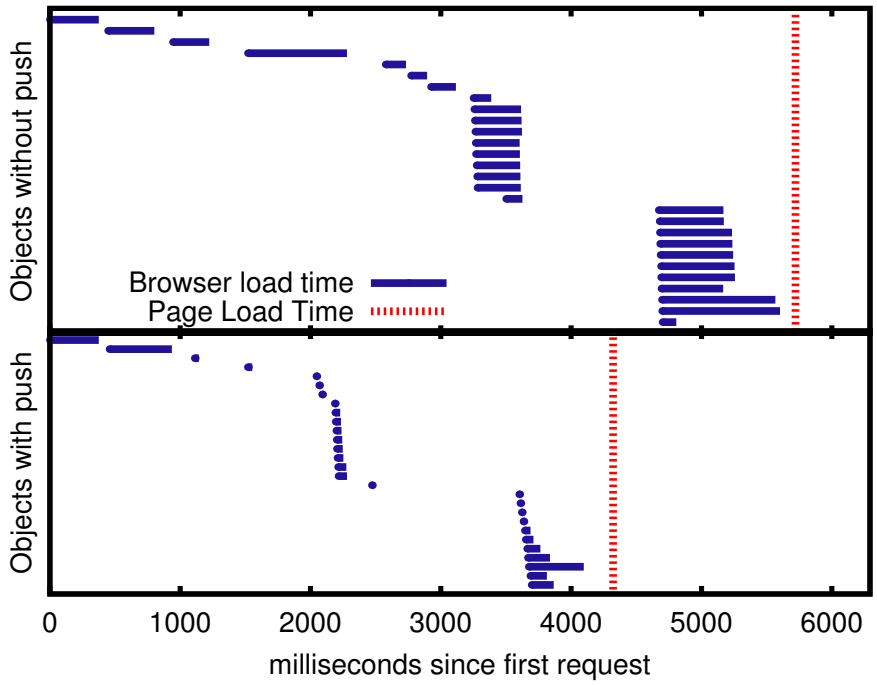


Figure 6.10: Waterfall diagram of loading the Ikea web site in a phone browser.

out on a phone over LTE, unless otherwise stated. Our plots show only the browser load times and not the push download times. The browser load time is the time for the browser to fetch and load a given object, from when the browser first makes the request to when it is fully loaded, extracted from the HTTP Archive (.har) file saved from the debug view in Google Chrome. When content is pushed, the browser load time for each object should be much smaller, as the push time is not shown.

First, we show the mirrored Ikea page in Figure 6.10. This is a fairly straightforward case: loading happens over several stages, and especially after the first few objects, the loading time is shorter because Server Push has already delivered (most of) the content. Rendering and other computation does play a major role in the page load time, but reducing the network loading time helps substantially. Note that a significant amount of content is loaded after some computation, like in Figure 6.9(a).

The BBC website also loads in batches, but is more complex, and more time is spent

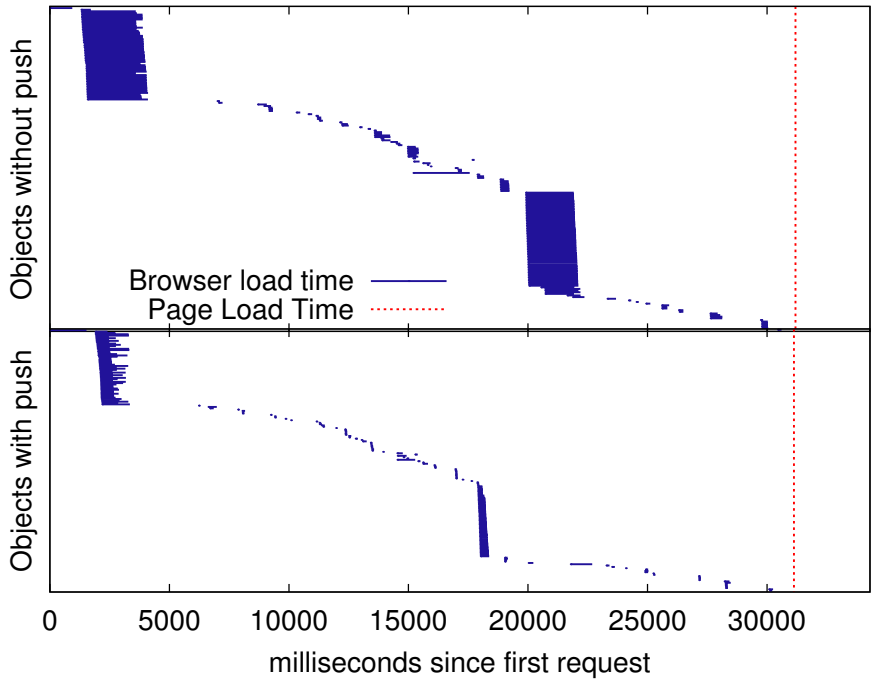


Figure 6.11: Waterfall diagram of loading the BBC website in a phone browser.

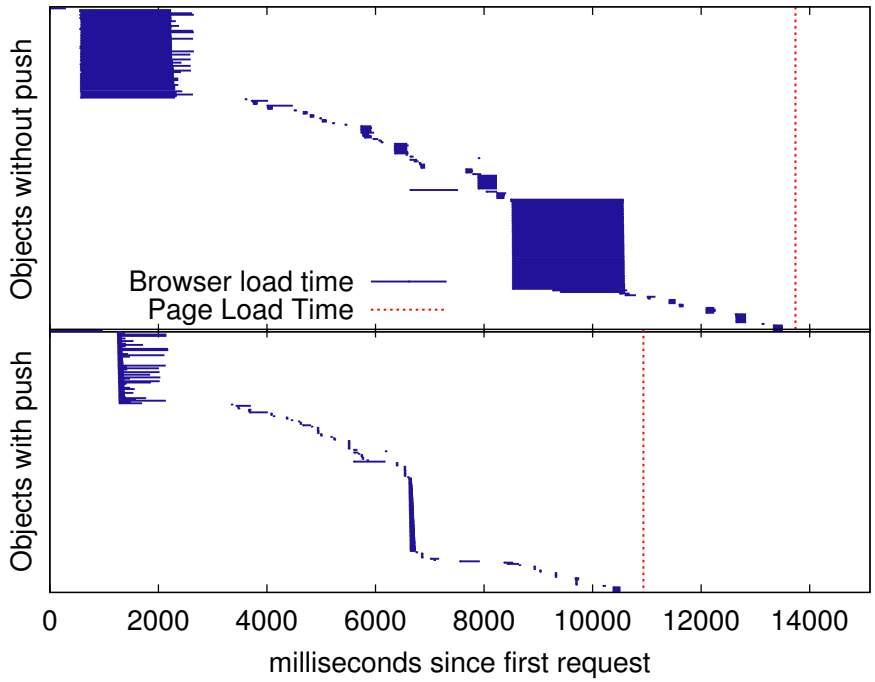


Figure 6.12: Waterfall diagram of loading the BBC website over WiFi on a laptop.

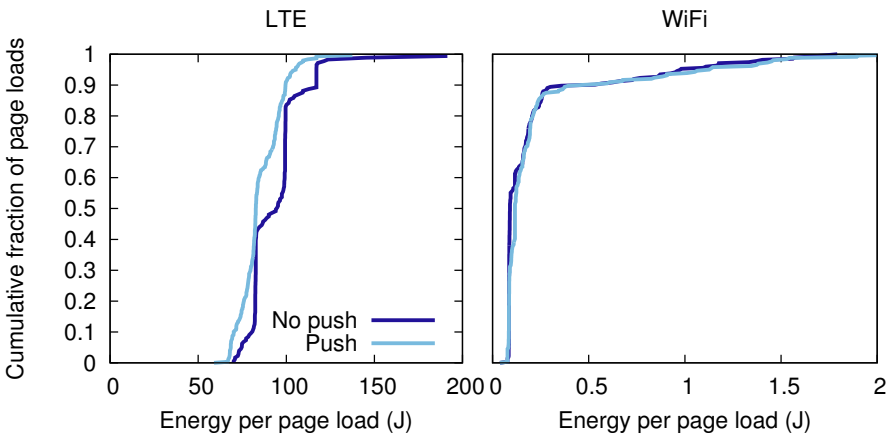


Figure 6.13: Radio energy trends for mobile devices. Server Push offers some savings for LTE only.

in computation. We show the results in Figure 6.11. While it's apparent that the network loading time is reduced substantially, only the computation and rendering time is on the critical path in this case. With a laptop on WiFi, we see better performance for Server Push, as shown in Figure 6.12. Server Push is not able to reduce the loading time of the first burst of traffic, as there is too much in total to load. The last chunk of network traffic has been fetched by the time the browser requests it. Because of the last burst of network traffic being fetched, close to two seconds are shaved off, or a little under 20%.

6.5 Energy Impact of Server Push

We next consider the impact on radio energy on mobile devices. Energy consumption is the biggest concern among users of mobile devices [93]. Ideally, if Server Push reduces the time that the radio is active, by essentially "compacting" requests into one burst, then it should result in energy savings.

We collected packet traces over WiFi and LTE, and calculated the energy consumed by the radio based on the model given in a recent paper [87]. The results are shown in Figure 6.13. We found there to be modest but observable power savings with LTE, but not

with WiFi. WiFi has very short timers to stay awake after data is sent, and so WiFi is less dependent on the distribution over time in which requests are made. With LTE, however, the radio stays active for some time after data is no longer sent. We see a 9% improvement on average, which is less than the PLT savings, as fixed-length tail timers mean that as a percentage the reduction in page load time won't be the same as the radio energy savings.

6.6 Discussion

In this chapter, we found that Server Push performs better under high loss and high latency conditions, and generally performs better on wireless networks (WiFi and LTE). While mobile devices tend to benefit less from Server Push than laptops for a given set of network conditions, the fact that they are more likely to experience poor network conditions in the first place means these are good devices to target.

We have focused on the performance benefits of Server Push, but there are other obstacles to Server Push being effectively and commonly used. The first is the problem of client caching. Since the server doesn't know what the client has cached, there is potentially more overhead in terms of data wasted with Server Push. There have been various solutions proposed, however [48, 85].

The second challenge, conversely, is dynamic content: we may not be able to determine what to push, and we leave predicting dynamic content to future work. The last challenge is that we assume that it's possible to determine what to push. When content is produced by a single server, this should be true. However, predicting third-party content may be harder. An interesting approach might be to have a proxy load the page and forward content using Server Push.

There are also a few questions we haven't addressed. We use the default order for pushing content, and don't explore alternate orderings based on, say, dependencies. There are a number of papers that have proposed systems to analyze web page loading dependencies [16, 83, 75]. An interesting direction for future work would be to explore and adapt

Table 6.3: Summary of recommendations

Push as much content as possible	Fig. 6.1
If possible, host everything on one server	§ 6.3.1
Develop tools for pushing third-party content	§ 6.3.1
Test if websites actually benefit	§ 6.3.2
Use for sites that are not compute-heavy	§ 6.4
Use Server Push when high latency/loss expected	§ 6.3.2

these methods to further optimize loading time by avoiding pushing content unnecessarily.

Finally, we analyze mirrored websites, because there are almost no real websites using Server Push. It is possible, however, that as Server Push is deployed on production servers, other factors affecting Server Push will become apparent. We hope that these findings will motivate more sites to make use of Server Push.

We have focused on web browsing, but Server Push may be applicable to many other uses of HTTP. We have not considered the impact on video content — we did not play any videos on Youtube or other video services we examined. Server Push could help the browser start to buffer content right away, but aside from that, we do not expect it will be very useful for this class of content, as the round trip time is unlikely to be the main limiting factor. We also have not examined mobile apps, which may be able to leverage Server Push even if they do not have the typical structure of an HTML page with images and other content embedded in it. We leave examining this to future work.

Recommendations: We summarize our recommendations in Table 6.3. Our first recommendation is that, if you are using Server Push, you should push as much as possible. We have not exhaustively examined all possible ways of choosing what to push — perhaps determining what is on the critical path for a particular device and set of network conditions would be helpful — but overall, pushing images and other frequently larger content, rather than just CSS or Javascript files, results in better performance.

Relatedly, the problem of third-party content is likely to be a significant impediment

to the success of Server Push. It probably isn't realistic to recommend that websites stop using third-party content, which suggests that examining ways to effectively deal with this content would be a valuable direction for future research. In the meantime, this introduces yet another reason to serve as much content as possible from the same server when using HTTP/2.

We also found that many sites do not benefit from Server Push at all. As a result, we recommend that the performance benefits of Server Push for a particular site should be tested — preferably under a range of performance conditions that are representative of how it will be used — before Server Push is deployed. Since mobile devices are limited in their compute power, compute-heavy pages where networking is not on the critical path are unlikely to do well.

The network characteristics also make a major difference. High latencies, and high losses — though not too high — lead to Server Push being more beneficial. In the context of a mobile app, it might be possible to profile what network performance the average user sees, and decide whether to use Server Push accordingly. In general, though, Server Push is more beneficial over wireless than modern, high-speed wired networks, hence our focus on mobile devices in this study.

Overall, whether or not websites benefit from Server Push depends on many factors, making it hard to predict whether or not Server Push should be used. The best way to determine whether to use Server Push is to measure how well Server Push performs on representative devices and networks.

6.7 Conclusion

Overall, we have found that Server Push can offer substantial performance benefits. Server Push works best when latency or loss rates are high (bandwidth has a smaller impact) and on sites where objects are requested late in the loading process. Mobile networks are particularly suitable for Server Push, and Server Push is likely to become substantially

more useful as mobile devices become more powerful. However, the way modern websites are constructed, with content divided across many servers, is a substantial problem. Furthermore, performance benefits vary greatly by website, and in some cases Server Push can be detrimental to performance, so it should be deployed cautiously, leveraging performance measurements to be sure that users will benefit.

CHAPTER VII

CellShift: A System to Efficiently Time-shift Data on the Cellular Network

7.1 Introduction

As mobile data usage continues to grow rapidly, and is predicted to increase tenfold by 2019 [23], novel approaches are needed to ensure users continue to experience strong cellular network performance. One potential approach is to flatten network loads, moving traffic at peak times to off-peak times. This reduces the cost of network usage and allows apps to increase data consumption without increasing network congestion. The key factor in determining network capacity is the *peak load* experienced by eNodeBs (base stations): network loads vary greatly over the course of the day, leading to network resources going unused at some times, while being heavily used at others.

Prior work has shown that users are willing to time-shift usage over several hours given the right incentives [47, 22, 112]. In fact, congestion-aware and time-dependent pricing has already been implemented to a degree in several countries [113]. Increasingly, mobile applications support time-shifting through mechanisms such as subscribing to feeds or channels, or through manual preloading [89, 40, 134], and recent work has shown apps can almost automatically be made delay-tolerant [78]. However, whether real-world cellular networks can benefit from such a system is a question that has yet to be addressed: there

needs to be sufficient, predictable variation in network load that can be leveraged in highly complex, dynamic cellular networks. Furthermore, a system that supports large amounts of highly delay-tolerant traffic could likely enable new types of applications.

In this chapter [102], by examining a wide variety of real cellular network loads from a large ISP around a major metropolitan area, we find that there is substantial variation in network load both from hour to hour and between eNodeBs, although we find that these variations happen over large enough time scales to require time-shifting over several hours. Predicting these trends hours in advance is essential for a long-term time-shifting system, so that the system can account for time-sensitive loads and unmodified apps. Using standard techniques to model diurnal load patterns, we are able to predict per-eNodeB trends with an average error of 0.02 of an eNodeB's capacity. As we examine trends in a wide range of locations, including both dense and underpopulated areas, and for a variety of users, we expect similar results to apply to other networks than the one we studied.

Furthermore, we demonstrate that we can efficiently and accurately predict the load that mobile users experience throughout the day as they move around, without actually requiring that we predict the user's location directly (with an average accuracy of 0.08 of an eNodeB's capacity). Even if a user's location is not predictable, they tend to visit locations with similar load patterns (for instance, a user might connect to one of several eNodeBs near their workplace). By leveraging this trend, we are able to predict the loads a user will experience over time scales of hours. While forecasting network loads would likely have implications on application design and network management, we focus on applying these forecasts to long-term time-shifting.

We introduce *CellShift*, a framework that leverages these predictions to schedule traffic designated as time-shiftable to meet deadlines set by the user or app. We make the problem of scheduling requests for millions of highly mobile users tractable by making short-term decisions at the 15 minute granularity at individual eNodeBs, with synchronization from user devices only happening every 15 minutes. These decisions nevertheless effectively

smooth network loads overtime scales of hours. Scheduling is done in a highly parallel manner that is scalable to a nation-wide network, and in a flexible enough manner to adapt to unexpected changes in network load, making this large-scale scheduling over long time periods practical.

We evaluate our approach at the scale of a large metropolitan area using anonymized traces from a real network. We demonstrate CellShift remains effective under a large range of network loads and scheduling constraints, showing that networks can support dramatic increases in load using CellShift. We also evaluate the data and battery overhead of a small-scale prototype implementation to demonstrate CellShift's energy overhead on the device is less than 2% in the worst case, due to the lightweight, periodic synchronization approach used.

Considering our original thesis statement, the dynamic, complex part of the cellular network in which we focus are the city-wide network load trends, in particular how they change due to user mobility and diurnal load patterns. In our system, ongoing measurements of network load, along with our prediction algorithm, can allow a system to be built that helps schedule delay-tolerant data, thus reducing congestion due to such traffic. In that way, this chapter supports the thesis statement that *because mobile devices experience uniquely dynamic and complex network conditions and resource tradeoffs, incorporating ongoing, continuous measurements of network performance, resource usage and user and app behavior into mobile systems is essential in addressing the pervasive performance problems in these systems.*

The contributions of this chapter are as follows:

- We conduct the first city-scale examination of per-eNodeB network usage trends over several months, using real data from a large ISP, and demonstrate that real-world, heavily-used networks experience substantial variations in load which are not effectively leveraged by standard app designs.

- We show that on a per-eNodeB basis, network load can be predicted with high ac-

curacy, in particular with an accuracy of 0.02 of total eNodeB capacity 15 minutes in advance. The network load a user will experience can be predicted up to a day in advance with an accuracy of 0.08 of total eNodeB capacity in a location-agnostic manner.

- Leveraging these findings, we present a novel design of a highly scalable system that schedules very large amounts of highly delay-tolerant data over time scales of hours on resource-constrained mobile devices. Requiring only application-level modifications, we reduce the impact of a 40% increase in network load on peak utilization by 58%, and achieve similar results for a variety of network loads, including reducing bursty loads by 76%. We incur battery overheads of less than 2% on the device to do so in the worst case.

7.2 Background and Motivation

There are a number of problems which must be addressed to make long-term time-shifting effective. Although prior work has shown there is user and carrier interest in long-term time-shifting, and that many classes of apps can be made delay-tolerant, several problems remain. In particular, it must be shown that network loads can be forecasted and time-sensitive loads can be efficiently scheduled on highly complex and dynamic cellular networks.

CellShift's goals: Ultimately, our goal is to lower the network peak load, as that's what determines the cost to the carrier. There needs to be the infrastructure to support the maximum load that will be experienced at a location, but those resources don't go away during off-peak hours.

We focus on determining whether real-world cellular networks can effectively support and benefit from time-shifting data on the scale of hours. Ultimately, our goal is to address the problem of growing network demands on cellular networks, reducing peak load while

increasing how much data each eNodeB can support. To do so, we demonstrate that there are opportunities on real cellular networks to leverage unused capacity due to variations in network load, and that these variations can be predicted and leveraged through an intelligent scheduling system. While such forecasts could be a generally useful tool both for managing cellular networks and developing more intelligent apps, we leave exploring such directions to future work.

In Figure 7.1, we explain the terms and metrics we use throughout the chapter, as well as show a simplified visualization of what CellShift is meant to accomplish. First, there is time-sensitive data which is not known in advance or controlled by CellShift (shown in dark blue), although it can be forecasted (the white line — note that forecasts are adjusted over time based on the current load). There is also data that can be time-shifted and is thus controlled by CellShift (light blue). The time-sensitive load varies throughout the day, as well as between eNodeBs, giving opportunities to time-shift delay-tolerant data to less heavily loaded times. However, due to uncertainties in our forecast, we may mis-schedule a bit of data during peak hours. To evaluate CellShift, we measure the reduction in the peak load caused by delay-tolerant data ($(A - C)/(A - D)$ as denoted in Figure 7.1), given an amount of additional delay-tolerant data to schedule (E/F). To evaluate our forecasting algorithm, we consider the absolute difference between the forecasted load and the actual load ($C - D$ averaged over each forecast interval).

Another proposed approach has been offloading to WiFi, but as we show in this chapter, there are ample opportunities to make use of available cellular resources through intelligent scheduling. Furthermore, while some major cities have widespread, free public WiFi, as was shown in one measurement study in Korea [73], others do not, and many users may not have access to free WiFi for much of the day (i.e. if their workplace does not provide free WiFi or if personal use is limited). As network loads increase, WiFi may likewise require innovative new approaches to address the rising demand.

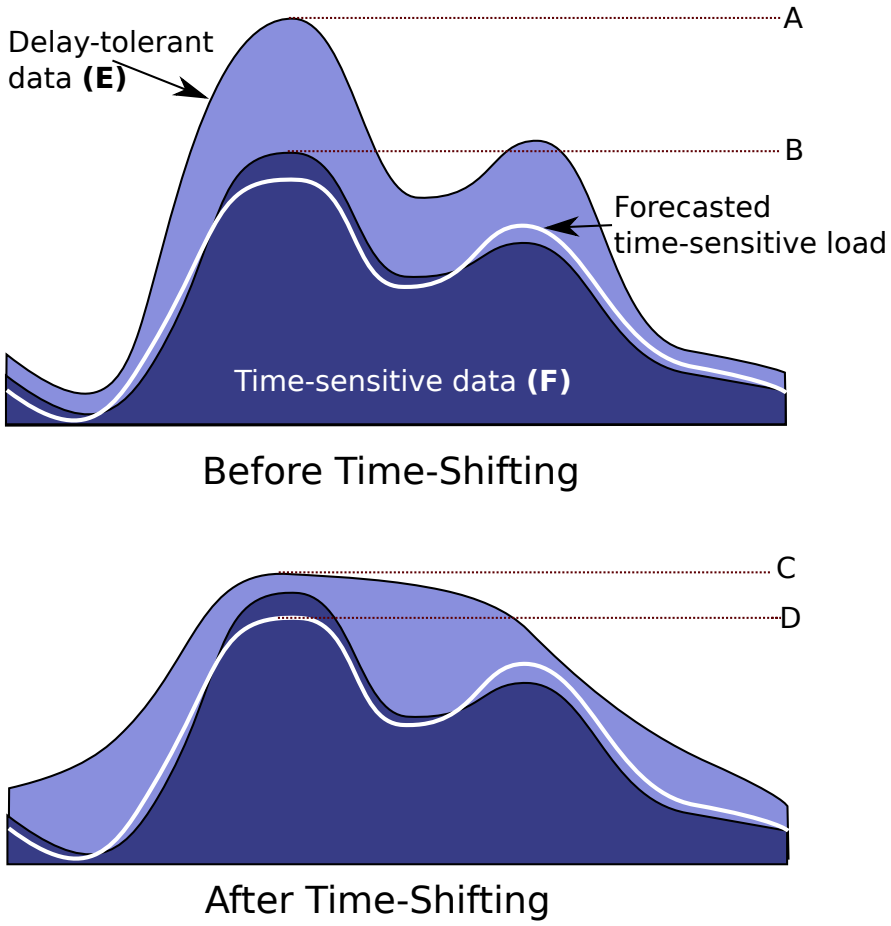


Figure 7.1: Overview of time-shifting. Delay-tolerant data is scheduled by CellShift around time-sensitive data. Our goal is for the peak load after time-shifting to be as close as possible to the peak time-sensitive load.

7.2.1 Incentives and Delay-Tolerant Data

With CellShift, the incentive to use the system for the user is that they get offered discounts on data downloaded using CellShift. Prior work has shown that users are willing to manually time-shift their data in exchange for discounts, so this is a reasonable model for how CellShift could be used [47, 113]. In fact, in some cases users actually increased how much data they used when there were incentives to use data during off-peak hours [113]. As a result, the incentives to use this system for the carrier not just that they can reduce peak loads, but also that they can support more data on their network, possibly resulting in

more business for them. In our model of how Cellshift is used, we also ensure that users always get the data they need: if we don't schedule data by the deadline, we just download at a suboptimal time. In this way, aside from the inconvenience of scheduling downloads in advance, there is no downside to the user; the user always gets their data.

Evidently, a lot of data is not going to be available for time-shifting over periods of hours. For instance, we expect that web browsing usually can't be time-shifted. However, it's possible that large videos, such as for TV shows, could be downloaded in advance. It is possible to do so on Netflix as of November 2016[1], suggesting a demand for such a service. Other examples of content that could be time-shifted on a long time scale include app updates, and large social media uploads where the user doesn't insist on making content available immediately. We separate the data into time-sensitive and delay-tolerant data. We try and forecast the time-sensitive data (by definition, we don't know about it in advance) and then schedule the delay-tolerant data around it.

Overall, we expect that time-shifting will enable new ways of using the network, and in particular the use of more video content, as it will become financially feasible to support large volumes of traffic. As such, we assume that the existing traffic in the network is fixed, an we examine what happens if we add a substantial volume of data to the network — can CellShift allow the network to support this new load?

7.2.2 Limitations

Evidently, we can't test our system on a real network, if we are building a system that works on the city scale. As a result, we analyzed our system based on traces, which introduced some limitations. First of all, our traces were at the granularity of 15 minutes. This meant our evaluation is somewhat coarse grained and in particular does not account for highly mobile users. Secondly, we don't actually know what the large loads our system enables would look like, both because we envision the system as enabling new ways of

[1] `http://www.nytimes.com/2016/11/30/business/media/now-netflix-users-can-watch-movies-offline-on-their-mobile-devices.html?_r=0`

using the network, and also because we didn't have any data on the nature of the current network loads available to us. As a result, we instead evaluate the impact of time-shifting a variety of artificial loads.

Overall, this is not a comprehensive real-world evaluation of whether a system like CellShift would work. This is a first step in the direction of determining if long-term time-shifting is feasible, and of evaluating many of the major limitations in such a system, in order to inform whether a real-world, city-scale implementation should be pursued. In this chapter, we show that this approach is promising, but that it requires long time periods to prefetch data, and thus would only work for limited types of loads.

7.3 System Design

CellShift's architecture, summarized in Figure 7.2, consists of both on-device components and network components which collaboratively monitor and predict network loads and schedule data accordingly. The in-network components would most likely be managed by the ISP, and with the exception of the in-network server (INS) responsible for scheduling, piggy-backs on existing cellular network functionality.

The on-device module gives apps access to per-user forecasts of time-sensitive load to incentivize time-shifting, and acts as a proxy for delay-tolerant data. The app developer uses this data and their knowledge of application semantics to determine what deadlines to select, including whether to solicit user input. Next, apps submit requests to be time-shifted to the API, which carries out the requests on the app's behalf. These requests consisting of the network request to make, the estimated size, a deadline and a unique id. We implement this proxy as an Android app in our prototype, but it could be implemented as an app library or system service.

The INS helps the on-device proxy determine when to carry out requests, by accounting for requests from other devices and predicted time-sensitive loads at the per-eNodeB granularity. Requests are scheduled only in the short term: the on-device proxy syncs with

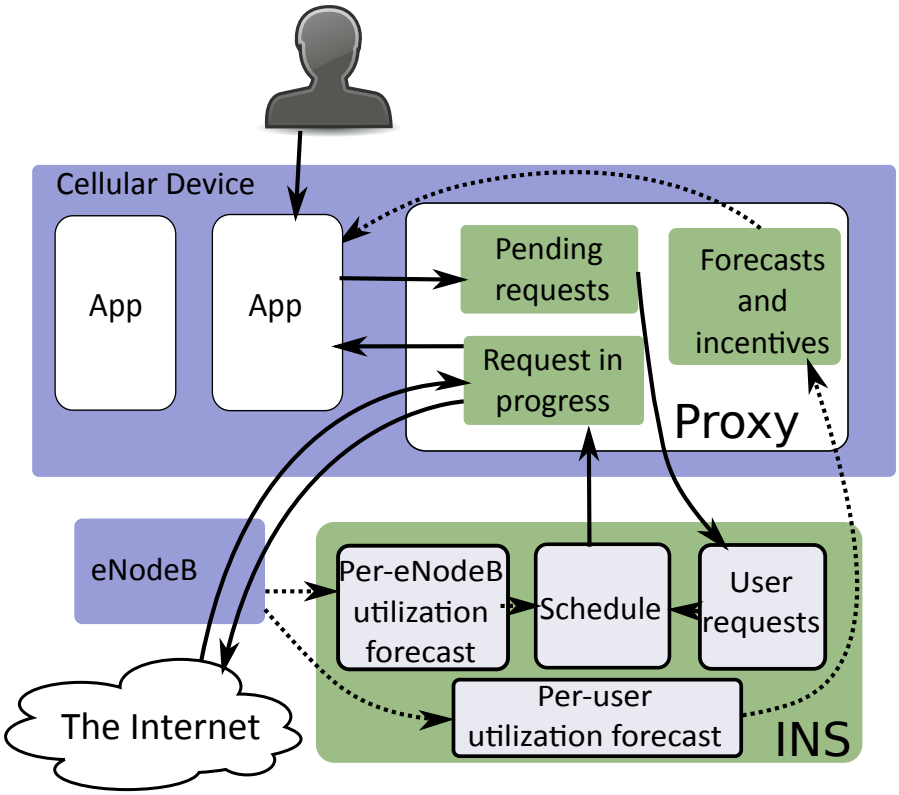


Figure 7.2: Architecture overview: Apps submit requests to an API on the phone, which schedules requests with the help of an in-network server (INS). The server may also provide per-user forecasts to help users or apps determine whether to time-shift data.

the INS a few times an hour and is only informed of what requests to fulfill before it next contacts the INS. We describe our scheduling approach in depth in §7.3.2. Note that applications only register time-shiftable requests with CellShift and so the remainder of the network load due to requests not submitted to CellShift must be inferred through other means (as described in §7.3.1). As we make decisions at the eNodeB granularity, our system is highly scalable: we can easily parallelize all decision-making by eNodeB, although the INS does not have to be physically situated at the eNodeB. Communication between the INS and proxy is all done at the HTTP level, with no modifications to low-level cellular network protocols.

To forecast network loads, additional data needs to be collected from the cellular net-

work, which today's networks already support. User associations with eNodeBs are needed for per-eNodeB scheduling, and historical net loads at eNodeBs are needed to predict future loads. Note that we do not store user location data for more than a few minutes, and cellular networks already need to know the eNodeB to which a device is connected to deliver data. As such, we do not introduce any new privacy issues. Furthermore, as existing cellular networks already make use of a variety of middleboxes for network management purposes, integrating additional functionality into these cellular networks is a practical solution. One advantage of implementing CellShift for cellular networks is that their management is centralized.

Next, we describe network load forecasting and scheduling in depth.

7.3.1 Forecasting algorithm and evaluation

There are two types of network trends that we forecast. Per-eNodeB load forecasts are needed to schedule requests around time-sensitive network loads, and per-user load forecasts (i.e the load at each successive eNodeB a device associates with throughout the day) allow apps to make informed decisions, including determining when to time-shift and how to set time-shifting deadlines. These forecasts can potentially allow apps to make network usage decisions beyond what CellShift supports, and cellular operators may find per-eNodeB forecasts to be useful for other network management operations.

To understand how to forecast network trends, we use a dataset that covers a major city over the first few months of 2014. Network load is measured by the fraction of available Physical Resource Blocks (PRBs) that are in use at a given time by each eNodeB. PRBs can be mapped to the amount of available bandwidth per user using the signal quality at the device. In our dataset, average PRB utilization over 15 minute time intervals have been collected for each eNodeB.

We forecast network load using the Holt-Winters algorithm [91], an exponential smoothing algorithm that accounts for periodic trends. We set the period to one day, and

eliminate the linear trend term (as we found this leads to overfitting). We use different sets of parameters for different forecast windows: short-term forecasts benefit from weighting short-term trends from the current day, whereas long-term predictions are based primarily on what has been seen in previous days.

Forecasting per-eNodeB loads is straightforward, but there are two challenges in extending forecasting to these per-user predictions. First, we must account for changes due to location as well as time which can introduce additional sources of inaccuracy. Second, we want to avoid storing user location beyond what is already part of the normal operation of cellular networks, due to the potential privacy implications of this data.

To deal with both problems, we first merge user location traces and eNodeB load traces to create location-agnostic per-user load traces, and use this new trace to predict the load a user will experience. We observed that users often connect to eNodeBs with similar loads even if they connect to different eNodeBs from day to day, likely due to these loads reflecting similar daily routines. For instance, a user who goes downtown every day may connect to different eNodeBs from day to day, but likely experiences a load pattern typical of that downtown area. Creating location-agnostic per-user load traces allows us to leverage these similarities. It also allows for a more robust and consistent approach to forecasting loads: if the user deviates from their normal routine, rather than having to predict user motion accurately we just allow the Holt-Winters algorithm to adjust the user's periodic load pattern based on daily load fluctuations, as normal.

In Figure 7.3, we show how accuracy varies with the forecast time range for per-user forecasts. Our accuracy is within about 0.08 total network load in most cases. Note that predictions are more accurate two hours or less in advance, as the short-term component of Holt-Winters can then account for shorter-term fluctuations that occur on the order of hours. In our per-eNodeB forecasts which are used for scheduling data (rather than motivating users or apps to time-shift), we make use of predictions 15 minutes in advance, a number chosen due to the granularity of our dataset. The median error of these per-

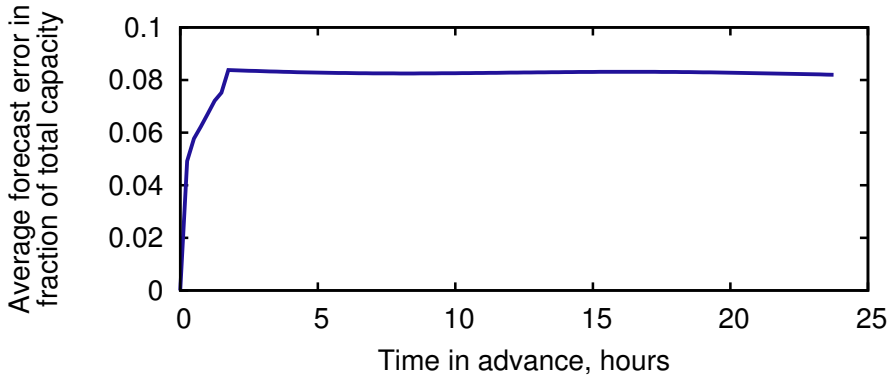


Figure 7.3: Impact of time interval on forecast accuracy when predicting on a per-user basis, in fraction of the total PRB utilization.

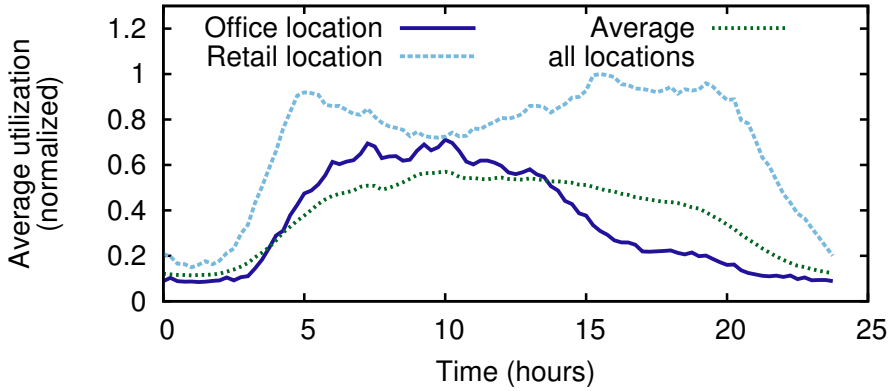


Figure 7.4: Example diurnal trends: note there is substantial variation between eNodeBs, either in the shape of the curve or when it peaks.

eNodeB predictions is 0.02, although 10% of the time the error is over 0.1. We will show that although these forecasts are accurate enough to significantly decrease the impact of delay-tolerant data on peak loads, these occasional errors do introduce some limitations.

We examined some other approaches to improve forecasts, including incorporating weekly trends, trying to forecast error, and weighting overestimates and underestimates differently, with no significant effect, and so we focused on a simpler but equally effective approach.

We next examine some example eNodeBs to better understand how network loads vary by location. Figure 7.4 shows how the average load patterns experienced by two eNodeBs

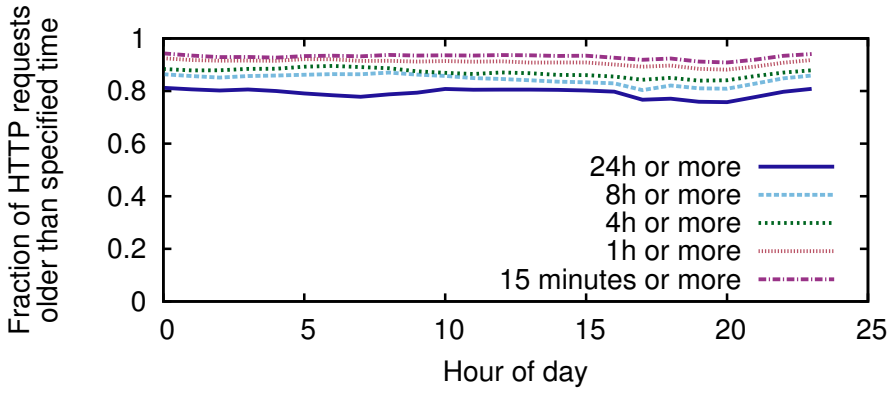


Figure 7.5: Fraction of HTTP requests older than various possible target deadlines, based on last-modified dates on content headers, based on a one-day sample of hundreds of millions of HTTP requests.

vary over the course of a day, and compare them to the average load across all eNodeBs. The eNodeB covering an office area near a transit center peaks earlier than average and is less heavily loaded in the late afternoon. An eNodeB in a retail area has two distinct peaks and a lull in the middle of the day. These load variations demonstrate examples of predictable usage patterns around which data can be scheduled. Note that the sorts of variations we can leverage are, in practice, on the order of hours. For instance, an increase in load due to commuters near a train station lasts several hours as commute times vary. This means that our deadlines must be on a similar scale, as we show in our experiments.

Given that the load patterns we detect vary on the order of hours, does that mean traffic we fetch in advance will be stale when it is used? To show this would not generally be the case, we evaluate a sample of all uncached anonymized HTTP data covering three million requests over one day (not the same dataset used elsewhere in the study), which includes short-lived content we would treat as unshiftable anyway. Even so, as we show in Figure 7.5, the majority of content is still valid hours later. Furthermore, we are targeting types of content that are relatively long-lived, such as static videos, music and podcasts. In conjunction with prior work suggesting users are willing to delay data by hours, this supports the notion that time-shifting data over time scales of hours is a practical and effective

solution for significant classes of traffic.

However, it is likely for a real implementation that there would be limits on how much data could be time-shifted. For instance, users wanting to watch a video later in the day would probably not be interested in viewing it when asleep, and may not know what video they want to watch until a few hours in advance.

7.3.2 Scheduling Algorithm

Next, we determine how the INS can best leverage these predictions to effectively schedule data. Due to the enormous scale of the networks in which we schedule data, and the fact that the devices on which we schedule data are highly mobile, we must determine a way to make this scheduling problem tractable. To do so, we show that effective scheduling decisions can be made on a per-eNodeB basis using only short-term decision making, while still achieving good long-term, global results.

The first step is to determine the amount of available capacity. The INS keeps track of a *target threshold* for each eNodeB, which is equal to the peak utilization that eNodeB has seen so far, multiplied by 0.8 (a number chosen through experimentation) to account for the slight inaccuracies in forecasting network load. The available capacity is the difference between this threshold and the actual load at this point in time. Since we cannot know the actual load in advance, we use our forecasts to estimate it.

Time is divided into scheduling intervals, equal to the frequency with which devices sync with the server, and requests are only scheduled one scheduling interval in advance. When scheduling, any requests that are about to reach their deadline are first fulfilled regardless of network capacity. If any free capacity remains, more data is then scheduled. Requests are scheduled to fill the estimated available capacity below the target threshold completely, if possible (i.e. we use a work-conserving approach). They are fulfilled in earliest-deadline-first order. Since our goal is to avoid increasing the eNodeB's peak load, downloading as much data as possible now without exceeding the target threshold is al-

ways a better decision than delaying downloads until later, even if future loads are lower than expected or the user moves to a lower-load location.

If the target threshold was exceeded by requests reaching their deadline (rather than a misprediction), the target threshold is increased to this new peak load, as data must have been scheduled too conservatively in the past for this to happen. This allows us to effectively schedule large data loads where completely eliminating the impact of delay-tolerant data is not possible. The periodic scheduling approach allows decisions to be made at the per-eNodeB level while resulting in effective time-shifting globally. It allows us to leverage more accurate, up-to-date forecasts while allowing devices to only periodically connect to the INS, allowing for a good tradeoff between scheduling accuracy and device overhead.

7.3.3 Alternate design approaches

We examined a number of alternative design approaches against which we compare CellShift to better understand the impact of the tradeoffs made.

Fixed download time: In this approach, user scheduling decisions are made once, rather than recalculated each scheduling interval. This also allows us to predict how much extra capacity will exist over the course of the scheduling interval and schedule content accordingly, acting as an admission control system. For instance, we might select the quality of the music or videos to prefetch. Upon submitting a request to time-shift data, the system calculates the expected best time to fulfill the download (i.e. the time with the most available capacity), incorporating estimates of user location at each point in time to determine how much capacity to reserve at each eNodeB. This allows devices to wake up less frequently to communicate with the server, but with some loss of accuracy in predicting the available load in the future, thus requiring that we be more conservative in the amount of data we schedule. We also assume we can predict user location with this approach, which we do not evaluate in this paper.

No inter-device coordination: An alternate approach is to make scheduling decisions

Table 7.1: Prototype overhead metrics for a Samsung S4 device scheduling new requests every 15 minutes. Data sent includes all bytes sent over the link for corresponding flows, and is a negligible fraction of an eNodeB's capacity.

Metric	Result/active interval	Result/day (constant requests)
Energy consumed from all sources[2]	6.7 J	644 J (1.8% battery capacity)
... due to server communication	6.4 J	614 J (1.7% battery capacity)
... due to forecasts	4.4 J	422 J (1.2% battery capacity)
Extra signaling overhead		
forecasts	5.7 kB	547 KB
coordination	5.4 kB	516 KB

on the device in isolation. The device still fulfills any requests that are still pending when their deadline is reached. Otherwise, it attempts to spread downloads as evenly as possible between underutilized time slots, with no knowledge of load from other devices. This schedule is adjusted each time new data on eNodeB utilization trends is received from the network. We show this approach is not very effective.

7.4 Prototype Implementation and Performance

We evaluate CellShift's overhead by developing and benchmarking a prototype system, which we use to demonstrate that server-side coordination is scalable and able to support large numbers of users, as well as efficiently forecast and schedule requests on behalf of these users with minimal overhead. We also demonstrate that the power and data overhead on the user device is small enough for resource-constrained mobile devices. We evaluate CellShift's effectiveness at scheduling loads at scale through a simulation in §7.5.

We developed a Android app consisting of two parts: a simple user-facing app that downloads large files, and an implementation of CellShift's on-device component. The former periodically generated requests to download files of various sizes from a web server we set up for this purpose, with various deadlines, and submitted these requests to the

CellShift component. The CellShift module set a periodic timer to wake up and send the size and deadline of each pending request to the INS. The INS then replied with a list of requests to fulfill before the next synchronization. The app also fetched and displayed network load forecasts.

The INS was implemented in Python, and made use of a trace of eNodeB loads from a real eNodeB to perform the scheduling. Each scheduling interval, the INS updated the forecast models for the eNodeB and for each user, then estimated the load over the next time interval to determine how much could be scheduled. It stored any requests received and determined which requests should be fulfilled next. When the device next connected to the INS, the device fetched a list of requests specific to that device which the INS determined should be carried out over the next scheduling interval. To give the most conservative performance values, we had the device constant generate and submit requests each scheduling interval

We first examine the power and data overhead. To examine the impact of CellShift's network communication in isolation (i.e. excluding the impact of any unrelated background traffic such as Android system generated traffic), we generated parameters for a standard model of LTE power consumption [56] using a power monitor, and then calculated the impact of the relevant TCP flows sent using tcpdump traces collected on the test phone. This model accounts for the impact of cellular RRC state transition dynamics, which have a significant impact when sending small amounts of data, and by analyzing each component in isolation we can calculate a worst-case value for energy consumption when there is no other traffic we can piggyback on. It also allows us to separately examine each component of network communication. Although the exact results would vary by cellular network, as configuration parameters such as RRC tail timers would affect the energy efficiency, we chose a network with a particularly long tail timer to measure an upper bound on the possible overhead.

The results are summarized in Table 7.1. These values correspond to a power overhead

of about 1.8% per day on a Samsung Galaxy S4 [107]. In practice, it would almost always be lower as it's unlikely for a user to be submitting requests 24 hours a day and for there to never be overlapping traffic. Power consumption could be further reduced if fast dormancy were enabled. Nevertheless, this overhead motivates the need for CellShift to support intermittent, rather than continuous, server synchronization.

Each coordination with the INS consumes 5.4 kilobytes of data on average. Headers make up the majority of the data consumed, but a more efficient protocol could reduce the data consumed further. This has a negligible impact on cell tower load, on the order of a thousandth of a percent for 1000 users all with poor signal strength.

We also measured the INS scheduling overhead. An artificial workload of data for 10,000 users, far more than eNodeBs can support today, takes less than a millisecond to schedule with an unoptimized python script. We designed our scheduling algorithm to be highly efficient and parallelizable. To stress-test the INS, a separate server then generated requests for 1500 simulated users. We increased the communication interval to five minutes for ease of testing, and had all simulated users submit requests almost simultaneously to ensure a high server load. The additional overhead from adding new users never exceeded tens of milliseconds (also with an unoptimized python-based server). The overhead of storing and scheduling requests was under half a megabyte. Per-user forecasts required 24 megabytes for 1500 users.

7.5 Simulation Evaluation

We evaluate CellShift through a city-scale simulation to ensure we are able to adapt to real user mobility and variations in network load over time and by location. The simulation encompasses over one hundred eNodeBs and hundreds of thousands of users, covering both the downtown core and some suburban areas of a major US metropolitan area. A real-world deployment could be even larger: each component of the analysis is parallelized by user or by eNodeB, allowing it to scale indefinitely.

We made use of a PRB utilization dataset similar to the one used to develop our forecast algorithm, collected in the first week of June 2014 (plus the last day of May to initialize the forecast models). This dataset is independent of the several month one used to develop the forecast algorithm, and its length was selected to ensure we evaluate both weekdays and weekends. We also used a set of anonymized user to eNodeB associations collected at the same times and locations. The privacy of the users was preserved as all user identifiers were anonymized prior to our analysis and we focus on the aggregate statistics across all the users in the data sets. PRB utilization can be mapped to bandwidth using RSRQ (Reference Signal Received Quality) [3] values, which we also collected. In a few cases, RSRQ values were missing in the dataset, and so we took the lowest value (supporting the lowest bit rate) to make the most conservative possible claim about how much data can be time-shifted.

The dataset of eNodeB to user associations contains an entry every time the user interacts with the network, such as when sending or receiving network traffic or connecting to a new eNodeB. Where we have a small gap in the data, we assume that the user is stationary during that time and connected to the same eNodeB. To bound the scope of our simulations (a limitation of our simulations rather than of CellShift), we do not schedule downloads for users who will be outside of the target area for an hour or more, and did not schedule any additional requests for users who spent less than five hours in our target area. If the user is only outside the area briefly, we delay all CellShift downloads until they return. We only consider users connected to the cellular network, and treat users on WiFi as being outside the target area, due to a lack of information on those users.

The main question we examine is whether CellShift can allow networks to support a substantial increase in load without unduly increasing peak traffic levels. We see CellShift as enabling new network services, as well as heavier but cost-effective network usage, such as making it more affordable for users to watch large videos on their devices. As such, for most of our test cases we examine the impact of data added to the network, although we also look at the impact of scheduling existing data.

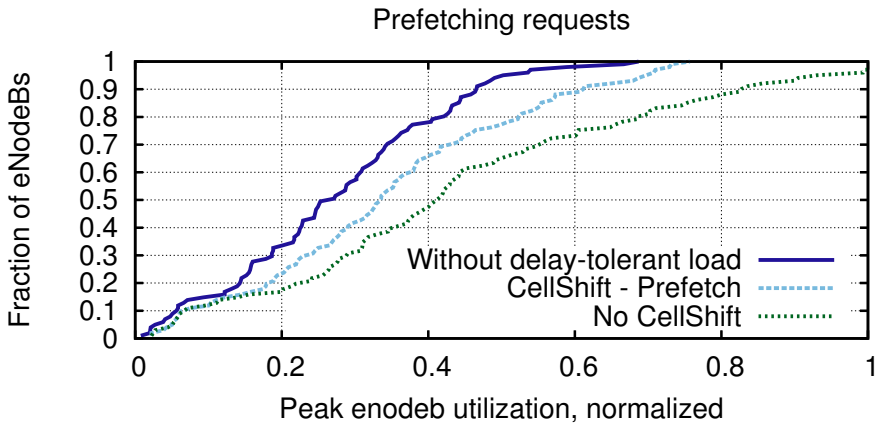


Figure 7.6: CDF of top loads due to prefetching traffic with fixed deadlines, comparing against no time-shifting. We also show the peak load if we were to remove the delay-tolerant load from the network entirely.

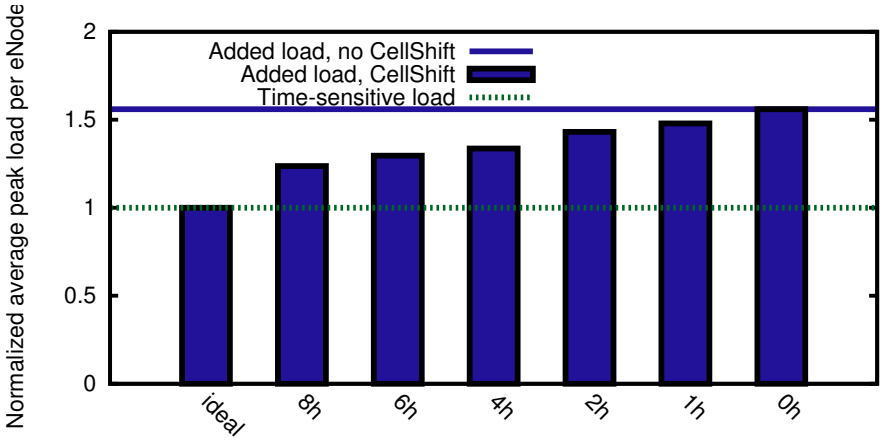


Figure 7.7: Impact of deadline length on prefetching. Prefetching is more effective when scheduling with less constrained deadlines, particularly deadlines of 4h or longer.

7.5.1 Impact of Scheduled Request Patterns

We first examine one request load and scheduling configuration, and then compare against alternate network loads and deadlines. For this test case (and other tests, unless otherwise stated), we use an 8 hour deadline. Data is scheduled at 15 minute intervals. Each individual user submits requests to an eNodeB-specific server. Each eNodeB then estimates the load in the next time interval and schedules requests accordingly. We then

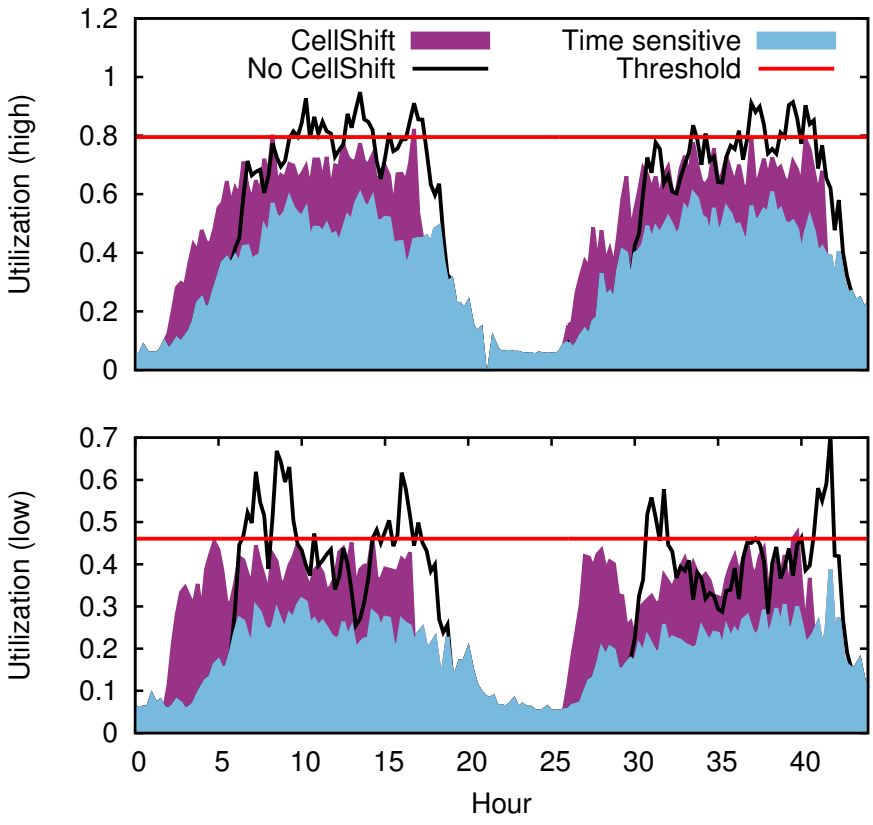


Figure 7.8: Example eNodeB time-shifting traces for two eNodeBs with different loads with 4 hour deadlines, along with the peak load seen on previous days ("Threshold"). Examples chosen to illustrate cases of how time-shifting works and represent roughly average cases.

calculate what the actual load resulting from that schedule would have been, given the real network load during that time interval.

For this test case, a randomly selected subset of users want to make five large network requests at randomly selected times during daytime hours, resulting in a 40% total increase in PRB load across all eNodeBs. We show the results of prefetching these requests in Figure 7.6. The distribution of peak loads at each eNodeB are shown in the form of a CDF, both with and without CellShift's time-shifting. We also show the peak loads due to time-sensitive requests alone, excluding delay-tolerant requests entirely (D in Figure 7.1). When prefetching, CellShift is able to reduce the average impact of added load on per-eNodeB maximum utilization by 58% ($(A - C)/(A - D)$ from Figure 7.1), for an overall data

increase of 40% (E/F).

Next, we examine the impact of our deadline length in Figure 7.7, with deadlines of a fixed time in each case. Unsurprisingly, we do better with a less constrained deadline. With a four hour deadline we reduce the impact on peak loads by 40%. Conversely, a one hour deadline only reduces peak loads by 14% on average. Recall that we observed when finding predictable variations in network loads that they tend to be on the order of hours, and as such, the cutoff for CellShift's effectiveness seems to be about 4 hours.

To better understand how data is time-shifted, we show example traces from two eNodeBs in Figure 7.8 for the four hour deadline case, before and after scheduling data with CellShift. The top trace is in a moderately busy area, whereas the second one has a lighter load. In both cases, there is less load this day than on other days, and so the target threshold is higher than the peak load on those days.

We are able to flatten these loads, fitting the scheduled loads to the shapes of each eNodeB's load pattern, and in most cases do not exceed the target threshold. Note we are able to shift some traffic to off-peak hours, as well as shift data between eNodeBs. During daytime hours, the network loads are almost flat, with slight variations due to limitations in the forecast accuracy. Note how there are dips in the time-sensitive load during the middle of the day for the first day of the top trace, which we flatten when adding the time-sensitive load, although later in the day due to a slight misprediction we slightly exceed the target threshold. For the second plot, which has a smaller number of users, we are able to flatten loads more dramatically as the impact of delay-tolerant data is much greater compared to the time-sensitive load.

We also examine a variety of alternate network loads to schedule. In Figure 7.9, we examine the impact of varying both the quantity and distribution over time of requests scheduled by CellShift. We start by examining a scenario where traffic that would normally be used during peak hours is delayed to off-peak hours ("delay"). When shifting data in the other direction, we are more likely start by seeing the smallest loads we are likely to see

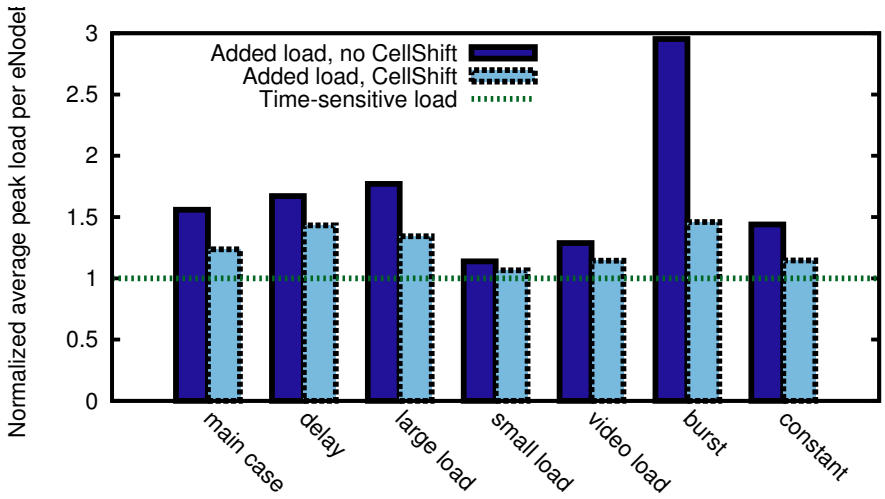


Figure 7.9: Time-shifting remains effective under a variety of loads, including both different sizes and distributions over time.

and so our flexible, short-term decisions work well. However, when delaying data away from peak hours, we have a lot of pending data to schedule during peak hours and thus are more likely to misschedule it, even if there are more opportunities to download the data later.

Next, we examine a larger request load where the added delay-tolerant data is 56% of the time-sensitive load ("large load"). We are able to reduce the impact of these requests on maximum utilization by 56% in this case, almost as much as with the smaller load. Examining a lighter load than our first example ("small load"), of a 10% increase in total data, we are able to reduce the impact by 52% on average.

For another use case where data is used during off-peak hours, we examine a load derived from real user video traces ("video load"). As described in work by Erman et al. [32], video traffic both contributes a large amount of data (30% of total HTTP traffic in 2011), and makes up a higher proportion of traffic during off-peak hours. We examine a set of traces of moderately-sized videos (less than 1 GB) collected separately from the eNodeB load data, and generate a load based on those traces. One limitation is that, as we are subtracting data from our traces rather than adding data, and the video traces were collected

separately from the network load traces, users who experience poor signal strength during off-peak hours but who have video traces assigned to them may find themselves driving PRB loads into the negatives in the no-timeshift case. Therefore, we do not adjust available bandwidth based on RSRQ values for this experiment. We find we can reduce this load by 50%.

With support for time-shifting, it is possible that new applications leveraging this technology would lead to traffic demands that diverge from the current pattern. To demonstrate we can still achieve good results with other load patterns, we examine a case where users download large amounts of data during their commute only (labelled "burst"). In this case, we actually reduce the average peak eNodeB load more, by 76%. Unsurprisingly, spiky loads benefit more from being flattened.

However, even very smooth loads can be time-shifted ("constant"). We randomly select users and give them a constant additional load during peak hours. We are able to reduce this load by 67%. Although this load is already smooth, the underlying time-sensitive data is not, and there is thus opportunity to smooth the total load. Furthermore, the small but constant series of requests submitted means that we are less likely to suddenly have a large amount of data to schedule when there are insufficient opportunities for doing so.

7.5.2 Impact of Forecasting and System Design

We next evaluate the role of various aspects of CellShift's design, shown in Figure 7.10, starting with the forecast accuracy. Per-eNodeB forecasts are needed to estimate the average utilization in the next time slot, but these forecasts, even 15 minutes in advance, are not perfect. We show that with **perfect foreasting** 15 minutes in advance we could almost completely eliminate the impact of CellShift-controlled requests. Basically, in that case we are able to leverage not just the larger, predictable variations that persist from day to day, but we are also able to capture sudden, sub-hour changes in network load that are challenging to forecast. We leave closing this gap to future work. One approach could be actively

monitoring the second-to-second eNodeB load immediately before a scheduled request to double-check it's a good time to fulfill the request. Modeling net flows of users to detect potential anomalies may also be a promising approach.

We also examine a number of alternate system designs with different tradeoffs, which we described in §7.3.2. The first is one where the INS calculates a single, optimal download time when a request is first issued, between four and eight hours in the future (**fixed deadline**). No further coordination is needed, allowing the device to sleep entirely until the scheduled time. To schedule these requests, the INS calculates the optimal time to schedule requests based both on predictions of network load at each eNodeB the user is expected to visit, as well as prior requests made to each eNodeB at each time. Each eNodeB keeps track of requests made to date. Once requests are scheduled at the eNodeB, that request is treated as fixed. As we are scheduling requests in advance and monitoring the global load on each eNodeB, we also determine a load the network can safely support and only allow those requests to be admitted.

This approach assumes user location can be predicted, but as we are examining this use case primarily for comparison purposes, we did not develop a long-term user location prediction system. User location has been shown to be predictable to some degree in prior work [131], especially at home and at work [62]. We are able to decrease the overhead of the added load by 44% with an eight hour deadline. In addition to achieving somewhat lower results, we are also limited to supporting half as much data.

We also examine a design with **no inter-device coordination**. In this scenario, devices spread load evenly among available time slots before the deadline (we also tried randomly selecting time slots, which performed worse). However, we would often simply move utilization peaks around, as we had no way to avoid other scheduled requests, and we and only reduced the impact of the data on peak loads by 18% on average. Variations in network load are small enough during the day that coordination is needed. With a 24 hour deadline this approach works better, but we are then essentially shifting most traffic to occur at night,

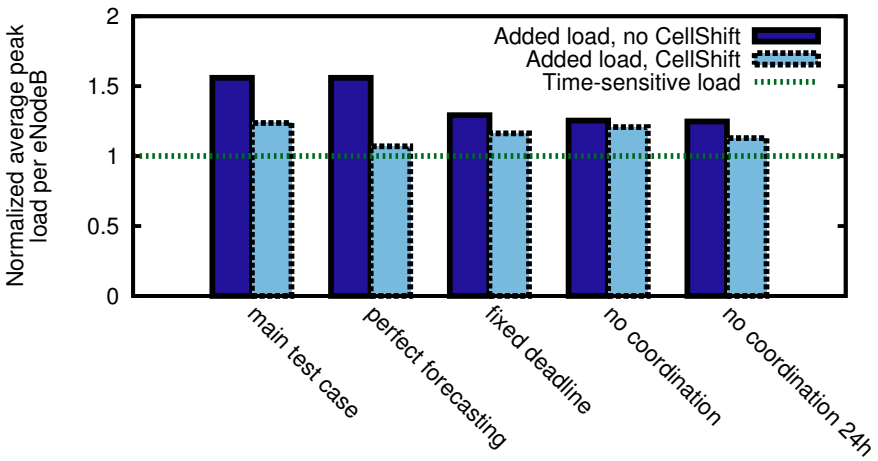


Figure 7.10: Impact of forecast accuracy and effectiveness of alternate design approaches. While perfect forecasting achieves better results, less flexible scheduling approaches can support less data.

which may be impractical for many use cases.

7.5.3 Alternate Cellular Network Characteristics

So far, our results have been based on network traces from a major cellular network, but networks may have different characteristics worldwide. We examine how challenging time-sensitive network loads may impact our results. We examine two hypothetical patterns in Figure 7.11: one where we greatly increase the time-sensitive network load, resulting in less spare capacity to work with, and another where we reduce the number of time-shifting opportunities, smoothing the baseline utilization by taking the average value over the last hour for each scheduling interval.

Although our results are dependent on leveraging variations over time and between eNodeBs, the artificially smoothed fixed data load surprisingly performs better than the real-world one, as shown in Figure 7.11 (in the form of CDFs, since we've altered the distribution of time-sensitive network loads as well). Although smoother load patterns offer fewer time-shifting opportunities, smoothing the data makes the peaks and valleys more

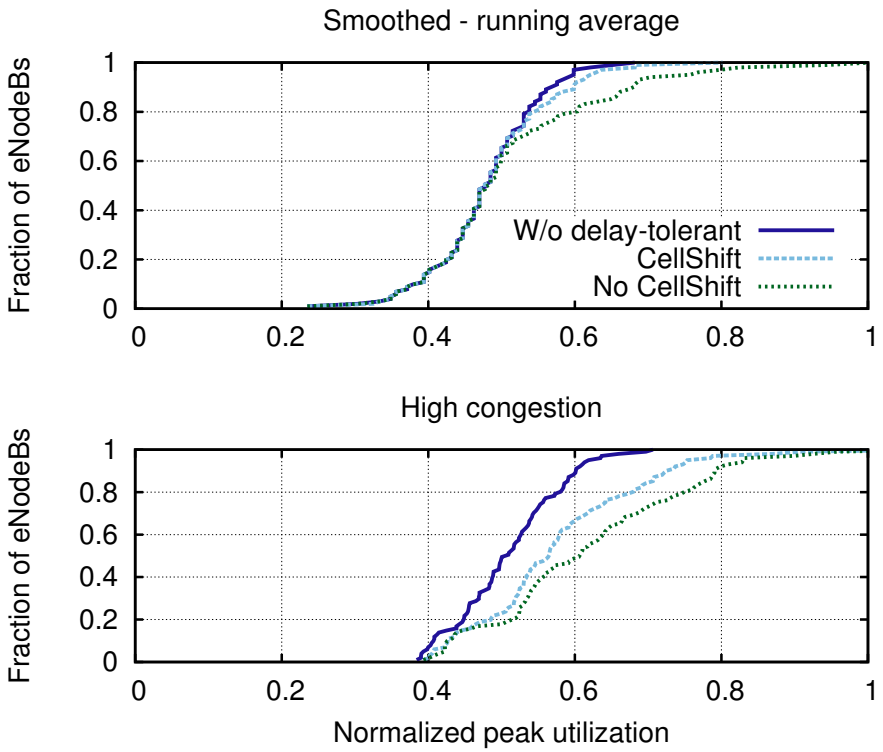


Figure 7.11: We examine some alternate loads of time-sensitive data. When non-CellShift-controlled, time-sensitive data is more even, time-shifting is easier, but when the network is already highly congested there is little room for CellShift to schedule data.

predictable, and we are thus less likely to inadvertently schedule too much data during a data spike. With time-shifting, we are able to reduce the impact of these added requests almost entirely. In fact, because this smoothing reduces any utilization spikes in the baseline data, even without time-shifting the CellShift-controlled requests have a lower impact on each eNodeB's maximum utilization.

With more congestion in the baseline load, time-shifting around these loads are more challenging and we only reduce the impact of CellShift-controlled traffic by 37% on average. When a network is already overloaded, there are fewer opportunities to time-shift the requests.

7.6 Discussion and future work

There are a few promising directions for future work. First, we have shown by simulating perfect prefetching that our biggest challenge in creating an effective time-shifting system is in the accuracy of our forecasts. Possible future directions include leveraging fine-grained user movement patterns directly, detecting anomalies in daily motion patterns, or even having the device get network utilization from the eNodeB every few seconds during a time slot where it is scheduled fulfill a request, although that would likely have a much higher energy overhead.

Furthermore, some types of loads can be more effectively scheduled than others, motivating which types of traffic should be targeted by such a system. Time-shifting most dramatically reduces loads on the network caused by large, bursty downloads, especially during peak hours, but there is less room to improve loads occurring more predominantly during off-peak hours. Furthermore, if all eNodeBs are congested, the benefits are minimal, since time-shifting allows us to use resources more efficiently rather than add network capacity. For instance, time-shifting content such as music or podcasts would be a particularly promising approach: users may want to listen to them during peak hours, they are generated in advance, they probably reflect predictable user preferences, and they contribute a substantial amount to network loads.

In developing CellShift, we have demonstrated that network loads vary significantly throughout the day, and that these variations are at least partially predictable (as well as identified limits on the predictability of this content). While we have presented one system which can leverage these predictions, we expect that these forecasts could be valuable in other ways. For instance, users could query such a database to determine what sorts of network loads to expect at a particular time and place, such as when travelling, or a gaming app could suggest entering a multiplayer or single-player mode depending on forecasted network conditions. Network operators may also find these forecasts facilitate network management, another interesting direction to explore in future work.

We do not consider the possibility of malicious devices trying to subvert the system, but as users must register a specific device with the system, identifying badly behaving users — such as users deliberately sending large amounts of data when network load is highest — would be straightforward. Finally, although we have analyzed our system on a large scale through trace-driven analysis, we have not deployed a prototype system on such a scale. Ideally, we would deploy and evaluate such a system with a large number of representative users in a major metropolitan area to fully evaluate its effectiveness.

7.7 Conclusion

In this chapter, we first showed that there are substantial variations in network load in heavily used networks that can be predicted. We then presented a system, CellShift, that can effectively time-shift content on cellular networks to leverage these variations, reducing the impact of CellShift-controlled network requests on eNodeB capacity by more than 50%. Furthermore, we can achieve these results in a network where the majority of data is time-sensitive and thus unknown to CellShift in advance, by modelling future network loads based on past trends. CellShift is a lightweight system that requires devices to submit requests to a per-eNodeB coordination server no more than a few times an hour, and schedules data by making only immediate, local decisions intermittently, with scheduling performed only at the eNodeB level. Nevertheless, CellShift achieves strong results in reducing network load overall. As the challenges in creating a practical time-shifting system have yet to be examined, namely predicting network load and scheduling content accordingly over long time scales in large and highly complex cellular networks, we also examine a number of alternate design approaches. CellShift compares favorably against them, showing that our approach offers a good tradeoff between overhead and effectiveness in making use of free network capacity to reduce peak loads.

CHAPTER VIII

Predicting App Network Traffic to Facilitate Prefetching

8.1 Introduction

Ensuring good performance on network-dependent mobile apps remains challenging. One possible solution is prefetching: not just long-term prefetching, but prefetching a on the order of a few hundred milliseconds in advance as well. A common problem in prefetching systems is determining *what* to prefetch. In addition, there are other cases where knowing what content will arrive in the future will facilitate decisions that can lead to improved performance, such as in Server Push.

In this chapter, we examine this problem in two parts. We first introduce two tools that allow application behavior to be predicted, and evaluate them using network traces. We then examine how these tools can be applied to real-world prefetching systems, and discuss the challenges that would need to be overcome for automated prefetching to be possible.

First, we examine two approaches to predicting app behavior: determining what activity a user will visit next, and determining what network requests will be issued soon (e.g. in the next activity). The former problem, as it turns out, is relatively straightforward to solve, as certain activity transitions are consistently far more common than others. This could allow static requests to be prefetched, that are the same each time the Activity is loaded. Determining more generally what requests will be issued soon is more challenging. We are

able to do so with a median accuracy of about 60% of requests, but there are challenges in applying this to a real system.

For the second problem of predicting app requests, we examine only a subset of apps: apps which rely on network traffic to fetch content, and leverage one or more HTML, JSON or XML files to determine what to fetch. News and social media apps are examples of such apps: games, video conferencing services, and chat services are not. We infer the structure of these apps — what series of requests lead to content being fetched, and how the app determines what parameters to pass along with the URL — and based on this model, predict what content will be fetched in the future. About 60% of objects are fetched in the median case in our trace-based simulation, and about 65% of bytes are fetched, although the success rate varies greatly by application. A major limitation, however, is that a substantial amount of extra data is fetched: often sometimes several times as much as what is needed. This limitation means that automated prefetching is likely not yet ready for real-world applications.

Next, we examine the challenges of applying these approaches to real systems. As a motivating use case, we consider a prefetching system built around cloudlets. Cloudlets have been proposed as a method to address many of the limitations of mobile devices, by offloading functionality to machines near access points [125, 110, 39, 12, 46, 109]. Acting as a proxy, they could prefetch content for user devices, and observe and predict mobile traffic. We discuss how our prediction system would apply to a cloudlet system and what would be needed to make such a system feasible. We find there are several obstacles to achieving the results in our trace-based evaluation. In particular, we find that the time to download content is a challenge, limiting how much content can be predicted and fetched in time for a real system. We discuss some potential approaches to address this problem, which would require future research directions.

Considering how this project supports this dissertation's thesis, it addresses not just to the complex and dynamic nature of network performance, which can be masked through

prefetching, but also leverages the unique nature of mobile app behavior. The measurements in question are not numerical measurements per se, so much as ongoing observations of the contents of network requests, and where and in response to what these requests occur. Using these measurements, we hope to be able to support systems, such as prefetching systems, that can improve performance on mobile devices. In this way, this section supports the thesis statement that *because mobile devices experience uniquely dynamic and complex network conditions and resource tradeoffs, incorporating ongoing, continuous measurements of network performance, resource usage and user and app behavior into mobile systems is essential in addressing the pervasive performance problems in these systems.*.

The main contributions of this chapter are as follows:

- An investigation of the predictability of activity transitions, determining that over half of all activity transitions are the most common transition, and the three most common activity transitions make up almost all activity transitions.
- An approach for predicting URLs requested based on prior network traffic, as well as the parameters passed with those URLs, with about 60% accuracy for a large class of apps in a trace-based evaluation.
- An examination of the feasibility of automated prefetching, the limitations of URL prediction for prefetching, and of what is needed to work towards an effective, truly automated cloudlet prefetching system that can achieve similar results to the trace-based evaluation.

8.2 Motivation and Use Cases

We address the problem of reducing network latency by better understanding network performance. We have discussed two methods for reducing latency that require knowing about traffic in advance: Server Push and prefetching. Generally, causing content to be

downloaded early can mask limitations in network performance, but this requires determining a systematic way to predict URL requests for apps, which has yet to be done.

Applicability to Server Push: While Server Push is not the use case we will focus on in this chapter, we briefly discuss it as motivation for predicting content. In particular, one recommended approach for implementing Server Push we discussed was by using a proxy. Third-party content and domain sharding are major limitations for Server Push, so why not have a proxy serve as a single point from which the client can request content from disparate sources? In this way, the user would get the full benefits of HTTP/2, including allowing Server Push to have all known content be piggybacked on the initial HTTP page. We did not discuss, however, how the proxy might know what content to push. In some cases, a sort of configuration file could perhaps be sent to the proxy, if the original server knows everything that will be pushed, but that might not be the case. What we would like would be for the server to be able to predict what content will be needed simply by observing traffic as it passes, and then build a model of what sort of content to expect when. This is precisely what we aim to do in this chapter.

Applicability to cloudlets: Cloudlets are servers near access points (either WiFi access points or cell towers) that phones can offload functionality to, allowing for increased performance [125, 110, 39, 12, 46, 109]. It has been proposed, in particular, that content can be prefetched to these cloudlets in a technique known as *data staging* [39]. A recent paper, focusing on offloading computation, showed that cloudlets still have the potential to approximately halve response times, as compared to offloading content to the cloud altogether [41]. However, determining what to prefetch in this case is also a problem.

We show an example system architecture for a cloudlet prefetching system in Figure 8.1. A cloudlet server near the access point acts as a proxy, and can observe all traffic that passes through it. It can then generate a model of what to prefetch, and use that model to prefetch content to a local prefetch cache. One advantage of a cloudlet is that the cost of mispredictions is lower, since unnecessary content never goes over the congested

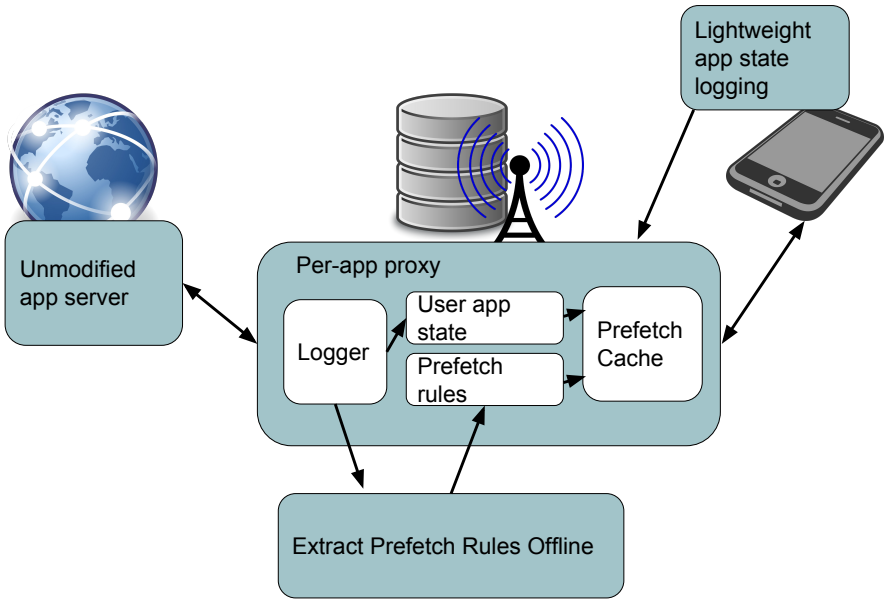


Figure 8.1: System diagram of how a cloudlet prefetching system might work.

and possibly expensive last hop. However, as we will show, one major challenge is that these mispredictions still take time to download, and there is a limited amount of time for prefetching to happen. Finally, there could potentially be some communication between the phone's OS and the cloudlet. In particular, we could communicate to the cloudlet what activity the device is in, or what apps have started, to let the cloudlet know what to prefetch.

System design goals: We therefore want to create a system that can a) predict as much network traffic as possible for an application, and b) do so with reasonable data download overheads. If we use the cloudlet system as our motivating example, we can assume the device has many times the capacity of a phone, and so we can err somewhat on the side of downloading more. We consider several metrics in evaluating the overhead of mispredictions: whether we can store data for a reasonable number of apps and servers on a single cloudlet, whether we can migrate data between cloudlets in a timely manner, and whether we can load the prefetched content sufficiently quickly.

Thus, we evaluate:

1. The amount of traffic successfully predicted
2. The amount of extra data we try and download unnecessarily
3. The estimated number of users/apps that can be stored on a cloudlet server, and difficulty of transferring the data between cloudlets
4. The estimated time and overhead of a migration, and;
5. The estimated time to fetch the prefetched content.

We are able to get positive results for the first metric, and reasonable results for the second if we are targeting cloudlets and can thus afford some excess downloads, but the final three points remain a challenge and will likely require new research directions to properly address. Finally, we do not yet evaluate the latency savings for prefetching, since due in particular to our last metric, it is not yet possible to build an effective prefetching system.

Assumptions and limitations: We only apply this method to certain types of traffic. We focus on apps (that aren't browsers), which usually have a constrained set of requests that they make, rather than web browsing, where the content fetched draws from a virtually limitless set. We also focus on apps that have network traffic, where the content fetched is pregenerated (i.e. no chat or video apps, or games).

We also assume that the cloudlet can be trusted, and thus can observe network traffic and make decisions as to what to prefetch. Some prior work examines how to use untrusted cloudlets [39], but ultimately, someone has to know what network traffic is being sent for this to work due to how we determine what to prefetch.

Much of our analysis is trace-based, which doe not include the server's response to malformed packets, or any rate-limiting done by the server, which we find to be a major limitation of a real-world deployment. We find that the cooperation of the server to, for instance, not log you out after sending a bunch of invalid requests, is necessary. For a

real-world deployment with the full cooperation of app developers this should be generally obtainable.

8.3 Activity prediction

The first aspect of application behavior predictability we examined was Activity transitions. Activities [8] are focused pieces of app functionality associated with a full-screen UI, such as a screen to search for restaurants or one that displays information about a restaurant. As users interact with an application, they traverse different Activities. By predicting Activity transitions, we can predict in a coarse grained way what an app will do.

To determine if Activity transitions are predictable, we make use of the Phonelab testbed [80]. For this testbed, researchers at the University at Buffalo have created a modified version of the Android OS that logs a variety of information of relevance to researchers. They then have volunteers use the phones. For our project, we introduce a new set of experiments to PhoneLab, that instrument Activity and other UI changes, logging the time, app, Activity, and type of transition (e.g. start, resume, stop, pause). We got a sample of data from 86 users over seven days, although not all users were active over all seven days.

Next, we examine whether the next Activity is predictable given the current Activity. We use a simple Markov chain approach, where we predict the next Activity based on what is most likely given the current Activity, base on how often it has lead to each Activity in the past. As we show in Figure 8.2, in almost every case guessing just the most common Activity would be accurate half the time, and guessing the top three Activities would be accurate almost all the time. Even trying to predict the next two Activities is pretty reliable, as shown in Figure 8.3. Note that since we are tracking transitions, A $\rightarrow$ B and B $\rightarrow$ A are separate pairs.

Finally, we examine the predictability of application entry points. In general, it is possible for more than one Activity to be an entry point. For instance, you might see one Activity when you launch Google Maps by clicking the app icon, and another when Google Maps is

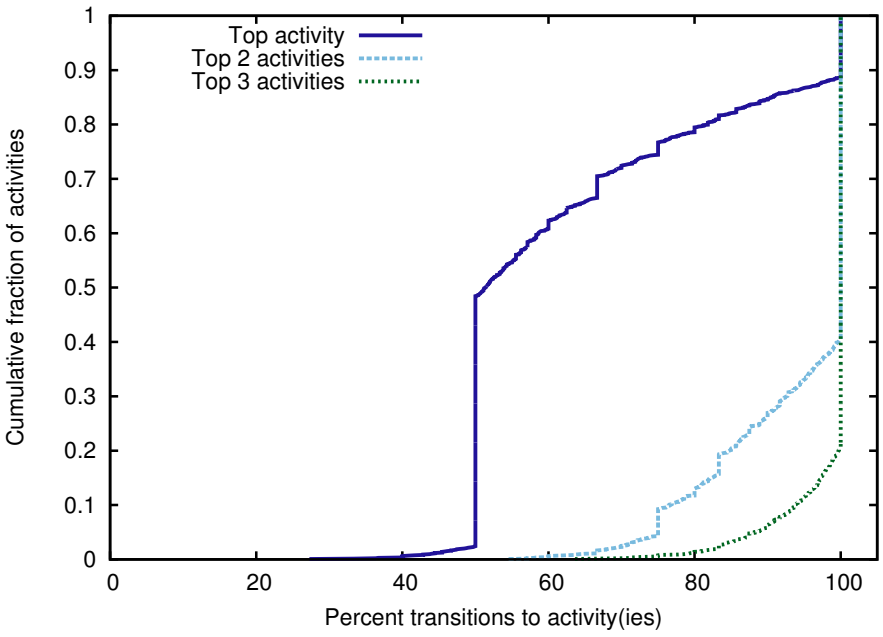


Figure 8.2: Fraction of Activity transitions from each Activity that go to the most common, top two, and top three next Activities.

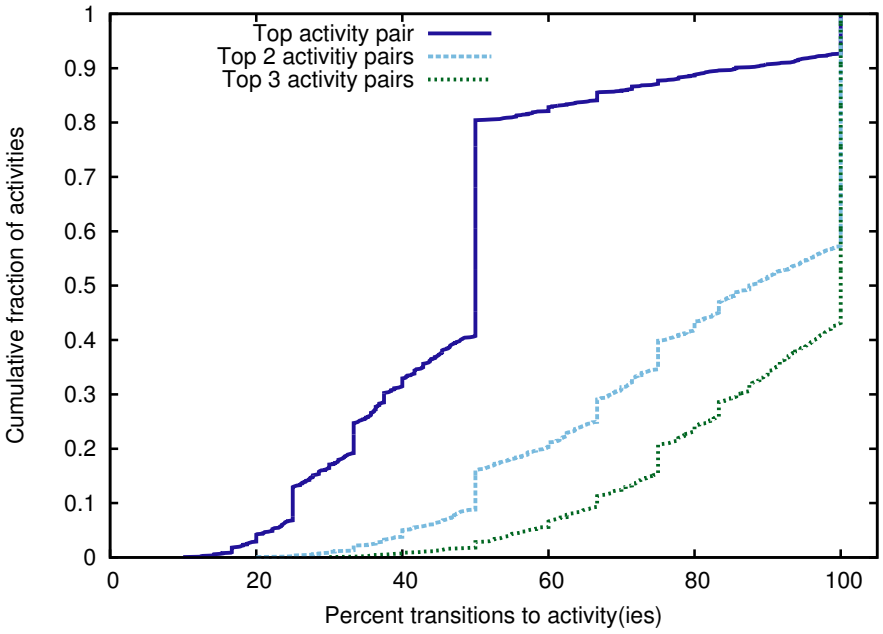


Figure 8.3: Fraction of Activity transition pairs from each Activity that go to the most common, top two, and top three next pair of consecutive Activities.

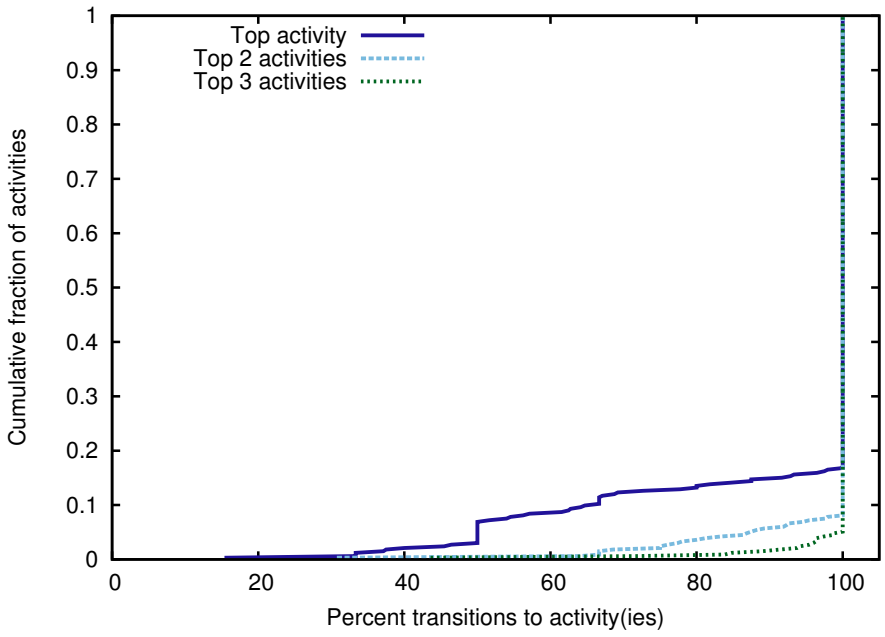


Figure 8.4: Fraction of application entry points covered by the top, top two, and top three next Activites.

launched by another application to give you directions. We expect that in the vast majority of cases, though, only one entry point is used, and as we show in Figure 8.4, this is indeed the case.

For relatively static apps, this suggests we can accurately predict what to prefetch. If the same network requests are made every time for each Activity, then those requests could be prefetched before you go to that Activity. For instance, a weather app might always fetch weather through the same request to the same online API, or a transit app might likewise fetch the current schedule using the same request every time. However, this is not a general approach: most apps are not this simple.

In the next section, we investigate a more sophisticated approach: one where we try and predict *individual network requests*, even when dynamic.

8.4 Traffic prediction

The basic idea of traffic prediction is that most app URLs are either requested at the same point in the app each time themselves, or can be derived in a straightforward way from prior requests. Commonly, applications follow a regular pattern: they load some configuration file, either when the app is launched, or daily, or in response to user interaction, and those configuration files populate the app with specially tailored content such as location- or day-specific content. Usually, the configuration file is a JSON file or XML file which is easily parseable. Additional information, such as the user's location, is often passed with the request. Sometimes, different content is loaded based on user actions, but unlike web pages, often what can be loaded is relatively constrained.

For example, consider a news app that loads content on a daily basis. The app might have an initial XML file with the names of all news articles, and for each, a link to a thumbnail, and a link to the text of the article. Thus, once you determine that the app loads content from that XML file, it is only necessary to determine how the app uses the XML file to fetch the thumbnails and the text of the content, and prefetch that content. There may be a number of pages the user can visit: politics, sports, local news, etc, but there are a limited number of such buttons a user can press, and thus a limited number of XML or other such files. At the very least, we would expect the XML files to be specified in another XML file. For another example, a food recommendation app such as Yelp might load a JSON file containing a list of the top 20 most popular websites in the area, and display them when the user clicks on "restaurants". This method will not cover every case — for instance, once the user decides to filter the websites down to "sushi" we cannot predict that this request will be made — but at least we can predict initial content loads, and if the "sushi" page loads its own XML file of content, we can prefetch based on that. Likewise, for a social media app, this would allow us to prefetch the user's timeline, but if the user searches for another user to add as a friend, that content would not be prefetched.

This method also only works with apps that fit this pattern. Notably, games tend to have

less structured content, and so do not work as well. Ad content, being more dynamically generated, may often not be prefetchable as well. And for an app like Amazon, which searches from a very large collection of content, the possibilities of what to prefetch are too great. However, this sort of prediction is a valuable tool for better understanding and modeling the behavior of a large class of apps.

8.4.1 Overview of Prediction Techniques

We separate the problem of predicting URLs into two parts: predicting the base url, and predicting the parameters. We summarize this in Figure 8.5. For instance, if we have a URL of the form `http://www.example.com/api?user=Alice&location=AnnArbor`, then we first try to predict the url `http://example.com/api`, and then try to predict the parameters that are associated with the URL in order to assemble the full request. The reason for this is that often the URLs and parameters come from separate sources. For instance, the url `http://example.com/api` might be hard-coded into the application as something to fetch whenever a certain activity is loaded, but the parameters come from some user-specific state maintained by the application.

To model the pattern of URL generation where template files such as XML files determine future requests, we look for two types of URLs. We first identify URLs that are statically generated by the app (i.e. the initial template file that, say, fetches the daily news). We then determine how other URLs can be derived from these static URLs. We predict parameters by inferring global application state, partly from prior requests and partly from global device information such as the device's location or the current time. The URLs generated are constructed from both the parameters and the root URLs, as summarized in Figure 8.5.

We developed this method using six apps, CNN, eBay, Flipboard, Groupon, NPR, and Pinterest. These were only used in developing this technique, and not used in our evaluation of this technique to ensure we were not simply memorizing the pattern of a few apps

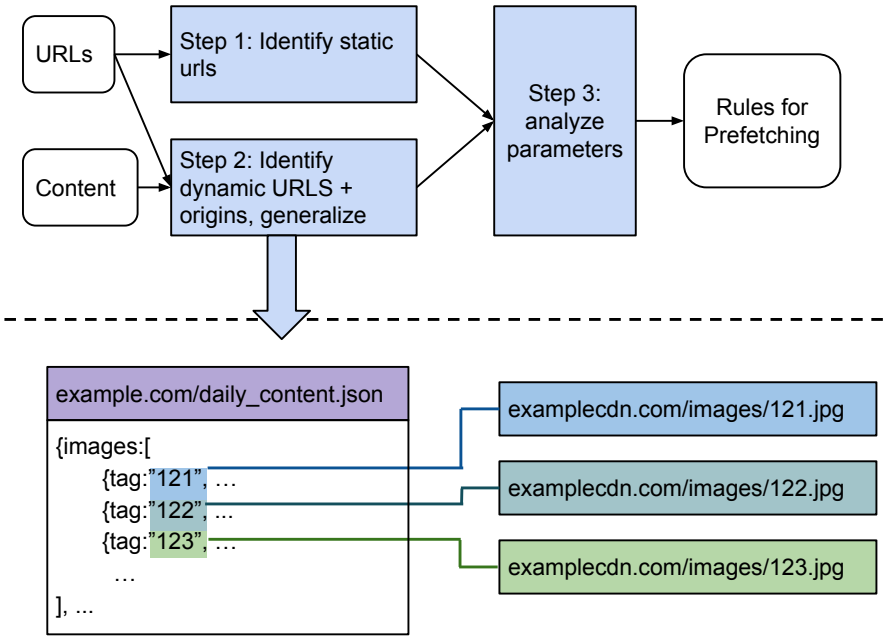


Figure 8.5: Overview of how URLs are predicted based on past URL patterns.

8.4.2 Predicting URLs

To predict URLs, our prediction engine takes in complete traces of network requests, including both the URLs and the content of the requests. We generate the traces by loading the app and manually interacting with it - scrolling all the way to the bottom of the page, then clicking on a few items if relevant and scrolling to the bottom of those. We generated several different such traces.

To create a model for predicting URLs, we go through the trace and attempt to determine where each URL came from. For each URL in the trace, we go through the following steps:

1. Analyze the URL in order to identify the common and unique portions of the URL.

2. Locate the unique portions of the URL in the past trace.

3. Create a representation of that URL portion's location.

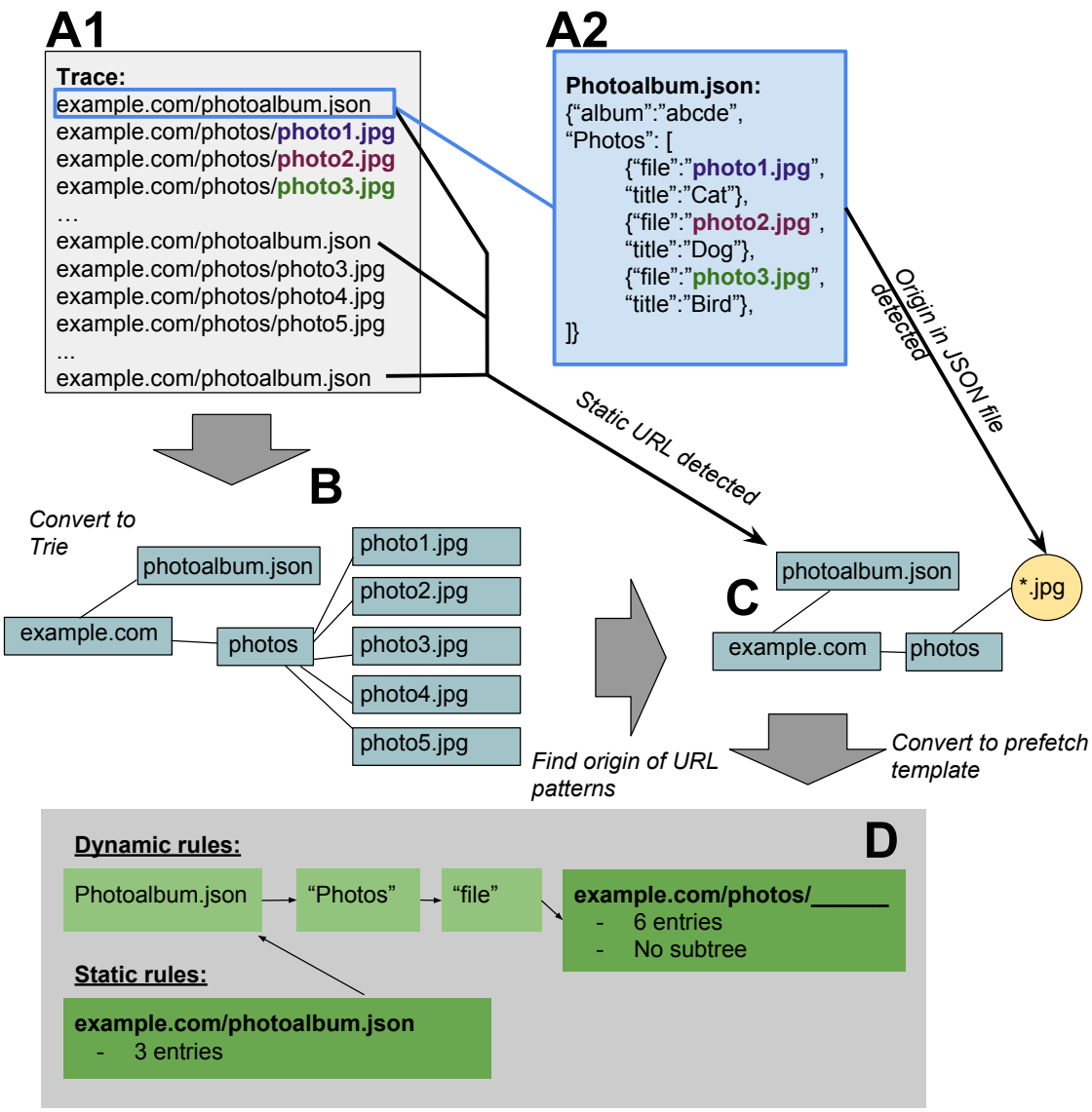


Figure 8.6: Steps to generate a prefetching template.

4. If not found, consider a candidate for being a static URL (a URL which is always fetched), if the URL continues to appear in later traces.

We summarize the steps to generate our prefetching templates in Figure 8.6. We start with a set of traces of network requests (A1) and the contents of the HTML, JSON and XML files in the trace (A2). We next look for URLs in the past, but when we look for those URLs, it is often not the precise URL which is in the trace. For instance, if the URL is `http://example.com/photos/photo1.jpg`, and another URL is

`http://example.com/photos/photo2.jpg`, then the JSON file specifying the first URL might specify `photo1.jpg`. To facilitate this search, we create a trie of all URLs, segmenting them by the portion between the slashes, in order to determine what part of the URL is unique to that URL and what is shared with others (B). The intuition is that the unique portions have to come from somewhere, but the common portions may be hard-coded in the app.

We search backwards in time for the URL, or the unique segment of the URL (C). We support JSON, XML, HTML and some other combinations such as a JSON file containing XML as one of the fields. When we search for the origin of the URL, the complete URL takes precedence over the URL's unique segment: we look for the longest match. We also allow for the portion of the URL in the file we are searching for being a larger match of the URL than the substring we identified, e.g. `photos/photo1.jpg`. We also allow the embedded URL to contain parameters, even though we normally treat parameters separately, since this is usually not the case. In practice, some engineering details such as a variety of methods of normalizing the content of the network responses that we parse are needed.

Once we've located the origin of our URL, we then create a representation of where the URL is located and how one would go about fetching similar URLs in the future (D). We call these representations *templates.* We create a tree of the tags to follow or the lists to expand in the JSON, XML, or HTML file to find the URL. Associated with the leaf node, we keep track of the suffix and prefix of the URL fragment we find there (as relevant). We keep track of how many items we find that match each leaf node.

It is possible that the files which we parse to find URLs lead to loading additional content which can also be parsed to find more URLs to prefetch. Thus, our tree of content to prefetch can have within it additional nested trees to prefetch.

If we don't locate our URL in a file somewhere, we keep track of that URL. If we find identical URLs in three different traces taken at different times, which were not predictable,

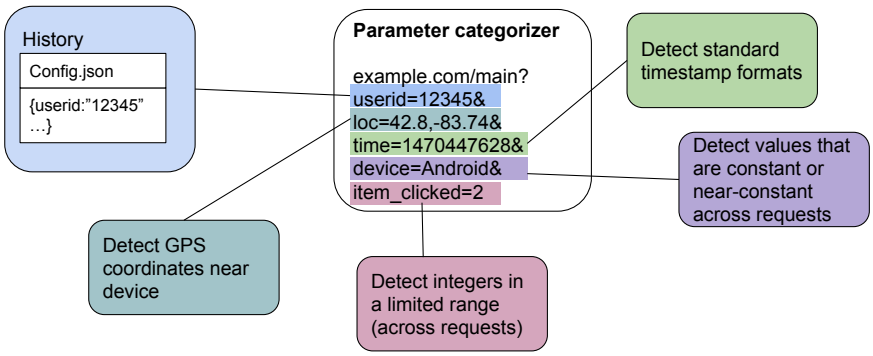


Figure 8.7: Common types of parameters in URLs and how to predict them.

we add them to our list of static URLs to prefetch. In §8.5.3, we show why we picked three as the cutoff.

There are many special cases which we have to deal with, particularly when extracting URLs from the originating files. However, we don't want it to be necessary to manually adapt the code to each individual app we process. Particularly when evaluating apps, we want to evaluate apps with no additional manual effort needed. Thus, we developed our tool to, as accurately as possible, be able to predict content for 6 apps, and then tested on a separate set of apps, without adapting to use those apps specifically, to evaluate how well we can predict URLs with no app-specific training.

8.4.3 Predicting parameters

As mentioned above, we keep track of parameter state globally, as opposed to on a per-URL basis, and don't assume that parameters originate from URLs embedded in JSON or other files. In many cases, parameters represent some global state of the app or device (such as a username, location or preference). We show an example of some of the types of parameters we might see in Figure 8.7.

We associate each parameter with a parameter key, which is the set of all other parameters in that URL. In doing so, we assume that parameters have the same meaning across the application when they are grouped together. For instance, we assume that if you have

the URLs `http://www.example.com/images/photo1.jpg?id=123&loc=AnnArbor`, `http://www.example.com/friends?id=123&loc=Ypsilanti` and `http://www.example.com/homepage?id=123&phone=Android` that the "id" and "loc" parameters refer to the same type of thing in the first URL, but that the "id" parameter might mean something different in the third. This is based on the observation that parameters like "id" may have different meanings in different contexts: future work could explore these assumptions in more depth.

We look for 8 different types of parameters: Unix timestamps; human-readable timestamps in several standard formats; latitudes and/or longitudes; parameters with constant values; parameters which can take on one of three or fewer values; parameters which take on a small range of integers; parameters which come from prior web content, similar to our URL analysis; and other parameters, which we can't currently predict.

With UNIX timestamps and other timestamp formats, it's a matter of looking for strings with particular formats and identifying irrelevant, static characters. For these, we can substitute in the current time in the same format. Likewise, latitudes and longitudes can easily be identified if the current device latitude and longitude is known.

Identifying static values is also a very powerful approach. Usernames, unique IDs, the device type, the user's city, and other such information would be hard to analyze, understand and predict. However, all we need to know is that these URLs change rarely or never for a particular user. Then, it is simply a matter of collecting that information from a few early requests, and then substituting it in as needed. We assume that mispredictions are acceptable, and so if we see fewer than three unique values for a given parameter, we simply try prefetching for all three (as we show in §8.6, though, it may not be so straightforward.) We set a limit of the number of permutations, however, as we found one case where we would have had to prefetch thousands of URLs.

Another common case is one where the parameter is different depending on what item in a list is clicked, and thus the parameter's value varies among a small list of integers. In

these cases, we also prefetch all possibilities, if there aren't too many of them.

Finally, some parameters do come from prior web content. If possible, we try and track the parameters in a different way. To find where they may come from in web content, we use a similar method for finding the location of those parameters' values. We then use our new template to extract the complete set of objects with a similar position in the originating file where we found the parameter. If the set of objects we found is very similar to the set of values we saw for the parameter (Jaccard similarity of 0.95 or more — a lower threshold lead to many false positives), then we have found the source of those parameters. Because these parameters are often very short, we need this extra layer of verification to make sure we've actually found the parameters we're looking for.

We only prefetch if the number of different permutations of parameters is less than 100.

8.4.4 Using The Prefetching Engine

The URL extracting templates, as well as the parameter templates, are placed into a data structure which stores instructions on how to generate URLs and their parameters. This data structure can then predict URLs based on prior network traffic. When a URL passes through the proxy, it is checked to see if it contains content needed to make prefetching decisions. The first such type of URLs are static URLs, but when these URLs are parsed to determine what to prefetch, these prefetched URLs may in turn lead to further content to prefetch after parsing them. These nested parsing decisions are indicated at the leaf node of the parsing tree.

At the same time, this data structure gathers information needed to maintain the state of URL parameters. Thus, when prefetch decisions are made, the URL is populated with the appropriate parameters and their values. In addition, we keep track of a list of every URL that gets prefetched.

We use a very simple statistical model for determining what to prefetch: if we have seen content a few times in the past, we prefetch it in the future. We explore the parameter

of how often we should have seen content in the future in §8.5.3

Our analysis in this section is trace-based. As we show in the next section, there are currently some fundamental obstacles to a real-world implementation, but we can at least analyze the accuracy of the prediction aspects. For this analysis, we simulate at each point in time receiving a URL, parsing it as appropriate, and determining what we would want to prefetch, including the parameters we would append to the URL. Then, in the future, when we fetch a URL, we first check if we correctly predicted the URL, as well as its parameters. We allow the timestamp to have changed, and the location to have drifted slightly as well. We simulate different levels of server delay between when we receive and parse an object and when we can assume we've prefetched the data by. We assume a 100 ms server processing delay to analyze content and prefetch objects. In practice, this may not be fast enough.

8.5 Evaluation of URL prediction

We chose 29 popular apps, selecting apps from a variety of categories in the app store, as well as the most popular apps overall. We chose apps that had a network component and that were not games, but aside that attempted to choose a variety of apps representative of popular apps generally. However, when evaluating the results, it is important to consider that apps expected not to work are specifically excluded.

We generated network traces by manually loading the page, refreshing it, scrolling down, and clicking on a few items. We generated at least 5 traces per app, each with as similar a set of user actions as possible. We expect there to be some natural variations between when items load, but we are not predicting user actions in this case. Determining how to predict traffic given a variety of user actions remains an open problem to address in the future, and would likely require a full user study to predict and characterize user browsing behavior and its implication for predicting network traffic.

We only prefetch one level of nested prefetch content at a time: even if an item we

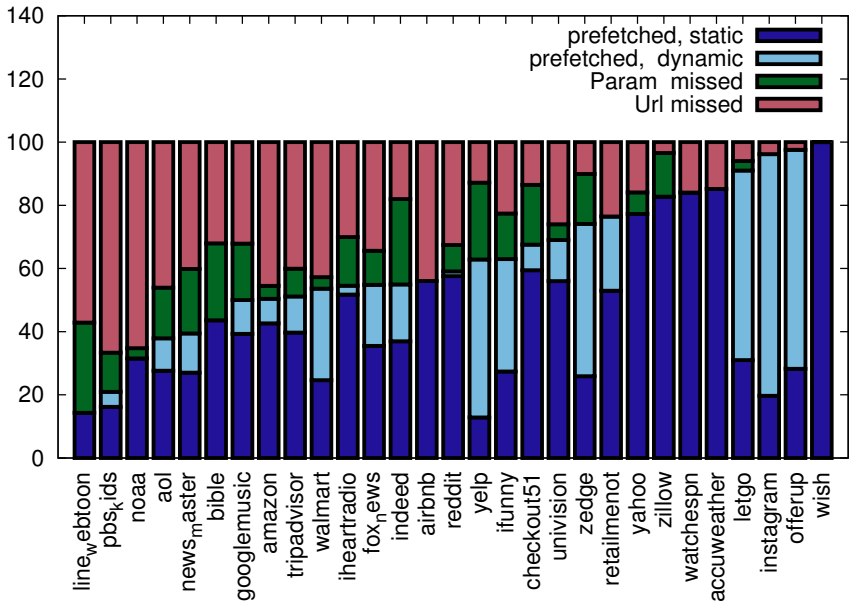


Figure 8.8: Distribution of successfully prefetched objects by application.

prefetch in turn contains content to be prefetched, we don't prefetch that second level of content until there is a hit in the prefetch cache. As we show, this keeps the false positive rate, and thus the number of items downloaded, under control.

8.5.1 Results of trace-based prefetching simulation

We show the number of objects we successfully prefetch in Figure 8.8, and the amount of data we successfully prefetch in Figure 8.9. Note that these values only include data that is prefetched that was later used; in the next section we discuss unnecessary downloads. We sorted the pages by the amount successfully prefetched, in those figures and we divide the content successfully prefetched into static content and dynamic content. Note that more prefetched objects are static, but a larger amount of prefetched data is dynamic: dynamic content often consists of images that change from day to day, while static content is more likely to be JSON or other text-based files. We also divide the content we did not successfully prefetch into content we missed because we didn't predict the URL, and

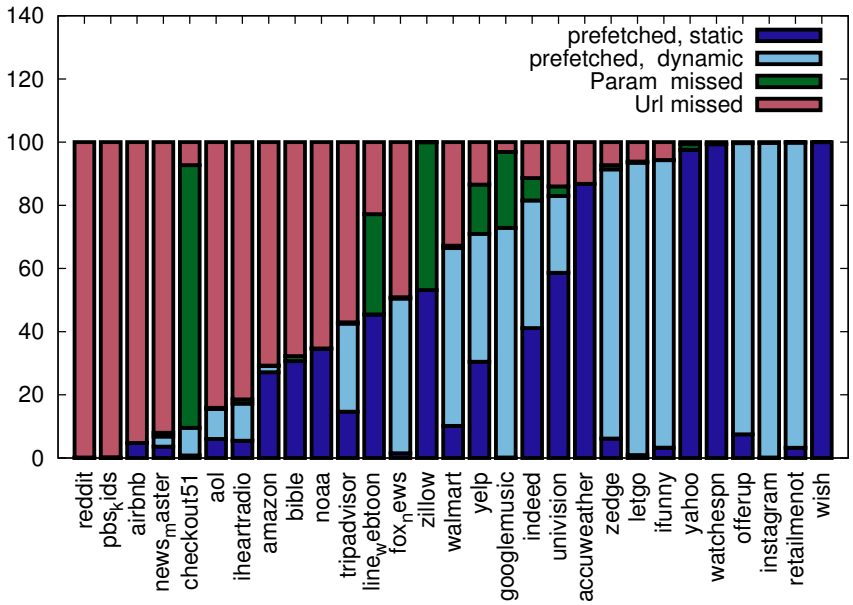


Figure 8.9: Distribution of the amount of data successfully prefetched by application.

content where we predicted the URL but not the parameter. The latter category is not all that large: pages with complicated, hard-to-predict parameters may be more likely to have complicated, hard-to-predict URLs.

8.5.2 Wasted downloads

One problem is the amount of downloaded content wasted due to excessive prefetching. In order to examine this, we keep track of all the URLs we predict and count how many never were used. Our evaluation of wasted downloads in the trace-based analysis is thus the number of unnecessarily downloaded objects, and not the amount of data. On the one hand, we may be downloading many large objects; on the other hand we may be just making a lot of rejected requests which return small error messages.

The results of examining the number of objects downloaded unnecessarily are shown in Figure 8.10. We show the total number of objects, as well as the objects where the parameters were mis-predicted, the number of static URLs that were fetched unnecessarily,

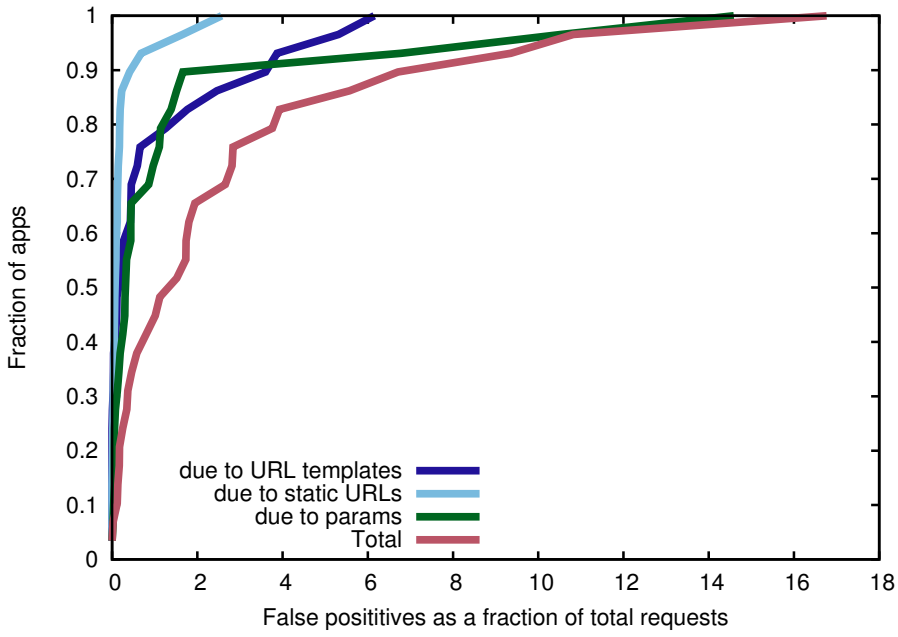


Figure 8.10: False positive rates of prefetching content, without immediately following nested links.

and the number of URLs fetched unnecessarily due to the URL templates generating unused URLs. First, the overhead of this prefetching is substantial. The median overhead is a little under 2x, and in the worst case can be as high as almost 17x. This motivates the use of cloudlets, as they would be able to tolerate this overhead.

It is also apparent that mispredicting static URLs does not contribute substantially to the overhead. Somewhat more surprisingly, mispredicted parameters are only responsible for about half the remaining excessive downloads, although in the worst case they can add substantially more overhead. This overhead would likely also depend on user behavior. For instance, for a news app, the user in the trace would view only one or two items. However, the prefetching template will prefetch all the articles. Thus, for someone who systematically goes through and reads each news article, the overhead of wasted downloads would be lower. We don't make this assumption, though, to avoid understating the overhead of prefetching.

In Figure 8.11, we examine the possibility of prefetching two levels of content —

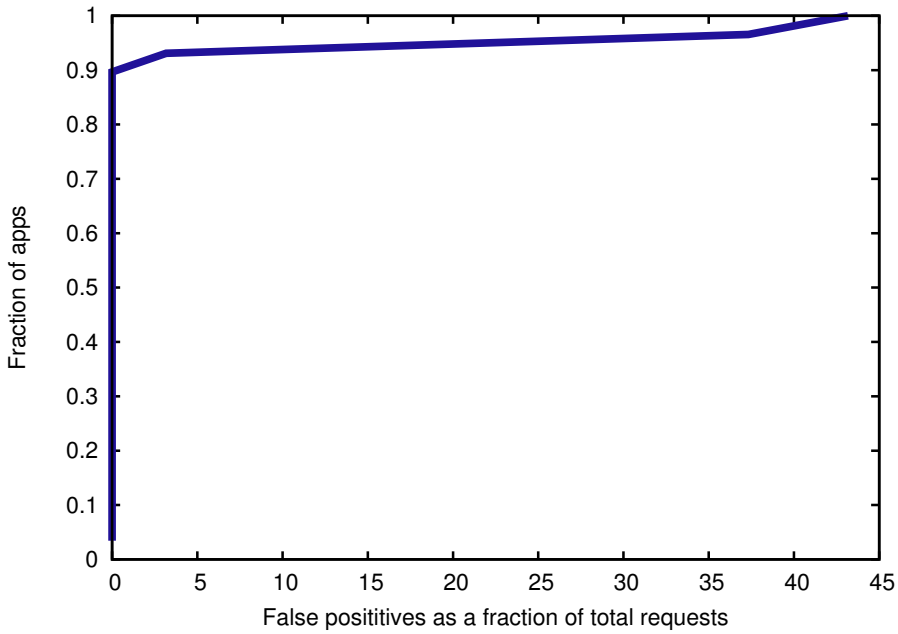


Figure 8.11: False positive rates of prefetching one additional layer of nested content.

after downloading an item, parsing it, and prefetching content accordingly, parsing the prefetched content to find more items to prefetch. Unfortunately, this approach for some apps can lead to unreasonable overheads, over 40x in one case. Fetching the 10 items linked to on a page might make sense, but fetching another 10 items for each of those would be excessive. Thus, for the analysis above, we only prefetch one layer of content.

8.5.3 Tradeoff between accuracy and excessive downloads

The accuracy and false positive rates shown depend on several parameters: the more certain we have to be before prefetching content, the less we will prefetch, but the less we will download unnecessarily. There are two main parameters we consider that determine our accuracy: how often we must have seen a static object before prefetching it, and how much we must have seen a set of dynamic objects corresponding to a template entry before seeing the object in the set.

The static case is straightforward. If we see an object more than N times, we download

it. We had this set to 2 in the tests above, i.e. if we see two identical requests we would prefetch: this was a fairly aggressive approach, which we believe can be justified based on our cloudlet design. In Figure 8.12, we show that it's necessary to filter so that only requests that are seen several times are made. If we prefetch everything we've seen before, the false positive rate is very high, as we would expect.

In the dynamic case (Figure 8.13, we look at how many *template matches* we hit for a given template. For instance, if we found that for a configuration file, matching "reviews" $\rightarrow$ "userdata" $\rightarrow$ "images" matches 5 different files which we wound up downloading later, and we have a cutoff of 5 or more, then we will prefetch objects that match that template in the future. As such, the number of matches that we can consider before prefetching is higher.

In order to prefetch, the number of copies must be greater than the number indicated as the cutoff threshold, so for a cutoff of zero, every template would be used if it could have been used to predict at least one URL in the test set. For small numbers (less than about 20), the false positive rate is significantly higher than the amount of data or number of objects prefetched. For a cutoff larger than 50, there are relatively few false positives, but also less is prefetched from the templates. Also, note how the amount of data prefetched falls faster than the number of objects: we're losing the ability to prefetch larger objects first. However, when false positives are a concern, setting the cutoff even at 6 seems reasonable.

8.6 Cloudlet Feasibility Analysis

Having examined our prediction system using trace-based analysis, we then examine the feasiblity and challenges of using it in the real world. We use cloudlets as our main motivating system, and start by considering what sort of overheads would be of concern for these systems. We examine how the total data downloaded can impact the storage required and the time to migrate data between cloudlets. We then build a simple prefetching proxy and find that there are challenges with being able to download prefetched content in time

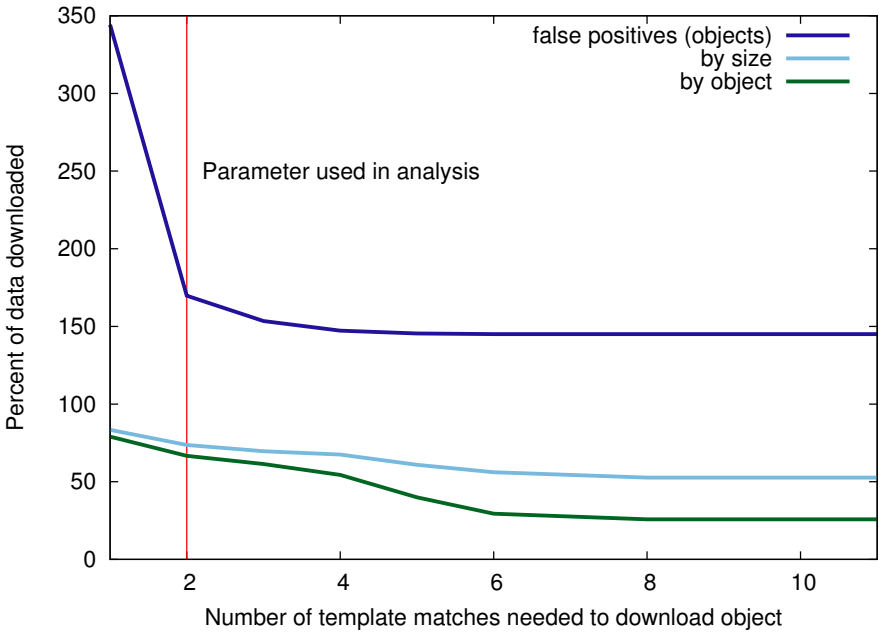


Figure 8.12: Impact of varying the parameter for how many times we need to have seen a static URL to prefetch it late.

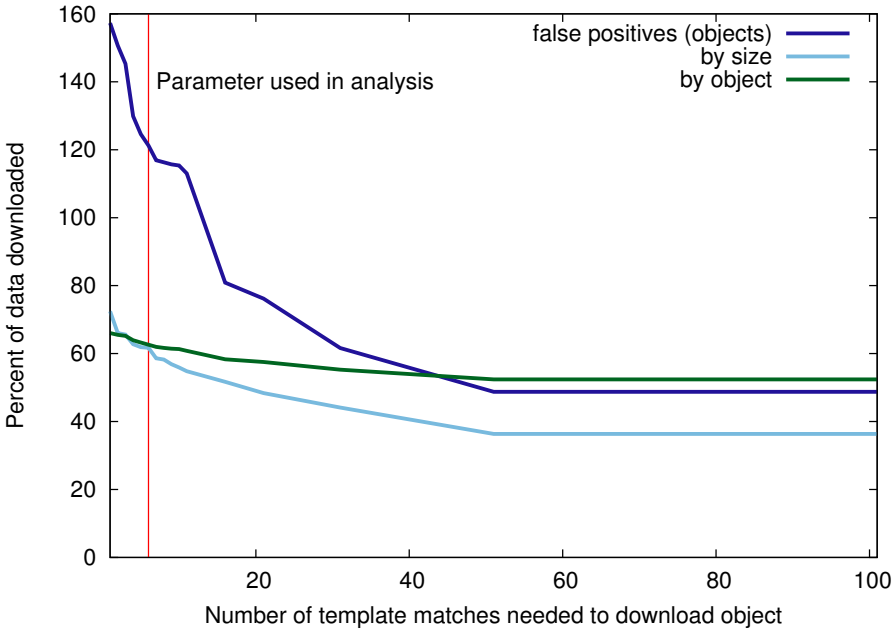


Figure 8.13: Impact of varying the parameter for how many times we need to have seen a match for this URL pattern when training in order to prefetch it later.

before it is needed.

First, we estimate the impact of the volume of downloaded content on a cloudlet. We use an approach similar to our analysis in the last section, only now we guess what the size of the content would be. We assume that for the objects we mispredict, their sizes are on average similar to the average size of the content we did download. For calculating false positives due to prefetching the right URL with the wrong parameters, we take as the object's size the size of the URL with the correct parameters. We expect this is a conservative estimate, as in many cases a smaller error message would be returned instead of real content.

We assume each app and each user has independent data stores and that we thus aren't optimizing by having only one copy of each object globally. This sort of optimization could potentially introduce privacy issues. We calculate the amount of data needed across each of the sessions we recorded, which are several minutes long each.

For our sessions, we average about 41 MB, including both static and dynamic content, but the median was 5 MB. As shown in Figure 8.15, the mean is distorted by a small number of apps (Instagram and PBS Kids in particular) which use over a hundred megabytes, due to their heavy use of high-resolution images and video, respectively. Overall, storage is not a problem. A 500 GB hard disk costs about thirty dollars when bought individually[1]. It is thus reasonable to support about 100,000 app instances for individual users on a single cloudlet, or 12,500 if we take the average instead of the median. Even with a disk of a few gigabytes only, a significant number of apps could be supported.

A bigger problem is bandwidth. It appears that the average bandwidth in the US is around 18 Mbps[2]. Assuming we can saturate the bandwidth, or most apps, several could be forwarded to the next cloudlet each second, but for something like Instagram, it could take 15 seconds or so to transfer all the data. If the user has several such apps, it could take

[1] https://www.amazon.com/Seagate-Pipeline-3-5-Inch-Internal-ST3500312CS/dp/B002CMOH26/

[2] http://gizmodo.com/americas-internet-inequality-a-map-of-whos-got-the-b-1057686215

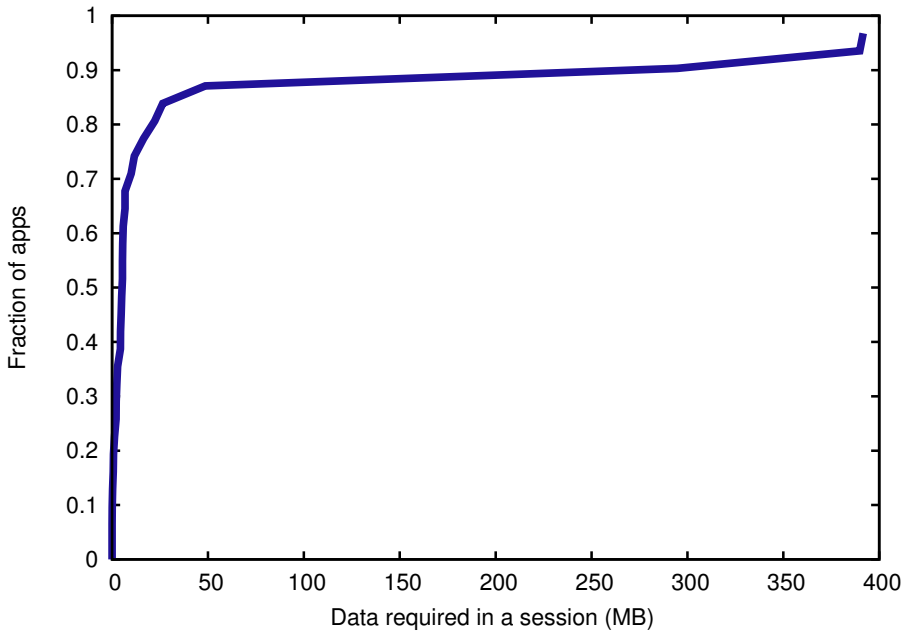


Figure 8.14: Amount of data, total, downloaded for a short session of using an app.

minutes. Given the low cost of storage, one solution could be to store data in advance on cloudlets the user is likely to visit next. Another solution would be to only focus on apps with lower sizes. We could also focus on the cases where the user is stationary for a period of time and not guarantee that Server Push will work for the first few minutes after moving elsewhere.

In the case of WiFi, cloudlets could be associated with a building rather than a specific AP, limiting the amount of data that needs to be moved around the building. An examination of traceroute results from WiFi in this building shows that the first hop after the wireless access point in the BBB (which, in fact, is in the School of Information building and not the BBB) is less than 2 milliseconds away, whereas a server on the other side of the continent `https://berkeley.edu`) is about 77 milliseconds away. This suggests for something like a university campus, only a few cloudlet sites may be needed, since in many cases the last hop adds only a small fraction of the latency, which would limit how often data would need to be migrated.

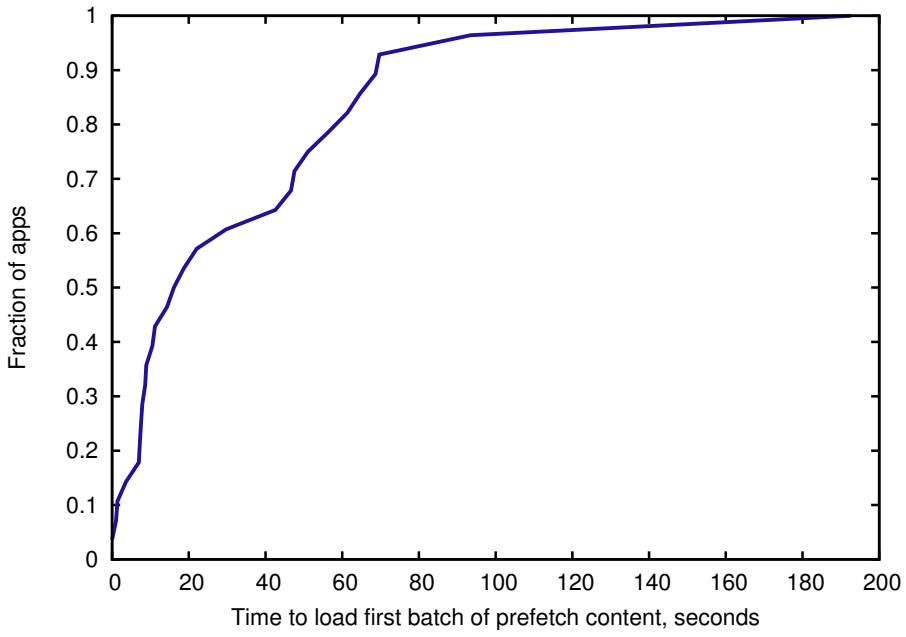


Figure 8.15: Time to download and process the content to be loaded on initial batch load.

We created a simple prototype to examine some of the challenges of applying prefetching to real devices. We were not able to get prefetching to work effectively in practice, primarily because content wasn't loaded fast enough. As we show in Figure 8.15, the time to download and parse the content could be quite substantial, and the apps with short download times often did not have much content that could be prefetched. This is an initial prototype, and not heavily optimized, but aggressively fetching a large amount of content can be slow, and we need to get the download and processing time down to about the time to download a single object to see benefits. A method of identifying or prioritizing the content to prefetch is could alleviate this problem. We also ran into some other engineering problems, such as the account apparently becoming logged out due to the requests made, which prevented prefetching from working effectively.

8.7 Conclusion and future work

This chapter is a first look at the problem of building a prefetch system which can automatically determine what to prefetch. I have focused on the problem of predicting content to fetch. As with other work in this dissertation, this involves observing and measuring characteristics of network traffic and making intelligent decisions based on the data collected. While it is not possible to determine what traffic an app will send in every case, a substantial amount of traffic can be predicted by leveraging the well-structured traffic patterns that many apps have. We also investigated other ways of predicting app behavior, in particular predicting activity transitions.

We have also discussed the challenges in building a full, practical prefetching system. The main problem is that the overhead, in terms of time to download data as well as to move that data around, remains excessive, and the data cost can be substantial as well. To be truly practical, only a few objects should be prefetched, and thus future work should examine what to prioritize. Work such as Wprof [126] has looked at dependencies and work building on that may be able to identify what content is on the critical path for generating the Activity (although the approach would need to be modified to track rendering dependencies in Activities rather than pages). Then, approaches to predict what content is on the critical path would be needed. Perhaps in many places the same items in the prefetch template are on the critical path, or perhaps some sort of machine learning technique could predict the key URLs.

Once these obstacles are overcome, a realistic evaluation on a real deployment would be possible. A user study would be a good method to make sure that the prefetch predictions are truly representative. Overall, this project has opened the door to promising new research directions that could lead to a practical, automated prefetching system.

CHAPTER IX

Conclusion

I have discussed five major projects in this thesis. Three focus on better understanding network performance and power consumption on mobile devices in order to understand how to build better mobile systems. Each of these suggests that app developers, network operators, and web page developers should make decisions based on comprehensive, ongoing measurements. The other two show how measurements could be used to support prefetching systems that improve user performance.

First, *Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks* examine how RRC states impact user-perceived performance in the wild, in particular state demotions. It introduces a technique for collecting an ongoing picture of RRC states and their performance impact globally. This is something which app developers could make use of when determining the power and performance tradeoffs they make when scheduling their newtwork traffic. It also uncovers a previously unknown performance problem, which affects only certain carriers, which apps should avoid. We suggest that carriers use ongoing measurements of RRC state performance to detect and address similar problems that crop up as cellular networks evolve.

We then examine the network power consumption of real apps through an ongoing user study in *Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild*. Here, we uncover a number of performance problems, both unknown problems and

problems which persist despite being well-known. We propose, based on these findings, that it is necessary for a service within Android to monitor application behavior and block network traffic that is likely unneeded but that nevertheless drains the user battery.

Next, we examine HTTP/2's Server Push, with an emphasis on whether it would be successful in improving networking performance on mobile devices, in *Push or Request: An Investigation of HTTP/2 Server Push for Improving Mobile Performance*. We determine that Server Push would be more effective at improving performance for cellular and WiFi networks than Ethernet networks, but that the performance benefits are mixed, depending heavily both on the web page and on the network conditions. Furthermore, at this point, due to how web pages are structured, a proxy would likely be needed to support Server Push since content is split over too many servers to work well with Server Push. This proxy could selectively push content based on factors such as observed network conditions and the performance of particular web pages.

All of these measurement studies suggest that there is a place for systems to intelligently leverage this measurement data in order to ensure improved performance. The final two projects focus on building systems that leverage measurements more directly.

For *CellShift: A System to Efficiently Time-shift Data on the Cellular Network*, we simulate a system that time-shifts network requests on the order of hours, by leveraging network load measurements at each eNodeB. This simulation uses real network performance data across a city. We find that we can predict an eNodeB's future load highly accurately, and predict the load a user will experience in a location-agnostic manner.

For *Predicting App Network Traffic to Facilitate Prefetching*, we examine how we can enable prefetching to cloudlets by predicting network traffic in advance. By observing prior network traffic, we are able to make intelligent decisions on what to fetch. While we only address the problem of predicting app behavior, and many open problems remain in the construction of a practical prefetching system, this system represents a promising first step in the direction of automated prefetching.

9.1 Discussion and Future work

Next, I examine several ways that future research could build on this thesis.

Enhancing the current analysis performed: For each project, there are a number of ways in which the analysis can be expanded upon. For the *RRC State Inference* project, a larger dataset could detect differences between device types or regions, and a more longitudinal one could determine if any changes have occurred over time. While we have not detected any signs of dynamic approaches to setting RRC state timers, as RRC state management continues to evolve, adapting our inference method to detect those cases would be useful.

The *Network Energy Efficiency* project could be improved by analyzing what content is actually viewed by the user or otherwise necessary, perhaps through Taintdroid [31] or another such system, to see how much background traffic could actually be eliminated. We also focus on background traffic, that being a likely source of more unnecessary traffic, but there could be inefficiencies in foreground traffic as well.

For the *Server Push* study, we do not examine stream prioritization. Instead, we use the default prioritization scheme, but exploring the tradeoffs of scheduling traffic in different orders would be potentially an interesting research direction. Similarly, we don't examine using existing systems to determine what content is on the critical path. In fact, examining how to use a system like WProf to handle stream prioritization would be an interesting research project in and of itself. Looking at how Server Push might be used for apps would also be an interesting direction for future work.

For Cellshift, while we have ample access to global load data, the content we time-shift is somewhat artificial, since we don't have details of the actual content sent. While it may not be possible to actually gain access to this data for privacy or legal reasons, detailed per-user network traces could be used in conjunction with the load and location data to more accurately simulate prefetching.

Building real-world systems: The three measurement studies each suggest a system

to dynamically leverage their measurements would be valuable. For the *Network Energy Efficiency* project, we propose that an Android system module should monitor application behavior, detect energy bugs, and fix them, such as by suppressing background traffic. Similar solutions to the ones proposed in our paper were implemented by Google (concurrent to our work), but a comprehensive evaluation of a variety of actual implementations could be informative. For the RRC project, having a system library that assists apps in scheduling traffic with the right power and performance tradeoffs, based on per-carrier RRC state performance characteristics loaded from a central database, would allow energy and performance problems to be evaded. How exactly to best make these tradeoffs in an automated way could also be explored.

For the Server Push study, a real-world proxy deployment based on our findings would allow Server Push to be much more effectively used. Even better, a system that adapts based on the network type — or even on current network conditions — and perhaps only starts pushing later in a user session, would allow Server Push to be used much more effectively. Our findings suggest that for Server Push to be used effectively, an intelligent approach that leverages measurements and observations of the client would be needed.

The *Cellshift* study was done essentially in simulation. For that project, a city-scale evaluation wouldn't be possible, but a user study of a sample of users in a town with a limited number of access points to monitor might be possible. It wouldn't be possible in that case to measure how network load can be reduced, as we couldn't recruit the thousands of users that would be needed to make a difference, but it might be possible to improve network performance for a few users by avoiding congested times.

Building a complete automated prefetching system: Finally, for the *App Traffic Prediction* project, a major problem is the high false positive rate. Reducing these false positives by determining what content actually impacts the UI, perhaps building on the work we did in QoE Doctor [20], would likely allow the network load due to unnecessary fetches to be reduced, since the number of target URL types to prefetch would be smaller. An-

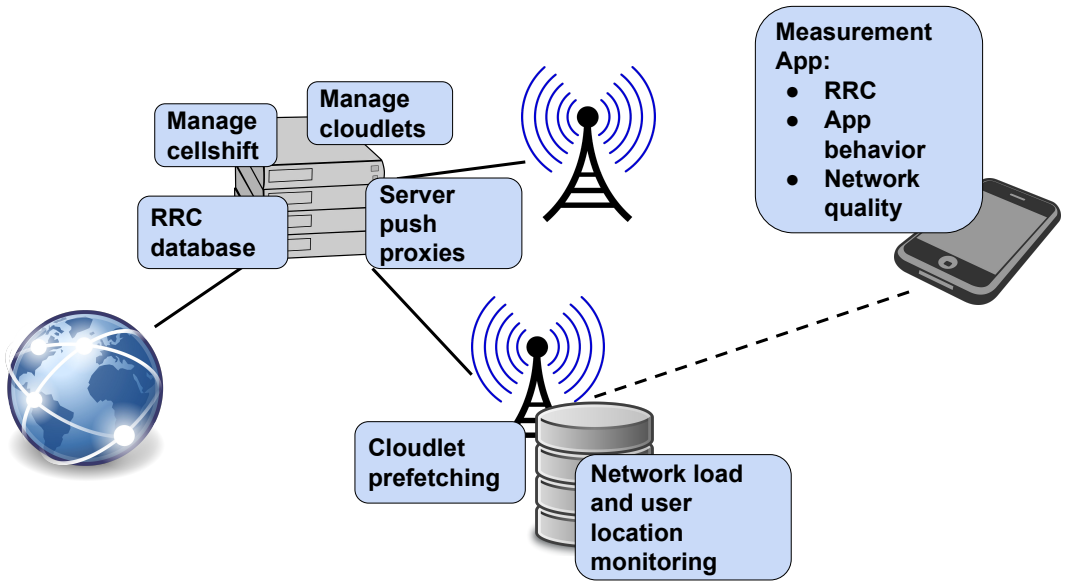


Figure 9.1: Overview of potential comprehensive measurement-oriented app and traffic management

other possibility might be to apply some sort of machine learning technique to filter out URLs unlikely to benefit from prefetching. Hopefully, this would reduce the overhead of prefetching content enough that our prediction framework would be useful. Overall, this seems like a potentially fruitful avenue for future research. Finally, a real-world system could be set up in a university building, with a number of cloudlets attached to WiFi access points. In addition to allowing the performance benefits of Server Push to be tested, the impact of migrating between access points could also be tested.

A network-wide system of comprehensive, intelligent traffic scheduling: Once it has been shown that building these systems is possible, and that they provide effective solutions to each of the problems they deal with, one can envision a network which transparently makes traffic scheduling decisions throughout the network to ensure good performance on mobile devices, summarized in Figure 9.1. In the middle is a powerful proxy, which selectively enables Server Push, monitors and predicts network conditions to schedule traffic over long time periods, orchestrates the cloudlets, and provides general RRC

state information to devices. Data would be prefetched to cloudlets based on a comprehensive network traffic prediction engine, and data would be loaded into trusted per-user VMs on each cloudlet. Devices would be adapted to use information collected from the network, including network performance trends and RRC performance, to better schedule network transitions, both on scales of hours and scales of seconds. These devices could also send back measurements to the central server about network conditions observed by the client, as well as expected network demands and user mobility. This system would be especially suitable for cellular networks, where central control of the network already exists, but a form of it might exist on university campuses or on particularly large corporate networks. Overall, through network measurements, a more intelligent, responsive and better performing network is possible.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Android Activity Testing. http://developer.android.com/tools/testing/activity_testing.html.

[2] Android DDMS. http://developer.android.com/tools/debugging/ddms.html.

[3] 3GPP. 3GPP TS 36.211 Physical Layer Measurements.

[4] 3GPP TS 35.331: Radio Resource Control (RRC) - UMTS, 2013.

[5] 3GPP TS 36.331: Radio Resource Control (RRC) - LTE, 2013.

[6] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's data compression proxy for the mobile web. In *Proc. NSDI*, 2015.

[7] ActivityManager.RunningAppProcessInfo documentation. https://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo.html.

[8] Android Developers. Activity. https://developer.android.com/reference/android/app/Activity.html.

[9] Apache Module mod_http2. https://httpd.apache.org/docs/2.4/mod/mod_http2.html.

[10] G. Association. Fast dormancy best practices v1.0, 2011.

[11] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, K. Papagiannaki, J. Crowcroft, and D. Wetherall. Staying Online while Mobile: The Hidden Costs. In *Proc. ACM CoNEXT*, 2013.

[12] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*.

[13] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi: Measurement, Design, and Implementation. In *Proc. ACM MobiSys*, 2010.

[14] M. Belshe, R. Peon, and M. Thomson. Hypertext transfer protocol version 2 (http/2). RFC 7540.

[15] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *Proc. ACM IMC*, 2011.

[16] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[17] C. C. Tossell, P. Kortum, A. Rahmati, C. Shepard, and L. Zhong. Characterizing web use on smartphones. 2012.

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

[19] G. Carlucci, L. De Cicco, and S. Mascolo. Http over udp: An experimental investigation of quic. In *Proc. ACM SAC*, 2015.

[20] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau. Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis. In *Proc. ACM IMC*, 2014.

[21] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *Proc. Sigmetrics*, 2015.

[22] X. Chen, J. Erman, S. Lee, and J. Van der Merwe. Mercado: Using Market Principles to Drive Alternative Network Service Abstractions. 2012.

[23] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014-2019. Technical report, Cisco, February 2015.

[24] C. Cox. *An Introductio to LTE*. John Wiley & Sons Ltd, 2012.

[25] K. Crawford. Battery life: How does the android battery tool work, and why should developers care? `https://www.apteligent.com/developer-resources/battery-life-how-does-the-android-battery-tool-work-and-why-should-developers-care/`.

[26] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. ACM MobiSys*, 2010.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. 2007.

[28] Developer preview - power-saving optimizations. `https://developer.android.com/preview/features/power-mgmt.html`.

[29] E. Halepovic, J. Pang, and O. Spatscheck. Can you GET Me Now? Estimating the Time-to-First-Byte of HTTP Transactions with Passive Measurements. 2012.

[30] Y. Elkhatib, G. Tyson, and M. Welzl. Can spdy really make the web faster? In *IFIP Networking*, 2014.

[31] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. Operating Systems Design and Implementation*, 2010.

[32] J. Erman, A. Gerber, K. K. Ramadrishnan, S. Sen, and O. Spatscheck. Over the Top Video: The Gorilla in Cellular Networks. In *Proc. ACM IMC*, 2011.

[33] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a spdy'ier mobile web? In *Proc. ACM CoNEXT*, 2013.

[34] K. R. Evensen, D. Baltrünas, S. Ferlin-Oliveira, and A. Kvalbein. Preempting State Promotions to Improve Application Performance in Mobile broadband Networks. In *Proc. ACM MobiArch*, 2013.

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

[36] F. Qian, Z. Wang, A. Gerber,Z. M. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Proc. ACM MobiSys*, 2011.

[37] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in Smartphone Usage. In *Proc. ACM MobiSys*, 2010.

[38] J. Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing*. Morgan & Claypool Publishers, 2012.

[39] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanaryanan. Data staging on untrusted surrogates. In *Proc. USENIX FAST*, 2003.

[40] Flipboard. Good news for commuters: Mobile data options. `http://inside.flipboard.com/2013/03/22/good-news-for-commuters-mobile-data-options/`, 2013.

[41] Y. Gao, W. Hu, K. Ha, B. Amos, P. Pillai, and M. Satyanarayanan. Are cloudlets necessary? Technical Report CMU-CS-15-139, School of Computer Science, Carnegie Mellon University.

[42] A. Gember, A. Akella, J. Pang, A. Varshavsky, and R. Caceres. Obtaining In-Context Measurements of Cellular Network Performance. In *Proc. ACM IMC*, 2012.

[43] U. Goel, M. Steiner, W. Na, M. P. Wittie, M. Flack, and S. Ludin. Are 3rd parties slowing down the mobile web? In *Proc. S3 Workshop*, 2016.

[44] E. Griffin. Fast dormancy - save your battery from 3g drainage. https://www.youtube.com/watch?v=O8L50sCY7CI, 2012.

[45] GSMA. Fast dormancy best practices. http://www.gsma.com/newsroom/wp-content/uploads/2013/08/TS18v1-0.pdf, 2011.

[46] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proc. ACM MobiSys*, 2013.

[47] S. Ha, S. Sen, C. Joe-Wong, Y. Im, and M. Chiang. Tube: Time-dependent pricing for mobile data. In *Proc. ACM SIGCOMM*, 2012.

[48] B. Han, S. Hao, and F. Qian. Metapush: Cellular-friendly server push for http/2. In *AllThingsCellular*, 2015.

[49] B. Han, P. Hui, V. A. Kumar, M. V. Marathe, G. Pei, and A. Srinivasan. Cellular Traffic Offloading Through Opportunistic Communications: A Case Study. In *ACM CHANTS*, 2010.

[50] Y. Z. Hao Liu, Yaoxue Zhang. TailTheft: leveraging the wasted time for saving energy in cellular communications. 2011.

[51] B. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proc. ACM MobiSys*, 2012.

[52] Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[53] R. Holly. Checking out Doze and App standby on the Android M Developer Preview. http://www.androidcentral.com/checking-out-doze-android-m-developer-preview.

[54] Http/2. https://http2.github.io/.

[55] Http/2 faq. https://http2.github.io/faq.

[56] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proc. ACM MobiSys*, 2012.

[57] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An In-Depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *ACM SIGCOMM Computer Communication Review*, 2013.

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

[59] A. L. Iacono and C. Rose. Infostations: New perspectives on wireless data networks. In *Next Generation Wireless Networks*. 2002.

[60] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *IMC*, 2011.

[61] N. Ingraham. Apple's app store has passed 100 billion app downloads. `http://www.theverge.com/2015/6/8/8739611/apple-wwdc-2015-stats-update`.

[62] S. Isaacman, R. Becker, R. Cáceres, M. Martonosi, J. Rowland, A. Varshavsky, and W. Willinger. Human Mobility Modeling at Metropolitan Scales, 2012.

[63] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Perfomance and Power Characteristics of 4G LTE Networks. In *Proc. ACM MobiSys*, 2012.

[64] A. E. J. Tadrous. On optimal proactive caching for mobile networks with demand uncertainties. *IEEE/ACM Transactions on Networking*, 2016.

[65] J. Wigard, T. Kolding, L. Dalsgaard, and C. Coletti. On the User Performance of LTE UE Power Savings Schemes with Discontinuous Reception in LTE. 2009.

[66] J. Jeong, M. Leconte, and A. Proutière. Mobility prediction using non-parametric bayesian model. *CoRR*, 2015.

[67] Y. Jin, N. Duffield, A. Gerber, P. Haffner, W.-L. Hsu, G. Jacobson, S. Sen, S. Venkataraman, and Z.-L. Zhang. Characterizing Data Usage Patterns in a Large Cellular Network. In *CellNet*, 2014.

[68] H. E. G. J.Tadrous, A. Eryilmaz. Proactive resource allocation: harnessing the diversity and multicast gains. *IEEE Transactions on Information Theory*, 2013.

[69] A. J. Khan, K. Jayarajah, D. Han, A. Misra, R. Balan, and S. Seshan. Cameo: A middleware for mobile advertisement delivery. In *Proc. ACM MobiSys*, 2013.

[70] L. Zhou, H. Xu, H. Tian, Y. Gao, L. Du, and L. Chen. Performance Analysis of Power Saving Mechanism with Adjustable DRX Cycles in 3GPP LTE. 2008.

[71] H. A. Lagar-Cavilla, K. Joshi, A. Varshavsky, J. Bickford, and D. Parra. Traffic backfilling: subsidizing lunch for delay-tolerant applications in UMTS networks. 2011.

[72] M. Laner, P. Svoboda, S. Schwarz, and M. Rupp. Users in Cells: a Data Traffic Analysis. In *WCNC*, 2012.

[73] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong. Mobile Data Offloading: How Much Can WiFi Deliver? In *Proc. CoNEXT*, 2012.

[74] Leila Modarres. At&t aro makes apps even better. `http://blogs.keynote.com/mobility/2012/08/`, 2012.

[75] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. Webprophet: Automating performance prediction for web services. In *NSDI*, 2010.

[76] M. Martins, J. Cappos, and R. Fonseca. Selectively Taming Background Android Apps to Improve Battery Lifetime. In *Proc. Usenix ATC*, 2015.

[77] S. Mohan, R. Kapoor, and B. Mohanty. Latency in hspa data networks. Technical report, Qualcomm, 2011.

[78] Y. Moon, D. Kim, Y. Go, Y. Kim, Y. Yi, S. Chong, , and K. Park. Practicalizing Delay-Tolerant Mobile Apps with Cedos. In *Proc. ACM MobiSys*, 2015.

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

[80] A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, and G. Challen. Phonelab: A large programmable smartphone testbed. In *Proceedings of First International Workshop on Sensing and Big Data Mining*, pages 1–6. ACM, 2013.

[81] S. Narayanan, Y. Nam, A. Sivakumar, B. Chandrasekaran, B. Maggs, and S. Rao. Reducing latency through page-aware management of web objects by content delivery networks. In *Proc. ACM SIGMETRICS*, 2016.

[82] J. Nejati and A. Balasubramanian. An in-depth study of mobile browser performance. In *WWW*, 2016.

[83] R. Netravali, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *Proc. NSDI*, 2016.

[84] N. S. Networks. Understanding smartphone behavior in the network, 2011.

[85] Nghttp2: HTTP/2 C library and tools. `https://nghttp2.org/`.

[86] 7 tips for faster http/2 performance. `https://www.nginx.com/blog/7-tips-for-faster-http2-performance/`.

[87] A. Nika, Y. Zhu, N. Ding, A. Jindal, Y. C. Hu, X. Zhou, B. Y. Zhao, and H. Zheng. Energy and performance of smartphone radio bundling in outdoor environments. In *WWW*, 2015.

[88] A. Nikravesh, D. R. Choffnes, E. Katz-Bassett, Z. M. Mao, and M. Welsh. Mobile network performance from user devices: A longitudinal, multidimensional analysis. In *Passive and Active Measurement Conference*, 2014.

[89] NPR One app FAQ. `https://www.fuzeqna.com/npr/ext/kb637-npr-one-app-faq`, 2015.

[90] NU JamLogger: A Study of User Activity and System Performance on Mobile Architectures. `http://www.ece.northwestern.edu/microarchitecture/jamlogger/`, 2009.

[91] Office for National Statistics. Methodology of the monthly index of services: Annex b: the holt-winters forecasting method. http://www.ons.gov.uk/ons/guide-method/user-guidance/index-of-services/index-of-services-annex-b--the-holt-winters-forecasting-method.pdf.

[92] J. Osofsky. More ways to drive traffic to news and publishing sites. https://www.facebook.com/notes/facebook-media/more-ways-to-drive-traffic-to-news-and-publishing-sites/585971984771628, 2013.

[93] C. P. Going into 2016, battery life is still the number one concern with our readers (poll results). http://www.phonearena.com/news/Going-into-2016-battery-life-is-still-the-number-one-concern-with-our-readers-poll-results_id76978, 2015.

[94] P. K. Athivarapu, R. Bhagwan, S. Guha, V. Navda, R. Ramjee, D. Arora, V. N. Padmanabhan, and G. Varghese. RadioJockey: Mining Program Execution to Optimize Cellular Radio Usage. 2012.

[95] T. C. Projects. Spdy: An experimental protocol for a faster web. Technical report.

[96] E. Protalinski. Facebook passes 1.23 billion monthly active users, 945 million mobile users, and 757 million daily users. http://thenextweb.com/facebook/2014/01/29/facebook-passes-1-23-billion-monthly-active-users-945-million-mobile-users-757-million-daily-users, 2014.

[97] F. Qian, S. Sen, and O. Spatscheck. Characterizing resource usage for mobile web browsing. In *Proc. ACM MobiSys*, 2014.

[98] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization. In *Proceedings of the 21st international conference on World Wide Web*, pages 51–60, 2012.

[99] QxDM Professional Proven Diagnostic Tool for Evaluating Handset and Network Performance. http://www.qualcomm.com/media/documents/files/qxdm-professional-qualcomm-extensible-diagnostic-monitor.pdf, 2012.

[100] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Procrastinator: Pacing mobile apps' usage of the network. In *Proc. ACM MobiSys*, 2014.

[101] A. Reda, B. Noble, and Y. Haile. Distributing private data in challenged network environments. In *World Wide Web Conference*, 2010.

[102] S. Rosen, J. Erman, V. Gopalakrishnan, Z. M. Mao, and J. Pang. Cellshift: A system to efficiently time-shift data on the cellular network. Technical Report CSE-TR-588-15, Department of Computer Science and Engineering, University of Michigan, 2015.

[103] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering fine-grained rrc state dynamics and performance impacts in cellular networks. In *Proc. ACM MobiCom*, 2014.

[104] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen. Revisiting network energy efficiency of mobile apps: Performance in the wild. In *Proc. ACM IMC*, 2015.

[105] S. Deng, and H. Balakrishnan. Traffic-Aware Techniques to Reduce 3G/LTE Wireless Energy Consumption. In *Proc. ACM CoNEXT*, 2012.

[106] S. Souders. Making a mobile connection. `http://www.stevesouders.com/blog/2011/09/21/making-a-mobile-connection/`, 2011.

[107] Samsung. Galaxy S4 Standard Battery. `https://www.samsung.com/us/mobile/cell-phones-accessories/EB-B600BUBESTA`.

[108] A. A. Sani, Z. Tan, P. Washington, M. Chen, S. Agarwal, L. Zhong, and M. Zhang. The Wireless Data Drain of Users, Apps, & Platforms. *ACM SIGMOBILE Mobile Computing and Communications Review*, 17(4), 2013.

[109] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 2001.

[110] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 2009.

[111] Scrapbook: Addons for Firefox. `https://addons.mozilla.org/en-US/firefox/addon/scrapbook/`.

[112] S. Sen, C. Joe-Wong, S. Ha, J. Bawa, and M. Chiang. When the Price is Right: Enabling Time-dependent Pricing of Broadband Data. In *Proc. SIGCHI*, 2013.

[113] S. Sen, C. Joe-Wong, S. Ha, and M. Chiang. A Survey of Smart Data Pricing: Past Proposals, Current Plans, and Future Trends. *ACM Comput. Surv.*, 46, 2013.

[114] C. Shi, K. Joshi, R. K. Panta, M. H. Ammar, and E. W. Zegura. Coast: Collaborative application-aware scheduling of last-mile cellular traffic. In *Proc. ACM MobiSys*, 2014.

[115] I. Singh, S. V. Krishnamurthy, H. V. Madhyastha, and I. Neamtiu. ZapDroid: Managing Infrequently Used Applications on Smartphones. In *Proc. UbiComp*, 2015.

[116] S. Singh, H. Madhyastha, K. S.V., and R. Govindan. Flexiweb: Network-aware compaction for accelerating mobile web transfers. In *Proc. ACM MobiCom*, 2015.

[117] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proc. ACM CoNEXT*, 2014.

[118] J. Spencer, M. Sudan, and K. Xu. Queuing with future information. *The Annals of Applied Probability*, 2014.

[119] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. Measuring and mitigating web performance bottlenecks in broadband access networks. In *Proc. ACM IMC*, 2013.

[120] J. Tadrous, A. Eryilmaz, H. El Gamal, J. Tadrous, A. Eryilmaz, and H. El Gamal. Joint smart pricing and proactive content caching for mobile services. *IEEE/ACM Trans. Netw.*, 2016.

[121] J. Tadrous, A. Eryilmaz, and H. E. Gamal. Proactive content download and user demand shaping for data networks. *IEEE/ACM Trans. Netw.*, 2015.

[122] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who Killed my Battery?: Analyzing Mobile Browser Energy Consumption. In *Proceedings of the 21st international conference on World Wide Web*, 2012.

[123] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. Rilanalyzer: A comprehensive 3g monitor on your phone. 2013.

[124] M. Varvello, K. Schomp, D. Naylor, J. Blackburn, A. Finamore, and K. Papagiannaki. Is the web http/2 yet? In *Passive and Active Measurement Conference*, 2016.

[125] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, MCS '12, 2012.

[126] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *Proc. NSDI*, 2013.

[127] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is spdy? In *Proc. NSDI*, 2014.

[128] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with shandian. In *NSDI*, 2016.

[129] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *Proc. of Workshop on Mobile Computing Systems and Applications*, 2011.

[130] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. ACM SIGCOMM*, 2011.

[131] H. Xiong, D. Zhang, D. Zhang, and V. Gauthier. Predicting Mobile Phone User Locations by Exploiting Collective Behavioral Patterns. In *IEEE UIC/ATC*, 2012.

[132] K. Xu. Necessity of future information in admission control. *CoRR*, 2015.

[133] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying Diverse Usage Behaviors of Smartphone Apps. In *Proc. ACM IMC*, 2011.

[134] Take your channels with you on the new YouTube app. `http://youtube-global.blogspot.com/2012/06/take-your-channels-with-you-on-new.html`, 2012.

[135] Y. Zaki, J. Chen, T. Pötsch, T. Ahmad, and L. Subramanian. Dissecting web latency in ghana. In *Proc. ACM IMC*, 2014.

[136] K. Zarifis, M. Holland, M. Jain, E. Katz-Bassett, and R. Govindan. Modeling http/2 speed from http/1 traces. In *Passive and Active Measurement Conference*, 2016.