Reducing 3G Energy Consumption on Mobile Devices

by

Shuo Deng

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

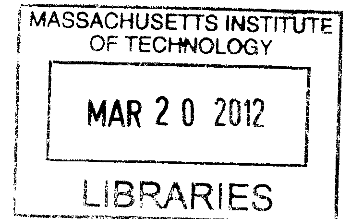
Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

ARCHIVES



© Massachusetts Institute of Technology 2012. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
February 3, 2012

Certified by ....
Hari Balakrishnan
Professor
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Reducing 3G Energy Consumption on Mobile Devices

by

Shuo Deng

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The 3G wireless interface is a significant contributor to battery drain on mobile devices. This paper describes the design, implementation, and experimental evaluation of methods to reduce the energy consumption of the 3G radio interface. The idea is to put the radio in its "Low-power idle" state when no application is likely to need the network for some duration of time in the future. We present two techniques, one to determine when to change the radio's state from "Active" to "Low-power idle", and the other to change the radio's state from "Low-power idle" to "Active". The technique for switching to Low-power idle mode is well-suited for the emerging "fast dormancy" [3, 4] primitive that will soon be common on smartphones. We demonstrate using an implementation and a trace-driven evaluation based on the measurement and trace collected from HTC G1 and Samsung Nexus S phones over various combinations of seven different background applications that our methods reduce the energy consumption of the 3G interface by 36% on average compared to the currently deployed scheme on the T-mobile network. In addition, if applications are able to tolerate a delay of a few seconds when they initiate a session, our methods reduce energy consumption by 52% on average, with a mean increase in delay of 6.46 seconds.

Thesis Supervisor: Hari Balakrishnan
Title: Professor

Acknowledgments

This thesis would not have been possible without the support of many people. Hence, I would like to take this opportunity to express my sincere gratitude to all of them.

First, I would like to thank my adviser, Professor Hari Balakrishnan, for giving me the opportunity to start my graduate study at MIT and work in his group, and for his guidance, attention and support throughout this project. After my first year of graduate study, he encouraged me to work as an intern in Microsoft India. It was a great experience not only for research, but also for exploring the world. There, I worked closely with my manager Venkat Padmanabhan and my mentor Vishnu Navda, who introduced me to the field of mobile networks. After coming back from the internship, Hari was very supportive on me continuing to work in this field and advised me to explore probability theory and machine learning techniques to solve the problem. I also sincerely appreciate his help in correcting the draft of this thesis carefully.

I would also like to thank my office-mates and colleagues in NMS group. Katrina LaCurts and Keith Winstein gave me very detailed comments on the writing of this thesis and also a lot of valuable suggestions at early stages of this project. Lenin Ravindranath and Jonathan Perry also spent a lot of time reading the draft of this thesis and pointed out improvement to make this thesis much stronger.

I would also thank my friends outside the lab. I am lucky to have Yan Zhao as my roommate, who listens to my endless complaints and always show me the bright side of life to cheer me up. Yuan Mei, a senior student in CSAIL used to be my roommate; she taught me a great deal about how to get used to life in the U.S. and how to be a graduate student at MIT. Yu Xin, graduate student in CSAIL working in the machine learning group, discussed a lot with me about machine learning topics that could be applied to my research.

My parents have supported me unconditionally all my life. Their life stories taught me to always hold a strong will toward life and keep inner peace. I thank my grandfather and grandmother a lot for being supportive of my study abroad and taking good care of my mother. Special thanks to my boyfriend, Xinming Chen, who was on the other side of the earth when I was working on this project, but who always ready to help me with

technical and non-technical problems, and who was willing to give up cozy life in China to end our long-distance relationship and start a new page of life together in the state of Massachusetts.

Contents

1 Introduction **15**

2 Background **21**

3 Related Work **23**

3.1 3G Energy Mitigation Strategies 23

3.1.1 Inactivity timer reconfiguration 23

3.1.2 Tail cutting 24

3.1.3 Tail sharing 24

3.2 3G Resource Usage Profiling 25

3.3 WiFi Power-saving Algorithms 26

4 Design **27**

5 MakeIdle Algorithm **31**

5.1 A Simple Example 32

5.2 Optimal Decision From Offline Trace Analysis 34

5.3 Online Prediction 36

6 MakeActive Algorithm **39**

6.1 Fixed Delay Bound 41

6.2 Learning Algorithm 41

6.2.1 Bank of Experts 42

6.2.2 Loss Function 43

7 **Implementation** **45**

8 **Evaluation** **47**

8.1 MakeIdle Evaluation 50

8.2 MakeActive Evaluation 55

8.3 CPU and Energy overhead of running algorithms 60

9 **Conclusion and Future Work** **61**

List of Figures

1-1 Energy consumed by the 3G interface. "Data" corresponds to a data transmission; "DCH Timer" and "FACH Timer" are each the energy consumed with the radio in the idle states specified by the two timers, and "State Switch" is the energy consumed in switching states. These timers and state switches are described in §2. 17

2-1 3GPP Radio Resource Control (RRC) State Machine. 22

4-1 System design. 28

5-1 The simplified model for 3G energy consumption. 33

5-2 Reducing $t_1 + t_2$ to 4.5 seconds brings more state switches while saving little energy. Further reducing $t_1 + t_2$ to a smaller value that brings the same number of state switches can save more energy. 34

5-3 When the gap between traffic bursts is small, reducing $t_1 + t_2$ to a small value may consume more energy, while *also* adding delays for each burst. . 35

6-1 "Shift" traffic to reduce number of state switches. 40

7-1 Socket-layer modifications. 46

8-1 The measured power consumption of the different 3GPP RRC states for the HTC (top) and Nexus S (bottom) smartphones. The screenshot from the monitor shows the current drawn; the voltage is 3.7 volts. The average power consumed while "Active" (Cell_DCH) is 1028 mW for HTC and 1804 mW for Nexus S. In "High-power idle" (Cell_FACH), the power consumed is 445 mW for HTC and 450 mW for Nexus S. "Low-power idle" (IDLE) is 0 watts for both devices. In this figure the power level is non-zero because of the CPU and LED screen power consumption. We did not measure the power consumed in Cell_PCH state because our measured network/phones do not support that state. The GSM specification suggests that the Cell_PCH state is just slightly higher than IDLE compared to Cell_FACH. 48

8-2 Energy saved for different applications. 50

8-3 CDFs of the percentage of energy saved over 27 traces, running various combinations of the seven applications. 51

8-4 The value of t_{wait} that maximizes expected energy saving is close to the value maximizes the actual energy saving for a particular application (Ebuddy). Similar results hold for all applications. 53

8-5 An example where fast dormancy should not be triggered when running applications with continuous and intensive data transmission. The bottom figure shows actually there is no proper t_{wait} that can reduce energy consumption. 54

8-6 The number of state switches, which is proportional to the signaling overhead, for the individual applications. Each bar is for a different method that determines when the radio should be turned to the Idle mode. 55

8-7 CDFs of the number of state switches across the 27 combined-application traces for the different methods to put the 3G radio in idle (off) mode. . . . 56

8-8 Energy saved for different applications. 57

8-9 CDFs of energy savings using the 4.5-second tail, MakeIdle alone, MakeIdle with MakeActive using learning, and MakeIdle with MakeActive using a fixed delay bound. 58

8-10 CDF of traffic burst delays. 59
8-11 Delay value changes as the learning proceeds. 59

List of Tables

8.1 Median and mean amounts of energy saved by the different methods relative to the 20-second tail scheme. 51

8.2 Median and mean of number of state switches over 27 traces. 56

8.3 Median and mean energy savings over the 20-second tail. 57

Chapter 1

Introduction

20% of the 5 billion active mobile phones today have "broadband" data service enabled on them, and this fraction is growing rapidly. Smartphones and tablets with wide-area 3G cellular connectivity have become a significant, and in many cases, dominant, mode of network access. Improvements in the quality of such network connectivity suggest that mobile Internet access will soon overtake desktop access, especially with the continued proliferation of 3G networks and the emergence of LTE and 4G.

Wide-area cellular wireless protocols need to balance a number of conflicting goals: high throughput, low latency, low signaling overhead (signaling is caused by mobility and changes in the mobile device's state), and mobile energy consumption. The 3GPP and 3GPP2 standards (used in 3G and LTE) provide some mechanisms for the cellular network operator and the mobile device to optimize these metrics[21, 2], but to date, deployed methods to minimize energy consumption have left a lot to be desired.

The 3G radio consumes significant amounts of energy; on the iPhone 4, for example, the stated talk time is "up to 7 hours on 3G" (i.e., when the 3G radio is on and in "typical" use) and "up to 14 hours on 2G".[1] On the Samsung Nexus S, the equivalent numbers are "up to 6 hours 40 minutes on 3G" and "up to 14 hours on 2G".[2] On these platforms, as well as other such smartphones, using the 3G radio halves the lifetime compared to using 2G; in addition, 3G roughly halves the lifetime compared to WiFi when used for Internet access.

[1] http://www.apple.com/iphone/specs.html

[2] http://www.gsmarena.com/samsung_google_nexus_s-3620.php

That 3G is a battery hog is well-known to most users anecdotally and from experience, and much advice on the web and on blogs is available on how to extend the battery life of your mobile device.[3] Unfortunately, essentially all such advice says to "disable your 3G data radio" and "change your fetch data settings to reduce network usage". Such advice largely defeats the purpose of having an "always on" broadband-speed wireless device, but appears to be the best one can do in current deployments.

Previous research [5], as well as our measurements, show that in general about 60% of the energy consumed by the 3G interface is "wasted energy", spent when the radio is not transmitting or receiving data. We show the measured values of 3G energy consumption for multiple Android applications in Figure 1-1. This bar graph shows the percentage of energy consumed by different 3G states. For most of these applications (which are all background applications operating without user input, except for Facebook), less than 30% of the energy consumed was due to the actual transmission or reception of data.

In principle, one might imagine that simply turning the radio off or switching it to a low-power idle state is all it takes to reduce energy consumption. This approach does not work for three challenging reasons. First, switching between the active and the different idle states takes time (a few seconds), so it should be done only if there is good reason to believe that making the transition is useful for a reasonable duration of time in the future. Second, switching states consumes energy, which means that if done without care, overall energy consumption may increase compared to not doing anything sophisticated. Third, there switching incurs signaling overhead on the wireless network, which means that it should be done only if the benefits are substantial relative to the cost on the network.

This thesis tackles these challenges and develops a solution to reduce 3G energy consumption without adversely affecting application performance or introducing a significant amount of extra signaling overhead on the network. Unlike currently deployed methods that simply switch between radio states after fixed time intervals—an approach known to be sub-optimal [20, 5, 10, 15]), our approach is to observe network traffic activity from/to the mobile device to switch between the different radio states based on the workload.

The key insight in our work is that by observing network traffic activity, a *control*

[3]http://www.intomobile.com/2008/07/23/extend-your-iphone-3gs-battery-life/

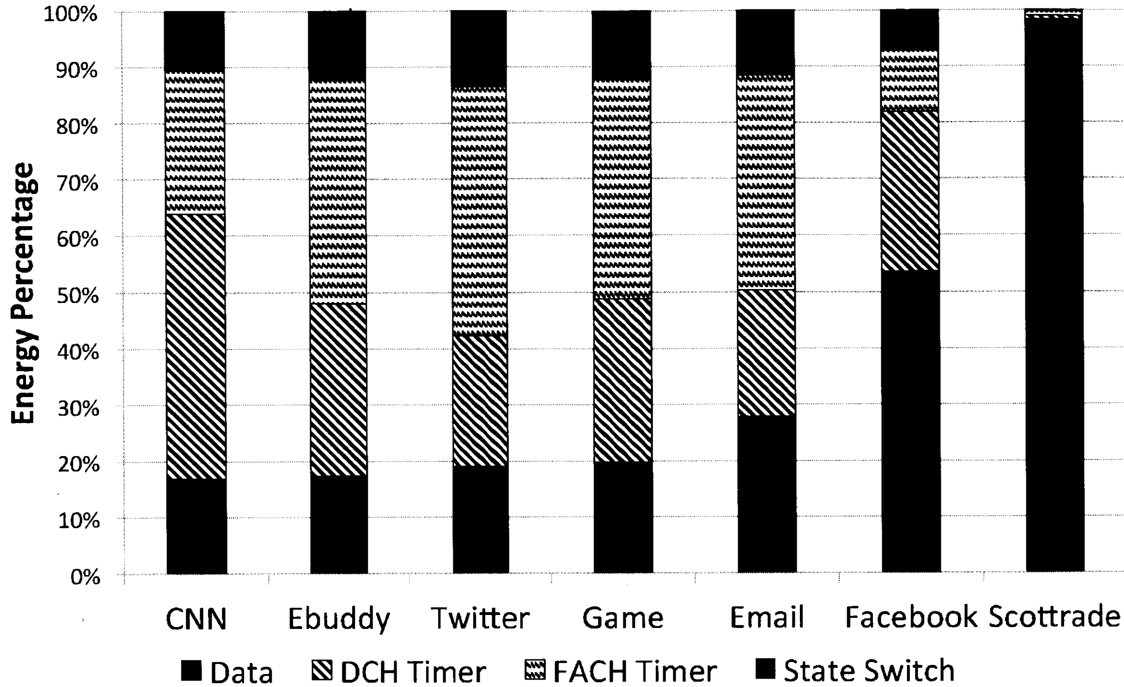


Figure 1-1: Energy consumed by the 3G interface. "Data" corresponds to a data transmission; "DCH Timer" and "FACH Timer" are each the energy consumed with the radio in the idle states specified by the two timers, and "State Switch" is the energy consumed in switching states. These timers and state switches are described in §2.

module on the mobile device can adapt the 3G radio state transitions to the workload. Our approach is to use statistical machine learning techniques to predict network activity based on past observations and make transitions that are suggested by the statistical models. We show that our approach is well-suited to the emerging *fast dormancy* mechanism [3, 4] that allows a radio to rapidly move between the "Active" and "Low-power idle" states and vice versa. Our goal is to support background applications on mobile devices, such as the applications measured above.

This thesis makes the following contributions:

1. A traffic-aware design to control the state transitions of a 3G radio taking energy consumption, latency, and signaling overhead into consideration. The design incorporates two algorithms:
   (a) MakeIdle, which uses aggregate traffic activity to build a conditional probability distribution of activity that predicts the end of active sessions.

   (b) MakeActive, which applies machine learning to delay the start of a new session by a few seconds to allow multiple sessions to all become active at the same time and therefore save energy.

2. An Android implementation done by replacing a part of the relevant Java library at the socket layer to enable devices running unmodified applications to increase their battery life while using the 3G service. To our knowledge, ours is the first implementation of such an energy-saving mechanism on smartphones.

3. Experimental results from our implementation and using trace analysis (on HTC G1 and Samsung Nexus S smartphones in T-mobile's 3G network), showing that our methods save 36% more energy of the 3G interface compared to the currently deployed scheme on the T-mobile network without incurring additional session delays (using MakeIdle alone). These results are from various combinations of seven different "background" applications, including email, news updates, instant messaging updates, etc. The background applications for which our system is well-suited are generally tolerant of some delay in session initiation; we find that by combining MakeIdle and MakeActive, our system consumes 52% less 3G energy on average for this mix of applications compared to the status quo, while increasing the average delay by (only) 6.46 seconds. Of course, these experimental results depend on the application mix. If the user mostly uses foreground and interactive applications, MakeIdle is applicable, but probably not MakeActive. Moreover, some background applications may send/receive data often (e.g., every second or two); for these, no scheme can provide significant gains (in our evaluation, one application has such a property and our scheme correctly infers that it cannot be optimized). Of the seven applications we examined, we found gains for five of them, and present the extent of the gains for various combinations: with delays allowed, the mean saving is 52% and the median is 59%.

The rest of the thesis is organized as follows: Chapter 2 explains the background and the basics of 3G network energy modeling. Chapter 3 discusses previous works on 3G network energy management and other wireless network energy saving techniques. Chapter 4 shows the design of our system. Chapter 5 and Chapter 6 explains our two algorithm,

MakeIdle and MakeActive separately. Chapter 7 and Chapter 8 focuses on the implementation, simulation and evaluation of our system. Chapter 9 concludes the thesis and discusses the possible directions of future work.

Chapter 2

Background

The Radio Resource Control (RRC) protocol, which is part of the 3GPP standard, incorporates the state machine for energy management shown in Figure 2-1.

In 3GPP networks, the base station maintains two inactivity timers, t_1 and t_2, for each mobile device. For a device maintaining a dedicated channel in the "Active" (Cell_DCH) state with the base station, if the base station sees no data activity to or from the device for t_1 seconds, it will switch the device from dedicated channel to a shared low-speed channel, and the device is now turned to the "High-power idle" (Cell_FACH) state. This state consumes less power than "Active", but still consumes a non-negligible amount of power. If there is no further data activity between the device and base station for another t_2 seconds, the base station will turn the device to either the Cell_PCH or IDLE state (T-mobile turns the device to IDLE). We refer to the Cell_PCH and IDLE states together as "Low-power idle", because the device consumes essentially no power in either state. Figure 2-1 summarizes the state transitions; not shown is the "OFF" state in which the radio is turned off.

The inactivity timers t_1 and t_2 are useful because a state transition from Idle to "Active" (Cell_DCH) incurs significant delays (≈ 3.62 seconds on the T-mobile network in our measurements). Each state transition also consumes energy on the device and incurs signaling overhead for the base station to allocate a dedicated channel to the device. The inactivity timers also prevent the base station from frequently releasing and re-allocating channels to devices which causes per-packet delay for the device to be high. Another popular 3G

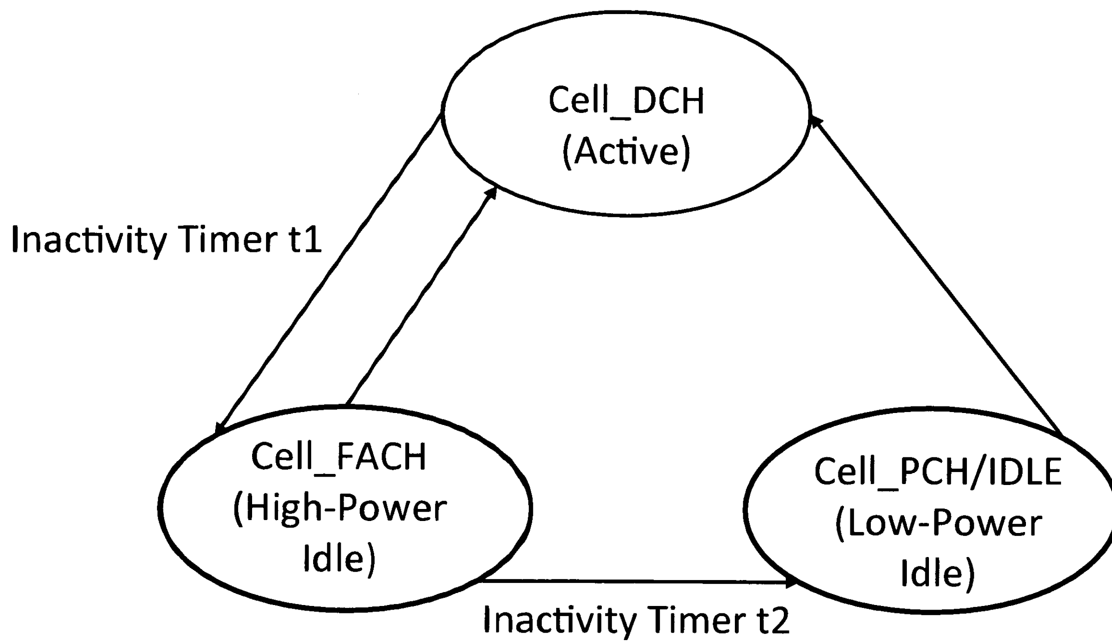


Figure 2-1: 3GPP Radio Resource Control (RRC) State Machine.

standard is 3GPP2 [2]. Although 3GPP2 networks use different techniques, from the perspective of energy consumption, they are essentially identical to 3GPP [20]; like 3GPP, 3GPP2 networks also have different power levels for different states on the device side, and use similar inactivity timers for state transitions. For concreteness, in this thesis, we focus on 3GPP networks.

Chapter 3

Related Work

The problem of understanding how energy is consumed by wireless network interfaces and strategies for reducing energy consumption have been studied for over a decade. We divide related work into measurement studies of 3G energy consumption and approaches to reduce that energy, 3G usage profiling, and WiFi power saving methods.

3.1 3G Energy Mitigation Strategies

Past work aimed at eliminating the tail energy (energy consumed in High-Power Idle and Low-Power Idle) falls into three categories: inactivity timer reconfiguration, tail cutting, and tail sharing.

3.1.1 Inactivity timer reconfiguration

Lee et al. [10] developed analytic models for energy consumption in WCDMA and CDMA2000 and showed that the inactivity timer should be dynamically configured based on user behaviors and battery restriction. Falaki et al. [6] proposed an empirical method by plotting the CDF of packet inter-arrival times for traces collected on smartphones communicating over 3G radio. They found that 95% of the packet inter-arrival time values are smaller than 4.5 seconds, and proposed setting the inactivity timer to a fixed value, $t_1 + t_2 = 4.5$ seconds. This value is much shorter than the current 20-second tail. It consumes less energy,

but does incur higher signaling overhead. However, the channel re-allocation will happen so often that the mobile device wastes a lot of energy to turn the radio from Low-power idle to active state. We refer this scheme as the *4.5-second tail* in this thesis.

Note that the 4.5-second tail scheme can be also implemented without base station side reconfiguration. The mobile device can also monitor its radio activity and initiate fast dormancy when the radio has been inactive for 4.5 seconds. In this thesis, we will compare our system, which adapts to network activity, against this scheme.

3.1.2 Tail cutting

Qian et al. [15] gave an algorithm, *TOP*, to help the device decide when to trigger fast dormancy based on the information provided by applications running on the device. Their algorithm requires the application to predict when the next packet will come and report to the OS. This approach requires modifications to the applications, and it is not clear how each application should make these predictions. Our work requires no modification to the application code and does not require the application to predict the traffic. All the prediction is done using a statistical learning model that works with unmodified applications by intercepting on socket calls.

3.1.3 Tail sharing

Balasubramanian et al. [5] propose an application-layer protocol, *TailEnder*, which aims to coalesce separate data transfers by delaying some of them. For delay-tolerant applications such as email, TailEnder allows applications to set a deadline for the incoming transfer requests; they suggest and evaluate a relatively long delay of 10 minutes for such applications. For applications that can benefit from prefetching, TailEnder prefetches 10 web documents for each user query. TailEnder handles specific classes of applications, but cannot handle streaming traffic, for example, music or video streaming, and does not seem well-suited for instant messaging applications that might be able to tolerate a delay of 0-20 seconds, but not 10 minutes. Liu et al. [11] proposed TailTheft, a traffic queuing and scheduling mechanism to batch traffic among different applications and share the tail

energy among them. The idea is to setup a timeout value for those delay-tolerant transfers, and transfers data when timeouts or other delay-sensitive transfer has already triggered the radio to active mode. The timer in their approach is on the order of 200 to 2000 seconds. Our MakeActive algorithm does not set a fixed timer; instead, we learn the proper timeout value automatically, and the delay is bounded by 20 seconds; and for most of the time the delay is much less than 20 seconds. Another tail sharing idea is prefetching. Instead of delaying traffic transfer, there are some applications that transfers predictable contents and can be prefetched before user actually send request for the data. Such applications include Search, which can prefech the first several searching results; Youtube, which split a video file into several pieces and transfer one piece at a time. Qian et al. [17] proposed a prefetching algorithm for Youtube that erases the tail between transfers of video pieces.

3.2 3G Resource Usage Profiling

Another related research topic is 3G network resource usage profiling on mobile devices. Qian et al. [16] designed a profiling system that collects TCP traces on mobile devices and analyzes them on the server. In their system, they designed an algorithm to infer RRC state machine states from the traces collected for each mobile device. Then they analyzed the energy consumption of particular applications using the traces, the RRC inference, and the energy model. Their analysis shows that some popular mobile applications do have traffic patterns that are not energy-efficient, due to low bit-rate transmission, inefficient prefetching, and aggressive refreshes. This work shows that mobile applications need to be carefully designed to take 3G network utilization/energy consumption into account. But this task can be hard for application programmers, who usually lack of knowledge of the details of 3G networks.

Our approach requires no coordination from applications, which can benefit legacy applications, and free developers from having to worry about the details of the cellular network protocols. Also, our approach not only focuses on a per-application based solution, but also tries to optimize the energy consumption when multiple applications are running together.

Xu et al. [19] analyze usage patterns of smartphone applications using the data collected from a carrier in US. The data shows that smartphone applications and web browsing generate a large portion of the network traffic. Our approach provides considerable energy savings for those applications. This work also shows that "streaming data is only accountable for a small fraction of the total network access time of all smartphone apps". In our work, we show that there is not much room for energy saving for streaming traffic patterns.

3.3 WiFi Power-saving Algorithms

Much prior work has focused on WiFi power-saving algorithms [8, 9, 18]. The problem in WiFi networks is qualitatively different from 3G; in WiFi, the time and energy consumed to transition between states is negligible; what is important is to dynamically determine the best sleep duration when the WiFi radio is off. In this state, no packets can be delivered, but the access point will be able to buffer them; the problem is finding the longest sleep time that ensures that no packets are delayed (say, by a specified maximum delay). In the 3G context, changing the state of the radio consumes time, energy, and network signaling overhead, but there is no risk of receiving packets with excessive delay because the base station is able to notify a mobile device that packets are waiting for it even if the device is in the Low-power idle (PCH or IDLE) state. Thus, we cannot simply apply existing WiFi Power-saving algorithms to 3G networks.

Chapter 4

Design

The key insight in our approach to reduce 3G energy consumption is that by observing and adapting to network activity, a *control module* can predict when to put the radio into its Low-power idle state, and when to move from the idle to active state. These state transitions take time (multiple seconds) and also add signaling overhead because each transition is accompanied by a few messages between the device and the base station. Hence, the intuition in our approach is to predict the occurrence of *bursts* of network activity, so that the control module can put the radio into the idle mode when it believes a burst has ended, which means there will not be any more traffic in the future for a relatively long period of time, and conversely, put the radio in active mode when another burst is starting.

To achieve the prediction, our approach needs to observe network activity. To make our approach work with existing applications, we should not require any change to the application code, so we cannot get notified about network activities from the application directly. To address this issue, we considered two approaches to monitor the network activity. The first is to make the control module observe all the packets going in and out of the device. The second is to intercept all socket calls such as "connect", "close", "read" and "write". In our design, we use the second approach for the following reasons: first, monitoring socket call brings much less overhead than monitoring each packets, since each socket call generates tens of packets. Second, our goal is to monitor the starts/ends of bursts of traffic, and socket calls such as "connect" and "close" can work as good indication. Also, each "read" or "write" call corresponds to several packet transmissions, which can give us enough in-

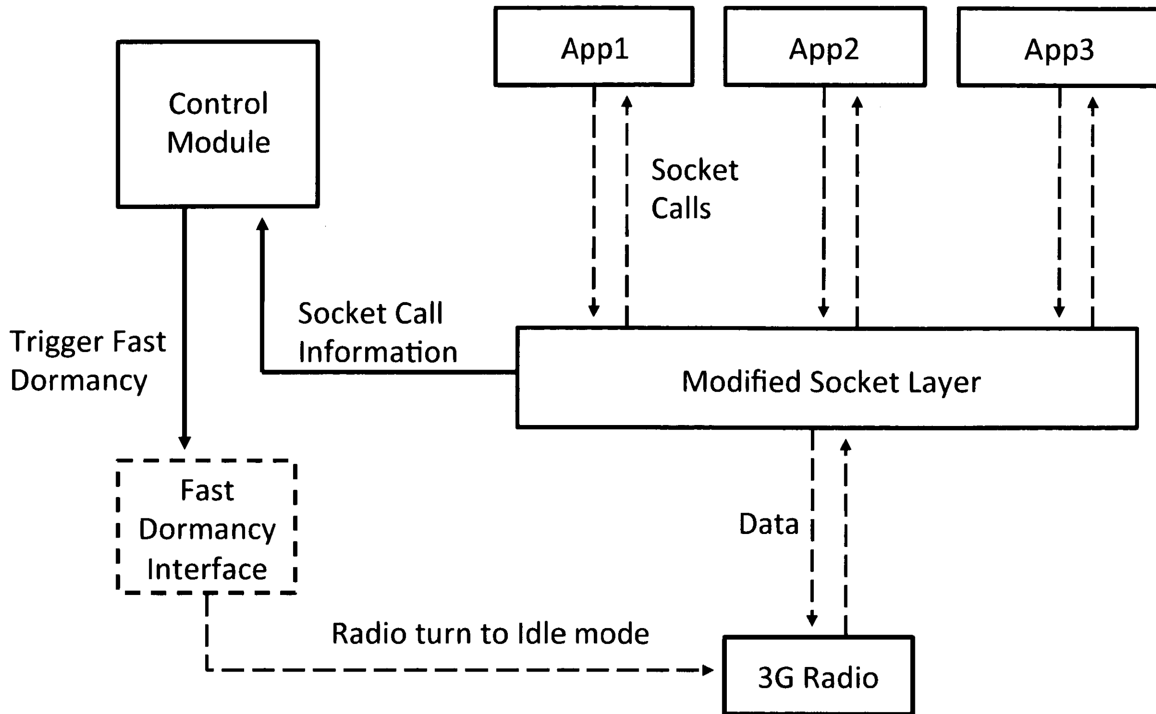


Figure 4-1: System design.

formation on when packets are sent/received. Thus there is no need to look into the packet inside a burst, so monitoring each packet is not necessary.

Our system involves two software modules: one that modifies the library used by applications to communicate with the socket layer, and the other that implements the control module, as shown in Figure 4-1. The first module informs the control module of all socket calls; in response, the control module configures the state of the radio. The fast dormancy interface is presented as a dashed module because our system uses it if it is available. When fast dormancy becomes available in the hardware, our implementation will be able to use the facility.

The control module implements two different algorithms. The first algorithm, MakeIdle, runs when the radio is in the active state (Cell_DCH) and determines when the radio should be put into the Low-power idle (IDLE or Cell_PCH) state.

The second algorithm, MakeActive, runs when the radio is in the Low-power idle (Cell_PCH or Cell_IDLE) state. In this state, it cannot send any packets without first mov-

ing to the Active (Cell_DCH) state; MakeActive determines how long the radio should be idle before moving to active state.

We note that although the main goal of the two algorithms, MakeIdle and MakeActive, is to reduce the energy consumed, it is important not to incur excessive signaling overhead, which occurs every time the radio changes its state. We evaluate this overhead in our experiments.

Chapter 5

MakeIdle Algorithm

3GPP Release 7 [3] proposed a feature called *fast dormancy*, which allows the device to actively release the channel by itself before the inactivity timer times out on the base station side when it decides not to use the data connection any more. One of the problems with Release 7 is that the base station loses control over the connection when mobile devices are able to disconnect by itself. In 3GPP Release 8 [4], the fast dormancy mechanism has been changed now, the mobile device first sends a fast dormancy request, and the base station will decide whether to release the channel or not. This way the base station gets more control over the connection. In Europe, Nokia Siemens Networks has deployed Network Controlled Fast Dormancy based on 3GPP Release 8. An iPhone running iOS4.2 in the network supports fast dormancy. Since it is not clear what policy the base station will use to decide whether to release the channel or not upon receiving the request, in our simplified model, we assume that whenever the phone sends a fast dormancy request to the base station, the base station will accept and release the channel.

Currently, there is no accepted method governing when a device should initiate fast dormancy, and the standard committee suggests that the "device should utilize the application-layer knowledge" [7] for this purpose. The reason is that traffic patterns on smartphones change with time, and wrongly invoking fast dormancy can add extra overhead and result in small savings or even higher energy consumption. The MakeIdle method runs when the radio is in the Active state and determines when to turn the radio to the Low-power idle state (either by fast dormancy or by the inactivity timer's timeout). Its main goal is to

minimize energy consumption.

We start by giving a simple example to illustrate the problem and explain why a constant timer value (for any choice of constant) does not work well (§5.1). We then show what the optimal decision is *given* complete knowledge of a packet trace; the result is that the radio should be turned to Low-power idle if there is a gap of more than a certain threshold amount of time in the trace. The value of this threshold depends on the time it takes to move the radio between different states and on the energy consumed in making state transitions (§5.2). Then, we develop an online method to predict idle durations that will exceed this threshold by modeling the idle time using a conditional probability distribution (§5.3).

5.1 A Simple Example

To explain why the problem of determining when to turn the radio from Active to Low-power idle is non-trivial, we consider a simple traffic workload as an example. Figure **??** shows the simplified power consumption model. (the actual power consumption activity can be refered to Figure 8-1. In Figure 8-1, the non-zero power value in the Low-power idle mode comes from the screen and CPU. We substract that part in our model and assume that the Low-power idle mode consumes no energy. In reality, the energy consumed by 3G IDLE mode is negligible [21])

In Figure 5-1, the areas of the two triangles on the left and right sides are the energy consumed by switching from Low-power idle to Active and from Active to Low-power idle. These values cannot be changed. The area of the dark rectangular block is the energy consumed by transmitting data. The areas of the dashed rectangular blocks are the energy consumed in the Active state (but not while sending or receiving data), which lasts for time t_1, and the High-power idle state (FACH), which lasts for time t_2.

From Figure 5-1 we see that if the amount of data transmitted is small, the energy consumed by inactivity timers contributes a large portion of the total energy consumed. Hence, if the time interval between packets (or packet bursts) is large, reducing the inactivity timer would make sense. This is the intuition in the proposal in [6], which suggests using $T = 4.5$ seconds.

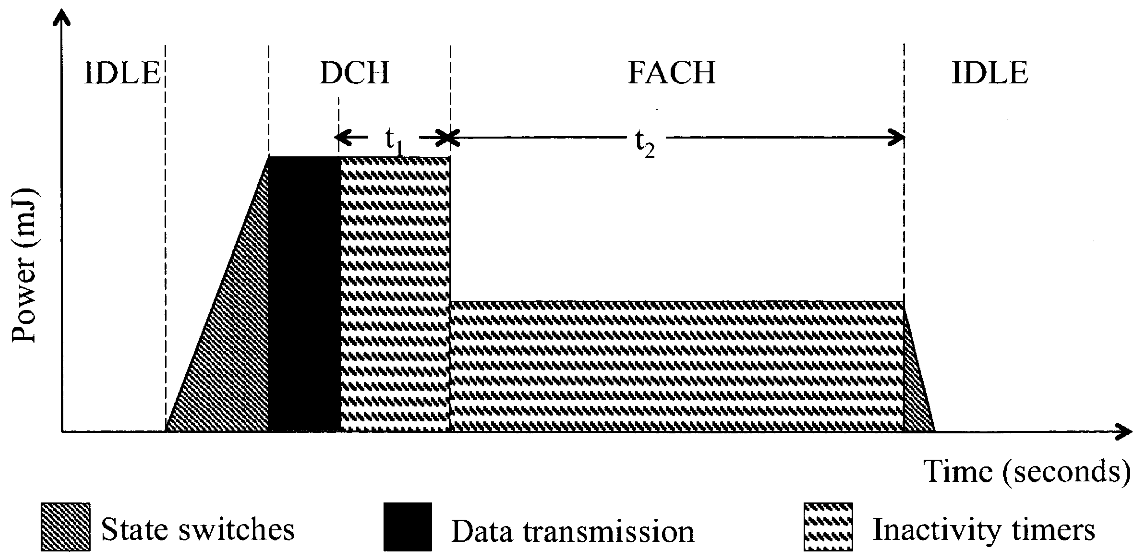


Figure 5-1: The simplified model for 3G energy consumption.

The question, even for this simple workload, is what the right value of T should be. The middle chart in Figure 5-2 shows that a fixed value may not save much energy compared to the status quo, while adding a lot more load on the base station because of the increased number of state switches. The bottom chart shows that what we would want for this simple workload is to turn the radio to Low-power idle after T seconds, where T is just a little larger than the longest inter-arrival time between packets within a single burst, but much smaller than the time between bursts. The reason for the "much" is that we don't want to turn the radio to Low-power idle and then find ourselves turning it back on soon after, for that wouldn't save much energy and also entail extra base station load.

If the traffic pattern were to change a little at this time and the bursts start occurring a bit closer together, as shown in Figure 5-3, a choice of T seconds will now no longer work, causing extra load and not saving energy. In this case, the right solution may well be to keep the radio in Active or High-power Idle mode and not move to Low-power idle at all (top chart).

Hence, changing the value of the inactivity timers not only changes the energy consumed, but may also change the number of state switches and traffic delays. Using a fixed value of inactivity timer does not always reduce the energy consumption. In the MakeIdle

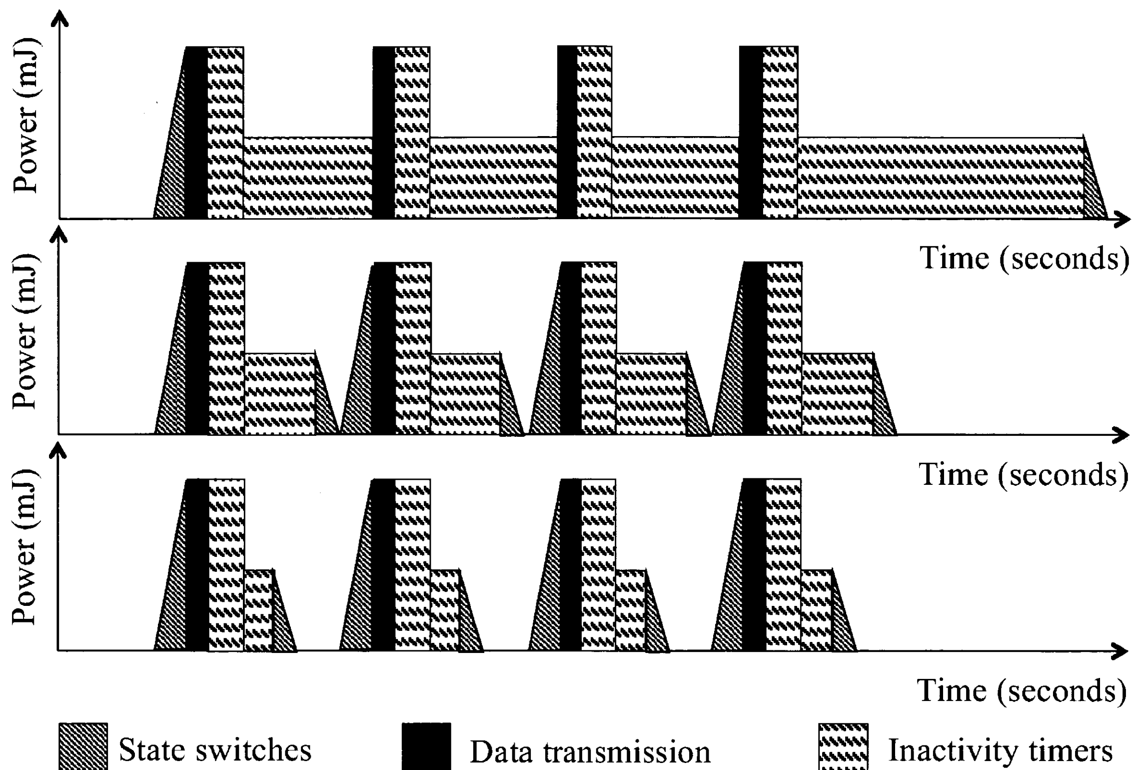


Figure 5-2: Reducing $t_1 + t_2$ to 4.5 seconds brings more state switches while saving little energy. Further reducing $t_1 + t_2$ to a smaller value that brings the same number of state switches can save more energy.

algorithm we dynamically choose the value of the inactivity timers to trigger fast dormancy optimally (we define what "optimally" means, below).

5.2 Optimal Decision From Offline Trace Analysis

Suppose we are given a packet trace containing the time-stamps of packets sent and received on a mobile device. Our goal is to determine offline when to turn the radio to the Low-power idle state using fast dormancy so that the total energy consumed is minimized. To simplify the problem, we will assume here that we do not want to allow any sessions to be delayed: i.e., we will assume that the radio should be turned to Low-power idle only if the time it takes to turn the radio to Low-power idle and back to Active is smaller than the inter-arrival time between packets. Since every packet eventually gets transmitted, we do not need to worry about the energy consumed by packets.

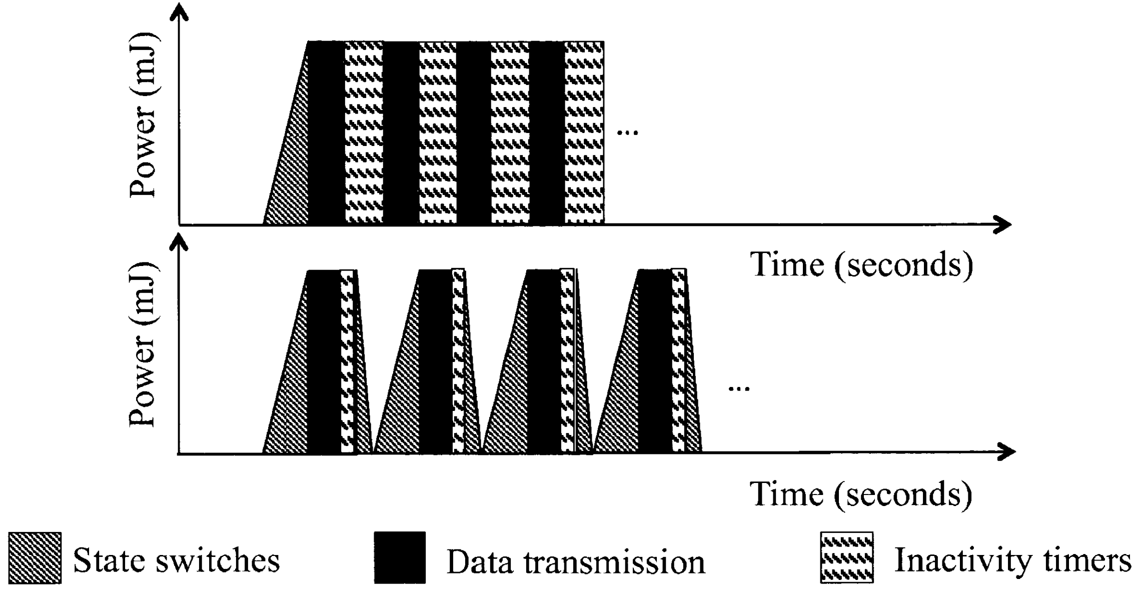


Figure 5-3: When the gap between traffic bursts is small, reducing $t_1 + t_2$ to a small value may consume more energy, while *also* adding delays for each burst.

If the inter-arrival time between two adjacent packets is t seconds, then the energy consumed by not triggering fast dormancy is given by

$$E_{noFD}(t) = \begin{cases} t \cdot P_{Active} & 0 < t \leq t_1 \\ t_1 \cdot P_{Active} + (t - t_1) \cdot P_{High_Idle} & t_1 < t \leq t_2 \\ t_1 \cdot P_{Active} + t_2 \cdot P_{High_Idle} & t > t_2 \end{cases} \tag{5.1}$$

In Equation (5.1), $E_{noFD}(t)$ is the energy consumed, t_1 and t_2 are the inactivity timers from the state machine described in §3 and Figure 2-1. P_{Active} and P_{High_Idle} are the power consumption numbers for the active state and High-power idle state; the power consumed by Low-power idle is negligible.

The energy consumed by triggering fast dormancy is:

$$E_{FD} = E_{MakeIdle} + E_{MakeActive} \tag{5.2}$$

Here, $E_{MakeIdle}$ and $E_{MakeActive}$ are the energy consumed by turning the radio from Active to Low-power idle and from Low-power idle to Active, respectively. These numbers

are fixed for a given type of mobile device and are easy to obtain from an offline experiment.

After calculating these two energy values, we should trigger fast dormancy if and only if $E_{FD} < E_{noFD}(t)$. For any given configuration of the RRC state machine, t_1 and t_2 are fixed numbers (as implemented in networks today), so t is the only variable in $E_{noFD}(t)$. Notice that because $E_{noFD}(t)$ is a monotonically non-decreasing function of t, there exists a value for t, which we call t_{energy}, for which $E_{FD} < E_{noFD}(t)$ if and only if $t > t_{energy}$. This expression quantifies the intuitive idea that after each packet, we should trigger fast dormancy only if we know that next packet will not arrive in the following t_{energy} seconds.

In addition, for now, we need to make sure not to delay any packets by keeping the radio in Idle state when a packet is ready to be sent or received. Let $t_{MakeIdle}$ and $t_{MakeActive}$ be the time taken for turning the radio from Active to Low-power idle state and from Low-power idle to Active respectively. These numbers are also fixed for a given mobile device and network, so we need to ensure that:

$$t \geq t_{MakeIdle} + t_{MakeActive} \tag{5.3}$$

Let

$$t_{FD} = max(t_{energy}, t_{MakeIdle} + t_{MakeActive}) \tag{5.4}$$

We should trigger fast dormancy if and only if $t \geq t_{FD}$ to minimize the energy consumption, assuming perfect future knowledge and no packet or session delays. We have measured the underlying parameters for the Android HTC G1 and the T-mobile network as deployed in the Boston area, finding that $t_{FD} = 3.98$ seconds. Other networks and devices are likely to have roughly similar (though not identical) values.

5.3 Online Prediction

To minimize energy consumption in practice, we need to predict future network activity; we need to know whether the next packet will come within t_{FD} seconds. In reality, we do this prediction by waiting for a short period of time and seeing whether any activity occurs.

If activity occurs, we reset and wait, but if not, we use the intuition that the longer the network is idle, the longer it may remain idle to trigger the transition to Low-power idle state. Here we call the short period of time used by waiting t_{wait}. We estimate the value of t_{wait} by a statistical method.

Suppose the current time at which we're making a decision is 0. Then, we choose a value for t_{wait} to make the following conditional probability "high enough":

$$p_{t_{wait}} = \mathbb{P}(\text{no packet in } t_{wait} + t_{FD} | \text{no function call in } t_{wait})$$

To decide how much is "high enough", we take energy into consideration: $p_{t_{wait}}$ *is "high enough" if the expected energy consumption from fast dormancy is less than the expected consumption of not using fast dormancy in the following* $t_{wait} + t_{FD}$ *seconds.*

The expected energy consumption for fast dormancy is:

$$\mathbb{E}[E_{FD}] = E_{MakeIdle} + E_{MakeActive}$$

The expected energy when not using fast dormancy in the following $t_{wait} + t_{FD}$ seconds can be calculated as:

$$\begin{aligned} \mathbb{E}[E_{noFD}] &= p_{t_{wait}} \cdot E_{noFD}(t_{wait} + t_{FD}) \\ &\quad + (1 - p_{t_{wait}}) \cdot \mathbb{E}[E_{inter_arrival < t_{wait} + t_{FD}}] \end{aligned}$$

Here, the $\mathbb{E}[E_{inter_arrival < t_{wait} + t_{FD}}]$ is the expected energy consumed before the next packet comes when the inter-arrival time between the current packet and next packet is less than $t_{wait} + t_{FD}$. This quantity can be calculated by observing that:

$$\mathbb{E}[E_{inter_arrival < t_{wait} + t_{FD}}] = \int_{t'=0}^{t_{wait} + t_{FD}} p(t') E'(t') dt',$$

where

$$p(t') = \mathbb{P}(\text{next packet at } t' + t_{wait} | \text{no packet in } t_{wait}).$$

We choose $t_{wait} = T_{wait}$ to maximize the following expression:

$$\mathbb{E}[E_{noFD}] - \mathbb{E}[E_{FD}] - E_{noFD}(t_{wait}) \tag{5.5}$$

Maximizing (5.5) is equivalent to maximizing the expected energy we can save by using fast dormancy. The factor $E_{noFD}(t_{wait})$ is the penalty paid for waiting for time T_{wait} to get to the point that we are confident enough that using fast dormancy can save energy.

Chapter 6

MakeActive Algorithm

The MakeIdle algorithm reduces the 3G wireless energy consumption by putting the radio in the Low-power idle state as soon as the radio is inactive, which may bring signaling and energy overhead when the radio is made active for next session's transmission.

Figure 5-3 shows that when we reduce the value of inactivity timers, we may bring more state switching from Low-power idle to Active and From Active to Low-power idle. This switching brings signaling overhead to the link between the device and base station. One idea to reduce the signaling overhead is to "shift" the traffic bursts in order to combine several traffic bursts together [15, 5], as shown in Figure 6-1.

Figure 6-1 shows that we can reduce the number of state switches by delaying earlier traffic bursts to buffer traffic together and send it out all at once. It is clear that the longer earlier bursts are delayed, the more bursts we can accumulate and the fewer state switches occur. This buffering not only reduces the number of state switches, but also eliminates the energy consumed by state switching. Theoretically, we get the minimum number of state switches when we delay sessions by an infinite amount of time. But this way the transmission can never proceed, so there is another tradeoff between the traffic delay and number of state switches. Our MakeActive algorithm is able to balance the tradeoff.

For many applications, one can save energy by delaying the start of the session or packet burst and waiting for additional data (from other sessions) to amortize the cost of making the radio active, without appreciably degrading the user's experience. This idea of delaying the activation of the radio has been explored before [5], but rather than waiting ten minutes,

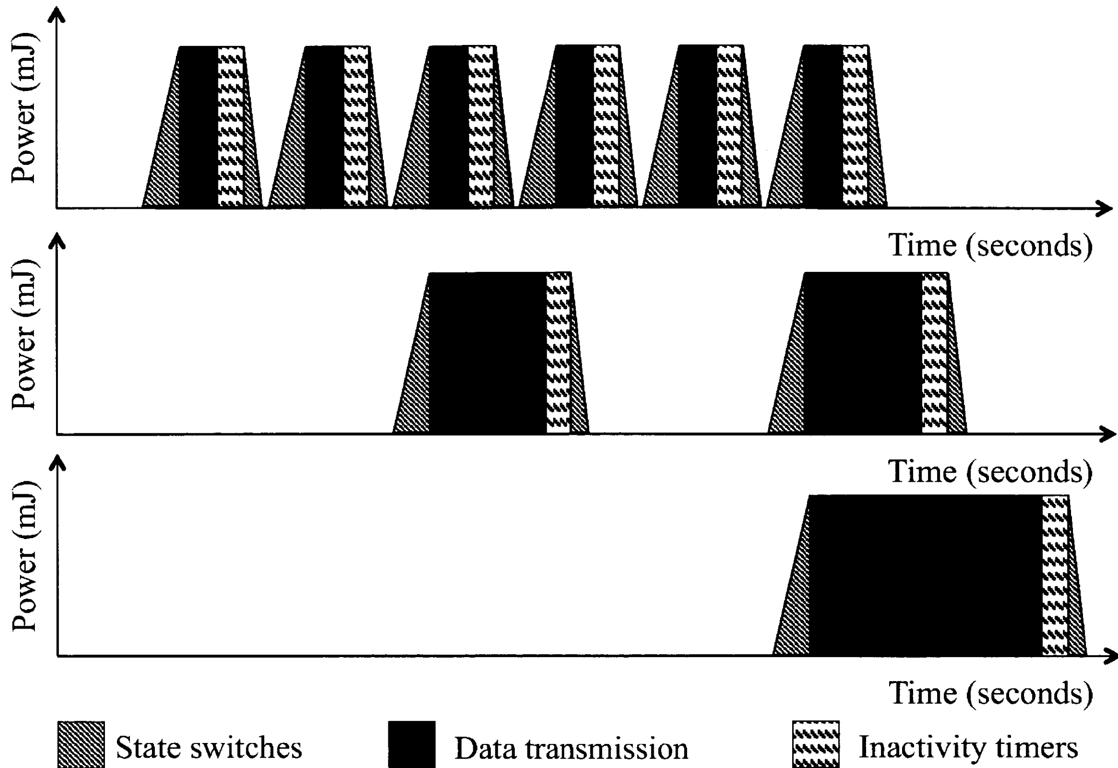


Figure 6-1: "Shift" traffic to reduce number of state switches.

our approach is to reduce this time to several seconds. Our goal here is background applications that can tolerate a small delay at the start of a session, not interactive applications like web browsing where delaying by a few seconds is unacceptable.

We first consider the relatively straightforward scheme in which the start of a session (i.e., a burst of packets) can be delayed by at most a certain maximum delay bound, T_{delay}, and evaluate the resulting energy consumption, which should be lower because multiple sessions may end up getting batched together. We then apply a machine learning algorithm to reduce the average delay experienced by a session while consuming the same amount of energy as when we use a constant maximum delay bound. Our contribution lies in the application of this algorithm to learn idle durations for the radio, balancing energy consumption with the cost of signaling overhead and increased session latency.

6.1 Fixed Delay Bound

A simple straw-man is to set a fixed delay bound, T_{delay}; we pick 20 seconds. When the radio is in Low-power idle state and a socket tries to start a new session at current time t, the control module decides to delay turning the radio to Active mode until $t + T_{delay}$, so that other new sessions which might come between time t and $t + T_{delay}$ will all get buffered and will start together at time $t + T_{delay}$. There is a trade-off between the delay bound and the number of sessions that can be buffered. Note that once a session begins, its packets do not get further delayed, which means that TCP dynamics should not be affected by this method.

The problem with fixed delay bound is that it is not adaptive to the traffic pattern. Every time the delay is triggered, the first traffic bursts will incur a delay of T_{delay}. We show in the evaluation that a large portion of the traffic bursts get delayed by T_{delay}. However, waiting as long as T_{delay} may be overkill; as data accumulates (especially from different sessions), there comes a point when the radio should be switched to Active and data sent before this delay elapses, which will reduce the expected session delay while still saving energy. We now describe a method that decides when we should start sending using a machine learning algorithm.

6.2 Learning Algorithm

We apply the *bank of experts* machine learning algorithm based on Hidden Markov Model (HMM). The hidden variable is the identity of the best expert, which corresponds to the best delay time in our case. Each iteration (each time the radio is in Low-power idle mode and a socket call is trying to use the 3G network), there should be one expert who proposes the optimal value for the delay. This algorithm has been applied to the 802.11 power saving mode configuration problem [13], but the problem setup is different for the 3G energy environment.

6.2.1 Bank of Experts

We bound the maximum delay to 20 seconds. Each expert "proposes" a fixed number of delay value T_i:

$$T_i = i,\ i \in 1 \dots 20$$

The output of the algorithm is the weighted average over all the experts:

$$T_t = \sum_{i=1}^{20} p_t(i) T_i$$

For each iteration of the updates, the HMM calculates the probability of each possible hidden state (in our case, the identity of the expert) based on some observation y_t. Here, we can define the probability of predicting observation y_t as $P(y_t|T_i) = e^{-L(i,t)}$ (The observation is the number of sessions we batched at time t, and $L(i,t)$ is the loss function which we will discuss in detail later). Then we can apply the following equation to get the weight $p_t(i)$:

$$p_t(i) = \frac{1}{Z_t} \sum_{j=1}^{n} p_{t-1}(j)\, e^{-L(j,t-1)}\, P(i|j,\alpha).$$

Here, Z_t is a normalization factor. The last part of the equation shows the probability of switching between experts. There are different versions of this algorithm for HMMs. The one we chose [?] supports switching between the experts and is suitable for cases where the observation may change rapidly, which matches the bursty character of network traffic. $P(i|j,\alpha)$ is defined as:

$$P(i|j,\alpha) = \begin{cases} (1-\alpha) & i = j \\ \frac{\alpha}{n-1} & i \neq j \end{cases}$$

$0 \leq \alpha \leq 1$ is a parameter that determines how quickly the algorithm changes the best experts. α close to 1 means the network condition changes rapidly and the best expert always changes. One problem with this algorithm is that it is hard to choose a goodα. In reality, α should not be a fixed value since the network traffic pattern may change rapidly or remain stationary. We use a more adaptive algorithm, Learn-α [12, 14], to dynamically choose α.

The basic idea is to first assign m α-experts and use the algorithm above to learn the proper value of α in each iteration, and then use the up-to-date α to learn T_t [12, 14]. The final equation for this "two-layer learning" is:

$$T_t = \sum_{j=1}^{m} \sum_{i=1}^{n} p'_t(j) p_{t,j}(i) T_i \tag{6.1}$$

Here, $p'_t(j)$ is the weight for the jth α-expert, which is given by:

$$p'_t(j) = \frac{1}{Z_t} p'_{t-1}(j) e^{-L(\alpha_j, t-1)} \tag{6.2}$$

This equation shows that $p'_t(j)$ is updated from the previous value $p'_{t-1}(j)$; the initial values are: $p'_1(j) = 1/m$. $-L(\alpha_j, t-1)$ is the α loss function, defined as:

$$L(\alpha_j, t) = -\log \sum_{i=1}^{n} p_{t,j}(i) e^{-L(i,t)} \tag{6.3}$$

$L(i,t)$ is the loss function, discussed in §6.2.2.

6.2.2 Loss Function

The loss function, $L(i,t)$, is a crucial component of the scheme and depends on the details of the problem to which the learning is applied. Because our goal is to reduce the delay as well as save energy and reduce signaling overhead by batching, $L(i,t)$ should express the trade-off between the total time delayed for all the buffered sessions and the number of session buffered, which is proportional to both the energy and the signaling overhead reduced by batching.

$$L(t) = \gamma Delay(T_t) + \frac{1}{b} : \gamma > 0$$

Here, γ is for scaling between the two parts of the loss function In our implementation, we choose $\gamma = 0.008$. $Delay(T_t)$ is the summation of time delayed over b sessions. b is the number of sessions currently buffered, which is proportional to the energy saved by combining b sessions into one session and sending them together. $1/b$ means that as the number of buffered sessions increases, the value of this part of the loss function reduces,

while the other part $\gamma Delay(T_t)$ may increase. This formula balances the tradeoff between delay and energy saving. Let t_k be the arrival time of the kth session, and t the time the algorithm takes to make a decision. Then we have:

$$Delay(T_t) = \sum_{k=1}^{b} t + T_t - t_k$$

In our weight updates, we apply this loss function to each expert i, indicating the loss that would have accrued had the algorithm used T_i instead of T_t as its blocking time. The equivalent loss per expert i is:

$$L(i,t) = \gamma Delay(T_i) + \frac{1}{b}$$

Chapter 7

Implementation

We have implemented the MakeIdle algorithm on the Android platform. The implementation has two parts: the modification of the socket layer and the control module.

We modified the `org.apache.harmony.luni.net.PlainSocketImpl` library, which provides the interface from the Android platform code to the Linux kernel. The library maintains a FileDescriptor (`fd` in Figure 7-1) bound to a local port for each socket, and inside the `connect`, `read`, `write`, `close` socket calls are operations to that FileDescriptor. In our modification, we create an extra FileDescriptor called `efd` for each socket during construction and bind it to a special local port. Inside each socket call the name of the function call and other useful information is written to `efd` and then the function call continues with its normal operations. Our socket layer modifications are shown in Figure 7-1.

The control module runs at the application layer as a server listening on the special local port mentioned above. When there is an `efd` created by any application trying to bind to this port, the control module will build a local socket connection between itself and the `efd`. The control module can therefore receive the function call information written to the `efd`. In this way, the control module records all socket function calls and runs the MakeIdle algorithm.

Socket Implementation

```
PlainSocketImpl.Java
--------------------
fd:FileDescriptor
netImpl: NetworkSystem
--------------------
connect(){
netImpl.connect(fd,..);
}

read(){
netImpl.read(fd,..);
}

write(){
netImpl.write(fd,..);
}

close(){
netImpl.close(fd,..);
}
```

Modified
Socket Implementation

```
PlainSocketImpl.Java
--------------------
fd: FileDescriptor
efd: FileDescriptor
netImpl: NetworkSystem
--------------------
connect(){
netImpl.write(efd,..);
netImpl.connect(fd,..);
}

read(){
netImpl.write(efd,..);
netImpl.read(fd,..);
}

write(){
netImpl.write(efd,..);
netImpl.write(fd,..);
}

close(){
netImpl.write(efd,..);
netImpl.close(fd,..);
}
```

Figure 7-1: Socket-layer modifications.

Chapter 8

Evaluation

We measured the power consumption and inactivity timer values using the Monsoon Power Monitor [1]. Figure 8-1 shows graphs of our measurements during a radio state switches cycle on HTC G1 (running Android 2.2) and Samsung Nexus S (running Android 2.3) smartphones in T-Mobile 3G network. During the High-power idle (Cell_FACH) and part of Active (Cell_DCH) states, there is no data transmission. The RRC state machine keep the radio on here in case a new transmission or reception may occur in the near future. Consistent with previous work, we use the term *tail* to refer to this duration when the radio is on but there is no data transmission [5] The two graphs show similar energy consumption characteristics for the two phones, so we use the HTC G1 in our evaluation.

We measured the inactivity timer values in the T-mobile 3G network in a the Boston area to be $t_1 \approx 3.22$ seconds and $t_2 \approx 16.22$ seconds. Here, $t_1 + t_2$ is about 20 seconds; we refer to this configuration of RRC state machine as the "20-second tail". The energy consumed at the end of a data transfer when the radio is in one of the two Idle states before turning off is termed the *tail energy* [5]; this energy can be 60% or more of the total energy consumption of 3G, as explained in §1.

We collected `tcpdump` traces on an HTC G1 phone running Android 2.2 Platform with the modified socket layer library to obtain all the socket function call records with timestamps. We ran the following popular applications. These applications have the "always on" property in that they usually send or receive data over the network whenever they run, without necessarily requiring user prompting or input.

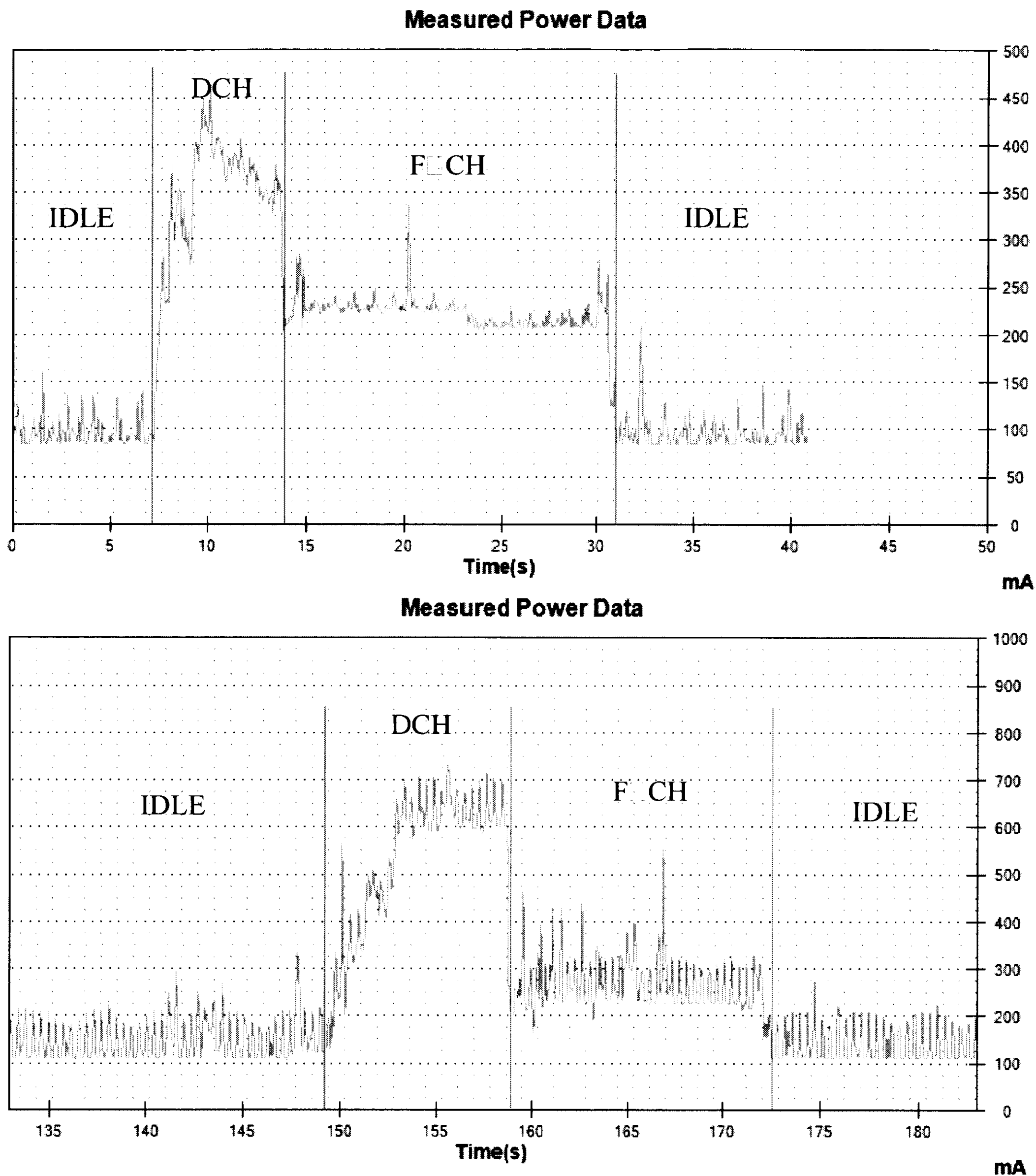


Figure 8-1: The measured power consumption of the different 3GPP RRC states for the HTC (top) and Nexus S (bottom) smartphones. The screenshot from the monitor shows the current drawn; the voltage is 3.7 volts. The average power consumed while "Active" (Cell_DCH) is 1028 mW for HTC and 1804 mW for Nexus S. In "High-power idle" (Cell_FACH), the power consumed is 445 mW for HTC and 450 mW for Nexus S. "Low-power idle" (IDLE) is 0 watts for both devices. In this figure the power level is non-zero because of the CPU and LED screen power consumption. We did not measure the power consumed in Cell_PCH state because our measured network/phones do not support that state. The GSM specification suggests that the Cell_PCH state is just slightly higher than IDLE compared to Cell_FACH.

Email: This application is run mostly in the background, synchronizing with an email server every five minutes.

Ebuddy: An IM application that sends heartbeat packets to the server periodically, typically every 5 to 20 seconds.

CNN News: A news reader that has a background process running to fetch breaking news.

Twidroyd: An Android Twitter application, which automatically fetches new tweets without user input.

Game with ad bar: A game that can run offline, but with an advertisement bar that changes the content roughly once a minute.

Scottrade: An application for monitoring the stock market, which updates roughly once per second when running in the foreground.

Facebook: A user logs into her Facebook account, read the news feeds, clicks to see pictures, and posts comments. When running in background, this application updates only every 30 minutes. We did not collect much background traffic from it. We use the foreground traffic trace as a comparison trace.

For each application, we collected a trace that was 30 minutes long. We also collected twenty 30-minute traces with at least two applications running at the same time. All the traces are collected in a same indoor location.

One caveat in our experimental results is that because fast dormancy is not yet supported on US 3G networks, we were unable to accurately measure the delay to turn the radio from Active to Idle and the energy consumed. We believe, however, that one can approximate this value by measuring the delay and energy consumed in turning the data connection off on the phone. In practice, we expect the delay and energy of fast dormancy switching to be lower, so we model the turn-off energy and delay for fast dormancy to be 50% of the values measured while turning the radio off. We evaluated our methods for other reasonable fractions than 50% and found that the results did not change appreciably; hence, we believe that our conclusions are likely to hold if one were to implement the methods on a device that supports fast dormancy.

8.1 MakeIdle Evaluation

First, we evaluate the MakeIdle algorithm without the MakeActive algorithm enabled. When new packets arrive from applications, we will not delay them, but just turn the radio to Active mode (if it is not already Active), and transmit data. We compare three different algorithms on each trace: the offline optimal algorithm (which we refer as "Oracle"), the 4.5-second tail, and MakeIdle. In each run of MakeIdle, we first run the applications for a short period of time to obtain an estimate of t_{wait}, and then run the experiment for a much longer time using the fixed t_{wait}. It would, of course, be better to estimate t_{wait} in real-time, but the results for a static t_{wait} are still good. The Oracle method is the one described in Section 5.2, where we can "see the future" and make optimal decisions. We calculate the percentage of energy saved by each algorithm over the currently deployed 20-second tail scheme.

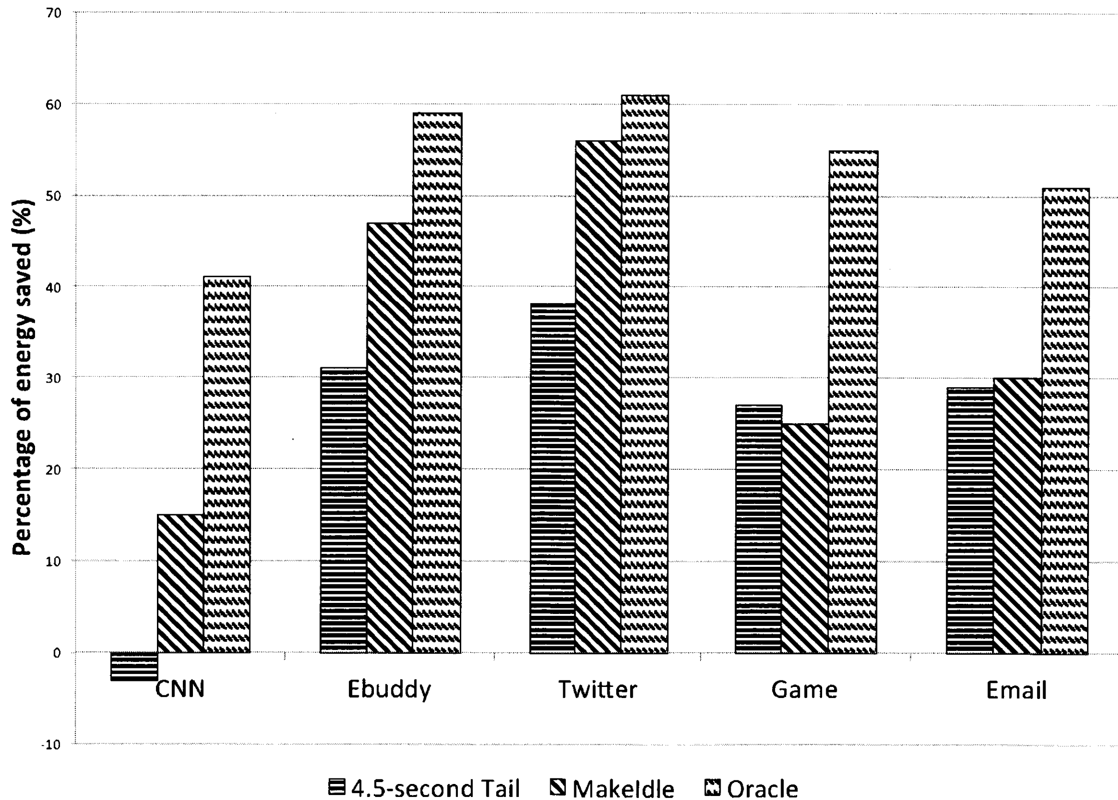


Figure 8-2: Energy saved for different applications.

Figure 8-2 shows the energy saved by the individual applications, while Figure 8-3 shows the CDF of the percentage of energy saved over 27 traces. The mean and median values corresponding to the CDFs in Figure 8-3 are listed in Table 8.1.

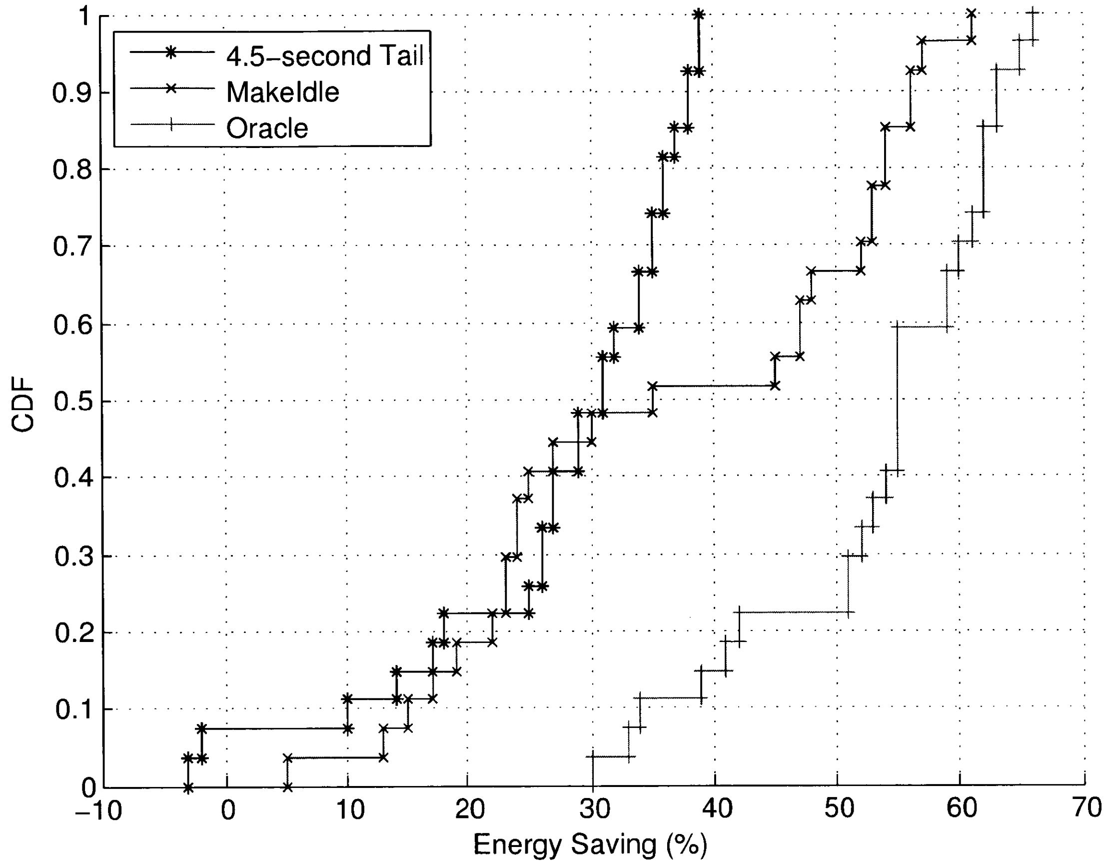


Figure 8-3: CDFs of the percentage of energy saved over 27 traces, running various combinations of the seven applications.

Algorithm	Median (%)	Mean (%)
4.5-second tail	31	27
MakeIdle	35	36
Oracle	55	53

Table 8.1: Median and mean amounts of energy saved by the different methods relative to the 20-second tail scheme.

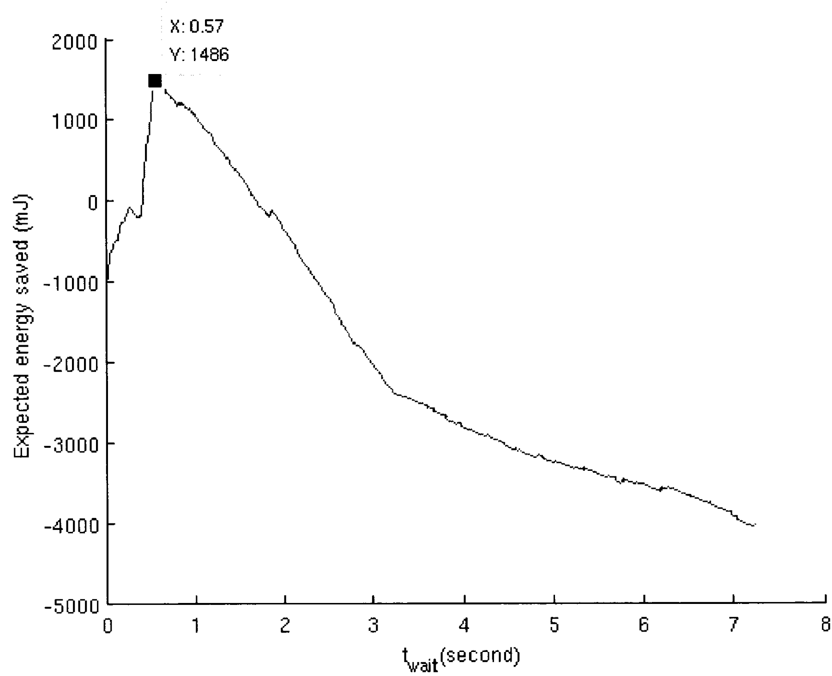
The reason MakeIdle saves more energy than the 4.5-second tail scheme is that MakeIdle makes decisions more quickly, turning the radio idle sooner in many cases. Both meth-

ods wait for some period of time, t_{wait}, after the last packet/function call activity has occurred, before changing the state of the radio to idle. We find that for MakeIdle, the typical value of t_{wait} is between 0.5 seconds and 1 second, rather than the higher value of 4.5 seconds. The combination of picking t_{wait} by calculating the expected energy savings, and being aware of network traffic, enables MakeIdle to make more aggressive decisions in many cases.

We conclude the evaluation of MakeIdle with two findings: one demonstrating the strength of the approach, and the other highlighting a limitation.

MakeIdle is able to correctly predict a good value of t_{wait} using a relatively modest amount of training, and the optimization yields values of t_{wait} that are close to the best possible. We demonstrate that here for one application, Ebuddy (the other applications and application mixes provided similar results). We first train the MakeIdle method on 10% of the trace data (3 minutes worth), then apply the energy optimization method described in Section 5 to find the value of t_{wait} that minimizes the expected energy consumption. The top curve in Figure 8-4 shows the results; the optimal value found by MakeIdle and shown in Figure 8-4(top) is 0.57 seconds. We then take the entire trace data and simulate a variety of different possible t_{wait} values between 0 and 7 seconds. In this offline full-trace simulation, we find the optimal for this trace to be 0.5 seconds, shown in Figure 8-4 (bottom) , which is very close to the value estimated by MakeIdle. This result suggests that MakeIdle works as intended.

However, MakeIdle will not always save energy. For some applications, we find that we should not trigger fast dormancy because the traffic is dense and there is little room for energy saving. Fortunately, MakeIdle is able to detect such cases during the course of its optimization. When computing the expected energy saving, MakeIdle returns *negative values* for the energy savings for all values of t_{wait}, as shown in one example in Figure 8-5. In this case, the radio should always remain active. Figure 8-5 (top) shows the expected energy saving for Scottrade. The value is always negative. The reason is that this application updates data roughly every second and continuously generates packets, so it does not make sense to trigger fast dormancy. Figure 8-5 (bottom) verifies this point using an exhaustive simulation across many possible values of t_{wait}, finding that there is no local maximum.



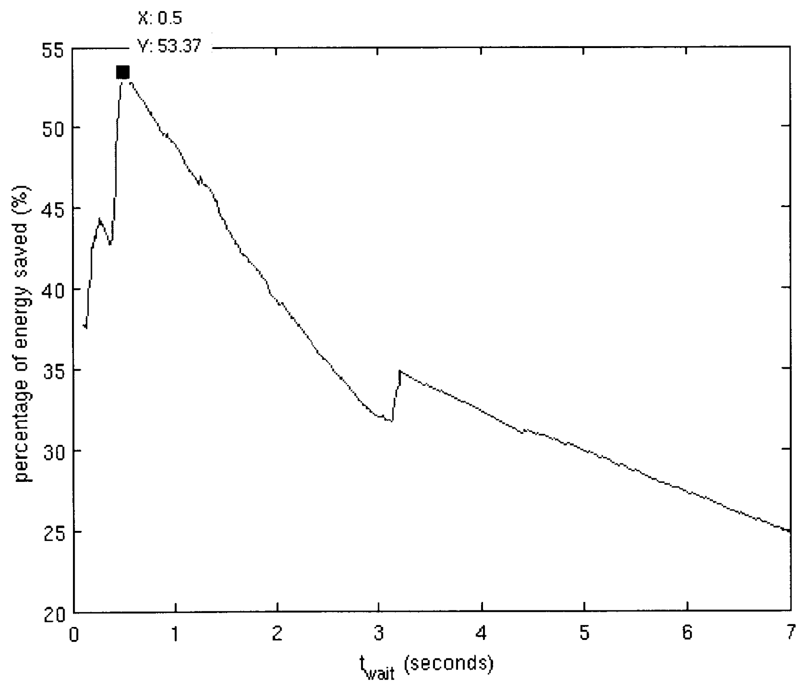
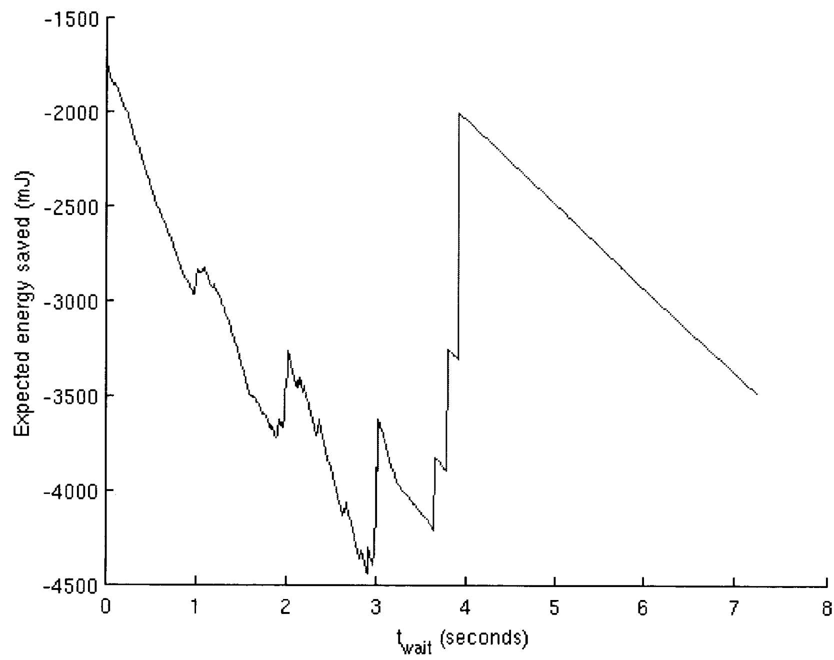


Figure 8-4: The value of t_{wait} that maximizes expected energy saving is close to the value maximizes the actual energy saving for a particular application (Ebuddy). Similar results hold for all applications.



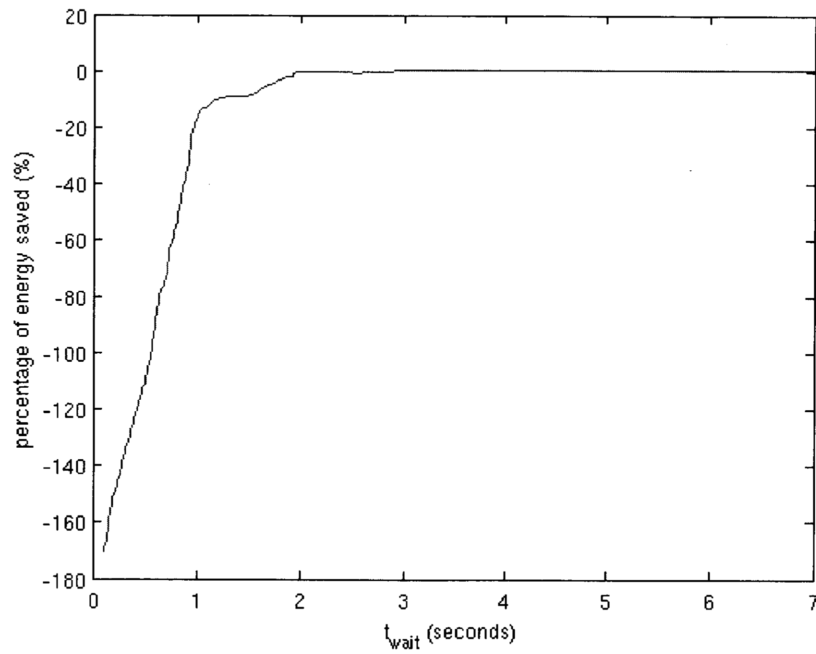


Figure 8-5: An example where fast dormancy should not be triggered when running applications with continuous and intensive data transmission. The bottom figure shows actually there is no proper t_{wait} that can reduce energy consumption.

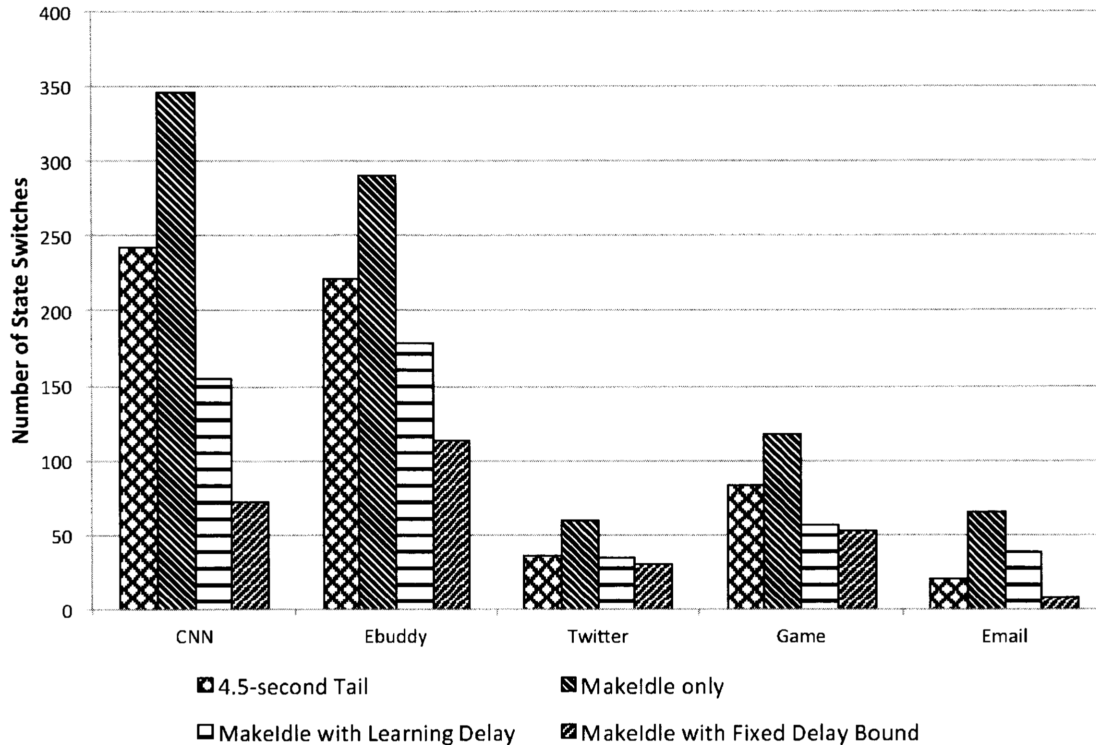


Figure 8-6: The number of state switches, which is proportional to the signaling overhead, for the individual applications. Each bar is for a different method that determines when the radio should be turned to the Idle mode.

8.2 MakeActive Evaluation

Although shortening t_{wait} with the MakeIdle algorithm saves considerable amounts of energy, it may bring more state switches between the Low-power Idle and Active states. We can reduce the number of state switches by introducing a little delay. Figure 8-6 shows the CDFs of the number of state switches for the "4.5 second tail" scheme, MakeActive, and "Oracle" schemes running on different single applications. Figure 8-7 shows the corresponding CDFs of the number of switches for different algorithms over 27 traces. For each CDF curve, the mean and median are given in Table 8.2.

Table 8.2 shows that using MakeIdle only, we introduce *more* state switches compared to the 4.5-second Tail method: 18% for the mean and 27% for the median. But when we also use MakeActive, the number of state switches *reduces* by 26% (mean) and 44%(median). For the fixed delay-bound method, which does not learn or adapt the delay to turn

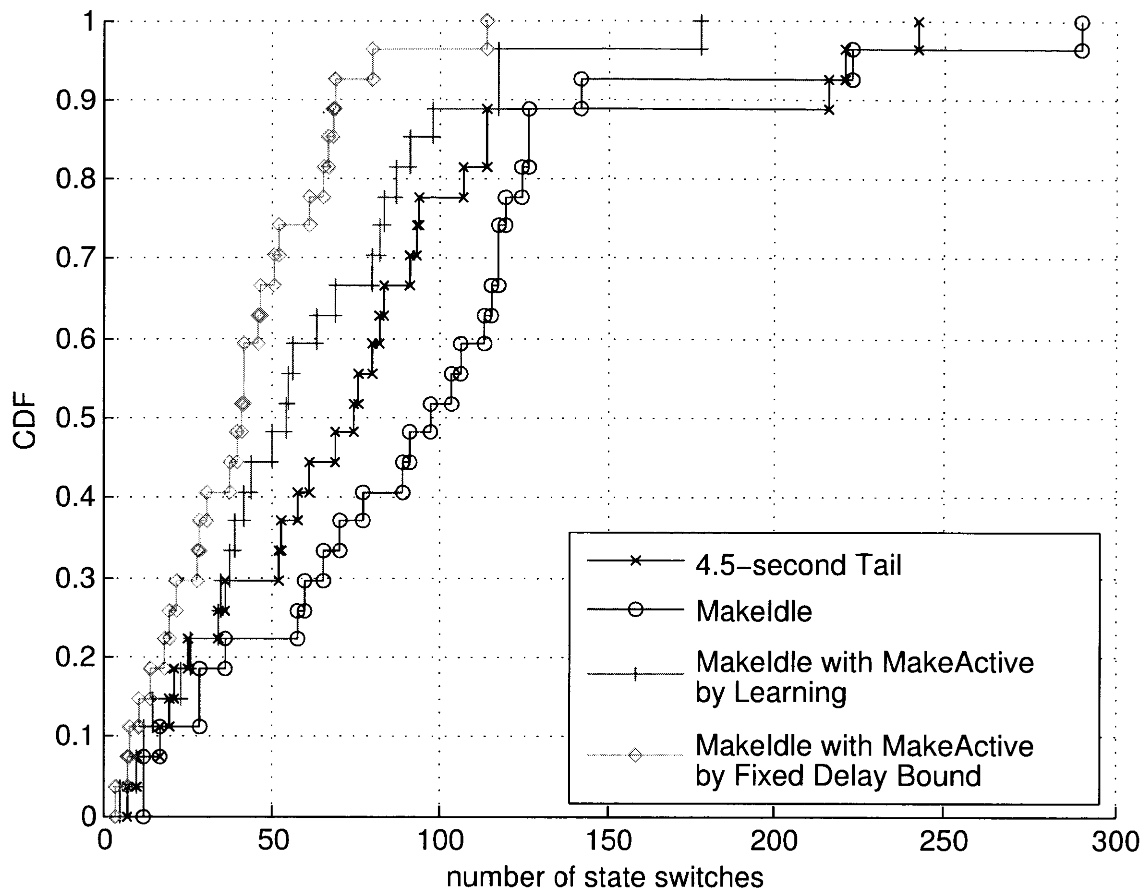


Figure 8-7: CDFs of the number of state switches across the 27 combined-application traces for the different methods to put the 3G radio in idle (off) mode.

the radio to the active mode, the number of state switches reduces by an even greater 49% (mean) and 44% (median). Of course, the fixed-delay scheme incurs higher latency.

Another contribution of the MakeActive algorithm is to reduce the energy consumed by state switching since it prevents state switching from happening frequently. Figure 8-8 shows the energy saved for individual applications, and Figure 8-9 shows the CDF of energy saved when we use MakeActive together with MakeIdle. There are fewer radio

Algorithm	Median	Mean
4.5-second tail	74	80
MakeIdle	97	95
MakeIdle with learning delay	41	59
MakeIdle with fixed delay bound	41	41

Table 8.2: Median and mean of number of state switches over 27 traces.

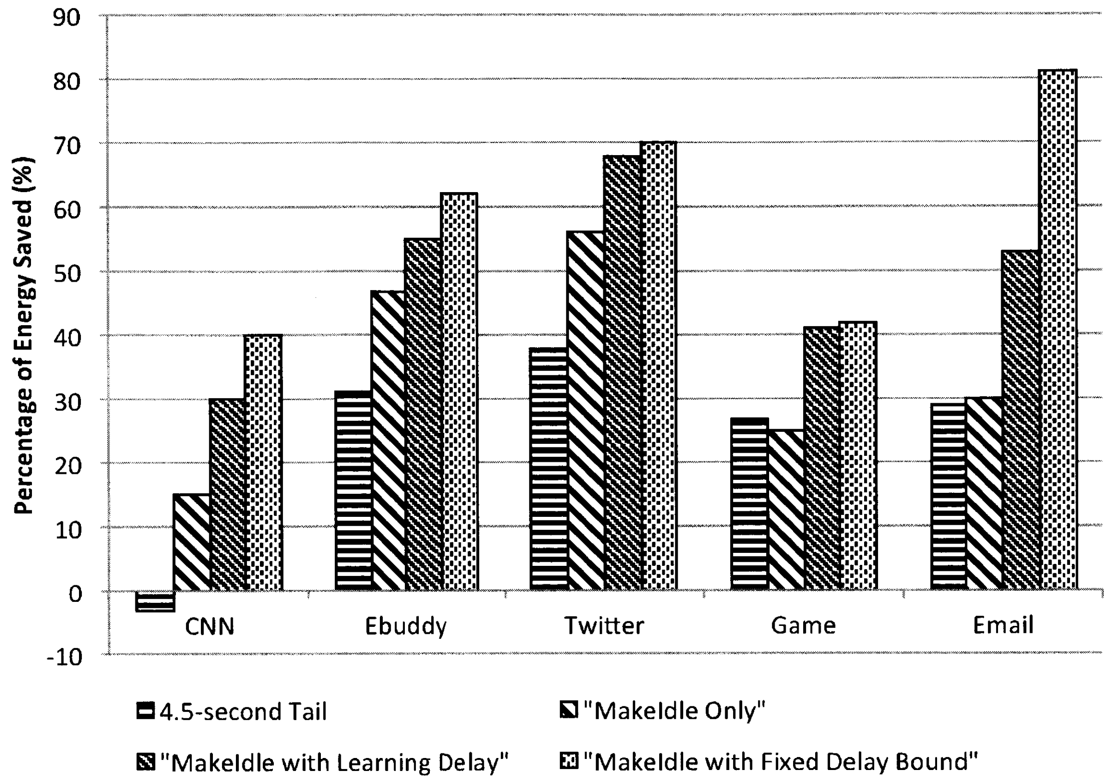


Figure 8-8: Energy saved for different applications.

state switches from Low-power idle to Active and from Active to Low-power idle, so more energy is saved. The mean and median energy savings for the different methods are listed in Table 8.3.

Algorithm	Median (%)	Mean (%)
4.5-second tail	31	27
MakeIdle	35	36
MakeIdle with learning delay	47	53
MakeIdle with fixed delay bound	52	59

Table 8.3: Median and mean energy savings over the 20-second tail.

In Section 6, we introduced two algorithms: the fixed delay bound, and the "bank of experts" learning algorithm. Figure 8-10 shows the CDF of delays incurred by traffic bursts over a half-hour trace. The maximum possible delay for each algorithm is 20 seconds. When using the fixed delay bound, there are a number of traffic bursts incurring 20-second delays, while with the learning algorithm, the delay value changes dynamically with the

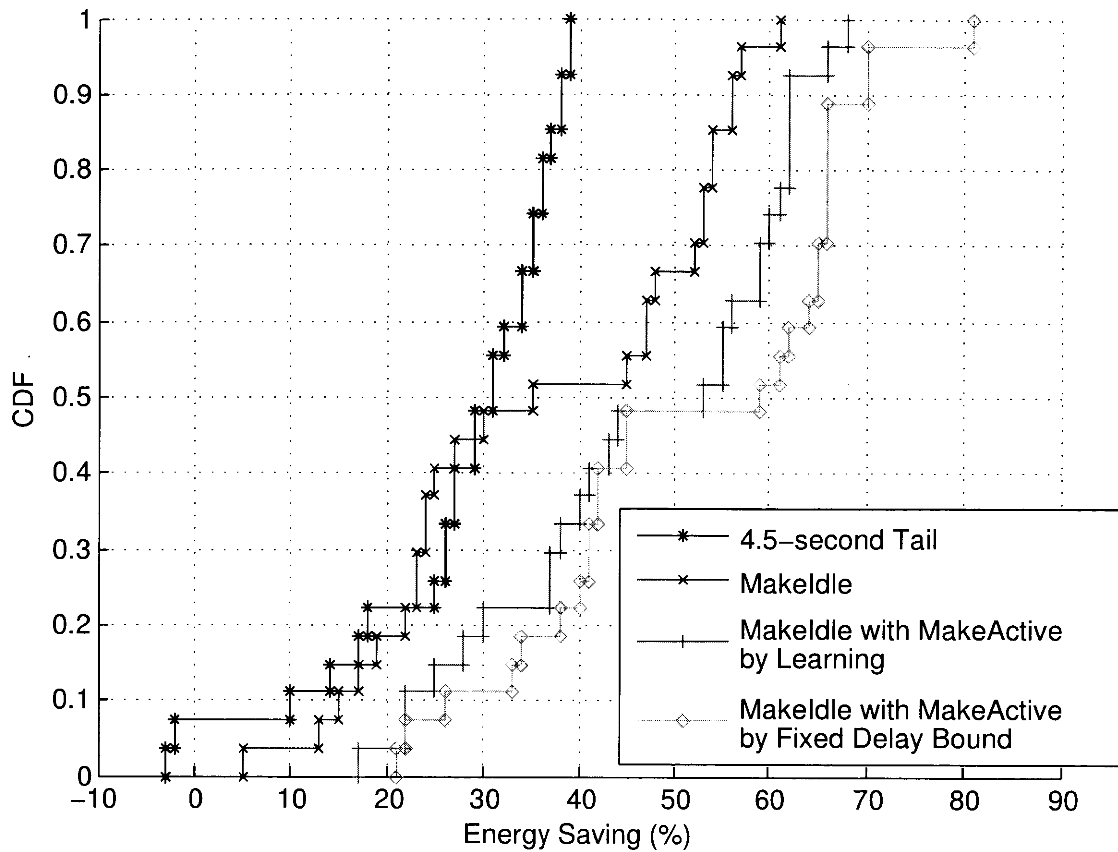


Figure 8-9: CDFs of energy savings using the 4.5-second tail, MakeIdle alone, MakeIdle with MakeActive using learning, and MakeIdle with MakeActive using a fixed delay bound.

number of buffered traffic bursts. For the learning algorithm, the median and mean delays are 6.72 s and 6.46 s, compared to 17.18 s (median) and 16.02 s (mean) for the fixed delay bound scheme.

Figure 8-11 shows how the learning algorithm works. The initial value of t_{delay} is 10 seconds, which is the average of all the experts. It decreases rapidly when the number of buffered bursts increases. When t_{delay} times out, the number of buffered bursts reverts to zero, and t_{delay} also increases a little, meaning the learning algorithm produces a longer delay when the number of buffered bursts decreases.

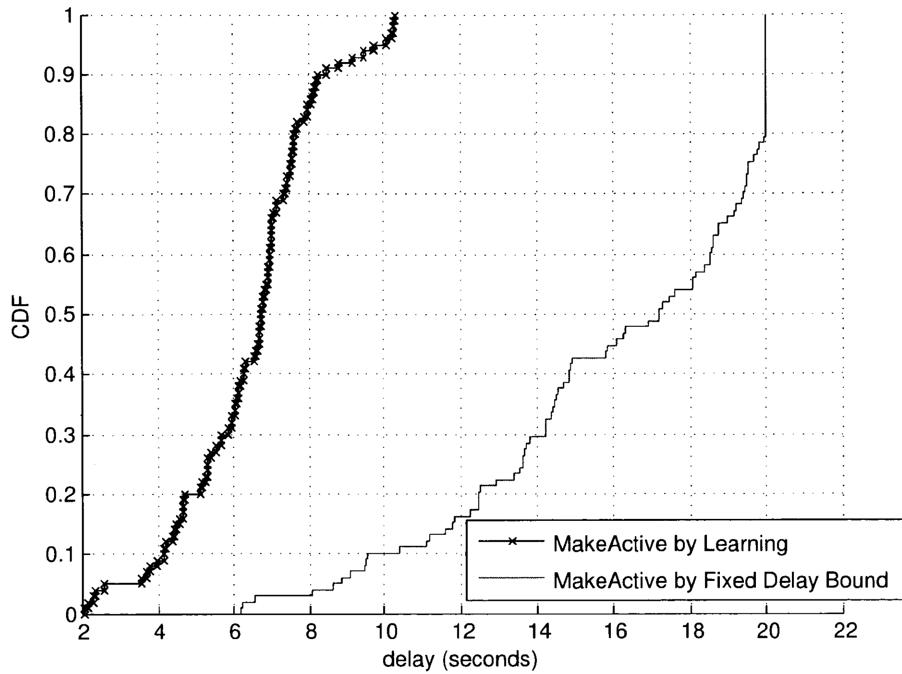


Figure 8-10: CDF of traffic burst delays.

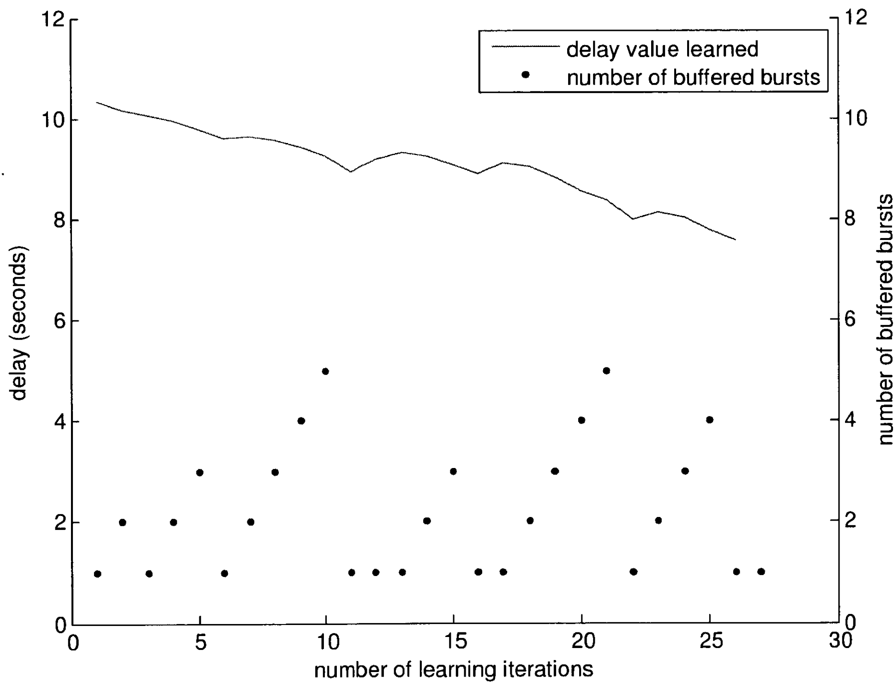


Figure 8-11: Delay value changes as the learning proceeds.

8.3 CPU and Energy overhead of running algorithms

Because our algorithms aim to reduce the energy consumption, it is important that they be lightweight. The measured CPU usage while running the two methods is less than 3%.

We compared the power consumption in the following two cases: first, running only the application that generates background traffic, and second, running both the control module and the application. We run each case on the phone for 10 minutes. The average power level for the first case is 869 mW, and for the second case is 890mW. When running the applications, the main energy consumers are the 3G interface and the screen. The power level with the phone doing nothing and the screen on is 360 mW. If we deduct this part from 869 mW, we conclude that the power consumed by the application running is 509 mW. The energy overhead introduced by our control module is 4%. If we assume most of this 509 mW is consumed by 3G interface, then the energy overhead is negligible compared to the savings it brings.

Chapter 9

Conclusion and Future Work

3G energy consumption is widely recognized to be a significant problem [15, 5, 11]. We developed a system to reduce the energy consumption by triggering fast dormancy using knowledge of the network workload. Our design reduced packet delays and signaling overhead by batching sessions, learning from network activity. We implemented the MakeIdle method on commodity phones running Android and evaluated our methods using traces collected on an HTC G1 phone running over the T-mobile 3G network (the results evaluated on a Nexus S phone were similar). Our results show that the MakeIdle and MakeActive methods can reduce the 3G energy by 36% on average over the status quo (20-second tail) with no session delays, and by 52% on average with a session delay of (only) 6.46 seconds.

The key idea in this thesis is to adapt the state of the radio to network traffic. To put the 36% saving (without any delays) or 52% saving (with delay) in perspective, we note that according to the Nexus S specifications, the reduction in lifetime from using the 3G radio instead of 2G is 7.3 hours; while it is not clear what application mix produces these numbers, one might speculate that saving 36% of the energy might correspond to an increase in lifetime by about 36% of 7.3 hours, or about 2.6 hours.

There are several areas for future work. First, we are planning to make our algorithm more configurable. In particular, when an interactive application is running in the foreground, the system should disable MakeActive and only run MakeIdle. Second, studying the effects of triggering fast dormancy on the base station side would be useful, considering issues such as handling multiple phones triggering the feature, and whether the base station

can actively help the phone to make decisions on fast dormancy by buffering incoming traffic for the phone. And last but not least, one could extend the system to include server or base station functions to coordinate with the mobile device to further reduce energy consumption.

Bibliography

[1] Monsoon power monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[2] Data Service Options for Spread Spread Spectrum Systems: Service Options 33 and 66, May 2006.

[3] 3GPP Release 7: UE Fast Dormancy behavior, 2007. 3GPP discussion and decision notes R2-075251.

[4] 3GPP Release 8: 3GPP TS 25.331, 2008.

[5] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Internet Measurement Conference*, pages 280–293. ACM, 2009.

[6] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A First Look at Traffic on Smartphones. In *Internet Measurement Conference*, Melbourne, Australia, 2010.

[7] GSM Association. Network Efficiency Task Force Fast Dormancy Best Practices, May 2007.

[8] R. Krashinsky and H. Balakrishnan. Minimizing Energy for Wireless Web Access with Bounded Slowdown. In *MobiCom*, 2002.

[9] R. Krashinsky and H. Balakrishnan. Minimizing Energy for Wireless Web Access with Bounded Slowdown. *ACM Wireless Networks*, 11(1–2):135–148, January 2005.

[10] Chi-Chen Lee, Jui-Hung Yeh, and Jyh-Cheng Chen. Impact of inactivity timer on energy consumption in WCDMA and CDMA2000. In *IEEE Wireless Telecomm. Symp.(WTS)*, 2004.

[11] Hao Liu, Yaoxue Zhang, and Yuezhi Zhou. Tailtheft: leveraging the wasted time for saving energy in cellular communications. In *Proceedings of the sixth international workshop on MobiArch*, MobiArch '11, pages 31–36, New York, NY, USA, 2011. ACM.

[12] C. Monteleoni. Online learning of non-stationary sequences. In *AI Technical Report 2003-011, S.M. Thesis*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 2003.

[13] C. Monteleoni, H. Balakrishnan, N. Feamster, and T. Jaakkola. Managing the 802.11 energy/performance tradeoff with machine learning. Technical Report MIT-LCS-TR-971, MIT CSAIL, 2004.

[14] C. Monteleoni and T. Jaakkola. Online learning of non-stationary sequences. In *Neural Information Processing Systems 16*, Vancouver, Canada, December 2003.

[15] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. TOP: Tail Optimization Protocol For Celluar Radio Resource Allocation. In *ICNP*, 2010.

[16] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 321–334, New York, NY, USA, 2011. ACM.

[17] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Characterizing radio resource allocation for 3g networks. In Mark Allman, editor, *Internet Measurement Conference*, pages 137–150. ACM, 2010.

[18] T. Simunic, L. Benini, P. W. Glynn, and G. De Micheli. Dynamic power management for portable systems. In *MobiCom*, 2000.

[19] Qiang Xu, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Jeffrey Pang, and Shobha Venkataraman. Identifying diverse usage behaviors of smartphone apps. IMC '11, pages 329–344. ACM, November 2011.

[20] Jui-Hung Yeh, Jyh-Cheng Chen, and Chi-Chen Lee. Comparative Analysis of Energy-Saving Techniques in 3GPP and 3GPP2 Systems. *IEEE Trans. on Vehicular Technology*, 58(1):432 –448, January 2009.

[21] Jui-Hung Yeh, Chi-Chen Lee, and Jyh-Cheng Chen. Performance analysis of energy consumption in 3GPP networks. In *IEEE Wireless Telecomm. Symp. (WTS)*, 2004.