A General-Purpose, Energy-Efficient, and Context-Aware Acoustic Event Detection Platform for Mobile Devices

---

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Doctor of Philosophy

by

S M Shahriar Nirjon

August

2014

Abstract

Humans are extremely capable of remembering, recognizing, and acting upon hundreds of thousands of different types of acoustic events on a day-to-day basis. Decades of research on acoustic sensing have led to the creation of systems that now understand speech, recognizes the speaker, and finds a song. However, apart from speech, music, and some application specific sounds, the problem of recognizing varieties of general-purpose sounds that a mobile device encounters all the time has remained unsolved. The overarching goal of this research is to enable rapid development of mobile applications that recognize general-purpose acoustic events. As these applications are meant to run on a mobile device, the technical goals of this research include – energy-efficiency, communication-efficiency, leveraging the user contexts such as the location and position of the mobile device in order to improve the classification accuracy, and ease of application development.

With this goal in mind, we have built a general-purpose, energy-efficient, and context-aware acoustic event detection platform for mobile devices called – the *Auditeur* platform. Auditeur enables mobile application developers to have their application register for and get notified on a wide variety of acoustic events. Auditeur is backed by a cloud service to store crowd-contributed sound clips and to generate an energy-efficient and context-aware classification plan for the mobile device. When an acoustic event type has been registered, the mobile device instantiates the necessary acoustic processing modules and wires them together to dynamically form an acoustic processing pipeline in accordance with the classification plan. The mobile device then captures, processes, and classifies acoustic events locally and efficiently. Our analysis on user-contributed empirical data shows that Auditeur's energy-aware acoustic feature selection algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of the maximum achievable accuracy. We implement seven applications with Auditeur, and deploy them in real-world scenarios to demonstrate that Auditeur is versatile, $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events, compared to state-of-the-art techniques. We perform a user study involving 15 participants to demonstrate that even a novice programmer can implement the core logic of an interesting application with Auditeur in less than 30 minutes, using only $15 - 20$ lines of Java code.

Besides Auditeur, three other systems have been developed in this research which empower some aspects of the platform. First, we introduce *sMFCC*, which is an approximation to a well-known acoustic feature called the *Mel-*

frequency cepstral coefficient (MFCC). The motivation behind sMFCC is to enable faster extraction of MFCC features in applications that must sample the microphone at a very high rate and yet has to meet the real-time requirement. Second, we have developed the *MultiNets*, which is capable of switching wireless networking interfaces (WiFi and 3G) on a mobile device based on a predefined policy, such as saving energy during wireless data communication. The motivation behind MultiNets is to enable energy-efficient mobile-cloud communication in applications that must communicate quite frequently with a remote server over the Internet. Third, we have developed the *Musical-Heart*, which is a system that provides a biofeedback-based and context-aware music recommendation service to a mobile device. The motivation behind Musical-Heart is to build a convenient, non-invasive, personalized and low-cost wellness monitoring system that obtains heart rate and activity level information from a pair of specially designed earphones while a user is listening to the music on his mobile device.

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

AUTHOR

The dissertation has been read and approved by the examining committee:

John A. Stankovic

Advisor

Kamin Whitehouse

Alfred Weaver

John Lach

Stephen G. Wilson

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

August

2014

”And that man can have nothing but what he strives for.”
(53:39)

Acknowledgement

"All praise belongs to Allah, Lord of all the worlds." (1:2)

At first I express my gratitude to Almighty Allah for giving me the strength, patience, and the ability to complete this dissertation. As a human being, I could only try; and throughout my entire life, whatever I have ever achieved is nothing but His mercy and kindness.

The past six years of my life has been a wonderful journey. I was fortunate to have Jack as my advisor who, without any doubt, is the best. He gave me the full freedom to explore any idea that I had. He gave me the flexibility to choose my own work habits. He has always been a trusted friend and the first person to whom I would ask for advice on any matter.

I express my gratitude to my parents. Without their love and support, I could never have come this far. I also thank my elder brother, Nejhum, who has been the forerunner of my life. And my special thank goes to my beloved wife, Anwica, who has brought joy, fullness, and order to my life.

I have met some wonderful people during my PhD. I thank my internship mentors – Jie Liu and Angela Nicoara; my research collaborators – Kamin Whitehouse, Sang Son, Taejoon Park, Guobin (Jacky) Shen, Xiaofan (Fred) Jiang, Feng Zhao, Cheng-Hsin Hsu, Jatinder Singh, Bodhi Priyantha, Gerald Dejean, Yuzhe Jin, and Ted Hart; my coauthors and colleagues – Robert Dickerson, Enamul Hoque, Philip Asare, Chris Greenwood, Carlos Torres, Stefanie Zhou, Hee Jung Yoon, Ho-Kyeong Ra, Can Basaran, Qiang Li, Ben Zhang, Sirajum Munir, and Shan Lin; and two great undergrads – Dezhi Hong and Peeratham Techapalokul. I also thank my teachers in BUET, specially my B.Sc. and M.Sc. advisors – Prof Masud Hasan and Prof Monirul Islam.

I thank the Muslim community in Charlottesville and the Bangladeshi community in UVA for making me feel at home in this foreign land. I thank everyone in UVA and in Charlottesville, VA for being such a wonderful people.

Finally, I thank the people of Bangladesh, whose tax money has made it possible for me to receive my primary, secondary and undergraduate education. I look forward to act my part in building the future of my country.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Our day starts with the sound of an alarm clock, and throughout the day, we hear and respond to a wide variety of sounds. Over the course of evolution, humans have mastered the capability of remembering, recognizing, and acting upon hundreds of thousands of different types of acoustic events on a day-to-day basis. The aim of this research is to make a mobile device capable of doing the same – i.e. to recognize sounds that it captures with the on-board microphone and to deliver the recognized acoustic events to applications that are interested in them.

Decades of research on acoustic sensing have led to the creation of systems that now understand speech, supports voice activated search feature, recognizes the speaker, and finds a song. For example, in October 2011, Apple introduced an intelligent personal assistant called – *Siri* [1], which offers conversational interaction with many applications, including reminders, weather, stocks, messaging, email, calendar, contacts, notes, music, clocks, and web browser. Following that footstep, Google introduced their *OK Google* feature which allows the user to speak into his Android phone or tablet to do things like web search, get directions, and send messages. Web services and applications like Shazam [2] have been created, which provides a music identification service. However, apart from speech, music, and some application-specific sounds, the problem of recognizing varieties of general-purpose sounds that a mobile device encounters all the time has remained unsolved.

The overarching goal of this research is to enable rapid development of mobile applications that recognize general-purpose acoustic events on mobile devices. As these acoustic event recognizers are meant to run on a mobile device, the technical goals include – energy-efficiency, communication-efficiency, leveraging the user contexts such as the location and position of the mobile device in order to improve the classification accuracy, and ease of application development.

In this chapter, we present the motivation behind this research (Section 1.1), identify the core research challenges (Section 1.2), and then state our thesis statement and provide an overview of our solution (Section 1.4). These are followed by a list of contributions (Section 1.3) and some caveats (Section 1.5). Finally, we describe the organization of

this thesis (Section 1.6).

1.1 Motivation

If we think of all the applications that are in the app-store of various mobile platforms and perform any kind of acoustic processing, soon we will realize that these applications relate to mainly two types of sounds – speech and music. We find it a great missed opportunity that all mobile devices come with an acoustic sensor, i.e. the on-device microphone, and yet we do not see many applications that recognize and act upon varieties of sounds that we hear all the time around us. Imagine the possibilities that we could turn into realities, if we could have mobile applications that recognize various types of sounds. By recognizing the sounds that someone makes during his sleep, such as – yawning, snoring, coughing, bed-movement, and mumbling, we could create an application that measures his sleep quality. By recognizing the sounds of whining, baby talks, and baby crying – we could create a baby monitoring application for a busy mother. Or, we could create a nice social monitoring application that keeps track of how much a person is talking, laughing, crying, or yelling, and how many people and who he is talking to. The *lack of a general-purpose acoustic event classification service* is the primary reason why there are not so many applications that detect such general-purpose sounds on a mobile device.

Mobile application developers depend heavily on APIs provided to them by the mobile OS platform for common tasks, but when it lacks necessary functions for their application, it costs developers weeks of time for implementation. Beyond just time, many developers do not have the necessary technical background to implement these functions correctly or efficiently. Because of this, there is a growing trend where mobile applications are paired with numerous web services by other providers to get access to specialized or computationally demanding tasks. Microsoft's Hawaii [3], for example, supports path prediction, key-value storage, translator, relay, rendezvous, OCR, and speech-to-text services. Google provides services such as web search, maps, play, YouTube, cloud drive, and Gmail. Other providers give access to data sources such as weather, financial data, airline information, and parcel tracking. However, to the best of our knowledge, there is no web service that provides a general-purpose acoustic event classification service for mobile devices.

Mobile devices, such as smartphones and tablets, have begun to use their microphones for various acoustic processing tasks, but most of the applications are special purpose acoustic event detectors. Examples of voice and music based systems include: speaker identification [4], speech recognition [5], emotion and stress detection [6, 7], conversation and human behavior inference [8, 9], music recognition [2], music search and discovery [10], and music genre classification. There are other types of applications that fall into the non-voice, non-music category, such as a cough detector [11], a heart beat counter [12], and logical location inference [13, 14]. These examples are limited in number and acoustic sensing is often one of multiple sensing modalities in these works. There are existing works [15, 16] that consider

multiple types of acoustic events. SoundSense [15] considers speech, music and ambient sound, and provides a mechanism to label clusters of ambient sounds to extend the set of classes. Jigsaw [16] considers speech and sounds related to activities of daily living. These systems still lack in providing tools and mechanisms to easily extend detection capabilities and effectively support development process. They are limited for general purpose acoustic event detection service for mobile devices and in exploiting the promising opportunities given by already prevailing mobile devices and tablets equipped with microphones.

To address this need, we have built a general-purpose acoustic event detection platform called the *Auditeur*. The basic idea of Auditeur is that – mobile device users all over the world will record and upload short-duration sound clips to the cloud, and a cloud-service will automatically create classifiers for those sounds. These classifiers can then be downloaded and run on the device to recognize future occurrences of that specific type of sound. Aside from the ability to dynamically create classifiers to recognize a wide variety of sounds, two aspects of Auditeur make it unique and ideal for mobile platforms. First, the classifiers created by Auditeur are energy-aware. An input to the classifier creation service is an energy bound, and based on that bound, Auditeur weaves the best acoustic processing pipeline that ensures that the classifiers run within the energy bound and yet achieve a very high accuracy. Second, the classifiers created by Auditeur are context-aware. Auditeur is capable of automatically determining the user-contexts, such as the location of the user and the position of the mobile device with respect to human body, and dynamically loads a different classifier whenever the user context changes. Auditeur also provides an easy-to-use API for the mobile application developers to foster the creation of applications.

Besides Auditeur, three other systems have been developed in this research which empowers some aspects of the Auditeur platform. First, we have developed the *sMFCC* acoustic feature, which is an approximation to a well-known acoustic feature called the *Mel-Frequency Cepstral Coefficient (MFCC)*. The motivation behind which is to enable faster extraction of MFCC features in applications that must sample the microphone at a very high rate and yet has to meet the real-time requirement. Second, we have developed the MultiNets, which is capable of switching wireless networking interfaces (WiFi and 3G) on a mobile device based on a predefined policy, such as saving energy during wireless data communication. The motivation behind MultiNets is to enable energy-efficient mobile-cloud communication in applications that must communicate quite frequently with a remote server over the Internet. Third, we have developed the Musical-Heart, which is a system that provides a biofeedback-based and context-aware music recommendation service to a mobile device. The motivation behind Musical-Heart is to build a convenient, non-invasive, personalized and low-cost wellness monitoring system that obtains heart rate and activity level information from a pair of specially designed earphones while a user is listening to the music on his mobile device. Timeline wise, Musical-Heart was developed as a standalone application prior to the creation of the Auditeur platform. However, we have shown that Musical-Heart's heart rate detection can also be implemented with Auditeur.

1.2 Technical Challenges

This thesis addresses a number of key technical challenges in building a general-purpose, energy-efficient, and context-aware acoustic event detection service platform for mobile devices.

First, applications are diverse in nature. Different applications require to recognize a different set of sounds, which in turn requires different types of preprocessing, acoustic feature extractors, and classifiers. Building *a generic platform that is capable of automatically generating many applications that recognize a wide variety of acoustic events* is therefore challenging.

Second, battery-life is a scarce resource for mobile devices. Building energy-efficient applications is in general a hard problem, and acoustic processing adds further complexity on top of it. Some applications may want to continuously sense the microphone for a very long period. Without carefully planning ahead of time, it is impossible to guarantee that an application will not drain the battery in the middle of its expected lifetime. One major challenge is therefore *to design an acoustic event classifier which is accurate yet runs under a given energy budget*.

Third, acoustic event classification is dependent upon the user contexts. A classifier that works perfectly indoors does not necessarily perform accurately when it runs outdoors. The types of possible acoustic events also change with user's location. Distance from the mobile device to the source of the sound also matters. For these reasons, we need to consider the user context when classifying acoustic events in order to achieve a higher accuracy. On-board sensors on a mobile device provides us with a rich set of user contexts, such as the location of the user and the position of the mobile device with respect to the body. One challenge is therefore to *leverage the user contexts in order to improve the accuracy of acoustic event classification*.

Fourth, in order to facilitate a rapid development of mobile applications, we provide APIs to the developers. An important challenge is *how to design a generic API that is easy to learn, easy to use, and expressive enough to specify different acoustic event types in the same manner.*

Fifth, different acoustic signals have different properties. Speech signals, for example, are extremely sparse in the frequency domain. This sparseness in speech allows us to design an efficient frequency domain feature extraction module that runs several times faster. We demonstrate *how the sparseness in speech signals is exploited to make the on-device acoustic processing faster*.

Sixth, there are some applications that must communicate to one or more remote servers over the Internet during its lifetime. A mobile device communicates to the cloud using either a WiFi or a cellular network. Each of these interfaces has its pros and cons. By selecting the right network interface to communicate with a remote server, it is possible to save energy and increase throughput. We demonstrate *how to make dynamic choices on which wireless interface to use for an efficient mobile-cloud communication*.

Seventh, as an illustration of a special-purpose acoustic event detection application, we have developed the Musical-Heart system. This system comes with several technical challenges of its own, such as – detecting heartbeats from acoustic signals collected from the ear, detecting the activity level of the user from accelerometer readings obtained from the ear, and recommending music based on the heart rate and the activity level of a person in order to achieve a target heart rate goal.

1.3 Thesis Statement and Solution Overview

1.3.1 Thesis Statement

This dissertation investigates the following hypothesis:

It is possible to detect and classify acoustic events on a mobile device, which is – more generic, highly-accurate, more energy-efficient, adaptive to user context, and easier to program – when compared to state-of-the-art special-purpose acoustic event detection systems that use on-device or cloud-assisted classifiers.

1.3.2 Solution Overview

In order to prove our hypothesis, we have built a total of four systems: Auditeur, sMFCC, MultiNets, and Musical-Heart. A conceptual integration of all four systems are shown in Figure 1.1.

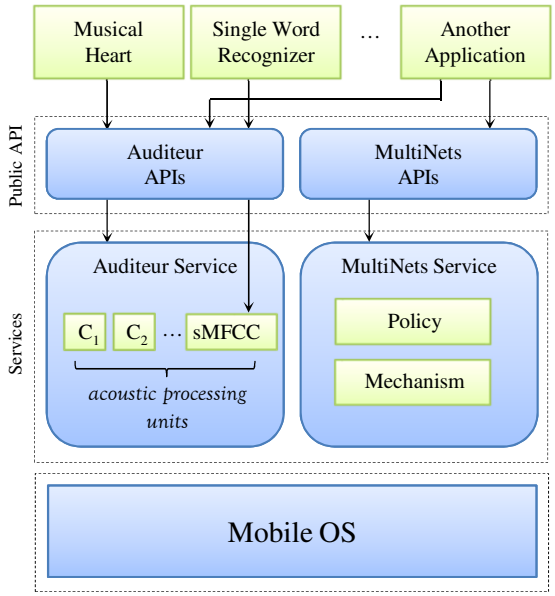


Figure 1.1: Conceptual integration of all four systems: Auditeur, sMFCC module, MultiNets, and Musical-Heart.

At the bottom layer of the figure, we have the mobile operating system which is the Android OS in our case. Both Auditeur and MultiNets run as a system service on top of the mobile OS. Auditeur offers services related to sound recognition, whereas MultiNets offers services related to wireless network interface switching. The sMFCC module is a part of the Auditeur service like other acoustic processing units inside it. The public API layer provides APIs to the applications with which they access the services offered by Auditeur and MultiNets. With Auditeur API, an application registers for and gets notified on various types of acoustic events. With MultiNets API, an application selects what policy (e.g., energy saving, or high throughput) should be applied during mobile-cloud communications.

The top layer consists of three running applications. The fist application is the Musical-Heart system which uses the Auditeur API. Auditeur notifies Musical-Heart application whenever it detects a heartbeat event in the streaming acoustic data from a special-purpose earphone hardware. The earphone is called the Septimu, which we have built as part of the Musical-Heart system.

The second application also uses the Auditeur API, but its specialty is that, it uses the sMFCC module besides other acoustic processing units in Auditeur. The application is a simple single spoken word recognizer that recognizes words from a fixed vocabulary. By using the sMFCC module instead of the regular MFCC module, this application achieves several orders of magnitude speedup in acoustic event detection with the cost of a slight reduction in accuracy.

The third application is a conceptual one which uses both the Auditeur and the MultiNets APIs. An example of such as application could be an integrated asthma monitoring system that assesses the severity of asthmatic wheezing and crackling sounds using the Auditeur service and talks to a remote server to obtain information such as the weather condition, pollen level, pollution level, and air quality, using the MultiNets service. Building such an application is currently in progress, however, the system is not part of this thesis.

1.4 Contributions

The contributions of this dissertation are the following:

- Auditeur– a versatile, general-purpose, flexible, extensible, context-aware, and efficient end-to-end acoustic event detection platform for mobile devices, backed by the cloud. The system comes with a sound recognition service on the mobile device, which is capable of dynamically wiring acoustic processing units, and running an energy- and context-aware classification plan on the phone, and provides APIs to register for and get notified on specific sound events.

- A collection of crowd-contributed, admission controlled, and contextually-tagged short sound clips in the cloud available to the developers, and an API to upload and manage them. The collection contains over 5000 tagged

sound clips which are collected by 35 volunteers, over a nine month period, from three different regions (Virginia, Toronto, and Beijing).

- An acoustic feature selection algorithm that automatically generates energy-aware acoustic event detection plans for a mobile device, given the types of sounds that an application wants to recognize and an energy budget. Our empirical evaluation shows that the algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of its maximum achievable accuracy.

- We implement seven applications with Auditeur, and deploy them in real-world scenarios to demonstrate that Auditeur is versatile, $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events, compared to state-of-the-art techniques. A user study demonstrates that novice programmers can implement the core logic of interesting applications with Auditeur in less than 30 mins and using only $15 - 20$ lines of Java code.

- sMFCC – an algorithm, inspired by [56, 57], which enables the extraction of frequency domain acoustic features inside a mobile device in real-time, without requiring any support from a remote server even when the sampling rate is as high as 44.1 KHz. We implement a simple spoken word recognition application using both MFCC and sMFCC features, show that sMFCC is expected to be upto 5.84 times faster and its accuracy is within $1.1\% - 3.9\%$ of that of MFCC, and determine the conditions under which sMFCC runs in real-time.

- We conduct a three months long empirical study and summarize the TCP characteristics in Android smartphones, complementing a similar study with iPhone users in [17, 18]. We devise a switching technique which is client-based, transparent to applications, and does not require any protocol changes.

- MultiNets– which is to the best of our knowledge, the first complete system of this kind and demonstrate its performance in a real-world scenario. MultiNets outperforms the state-of-the-art Android system either by saving up to 33.75% energy, or achieving near-optimal offloading, or achieving near-optimal throughput while substantially reducing TCP interruptions due to switching.

- We design and implement three switching policies for MultiNets. Our analysis on usage data collected from real mobile device users shows that with switching, we can save 27.4% of the energy, offload 79.82% data traffic, or achieve 7 times more throughput on average.

- *Septimu*, the first wearable, programmable hardware platform designed around low-cost and small form-factor IMUs and microphone sensors that are embedded into conventional earphones, and communicate to the phone via the audio jack and Bluetooth.

- *Musical-Heart*, a complete sensing system that monitors the user's heart rate and activity level – passively and without interrupting the regular usage of the phone – while the user is listening to music, and recommends songs based on the history of heart's response, activity level, desired heart rate and social collaboration.
- We devise three novel algorithms in Musical-Heart: (1) a threshold free, noise resistant, accurate, and real-time heart rate measurement algorithm that detects heart beats from a mixture of acoustic signals from the earbuds, (2) a simple yet highly accurate, person-specific, 3-class activity level detection algorithm that exploits accelerometer readings from the earbuds, and (3) a PI-controller that recommends music to the user based on the past history of responses to different music and helps maintain the target heart rate at different activity levels.
- We perform a pilot study by collecting ground truth data of heart rates, and summarizing it to show the effect of music on heart rate at various activity levels. The dataset is further used to show that the detected heart rate is $75\% - 85\%$ correlated to the ground truth, with an average error of 7.5 BPM. The accuracy of the person-specific, 3-class activity level detector is on average 96.8%, where these activity levels are separated based on their differing impacts on heart rate.
- We demonstrate the practicality of Musical-Heart by deploying it in two real world scenarios, and show that Musical-Heart helps the user in achieving a desired heart rate intensity with an average error of less than 12.2%, and the quality of recommendations improves over time.

1.5 Caveats

The systems and tools that are built during this research come with a few caveats:

- Auditeur is capable of detecting sound events that are recognizable from short-term time and frequency domain features. Recognizing sounds that depend heavily on temporal variations, such as long spoken sentences, are not suitable for it. Especially, speech recognition is a complex and well-studied problem having several well-known solutions. Hence, we leave it out of the scope of Auditeur.
- The acoustic features and classifiers used in Auditeur are shown to recognize a wide variety of sounds, but yet no such list is ever exhaustive. It is possible that for a new type of acoustic classification problem, we may have to incorporate new acoustic processing modules into the system in order for Auditeur to be able to detect that sound. Auditeur is an extensible system where addition of a new feature extractor or a new classifier unit is easy. Such extensions do not require any change to the framework.
- The scalability of the services running on the cloud is not addressed in this thesis as this is by itself a well-formed research problem. Our implementation of Auditeur is a proof of concept and runs on an Amazon EC2 instance.

In the future, we plan to move our server to Google App Engine which provides automatic scaling of applications without requiring us to manage the machines by ourselves.

- The general applicability of sMFCC acoustic feature is not fully explored in this research. We only have studied speech signals and shown that sMFCC is applicable to speech. In the future, we plan to explore the applicability of sMFCC on sounds other than just speech and make the choice between MFCC and sMFCC automatic in Auditeur.

- The absolute values of the measured energy consumption of Auditeur, MultiNets, and Musical-Heart are dependent on the model of the mobile device as well as the operating system. However, when comparing energy consumptions of two systems, we ran both systems on the same mobile device and repeated an experiment multiple times so that the comparison is fair.

- We perform an empirical study which shows that the TCP sessions on a mobile device are short-lived – which is the basis of uninterrupted interface switching mechanism in MultiNets. In the future mobile systems, this assumption may not be true and applications running on MultiNets will encounter interruptions during switching. It is thus advised to use the MultiNets APIs to create short-lived TCP sessions so that the interruptions do not happen.

- The pilot study that we conducted in Musical-Heart to model the relationship between music and heart rate involves 35 participants and the full system was tested on 4 volunteers only. This only provides us with an existential proof of the concept of controlling heart rate with music. The generalizability of the system requires further deployment and investigation. In the future, we plan to make the Musical-Heart service public, so that we can have more data to support our claim.

1.6 Organization of the Dissertation

The rest of the dissertation is organized as follows:

- Chapter 2 presents the state-of-the-art in technologies related to Auditeur, sMFCC, MultiNets, and Musical-Heart systems that are developed in this thesis.

- Chapter 3 describes some basic terminology that are used in the later chapters.

- Chapter 4 provides an overview of the mobile-cloud platform by introducing Auditeur, providing an overview of its workflow, and describing some key properties of the system.

- Chapter 5 describes the implementation of the Auditeur platform by providing the implementation details of the on-device processing components as well as the cloud services.
- Chapter 6 provides an in-depth evaluation of the Auditeur platform.
- Chapter 7 describes the motivation, algorithm, and evalution of the sparse MFCC (sMFCC) feature.
- Chapter 8 describes MultiNets's policy oriented dynamic network interface switching algorithm and its evalution.
- Chapter 9 describes the details of the Musical-Heart system – including its hardware, heart rate measurement algorithm, activity level detection algorithm, biofeedback-based music recommendation algorithm, and an in-depth evaluation of the system.
- Chapter 10 concludes the thesis by summarizing the contributions and describing the future work.

Chapter 2

Related Work

This chapter presents the state-of-the-art in technologies related to Auditeur, sMFCC, MultiNets, and Musical-Heart systems that are developed in this thesis. The chapter covers existing systems that detect special types of acoustic events (Section 2.1), examples of web services that are commonly used by mobile application developers (Section 2.2), resource aware sensor data classification systems which are related to Auditeur's energy-aware classification algorithm (Section 2.3), approaches in switching network interfaces that are related to MultiNets (Section 2.4), works that exploit sparseness in data (Section 2.5), various techniques for heart rate detection on a mobile device (Section 2.6), techniques that detect activity context of a mobile device user (Section 2.7), and existing approaches to automatic music recommendation (Section 2.7) which are related to Musical-Heart's biofeedback-based music recommendation system.

2.1 Special-Purpose Acoustic Event Detection

Acoustic sensing on mobile devices has been used in many works. Examples of voice and music sensing applications are, speaker identification [4], speech recognition [5], emotion and stress detection [6, 7], conversation and human behavior inference [8, 9], music recognition, search and discovery [2, 10]. There are some limited number of applications that consider other types of sounds. MusicalHeart [12] uses a special-purpose microphone to count heart beats, [11] counts coughs, Nericel [19] detects horns, Ear-phone [20] monitors noise pollution, SurroundSense [13] and CSP [14] infer logical location. All these systems detect only one specific type of sound. SoundSense [15] distinguishes voice from music and noise, and clusters other sounds. But unlike Auditeur, none of these systems aim at solving the general purpose acoustic event detection problem. Some works are technically similar to Auditeur, but solve different problems. TagSense [21] senses people, activity and context in a picture, and creates tags on-the-fly, whereas Auditeur involves people in the sound tagging process, but provides automated tag suggestions. Sphinx [5] uses generic processing units which are similar to the APUs in Auditeur, but they have a simpler pipeline, highly tuned to solve the speech recognition

problem. Pickle [22] creates privacy preserving classifiers on the cloud whereas we preserve privacy by providing the user with private spaces, and classifiers in Auditeur are trained for energy efficiency.

2.2 Web Services for Mobile Devices

There are several web services available for mobile application developers. Hawaii [3] provides a social mobile sharing (SMASH) a service for rapid prototyping of social computing applications, a service to predict a user's destination using the current route, a key-value storage for application-wide state information, a text translation service, providing relay points in the cloud for applications' communication, an optical character recognition service (OCR), and an English speech-to-text service. But Hawaii does not have an acoustic event detection service. Google provides services such as: web search, Google maps, application store (Google play), video streaming (YouTube), Google cloud drive, and an email service (Gmail). Other providers give access to data sources such as weather, financial data, airline information, and parcel location tracking. Unlike Auditeur, none of these provide a sound recognition service and API to the developers.

2.3 Resource Aware Classification

There are several works that deal with resource aware sensor data classification on mobile devices. Kobe [23] is a tool to generate energy and latency aware classifiers, but unlike Auditeur, it requires uninterrupted mobile-cloud communication for adaptive code-offloading (as it is required in systems like MAUI [24] and Odessa [25]), it is not sensitive to user context, and their set of features is predetermined and fixed by the developer. Orchestrator [26] enables multiple applications to effectively share resources (e.g. sensors) whose availability changes dynamically. The optimum plan selection problem in Orchestrator is similar to the acoustic feature selection problem in Auditeur. However, unlike Auditeur, their selection algorithm is not scalable as they enumerate an exponential number of plans while Auditeur uses a polynomial time algorithm to eliminate the need for enumeration of exponential number of subsets of features and thus limits the total number of plans to consider. [27] proposes a heuristic algorithm to select a subset of sensors and their tolerance levels so that they can infer multiple contexts in an energy efficient manner. Feature selection is a problem studied by many; [28] provides a survey. WEKA [29] implements several of these attribute selection methods. But, our problem is different as our selection criterion is to minimize energy consumption, not to minimize the number of features. Like Auditeur, symmetrical uncertainty has been used in [30] as a goodness measure.

2.4 Network Interface Switching

Switching among multiple network interfaces of a mobile device has been considered in the literature. New protocols in various layers have been designed to support switching among access networks. Wang et al. [31] propose a rate control algorithm for multiple access networks, which needs to be integrated into application-layer protocols. Kim and Copeland [32], and Wu et al. [33] propose TCP variants that result in better performance by switching among access networks. Mobile IP [34] uses foreign/home agents to forward network traffic from/to a smartphone that move among access networks, but incurs additional network latency. Mobile IP v6 [35] uses optimized routes for lower network latency, but it still relies on deploying foreign/home agents for mobility management. In contrast to MultiNets, widely deploying TCP variants and mobile IP agents in the Internet incur tremendous costs and burden, and may take years to be done.

Gateways between mobile devices and the Internet can be used for accessing multiple network interfaces. Balasubramanian et al. [36] design a system to reduce the network traffic over cellular networks by transmitting delay tolerant data over WiFi and real-time data over cellular networks. Sharma et al. [37] propose a system that uses gateways to aggregate network resources from multiple access networks among several collaborative smartphones. Armstrong et al. [38] use a proxy to notify the phone via SMS about content updates and suggests interface to use. Unlike MultiNets, gateway solutions require deploying expensive gateways, incurs additional network latency, and may need users to configure the proxy settings in applications.

A master/slave solution [39] chooses an always-connected access network as the master network, and uses other access networks as slave networks for opportunistic routing. Different from MultiNets, deploying master/slave solutions requires complete control over multiple access networks, which is difficult due to business reasons. Higgins et al. [40] propose intentional networking where applications provide hints to the system and system chooses the best interface opportunistically. In contrast, MultiNets is completely transparent to the existing aplications. Rahmati et al. [18] demonstrate the feasibility of TCP flow migration on iPhones, but they do not address or rigorously quantify the policies and different benefits of switching interfaces.

2.5 Exploiting Sparseness in Speech

Sparseness in data is exploited in many application domains such as learning decision trees [41], compressed sensing [42], analysis of Boolean functions [43], large scale time series data analysis [44], similarity search [45], and homogeneous multi-scale problems [46]. In our work, we perform an empirical study on the sparseness in speech data collected on smartphones with the goal of exploiting the sparseness to expedite the MFCC feature computation. MFCC is a widely used feature for analyzing acoustic signals [47, 48, 49, 50, 51]. MFCC is used for spoken word

recognition [47], voice recognition [48], speaker identification [49], music modeling [50], and music similarity measure [51]. [52] presents a nice comparison of different MFCC implementations. However, in our work, we introduce a sparse MFCC (sMFCC) which is a sparse approximation of MFCC and is efficient to extract.

Comparison of several speech recognition techniques on mobile devices is described in [53]. [4] performs speaker identification, [15] classifies sound into voice, music or ambient sound, and [9] classifies conversion. But all of these applications limit their sampling rate to its minimum. [12] uses a high sampling rate to extract heart beats from acoustic signals, but the system is not fully real-time. In this dissertation, we analyze the feasibility of using sMFCC features that is expected to run in real-time and at higher data rates.

2.6 Heart Rate Detection on Mobile Devices

The smartphones applications that measure heart rate use three basic principles: camera, microphone, and accelerometer based. Camera based applications (Instant Heart Rate, PulsePhone) detect pulses by measuring the changes in the intensity of light passing through the finger. Microphone based applications (Heart Monitor, iRunXtream) require the user to hold the microphone directly over heart, or neck, or wrist to detect beats. Accelerometer bases techniques (iHeart) are indirect, in which, the user measure his pulse with one hand (e.g. from neck) and taps or shakes the phone with other hand in the rhythm of the heart beats. Our approach in Musical-Heart is microphone based, but the difference is, in all of the existing applications, the heart rate monitoring requires active engagement of the user, whereas Musical-Heart obtain heart beats from the ear without interrupting the user's music listening activity. Other approaches that detect heart beats from the ear [54, 55] use a combination of infrared LED and accelerometers.

2.7 Inferring Activity Context on Mobile Devices

Several works are related to our activity detection algorithm. [56] concludes that it is practical to attribute various activities into different categories which is similar to the activity levels in Musical-Heart. [57] describes an accelerometer based activity classification algorithm for determining whether or not the user is riding in a vehicle, or in a bus, or another vehicle for cooperative transit tracking. In our work, we only require to detect whether someone is indoors vs. outdoors, and his current speed. [58] presents a mobile sensing platform to recognize various activities including walking, running and cycling. [59] presents a wearable platform for motion analysis of patients being treated for neuromotor disorders. [60] detects user's caloric expenditure via sensor data from a mobile phone worn on the hip. But these works require the user to wear sensors at specific positions of the body. In our case, the user naturally wears the earphone in the ear. [61] performs activity recognition from user-annotated acceleration data, which is different than

our unsupervised learning. [62] presents a high performance wireless platform to capture human body motion. In our system, we do not require a high speed communication to the server.

2.8 Music Recommendation Systems

Music has been shown to improve sleep quality [63, 64], cope with stress [65], improve performance [66, 67], influence brain activity [68], and increase motor coordination [69]. However, we only consider the response of the heart to music. Studies have shown that exercise intensity and tempo are correlated linearly [70], and relaxing music (e.g., Bach, Vivaldi, Mozart) result in reduction of heart rate and its variability [71]. We perform a similar study, but the difference is ours involves smartphone users who are mobile and whose activity levels change.

Existing approaches for automatic music recommendation are: using one or more seed songs [72, 73], retrieve similar music by matching the tempo [74], and learning the habit (e.g., music genre, preferences, or time of day) of the user [75, 76]. None of these consider any physiological effects of music on the user. Physiological information, such as gait and heart rate, have been considered in [77, 78, 79, 80]. But the downside of these approaches are, first, the user has to use a separate device (e.g. ECG or pulse oximeter) which is an inconvenience, second, the music recommendation is based on an overly simplistic and general rule of thumbs, i.e., to suggest music with a matching tempo, which is not personalized, and third, the recommendation is not situation aware and mostly targeted to a single scenario of jogging.

2.9 Summary

This chapter presents the state-of-the-art in technologies related to Auditeur, sMFCC, MultiNets, and Musical-Heart systems that are developed in this thesis. We have covered some systems and algorithms that are related to Auditeur's acoustic event detection and energy aware classification, MultiNets's network interface switching technique, and Musical-Heart's heart rate detection and biofeedback-based music recommendation techniques.

Chapter 3

Background and Terminologies

This chapter describes some basic terminologies that are used in the later chapters. The first four sections (Sections 3.1-3.4) of this chapter define acoustic events, tagged soundlets and their types, and the mobility contexts that are used in the Auditeur platform. The next two sections (Sections 3.5- 3.6) describe the MFCC acoustic feature and the sparse FFT which are required to understand the sMFCC algorithm in Chapter 7. The last three sections (Sections 3.7- 3.9) explain basic concepts, such as the ECG, heart beats, pulse, tempo, pitch, energy, and heart rate training zones, which are used in Chapter 9 where we describe the MultiNets system.

3.1 Acoustic Events

We define an *acoustic event* as a short-duration ($1 - 3$s) sound that we can name. Examples of acoustic events include – speech; music; sound of a machine such as a printer, a telephone, or an air conditioner; sound of a vehicle such as a car, a bus, or an airplane; sound of human emotions such as laughter, yelling, or crying; etc.

Suppose, in an audio clip of 60 seconds, a person is talking for 15 seconds, laughing for 3 seconds, and the rest of the clip contains silence. Considering each acoustic event having a duration of a second, the clip contains a total of 15 talking events, 3 laughter events, and 42 silence events.

The duration of an acoustic event is called the *window size* in Auditeur. It denotes the amount of audio data that are taken for classification as a single event. The developer of an application decides the window size based on the requirement of an application. The example in the previous paragraph has a window size of 1 second which is the default window size in Auditeur.

3.2 Tagged Soundlets

A *tagged soundlet* is a short-duration ($3 - 30$s) audio clip, recorded on a mobile device along with two types of contextual information: user given tags and the mobile device generated context. The logical tags associated with a soundlet are the user given identifiers that describe its content and the surrounding environmental context. Examples of tags are in Table 6.2. Other than these two types of tags, we also keep a record of some meta information about the sound clip. The list of Meta information contains – the audio file information (e.g., audio format, sampling rate, and PCM data), and recording information (e.g., user ID, device model, date-time, GPS location, and environmental noise level).

3.2.1 Content and Container Tags

While the user chooses the tags that he wants to use to describe a soundlet, we require him to enlist two kinds of tags: *content*, and *container* tags. These two types of tags are used during the upload of a soundlet to the cloud. The content tags describe what the sound *is*, and the container tags describe the background which contains the sound. For example, a $15s$ recording of Alice's voice at her office should have {`voice, female, Alice`} in the list of content tags, and {`office`} in the list of container tags.

3.2.2 Lookfor and Within Tags

Two types of tags are used to make a request for a classifier to the cloud from the mobile device. The *lookfor* tags describe a class of soundlets that the user wants to recognize, and the *within* tags describe the class of soundlets that represents the universe of sounds, i.e. the set of sounds that might be present in the environment along with the sound that he wants to recognize. For example, when creating a training set for a classifier that detects Alice's voice at her office: the lookfor tags should have {`voice, Alice`} on the list, the within tags should include{`voice, printer, phone`}, and the container should be {`office`} as before.

3.3 Public and Private Soundlets

The space of soundlets is logically partitioned into two subspaces from a user's point of view – *public* and *private*. A *user* of Auditeur is anyone who has an account in Auditeur. Typically, the user is an application developer, who uses the public space by default, but may use a private space with proper permissions from an end-user.

3.3.1 Public Soundlets

The public space contains shared soundlets, contributed by all users of Auditeur. This is visible to everyone, and does not require any authentication to access. However, there are two constraints that must be met by each soundlet.

There is a predefined fixed set of tags that can be attached to public soundlets. The user cannot create, modify or delete any tag. The set of tags is maintained by the system admin of Auditeur. Although, at present, there are over 1000 different tags, we admit that no fixed set is ever sufficient to cover the enormous possibilities of tags. However, limiting the tags in the public space keeps the space manageable and provides a way to ensure the confidence in sound recognition.

Every public soundlet goes through a sanity check to ensure that it is not an outlier with respect to other soundlets with the same tags. Thus, a public soundlet is not immediately made visible to everyone until it passes an outlier detection test. This is to ensure that no attacker can mess up the public space by filling it with soundlets with non-representative tags. A service running on the cloud automatically detects non-conforming soundlets using a distance based outlier detection technique [81] which is described in Section 5.2.4.

3.3.2 Private Soundlets

Private soundlets are user specific and are not shared. The user has to access his private space using proper authentications. Unlike public soundlets, the private ones are not limited by the tag and sanity constraints, i.e. a user can upload any sound he likes into his private space and attach whatever tag he wants to describe it. However, the list of content and container tags has to be the same (in number of tags and the spelling of tags) in order for two soundlets to be treated as belonging to the same class.

We keep provision for private space in our design in order to complement the public space which is limited by the choices of tags and has a high degree of sanity requirement. The private space is more flexible, tailored to the specific needs of a user, and if properly used, an infinite number of possible sounds can be stored and recognized by our system making it highly extensible.

3.4 Mobility Contexts

The mobility context refers to additional information about the soundlets which may or may not be conveyed by the tags. These are similar to the tags in the manner they are used by the recognition algorithm, but they are different in the manner they are generated. The mobility context is either automatically generated by an algorithm using information from the on-board sensors, or they are assumed. The developer of the application specifies whether an automatically generated context or an assumed context – whichever is more suitable to the application – should be used.

Aside from basic audio information such as the sampling rate, encoding, duration, and timestamp, Auditeur generates three types of contexts, which are: (1) the location of the user, (2) the position of the device with respect to body, and (3) the environmental noise level. Section 5.1.2 provides an elaboration of mobility contexts. Note that, the developer of an application can always override the context generation process. We keep this provision in Auditeur as context generation has its overhead and in many cases an application is used only in a few specific presumed contexts. For example, an application that detects vehicle sounds can assume that the location is outdoors instead of periodically getting it computed.

3.5 Mel-Frequency Cepstral Coefficients

The Mel-Frequency Cestrum Coefficients (MFCC) is one of the most popular short-term, frequency domain acoustic features of speech signals [82]. The MFCC have been widely used in speech analysis because of their compact representation (typically, each speech frame is represented by a 39-element vector), close resemblance to how human ear responds to different sound frequencies, and their less susceptibility to environmental noise.

The MFCC feature extraction starts with the estimation of the power spectrum which is obtained by taking the square of the absolute values of the FFT coefficients. However, prior to computing the power spectrum, typically each speech frame passes through a pre-emphasis filter which is followed by a windowing process. The log-power spectrum is used instead of the power-spectrum as human hearing works in decibel scales. The log-power spectrum then goes through a filtering process. A filter-bank with around 20 triangular band-pass filters is applied to reduce the dimensionality. These filters follow a Mel-scale which is linear up to 1 KHz and logarithmic for the larger frequencies – resembling human hearing. Finally, a discrete cosine transform (DCT) is performed to compress the information and to make the vectors uncorrelated. Only the lower-order coefficients (typically 13) are used and the rest are discarded. The 13-MFCCs plus the deltas and double deltas constitute a 39-element feature vector for each speech frame.

3.6 Sparse Fast Fourier Transform

The discrete Fourier transform (DFT) is one of the most significant algorithms in the digital signal processing domain. The fastest algorithm that computes DFT of an n-dimensional signal is $O(n \log n)$-time. However, a recent algorithm [83, 84] coined sparse FFT (sFFT) has broken this bound for a special case of DFT where the signals are sparse. A signal is considered sparse if most of its Fourier coefficients are small or very close to zero. For a small number k of non-zero Fourier coefficients, sFFT computes the Fourier transformation in $O(k \log n)$-time.

The basic idea of sFFT is to hash the Fourier coefficients into a small number of bins. The signal being sparse in the frequency domain, it is less likely that each bin will have more than one large coefficient. The binning process is

done in $O(B \log n)$ where B is the number of bins – by at first permuting the time-domain signals and then filtering them. Each bin at this point ideally has only one large Fourier coefficient, and only such 'lonely' coefficients are taken into the solution. The process is repeated $\Theta(\log k)$ times, each time varying the bin size $B = k/2^r$ where the integer $r \in [0, \log k]$, so that all k coefficients are obtained. The overall running time of the algorithm is dominated by the first iteration, and hence the time complexity is $O(k \log n)$. The algorithm is probabilistic, but for exact k-sparse signals (i.e. at most k of the coefficients are significant), the algorithm is optimal, as long as $k = n^{\Omega(1)}$.

3.7 ECG, Heart Beat and Pulse

The electrocardiogram (*ECG*) is the most reliable tool for assessing the condition of the heart and measuring the heart rate. Getting an ECG reading is a non-invasive procedure in which a set of electrodes is attached to specific regions of a body to record the electrical activity of the heart.

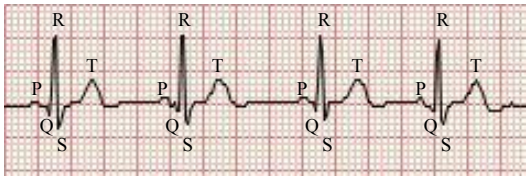


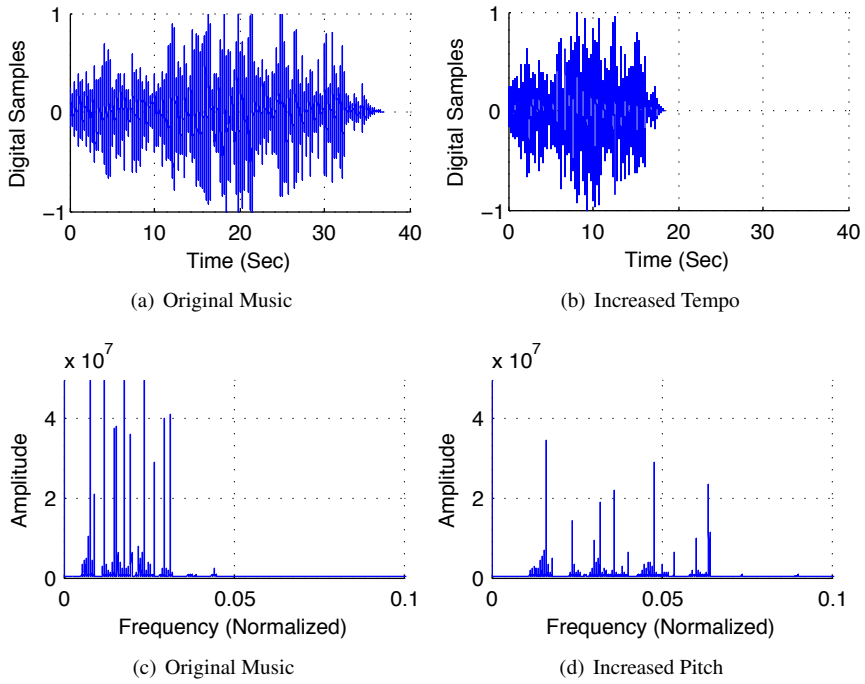
Figure 3.1: The interval between consecutive R waves determines the heart rate.

Figure 3.1 shows a portion of an ECG record comprising of four consecutive *heartbeats*. The various waves (or, peaks) that are seen on a heartbeat are historically denoted by the letters: P, Q, R, S, and T. R waves are more visible than others in an ECG record. Hence, the instantaneous heart rate is usually obtained by taking the inverse of the time interval between two consecutive R waves. Since the instantaneous heart rate varies with respiration, the average of $30 - 60$ instantaneous readings is taken to obtain the average heart rate, expressed in beats per minute (BPM). While the heart rate directly refers to the frequency of the heart's beating, the *pulse*, on the other hand, is the contraction and expansion of an artery due to the beating of the heart. The heart rate and the pulse rate are usually the same – unless for some reason, blood finds it difficult to pass through the arteries. Pulses can be felt on different parts of the body, e.g., neck, wrist, and ears.

3.8 Tempo, Pitch and Energy

As with human heartbeats, the rhythmic, repetitive and recognizable pulsating sound in music, often produced by percussion instruments (e.g., drums), is called the *beat*. The *tempo* of a song is the indicator of the number of beats played per unit time. The tempo of a music is similar to heart rate in human and has the same unit, i.e. beats per minute

(BPM). The *pitch* is related to the frequency of the sound wave, and is determined by how quickly the sound wave is making the air vibrate. The *energy* of a signal is computed simply by taking the root average of the square of the amplitude, called root-mean-square (RMS) energy.



(a) Original Music (b) Increased Tempo

(c) Original Music (d) Increased Pitch

Figure 3.2: Illustration of tempo and pitch of music.

Figures 3.2(a) and 3.2(c) show the time domain and frequency domain characteristics of a 40-second slice of music. Figure 3.2(b) shows the effect of increasing the tempo and Figure 3.2(d) shows the effect of increasing the pitch of the music. Increasing the tempo squeezes the samples into a shorter duration, and increasing the pitch shifts the power spectrum towards a higher frequency.

3.9 Heart Rate Training Zones

Depending on the amount of effort (*intensity*) a person spends during exercise, he or she can be at different heart rate *training zones*. Heart rate training zones are expressed in terms of the percentage of the maximum heart rate of a person. The maximum heart rate of a person can either be directly measured or be computed using Miller's formula: $HR_{max} = 217 - age \times 0.85$. Table 3.1 shows 5 heart rate zones, the intensity ranges and their corresponding effects. The target heart rate of a person given the intensity value, I, is calculated by: $HR_{target} = HR_{rest} + (HR_{max} - HR_{rest}) \times I$.

Zone	Effect
Healthy $50\% - 60\%$	Strengthens heart, improves muscle mass. Reduces fat, cholesterol, and blood pressure.
Temperate $60\% - 70\%$	Basic endurance. Fat burning.
Aerobic $70\% - 80\%$	Strengthen cardiovascular system. Step up lung capacity.
Anaerobic $80\% - 90\%$	Getting faster. Getting fitter.
Red Line 90%-100%	Working out here hurts. Increased potential for injury.

Table 3.1: Heart rate training zones.

3.10 Summary

The material in this chapter provides the basic terminologies that are used in the later chapters of this dissertation. We define acoustic events, soundlets and their types, and mobility contexts that we use in Auditeur. We provide the basic background on MFCC acoustic feature and sparse FFT. We also present some basic concepts of heart rates, music related acoustic features, and heart rate training zones which are used in Musical-Heart.

Chapter 4

Mobile-Cloud Platform Overview

This chapter provides an overview of *Auditeur*, which is a general-purpose, energy-efficient, and context-aware acoustic event detection platform for mobile devices. Auditeur enables application developers to have their application register for and get notified on a wide variety of acoustic events. Auditeur is backed by a cloud service to store user contributed sound clips and to generate an energy-efficient and context-aware classification plan for the mobile device. When an acoustic event type has been registered, the mobile device instantiates the necessary acoustic processing modules and wires them together to execute the plan. The mobile device then captures, processes, and classifies acoustic events locally and efficiently. In this chapter, we at first introduce the Auditeur platform and describe the motivation behind the system (Section 4.1). We also compare and contrast various aspects of Auditeur with other related systems in more detail than we have done earlier in Chapter 2. The next section provides an overview of the workflow of Auditeur by describing its three steps along with an example API usage (Section 4.2). The third section provides an overview of the key properties of Auditeur (Section 4.3).

4.1 The Auditeur Platform

Being able to deliver mobile applications quickly to customers is crucial in today's highly competitive application market. Developers depend heavily on APIs provided to them by the mobile OS platform for common tasks, but when it lacks necessary functions for their application, it costs developers weeks of time for implementation. Beyond just time, many developers do not have the necessary technical background to implement these functions correctly or efficiently. Because of this, there is a growing trend where mobile applications are paired with numerous web services by other providers to get access to specialized or computationally demanding tasks. Microsoft's Hawaii [3], for example, supports path prediction, key-value storage, translator, relay, rendezvous, OCR, and speech-to-text services. Google

provides services such as web search, maps, play, YouTube, cloud drive, and Gmail. Other providers give access to data sources such as weather, financial data, airline information, and parcel tracking.

Mobile devices, such as smartphones and tablets, have begun to use their microphones for various acoustic processing tasks, but most of the applications are special purpose acoustic event detectors. Examples of voice and music based systems include: speaker identification [4], speech recognition [5], emotion and stress detection [6, 7], conversation and human behavior inference [8, 9], music recognition [2], music search and discovery [10], and music genre classification. There are other types of applications that fall into the non-voice, non-music category, such as a cough detector [11], a heart beat counter [12], and logical location inference [13, 14]. These examples are limited in number and acoustic sensing is often one of multiple sensing modalities in these works. There are existing works [15, 16] that consider multiple types of acoustic events. SoundSense [15] considers speech, music and ambient sound, and provides a mechanism to label clusters of ambient sounds to extend the set of classes. Jigsaw [16] considers speech and sounds related to activities of daily living. These systems still lack in providing tools and mechanisms to easily extend detection capabilities and effectively support development process. They are limited for general purpose acoustic event detection service for mobile devices and in exploiting the promising opportunities given by already prevailing mobile devices and tablets equipped with microphones.

To address this need, we have built a general-purpose acoustic event detection platform called *Auditeur*. Compared to the previous works in the literature, we position Auditeur as a developer platform rather than just a system detecting a set of sounds. Auditeur provides APIs to developers to enable their application to register for and get notified on a wide variety of acoustic events such as speech, other types of man-made sounds, such as emotional or physiological sounds, music, environmental sounds, sounds observed in households, offices, or public places, sounds of vehicles, tools and so forth. Auditeur achieves this capability of classifying general-purpose sounds by utilizing its *tagged soundlets* concept which are crowd-contributed, short-duration audio clips recorded on a mobile device along with a list of user given tags and automatically generated contexts. The cloud hosts the collection of tagged soundlets and provides a set of services, to upload new soundlets and to obtain a detailed *classification plan* to recognize sounds specified by the list of tags as parameters. Typically a plan contains one or more acoustic processing *pipeline configurations*, corresponding to different energy-requirements and user-contexts, specifying the required acoustic processing units, their parameters, and how they are connected.

Several salient features when taken in combination make Auditeur unique. First, Auditeur provides a simple yet powerful API which novice developers with basic object-oriented programming skills can learn easily. Second, Auditeur supports both personalization and generalization. It is possible to manage personal sounds and classifiers as well as obtain sounds and classifiers that were created by other users of the system. Third, the acoustic processing pipeline inside the mobile device is flexible and extensible. The device contains a wide range of acoustic processing primitives. However, only the ones that are required to recognize the desired sounds (as mentioned in the classification plan from

the cloud) are dynamically instantiated and wired to form the pipeline. Addition of a new primitive is easy since the processing units have a generic structure and are dynamically wired. Fourth, the sound recognition service running on the device is adaptive. Applications are notified on a change of context or at a specific battery level, and a new pipeline configuration is loaded to adapt to the change. Fifth, Auditeur is designed for efficiency in communication, computation, and energy consumption. The mobile device needs to connect to the cloud only once to obtain the classification plan. Hence an uninterrupted Internet connectivity is not a requirement. Acoustic processing in Auditeur is efficient as only the necessary components run within the mobile device. An energy-aware acoustic processing plan to be executed by the mobile device is generated at the cloud to increase the lifetime of the device while maintaining high accuracy.

Partitioning the planing and execution between the cloud and the mobile device let Auditeur have the best of both an on-device [15, 4, 6] sound recognizer and a cloud-assisted [7, 2, 11] one. On-device recognizers perform all of the processing inside the device. Typically, they are highly tuned to the application scenario, and use a fixed set of features and an offline-trained, fixed classifier. Thus, they are rigid and are not usable if either the application or its context changes. Cloud-assisted recognizers, on the other hand, send unprocessed or partially processed data to a server, and rely on web services to perform further processing and classification. This approach is more flexible, but has several limitations such as the requirement for an uninterrupted Internet connectivity, high bandwidth, and the expense of sending a large chunk of data over the cellular network. Our approach for Auditeur takes the best of these two strategies. Auditeur performs the signal processing and feature classification completely inside the mobile device, but uses the cloud to store user contributed sound clips, build new classifiers, and obtain an energy-aware classification plan. Once the mobile device receives the classification plan from the cloud, it dynamically instantiates the components required to execute the plan, and keeps detecting acoustic events efficiently and locally.

Auditeur and its energy and context-aware classification planning architecture are different from existing context monitoring and mobile sensing platforms [85, 86, 26, 23, 8]. Symphony [85] requires developers specify their hand-tuned processing pipelines, whereas the acoustic processing pipeline in Auditeur is generated automatically. SeeMon [86] maintains an essential set of sensors to optimize the sensing and transmission cost associated with the sensors. Orchestrator [26] provides a plan-based execution framework with policy-based system optimization which dynamically adapts to the changing system and application situations. It opportunistically changes its execution plan based on the current status of resources under a specified policy such as minimum accuracy or minimum overall energy cost. Unlike [86, 26], Auditeur maintains an informative set of acoustic features and takes a deterministic approach in maximizing its classification accuracy while satisfying the developer specified minimum lifetime goal. Certain aspects of Kobe [23] such as the API to create a classifier, the search for an optimum configuration, and runtime adaptation are conceptually similar to Auditeur's, however, they are quite different in design and technical details. For example, Kobe requires explicit declaration of the processing pipeline, performs an exhaustive search, and offloads code to server, whereas Auditeur generates pipelines automatically, and does not offload code to satisfy its energy constraint.

DarwinPhones [8] proposes a collaborative approach of exchanging classifiers and classification results among mobile devices. Auditeur's shows it collaborative nature at the data level via its crowd-contributed tagged soundlets.

The contributions of Auditeur are the following:

- A versatile, general-purpose, flexible, extensible, context-aware, and efficient end-to-end acoustic event detection platform for smartphones, backed by the cloud.
- A sound recognition service on the smartphone, capable of dynamically wiring acoustic processing units, and running an app-tailored and context-aware classification plan on the phone, and an API to register and get notified on specific sound events.
- A collection of crowd-contributed, admission controlled, and contextually-tagged short sound clips in the cloud available to the developers, and an API to upload and manage them.
- An acoustic feature selection algorithm that generates energy-aware acoustic event detection plans for smartphones. Our empirical evaluation shows that the algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of its maximum achievable accuracy.
- We implement 7 applications with Auditeur, and deploy them in real-world scenarios to demonstrate that Auditeur is versatile, $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events, compared to state-of-the-art techniques. A user study demonstrates that novice programmers can implement the core logic of interesting apps with Auditeur in less than 30 mins, using only $15 - 20$ lines of Java code.

4.2 Workflow of Auditeur

Auditeur comes with a complete suite of APIs to enable developers create an application with minimal effort. The process of creating an application involves three main steps: tagging and uploading soundlets, obtaining a classification plan, and executing the plan to detect acoustic events. These steps are described as follows.

4.2.1 Tagging and Uploading Soundlets

Auditeur provides an API to record, add tags, and upload soundlets to the cloud. The API provides an interface for the default sound capture device, however the developer may choose a different one (such as a Bluetooth microphone) or an already recorded sound clip. After the sound is captured, the tagging process automatically generates the mobile device contexts, and the user can review and change it prior to uploading. The sound clip is then uploaded to the cloud. At this stage, the upload of the public soundlet could fail if it violates the tag or sanity constraints.

Uploading soundlets to the cloud is energy consuming. However, this is done only once and is used to create classification plans for several applications. An alternative to uploading the sound clips is to extract the features locally and then upload the acoustic features only. But we do not follow this approach to keep our design simple and extensible. By uploading only the features we lose the opportunity to explore the possibility of considering new acoustic features and creating more sophisticated classification plans.

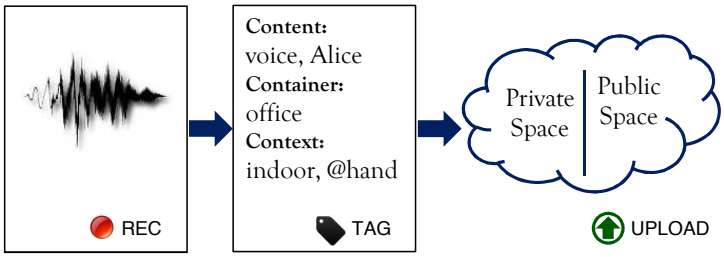


Figure 4.1: Recording, tagging, and uploading soundlets to the cloud.

4.2.2 Obtaining a Classification Plan

An energy-aware acoustic event detection plan is generated in the cloud upon receiving a request from the mobile device. A request contains: sounds the application is looking for, other unwanted sounds that might occur in the environment, contextual information, and energy constraints. The tags and contexts are then used to create a comprehensive training set using a subset of the currently available soundlets in the desired space. To meet the energy constraint, the dimensionality of the training set is reduced by selecting a subset of features, considering the energy consumptions of different acoustic processing units. Finally, a number of classifiers are trained, and the one showing the highest accuracy during cross-validation tests while meeting the energy criteria is chosen. The sound processing pipeline, feature set, the classifiers, and their parameters altogether form the contents of a downloadable configuration file. Since communication to the cloud and training classifiers on-the-fly is costly, it is recommended to obtain classification plans corresponding to different contexts and energy requirements all at once prior to the start of detecting sounds on the device.

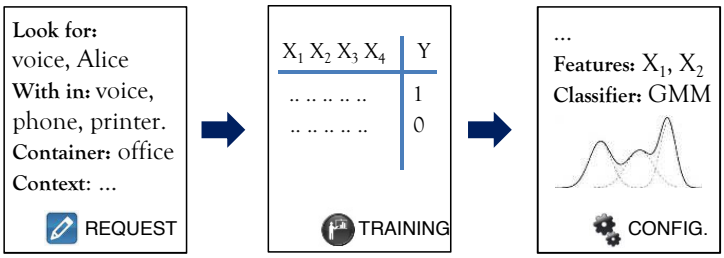


Figure 4.2: Request for a classifier, training the classifier, and generating configuration file for download.

4.2.3 Acoustic Event Detection

The cloud generates an application tailored acoustic event detection plan that directs the mobile device on which components should be instantiated on the mobile device and how they should be wired together to form an *acoustic processing pipeline*. The plan could contain more than one pipeline configuration, corresponding to different contexts or energy-requirements. The mobile device reads the plan from an XML file, instantiates and initializes the processing units with the specified parameters, and wires them together to form a pipeline. Once the pipeline is ready, an API call starts reading audio samples from the micromobile device, and pushes data down the pipeline. The application is issued a callback once the desired sound event is detected. A change in context, such as moving from indoors to outdoors, or putting the mobile device inside the pocket, automatically loads a different pipeline configuration by default, however, the developer can override this behavior.

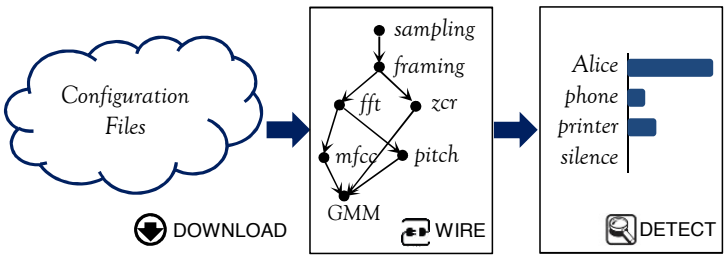


Figure 4.3: Downloading configuration, wiring components, and sound event detection.

4.2.4 API Example

Figure 4.4 shows a code snippet which demonstrates the usage of Auditeur API for recording, tagging, and uploading soundlets to the cloud, and obtaining and using a classification plan from the cloud. Lines $2-3$ create a `SoundletRecorder` to record a 15 seconds audio clip and obtain a `Soundlet` object. Lines $4-5$ create the content and container tags, and lines $6-7$ set the tags and make the soundlet private. Lines $8-10$ grab the `SoundletManager`, and upload the soundlet to the cloud. The second parameter of 3000 in Line 10 is the window size in ms. Lines $12-13$ create the lookfor and within tags. Line 14 creates a `SoundletDetector` and registers the application as a listener. Line 15 obtains a classification plan and initializes the detector. This is done by supplying three sets of tags (i.e. `loookfor`, `within`, and `container`) and an energy constraint indicating that it is a long running (8 hours or more) application. Line 17 starts the continuous sound event detection process with a window size of 3000 ms.

```
1  //Record and upload a soundlet.
2  SoundletRecorder recorder = new SoundletRecorder();
3  Soundlet soundlet = recorder.recordSoundlet(15000);
4  String[] content = {"voice", "female", "Alice"};
5  String[] container = {"office"};
6  soundlet.setTags(content, container);
7  soundlet.setSharing(false);
8  SoundletManager manager = new SoundletManager(
9      getSystemService("SoundletService"));
10 manager.uploadSoundlet(soundlet, 3000);
11 //Get a classifier and register for a match event.
12 String[] lookfor = {"voice", "Alice"};
13 String[] within = {"voice", "printer", "mobile device"};
14 SoundletDetector detector = new SoundletDetector(this);
15 detector.setPipeline(manager.getConfig(
16         lookfor, within, container, SoundletDetector.LONG_RUNNING_APP));
17 detector.start(SoundletDetector.CONTINUOUS, 3000);
```

Figure 4.4: An example code snippet to create, tag, upload, get classification plan, and detect soundlets.

4.3 Key System Features

4.3.1 People in the Loop

The person who contributes a soundlet knows it the best. Auditeur provides an API to let people record their own sounds, and tag them appropriately. An alternate approach, such as automated tagging, might help the user with some suggestions, but is not sufficient since a whole bunch of different sounds is often indistinguishable. For example, aerosol spray, steam, waterfall, and white noise sound almost the same. The tagging process in Auditeur is guided by the concept of the content and container tags, which keeps the background explicit from the sound of interest. The sound recognition process is guided by the *lookfor* and *within* tags, which specify the sound of interest as well as other possible sounds in the environment – the best way to identify these is to engage the end-user.

4.3.2 Personalization as well as Generalization

Auditeur has provision for both personalization and generalization in sound recognition. This is achieved by the two logical spaces: public and private spaces, which are used to store soundlets and create classifiers. The public space contains soundlets that are shared by everyone, and hence this collection is enormous, and the classifiers created using these soundlets have more generalization ability in sound recognition tasks. The public space is for developers who want to build and test an application quickly, and for applications that do not require personalization. The private space, on the other hand, ensures privacy, and the classifiers created using the private soundlets are tailored to meet the end-user's need.

4.3.3 Cloud-Directed On-Device Processing

Sound recognition on the mobile device is not a one-size-fits-all problem. Every problem is unique and involves different sets of tasks along the pipeline. This is why, in Auditeur, we keep provision for dynamically creating a chain of tasks for a specific problem. An algorithm running on the cloud (Chapter 5.2) decides which set of tasks are appropriate for a given problem, and creates a configuration file describing the tasks and parameters required to solve it. The mobile device downloads the configuration file, constructs the chain of tasks, and executes them locally. Communication with the cloud is not continuous, rather the mobile device is required to connect to the cloud only once to get the execution plan.

4.3.4 Designed for Efficiency

Mobile mobile devices are limited by their processing capability and battery life. A novel feature of Auditeur is that, it provides the mobile device with an API to obtain energy-aware classification plans from the cloud. For this to work, we profile the energy consumption of all the tasks that are involved in the sound recognition process on the mobile device. The cloud uses this profile to formulate an optimization problem (Chapter 5.2) that selects the set of acoustic features required to recognize desired soundlets, under a given energy constraint. That means, a mobile device having 30% remaining battery life can use a classifier which is aware of its current condition and is different from the one being used since the battery was full.

4.3.5 Context Awareness

Auditeur considers the contexts in which the sound is recorded and is matched. Keeping contextual information such as the location, body position, and environmental noise level makes the sound matching adaptive to change. This is where we exploit the mobility of the device to make our solution elegant when compared to standard sound matching algorithms. Contexts increase the adaptability of our system in different ways. For example, the physical location of the device greatly eliminates a large number of unlikely sounds from consideration during the sound matching process. Body position of the mobile device is used to normalize the received signal strength, and the environmental noise level guides the noise reduction process. We describe a set of experiments in Chapter 6.4.3 to demonstrate the benefits of using contextual information.

4.4 Summary

This chapter provides a high level overview of the Auditeur platform, deferring the implementation details and its evaluation to the next two chapters. We position Auditeur as a developer platform for creating energy-efficient and

context-aware acoustic event detection mobile applications. Using Auditeur involves three simple steps, which are – tagging and uploading sound clips, obtaining an energy and context aware classification plan, and detecting acoustic events. Auditeur provides APIs for each of these three steps. Auditeur keeps people in the loop by letting them tag, upload, and share sound clips. The system is designed for efficiency in computation and communication.

Chapter 5

Implementation of the Auditeur Platform

This chapter describes the implementation of the Auditeur platform in detail. Tasks involved in recognizing acoustic events inAuditeur are manifold. There are some tasks that are performed before the creation of an application. Examples of such tasks are – collecting and building up a database of tagged sound clips, and creating energy and context aware acoustic event classification plans. Recognizing acoustic events in real-time on a mobile device, on the other hand, involves another set of tasks – capturing audio data from the microphone, preprocessing and feature extraction, and classifying and delivering the events to interested applications. Each application created using the Auditeur platform is unique as acoustic processing units and their parameters are different for different applications. The process of creating a mobile application that recognizes a special type of sound, using a generic mobile-cloud platform requires making an important design decision – how to distribute the acoustic processing tasks between the mobile device and the cloud.

In Auditeur, we decide to distribute the tasks between the mobile device and the cloud in a way that – the mobile device contains the mechanism for sound capture, processing, and recognition, while the cloud provides the logic for building classifiers and tuning the parameters for the sound processing modules on the mobile devices. Auditeur performs the signal processing and feature classification completely inside the mobile device, but uses the cloud to store user contributed sound clips, build new classifiers, and obtain an energy-aware classification plan. Once the mobile device receives the classification plan from the cloud, it dynamically instantiates the components required to execute the plan, and keeps detecting acoustic events efficiently and locally. In this chapter, we at first describe the on-device processing components (Section 5.1), and then the in-cloud processing components (Section 5.2).

5.1 On-Device Processing Components

The implementation of on-device processing is modular and layered as shown in Figure 5.1. At the bottom layer, we have three services that are running in the background – the sound engine service, the context service, and the

communication service. The sound engine service is the one that performs the actual sound recognition task, while the other two manage contexts and communications to the cloud. These services, along with some internal APIs, support the public API layer, which interacts with the applications.

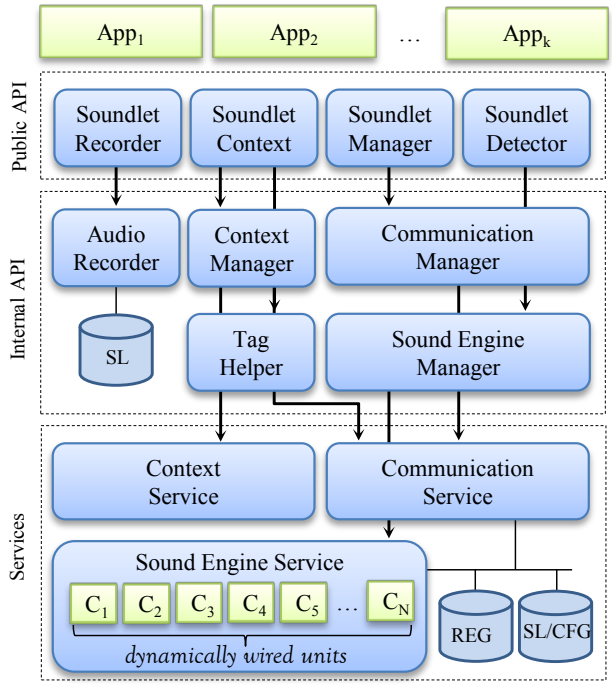


Figure 5.1: On-device processing components.

5.1.1 The Sound Engine

The `SoundEngine` is a singleton, persistent service that is responsible for instantiating and initializing the acoustic processing units, forming a processing pipeline that detects sound events, and providing registration and notification services to the running applications.

Acoustic processing in Auditeur is divided into a five stage pipeline: (1) preprocessing, (2) frame level feature extraction, (3) frame level classification (frame admission), (4) window level feature extraction, and (5) window level classification. Each stage involves multiple tasks, and each task is performed at an *acoustic processing unit* (APU).

Figure 5.2: The acoustic processing pipeline in Auditeur.

Figure 5.2 shows the high level structure of the pipeline. The process starts with a *preprocessing* stage which captures audio from microphone, converts the byte stream into a stream of fixed sized frames, and applies filtering, windowing, and noise compensation as needed by the application. Each frame then passes through a *frame level feature extraction* stage. We have implemented a total of 13 time and frequency domain feature extractor modules, but not all of them might be used in single sound recognition task. These frames are then classified using a *frame level classifier* which acts as an admission controller, and decides whether or not to process a frame any further. If a frame is admitted to the *window level feature extraction* stage, a fixed number of consecutive frames are gathered to form a window, and up to 12 statistics are computed per feature per window, resulting in a maximum of 221-element feature vector. Each window is then classified by a *window level classifier*.

Table 5.1 shows the list of acoustic processing units that are implemented on the mobile device. We have implemented 4 preprocessors, 13 feature extractors, 12 statistical units, and 6 classifiers. We have put more emphasis on the frequency domain features as they are more robust to noise. We have implemented the classifiers that are commonly known to solve sound recognition problems, and have implemented only their classification logic inside the mobile device as their training happens in the cloud.

APU	**List**
Preprocessor	Sampling, Framing, Filters, Windowing.
Features	FFT, ZCR, RMS, 13-MFCCs, Low Energy (Weak) Frame Rate Spectral {Entropy, Energy, Flux, Rolloff, Centroid}, Bandwidth, Phase Deviation, Pitch.
Statistics	Mean, Stddev, Geometric Mean, Harmonic Mean, Range, Moment, Zscore, Skewness, Kurtosis, Median, Mode, Quartile.
Classifiers	Naive Bayes, Decision Tree, GMM, MLP, SVM, kNN.

Table 5.1: Acoustic processing units in Auditeur.

The acoustic processing units (APUs) are the building blocks of the acoustic processing pipeline. APUs are dynamically instantiated and wired at runtime to form the pipeline. Each APU keeps a list of its immediate successor APUs, and implements two methods: `process` and `forward`. The `process` method performs its intended work, and the `forward` method pushes its output to its children. Instantiation of the APUs and the wiring process is guided by an XML-complaint configuration file that is obtained from the cloud. The configuration file is essentially the description of a directed acyclic graph where the nodes are the APUs and the directed edges denote the wiring among them.

Figure 5.4 shows an example of dynamically created pipeline based on the configuration file in Figure 5.3. This is similar to the sound processing as in [15]. In this example, 2 frame level features (RMS and spectral entropy) are used to admit frames using a decision tree classifier, and 2 window level features (spectral flux and zero crossing rate) are used to classify the window using a GMM.

```
<?xml version="1.0" encoding="utf-8"?>
<nodes>
     <node id="1" label="frame" />
     <node id="2" label="rms" />
     <node id="3" label="fft" />
     <node id="4" label="zcr" />
     <node id="5" label="entropy" />
     <node id="6" label="flux" />
     <node id="7" label="DT" parents="2" >
            <params> ... </params> </node>
     <node id="8" label="admitter" window="32" />
     <node id="9" label="stat.mean" window="32" />
     <node id="10" label="stat.mean" window="32" />
     <node id="11" label="aggregator" parents="3" />
     <node id="12" label="GMM" >
            <params> ... </params> </node>
</nodes>
<edges>
     <edge source="1" target="2" />
      ...
</edges>
```

Figure 5.3: Example of an XML file, describing the APUs and wiring to create the processing pipeline.

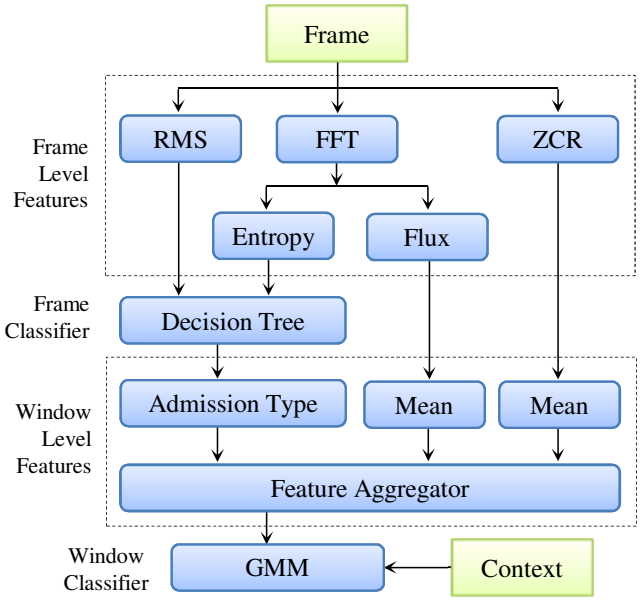


Figure 5.4: An instance of a dynamically wired pipeline corresponding to the XML file in Figure 5.3.

The sound engine internally maintains a register which keeps records of applications, desired sound events, and notification logic. Whenever an event, registered by a running application is detected, it broadcasts a notification according to the notification logic. Currently, we support three types of notification: continuous (always notify), counter-valued (up to a number of events), and timed (up to a time duration after the registration or the first event).

5.1.2 Mobility Context Generation

Auditeur provides three types of contextual information: location, position, and noise level. The location context refers to the GPS coordinates and whether the location is indoors or outdoors. While uploading a soundlet, we store the GPS coordinate if it is available. The availability means – we are getting a location fix and also the application has permission to use GPS. However, the current version of Auditeur only uses indoors or outdoors information. Yet we kept the GPS coordinates to investigate more fine grained location aware sound matching which we leave as future work. We adopt a simplified version of [87] to detect indoors or outdoors using only the accelerometer and cellular signals.

The position context refers to the position of the mobile device with respect to human body which could be either direct (i.e. the sound source is close to the microphone), in pocket, or at a distance. This is similar to the phone contexts used in [15], and is estimated by the same principle. The noise level refers to the environmental noise level in dB scale.

Auditeur provides APIs to specify an action to be taken whenever there is a change in context. For example, when a person comes home from outside, the context engine would detect the change, and take actions that the developer programmed for, which is by default – loading a different pipeline configuration from the classification plan that is appropriate for home environment.

Auditeur provides APIs to enable and disable context checking, and to control the frequency of monitoring the change in each context. As context checking is costly, the developer of the application should choose an appropriate duty cycle to check and balance its benefit over cost.

5.1.3 Communication to the Cloud

Applications communicate to the cloud via the communication service, which submits new soundlets to the cloud through a Web HTTP interface. We implement a basic web application using JAVA and the Spring 3 Framework for this purpose. For hosting, we use the Amazon web services infrastructure by running our application in a Tomcat 7 web container on an Amazon EC2 instance. Soundlets, along with their tags and contexts, are serialized in JSON format and are submitted to the Web. Afterwards, the data is committed to the database as described in Section 5.2.5.

When an application requests a classification plan, the communication service issues a GET request to the web interface on its behalf with search parameters such as the tags, contexts, and energy constraints. In response, the cloud creates the classification plan, serializes it as a JSON object, and sends it to the phone.

5.1.4 Public and Internal API

Four components constitute the public API layer. The `Soundlet Recorder` records audio from the microphone, converts it to a Soundlet, and stores them locally. The `SoundletContext` uses the `ContextManager` to get the current context, and the `TagHelper` to get a list of matching tags from the cloud. The `SoundletManager`, with the help of

`CommunicationManager`, manages the communication between the phone and the cloud to upload soundlets and to download classification plans. The `SoundletDetector` registers the running application as a listener for desired sound events, initializes the `SoundEngine`, and notifies the application at desired sound events. Table 5.2 describes the public classes in Auditeur API and their description in brief.

Class	Description
SoundletRecorder	Records audio from microphone; Converts audio to a soundlet object.
SoundletContext	Generates user contexts; Helps with sound tagging.
SoundletManager	Manages uploads of tagged soundlets; Manages downloaded classification plans.
SoundletDetector	Registers an application as a listener for an event; Initializes the sound engine; Delivers acoustic events to applications.

Table 5.2: Public classes in Auditeur API and their descriptions.

5.2 In-Cloud Processing Components

The primary job of the cloud service is to generate an energy-aware acoustic event detection plan. Given a set of tags, mobile contexts, and an energy bound, the service creates an energy aware acoustic event detection plan. This section describes the creation of an energy-aware acoustic event detection plan on the cloud in detail. Other services such as tag suggestion, storage and caching are described briefly.

5.2.1 Training Set Generation

The cloud creates a training set by choosing a subset of the soundlets in the cloud. First, it selects the soundlets having all of the *container* tags – indicating that the training is environment specific. It then filters out soundlets that do not contain any of the *lookfor* or *within* tags assuming that those are not present in the acoustic environment. Among the remaining soundlets, the ones having all of the *content* tags are marked as positive examples, and the rest are marked as negative examples. Finally, for each soundlet, a 221 element feature vector is computed, and an extended training set is created. The training set has a total of 222 columns (221 elements of the feature vector plus the target class label) and the number of rows equals to the number of soundlets. However, not all of these 221 features are used in classification. The dimension of the feature vector is reduced by a feature selection algorithm which follows next.

5.2.2 Feature Selection

Unlike feature selection [28] in the machine learning literature, where the criterion is to minimize the *number* of features, Auditeur's goal is to minimize the total *energy consumption* of feature extraction on the mobile device, while maximizing their ability to retain enough information for an accurate classification. A brute force approach to solve this problem is to enumerate all possible subsets of the aggregate feature set, and then to pick the subset that results in the highest accuracy when used with a classifier, under a given energy bound. But the size of the maximum feature set in Auditeur being 221, a brute force enumeration of all 2^{221} subsets, each time a classification plan is requested a phone, is not feasible even though the algorithm is running on the cloud. To handle this, we take an iterative approach to this problem. A formal description of the problem and its solution are described next.

Given a training set T of size $n \times m$, where n is the number of samples and m is the dimension of the feature space, the target class labels C of size $n \times 1$, the energy cost of computing each feature on the phone $E = \{e_1, e_2, \ldots e_m\}$, and an energy budget B, the goal is to select a subset of features that uses no more than B units of energy while maximizing their ability to retain enough information for an accurate classification.

Since we are selecting a subset of the features, we need a metric to measure the goodness of a feature. A feature is *relevant* if it is correlated to the target class, and is *non-redundant* if it is not highly correlated to other features. So there are two properties of the best subset:

Redundancy Criterion: The selected features are less correlated among themselves than any other subset.

Relevance Criterion: The selected features are more correlated to the target class than any other subset.

In Auditeur, we take an information theoretic approach in measuring the correlation, which is based on the concept of *entropy*. The entropy of a feature X is defined as

$$H(X) = -\sum_i P(x_i) \log(P(x_i)), \tag{5.1}$$

and the entropy of X, after observing Y is defined as

$$H(X|Y) = -\sum_j P(y_j) \sum_i P(x_i|y_j) \log(P(x_i|y_j)), \tag{5.2}$$

where $P(x_i)$ and $P(x_i|y_j)$ are the prior and posterior probabilities of X, respectively. The difference between Equation (5.1) and (5.2) denotes the *information gain* [88], indicating how much is the added value of taking feature X, given that Y is already selected. In our algorithm, we use the normalized information gain, which is called the *symmetrical uncertainty* [89], to limit its value in the range $[0,\ 1]$.

$$SU(X|Y) = 2\left[\frac{H(X) - H(X|Y)}{H(X) + H(Y)}\right], \tag{5.3}$$

We now formulate a recurrence relation which is solved using an iterative technique. Let $f_j(b)$ be the best subset of features within an energy bound of b, considering the first j features, for $(1 \leq j \leq m)$. Based on the decision on X_j, there can be two cases:

Case 1: X_j is not taken. In this case, $f_{j-1}(b)$ will be a candidate solution for $f_j(b)$ as $f_{j-1}(b)$ is the best solution using the first $(j-1)$ features within the bound b, according the definition of $f(.)$.

Case 2: X_j is taken. In this case, we first reduce the energy bound by the amount e_j, which is the energy cost of the $j-th$ feature. All feasible sets for the new energy bound is $f_{j-1}(b-e_j-t)$ where, $0 \leq t \leq b-e_j$, as these are the sets where we can add X_j without crossing the energy bound b. We then look for the set that has the least correlation (i.e. the least symmetrical uncertainty) with X_j. Let's assume that after this search, we find the least correlated set, $f_{j-1}(b-e_j-t_l)$. So, our candidate solution in this case is $f_{j-1}(b-e_j-t_l) \cup \{X_j\}$.

$$f_j(b) = \begin{cases} f_{j-1}(b) & \text{, if } X_j \text{ is not taken.} \\ f_{j-1}(b-e_j-t_l) \cup \{X_j\} & \text{, if } X_j \text{ is taken.} \end{cases}$$

Finally, between these two candidate solutions, the one that is more correlated (i.e. symmetrical uncertainty is higher) to the target class C is chosen as $f_j(b)$. This is to enforce that we prefer a more relevant set over a less relevant one.

There are two implementation issues that require further explanation:

First, our algorithm assumes that energy is integer valued; The definition of entropy also requires nominal values. To handle this, we quantize both of them to fit into our algorithm.

Second, the energy budget B is for feature extraction only. It is derived by subtracting the energy cost of other APUs and services, e.g., pipeline overhead and context generation service, that run on the phone from the total budget for acoustic processing.

5.2.3 Processing Units and Parameter Selection

The energy consumption and the classification accuracy depend on the choices of parameters for other APUs as well. Table 5.3 describes the choices of parameters considered in Auditeur for the sampling rate, frame size, window size, and number discretization levels for feature quantization. There is a default value for each of the parameters which is used if the developer does not specify any. A negative value for a parameter forces the cloud service to try out all choices for that parameter.

The higher the sampling rate the more energy it consumes, but provides better classification results. Frame size, window size, and discretization levels are problem dependent, and have effect on accuracy. The choice of classifier is also problem dependent, and their parameters are computed by analyzing the training set using standard practices.

Processing Unit	Parameter	Choices	Default
Sampling	rate (KHz)	8, 22.05, 44.1	44.1
Framing	length (ms)	32, 64	32
Window Size	length (sec)	1, 3, 5	1
Feature Extraction	discretization (level)	8, 16	16

Table 5.3: Choices of parameters for APUs.

Considering all 36 combinations of these units and 6 classifiers, the cloud generates a maximum of 216 different execution plans (all within the total energy budget), and selects the one that results in the highest accuracy in 10-fold cross validation test.

5.2.4 Tag Matching Service

The tag matching service is used in two scenarios: (1) admission control of a public soundlet, and (2) finding matching tags for an untagged soundlet. Both of these are done using a distance based outlier detection technique [81], i.e., extracting the feature vector of the soundlet, computing similarity scores of the feature vector against a subset of the existing soundlets, and returning the top k tags in the order of decreasing similarity. For admission control, one or more of the user given tags must be in the list of k tags, and for tag-suggestions, the list of k tags is sent to the phone.

The collection of soundlets against which the similarity is measured is selected using the mobility contexts. For example, location (indoors or outdoors) and position contexts are used to eliminate any unlikely soundlets from considerations. If GPS coordinates are available, soundlets that are recorded within the vicinity of the untagged soundlet are always considered for a similarity match.

5.2.5 Storage and Caching

The cloud stores thousands of tagged soundlets and their acoustic features, therefore fast search and retrieval of items are paramount. We choose to use MongoDB which is a NOSQL database over a relational database for many reasons. First, the number of features for an instance can vary considerably depending on the processing configuration used at the time, therefore, a schema would be too rigid for our purposes. Secondly, NOSQL databases are optimized for write once, read many times situations and tend to scale linearly using replicas in the cloud infrastructure. For efficient queries into the training set, we create an index on the tag and context attribute set.

When a classification plan is requested from a client, the local cache of prepared classifiers is first searched. If a classifier for the particular tag combination and context are not found, a new classifier is generated from the set of matching instances in the training set and committed to the cache. If a new sound instance is added with the same tag combination as existing classifiers, the cache entry is marked dirty allowing an updated classifier to be created. Our

default strategy uses on-demand creation of classifiers since many queries are very specific, however we also allow the cloud to preemptively create new classifiers on a daily basis from popular search queries.

5.3 Summary

This chapter describes the implementation of the Auditeur platform in detail. Acoustic processing tasks in Auditeur are distributed between the mobile device and the cloud. Inside the mobile device, we have implemented a sound engine service, which contains the implementation of 4 preprocessing units, 13 feature extractors, 12 statistical units and 6 classifiers. Inside the cloud, there is a service that computes an energy- and context-aware acoustic event detection plan for a mobile device, given the type of acoustic events that the device wants to recognize. The cloud implements an acoustic feature selection algorithm that selects a subset of features with the properties that – when these features are extracted inside a mobile devices, the total energy consumption will be less than the energy bound, and the selected set of features are not redundant with respect to each other and are relevant to the target class. The plan generated by the cloud contains a list of acoustic processing units, such as preprocessors, feature extractors, and a classifier, their parameters, and wiring information. The mobile device obtains the plan from the cloud, instantiates the units, and executes the plan in order to detect acoustic events locally.

Chapter 6

Evaluation of the Auditeur Platform

This chapter provides an in-depth evaluation of the Auditeur platform. We describe four types of evaluations. First, we measure the CPU utilization and memory footprint of Auditeur, and the energy consumption of different processing units of Auditeur (Section 6.2). Second, we perform an empirical study on different categories of sounds to demonstrate the efficacy of our energy-aware feature selection algorithm (Section 6.3). Third, we implement seven applications, each in three ways: in-phone, in-cloud, and using Auditeur, and compare their energy consumption (Section 6.4.1), accuracy (Section 6.4.2) and context awareness (Section 6.4.3). Fourth, we perform a usability study of Auditeur API (Section 6.5).

6.1 Experimental Setup

We collect data from two sources. The first one is our own collection, obtained from a total of 35 participants from two different regions (Virginia and Beijing), collected over nine months, resulting in a database of about $5,000$ tagged soundlets with contextual information. The group of participants is comprised of undergraduate and graduate students, researchers, professionals, and their family members. Their ages are in the range of $10 - 60$, and they have diversities in speaking-style, life-style and ethnicity. The smartphones we have used during the data collection are Galaxy Nexus phones running Android 4.0 OS, each having a 1.2 GHz TI OMAP4460 dual core processor, 1 GB RAM, 32 GB internal storage, and 28 GB USB storage.

The other source of data is mainly websites including findsounds.com and grsites.com, yielding another 2000 tagged soundlets. The web crawled data is used only in the empirical evaluation, not in our case-studies. We use only those sound clips whose file format, number of channels, bit resolution, and sampling rates are the same as our phone recordings. We do not mix up web crawled data with data from phones in most cases. For example, sound clips that are tagged with 'monkey' are all from the web as we do not have any user who contributed any monkey sounds. For some

	CPU	Memory
Auditeur (silence)	6%	6.5 MB
Auditeur (active)	16% (42%)	11.8 MB (22 MB)

Table 6.1: CPU and memory footprints.

rare sounds, we re-recorded the web-audios with a smartphone. For example, we play the 'siren' sounds loudly on a laptop and then record that on a smartphone.

6.2 System Measurements

6.2.1 CPU and Memory Footprint

We measure CPU utilization and memory footprints of Auditeur when used in an application with a simple GUI. The GUI is used to select different sound recognition tasks that are run during the experiment. We use two utility applications (Norton and OS Monitor) to measure CPU and memory usages. Table 6.1 shows the average (and the maximum in brackets) CPU and memory usages of the applications. The CPU usage is only 6% during silence when the system is duty cycling, 16% on average, and could reach up to 42% in the worst case if all of the processing units are used in an application (which is unlikely). The memory usage is about 6.5 MB during silence, but reaches 11.8 MB for an average application, as heap space is allocated for the dynamically instantiated units. However, this is not so high when compared to services such as Maps (62.7 MB), Music (54.9 MB), and Calendar (18.9 MB) on Android. The total size of the binary is only 596 KB.

6.2.2 Energy Measurements

We measure the energy consumption of each acoustic processing unit on the phone with a high precision power monitor [90]. Prior to the experiment, we kill all background services and running applications, disable wireless connectivity, and set the screen brightness to its minimum. We use two minute long audio files recorded at 44.1 KHz, and measure the total energy consumption.

Figure 6.1 shows the average energy consumption to process a 32 ms frame for each unit. We observe that, the most expensive units are the frequency domain feature extractors. However, all of these components (marked with an asterisk (*)) use the same FFT which is computed only once per frame to save energy. Despite this, the extraction of frame level features, together with the computation of window level statistics, account for 98.48% of the total energy consumption of the processing pipeline. This signifies why Auditeur emphasizes carefully choosing the features to increase the lifetime of the device. The sum of individual energy consumptions however is not an accurate estimate of

the actual energy consumption; but from our experience with Auditeur we have seen that, this is a conservative estimate, and the phone lasts longer than the estimated lifetime in practice.

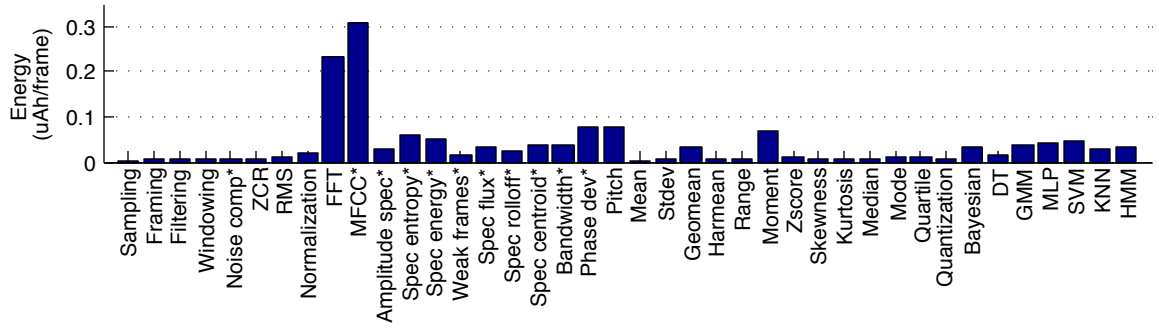


Figure 6.1: Feature extraction accounts for 98.48% of the total energy consumption of the processing pipeline.

6.3 Empirical Evaluation

We analyze the trade-off between accuracy and energy savings, compare the feature selection algorithm with a greedy heuristic, illustrate the energy efficiency, and measure the processing delay.

We use our empirical dataset in these experiments, which is segmented into 10 categories of sounds as described in Table 6.2. In this study, we are showing intra-category sound recognition accuracy within each dataset. Note that, these are very demanding datasets with many similar sounds in different categories within the same dataset to stress the accuracy metric, and even for humans these are hard to distinguish. However, later in our case studies (Section 6.4), we consider real-world scenarios where we have sounds from different classes.

6.3.1 Lifetime and Accuracy Trade-off

Auditeur's feature selection algorithm trades off less informative features for a longer lifetime of the device. The goal of this experiment is to quantify how much accuracy Auditeur compromises to achieve this.

Figure 6.2 shows the classification accuracy of Auditeur for different datasets. For each dataset, the first bar corresponds to the highest achievable accuracy considering no energy bound, and the other three bars correspond to minimum lifetime requirements of 300, 400, and 500 minutes, respectively. The number on top of each bar denotes the dimension of the feature vector. We see that, for an unbounded energy, Auditeur would use all acoustic features and achieve the highest average 10-fold cross validation accuracy of $65.4\% - 97.2\%$. The accuracy is below 70% for some datasets, such as D7, D3, and D9, which contain varieties of similar sounds. However, in an actual application, not all of these might be used or they might be representing the same class; hence the accuracy will be higher in practice as evident from Section 6.4.2. The tighter the energy bound becomes, i.e. the larger the minimum lifetime requirement is,

ID	Dataset	Count	Subcategories
D1	Alert	334	alarm, bell, horn, siren, whistle.
D2	Animal	145	cat, cow, dog, horse, monkey, tiger.
D3	Household	453	air conditioner, doorbell, doorslam, fan, spray, vacuum, water tap, and more.
D4	Instruments	309	drum, guitar, piano, violin, flute.
D5	Music	1253	country, folk, indian, jazz, rap, rhymes, rock, spiritual, and more.
D6	Non-speech	783	asthma whizz, chew, clap, cough, cry, foot-steps, laughter, whistle, yell.
D7	Office	653	deskbell, keyboard typing, mouse clicks, phone, printer, sharpener, stapler.
D8	Speech	1725	female, male, child, speaker ID.
D9	Tools	439	broom, chain saw, drill, grinder, hammer, lawnmower.
D10	Vehicles	815	airplane, ambulance, bus, car, subway.

Table 6.2: Description of the empirical dataset.

Auditeur selects less number of features to maintain the lifetime goal. For a 500 min (8.3 hours) lifetime, Auditeur chooses only 18 features on average to keep the system running, sacrificing about 8.17% accuracy. However, for a moderate lifetime of 400 mins (6.67 hours), the difference in accuracy is $< 2\%$.

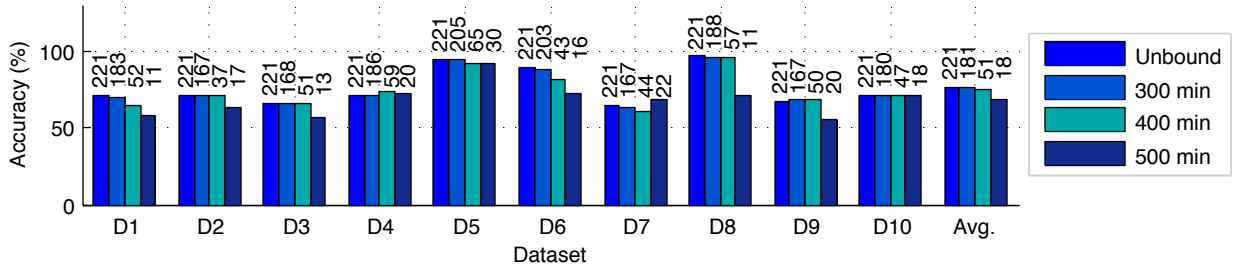


Figure 6.2: Auditeur increases the lifetime by $33.4\% - 66.7\%$, sacrificing $2\% - 8.17\%$ accuracy.

6.3.2 Evaluating Feature Selection Algorithm

We compare Auditeur's energy-aware feature selection algorithm with Weka's [29] symmetrical uncertainly attribute evaluator. Since Weka does not consider energy, we choose the first few highest ranked attributes as long as the sum of their energy costs remains within the bound. Figure 6.3 compares the average accuracy of a classifier for different minimum lifetime bounds, when the classifier uses these two feature selection methods. The error bars shown on the plot represent the stanard deviation. When the energy bound is lower, i.e. the minimum lifetime is as high as $500 - 600$ minutes, less number of features are chosen by both of the algorithms. But Auditeur selects the subset that is optimal within the bound, whereas the greedy algorithm performs poorly. The gap between these two, however, gets closer as

the bound becomes loose since both algorithms then select enough number of features to classify the sounds properly. Therefore, applications that require long term ($8 - 10$ hours) continuous sound recognition, Auditeur would provide $8\% - 14.9\%$ higher accuracy than the greedy algorithm.

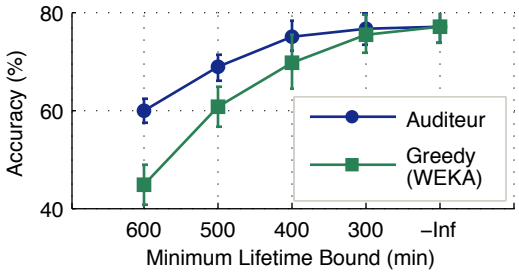


Figure 6.3: Auditeur achieves $8\% - 14.9\%$ higher accuracy than WEKA for long running applications.

6.3.3 Illustration of Energy Efficiency

We illustrate the energy efficiency of Auditeur with a simple example scenario. In this scenario, we run the same workload on two identical Android phones for about 9 hours. Both of the phones run Auditeur, but one assumes an infinite energy bound, and the other one changes its classification plan after two hours to last longer.

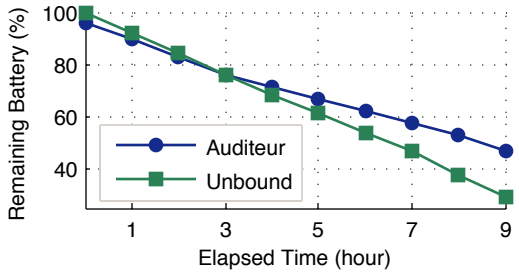


Figure 6.4: Auditeur's remaining battery is 18.12% higher than the unbounded one after 9 hours.

Figure 6.4 plots their remaining battery life at each hour mark. We see that, the phone with unbounded energy drains battery at the rate of 7.89% per hour, and after 9 hours, its remaining charge is only 29%. The other one (denoted by Auditeur) initially drains battery at the rate of 6.67% per hour, but this rate is reduced to 4.81% after reducing the energy bound by 15% at hour 2. After 9 hours, Auditeur's remaining battery is 47.12%, which is 18.12% higher than the unbounded one. That means, Auditeur would last for 31 hours, whereas the unbounded one will die after 19 hours.

6.3.4 Processing Delay

We measure the processing delay, i.e., the average duration of a $1s$ long frame inside the pipeline. To obtain this, we capture $60s$ audio, and process it with 300 different pipeline configurations, each having a different minimum lifetime

bound. The total time to process the frames is normalized to compute the processing delay of a frame.

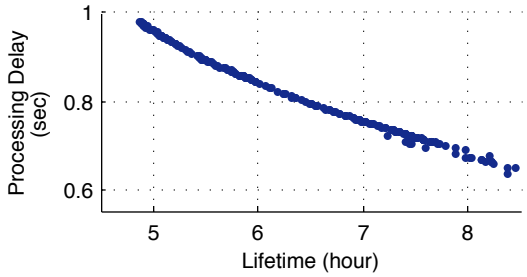


Figure 6.5: Processing delay for $1s$ frames is always $< 1s$.

Figure 6.5 shows processing delays of $1s$ long frames for different lifetime bounds, ranging approximately $5 - 8$ hours. We observe that, the delay is higher ($0.98s$) at the beginning, and drops below $0.7s$ as the bound crosses the 8-hour mark. This is because, at tighter bounds, Auditeur extracts a small number of features to meet its lifetime goal, which results in shorter processing delays. However, overall the delay is always $< 1s$ for a $1s$ frame.

6.4 Case Studies

We implement and evaluate seven applications with Auditeur and compare their energy efficiency, accuracy, and context awareness against two baselines: the in-phone and the in-cloud implementations of the applications. The features and parameters for the baselines are taken from corresponding existing works. Both baselines extract features on the phone, but the in-cloud version sends them to the cloud in real-time to get the classification results, while the in-phone version does everything on the device. The definitions of in-phone and in-cloud implementations thus consider any hybrid approach (such as – partitioning or offloading computation to the cloud at runtime [23]) as an in-cloud implementation. For communication, WiFi is used indoors and 3G outdoors. Auditeur however downloads all of its classifier configurations, corresponding to different location and position contexts, at the beginning and does not use wireless connectivity afterwards.

Five Android phones and a tablet are used at the same time during these experiments. Two phones run the baselines, one runs Auditeur, one runs Auditeur with only location context enabled, and one runs Auditeur with only position context enabled. The tablet is used for recording the ground truth and bookkeeping purposes. Each phone stores the timestamp, detected events, and the battery level.

Table 6.3 shows the list of applications that we study. We replicate the first three applications from existing literature [15, 4, 12], and add four more to demonstrate Auditeur's performance in different real-world scenarios. Ground truths for Speaker Sense and Sound Sense are obtained from six volunteers, by logging who is talking and what music is played with a tablet. These experiments are done in ten $15 - 120$ minutes long sessions at indoors and outdoors.

App (Short Name)	Detected Events
Sound Sense (Sound)	male, female, music.
Speaker Sense (Speaker)	person identification.
Musical Heart (Heart)	heart beats.
Music Match (Music)	music genre recognition.
Vehicle Sense (Vehicle)	car, bus, subway, trolley.
Kitchen Sense (Kitchen)	door, blender, pots, stove, microwave, tap.
Sleep Monitor (Sleep)	talk, cough, steps, bathroom door, fan.

Table 6.3: Applications in case studies.

Musical Heart uses a subset of the dataset of [12] (L1 activity level) which is read from files. Music Match is trained on 200 English songs of different genre, and then the tablet randomly chooses and plays them at regular intervals, while the phones listen and classify them. Vehicle Sense and Kitchen Sense are trained on samples collected from members of two two-person families, and are tested on them separately in $3-5$ hours long experiments. Two volunteers participate in the Sleep Monitor experiment separately for two consecutive days where each session lasts for about $6-8$ hours. We record the entire sleep duration with a tablet, and perform a post-facto analysis, with visualization and manual classification, to identify the events.

6.4.1 Power Consumption

We compare the power consumptions of Auditeur with in-phone and in-cloud implementations. The power consumption is obtained from the total energy consumption, which we calculate using the values that are logged periodically into the phone, i.e. the remaining battery life, voltage, running time, and the battery capacity.

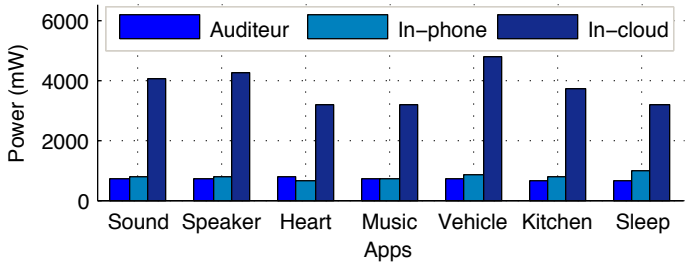


Figure 6.6: On average, Auditeur is 11.04% and 441.42% less power hungry than in-phone and in-cloud versions.

Figure 6.6 shows that, power consumptions of in-cloud implementations are $3.3-6.8$ times higher than the other two, as they continuously send and receive data over the Internet. Especially, for outdoor experiments where 3G is used (Vehicle, Sound, and Speaker), the mean is 4093 mW. Power consumption of in-phone version is comparatively closer to Auditeur. For long running experiments (Sound, Speaker, Vehicle, Kitchen, and Sleep), in-phone ones consume 19.56% more power than Auditeur, as unlike Auditeur, they do not have energy bounds. For short duration experiments

(Heart and Music), in-phone versions are slightly more energy efficient than Auditeur. However, this is not a problem for Auditeur because of two reasons: (1) when sufficient energy is available, Auditeur uses slightly more power to achieve a higher detection accuracy, and (2) even in such cases, developers can specify a tighter energy bound to achieve a lower power consumption. Overall, the power consumption of Auditeur is on average 11.04% less than the in-phone version.

An Auditeur-powered application's lifetime can be further extended by duty cycling. However, this has to be done by the application developers by explicitly specifying the cycling interval for their application using the API (see line 17 of the code snippet in Figure 4.4). Since none of the baseline applications in this experiment implement duty cycling, for a fair comparison of energy consumption, we do not perform duty cycling in Auditeur as well.

6.4.2 Detection Accuracy

The accuracy of event detection is the percentage of events that are detected and classified correctly. All three implementations use the same silence vs. non-silence detection method whose accuracy is almost perfect (98.72%). Hence, we consider the classification of non-silent, fixed length time windows as the overall accuracy.

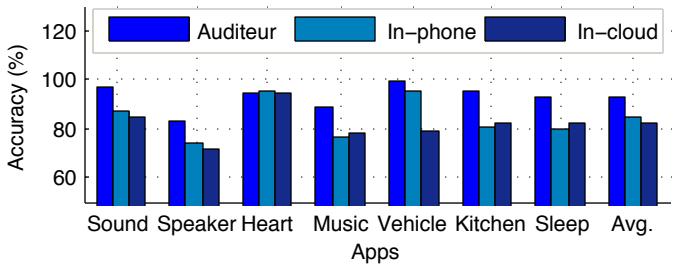


Figure 6.7: On average, Auditeur is 10.71% and 13.86% more accurate than in-phone and in-cloud versions.

Figure 6.7 shows the accuracy of event detection for three different implementations of the seven applications. Auditeur shows higher accuracies than in-phone and in-cloud versions in 6 out of seven applications. This is because of Auditeur's adaptiveness to changing contexts, and the limitations of the other two. Both in-phone and in-cloud versions use offline-trained classifiers, which are trained on all soundlets disregarding their contexts. For example, the accuracy of voice related applications, e.g., the Sound and Speaker, depends on the location (indoors vs. outdoors) due to the presence of reverberation. Auditeur being aware of such contextual information are capable of handling them separately, which the other two cannot. Furthermore, the in-cloud versions lose $14.3\% - 26\%$ accuracy in outdoor scenarios (Vehicle, Sound and Speaker), due to long communication delays in a 3G network. The accuracy of Auditeur in case of Heart application however is similar to the other two versions. This is because, the Heart application, which is one of our previous works [12], uses a highly sophisticated algorithm for heart rate detection and has a very high

accuracy. It is hard to beat such an algorithm using Auditeur especially when the accuracy is close to 100%. Overall, Auditeur shows $10.71\% - 13.86\%$ higher accuracy than the other two versions.

6.4.3 Context Awareness

Auditeur uses different pipeline configurations for different location (indoors vs outdoors) and body position (direct, pocket, or distant) contexts to boost its accuracy. In this experiment, we measure their individual effects on accuracy.

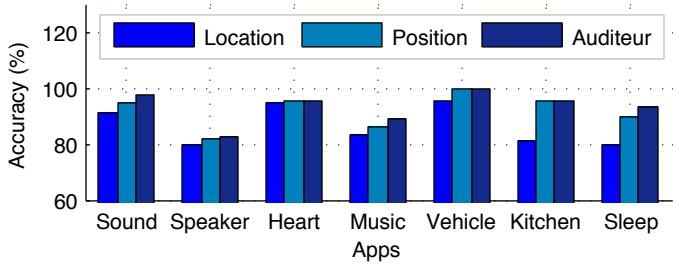


Figure 6.8: Position context is in general more effective than location context.

Figure 6.8 compares Auditeur with two other versions of it: (1) only location, or (2) only position context enabled. A combination of the two always results in a higher accuracy, but their individual contribution varies with applications. Location context is ineffective in Heart, Vehicle, Kitchen, and Sleep, since these are performed entirely indoors or outdoors. For other applications, location alone improves accuracy over baselines (of Figure 6.7) by $4.08\% - 7.93\%$. Position contexts are more effective than location contexts in environments where the phone may be in any of the three position contexts, such as in Sound, Speaker, Music, Kitchen (pocket and table), and Sleep (table and under pillow), contributing 6.45% more accuracy over the location only version.

6.5 User Study

We perform a user study to evaluate the usability of Auditeur API, i.e. how easy or hard it is to learn and use the API. Another motivation is to observe what application ideas burgeon when such an API is made available to the public.

A total of 15 undergraduate students from 3 universities participated in this study. We recruited them by posting an online advertisement and their participation was voluntary. Their experience-levels in Java and Android programming are $0 - 5$ years (avg. 16.6 months), and $0 - 12$ months (avg. 4.73 months), respectively. We provided them with a documentation explaining the API and an example code snippet. Each of the participants was asked to read the documentation to learn the API, think of an application scenario, code it, and then comment on their overall experience.

Since developing a fully functioning application involves other non-sound recognition tasks, such as, programming the GUI and handling events, which are not part of our study, we ask the participants to code only the portion of the

application that is Auditeur-specific, while the rest of the application has already been coded for them. This is done to remove any biases that originate from sources unrelated to learning or coding with the Auditeur API. Figure 6.9 shows the learning time and coding time as they have reported in their answers. About 75% of the participants take less than 15 mins to learn the API, and about 60% of them program the core logic in just 10 mins using only $15 - 20$ lines of Java code. The worst case learning time is as high as 30 mins, while the highest of the coding times is about an hour. This indicates that, Auditeur is very easy to learn and code.

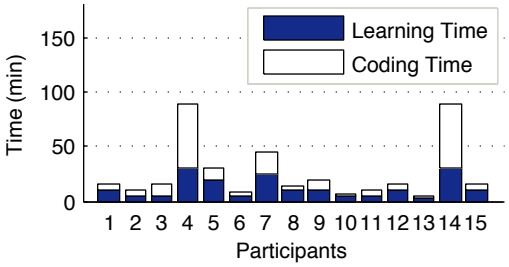


Figure 6.9: Learning and coding time.

It is interesting to observe how our participants have come up with a number of fascinating application ideas and have been able to realize them with Auditeur. Table 6.4 describes some of the ideas that we find more interesting than others.

App	Short Description
Snore	Detect snoring and vibrate the phone until he stops.
Honk	Detect car honks on road and alert inattentive user.
Asthma	Detect asthmatic wheezing sound and alert the caregiver.
Subway	Detect the arrival of a train in subway.
Dog	Detect pet dog barking while the user is away from home.

Table 6.4: Interesting application ideas found in our study.

We also ask our participants to comment on the Auditeur API. Some of those comments turn out to be helpful and are actually incorporated in the system. For example, the participant who implemented the Dog application, asked if we could provide an API to get the last few seconds of audio, so that she knew what caused her dog to bark (e.g. an intruder came). Another participant asked if we could provide a confidence of a match instead of just the detected event. These comments have led us to expose some internal API to obtain last few minutes of audio and the probability distribution of the detected events from the application layer.

6.6 Discussion

Auditeur is capable of detecting sound events that are recognizable from short-term time and frequency domain features. Recognizing sounds that depend heavily on temporal variations, such as recognizing long spoken sentences, are not suitable for it. Especially, speech recognition is a complex, well-studied problem having several well-known services, and hence we leave it out of the scope of Auditeur.

To make sure that people do not upload soundlets with wrong labels, Auditeur performs sanity checks and discards the sound if there is a lack of similarity. This was not an issue in our experiments since the data were uploaded by our trusted users and we only had to deal with unintentional human errors. However, this could be a big issue when the system is open to the public. A more sophisticated sanity checking mechanism will be required at that moment. This is a limitation of our current implementation and we leave it as future work.

Lack of sufficient training examples, specially for private spaces, is another issue. To cope with this, we suggest developers to utilize sounds from public spaces as long as the private space is small, while keep adding new soundlets to the private space. Auditeur provides an API to retrieve last few minutes of audio; developers should use this facility and upload new soundlets via user-feedback.

The features and classifiers used in Auditeur are shown to recognize varieties of sounds, but yet no such list is ever exhaustive. However, Auditeur is an extensible system where addition of a new feature extractor or a new classifier unit is easy since APUs have a generic structure and are dynamically wired. Hence, such extensions do not require any change to the framework.

The energy consumption and lifetime depend on the phone model and battery capacity. The relative energy costs of the processing units are, however, implementation dependent, and remain the same. This is, therefore, a scaling problem, and the lifetime bound for such cases should be considered as power levels instead of an absolute value. We leave it as a future work to remotely profile energy consumption of the device to provide device tailored lifetime values.

Scalability of the services running on the cloud is not addressed in this chapter which is by itself a well-formed problem. Our current implementation is a proof of concept which runs on an Amazon EC2 instance. However in future we plan to move our server to Google App Engine which provides automatic scaling of applications without requiring us to manage the machines by ourselves.

6.7 Summary

This chapter describes a series of experiments to evaluate various aspects of the Auditeur platform, such as system overhead, energy consumptions and usability of the API, as well as energy-efficiency, context-awareness and accuracy of the applications that are created using the platform. Our analysis on user-contributed empirical data shows that

Auditeur's energy-aware acoustic feature selection algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of the maximum achievable accuracy. We implement seven applications with Auditeur, and deploy them in real-world scenarios to demonstrate that Auditeur is versatile, $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events, compared to state-of-the-art techniques. We perform a user study to demonstrate that novice programmers can implement the core logic of interesting applications with Auditeur in less than 30 minutes, using only $15 - 20$ lines of Java code.

Chapter 7

Exploiting Sparseness in Speech Signals

Auditeur is a general-purpose acoustic event detection platform supporting the creation of mobile applications that recognize a wide variety of acoustic events. However, some acoustic signals have special properties which can be exploited to make acoustic event detection even more efficient. Speech is such a signal which is highly sparse in the frequency domain. This chapter is dedicated to speech signals where we show that by exploiting the sparseness in speech, we are able to compute a frequency domain acoustic feature up to ≈ 6 times faster. The speedup however comes at a price of up to 4% reduction in accuracy. This is why we do not make this feature a default in Auditeur. Instead, we keep it as an option for applications that must process speech events at such a faster rate.

Due to limited processing capability, contemporary mobile devices cannot extract frequency domain acoustic features in real-time on the device when the sampling rate is high. We propose a solution to this problem which exploits the sparseness in speech to extract frequency domain acoustic features inside a mobile device in real-time, without requiring any support from a remote server even when the sampling rate is as high as 44.1 KHz. We perform an empirical study to quantify the sparseness in speech recorded on a smartphone and use it to obtain a highly accurate and sparse approximation of a widely used feature of speech called the Mel-Frequency Cepstral Coefficients (MFCC) efficiently. We name the new feature the sparse MFCC or sMFCC, in short. We experimentally determine the trade-offs between the approximation error and the expected speedup of sMFCC. We implement a simple spoken word recognition application using both MFCC and sMFCC features, show that sMFCC is expected to be up to 5.84 times faster and its accuracy is within $1.1\% - 3.9\%$ of that of MFCC, and determine the conditions under which sMFCC runs in real-time.

7.1 The Sparse MFCC Feature

All major mobile platforms these days support numerous voice driven applications such as – voice commands (e.g. to launch an application or call some contact), voice-enabled search (e.g. Google's voice search), voice recognizing

personal assistant (e.g. iPhone's SiRi), and voice-based biometrics. There are also non-voice sound driven applications, such as the music matching service from Shazam [2]. All of these applications require fast acoustic feature extraction both in time-domain and frequency-domain in order to offer fast, real-time services. While using only the time-domain acoustic features is sufficient in a limited number of applications, the frequency-domain features are a must for a robust and accurate encoding of acoustic signals.

State-of-the-art mobile applications and platforms that extract acoustic features are primarily of two kinds. The first kind records the audio and sends it to a remote server over the Internet for further processing. This method has several limitations such as the requirement for an uninterrupted Internet connectivity and high bandwidth, and the associated expense of sending a large chunk of audio data over the cellular network. The second kind, on the other hand, performs the entire signal processing task inside the phone. But the limitation of this approach is that in order for fast and real-time feature extractions, they must limit the sampling rate to the minimum. For example – SpeakerSense [4] and SoundSense [15] limit their maximum sampling rate to 8 KHz. Hence, the quality of sampled speech suffers from the aliasing problem and the extracted features are often of low quality [91]. Although a sampling rate of 8 KHz satisfies the Nyquist criteria for human speech ($300 - 3300$ Hz), practically the higher the sampling rate the better it is in producing high quality samples. Furthermore, for non-speech acoustic analysis, a sampling rate of 44.1 KHz is required to capture the range of frequencies in human hearing ($20 - 20000$ Hz). But the problem is – there is no efficient algorithm that extracts frequency domain acoustic features inside the phone in real-time at such high sampling rates.

In this chapter, we describe a novel solution to this problem which enables the extraction of frequency domain acoustic features inside a mobile device in real-time, without requiring any support from a remote server even when the sampling rate is as high as 44.1 KHz. We are inspired by a recent work [83, 84] coined sparse Fast Fourier Transform (sFFT) – which is a probabilistic algorithm for obtaining the Fourier transformation of time-domain signals that are sparse in the frequency domain. The algorithm is faster than the fastest Fourier transformation algorithm under certain conditions. Our goal is to analyze the feasibility of applying the sFFT to extract a highly accurate and sparse approximation of a widely used feature for speech, called the Mel-Frequency Cepstral Coefficients (MFCC) on a mobile device. Besides speech recognition, MFCC features are widely used in many other problems such as speaker identification [49], audio similarity measure [51], music information retrieval [50], and music genre classification. However, we limit our scope to speech data only.

We perform an empirical study involving 10 participants (5 male and 5 female participants) where we collect more than 350 utterances of speech per person, recorded at different sampling rates. In our study, we quantify the sparseness of speech and show that human voice is suitable for applying sFFT to compute frequency domain acoustic features efficiently. We analyze the sensitivity of sFFT in approximating MFCC, and based on this, we design an algorithm to efficiently extract MFCC features using the sFFT instead of the traditional FFT. We name this new feature the sparse

MFCC or sMFCC, in short. We experimentally determine the trade-offs between the approximation error and the expected speedup of sMFCC. As a proof of concept, we implement a simple spoken word recognition application using both MFCC and sMFCC features and compare their accuracy and expected running time on our collected data.

The main contributions of this chapter are:

- A study on 10 smartphone users to quantify the sparseness of speech data recorded on a smartphone and to analyze the feasibility of using sFFT for frequency domain feature extraction.
- We describe an efficient algorithm for computing a highly accurate and sparse approximation of MFCC features which exploits the sparseness in speech.
- We implement a simple spoken word recognition application using both MFCC and sMFCC features, show that sMFCC is expected to be upto 5.84 times faster and its accuracy is within $1.1\% - 3.9\%$ of that of MFCC, and determine the conditions under which sMFCC runs in real-time.

7.2 Motivation

Mobile devices allow a fixed number of sampling rates to capture raw audio signals from the microphone. Ideally the choice of an appropriate sampling rate should be driven by the application's QoS requirements. But often the developers are forced to choose a lower sampling rate than the desired one due to the limited processing power of the device. For example, in general-purpose acoustic processing, a 44.1 KHz Nyquist sampling rate is required to capture the range of frequencies in human hearing ($20 - 20000$ Hz). Even in speech processing problems, oversampling at 16 KHz helps avoid aliasing, improves resolution and reduces noise [92]. But at higher sampling rates, the real-time performance of the smartphone gets worse and the developers are forced to select the minimum rate compromising the quality of sampled speech.

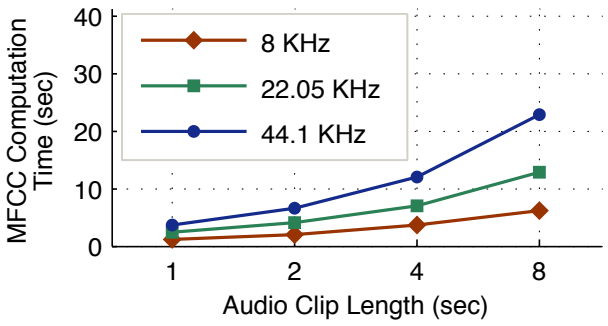


Figure 7.1: The MFCC computation time is $2 - 4$ times longer than the audio clip length when the sampling rate is high.

Figure 7.1 compares the time to compute MFCC feature vectors from audio records of different durations at 3 different sampling rates. The experiment is done on a Nexus S smartphone running Android 2.3 that supports 8 KHz, 22.05 KHz, and 44.1 KHz sampling rates. We see that, the time to compute MFCC features is always longer than the duration of the recorded audio when the sampling rate is higher than 8 KHz. For example, the computation of MFCC vectors of a 4-second recording takes on average 7.02 s at 22.05 KHz, and 11.85 s at 44.1 KHz. Therefore, at these higher rates, the application is not capable of real-time performance.

Our goal is therefore to investigate the problem: whether or not it is possible to compute MFCC feature vectors in real-time on a smartphone when the data rate is high? In an attempt to answer this question, we study the nature of human speech recorded on a smartphone. We hypothesize that the sparseness in speech can be exploited to compute a close approximation of MFCC feature vectors on a smartphone in real-time. While the focus of this work is on speech, an analysis of the general-purpose acoustic signals based on similar principles is under investigation and we leave it as our future work.

7.3 The Sparse MFCC Algorithm

The idea of sparse MFCC algorithm is to compute a sparse approximation of MFCC features from a given frame of discrete time-domain signals x_n of length n. The algorithm uses a modified version of sFFT as a subroutine. We denote the new feature by sMFCC to signify its relation to sFFT. Like the sFFT to FFT, sMFCC is an approximation to MFCC, where the approximation error is defined by,

$$error(k) = 1 - \mathbf{M\hat{F}CC} \cdot \mathbf{sM\hat{F}\hat{C}C(k)} \tag{7.1}$$

where, $\mathbf{M\hat{F}CC}$ and $\mathbf{sM\hat{F}\hat{C}C(k)}$ are unit vectors, and their scalar product is subtracted from unity to obtain the approximation error. sMFCC is expressed as a function of the sparseness parameter k, which is one of the key parameters to the sFFT algorithm. The following two sections describe the sMFCC extraction algorithm in detail.

7.3.1 Estimation of Sparseness

Since the value of k is a key input to the sFFT algorithm and sFFT is used in our computation, the first step of sMFCC algorithm is to find the optimum value of k, denoted by k^*, for which the MFCC approximation error is within a small, non-negative threshold δ, i.e.,

$$k^* = \min_{\text{error(k)} < \ \delta} k \tag{7.2}$$

In order to obtain k^*, we first compute the MFCC using the standard FFT algorithm which runs in $O(n \log n)$. We then perform an iterative $O(n)$ search for $k \in [1, n]$ until we find the optimum k^*. The computation of sMFCC(k), for $k \in [1,\ n]$, is optimized by precomputation. We precompute the FFT, keep the FFT coefficients sorted in non-increasing order, and take only the largest k coefficients while making other coefficients zero – while computing sMFCC(k). Note that, this step of our algorithm does not use sFFT and runs in $O(n \log n)$. The shape of the function $error(k)$ (Figure 7.4 in Section 7.5.2) however suggests that, instead of a linear search over all values of k, we could expedite the process with a binary search. However, k is estimated once per utterance, i.e. using the first $5 - 10$ frames once voice activity is detected, and hence the amortized cost of this step is not significant.

7.3.2 Computing sMFCC

Once we obtain the sparsity parameter k, we compute the sMFCC for each frame in 3 steps which we describe next.

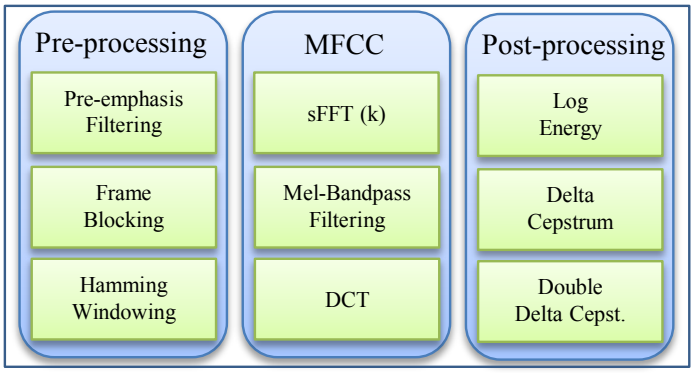


Figure 7.2: The computational tasks involved in the sMFCC feature extraction process is shown.

Pre-processing: The time-domain signals are first passed through a high-pass pre-emphasis filter (Eq. 3) to amplify the high-frequency formants that are suppressed in speech. We then segment the signals into frames of ≈ 64 ms with an overlap of 1/3 of the frame size. A hamming window (Eq. 4) is applied to each frame to ensure the continuity between the first and last points which is required for FFT.

$$x[i] = x[i] - 0.95 \times x[i-1] \tag{7.3}$$

$$hamm(i) = 0.54 - 0.46 \cos(\frac{2\pi i}{n-1}) \tag{7.4}$$

MFCC: We modify the sFFT algorithm to fit into our algorithm. The sFFT algorithm tries to extract k Fourier coefficients in $\log k$ iterations to guarantee the retrieval of all k coefficients. However, in our experience, sFFT returns most (at least 75%) of the k coefficients in a single iteration. We, therefore, modify the sFFT by running it for only a single iteration with a slightly larger sparseness parameter of $k = \min(n, \lceil 4k^*/3 \rceil)$ to speed up the process. Once the

Fourier coefficients are obtained, we follow the standard procedure of MFCC [47]. We apply 20 triangular band-pass filters (called Mel-banking) to obtain 20 $\log$ energy terms, perform a DCT to compress them, and take the first 13 coefficients to constitute a 13-element sMFCC vector M.

Post-processing: The 13-element sMFCC vector is augmented to include the delta and double delta cepstrums to add dynamic information into the feature vector, and thus we obtain a 39-element feature vector.The deltas Δ and double deltas Δ^2 are computed using the following two equations,

$$\Delta_i = M_{i+2} - M_{i-2} \tag{7.5}$$

$$\Delta_i^2 = M_{i+3} - M_{i-1} - M_{i+1} + M_{i-3} \tag{7.6}$$

7.4 Experimental Setup

We perform an empirical study involving 10 volunteers, in which, we record their speech using a smartphone in home environments. Each participant was given a list of 86 English words and a paragraph from a book. The wordlist includes 10 digits, 26 characters of the English alphabet, 25 mono-syllable and 25 poly-syllable words. Participants were asked to utter each word 4-times – clearly and at a regular pace. There were about a 2-second gap between two spoken words so that we could extract and model each word separately. The group of participants is comprised of undergraduate and graduate students, researchers, professionals, and their family members. Their ages are in the range of $20 - 60$, and they have diversities in speaking style and ethnicity. The smartphone we used during the data collection is a Nexus S phone running Android 2.3.6 OS. It has a 1 GHz Cortex A8 processor, 512 MB RAM, 1 GB internal storage, and 13.31 GB USB storage. The execution time of MFCC is measured on the smartphone using Android's API, and speedup of sMFCC is the ratio of running times of MFCC and sMFCC.

7.5 Experimental Results

We conduct four sets of experiments. First, we quantify the sparseness of speech in our empirical dataset. Second, we show the approximation error in sMFCC. Third, we establish the condition for speedup in sMFCC. Finally, we describe a simple spoken word recognizer to quantify the cost and benefits of sMFCC over MFCC.

7.5.1 Sparseness in Speech

The sparseness of signal is defined by the number of negligible Fourier coefficients in its spectrum. A coefficient is considered negligible if it contains a very small amount of signal power. Sparseness in audio signals depends on the audio type. In this chapter, we study clean speech signals only.

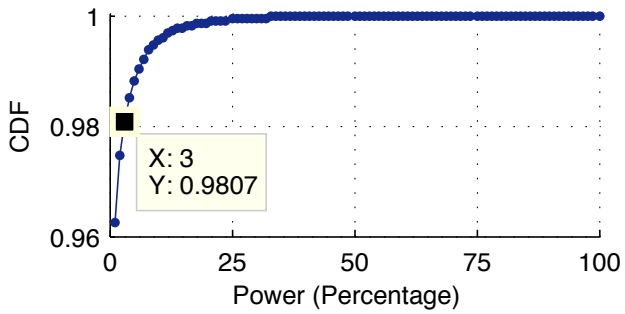


Figure 7.3: 98.07% of the Fourier coefficients in our dataset contains only 3% or less power.

Figure 7.3 shows the cumulative distribution function (CDF) of power in the Fourier spectrum of the utterances in our dataset. To obtain this plot, we compute the FFT of all the utterances in our dataset, take the squared magnitude of FFT to obtain the signal power, construct a 100-bin histogram where each bin corresponds to a range of powers, compute the fraction of Fourier coefficients that are in each bin, and compute the CDF. Each point on the plot tells us, what fraction of the signals has power less than or equal to the range corresponding to the X-coordinate. For example, the marked point on the plot denotes that 98.07% of the Fourier coefficients in each utterance of our dataset contains only 3% or less power. The rest 1.93% coefficients that are significant are permuted (see [83, 84] for the details) in the frequency domain so that the spectrum becomes extremely sparse and ideal for the application of sFFT.

7.5.2 Sparse Approximation Error in sMFCC

The quality of sMFCC features depends on the choice of an appropriate k. The larger the value of k, the closer it is in approximating MFCC. In this experiment, we analyze the sensitivity of k to the MFCC approximation error.

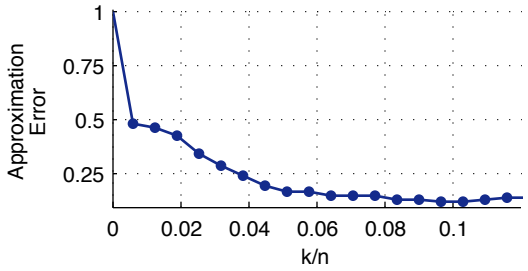


Figure 7.4: The higher the value of k, the better approximation of MFCC we get.

Figure 7.4 shows the approximation error for the range of sparseness $k/n \in [0.00625, 0.125]$. We consider this range since it contains most of the significant Fourier coefficients and is also important for the discussion of speedup in Section 7.5.3. Each point on the plot corresponds to the mean approximation error of sMFCC for a given k/n, where the mean is taken over all 64 ms frames in all the utterances in our dataset. The frame size is $n = 4096$ samples, which is the next power of 2 that holds a 64 ms frame at 44.1 KHz. This figure guides us in choosing the parameter k in our

sMFCC computation algorithm if we want to keep the MFCC approximation error below a desired threshold. A very close approximation ($< 1\%$ error) is possible by choosing $k/n = 0.2$ or higher. However, such close approximation may not be required in an actual application which we will see in Section 7.5.4. The reason is that human speech being sparse, even at a smaller k/n ratio, the absolute value of approximation error is not high.

7.5.3 Speedup in sMFCC

Sparseness in speech is the source of expected speedup in sFFT and hence in sMFCC as well. Figure 7.5 shows the speedup in sMFCC for the range of sparseness $k/n \in [0.00625, 0.125]$. We observe the maximum speedup of 5.84 when the sparsity parameter is at its minimum. As we consider more and more FFT coefficients while computing sMFCC, the speedup decreases and after $k/n = 6.769\%$ the regular MFCC becomes faster than its sparse counterpart. This limitation comes from the fundamental bound of sFFT, which says, sFFT is faster than FFT when $k/n < 3\%$ [84]. However, our modified version of sFFT is faster for the reason we discussed earlier in Section 7.3.2, and hence we have a larger bound of 6.796%.

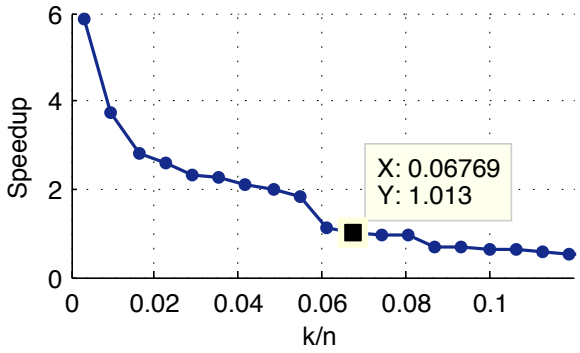


Figure 7.5: sMFCC has a better running time than MFCC as long as the sparseness $k/n < 6.769\%$.

7.5.4 A Simple Spoken Word Recognizer

The goal of this experiment is to analyze the tradeoff between the accuracy and expected speedup of sMFCC features in an application scenario. To do so, we implement a simple spoken word recognizer that is essentially a speech-to-text program for a single word from a fixed vocabulary. We empirically determine the trade off between accuracy and speedup for different values of k/n. The developer of the application should decide what k/n value to pick for his application. For different applications, the most suitable value of k/n will be different, and needs to be chosen from a similar trade off curve. The recognizer consists of two parts: a smartphone application and a word recognizer running on a PC. The word recognizer running on a PC is for proof of concept, our envisioned use case is however to run it on the phone.

At first, the user turns on the application on the phone and presses the 'speak now' button. The smartphone then starts sampling the microphone at 44.1 KHz and keeps producing speech frames until the user presses the 'stop' button. Each frame goes through the MFCC feature extraction process which happens in real-time. Each spoken word produces a number of frames, and a 39-element MFCC feature vector is obtained for each frame. We take the mean and the standard deviation of each of the 39 MFCC coefficients over all frames to obtain a single 78-element feature vector which is used in the classification step. The feature vectors are then sent to a PC for classification and further analysis. The ground truth is obtained by taking notes manually. We train a Support Vector Machine (SVM) classifier in order to recognize the words. A 3-fold cross validation is used to determine the accuracy of the classifier where 75% of each user's data is taken for training and the rest is used for validation.

Accuracy

We compare the accuracy of the sMFCC-based SVM classifier with that of the MFCC-based one. The baseline MFCC-based classifier is essentially a special case of sMFCC-based one with a sparseness $k/n = 1$, and has a recognition accuracy of 85.85%. Figure 7.6 compares the recognition accuracy of sMFCC-based classifier to the baseline for the same range of k/n we have been using throughout the chapter. We observe that, the accuracy of sMFCC-based classifier is initially 3.9% lower than the baseline, and the two accuracies becomes practically identical once k/n reaches 0.12. However, from the discussion in Section 7.5.3 we know that, sMFCC runs faster than MFCC only when the k/n ratio is within the 6.679% bound. For this boundary case, sMFCC shows an accuracy of 84.75%, which is only 1.1% lower than the baseline. In summary, with sMFCC-based classifier for our simple word recognition problem, we can achieve a faster running time than the baseline with a very small (1.1%-3.9%) sacrifice in recognition accuracy.

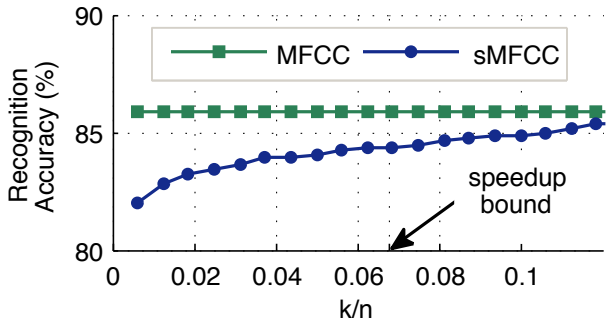


Figure 7.6: The recognition accuracy of sMFCC-based classifier is within $1.1\% - 3.9\%$ of the MFCC-based one inside the speedup zone.

Computation Time

The MFCC feature extraction process runs once per spoken word. Hence, the execution time depends on the duration of the spoken word which varies from person-to-person and from word-to-word. In our dataset, the duration of speech ranges from the minimum of 400 ms to the maximum of 2.88 s. We, therefore, compute the expected feature extraction time $E[\phi_{feat}(d_i)]$ using the following equation,

$$E[\phi_{feat}(d_i)] = \sum f_i \times \phi_{feat}(d_i) \tag{7.7}$$

where, d_i is the duration of speech, f_i is the frequency of utterances with duration d_i, and $\phi_{feat}(d_i)$ is the feature extraction time (either MFCC or sMFCC).

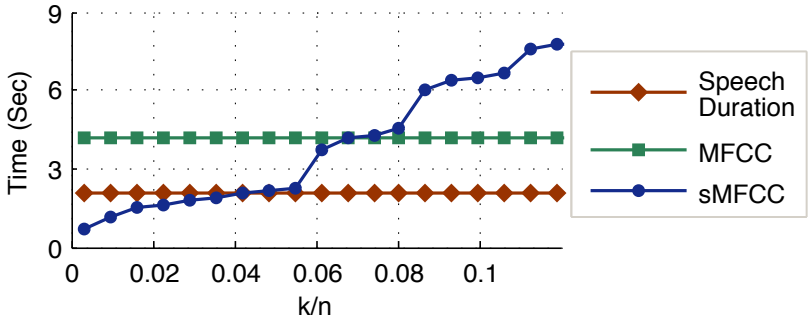


Figure 7.7: The sMFCC feature extraction algorithm runs in real-time as long as $k/n < 4.835\%$.

Figure 7.7 shows the mean speech duration and the expected computation times of MFCC and sMFCC feature extraction process. We see that, the expected MFCC computation time is 4.21 s, which is about 2 times higher than the duration of speech (2.11 s). On the other hand, the expected computation time for sMFCC varies with k/n: it increases as k/n increases, is lower than the duration of speech as long as $k/n < 4.835\%$, and crosses the MFCC computation time when k/n reaches the speedup limit of 6.679%. Hence, applications that require fast and real-time feature extraction should set the k parameter such that k/n is below the real-time limit of 4.835%. At this limit, the accuracy of the sMFCC-based word recognizer is 83.97%, which is only 1.88% lower than the accuracy of the baseline MFCC-based classifier.

Discussion

7.6 Summary

This chapter describes a novel technique which exploits the sparseness in speech signals to speedup the computation of MFCC features on a mobile device. This technique is beneficial when an application must sample speech signal at a very high rate (44.1 KHz or more), and requires a real-time performance. We name this new feature sparse MFCC or sMFCC, in short. Computation of sMFCC is up to ≈ 6 times faster than MFCC features, but this speedup comes with a cost of $1.1\% - 3.9\%$ reduction in accuracy. In Auditeur, we include sMFCC as an optional feature, which a developer may choose to use only if the application must sample the microphone at 44.1 KHz or more and the speedup is more concerning than the accuracy of event detection.

Chapter 8

Mobile-Cloud Communication Efficiency

Auditeur performs on-device acoustic event detection in real-time as opposed to sending audio data to the cloud over the Internet. This makes acoustic event detection up to 441% more energy efficient and 13.86% more accurate when compared to in-cloud classification, as shown in Chapter 6. However, acoustic event detection may not be the only purpose of a mobile application. An application may be using Auditeur to detect acoustic events locally (on device), but at the same time it may be constantly using the Internet to communicate with several web services for other tasks. For example, [93] is a mobile application that characterizes the ambiance of a local business (e.g. noise, chatter, music level of a restaurant) while a user checks-in to a place on his social networking application. This application performs audio classification on-device but communicates to the application server of the social networking application over the Internet. Musical-Heart [12] is another application that detects heartbeats from acoustic signals on-device, but communicates to a remote server to get recommendations on next songs to play. With Auditeur, we envision many such applications where audio sensing and classification may be one of many modalities of sensing and context inference. For these types of applications, *MultiNets* provides a way to achieve mobile-cloud communication efficiency, which is the topic of this chapter.

In this chapter we present MultiNets, a service which is capable of dynamically switching wireless network interfaces on mobile devices. MultiNets is motivated by the need of mobile platforms to save energy, offload data traffic, and achieve higher throughput. We describe the architecture of MultiNets and demonstrate the methodology to perform wireless network interface switching in Linux based mobile OSes such as Android. Our analysis on mobile data traces collected from real users shows that with dynamic switching we can save 27.4% of the energy, offload 79.82% of the data traffic, and achieve 7 times more throughput on average. We deploy MultiNets in a real world scenario and our experimental results show that depending on the user requirements, it outperforms the state-of-the-art Android system either by saving up to 33.75% energy, or achieving near-optimal offloading, or achieving near-optimal throughput while

substantially reducing TCP interruptions due to switching.

8.1 The MultiNets Engine

Cellular networks today provide nationwide coverage in several countries all over the globe. The proliferation in mobile applications like mobile TV, video on demand, video conferencing, tele-medicine, and numerous location based services is attracting an increasing number of consumers. However, the challenges to effective use of mobile networks remain manifold. Firstly, the mobile data traffic is soaring at a high rate. A recent study forecasts that global mobile data traffic will increase by 39 times with a compound annual growth rate of 108% over the next five years [94]. Secondly, the usable battery lifetime of mobile devices has become alarmingly low with feature packed mechanisms like touch screens and accurate positioning system. Thirdly, modern mobile applications typically require high throughput and/or fast response time which are difficult to deliver with scarce cellular bandwidth and expensive spectrum.

Contemporary mobile devices come equipped with wireless network interfaces such as WiFi. This offers an attractive proposition to alleviate the staggering increase in data traffic over cellular networks owing to high bandwidth and low cost (or, often no cost) offered by WiFi networks. Although, the spatial coverage of WiFi is not comparable to the cellular networks due to short coverage range of the WLAN technology, the availability of WiFi is becoming more pervasive in houses, offices, campuses, stores, coffee shops, and even many public transport systems – locations where most mobile users tend to spend most of their time.

The state-of-the-art devices tend to leave the choice of selecting the network to the end-user, which we argue is not only inefficient, but also undesirable in terms of usability. A user should instead be able to decide the high level goal and the device should switch to suitable interface to achieve that intent. By *real-time switching*, we mean activating a new network interface and deactivating the current one – dynamically and without interrupting existing connections.[1] Multifold benefits can be realized by switching in real-time. For example, battery life is prolonged if the device stays over the cellular network during its idle time and switches to WiFi during its data activity. An end-user who is concerned about the battery life can set the device to energy-saving mode, and the device can monitor the user's activity and perform the switching when appropriate. Similarly, by switching to the network that has the highest bandwidth, the mobile device can provide a better user experience with faster data rates. Switching is also a solution to the skyrocketing data problem faced by the mobile operators who want to offload data traffic to WiFi network to conserve cellular bandwidth.

Switching from one network interface to the other is challenging due to the connection-oriented nature of the ongoing data sessions. Unless properly dealt with, switching between interfaces results in interruptions, loss of data, and undesirable user experience. Existing works that attempt to solve this problem either require additional

[1] *Real-time*, in this context, stands for switching interfaces in real-time as opposed to meeting any deadline.

infrastructure supports such as gateways [95, 36, 96, 97, 37, 98] and masters [39], or require changes in the network protocols [31, 32, 33, 34, 35], and thus are not practical since modification to infrastructure tends to be extremely expensive and modification to a standard network protocols is not a compatible pathway with regard to existing and deployed systems and applications.

In this chapter we present MultiNets, a pragmatic client-based solution, which does not require any changes to the network protocols, and enables existing applications to run transparently without any modification. MultiNets is able to switch between cellular and WiFi interfaces in real-time and makes switching decisions based on one of its three interface selection policies: *energy saving*, *data offloading*, and *performance*. These policies address three crucial needs of a mobile device in being able to conserve battery power, offload data to WiFi, and increase throughput, respectively. A user of mobile device equipped with MultiNets can select one of these high-level policies and MultiNets performs switching accordingly. We note that, the authors in [18] characterize TCP flows on iPhones to analyze the feasibility of flow migration between interfaces. In contrast to our work, they do not consider issues such as the policies determining when to switch, or rigorously quantify different benefits that are achieved by switching.

We have implemented MultiNets on Android-based mobile devices. However, the design and principles of MultiNets are general enough to be adopted in any other mobile OSes. Like all state-of-the-art mobile OSes, Android does not perform dynamic switching. Access to network interfaces in Android is exclusive, i.e., either the cellular or the WiFi is active at a time. The cellular network is the default network and is assumed to always be present. On the other hand, WiFi has to be manually turned on, and typically the user is prompted to select WiFi when it is available. A limitation of Android is that switching is not seamless i.e., all the TCP connections are bound to be interrupted. Furthermore, when the device is connected to WiFi, switching back to the cellular network can only be done by manually turning off the WiFi connectivity. In MultiNets, we obviate this exclusive network access and make it possible to keep both the interfaces on simultaneously for seamless and non-disruptive switching. In addition, this feature can be used to simultaneously access multiple network interfaces by the applications that are developed on top of MultiNets.

We perform extensive experiments to evaluate various aspects of MultiNets. First, we measure the system overhead and switching time between cellular and WiFi. Second, we analyze our collected data traces from real mobile phone users and quantify the benefits of each of the three policies separately. Finally, we demonstrate the performance of our system in a real-world scenario.

The rest of this chapter is structured as follows. Section 8.2 presents an empirical study and describes the switching algorithm. Section 8.4 presents in detail how MultiNets is implemented. Section 8.5 shows performance results.

This chapter makes the following contributions:

- We conduct a three months long empirical study and summarize the TCP characteristics in Android smartphones, complementing a similar study with iPhone users in [17, 18]. We devise a switching technique which is

client-based, transparent to applications, and does not require any protocol changes.

- We design and implement three switching policies. Our analysis on usage data collected from real mobile device users shows that with switching, we can save 27.4% of the energy, offload 79.82% data traffic, or achieve 7 times more throughput on average.
- We present MultiNets, which is to the best of our knowledge, the first complete system of this kind and demonstrate its performance in a real-world scenario. MultiNets outperforms the state-of-the-art Android system either by saving up to 33.75% energy, or achieving near-optimal offloading, or achieving near-optimal throughput while substantially reducing TCP interruptions due to switching.

8.2 Switching Network Interfaces

In this section, we describe the problem of seamlessly switching interfaces, and provide a solution to this problem based on our study on the characteristics of data flows in Android devices.

8.2.1 The Network Interface Switching Problem

Switching from one wireless interface to another is not as trivial as it may appear to be at first. Simply turning on one interface and turning off the other does not work as it results in interruptions, partially loaded web pages, loss of data, annoying error messages and user dissatisfaction in general. It is rather a challenging problem to *transfer connection oriented data traffic from one interface to another under the constraints of no user interventions, no interruptions, no changes of network protocols and requiring no extra support from the existing network infrastructure*.

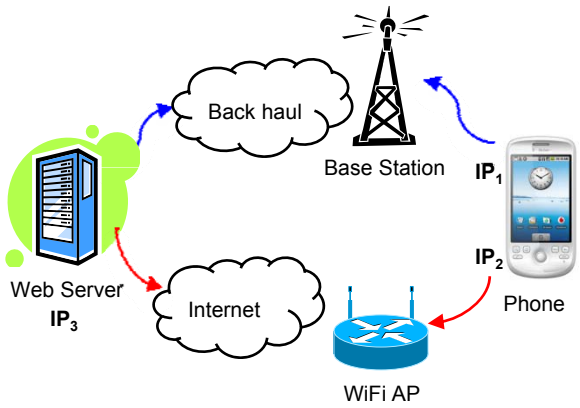


Figure 8.1: A phone is trying to switch TCP sessions from 3G(IP_1) to WiFi(IP_2).

Figure 8.1 illustrates this problem briefly. An end host, having two interfaces (IP_1 and IP_2) creates a TCP connection at it's port A with the remote server's (IP_3) port B. The connection is uniquely identified by the pair

$(IP_1/A, IP_3/B)$. We now analyze what happens if the host decides to turn off its interface, IP_1 and wants to continue the communication over IP_2. By changing the routes of all outgoing packets, the host may be able to send the next data packets using IP_2, but these packets will not be recognized as belonging to the same session at the server, as to the server, IP_1 and IP_2 appear as two different hosts. If we change the packet headers at the host to carry IP_1 as their sources even if they are sent using IP_2, the packets will be either dropped at IP_2's network or even if they get to the server, the ACKs will not reach the host as the server will send ACK to IP_1 which is closed.

8.2.2 Potential for Switching Interfaces

A client based solution that deals with this problem has to wait for all ongoing sessions over the first interface to finish, before it turns off the interface and activates the other one in order to prevent any interruptions. This waiting time can be theoretically infinite, but in practice, it depends on the usage of the mobile device and the characteristics of the applications that are running. To understand the type of data flows in mobile devices, we conduct a 3-months long experiment to collect the usage data from 13 Android device users. These users are of different ages, demography, sex, and used a variety of applications. The results are therefore not homogeneous, rather highly diverse as evident later in Figures 8.15, 8.16, and 8.17. The results are also consistent with a 3-months long study involving 27 iPhone users in [17, 18].

In our study, the users used a total of 221 applications, and 35 of these applications require Internet connectivity. We analyze the collected data traces of these 35 applications and study the characteristics of the TCP sessions. We study TCP since our earlier work shows that almost all (99.7%) mobile traffic is TCP [17]. From this log, we try to answer three questions: *(1) how many concurrent TCP sessions are there at any instant of time within a mobile device? (2) what are the durations of these sessions? and (3) how much activities are there over these sessions?* Answers to these questions are crucial since if we see that there are a large number of TCP sessions having long durations and high data activities, it is not worthy to wait for them and not wise to close them.

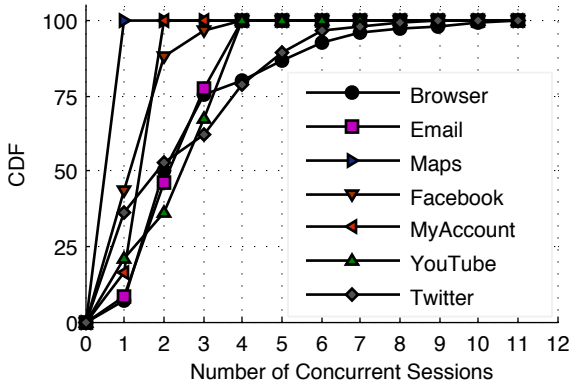


Figure 8.2: A steep rise in CDF in between $1-3$ indicates that the mean concurrency lies in that interval.

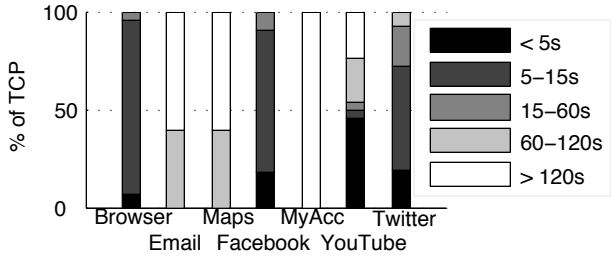


Figure 8.3: Applications with high concurrency tend to have most of their sessions with a lifetime of < 15 sec.

Figure 8.2 shows the cumulative distribution functions (CDF) of concurrent TCP sessions of the most popular 7 applications in the order of their usages. This is averaged over 10-minutes time windows of all users. This figure has to be studied in conjunction with Figure 8.3 which classifies these sessions into five classes based on their durations. In Figure 8.2, we observe that the concurrency of the TCP sessions has a steep rise in between $1 - 3$. This means that the average value (~ 2) lies in this region. Therefore, turning off the interface at any time may interrupt about 2 sessions on average, assuming that only one application is active at a time. In case of multi-tasking mobile devices, the number of interruptions is not much higher. Applications like Browser and Twitter seem to show high concurrency (of maximum 10-11), but Figure 8.3 shows that about 80% of their sessions have lifetime < 15 seconds. For these applications, we may on average have to wait for 15 seconds before switching to the other interface. Applications like Email, Maps, and T-Mobile's MyAccount have very high percentage of long lasting sessions (> 120 seconds) which may seem a barrier to waiting for them to finish. However, the number of such long TCP sessions are very small (about 1), and with these TCP sessions, the applications keep themselves connected to a specific server (e.g., in case of Google Maps, it is `74.125.45.100`) for the entire lifetime, and that explains why they are so long.

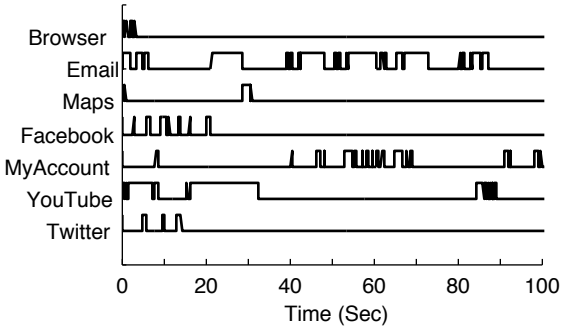


Figure 8.4: Data activity in long sessions are not continuous, rather they have an average burst length of ~ 3 sec.

We conduct further investigation to examine the data activities in these long sessions. Figure 8.4 shows the presence of data activity of the longest lasting TCP sessions of each of these applications for a randomly selected user for the first 100 seconds as an example. This shows that, the data activities over these longest TCP sessions are not continuous,

but rather sporadic. Averaged over all usages we see an average of 3 seconds data activity between any two gaps over these sessions. This indicates that we may have to wait on average about 3 seconds for these applications before switching to a new interface to prevent any data loss. Although this technique will cause that TCP session to terminate, luckily, mobile applications are written keeping in mind of the sudden loss of network connectivity. Therefore, in such cases, when we switch to a new interface, the application considers it as a loss of connectivity and re-establishes the connection with the server. We empirically observed this in all 35 applications.

8.2.3 Summary of Findings

The characteristics of TCP sessions in mobile devices are summarized as follows:

- Average lifetime of TCP sessions is ~ 2 seconds.
- Average concurrency of these sessions is < 2.
- TCP activities are in bursts of average ~ 3 seconds.
- There exists some sessions that are alive during the entire lifetime of the application, keeping the application connected to its server. Disconnections of such sessions are automatically reestablished by the application.

8.3 Switching in MultiNets

MultiNets handles connectionless and connection-oriented sessions separately during the switching. UDP and TCP are the dominant transport protocols that we have observed in Android, and therefore we use them in this section for illustration.

8.3.1 Connectionless Sessions

Connectionless sessions (e.g., UDP) are rare: less than 0.3% traffic amount. They are easier to switch. UDP applications communicate using `DatagramSocket` and each connection is bound to a port and assigned an IP address of an available interface by the OS. To switch network interface, MultiNets first turns on the new interface and removes the default route over the old interface. We have found that doing so does not affect the functionality of `DatagramSocket`: the out-bound traffic is sent with the IP of the new interface, while the in-bound traffic is received at the old interface. MultiNets then turns off the old interface, which initially incurs some packet loss of the in-bound traffic, but we have observed that, in most cases, this is handled by the application layer.

8.3.2 Connection-Oriented Sessions

Connection-oriented sessions are mostly TCP, comprising of about 99.7%. These are trickier to switch as explained earlier. MultiNets performs the following steps for switching these sessions:

Step 1: MultiNets counts the number of ongoing TCP connections on the *old* interface. We should not harm these connections. We exclude the sessions that have gone past the ESTABLISHED state during the counting.

Step 2: If the count is non zero, MultiNets adds new routing table entries for all these connections explicitly specifying the destination address, gateway and mask fields for the *old* interface. This is to ensure that the ongoing TCP sessions still remain in the *old* interface.

Step 3: MultiNets now brings up the *new* interface and adds routing table entries for it including the default route and removes the default route of the *old* interface from the routing table. Any new connections start using the *new* interface from now on.

Step 4: MultiNets waits for the ongoing TCP sessions over the *old* interface to finish or until a timeout (determined experimentally) – whichever happens first. Finally, it tears down the *old* interface completely and the system moves on to the new interface.

The users of MultiNets have to configure the WiFi network by providing the authentication information only once. After that, MultiNets switches the interfaces dynamically without requiring any manual intervention. The proposed switching solution in MultiNets is fully client based – it does not require additional support from the access points or gateways. Furthermore, MultiNets does not require any modification to the network protocols. It only reads the transport information and adds or removes routing table entries to perform a switch. This is why, existing applications run transparently on MultiNets without any change. There is a possibility that during the Step 4 of the switching, a very long TCP session may get interrupted due to timeout. We empirically derive that, a timeout value of 30 seconds or more makes this interruption a rare phenomena. We notice that the proposed switching technique may also be applicable to other applications, e.g., Alperovich and Noble [99] propose a similar technique to switch among homogeneous WiFi access points, which is quite different from MultiNets.

8.3.3 Switching API

MultiNets uses the API shown in Figure 8.5 to switch to a new interface. The method `switchInterface()` takes the name of the interface as an argument and returns either success or failure. Upon failure, it throws an exception explaining the reason of failure.

```
1 SwitchingManager mgr = new SwitchingManager(getSystemService("SwitchingService"));
2 try {
3     if (mgr.switchInterface(mgr.MOBILE) == true) {
4         // Success. New sessions start over 3G now
5     }
6  } catch(SwitchingException ex) { ex.printStackTrace(); }
```

Figure 8.5: Using switchInterface() method.

8.4 Design and Implementation

The design of MultiNets is modular, consisting of three principal components – *Switching Engine*, *Monitoring Engine*, and *Selection Policy* as shown in Figure 8.6. These components isolate the mechanism, policy and monitoring tasks of the system, and allow extending their capabilities without requiring any changes to the architecture.

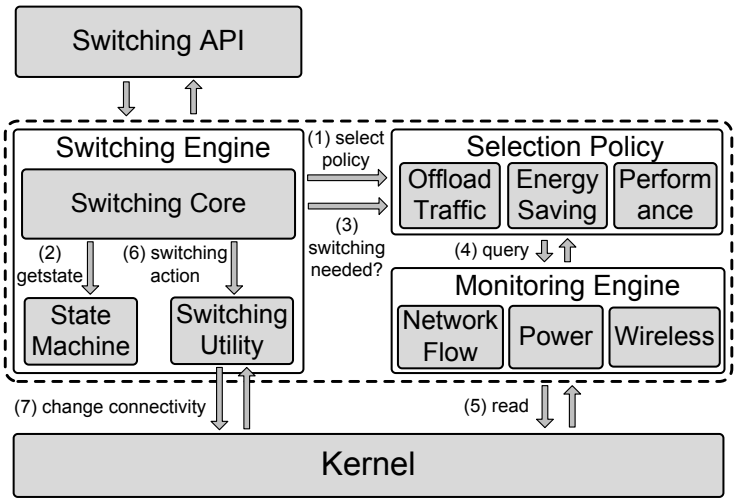


Figure 8.6: MultiNets Architecture.

8.4.1 The Switching Engine

The Switching Engine performs the switching between cellular and WiFi. It maintains an internal state machine to keep track of the connectivity status. It also has a Switching Utility module that performs some low level tasks related to switching. The Switching Core coordinates these two.

The State Machine

Figure 8.7 shows the state diagram together with all the states and transitions. The system remains at *NoConnectivity* state (S_0) when neither cellular nor WiFi is available, and keeps seeking for a network to connect to. The states *ConnectedToCellular* (S_1) and *ConnectedToWiFi* (S_3) are similar. At these states, the device uses only one wireless interface and periodically checks with the Selection Policy (see Section 8.4.3) to see if a switch is needed. The states *SwitchingToWiFi* (S_2) and *SwitchingToCellular* (S_4) are the transition states. Both of the interfaces are active during these states, but only the new connections start over the new interface while existing sessions still remain in the previous interface. The engine stays at these states as long as the old interface has active TCP sessions or until a timeout. Under normal circumstances, the system moves around within the states $\{S_1, S_2, S_3, S_4\}$ circularly. To cope with any loss of connectivity, the system makes some transitions shown in dotted arrows. A loss of WiFi connectivity at S_2 takes the system to S_3, but it immediately starts switching back to cellular upon detecting such a disconnection.

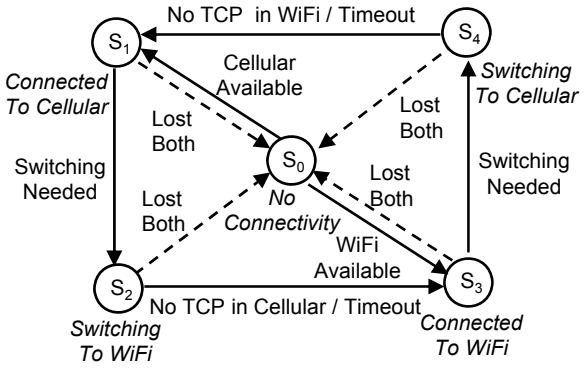


Figure 8.7: MultiNets State Diagram.

The Switching Utility

The Switching Utility provides the utility methods to perform the switching. It includes the following tasks – counting the ongoing TCP sessions over a specific network interface, updating the routing table to keep the existing TCP sessions over a specific interface, adding and deleting default routes of the network interfaces, and connecting, re-connecting or tearing down interfaces. These methods are called by the core switching module to perform a switch.

8.4.2 The Monitoring Engine

The Monitoring Engine is responsible for monitoring all the necessary phenomena pertaining to switching. It contains several different monitors, each of which observes one or more system variables, and holds the latest values of those variables. We have implemented 4 monitors – *(i) Data Monitor:* Monitors the amount of transmitted and received data

over WiFi and cellular interfaces in bytes and packets since the interface is turned on, *(ii) Wireless Monitor:* Monitors the connectivity status, signal strengths, and information of access points, *(iii) Network Flow Monitor:* Monitors the number and state of all TCP and UDP sessions, routing information from the routing table, and *(iv) Power Monitor:* Monitors the state of the battery and its voltage, current, and capacity. All these monitors are singleton and are created in an on-demand basis. They have a common interface to answer to all the queries. The query and its response form a $< key, value >$ pair. The Selection Policy component (see Section 8.4.3) issues these queries. The modular design of the monitors and a common interface to talk to them allow us to add new monitors into the system and to extend the capability of the existing monitors easily.

8.4.3 The Selection Policy

The Selection Policy defines the policy for interface switching. By separating the policies from the rest of the system, we are able to add new policies or modify the existing ones without requiring any change to the other parts of the system. For example, one of our policies is based on the fact that WiFi is much faster than the cellular network. But in future, this situation may change and cellular data connectivity may outperform WiFi and thereby requiring a change in current policy or adding a new policy that leverages that. Currently, we have developed and implemented three policies which are described next. Only one of these policies is active at a time. The user of MultiNets determines which policy is to be used.

Energy Saving Policy

The aim of this policy is to minimize the power consumption. We describe an optimum energy saving algorithm in Appendix A which assumes that we have the knowledge of future data traffic. For a realistic setup, we propose a switching heuristic, which is inspired by our energy measurements. According to this policy, the mobile device connects to the cellular network when it is idle, and starts to count the number of bytes sent over the cellular network after the user launches an application. As soon as the total amount of data over the cellular network exceeds a threshold τ, the device decides to switch to WiFi. The mobile device switches back to the cellular network once the WiFi network is idle for ζ seconds. We empirically derive the best τ and ζ values in Section 8.8.1. This policy levarages one fact that the idle power of WiFi is much higher than that of cellular. Techniques such as [100, 101] may save some part of the energy that is consumed for scanning WiFi APs– which accounts for about 40% of the idle energy. But even after applying such techniques, WiFi's idle power remains more than 50 times higher. Hence, switching interfaces dynamically is a better option to save energy.

Offload Policy

The aim of this policy is to offload cellular data traffic to any available WiFi network. According to this policy, whenever WiFi is available, we switch to WiFi. We only switch back to the cellular network when WiFi's signal strength is dropping below a threshold η dBm. The advantage of this policy is to reduce data traffic on cellular networks. But the downside is that, if the network is not being used, only to keep the WiFi interface idle is more expensive in terms of energy.

Performance Policy

The aim of this policy is to maximize the network throughput. It achieves this by switching to the interface with the highest bandwidth. Let, $B_W(s)$ and $B_C(s)$ be the bandwidth functions for WiFi and cellular networks respectively where s denotes the signal strength, which is read via Android system API. We empirically derive the bandwidth functions $B_W(s)$ and $B_C(s)$ in Section 8.8.3 through extensive experiments. The performance policy compares the values of these two functions every δ seconds, and switches to the network interface with the higher bandwidth.

8.4.4 Layered Implementation

We have closely studied the software architecture of the data connectivity in Android. Like Android, the implementation of MultiNets is layered. Classes and methods of our system that are similar to those of Android are implemented at the same layer. Yet, our system is vertically distinguishable from Android as shown in Figure 8.8.

At the bottom of the architecture, we have the unmodified Linux Kernel. Right above the Kernel, we have a layer of native C/C++ modules that perform the lower level tasks of file I/O to get all the information used by the Monitors and some socket I/O to add, remove or update routing table entries. We improve the implementation of Android's `ifc_util.c`, `route.c`, and `netstat.c` by adding these non-existing modules and put them into our own module `swi_utils.c`. But no changes are made into the network protocols. These modules are wrapped by JNI and are called from the Internal Classes layer. The Switching Service, Monitoring Service and Selection Policy are implemented as system services at the Internal Classes layer. These services are created during the device start-up and they run as long as the device is running. We have modified the Android's System Server to start these services when the device starts. All the changes are done by adding 209 lines of C/C++ code and 650 lines of Java code to Android (Eclair 2.1).

8.4.5 The Switching API

We provide API to configure and control the Switching Engine which is described in Table 8.1. We use this API to extend Android's built in wireless control settings application so that the Switching Engine can be stopped, restarted and configured to run in different modes from the application layer. When the engine is stopped, this API can also

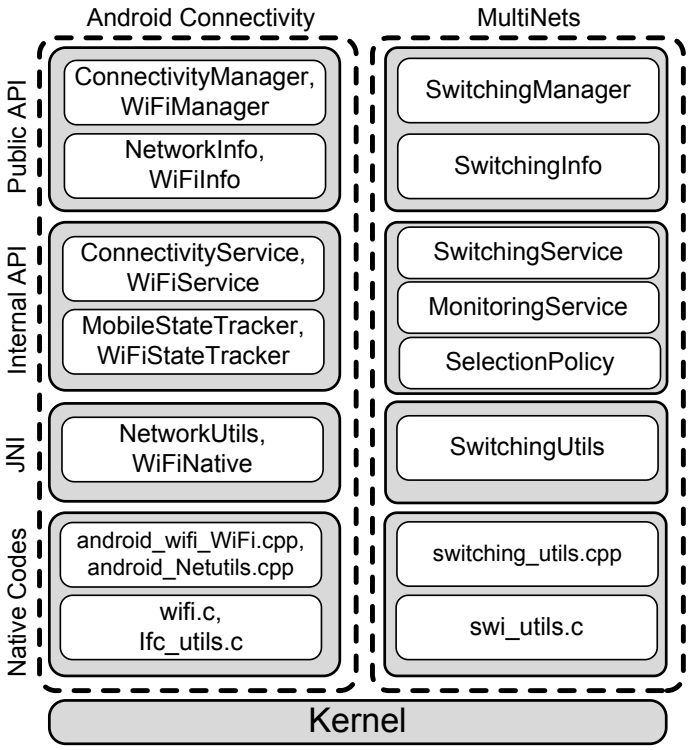


Figure 8.8: Layered Implementation of MultiNets.

Method	**Description**
getInfo	Returns the status, state and current policy.
activateEngine	Activates or deactivates the engine.
setPolicy	Sets the current Selection Policy.
switchInterface	Request to switch to a particular interface.
useInterface	Request to use a specific interface.

Table 8.1: Description of SwitchingManager API.

be used by the application programmers to switch interfaces, send a specific flow using a specific interface, or to use multiple interfaces simultaneously.

Two of the methods in the API are very useful from the application programmers' point of view. The first one is, the `switchInterface()` method, which allows the programmers to switch interface when needed. This is useful for those kinds of applications that need to send sensitive data (e.g. user credentials) over the cellular network, but for all other purposes prefer to be on WiFi. Another important method is, the `useInterface()` method. It is useful to send or receive data using a specific interface for a specific connection. Note that, it does not switch the interface, rather if the preferred network is available, it sends the data using that interface for the specified connection only. With this method, an application can use multiple interfaces simultaneously.

```
1  SwitchingManager mgr = new SwitchingManager(getSystemService("SM"));
2  String ip = "12.71.54.184"; int port1 = 5050, port2 = 5051;
3  InetAddress addr = InetAddress.getByName(ip);
4  try {
5      if (mgr.useInterface(addr, mgr.MOBILE)){
6          Socket ms = new Socket(ip, port1);
7          // transfer sensitive data over secured cellular network.
8      }
9     if (mgr.useInterface(addr, mgr.WIFI)){
10         Socket ws = new Socket(ip, port2);
11         // transfer less sensitive data over public WiFi.
12     }
13 } catch(SwitchingException ex){ ex.printStackTrace(); }
```

Figure 8.9: Using useInterface() method.

Figure 8.9 shows an example usage of this method. Both of these methods are requests to the switching system. The requests may fail if the application does not have proper permissions or the Switching Engine is currently running.

8.5 Experimental Setup

Our evaluation consists of three sets of experiments. First, we measure the system overhead (Section 8.6) and switching time overhead (Section 8.7) to determine the system parameters, and energy consumption of WiFi and 3G interfaces (Section 8.7.1). These data that are used in the later experiments. Second, we discuss a set of experiments (Section 8.8) that are trace based, where we apply the three policies on the 3 month long collected data traces to demonstrate the benefits of switching. Third, we demonstrate the performance of MultiNets in a real world scenario (Section 8.9).

8.5.1 Hardware Setup

All experiments are performed on multiple Android Developer Phones 2 (ADP2) [102]. The mobile devices are running MultiNets which is developed on top of Android OS (Eclair 2.1-update 1). The devices are 3G-enabled T-Mobile phones that use 3G, EDGE, GPRS and WiFi 802.11 b/g connectivity and are equipped with an 528 MHz ARM processor, 512 MB flash memory, 192 MB RAM, and 1 GB microSD card.

8.5.2 Software Setup

We have used benchmarks, data traces from real users, and real usage of our system as our workloads. We have used a number of benchmarks that are available in the market, such that they as a whole exercise different aspects of the system. Although these benchmarks are useful to evaluate the overhead of the system, we find none of them useful for evaluating the performance of the connectivity of the phone. Driven by this need, we have developed a *data logger* that

is capable of logging various important information of the running applications within the phone periodically and send it to a remote server. 13 volunteers from our research lab, including research scientists, graduate students, faculty, and staffs of age group 25 to 35, were equipped with these phones with our data logger and they carried around the phones to wherever they wanted and used them for both voice and data connectivity for 90 days. The information that we collect from these logs include the names and types of the applications, the frequency and the duration of their usage, and the data usage information for each wireless interface for each user. For each of these applications, we have the total number of bytes and packets transmitted and received over cellular and WiFi. We modified the Linux's `netstat` tool for Android to get the information about all the TCP and UDP sessions, which include IP addresses, ports, start time, and durations.

We then implement a *traffic generator* to reproduce the data sessions. The traffic generator replays the exact same sessions as that are in the log except for that they are now using different server IPs which are situated in our lab instead of the original ones. We load the information about all the sessions into the traffic generator running on the phone and start a process that replays those sessions.

8.6 System Overhead

The Switching Engine starts several background system services at the device startup. Running such system services may add additional overhead to the system. The goal of this experiment is to derive a minimal sleeping interval for the monitoring services so that their overhead is reasonable. We run a set of benchmarks on the device, with and without the Switching Engine and compare the two scores. None of these benchmarks use any data connectivity and, hence, no switching happens during this experiment. The overhead is due to the engine's continuously monitoring and checking for an opportunity to switch only. We use seven sets of benchmarks that are available in Android Market that has been downloaded $10,000$ to $50,000$ times. Table 8.2 describes the benchmarks.

Benchmark	**Description**
Linpack	Solves dense NxN system of linear equations.
Fps2d	Measures 2D graphics frames per second.
CMark	Measurements performance of java programs involving prime generation, loop, logic, method, and floats.
Graphics	Draws opacity and transparent bitmaps.
Cpu	CPU performance of MWIPS, MFLOPS, and VAX MIPS (SP and DP).
Mem	Memory copy operation.
File	File create, write, read, and delete.

Table 8.2: Description of the benchmarks.

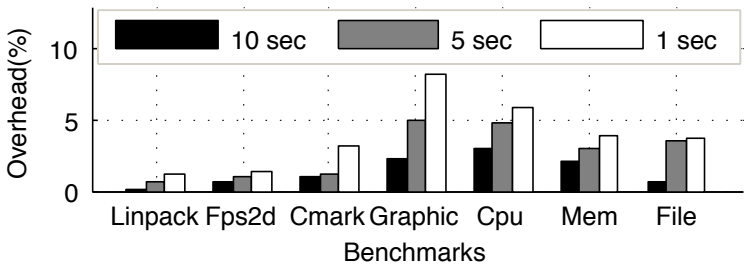


Figure 8.10: A monitoring interval of 5 sec or more keeps the overhead below 5%.

Figure 8.10 shows the benchmark scores of the device for running the Switching Engine at 10, 5, and 1 seconds sleeping intervals. The scores are normalized to the scores achieved by a phone running original Android. We see that the more the sleeping interval, the smaller the overhead and the closer the score is to that of without running the engine. But if this interval is large, the responsiveness of the engine becomes lazy. We therefore run the engine in 5 seconds interval which keeps the overhead below 5% and at the same time the responsiveness is also good. Note that, this overhead is due to the polling style implementation of the monitoring engine and is not an inherent problem of the switching technique itself. A more efficient implementation of the engine is left as our future work.

Lines of code	Added	Modified
C/C++	209	0
Java	642	8

Table 8.3: Lines of code.

8.7 Switching Time

The switching time is the duration between the instant when the engine decides to switch and the instant when it completely connects to the new interface and disconnects the old one.

Figure 8.11 illustrates an example of switching. In this scenario, we start sending data over the 3G, and switch to WiFi when the data rate over 3G exceeds 16 KBps, and switch back to 3G when the WiFi is idle for the last 30 seconds. The timeout for all ongoing TCP sessions over the old interface is set to 20 seconds. The parameters chosen for this experiment are for the demonstration purpose only, they are not set for optimizing anything. The top figure shows the data rate over 3G and the bottom one shows the same for WiFi. We start using 3G for browsing various web pages at t_0. At t_1, when the Monitoring Engine detects that the threshold of 16 KBps is exceeded, the Switching Engine decides to switch to WiFi. It turns on the WiFi connectivity and guides all new sessions to start over WiFi while keeping old sessions active over 3G. This continues till t_2, when the timeout occurs and all existing TCPs over 3G are closed. The duration of $[t_1, t_2]$, is the switching time during when both the interfaces have one or more active TCP sessions.

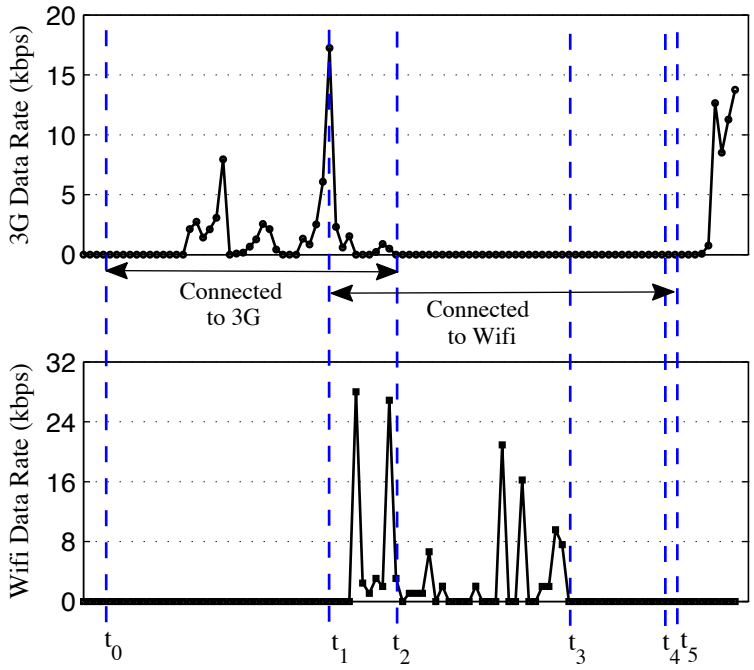


Figure 8.11: Both interfaces are active during the switching times $[t_1, t_2]$ and $[t_4, t_5]$.

After t_2, the phone is connected to WiFi only and continues at this state until it discovers at t_4 that it has been idle since $t_3 = t_4 - 30s$. At t_4, the engine again initiates a switching from WiFi to 3G. Since this time we do not have any ongoing TCP over WiFi, the switching to 3G happens almost immediately at t_5.

Type	**Switching time (msec)**
3G to WiFi	1212
WiFi to 3G	196

Table 8.4: Minimum time to switch to WiFi is 6 times higher than switching to 3G.

Table 8.4 shows the measured switching time overhead. In this case, we do not send or receive any data over any interface. Switching to WiFi takes about 1 second more than switching to 3G. This is because, connecting to WiFi goes through a number of steps involving scanning for access points, associating with one of them, handshaking and a dhcp request, which are not required for 3G.

Figure 8.11 reveals that the timeout (20 secs) is the principal component that determines the switching time. To prevent TCP interruptions, we should set the timeout to infinity. But doing so would increase the energy consumption as both interfaces are on during the transition time. Hence, we conduct several experiments to quantify the trade off between switching time and number of interrupted TCP sessions.

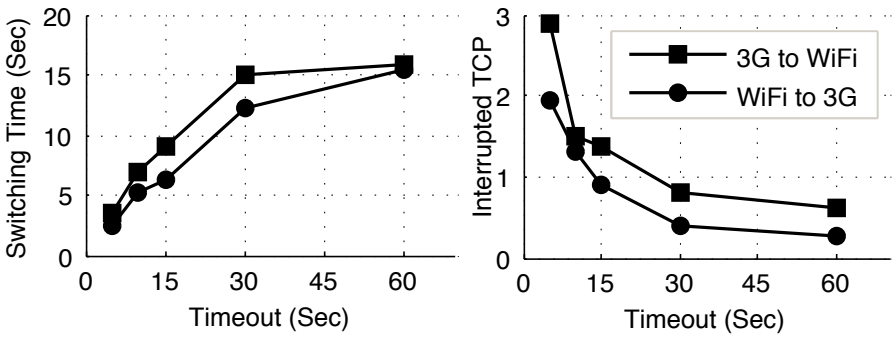


Figure 8.12: Timeout value of 30 sec or more makes the switching time ~ 15 sec, and keeps the TCP disconnections < 1.

Figure 8.12(left) shows the switching time in presence of data activity for different timeout values. We see that the switching times are closer to the corresponding timeouts for values < 5 seconds. As most of the TCP sessions are short-lived and they finish in $10 - 15$ seconds, towards the right of the figure this difference is higher. There are still some sessions that remain till the end of timeout even for values > 30 seconds, but they are small in number. Figure 8.12(right) plots the number of TCP sessions that gets disconnected against varying timeouts. We see that this number becomes less than 1 after 15 seconds and continues to get lower with increasing values. These long sessions are the ones that the applications use to communicate with their servers and any interruptions of these automatically initiate new connections with the server, and hence there are no visual interruptions for this. Yet, we set the timeout to 60 seconds to reduce the number TCP disconnections and maintain an average switching time of 15 seconds.

8.7.1 Energy Measurements

We used a high precision digital multimeter [103] to obtain fine-grained energy measurements once every 1 msec as shown in Figure 8.13. To measure the power drawn by the battery, we opened the battery case to place a 0.1 ohm resistor in series with the battery. The voltage drop across the resistor was used to get the battery current, and this current and the input voltage to the phone were used to calculate the energy consumption. The battery charger was disconnected to eliminate interference from the charging circuitry during this measurement.

Figure 8.14 shows the average energy consumption for both downloading and uploading of 4 KB to 4 MB data over 3G and WiFi networks. Our experiments were performed using a remote server running Linux 2.6.28 with a static IP address located in our lab. We repeated the experiment ten times for each data size and averaged the results. The standard deviation for all these measurements is less than 5%.

For all experiments, we configured the smartphone screen brightness to minimum. We measured the average idle power consumed by the 3G and WiFi network interfaces and found them to be 295.85 mW in case of WiFi and 3.11 mW in case of 3G. In addition, the average energy costs of turning on WiFi and 3G interfaces are 7.19 J and 13.13 J,

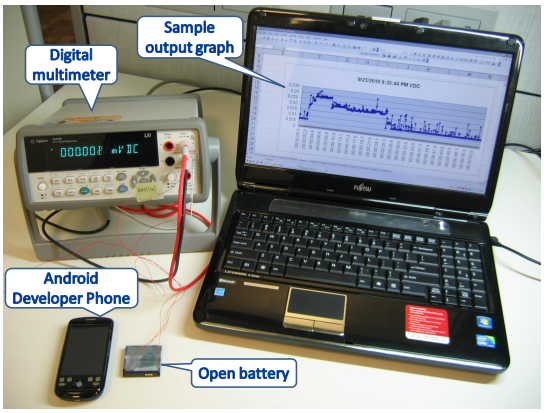


Figure 8.13: The digital multimeter, an open battery used for energy measurements, and a sample output graph is shown.

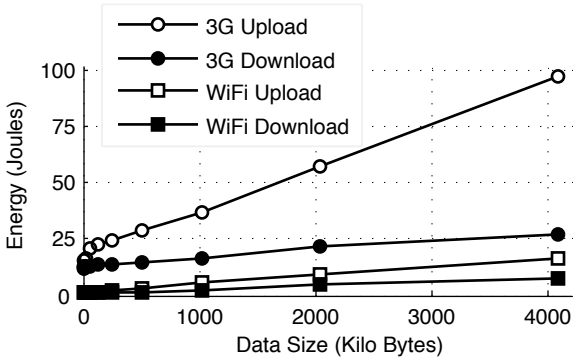


Figure 8.14: Energy consumption during data activity is higher for 3G than WiFi, and uploading is more costly than download.

respectively. We subtract these numbers during the energy measurements in Figure 8.14 and report only the energy costs for data transfers.

Every data transfer request contains an energy overhead of 12 J for 3G and 1 J for WiFi. After the connection has been established, the energy used to both upload and download is increasing with the amount of data being uploaded or downloaded. These observations are consistent with previous work [104, 105]. Figure 8.14 shows that the energy needed to upload data is higher than that for download since the upload bandwidth is typically smaller. Similarly, because of lower bandwidth of 3G, 3G consumes more energy than WiFi for the same amount of data transfer.

We use the energy measurements for data transfers, turning and keeping on the interfaces to obtain a simple energy model. The energy model is used to estimate the energy for downloading or uploading varying data sizes across 3G and WiFi networks without any other activity on the mobile device. We model the energy cost as follows:

$$E = E_{ON} + E_D(d) + \bar{P} \times (d/\bar{R}),$$

where E_{ON} is the energy to turn on the interface, $E_D(d)$ is the energy to transfer d bytes of data, $\bar{P}$ is the average power to keep the interface on, and $\bar{R}$ is the average data rate. We determine $E_D(d)$ by using linear interpolation and extrapolation on sample points in Figure 8.14.

8.8 Trace Driven Experiments

8.8.1 Energy Efficiency

Using the energy model as described in the previous section, we estimate the average daily energy consumption for each user in our data traces. We have considered data transfer over WiFi and 3G, and also considered the idle power. We then compute the optimum energy consumption of each user assuming they switched optimally. We use dynamic programming to get this optimum value, which is described in Appendix A. While the algorithm achieves the optimum energy consumption, it assumes that the future data usage is known, which is not realistic. Therefore in MultiNets, we use a simple heuristic to switch interfaces. As data communication in WiFi is cheaper, for switching from 3G to WiFi, we use a data threshold of τ KB. If the phone crosses this limit, we switch to WiFi. On the other hand, since idle power of WiFi is much higher than 3G, we switch the phone back to 3G when data activity is absent over WiFi for the last ζ seconds. We systematically tried various τ and ζ values using the data traces, and found that $\tau = 4$ KB and $\zeta = 60$ seconds minimizes the deviation from the optimum energy saving. Therefore, we use these two values throughout the chapter if not otherwise specified.

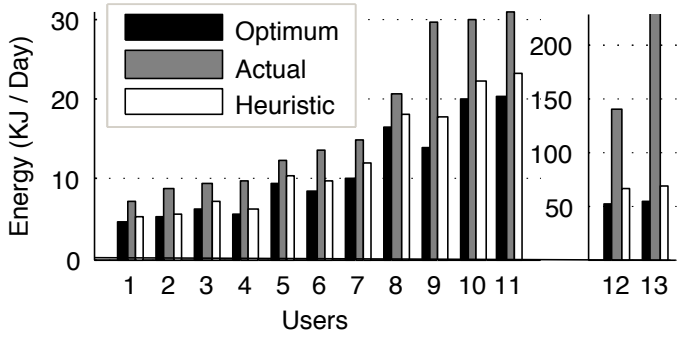


Figure 8.15: Energy saving heuristic cuts down the average daily energy usage by 27.4% and is close to optimum.

Figure 8.15 shows the average daily energy consumptions of all the users for three strategies: optimum, actual and the heuristic that we use in MultiNets. This figure shows that switching optimally saves on average 24.17 KJ energy per user per day, which is as high as $89 - 179$ KJ for some users (e.g., 12, 13). We also see that our simple heuristic achieves near-optimal energy consumption with an average deviation of only 13.8%, and we are able to cut down the daily energy usage by 27.4% (21.14 KJ) on average.

8.8.2 Offloading Traffic

In order to estimate how much data traffic we are able to offload from 3G to WiFi network with MultiNets, we analyze the data traces that we have collected. For each user, we compute the average daily WiFi usage and compare it to the amount of data that is possible to offload if MultiNets was used. Our offloading strategy is to switch all 3G traffic to WiFi whenever we find a connectible access point. We consider an access point connectible if and only if its signal strength s is above $\eta = -90$ dBm and has been used by the user in the past. The threshold -90 dBm is derived empirically. When the signal strength is below it, the WiFi is not usable.

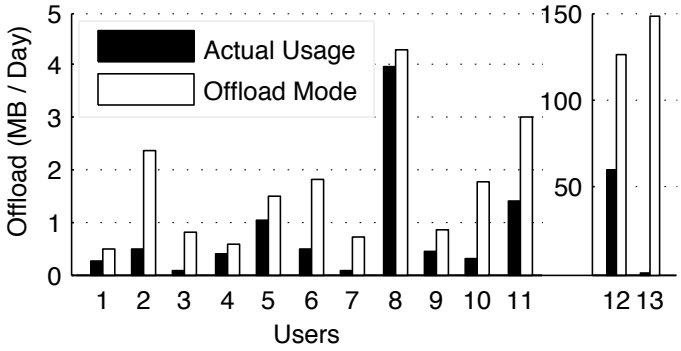


Figure 8.16: An average of 22.45 MB more data per day per user is offloadable using dynamic switching.

Figure 8.16 shows this comparison for each of the users. We see that, for some users (e.g., 3, 7) we are able to offload $11 - 14$ times more data, and for some users who does not tend to use available WiFi at all (e.g., 13) this difference is about 150 MB per day. Considering all users, with switching, we are able to offload on average 22.45 MB of data per day per user which is 79.82% of the average daily usage (28.13 MB).

8.8.3 Throughput

Performance of web applications get a significant boost by switching to the interface with the higher bandwidth. In our data trace, we have recorded the signal strengths of both the cellular and all available WiFi networks at 30 seconds intervals. We conduct extensive measurements using `iperf` tool to find the correlation between signal strength s and bandwidth B. We run `iperf` server on our server, and `iperf` client on Android phones, and `iperf` packets traverse through the Internet. We have taken measurements in both indoor and outdoor environments and report the average of 10 measurements at varying signal strengths in Table 8.5. We define the bandwidth function $B_W(s)$ (for WiFi) and $B_C(s)$ (for cellular) using linear interpolations on measurement samples in this table.

Using the bandwidth functions, we calculate the average daily throughput of each user for his actual usage, and we also do the same if MultiNets was used. Figure 8.17 shows that, with MultiNets, it is possible to achieve an average throughout of 2.58 MBits/sec, which is 7 times more than the actual usages. For some users (e.g., $1, 2, 4, 13$) this gain

WiFi		HSPA+		3G	
Signal (dBm)	**Bandwidth (Mbps)**	**Signal (dBm)**	**Bandwidth (Kbps)**	**Signal (dBm)**	**Bandwidth (Kbps)**
≤ -50	8.58	-65	929	-63	138
$(-50, -60]$	7.06	-73	858	-89	115
$(-60, -70]$	6.16	-89	746	-101	104
$(-70, -80]$	3.99	-97	509		
> -80	1.25				

Table 8.5: Bandwidth of WiFi, HSPA+, and 3G at different signal levels measured by `iperf`

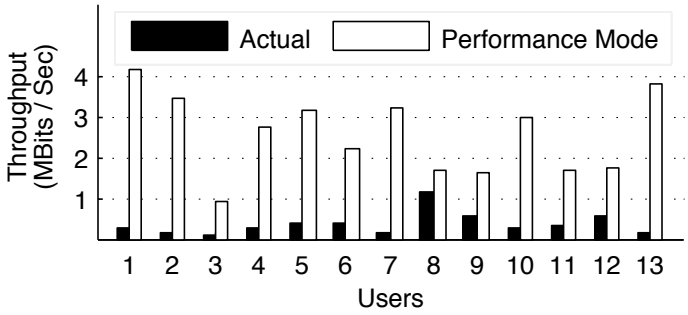


Figure 8.17: The achievable throughput is 7 times higher with switching.

is about $14 - 24$ times. These are the users who tend to remain in the 3G network even if WiFi is available for them to connect.

Note that, based on our measurement results, even if we take decisions to switch based on the signal strengths, state-of-the-art 3G network being always slower than WiFi, the policy selects WiFi almost as if it were in Offload mode. This is, however, not always true, e.g., with the rapid advancement of cellular data network technology, this gap is diminishing. Our measurements with recent High Speed Packet Access Network (HSPA+) in Table 8.5 shows that this network is about eight times faster than 3G. We believe, in near future, cellular networks will have a comparable bandwidth to WiFi, and the performance mode of MultiNets will have a higher impact at that moment. Finally, measurement studies report that WiFi throughput may be lower than 3G throughput under certain practical circumstances [36].

8.9 Deployment Experiment

To quantify the performance of our system in a real-world scenario, we conduct actual experiments at Stanford University campus. We have chosen this campus since it has WiFi connectivity both inside and outside of the buildings and also has several areas where WiFi is either completely unavailable or has a very poor signal strength. High availability of WiFi is important for us since we want to demonstrate that our system is switching back to 3G to conserve energy even

in presence of WiFi. On the other hand, loss and reconnection of WiFi connectivity is important to demonstrate that our system is capable of switching smoothly.

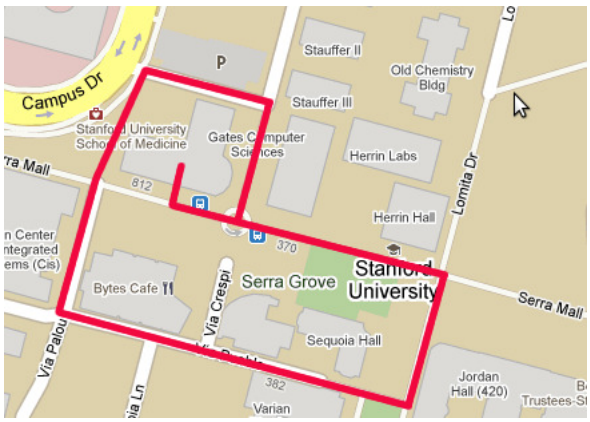


Figure 8.18: We encounter 37 WiFi APs, average signal strengths of -68.46 dBm (inside) and -82.34 dBm (outside), and 28 WiFi disconnections during the tour.

We take 4 ADP2 phones with us. Two of these have our system installed and the other two run Android (Eclair 2.1). All 4 phones are fully charged and their screen brightnesses are set to the lowest level. For a fair comparison, we use our traffic generator to replay the same data traffic in all of them. The traffic generator runs in the phone and sends and receives data over the Internet to and from our server which is situated in our lab at 4 miles distant from the campus. The phones replay the traffic patterns of the most popular 6 applications from our data traces having sessions of varying numbers, durations, delays and concurrencies. Once started, the phones run each of these applications for 10 minutes followed by a 10 minutes break, repeatedly one after the other. We log the transmitted and received bytes, signal strengths, MAC addresses of WiFi APs, battery current, voltage, and capacity into the file system of the phone every 2 seconds for later analyses.

8.9.1 Energy Efficiency

In this experiment, we configure one of our phones into the energy saving mode. We take another two phones that run Android– one with WiFi enabled, and the other staying over 3G only. We start the traffic generator in all 3 phones and begin our 168 minutes long campus tour starting from the Computer Science building. We move around all 5 floors of the building for an hour, then take an hour long round trip tour within the campus, and finally get back to the building to spend the rest of the tour as shown in Figure 8.18. During this tour, we encounter 37 different WiFi APs, an average signal strength of -68.46 dBm inside the building and -82.34 dBm outside the building, 28 disconnections from WiFi to 3G, and a total of 49 switchings by our system.

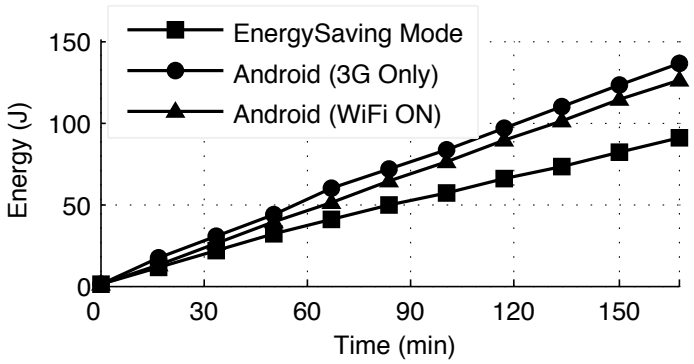


Figure 8.19: Energy saving mode saves about $28.4\% - 33.75\%$ energy as compared to Android.

Using the instantaneous values of current and voltage obtained from the log, we compute the energy consumption of each of these phones and plot the cumulative energy consumption in Figure 8.19. Despite the fact that the battery voltages and currents read from the Android system are not in high precision, we still see a clear difference of the energy consumption among these 3 phones. We see that for the same data traffic, our system achieves about $28.4\% - 33.75\%$ energy savings as compared to state of the art Android systems.

8.9.2 Offloading and Throughput

In this experiment, we compare the offloading and throughput of our system with those of the state-of-the-art Android. In MultiNets, we set a switching timeout of 30 seconds. Recall that, cellular network being much slower than WiFi, the outcomes of Offload and Performance modes are the same, although their decision mechanisms are completely different. Therefore, we present them together in Table 8.6. This table also gives the achievable lower and upper bounds of Android on offloading and throughput. The lower bound is derived by disabling WiFi interface (3G Only), and the upper bound is achieved by always switching to WiFi whenever it is available (WiFi ON). This table shows that: MultiNets (i) leads to three times higher throughput than the Android lower bound, (ii) achieves near-optimal offloading and throughput, and (iii) experiences *zero* TCP disconnections throughout the experiments, while Android upper bound results in *eight* TCP disconnections.

System	Offload (MB)	Throughput (kbps)	Disconnections (Count)
MultiNets	45.41	116.20	0
Android (3G Only)	0	39.29	0
Android (WiFi ON)	44.54	116.26	8

Table 8.6: For near-optimal offloading and throughput, MultiNets experiences no TCP disconnections throughout the experiments.

8.9.3 Energy Efficiency vs. Offload Trade-off

It is interesting to see the trade-offs between the energy savings and offloading. Table 8.7 shows that, MultiNets in energy-saving mode consumes about 55.85% less energy than offload mode, but sacrifices about 14.25% of offloading capability. The reason behind is that, energy saving mode keeps the phone in 3G while it is idle. When data transmission starts, it keeps the phone in 3G mode for a while before completely switching to WiFi, and hence the overall WiFi offloading is slightly lower in this case. This experiment illustrates that, users of MultiNets achieve different objectives by putting the system in different modes.

Mode	**Energy Consumption (J)**	**Offload (MB)**
Energy Saving	90.36	38.94
Offload	204.65	45.41

Table 8.7: Energy saving mode saves 55.85% more energy, but sacrifices 14.25% offloading.

8.10 Summary

This chapter describes the design, implementation, and evaluation of MultiNets, which is a service that monitors the wireless data usage on a mobile device and dynamically switches the network interfaces (WiFi and 3G) based on some predefined policies, such as energy efficiency, offloading data to WiFi, and maximizing throughput. MultiNets brings efficiency in wireless data communication to mobile applications that quite frequently communicates to remote servers over the Internet. Our analysis on mobile data traces collected from real users shows that with dynamic switching we can save 27.4% of the energy, offload 79.82% of the data traffic, and achieve 7 times more throughput on average. MultiNets runs independent of Auditeur, and is not part of the core Auditeur platform. However, MultiNets is complementary to Auditeur. Energy savings achieved by using MultiNets adds to the energy savings achieved with Auditeur– as the former optimizes the energy consumption due to mobile-cloud communication and the later optimizes the energy consumption due to on-device computation.

Chapter 9

A Special Type of Acoustic Event

In this chapter, we deal with a special type of acoustic event – which is our *heartbeat*. In Chapter 6.4, we briefly introduced one such application which was created using the Auditeur platform. The application considers each heartbeat as an acoustic event and counts the total number of events over a 10 seconds period with an accuracy of over 95%. However, some of the details, such as – *how we obtained the audio signals containing heartbeats*, and *what the purpose of that application was*, has been left out in that chapter. This chapter fills those gaps by describing – the wearable hardware platform which we built and used to obtain heartbeat signals from the ear, a heart rate measurement algorithm which is more accurate and more robust than an Auditeur-powered application, and the development and evaluation of a complete application which uses heart rate as a means to recommend the next song to play on a mobile device, called – the *Musical-Heart*.

Musical-Heart is a biofeedback-based, context-aware, automated music recommendation system for smartphones. We introduce a new wearable sensing platform, Septimu, which consists of a pair of sensor-equipped earphones that communicate to the smartphone via the audio jack or Bluetooth. The Septimu platform enables the Musical-Heart application to continuously monitor the heart rate and activity level of the user while he is listening to the music. The physiological information and contextual information are then sent to a remote server, which provides dynamic music suggestions to help the user maintain a target heart rate. We provide empirical evidence that the measured heart rate is $75\% - 85\%$ correlated to the ground truth with an error of < 2 BPM when the user is stationary, and 7.5 BPM on average. Such an error may not be desirable in applications that require a precise heart rate measurement. For those applications, we recommend an IR-based heart rate detector which is more accuate than an acoustic-based one. The accuracy of the person-specific, 3-class activity level detector is on average 96.8%, where these activity levels are separated based on their differing impacts on heart rate. We demonstrate the practicality of Musical-Heart by deploying it in two real world scenarios and show that Musical-Heart helps the user achieve a desired heart rate intensity with an

average error of less than 12.2%, and its quality of recommendation improves over time.

9.1 The Musical-Heart Application

Exercise and heart health are so closely related that it is common to see modern workout equipment (e.g., treadmill or elliptical machines), and jogging accessories (e.g., wrist watches or music players) have built-in receivers that continuously receive heart rate information from measurement devices worn on the chest, wrist, or finger. There are, however, many limitations of these heart rate monitoring devices. First, the user has to carry an extra device while exercising, which is an inconvenience. Second, some of these devices require the user to wear a chest strap, which is not only uncomfortable, but also requires extra effort to put on and take off. Third, the best of these devices cost about $\$400$, which is prohibitively expensive for the average person.

We propose a convenient, non-invasive, personalized, and low-cost wellness monitoring system, designed to obtain heart rate and activity level information from a pair of specially designed earbuds while the user listens to music on a smartphone. An intelligent application on the phone uses physiological and activity level information from sensors to recommend, play, and share appropriate music for the user's desired intensity level. The system is *convenient* since the sensors are embedded into the earphone and the user does not have to carry (or worry about forgetting to carry) any additional device. The system is *non-invasive*, since wellness monitoring comes as an additional feature of the earphone that most of the smartphone users already wear in order to listen to music. The system is *personalized* – the user sets his or her personal goal (such as a target heart rate zone) and the device recommends and plays appropriate music– considering his current heart rate, target heart rate, past responses to music, and activity level. The system is *low-cost* – the cost is lower than other standard heart rate monitoring hardware because it is embedded in the earbud.

We offer a complete system comprised of both hardware and software components. We introduce the *Septimu* platform, which consists of a pair of sensor equipped earphones and a baseboard. Both earbuds contain an inertial measure- ment unit (IMU), an analog microphone and an LED. The baseboard drives the sensors and communicates to the smartphone via the audio jack or Bluetooth. An intelligent music player application, *Musical-Heart*, runs on the smartphone and periodically samples the accelerometer and microphone to determine the activity level and heart rate of the person. The information is sent to the server over the Internet, and the server uses it to recommend appropriate music that will help the user maintain their target heart rate. The system is also able to predict the heart's response and enables sharing and streaming of this situation-aware music.

In order to address the challenges of recommending music based on physiological data in real time, we have developed three main algorithmic solutions that will be outlined in this chapter. First, a fast, accurate and real-time heart rate measurement algorithm is described, which extracts heartbeats from a mixture of acoustic signals from the earbuds. The algorithm is noise resistant, threshold free, and built upon dynamic programming principles to find the

optimum solution in real time. Second, a simple yet highly accurate, person-specific, 3-class activity level detection algorithm is described, which leverages the stable accelerometer readings from the earbuds to detect activity levels and posture information in real time. The goal of the activity detector in this work is not to identify specific activity types, but their categories that have an impact on heart rate. However, the algorithm is further enhanced by opportunistically incorporating contextual information – such as the location and speed of the user – from other in-phone sensors. Third, a novel control-theoretic approach for recommending music is described, which constructs a personalized model of each user's responses to different music at different activity levels and uses the model to suggest the best music to satisfy a given heart rate goal.

We design and implement a three-tier architecture for Musical-Heart. Tier 1 is the firmware, which is written in nesC (TinyOS) and runs on the Septimu baseboard. Its role is to drive the sensors, perform on-board processing to reduce data traffic to the phone, and maintain communication with the phone. Tier 2 is the smartphone application, which is written in Java (and native C) and runs on the Android OS. It implements the key algorithms and manages the communication with the server. Tier 3 consists of a set of RESTful web services for extracting and matching music features, music recommendation, sharing, and streaming. We thoroughly evaluate the system components and the algorithms using empirical data collected from 37 participants for heart rate detection and 17 participants for activity level inference. We demonstrate the practicality of Musical-Heart by deploying it in two real-world scenarios and evaluate its performance on 4 users. The main contributions of this chapter are:

- *Septimu*, the first wearable, programmable hardware platform designed around low-cost and small form-factor IMUs and microphone sensors that are embedded into conventional earphones, and communicate to the phone via the audio jack.

- *Musical-Heart*, a complete sensing system that monitors the user's heart rate and activity level – passively and without interrupting the regular usage of the phone – while the user is listening to music, and recommends songs based on the history of heart's response, activity level, desired heart rate and social collaboration.

- We devise three novel algorithms: (1) a threshold free, noise resistant, accurate, and real-time heart rate measurement algorithm that detects heart beats from a mixture of acoustic signals from the earbuds, (2) a simple yet highly accurate, person-specific, 3-class activity level detection algorithm that exploits accelerometer readings from the earbuds, and (3) a PI-controller that recommends music to the user based on past history of responses to different music and helps maintain the target heart rate at different activity levels.

- We perform an empirical study by collecting ground truth data of heart rates, and summarizing it to show the effect of music on heart rate at various activity levels. The dataset is further used to show that the detected heart rate is $75\% - 85\%$ correlated to the ground truth, with an average error of 7.5 BPM. The accuracy of the

person-specific, 3-class activity level detector is on average 96.8%, where these activity levels are separated based on their differing impacts on heart rate.

- We demonstrate the practicality of MusicalHeart by deploying it in two real world scenarios, and show that MusicalHeart helps the user in achieving a desired heart rate intensity with an average error of less than 12.2%, and the quality of recommendations improves over time.

9.2 Usage Scenarios

We describe two motivating scenarios of Musical-Heart which are realized and evaluated later in Section 9.8.

9.2.1 Personal Trainer: Goal-Directed Aerobics

Alice exercises regularly. Today she wants to practice cardio exercise by going for a jog. She starts the MusicalHeart application on her smartphone and sets her goal to *cardio*. The system initially detects that Alice is standing still and her heart rate is in the healthy zone (50% of maximum rate). The application creates a playlist of songs dynamically and starts playing one that helps Alice to warm up and attain the heart rate for cardio. Alice then starts jogging while the application keeps monitoring her heart rate and activity level. After a few minutes, the application automatically detects that Alice's activity level has changed to a higher level. It dynamically adjusts the playlist according to the new activity level and current heart rate, so that the next song played is suitable for Alice's changing exercise intensity. The system keeps doing this until Alice's jogging session is over. In the end, it adds one more entry into Alice's health journal that keeps her up-to-date about her heart health and exercise history.

9.2.2 Music Recommendation: Biofeedback and Collaboration

Bob is a middle-aged person who takes the bus on his way home every evening. He enjoys listening to music during this idle time to get some relief after a long tiring day. But he is bored with all the songs that are on his mobile device since he has listened to them at least a hundred times. Today he wants to try some new music, but it has to be something appropriate for the moment– a calm and heart soothing song. He opens up the MusicalHeart application and notices that it already knows about his activity level and context (i.e. traveling). The application connects to the server and obtains a list of melodious and heart softening songs– including some of Bob's favorite titles and novel songs shared by other travelers who enjoy similar music at this activity level. Bob wants to try one of the new titles and the sever starts streaming the song. While Bob is enjoying the new songs, the application is continuously monitoring his heart's responses and reporting the information back to the server for use in future recommendations.

9.3 System Architecture

We present a brief description of the system architecture of Musical-Heart in this section. We describe the hardware platform, software running on the smartphone, and services that run on the server. We defer the algorithmic details to subsequent sections.

9.3.1 The Septimu Platform

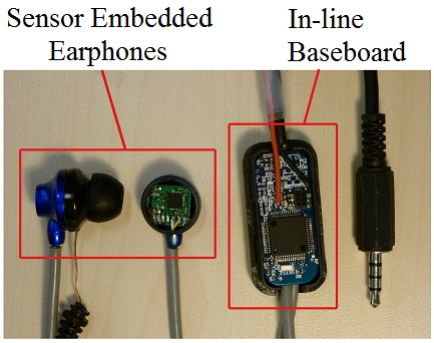


Figure 9.1: The Septimu hardware platform.

We introduce *Septimu*, which is a redesigned earphone accessory unit for smartphones that enables continuous in-situ wellness monitoring without interrupting the regular usage of the earphone. It is a redesign of the conventional earphone that has additional sensors and communicates with the smartphone via the audio jack interface. Septimu consists of a hardware and a software system. The hardware is comprised of two sensor boards, which are embedded into two earbuds, and a baseboard that collects data from the earbuds and communicates with the smartphone. The current sensor board incorporates an IMU (3-axis accelerometer and gyroscope), an analog microphone and a LED. The baseboard contains a microprocessor (TI MSP430F1611) and peripheral circuits to communicate with the audio jack on the mobile device using HiJack [106]. However, we could not achieve simultaneous communication and power harvesting from the audio jack as described in [106]. We believe this is the limitation of the phones that we tried which are different from the ones (i.e. iPhone) used in their work. Current version of Septimu is powered by a thin film battery. Figure 9.1 shows the two earbuds and the baseboard. The earbuds, with all the added sensors, has a dimension of 1×1 cm^2.

The software running on the Septimu baseboard is based on TinyOS. The microprocessor samples the IMU through I2C bus 25 times per second on each sensor board, with a full scale of $\pm 2g/s$ for accelerometer and ± 250 degree/sec for gyro. Data in the microprocessor is transmitted to mobile phone via microphone tip on the audio jack with Manchester coding, i.e., a logic 0 is represented by a High-Low sequence and a logic 1 is represented by a Low-High sequence. Each byte of samples is sent from the LSB, together with a start bit, a stop bit, and a parity bit. The output is generated

at a general IO port on microprocessor and the transmission rate is controllable by software. Septimu communicates with the phone via a digitally controlled multiplexer, to deliver both digital data from the IMU and analog audio data from the microphone in a time-multiplexed manner. The IMU sensor data is reliably transmitted at up to 500 bytes per second while microphone is sampled at 44.1 kHz.

9.3.2 Processing on Smartphone

The *MusicalHeart* application runs on the smartphone. It uses data from the Septimu sensors to measure heart rate and to detect activity levels. It also uses other on-device sensors to obtain contextual information. Based on all of this information, it then suggests a list of music to the user. Figure 9.2 shows a schematic of the software architecture of the application. A complete walk through of the data flow and data processing follows.

1. The user starts the Musical-Heart application on the smartphone which is connected to Septimu. The user may specify a goal, e.g., the target heart rate zone. By default, the system uses the user's current activity level as a basis to suggest music.

2. The application starts a *SensorDataDispatcher* service which generates 4 sensor data streams. The first two streams correspond to the IMU and the microphone units of Septimu. The other two streams correspond to the in-phone GPS and the WiFi scan results. The GPS and WiFi being power hungry are sampled once per minute and more frequently only when the user is detected to be moving. The streaming rate of the Septimu IMU is 50 Hz, while the maximum sampling rate of the microphone is 44.1 KHz.

3. The sensor streams are consumed by 3 processing units. The *HeartRateDetector* component processes the Septimu's microphone data to obtain the heart rate using the algorithm described in Section 9.4. The *ActivityLevelDetector* component processes the Septimu's IMU data to obtain the activity level of the user. The GPS and WiFi data are processed by the $ContextDetector$ component to obtain contextual information, such as the location (indoor vs. outdoor) and speed of the user. The details of the activity level and context detection algorithms are described in Section 9.5.

4. The *Combiner* component combines the processed data obtained from the three processing units into a time series of 3-tuples (Heart rate, Activity Level, Context). It maintains an in-memory circular queue to hold the tuples that are generated during the last 10 minutes. Thus the Combiner possesses the high level information of the user, i.e., what activity level the person is in, what the heart rate is, and any contextual information such as whether is at home, or traveling.

5. The *MusicRecommender* is responsible for suggesting music to the user and playing the song. It has a music player which is an extension of an ordinary music player, but with the added capability of loading and playing songs based on biofeedback. It kicks in once prior to the end of a song and consults the Combiner to get user's information

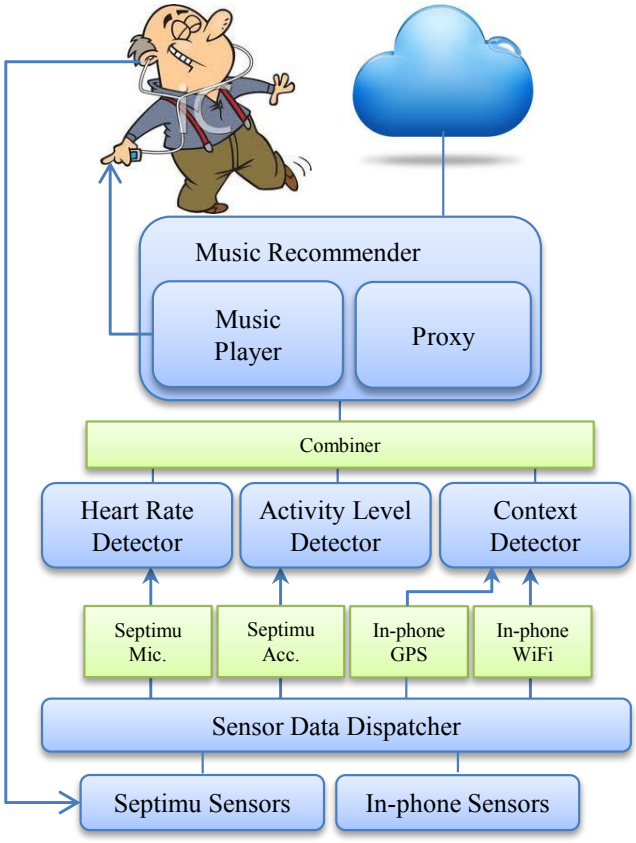


Figure 9.2: System diagram of Musical-Heart.

during the last song. It computes a report consisting of the heart rate information (start, end, minimum, maximum, average, fall time, and rise time), activity level (low, medium and high), contextual information (location, and velocity), and the user specified goal. All this information is sent to the server via a proxy. The proxy is responsible for maintaining communication with the server and obtaining the suggestion for the next song to be played.

9.3.3 Web Services

There are several advantages for storing and serving the *MusicSuggestionService* on the Web. If the information gathered by the Combiner exists in a shared location, the system can leverage the learned parameters from how people will generally react to a particular song offering better suggestions in the future.

To implement this, we pushed the MusicSuggestionService onto a RESTful Web Service that runs on a web server as shown in Figure 9.3. Our particular implementation uses a WEBrick server to expose a HTTP interface to the smartphone. During each song, the Combiner reports the user authentication, song ID, current heart rate and activity level information to the MusicSuggestionService in a POST request. Next, important features related to the song are fetched from a Music Information Retrieval database. There are several tools, e.g., openSMILE, MIRToolbox,

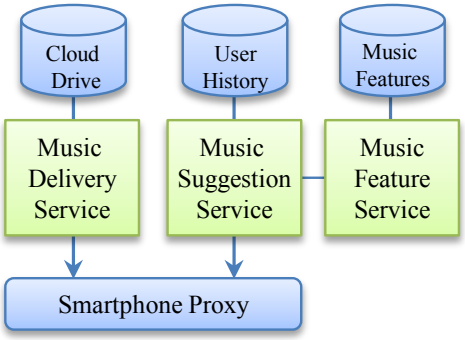


Figure 9.3: Web services in Auditeur.

jAudio that can extract musical features to create a custom feature database. However, we use the EchoNest web service that already has the data for 30 million songs. When an artist and title of a song is given, a report of the song's tempo, pitch, energy, loudness, and musical mode is returned as a JSON object. This information is used by the MusicSuggestionService to make a prediction for the appropriate song to play when a particular target heart rate is requested. The details of the algorithm is described in Section 9.6. When a song is selected, it is delivered to the smartphone either from the user's cloud drive, or is played from a local MP3 file on the smartphone.

9.4 Heart Rate Measurement

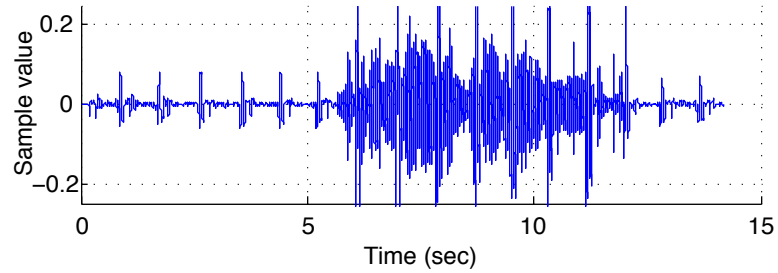
A tiny microphone is embedded inside Septimu which utilizes an in-ear design. This forms a resonant chamber inside the ear, amplifying the sound of heart beats. We have chosen acoustic sensing because of its potential use as a clinical stethoscope. Other sensing methods, such as IR-sensors, do not offer such opportunities and thus limit our possibilities. However, we show a comparison of the acoustic and IR-based heart rate detection techniques in Section 9.7.4.

Heart beat detection algorithms [107, 108] that are used to detect R waves in an ECG do not work for our problem due to the varying nature of the received signals and the presence of noise. ECG waves are stable and uniform in nature, and ECG is performed in a very controlled environment such as a hospital. In our case, the received signals differ in shapes and sizes as different people have different sizes of ear canals. The received signal also depends on how tightly the earbuds fit in someone's ear. Furthermore, our system is designed to support detection of heart beats even when the person is engaged in high levels of activities. These call for a new algorithm that is accurate and robust to detect heart beats from acoustic signals from the ear.

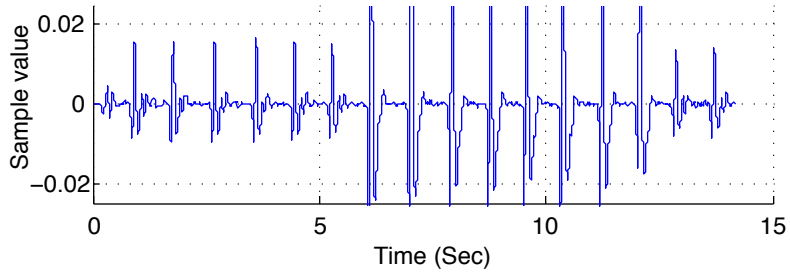
The raw audio samples that we collect from the microphone is a mixture of the music, the heart beats, human voices and other background noise. Measuring the heart rate from the audio samples is a two-step process – *(1) Filtering*: separating the heart beats from other signals, and *(2) Detection*: identifying the R waves and measuring the heart rate.

9.4.1 Filtering

Our heart beats in a specific rhythm. The resting heart rate of an adult person lies somewhere in between $40 - 100$ BPM, and the rate may reach up to 220 BPM during exercise. To extract the signals corresponding to the heart-beats, we eliminate any signal whose frequency is higher than the cutoff frequency, f_c = 3.67 Hz, corresponding to the maximum 220 BPM. Since the microphone sensor has a sampling frequency, f_s = 44.1 KHz, the normalized cutoff frequency of the low-pass filter is calculated by: $W_n = 2 \times \frac{f_c}{f_s} = 1.66 \times 10^{-4}$. In our implementation, we use a second order Butterworth filter. The filter coefficients, $a = [1.0000, -1.9993, 0.9993]$ and $b = 10^{-7} \times [0.0680, 0.1359, 0.0680]$, are obtained from the standard chart for Butterworth filters and are plugged into the standard difference equation to filter out the unwanted signals.



(a) Before filtering



(b) After filtering

Figure 9.4: Heart beats are extracted from the mixture of heart beats, music and noise by low-pass filtering.

Figure 9.4 illustrates the effect of applying the filter on a mixture of music and heart beat signals. The duration of the experiment is 15 seconds and the presence of music is from $6 - 12$ seconds. We see that the heart beat signals are clearly visible after the filtering. Due to some higher order harmonics of the music, the heart beat attains a constant gain. But this does not affect heart rate since the gain is only in the amplitude of the signal which does not affect the rate.

9.4.2 Detection

The heart beat detection algorithm takes an array of signal amplitudes with timestamps as inputs and returns the positions of the detected beats. The algorithm is applied on a fixed sized time window of 10 seconds. The smartphone accumulates 10 seconds of signals and the following steps are performed to detect the beats:

1. Selection: A set of candidate R waves is selected by taking the peaks that are at least one standard deviation larger than the mean. R wave being the largest, this step practically never misses any of them.

2. Matching: Each of the candidate R waves is matched to its nearest S wave. Since an R wave is a local maxima, the next local minima is the corresponding S wave. We denote the set of RS pairs with χ.

3. Quality Assignment: Each RS pair, $x_i \in \chi$ is assigned a real number, $q(x_i) \in [0, 1]$, which is its normalized peak-to-peak distance. The closer the value is to unity, the more likely it is to be an actual heart beat.

4. Dynamic Programming: Given the timestamps, $\{t(x_i)\}$ and the quality values, $\{q(x_i)\}$ of all the candidate RS pairs, our aim is to find the subset, $\{x_i\} \subset \chi$, which contains all (and only) the actual heart beats.

There are $2^{|\chi|}$ possible subsets that could be a possible solution. In order to find the optimum one, at first, we quantify the candidate solutions using two metrics based on the following two properties of the optimum solution:

Property 1: The variance of the time differences, $\Delta t(x_i, x_{i-1})$ between two consecutive R waves, x_i and x_{i-1}, in the optimum solution, is minimum.

Property 2: For two solutions with the same variance, the optimum one has the larger sum of quality.

The first property comes from the fact that the heart beats are periodic and there is hardly any change in the heart rate within the 10 seconds time window. Our empirical study in Section 9.7.3 shows that it takes about $25 - 50$ seconds to see any noticeable change in heart rates. The second property ensures that we take the beat sequence with a smaller period, which has the larger sum of qualities. Hence, we define the following metrics for $X \subset \chi$:

$$Q(X) = \sum_{x \in X} q(x), \text{ and } V(X) = \sum_{x_i, x_{i-1} \in X} \Delta t(x_i, x_{i-1})^2 \tag{9.1}$$

A brute force algorithm to find the optimum solution is to compute $V(X)$ for all $2^{|\chi|}$ subsets, and take the one that has the minimum $V(X)$ while breaking any tie by choosing the one with the largest $Q(X)$. But this is impractical, since even for a 10 second window, the size of the search space may exceed 2^{36} in the worst case. Instead, we simplify the search by dividing the search space. Each of the subspaces corresponds to a particular value of heart rate within the range of $40 - 220$ BPM. The search is performed using the following dynamic programming approach.

Let us assume, we know the heart rate in advance. This gives us $|X|$, i.e., the number of actual heart beats in χ, and also sets the minimum time interval between two consecutive beats. We define $f(i, k)$ as the cost of selecting k beats out of the first $i \leq |\chi|$ candidates. We now apply the following recurrence equations to find the optimum value of $f(|\chi|, |X|)$, which is the $Q(X)$ for a given heart rate:

$$f(i,k) = \begin{cases} \max\limits_{1 \le j \le i} \{q(x_j)\}, & \text{if } k = 1, i \ge 1 \\ \max\{f(i-h, k-1) + q(x_i), & \\ f(i-1,k)\}, & \text{if } 1 \le k \le i, i \ge 1 \\ -\infty, & \text{otherwise} \end{cases} \tag{9.2}$$

Here, h ensures the minimum time interval constraint; it denotes the number of previous candidate beats to skip in order to maintain the given heart rate. Once we have the optimum solution we compute $V(X)$ using the selected beats. We solve the above recurrence once for every heart rate values in the range $40 - 220$ BPM, and the one that gives us the minimum $V(X)$ is the heart rate of the person. Ties are broken with the largest $Q(X)$. Note that, if we only maximize $Q(X)$ without considering $V(X)$, the algorithm will greedily choose all candidate R waves; the job of $V(X)$ is to discard the false peaks that are out of rhythm.

Time Complexity: Let, s, c, and b be the number of samples, candidate beats, and actual beats, respectively. The complexity of selection and matching steps is $O(s)$, quality assignment is $O(c)$, and the dynamic programming is $O(cb)$. Since $c \ge b$, the overall theoretical worst case time complexity is $O(s + c^2)$. However, for a 10 seconds time window, we have $6 \le b \le 37$, which is essentially a constant. A candidate beat that is not an actual R wave is either a high peaked T wave that immediately follows an R wave, or the shadow of the R wave (called R' in medical literature). The wave form being periodic, assuming c a constant multiple of b, the overall time complexity is practically $O(s)$.

9.5 Activity and Context Detection

A pair of IMUs are embedded inside Septimu, which allows us to infer the activity level of the user. Combined with existing sensors on the phone, we are able to detect the context of the user which enables more suitable music recommendations.

9.5.1 Activity Level Detection

The activity level is an important indicator of what kind of music a person may prefer. For example, when a person is jogging, which is a high activity level, he may prefer music with faster rhythms. On the contrary, when a person is sitting and is at rest, he may prefer slower music. A person's activity level is considered to suggest appropriate songs. We use acceleration collected from the earbuds to detect the wearer's activity levels, which are divided into three categories: (1) L_1: low level, such as lying and sitting idle; (2) L_2 : medium level, such as walking indoors and outdoors; and (3) L_3: high level, such as jogging. We use standard machine learning techniques to distinguish the activity levels.

Feature Extraction

Given the raw 3-axis accelerometer data from the Septimu sensors, we extract the feature values as follows:

Step 1: For each 3-axis accelerometer data sample, (a_x, a_y, a_z) from the ear buds, we calculate the linear magnitude of acceleration: $|a_{ear}| = \sqrt{a_x^2 + a_y^2 + a_z^2}$.

Step 2: The standard deviation of $|a_{ear}|$ within each 1 second time window (i.e. 50 samples) is calculated – which is our feature value, x_i.

Training and Classification

Instead of using a generic threshold, we train an unsupervised learner so that our activity level detection is person specific. We use k-means clustering algorithm to cluster $\{x_i\}$'s to find out the 3 cluster means corresponding to 3 activity levels. Given the 3-axis accelerometer data from the Septimu sensors for T seconds, the steps classifying the activity level are described as follows:

Step 1: We obtain a set of T values, $\{x_i\}$, one value for each second, following the same procedure as the feature extraction stage.

Step 2: Each of these x_i's is then classified considering their minimum distance from the cluster heads obtained during the training.

Step 3: We take a majority voting to determine the most likely activity level of the person over the T seconds duration.

In our implementation, we choose the value of T to be 60 seconds, because a more fine grained activity level detection is not necessary in our application. We are concerned about the user's activity level during the duration of a song. If a user, for example, stops for moment to tie his shoelaces during jogging, we classify the entire duration as L_3 activity.

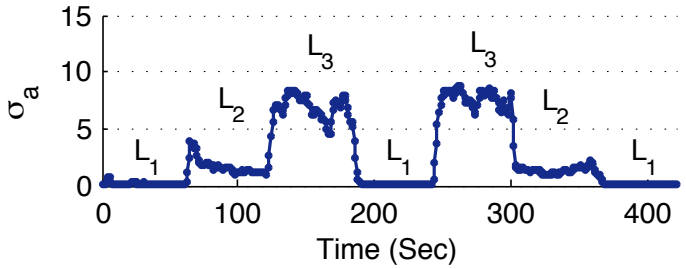


Figure 9.5: The standard deviation of accelerometer readings are used to distinguish among the activity levels.

As an illustration of the algorithm, in Figure 9.5, we plot the standard deviations of accelerometer readings at each second during a 7 minute long experiment performed by one of our participants. The same trend is observed when

we tested our algorithm on 17 participants; see Section 9.7.5. The user performs this sequence of activities: sitting, walking, jogging, sitting, jogging, walking, and sitting. We see a clear distinction among the 3 activity levels.

9.5.2 Augmenting Activity Levels with Contexts

Adding contextual information along with the activity levels makes the system more intelligent in choosing music. For example, sitting at home and sitting inside a bus are both level 1 activities. But if we can differentiate between these, we can suggest music based on the context – e.g. what music is listened to by other users during traveling, vs. when they relax at home. Since there are infinite possibilities, we conduct a survey on 208 people, asking one simple question – *when do you listen to music the most?* The users were allowed to mention at most 5 contexts. The summary of responses results in a list of the 7 most common contexts and is shown in Table 9.1.

Context	Description	Poll Result
LIE	Lying on bed	14%
SEATED	Sitting idle, or taking a break at desk	41%
TRAVEL	Traveling by bus, or car	24%
SLOWMOVE	Waiting at bus stop, walking on campus	7%
BIKE	Biking	2%
GYM	Exercising at gym	6%
JOG	Jogging or running	4%

Table 9.1: Activity contexts and corresponding poll result.

We augment activity levels with 3 types of information that helps to distinguish among the 7 activities: indoors/outdoors, velocity, and head angle (Septimu accelerometer's pitch and roll angles). Table 9.2 presents the context detection technique in a tabular form.

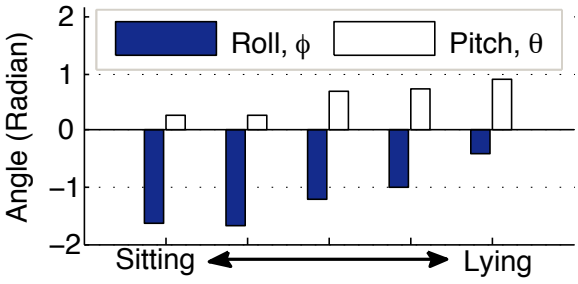


Figure 9.6: The roll and pitch angles obtained from Septimu are used to differentiate between sitting and lying.

To implement this, we detect whether a person is indoors or outdoors, and measure his velocity, roll and pitch angles. We rely on Android's location service to periodically obtain the location updates from its location providers once per minute. The pitch and roll angles of the accelerometer data is used to distinguish between lying and seated contexts. Figure 9.6 shows an example scenario where one of our participants lies down straight from a seated position. The

Context	Activity Lev	In-Out	Velocity	Roll, Pitch
LIE	L1	Indoor	-	$> -1.5, > 0.5$
SEATED	L1	In/Out	-	$< -1.5, < 0.5$
TRAVEL	L1	Out	$> 25\ mph$	-
SLOWMOVE	L2	In/Out	$< 3\ mph$	-
BIKE	L2	Out	$< 15\ mph$	-
GYM	L3	In	-	-
JOG	L3	Out	-	-

Table 9.2: The context detection algorithm is shown in a tabular form. Dashed entries are don't cares.

earbuds being stably situated, we observe the same trend in all 17 participants. Based on the empirical measurements, we use static thresholds of -1.5 for roll and 0.5 for pitch angles. The thresholds are fairly conservative and unlikely to make errors unless the earbuds are deliberately worn in an unusual way (e.g. upside down).

9.6 Music Suggestion and Rating

We describe a feedback control theoretic approach for suggesting music to the user. Existing approaches [79, 77, 78] apply simple rules of thumb to suggest music, i.e., *suggest the next song with a higher (or lower) tempo if the heart rate is falling behind (or rising above) the desired rate*. Although none of these works are built upon any sound mathematical basis, typically they mimic proportional controllers (P-controller) when viewed in the light of control theory. These controllers therefore suffer the same inherent problems of any P-controller. They are too responsive to the control input (harmful for human as the heart has to respond very fast to the changed music), have non-zero steady state error (harmful because the heart rate may settle at a different rate in the steady state), and have no memory of the past (does not consider the effect of heart's response to the previous song).

9.6.1 PI-controller Design

We propose a Proportional-Integral Controller (PI Controller) in order to achieve the desired heart rate in human by correctly selecting the music. A PI-controller is a nice fit to our problem since it has a slower response time than a P-controller, a minimum steady state error, and takes the past errors into account. These are desirable properties of our music suggestion algorithm as they provide a slower change in heart rate which is comfortable for the user, attain the desired heart rate with minimum error, and consider the errors that are made in the previous stages. We now describe the design of the controller in detail.

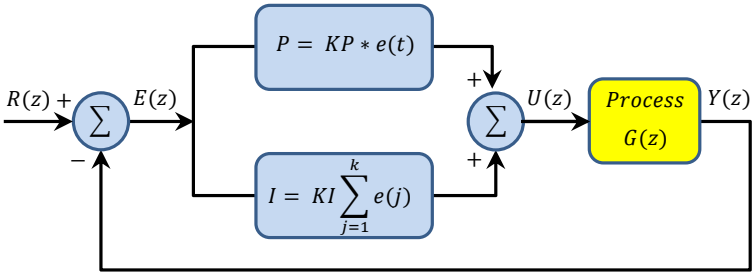


Figure 9.7: The PI controller uses the desired and current heart rate to suggest appropriate music.

Model

Figure 9.7 shows a schematic diagram of the PI-controller. The process having the transfer function $G(z)$, represents the part of the human heart that is sensitive to music. The input to the process, $U(z)$, represents the suggested change in the feature of music, the output, $Y(z)$, represents the current heart rate, $R(z)$ represents the desired heart rate, and $E(z)$ represents the difference between the two rates. K_P and K_I are the coefficients of the P- and I-controllers that we are going to compute. The transfer function of the PI-controller is:

$$\frac{U(z)}{E(z)} = \frac{(K_P + K_I)z - K_P}{z - 1} \tag{9.3}$$

And the transfer function [109] of the feedback loop is:

$$F_R(z) = \frac{[(K_P + K_I)z - K_P]G(z)}{(z - 1) + [(K_P + K_I)z - K_P]G(z)} \tag{9.4}$$

System Identification

System identification is the empirical procedure to model the empirical transfer function of the system being controlled. In our system, we model it by empirically estimating the change in heart rate when there is a change in control input which is a function of the features of the music. We assume a first order system with the transfer function:

$$G(z) = \frac{A}{z - B} \tag{9.5}$$

Taking the inverse z transform of Eq. 9.5, we get the system response in time domain:

$$y(k + 1) = Au(k) + By(k) \tag{9.6}$$

We define u as a linear combination of 3 features of music: tempo (f_1), pitch (f_2), and energy (f_3), and express it by: $u = \sum \alpha_i f_i$. It can be thought of as a *composite feature* of a song that has linear effect on changing the heart rate. The coefficients, α_i are estimated by applying linear regression on the empirical heart rate responses from *all*

users. While the definition of u is generic, the values of A and B parameters are *person specific*. For each user, these parameters are recomputed by the server after each use of the application. When a user finishes listening to a song, the server computes $u(k)$ of the song, and adds $u(k)$, $y(k)$, and $y(k+1)$ into the person's history. This history is then used to estimate A and B using *least squares* regression.

Design Goals

There are four design goals that need to be specified in designing any controller: stability, accuracy, steady state error and overshoot. These are summarized in Table 9.3. The stability of the controller is achieved when the poles of Eq. 9.4 are within unit circle [109]. We allow a steady state error of 5 BPM. The settling time of 1 unit means the duration of a single song, and we allow a maximum overshoot of 10%.

Goal	Requirement
Stability	Poles of F_R are inside unit circle.
Accuracy	Steady-state-error < 5 BPM
Settling Time	$k_s \leq 1$
Overshoot	$M_P < 0.1$

Table 9.3: Design requirements of the PI-controller.

Characteristic Polynomials

Using the desired properties of the system from Table 9.3, we compute the desired poles, $e^{\pm j\theta}$, of the system as:

$$r = e^{-4/k_s} = 0.018 \tag{9.7}$$

$$\theta = \frac{\pi \ln(r)}{\ln(M_P)} = 5.458 \tag{9.8}$$

The *desired* characteristic polynomial having these two poles is:

$$(z - re^{j\theta})(z - re^{-j\theta}) = z^2 - 0.025z + 0.00034 \tag{9.9}$$

The *modeled* characteristic polynomial as a function of K_p and K_I is obtained by substituting $G(z)$ in Eq. 9.4 and taking the denominator of the system transfer function, $F_R(z)$:

$$z^2 + [A(K_P + K_I) - (1 + B)]z + (B - AK_P) \tag{9.10}$$

Equating the desired polynomial to the modeled polynomial, we solve for K_P and K_I to obtain:

$$K_P = \frac{B - 0.00034}{A}, \text{ and } K_I = \frac{0.9755}{A} \tag{9.11}$$

9.6.2 Automated Music Rating

The server only has knowledge of the user responses to a song if that song has ever been listened to by the user. For each such song, a personalized rating is automatically computed by the system. A song gets different ratings at different activity levels and desired goals based on the steady state BPM error that it makes. We rate each song in a linear scale of $0 - 5$, where 5 corresponds to BPM error < 5, and 0 corresponds to BPM error of 20 or more.

For the songs that the user has not yet listened to, but some other user of Auditeur has, we estimate their expected rating by considering the similarities among songs and other user's ratings. Given a set of rated songs $\{s_i\}$, and their corresponding ratings, $\{r_i\}$, the expected rating of an unknown song, s_u at the same activity level with the same goal is computed by:

$$\hat{r}(s_u) = \alpha \sum r(s_i)p(s_i, s_u) + (1 - \alpha)\bar{r}(s_u) \tag{9.12}$$

Where, $p(s_i, s_u)$ is the Bhattacharyya coefficient [110] which computes the similarity score between two songs, $\bar{r}$ represents the average rating of the song by other users, and α controls how much we want to rely upon other users' feedback. Each song is represented by a 3 element feature vector corresponding to the tempo, pitch, and rms-energy of the music. The value of α is a function of amount of rated song by a user. Initially, α is set to 0.1, and is linearly incremented up to 0.7 when the number of rated songs exceeds 100. Hence, in the long run up to 30% of the suggested songs come from other user's responses.

9.7 Technology and Algorithm Evaluation

We describe three sets of experiments in this section. First, we evaluate the performance of Septimu hardware and firmware. Second, we describe and summarize the findings of our empirical study which is used as the empirical dataset for other experiments. Third, we describe a set of experiments where we use the empirical data to evaluate the performance of the heart rate measurement, activity level detection, and context detection algorithms.

9.7.1 Experimental Setup

The experiments are performed using multiple Android Nexus S smartphones that run Android OS (Gingerbread 2.3.6). The devices use WiFi 802.11 b/g connectivity and are equipped with a 1GHz Cortex A8 processor, 512 MB RAM, 1

GB Internal storage, and 13.31 GB USB storage. Note that, Septimu is generic hardware platform that does not require any specific operating system such as Android. Septimu even works with desktop computers without any change. It is just that our particular implementation of Musical-Heart is done on Android for a demonstration. The web server run on a computer running Ubuntu Linux 11.10 with 4Gb of RAM and a AMD Phenom X3 processor.

9.7.2 Evaluation of Septimu Platform

We measure the maximum data rate of Septimu by sending a known bit pattern $(0xA5A5)$ from the MSP430 to the smartphone at varying rates and then calculating the bit error rate (BER) at the receiving side. We keep transmitting data for 120 seconds at each rate. We observe that Septimu has almost zero (10^{-5}) bit error up to 5.56 kbps, but the BER keeps rising at higher rates. The limitation comes from the limited processing ability of MSP430. However, in our system, we use a data rate of 3 kbps which is sufficient for our application and ensures minimum error.

We measure the power consumption of Septimu hardware and the android mobile device with a power meter [90]. The Septimu consumes 42 mW power in a battery powered case. For the mobile device, the power measurements are summarized in Table 9.4. The estimated battery-life of a smartphone (having a 1500 mAh battery) with Septimu connected is about 22 hours.

State	LCD	Audio	Septimu	Power
Standby	off	off	disconnected	$<$ 15 mW
Home Screen	on	off	disconnected	420 mW
app	off	on	disconnected	200 mW
app	off	on	connected	250 mW

Table 9.4: Power draw breakdown of the application running on an Android phone (Samsung Nexus S).

We obtain the memory usage of Septimu firmware from the compiling information in TinyOS. Out of 10 KB ROM and 48 KB RAM of the MSP430, Septimu uses only 5.8 KB of ROM, and 288 bytes of RAM.

9.7.3 Empirical Study

We perform an empirical study involving 37 participants, in which, we measure their heart rates during three different levels of activities, and both with and without the presence of music. The activity levels corresponds to {lying, sitting idle}, {walking indoors and outdoors}, and {jogging}. The number of participants in each activity level are 17, 10, and 10, respectively. The participants listen to different types and numbers of music items during multiple sessions. Each of these sessions is $10 - 60$ minutes long. The durations of the songs are about $2 - 5$ minutes and there are about $1 - 5$ minutes gaps in between two songs. The collection of songs comprises of the most watched 100 songs in year 2011 on YouTube. The songs vary in genre, language, length, and volume level. The group of participants is comprised of

undergraduate and graduate students, researchers, professionals, and their family members. Their ages are in the range of $20 - 60$, and they have diversities in sex, physical structure, and ethnicity. The properties of the music are obtained using MIRtoolbox [111].

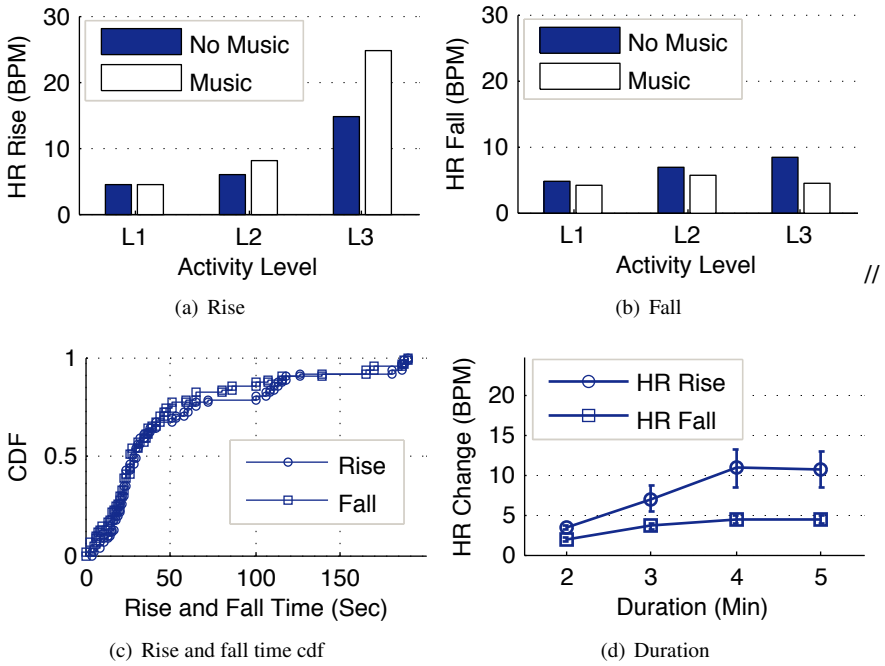
In order to obtain the ground truth of the heart rate, we use one 3-lead ECG device [112] and one pulse oximeter [113]. Both of the devices measure and store heart rate data at a rate of 1 sample per second. While an ECG is the most reliable method to measure the heart rate, this is not always convenient, as we have to attach three electrodes on to the bare chest of the subject. Specially, for outdoor experiments, we require another means to obtain the ground truth. We use a fingertip pulse oximeter for this purpose, which is worn at the index finger and the instantaneous heart rate is measured and stored in its internal storage.

Heart Rate: Rise and Fall

It is a well-known fact that music has a profound effect on our heart [70, 71]. We are not therefore trying to reprove their relationship; rather, our objective is to study the effect of music on smartphone users who are possibly mobile and therefore at different levels of activities. We are interested to know, e.g., how much is the effect of music in rising or dropping one's heart rate, how long it takes to make the effect and whether these have any relation to the activity levels. In Figure 9.8, we answer these questions from our empirical observations. Figure 9.8(a) and 9.8(b) show the amount of rise and fall in heart rates at different levels of activities – both in presence and in absence of music. We obtain these plots by first dividing the songs into two classes – one that tends to raise the heart rate and the other that slows down the heart. This distinction is made by computing the duration of the largest rise or the largest fall of heart rates. We do this after taking the 10 seconds moving average to rule out any transient behavior in the instantaneous readings. We observe that, the rise in heart rate is more in the presence of music and also the fall of heart rate is less when music is playing. The higher the activity level, the more effective it is in raising the rate or resisting the rate to drop. However, this rise and fall in heart rate is not instantaneous. Figure 9.8(c) shows the cumulative distribution function (cdf) of the rise and fall time of heart rates. A rapid growth in cdf happens when the time is in between $25 - 50$ seconds. On average, the time for heart rate to rise or fall by at least 10% is about 50 seconds. Figure 9.8(d) shows the amount of the rise and fall. The longer a person listens to the music, the more is its effect. For example, listening to an exciting music for 5 minutes raises the heart rate by 10 BPM, while a soothing music for 5 minutes calms down the heart rate by 5 BPM on average.

Effect of Tempo, Pitch, and Energy

Tempo, pitch and energy are three important features of music that correlate to changes in heart rate [71, 67]. But their effect varies with activity levels. For example, tempo is more effective in raising the heart rate when a person is jogging or exercising in rhythm with the music. On the other hand, pitch and energy have greater effect than tempo during the



(a) Rise (b) Fall

(c) Rise and fall time cdf (d) Duration

Figure 9.8: Rise and fall of heart rates.

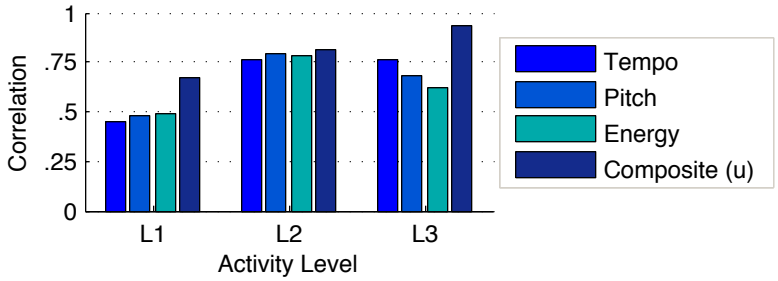


Figure 9.9: The composite feature is more correlated to heart rate change at all three activity levels.

low activity level. This is why we introduce the composite music feature u, which is a linear combination of these three and is activity level specific. Figure 9.9 shows the correlation of tempo, pitch, energy and the composite feature, u with heart rate. We observe that, u is more correlated than the other 3 features and is consistent over different activity levels.

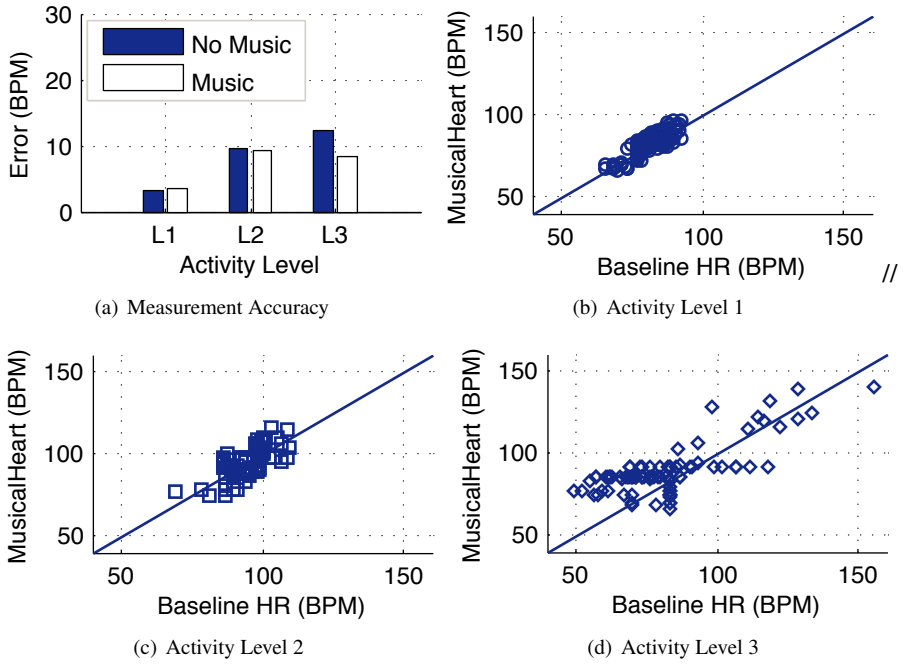
9.7.4 Evaluation of Heart Rate Measurement

We evaluate our heart rate measurement algorithm using two datasets. The first dataset contains the empirical data that we collected from 37 smartphone users. The other one is the MIT-BIH Arrhythmia dataset [114] which is the most used dataset for evaluating heart rate measurement algorithms, and contains data from 48 real patients. We also compare our acoustic-based technique with an implementation of IR-based technique.

Performance on Empirical Dataset

Figure 9.10 shows the performance of our heart rate measurement algorithm when applied to the empirical dataset. Figure 9.10(a) shows the BPM errors of Musical-Heart when compared to the baseline, i.e., pulse oximeter in most cases. At each activity level, the error is similar regardless of the presence of music, because, after filtering out any unwanted signals, the algorithm handles the two cases in the same way. But with higher levels of activities, the error tends to increase. We investigated these data traces manually and found that the reception of the audio signals were very poor in those cases. This happened due to the loose contact of the earphone after a long period of jogging and continued until the user fit it well again. Overall, we observe an average error of 7.5 BPM when compared to the pulse oximeter.

Since there might be some differences between pulses obtained from the ear and from the finger tip, it is desirable to see the correlation between the two besides comparing their absolute values. Furthermore, for our application and in many medical applications, the rate of change of heart rate is of more importance than the absolute value. Figure 9.10(b), Figure 9.10(c), and Figure 9.10(d) show the correlations at different activity levels. The correlation coefficients are 0.85, 0.84, and 0.75 respectively. At low and medium levels, the readings are very close to the ideal (i.e. the diagonal line). But we see some non correlated horizontal and vertical points during the high activity level. The horizontal points correspond to the looseness of the earphone, and the vertical points corresponds to the inability of the pulse oximeter to report the pulse during high movements, as the device is recommended to be used at rest for its best performance.



(a) Measurement Accuracy (b) Activity Level 1

(c) Activity Level 2 (d) Activity Level 3

Figure 9.10: Performance of heart rate measurement algorithm on empirical data.

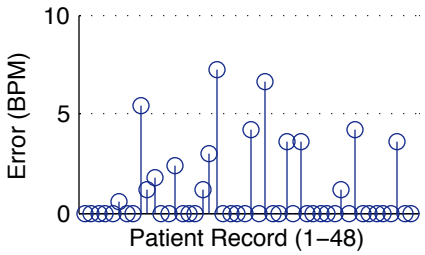


Figure 9.11: Performance of heart rate measurement on MIT-BIH Arrythmia dataset.

Performance on MIT-BIH Arrythmia Dataset

As we do not know the actual heart condition of our participants who possibly have sound health with good rhythmic heart beats, we wanted to test our algorithm on the data that are collected from actual patients. The MIT-BIH Arrythmia dataset is a standard dataset that has been used in Biomedical research community for decades. It contains raw ECG data from 48 actual patients who have known heart diseases – annotated with various information such as the heart rate. We use only the data that is obtained from ECG Lead II, since the waveform of this lead closely resembles the recording obtained from the ear. We slightly modify our algorithm to detect heart beats from this ECG data and Figure 9.11 shows the BPM errors for all 48 patient records. Our algorithm's average error is about 1.03 BPMs, and has zero errors for most records. In some datasets we observe error as high as 7 beats. But we manually checked that these records contain so wiggly waveforms that even human eyes cannot detect heart beats in it due to the noise.

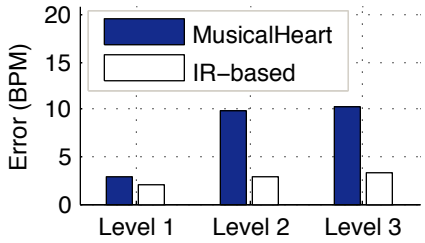


Figure 9.12: Comparison of heart rate measurement with an IR-based method.

Comparison with IR-based Technique

In this experiment, we compare the errors of our acoustic-based solution with that of an IR-based one. To do so, we design and implement a pair of Septimu-like earbuds that includes IR sensors. A microcontroller samples the IR-sensors at 50 Hz and transmits the data to a PC using the Bluetooth. We analyze the frequency spectrum of the received data and identify the peaks which correspond to the heart beats. We use a pulse-oximeter to measure the ground truth. Figure 9.12 compares the errors of the acoustic-based technique with that of the IR-based technique at three activity levels. For each level, we take 32 heart rate measurements and compute the mean error. We observe that, the IR-based solution beats the acoustic solution in all three levels. The mean error across all levels of activity for the acoustic sensor is 7.5 BPM and for the IR sensor, it is 2.69 BPM. Hence, this plot depicts the trade-off between the acoustic solution's

multiple usages and IR-based solution's accuracy. For apps where this difference is important we would include an IR sensor in the ear bud. We leave it as a future work to combine the two sensing methods to achieve the best of both.

9.7.5 Evaluation of Activity Level Detection

We evaluate the accuracy of activity level detection algorithm, in which, we use only the Septimu's accelerometer data. A total of 17 users participate in this experiment. We conduct two rounds of experiments with them. First, each of the users perform exactly 3 activities, corresponding to 3 activity levels, from the list in Table 9.1. Each of these activities lasts about $1-5$ minutes. For each person, we separately train one classifier with 60% of the collected data, and run cross validation on the remaining 40% to obtain the confusion matrix of Table 9.5. We see that, the average accuracy of the learner is almost perfect at all three levels, except for a few cases where it gets confused by some L_3 activities. This has happened since some users tend to slow down after jogging for a while, which is classified by the algorithm as walking. Overall, the accuracy is 99.1%.

		Predicted		
		L_1	L_2	L_3
Actual	L_1	0.9998	0.0002	0
	L_2	0	0.9997	0.0003
	L_3	0	0.0280	0.9720

Table 9.5: Single activity detection.

		Predicted		
		L_1	L_2	L_3
Actual	L_1	0.9890	0.0110	0
	L_2	0	0.9510	0.0490
	L_3	0	0.0370	0.9630

Table 9.6: Activity sequence detection.

In the next round of experiments, the users perform a series of activities in sequence: $\{L_1, L_2, L_3, L_1, L_3, L_2, L_1\}$. The sequence is chosen so that all six transitions between any two activity levels happen. Each of the activities in the sequence is 60 seconds long. After the experiment, all the collected data are classified using the same classifier from the first set of experiments. This is done since in Musical-Heart, we first train the system for each activity level, and then the trained classifier is used to recognize different activities. The confusion matrix of this second experiment is shown in Table 9.6. We see some confusion between $\{L_1, L_2\}$ and $\{L_2, L_3\}$ during the transitions. But this is transient and does not affect the music recommendation algorithm. Overall, the accuracy is about 96.8%.

9.7.6 Fitness of System Model

Modeling human physiological response with a first order system may seem overly simplistic, but our empirical study reveals that the goodness of fit of the model is pretty accurate when the number of songs listened to by a user is 40 or

more. Figure 9.13 plots the fitness of the model against the number of songs. In our empirical dataset, the number of songs any person has listened to is in between $20 - 55$. Using the history of responses from 37 people, we compute the model parameters A and B for each person. We measure the goodness of the model by computing $R^2 = 1 - \frac{var(y-\hat{y})}{var(y)}$, where y and $\hat{y}$ are the actual and predicted responses of the system. The closer the value is to 1, the better is the model.

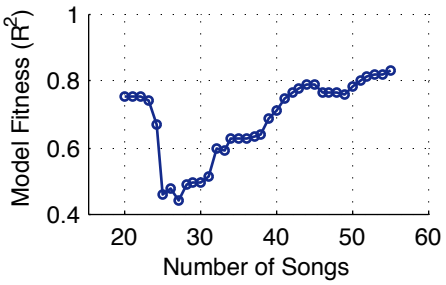


Figure 9.13: The fitness of model gets better as the user listens to more and more songs.

9.8 Real Deployment

We deploy Musical-Heart in two real world scenarios– similar to the use cases that we described earlier in the chapter. Four volunteers participate in each of the experiments. Two of them are male and two are female. All four participants are healthy and they belong to the age group of $21 - 30$. The participants are instructed to feel the music and to stay in rhythm of the music.

Time	Intensity	Pace
5 min	60%-70%	Walk at a comfortable pace to warm up.
3 min	70%-80%	Increase speed a few increments until working harder than the warm up pace. This is the baseline.
2 min	80%-90%	Increase speed again until working slightly harder.
3 min	70%-80%	Decrease speed back to baseline.
2 min	80%-90%	Increase speed once again until working slightly harder than baseline.
5 min	60%-70%	Decrease speed back to a comfortable level.

Table 9.7: The cardio exercise program that our participants take part in during the experiment.

9.8.1 Goal Directed Aerobics

The goal of this experiment is to demonstrate that Musical-Heart measures heart rate, detects activity levels, and suggests appropriate songs while a person is exercising. The 20 min long cardio program that our participants take part in, is described in Table 9.7. The program mentions the duration, target intensity, and the required change in pace. Prior to the experiment, we measure the resting heart rate and estimate the maximum heart rate of each participant – which are used to calculate the target heart rate using the equation in Section 3.9. The controller parameters at each of the

3 activity levels of a participant are estimated from the history of at least 40 previous responses of that person. This information is used by the controller to suggest appropriate music in run-time.

Figure 9.14 shows the variations in heart rate (normalized to intensity) during jogging for all 4 participants. The stairs represent the desired intensity, the curved lines represent the achieved intensity, and the arrows represent the activation of the control input. An upward (downward) arrow denotes a positive (negative) control signal corresponding to a suggested song that helps to rise (fall) the current heart rate. The value of control signal, u is used as the search-key in the music database to find a matching song.

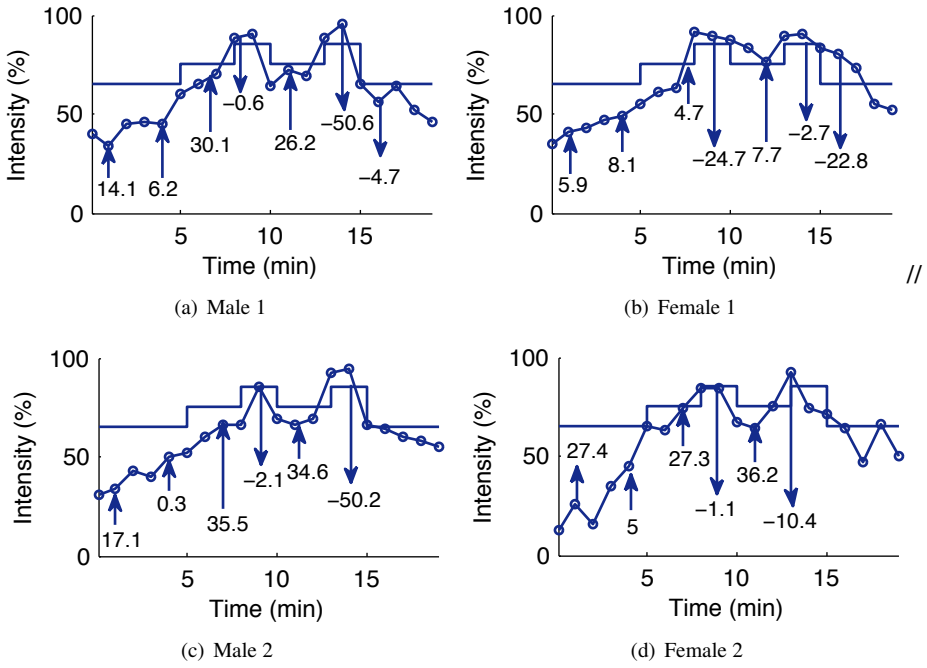
Figure 9.15 shows the variations in accelerometer readings corresponding to Figure 9.14. The curved lines represent the standard deviations of accelerometer readings. The detected activity level of a person is computed from these readings using the person specific thresholds as described in Section 9.5.1. The stairs represent the boundary between the detected activity levels L_2 and L_3. For example, according to the cardio program in Table 9.7, all 4 participant should be in L_2 for the first 5 min, then in L_3 for the next 10 min, and finally in L_2 during the last 5 min. The detected activity levels accurately match the desired levels for all 4 participants, except for the first male person (Male 1) who seems to have slowed down a little at some point during his $7-8$ min interval.

We illustrate the first control signal activation event of the first male participant as an example. The person is at the intensity level of 38%, while his desired level is 65%. From the knowledge of his resting and maximum heart rates, 70 and 170 BPM, these two intensity levels correspond to 108 and 135 BPM, respectively. Figure 9.15(a) shows that he is at activity level L_2 at that moment. Using the control parameters at activity level L_2 of this person, $\alpha_1 = -0.0196,\ \alpha_2 = -69.8587,\ \alpha_3 = 0.0213,\ A = 0.92\ and\ B = 1.13$, we obtain, $u = (135 - 1.13 \times 108)/0.92 = 14.1$. The database of music is then searched for an appropriate music that has the composite feature value of 14.1. Using this control approach, overall, the mean deviation in intensity levels for the 4 participants are: 11.4%, 13.2%, 12.1%, and 11.8%, respectively.

9.8.2 Music Recommendation via Biofeedback and Collaboration

The goal of this experiment is to demonstrate Musical-Heart's bio-feedback based rating and collaborative recommendation features. We perform a 3-day long experiment with 4 users, in which, each user uses Musical-Heart app while riding on a vehicle (e.g., bus or car) and looks for heart soothing songs. The server initially has the history of at least 40 responses from the first two users, while the other two users are new. We expect that the more a user uses Musical-Heart, the better is the quality of music recommendation.

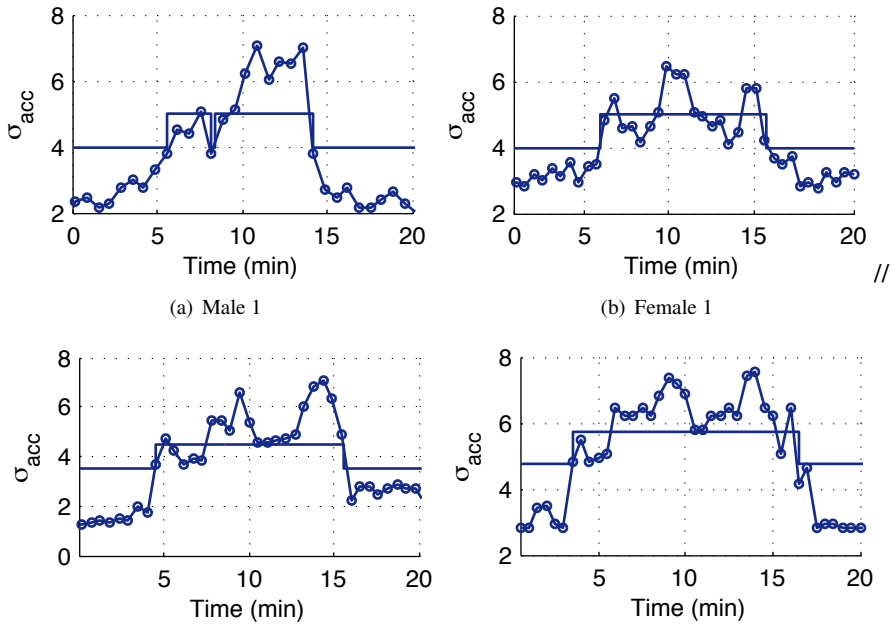
Figure 9.16(a) shows the automated ratings obtained from 4 users over a 3-day period. The first two users are regular users of Musical-Heart and the system therefore consistently suggests high rated songs to them with average ratings of 3.56 and 3.78. For the new users of the system, initially, the quality of suggestion is in the range of $2.32 - 2.79$, but the



(a) Male 1 (b) Female 1

(c) Male 2 (d) Female 2

Figure 9.14: The desired intensity, achieved intensity and activation of control signals are shown.

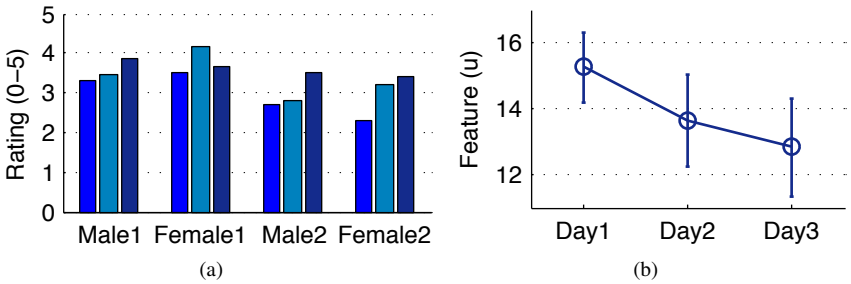


(a) Male 1 (b) Female 1

(c) Male 2 (d) Female 2

Figure 9.15: Standard deviation of accelerometer data and boundary between activity levels L_2 and L_3 are shown.

more they use the application, the better the quality of suggestion gets, i.e., $3.42 - 3.52$. The improvement happens due to the learning and collaborative nature of the recommendation algorithm. Figure 9.16(b) shows the value of the composite feature, u of the suggested songs on each day. The composite feature being highly correlated to raising the heart rate, the less its value, the better it is in calming down the heart. We see that the average value of u gets lower on



(a) (b)

Figure 9.16: (a) The automated ratings for the new users are increased. (b) The reduction of u indicates increased accuracy of soothing music suggestion.

each day, which indicates that Musical-Heart is indeed suggesting heart soothing song to its users. Note that, the values of u in this experiment are not comparable to those in the previous experiment since all 4 users are in activity level L_1 which is different.

9.9 Summary

In this chapter, we introduce Septimu, a wearable sensing platform consisting of a pair of sensor equipped earphones, and as an application of it, we design and implement, Musical-Heart which is a novel bio-feedback based, context aware, automated music recommendation system for smartphones. We provide empirical evidence that the system is capable of detecting heart rate from the ear which is $75\% - 85\%$ correlated to the ground truth and has an average error of 7.5 BPM. We deploy Musical-Heart in two real world scenarios and show that it helps the user in achieving a desired exercising intensity with an average error of less than 12.2%, and its quality of music recommendation improves over time. The version of Musical-Heart that we have described in this chapter does not use Auditeur as a means for detecting heartbeats. However, it is possible to implement MultiNets using Auditeur, which we have shown in one of the case studies in Chapter 6.4.

Chapter 10

Conclusion

10.1 Summary and Key Contributions

This thesis presents the design, implementation, and evaluation of a general-purpose, energy-efficient, and context-aware acoustic event detection platform for mobile devices. This is a useful platform and would save months of development time for developers who want to build applications that act upon acoustic events. As part of this thesis, we have developed a total of four systems. Each of which contributes to the primary objective of this dissertation which is to enable rapid development of mobile applications that recognize general-purpose acoustic events on mobile devices and are energy-efficient, highly accurate, adaptive to user's context, and easier to program.

10.1.1 A Mobile-Cloud Platform for Acoustic Event Detection

At the heart of the thesis, we have the Auditeur platform. Aside from its ability to recognize a wide variety of sounds, the platform is shown to automatically create acoustic event classifiers that are accurate, energy efficient, and adaptive to user's context. We provide empirical evidence that Auditeur's energy-aware algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of the maximum achievable accuracy. Seven applications have been implemented with the Auditeur API to demonstrate its versatility and to show that applications developed with Auditeur are $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events compared to state-of-the-art techniques. A user study on 15 undergrads shows that even novice programmers can implement the core logic of interesting applications with Auditeur in less than 30 minutes, using only $15 - 20$ lines of Java code.

10.1.2 Exploting Sparseness in Speech

Due to limited processing capability, contemporary mobile devices cannot extract frequency domain acoustic features in real-time on the device when the sampling rate is high. We propose a solution to this problem which exploits the sparseness in speech to extract frequency domain acoustic features inside a mobile device in real-time, without requiring any support from a remote server even when the sampling rate is as high as 44.1 KHz. We perform an empirical study to quantify the sparseness in speech recorded on a mobile device and use it to obtain a highly accurate and sparse approximation of a widely used feature of speech called the Mel-Frequency Cepstral Coefficients (MFCC) efficiently. We name the new feature the sparse MFCC or sMFCC, in short. We experimentally determine the trade-offs between the approximation error and the expected speedup of sMFCC. We implement a simple spoken word recognition application using both MFCC and sMFCC features, show that sMFCC is expected to be up to 5.84 times faster and its accuracy is within $1.1\% - 3.9\%$ of that of MFCC, and determine the conditions under which sMFCC runs in real-time.

10.1.3 Mobile-Cloud Communication Efficiency

Auditeur performs on-device acoustic event detection in real-time as opposed to sending audio data to the cloud over the Internet. However, there are applications that must communicate to a remote server in order to use the web services offered by it. To reduce the energy-cost due to wireless communication, we have developed MultiNets, which is capable of dynamically switching to the most energy-efficient wireless network interface (3G and WiFi) at runtime, and without requiring any modification to the application. We describe the architecture of MultiNets and demonstrate the methodology to perform wireless network interface switching in Linux based mobile OSes such as Android. Our analysis on mobile data traces collected from real users shows that with dynamic switching we can save 27.4% of the energy, offload 79.82% of the data traffic, and achieve 7 times more throughput on average. We deploy MultiNets in a real world scenario and our experimental results show that depending on the user requirements, it outperforms the state-of-the-art Android system either by saving up to 33.75% energy, or achieving near-optimal offloading, or achieving near-optimal throughput while substantially reducing TCP interruptions due to switching.

10.1.4 An Application Detecting a Special Type of Acoustic Event

We present Musical-Heart, which is a biofeedback-based, context-aware, automated music recommendation system for mobile devices. We introduce a new wearable sensing platform, Septimu, which consists of a pair of sensor-equipped earphones that communicate to the smartphone via the audio jack. The Septimu platform enables the Musical-Heart application to continuously monitor the heart rate and activity level of the user while listening to music. The physiological information and contextual information are then sent to a remote server, which provides dynamic music suggestions to help the user maintain a target heart rate. We provide empirical evidence that the measured heart

rate is $75\% - 85\%$ correlated to the ground truth with an error of < 2 BPM when the user is stationary, and 7.5 BPM on average. Such an error may not be desirable in applications that require a precise heart rate measurement. For those applications, we recommend an IR-based heart rate detector which is more accurate than an acoustic-based one. The accuracy of the person-specific, 3-class activity level detector is on average 96.8%, where these activity levels are separated based on their differing impacts on heart rate. We demonstrate the practicality of Musical-Heart by deploying it in two real world scenarios and show that Musical-Heart helps the user achieve a desired heart rate intensity with an average error of less than 12.2%, and its quality of recommendation improves over time.

10.2 Limitations and Future Improvements

There are some notable extensions and improvements to the research we have presented in this research.

First, Auditeur's set of acoustic processing units has a limited number of primitives. While we have shown that, with these primitives, Auditeur is capable of recognizing a wide range of acoustic events, yet, for a completely new acoustic event detection problem, the list might be inadequate. One aspect of Auditeur is that, it is extensible. Adding a new primitive does not require any change into the framework. In the future, we would like to explore different types of acoustic event detection problems that we have not covered in this thesis. We are particularly interested in studying physiological sounds and sounds related to home safety, as these have potential applications such as – personal wellness monitoring and mobile health. One such problem that we plan on pursuing in the near term is to analyze asthma sounds collected with an electronic stethoscope connected to a mobile device.

Second, the sMFCC acoustic feature is faster to extract, however its applicability is not fully explored in this research. We have made sMFCC an optional feature that the developer may choose to use instead of the regular MFCC features. In the future, we look forward to explore the applicability of sMFCC on sounds other than speech and make the choice between MFCC and sMFCC automatic in Auditeur.

Third, the MultiNets engine currently considers switching between WiFi and 3G interfaces on a mobile device only. However, 4G is prevailing these days and a newer version of WiFi (IEEE 802.11ac) has received approval earlier this year (Jan 2014). These newer networks supposedly have different characteristics in terms of energy cost and throughput. Mobile and wearable devices, such as tablets, smart watches, and smart glasses, have also started supporting multiple network interfaces. In the future, we plan on studying the characteristics of 4G and 802.11ac networks on different devices and platforms, and extend the capability of MultiNets by incorporating these newer technologies.

Fourth, the Musical-Heart system currently considers only three acoustic features of a song, i.e. the tempo, pitch, and energy. However, there are other music features such as the loudness, liveness, speechness, and danceabilty, which may also have correlation to the heart rate. The pilot study that we conducted also had a small number of users. In the

future, we plan to perform large scale studies involving more users and additional music features to make Musical-Heart even more accurate.

10.3 Future Research Directions

One of our future research directions is to extend the ideas in Auditeur and approach a more generic problem – which is to *automatically generate classifiers for general purpose sensor data*. This generalization is far more complex a problem than acoustic classification at least for two reasons: first, different sensor streams have different characteristics, and second, we do not also know in advance the exact number of streams that are available in an opportunistic sensing scenario. However, a solution to this problem would foster the development of plethora of mobile applications that require recognizing a wide range of user-contexts from multiple sensing modalities.

Another research idea that we plan to pursue is to approach the sensor data classification problem in a different way than it has been done in Auditeur. One limitation of Auditeur is that, it assumes a fixed set of features – which is neither exclusive nor exhaustive. It is highly likely that for a new type of acoustic classification problem, we may have to incorporate new feature extractor modules into the system. An alternative way to deal with this feature crisis problem is to have the system automatically learn the features prior to creating a classifier. This approach is similar to a recent advancement in machine learning field called the *deep learning* [115], which has successfully been applied to image recognition problems. We would like to investigate the idea of *deep learning for shallow devices*, i.e. the potential of deep learning algorithm in general-purpose sensor data classification problems in a constrained setup, where the device has a limited amount of energy.

Appendices

Appendix A

Optimum Energy Computation in MultiNets

Given, the data usage D_i of a user for every T seconds, energy for turning on (E^W_{ON}, E^C_{ON}), data transfer ($E^W_D(D_i)$, $E^C_D(D_i)$), and idle power (P^W, P^C) of the WiFi and cellular, and the availability of WiFi, our goal is to find the optimum energy consumption considering switching. The problem is an instance of classical multi-stage graph optimization problem where the time points are the stages and the interfaces are the choices (sometimes only one choice of 3G) at each stage. Let us define $f(i, C)$ as the optimum energy from the starting time (time = 1) till $i - th$ time such that we use 3G at time i. Similar is the definition of $f(i, W)$ for WiFi. We now formulate the following recurrence relations:

$$f(i,C) = min \begin{cases} f(i-1,C) + P^C \times T + E^C_D(D_i) \\ f(i-1,W) + E^C_{ON} + P^C \times T + E^C_D(D_i) \end{cases}$$

$$f(i,W) = min \begin{cases} f(i-1,W) + P^W \times T + E^W_D(D_i) \\ f(i-1,C) + E^W_{ON} + P^W \times T + E^W_D(D_i) \end{cases}$$

with initial conditions $f(0, C) = E^C_{ON}$, $f(0, W) = E^W_{ON}$. These recurrences are written assuming WiFi is available. To address the cases when WiFi is not available, we ignore $f(i - 1, W)$ in $f(i, C)$, and make $f(i, W) = f(i, C)$. Finally, the optimum energy is computed as the minimum of $f(n, W)$ and $f(n, C)$ where n is the total number of time points.

Appendix B

Media Coverage of Musical-Heart

- ”Teaching Old Microphones New Tricks”, The Economist, Jun 2013.
- ”Exercise Music App Listens to Your Heart to Rev it Up”, New Scientist, Oct 2012.
- ”DJ Meets Gym Coach with App that Keys Songs to Heart Rate”, CNET, Oct 2012.
- ”Music App that Gets Your Heart Pumping During Exercise”, The Times of India, Oct 2012.
- ”University Researchers Run to the Beat”, Cavalier Daily, Oct 2012.
- ”There is an Exercise App that Can Control Your Heart Rate with Music”, Gizmodo, Oct 2012.
- ”App Syncs Music With Heart Rate, Activity”, Runners World, Oct 2012.
- ”The Musical Heart Keeps Workouts on Pace”, Trend Hunter, Oct 2012.
- ”Musical Heart Helps You Maximize Your Workout”, Ubergizmo, Oct 2012.

Bibliography

[1] Siri. http://www.apple.com/ios/siri//.

[2] Avery Wang. An industrial-strength audio search algorithm. In *ISMIR '03*.

[3] Microsoft's Project Hawaii. http://research.microsoft.com/en-us/projects/hawaii/.

[4] Hong Lu, A. J. Bernheim Brush, Bodhi Priyantha, Amy K. Karlson, and Jie Liu. SpeakerSense: energy efficient unobtrusive speaker identification on mobile phones. In *Pervasive '11*, San Francisco, CA.

[5] Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. Sphinx-4: a flexible open source framework for speech recognition. Technical report, Mountain View, CA, 2004.

[6] Hong Lu, Mashfiqui Rabbi, Gokul T. Chittaranjan, Denise Frauendorfer, Marianne Schmid Mast, Andrew T. Campbell, Daniel Gatica-Perez, and Tanzeem Choudhury. StressSense: Detecting stress in unconstrained acoustic environments using smartphones. In *UbiComp '12*, Pittsburgh, PA.

[7] Kiran K. Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J. Rentfrow, Chris Longworth, and Andrius Aucinas. EmotionSense: a mobile phones based adaptive platform for experimental social psychology research. In *Ubicomp '10*, Copenhagen, Denmark.

[8] Emiliano Miluzzo, Cory T. Cornelius, Ashwin Ramaswamy, Tanzeem Choudhury, Zhigang Liu, and Andrew T. Campbell. Darwin Phones: the evolution of sensing and inference on mobile phones. In *MobiSys '10*, San Francisco, CA.

[9] Emiliano Miluzzo, Nicholas D. Lane, Kristof Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the CenceMe application. In *SenSys '08*, Raleigh, NC.

[10] Sound Hound. http://www.soundhound.com/.

[11] Eric C. Larson, TienJui Lee, Sean Liu, Margaret Rosenfeld, and Shwetak N. Patel. Accurate and privacy preserving cough sensing using a low-cost microphone. In *UbiComp '11*, Beijing, China.

[12] Shahriar Nirjon, Robert Dickerson, Qiang Li, Philip Asare, John Stankovic, Dezhi Hong, Ben Zhang, Guobin Shen, Xiaofan Jiang, and Feng Zhao. MusicalHeart: A hearty way of listening to music. In *SenSys '12*, Toronto, Canada.

[13] Martin Azizyan, Ionut Constandache, and Romit Roy Choudhury. SurroundSense: mobile phone localization via ambience fingerprinting. In *MobiCom '09*, Beijing, China.

[14] Yohan Chon, Nicholas D. Lane, Fan Li, Hojung Cha, and Feng Zhao. Automatically characterizing places with opportunistic crowdsensing using smartphones. In *UbiComp '12*, Pittsburgh, PA.

[15] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. SoundSense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09*, Poland.

[16] Hong Lu, Jun Yang, Zhigang Liu, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *SenSys '10*, Zurich, Switzerland.

[17] A. Rahmati, C. Shepard, A. Nicoara, L. Zhong, J. Singh. Mobile TCP Usage Characteristics and the Feasibility of Network Migration without Infrastructure Support. In *Proc. of MobiCom (Poster Session)*, Chicago, USA, September 2010.

[18] A. Rahmati, C. Shepard, A. Nicoara, L. Zhong, J. Singh. Seamless flow migration on smartphones without network support. *Technical Report 2010-1214, Rice University*, December 2010.

[19] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. Nericell: Using mobile smartphones for rich monitoring of road and traffic conditions. In *SenSys '08*, Raleigh, NC, 2008.

[20] Rajib Kumar Rana, Chun Tung Chou, Salil S. Kanhere, Nirupama Bulusu, and Wen Hu. Ear-phone: An end-to-end participatory urban noise mapping system. In *IPSN '10*, Stockholm, Sweden.

[21] Chuan Qin, Xuan Bao, Romit Roy Choudhury, and Srihari Nelakuditi. TagSense: a smartphone-based approach to automatic image tagging. In *MobiSys '11*, Bethesda, MD.

[22] Bin Liu, Yurong Jiang, Fei Sha, and Ramesh Govindan. Cloud-enabled privacy-preserving collaborative learning for mobile sensing. In *SenSys '12*, Toronto, Canada.

[23] David Chu, Nicholas D. Lane, Ted Tsung-Te Lai, Cong Pang, Xiangying Meng, Qing Guo, Fan Li, and Feng Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *SenSys '11*, Seattle, Washington.

[24] Eduardo Cuervo, Aruna Balasubramanian, Daeki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *MobiSys '10*, San Francisco, CA.

[25] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, Bethesda, Maryland, USA.

[26] Seungwoo Kang, Youngki Lee, Chulhong Min, Younghyun Ju, Taiwoo Park, Jinwon Lee, Yunseok Rhee, and Junehwa Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *PerCom '10*, Mannheim, Germany.

[27] Nirmalya Roy, Archan Misra, Christine Julien, Sajal K. Das, and Jit Biswas. An energy-efficient quality adaptive framework for multi-modal sensor context recognition. In *PerCom '11*, Washington, DC, USA.

[28] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

[29] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[30] Lei Yu and Huan Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *International Conference on Machine Learning*, pages 856–863, 2003.

[31] B. Wang, W. Wei, J Kurose, D. Towsley, K. Pattipati, Z. Guo, Z. Peng. Application-Layer Multipath Data Transfer via TCP: Schemes and Performance Tradeoffs. *Elsevier Performance Evaluation*, 64(9-12):965–977, October 2007.

[32] S. Kim, J. Copeland. TCP for Seamless Vertical Handoff in Hybrid Mobile Data Networks. In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM'03)*, San Francisco, USA, December 2003.

[33] X. Wu, M. Chan, A. Ananda. TCP HandOff: a Practical TCP Enhancement for Heterogeneous Mobile Environments. In *Proc. of IEEE International Conference on Communications (ICC'07)*, Glasgow, UK, June 2007.

[34] C. Perkins. Mobile IP. *IEEE Wireless Communications Magazine*, 35(5):84–99, May 1997.

[35] P. Nikander, J. Arkko, T. Aura, G. Montenegro. Mobile IP Version 6 (MIPv6) Route Optimization Security Design. In *Proc. of IEEE Vehicular Technology Conference (VTC'03-Fall)*, Orlando, USA, October 2003.

[36] A. Balasubramanian, R. Mahajan, A. Venkataramani. Augmenting Mobile 3G using WiFi. In *Proc. of MobiSys*, San Francisco, USA, March 2010.

[37] P. Sharma, S. Lee, J. Brassil, K. Shin. Handheld Routers: Intelligent Bandwidth Aggregation for Mobile Collaborative Communities. In *Proc. of BroadNets*, San Jose, USA, October 2004.

[38] T. Armstrong, O. Trescases, C. Amza, E. Lara. Efficient and transparent dynamic content updates for mobile clients. In *Proc. of MobiSys*, Uppsala, Sweden, June 2006.

[39] K. Pahlavan, P. Krishnamurthy, A. Hatami, M. Ylianttila, J. Makela, R. Pichna, J. Vallstron. Handoff in Hybrid Mobile Data Networks. *IEEE Personal Communications*, 7(2):34–47, April 2000.

[40] B. Higgins, A. Reda, T. Alperovich, J. Flinn, T. Giuli, B. Noble, D. Watson. Intentional networking: opportunistic exploitation of mobile network diversity. In *Proc. of MobiCom*, Chicago, USA, September 2010.

[41] E. Kushilevitz and Y. Mansour. Learning decision trees using the fourier spectrum. In *ACM Symposium on Theory of Computing 1991 (STOC '91)*.

[42] D. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.

[43] R. O'Donnell. Some topics in analysis of boolean functions (tutorial). In *ACM Symposium on Theory of Computing 2008 (STOC '08)*.

[44] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time series data. In *ACM SIGMOD International Conference on Management of Data 2012 (SIGMOD '10)*, pages 171–182, Indiana, USA.

[45] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *International Conference on Foundations of Data Organization and Algorithms*, pages 69–84, 1993.

[46] I. Daubechies, O. Runborg, and J. Zou. A sparse spectral method for homogenization multiscale problems. *Multi-scale Modeling Simulation*, 6(3):711–740, 2007.

[47] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357 – 366, aug 1980.

[48] Lindasalwa Muda, Mumtaj Begam, and I. Elamvazuthi. Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques. *Journal of Computing*, 2(3):138–143, 2010.

[49] K.S.R. Murty and B. Yegnanarayana. Combining evidence from residual phase and mfcc features for speaker recognition. *Signal Processing Letters, IEEE*, 13(1):52 – 55, jan. 2006.

[50] B. Logan. Mel frequency cepstral coefficients for music modeling. In *International Symposium on Music Information Retrieval 2000 (ISMIR '10)*.

[51] Jesper Jensen, Mads Christensen, Manohar Murthi, and Soren Jensen. Evaluation of mfcc estimation techniques for music similarity. In *European Signal Processing Conference 2006 (EUSIPCO '06)*.

[52] Zheng Fang, Zhang Guoliang, and Song Zhanjiang. Comparison of different implementations of mfcc. *Journal of Computer Science and Technology*, 16(6):582–589, nov 2001.

[53] Anuj Kumar, Anuj Tewari, Seth Horrigan, Matthew Kam, Florian Metze, and John Canny. Rethinking speech recognition on mobile devices. In *2nd International Workshop on International User Interfaces for Developing Regions (IUI4DR)*, Palo Alto, CA, 2011.

[54] Ming-Zher Poh, Kyunghee Kim, A.D. Goessling, N.C. Swenson, and R.W. Picard. Heartphones: Sensor earphones and mobile application for non-obtrusive health monitoring. In *International Symposium on Wearable Computers 2009 (ISWC '09)*, pages 153–154.

[55] P. Celka, C. Verjus, R. Vetter, P. Renevey, and V. Neuman. Motion resistant earphone located infrared based heart rate measurement device. *Biomedical Engineering*, Feb 2004.

[56] Nishkam Ravi, Nikhil Dandekar, Preetham Mysore, and Michael L. Littman. Activity recognition from accelerometer data. In *17th Conference on Innovative Applications of Artificial Intelligence 2005 (IAAI '05)*, pages 1541–1546, Pittsburgh, PA.

[57] Arvind Thiagarajan, James Biagioni, Tomas Gerlich, and Jakob Eriksson. Cooperative transit tracking using smart-phones. In *8th ACM Conference on Embedded Networked Sensor Systems 2010 (SenSys '10)*, pages 85–98, Zurich, Switzerland.

[58] T. Choudhury et. al. The mobile sensing platform: An embedded activity recognition system. *IEEE Pervasive Computing*, 7(2):32–41, April 2008.

[59] Konrad Lorincz, Bor-rong Chen, Geoffrey Werner Challen, Atanu Roy Chowdhury, Shyamal Patel, Paolo Bonato, and Matt Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *7th ACM Conference on Embedded Networked Sensor Systems 2009 (SenSys '09)*, pages 183–196, Berkeley, CA.

[60] Tamara Denning, Adrienne Andrew, Rohit Chaudhri, Carl Hartung, Jonathan Lester, Gaetano Borriello, and Glen Duncan. Balance: towards a usable pervasive wellness application with accurate activity inference. In *10th Workshop on Mobile Computing Systems and Applications 2009 (HotMobile '09)*.

[61] Ling Bao and Stephen S. Intille. Activity recognition from user-annotated acceleration data. *IEEE Pervasive Computing*, 3001:1–17, 2004.

[62] Ryan Aylward and Joseph A. Paradiso. A compact, high-speed, wearable sensor network for biomotion capture and interactive media. In *6th International Conference on Information Processing in Sensor Networks 2007 (IPSN '07)*, pages 380–389, Cambridge, MA.

[63] L Harmat, J Takacs, and R. Bodizs. Music improves sleep quality in students. *Advanced Nursing*, 62:327–335, May 2008.

[64] HL Lai and M Good. Music improves sleep quality in older adults. *Advanced Nursing*, 49(3):234–44, Feb 2005.

[65] E Labbe, N Schmidt, J Babin, and M. Pharr. Coping with stress: the effectiveness of different types of music. *Applied Psychophysiology Biofeedback*, 32(3–4):163–8, Dec 2007.

[66] SD Simpson and CI Karageorghis. The effects of synchronous music on 400-m sprint performance. *Sports Sciences*, 24(10):1095–102, Oct. 2006.

[67] J Edworthy and H Waring. The effects of music tempo and loudness level on treadmill exercise. *Ergonomics*, 49(15):1597–610, Dec 2006.

[68] N Jausovec, K Jausovec, and I Gerlic. The influence of mozart's music on brain activity in the process of learning. *Clinical Neurophysiology*, 117(12):2703–14, Dec 2006.

[69] G Bernatzky et. al. Stimulating music increases motor coordination in patients afflicted with morbus parkinson. *Neuroscience Letters*, 361(1–3):4–8, May 2004.

[70] Costas I. Karageorghis, Leighton Jones, and Daniel C Low. Relationship between exercise heart rate and music tempo preference. *Research Quarterly for Exercise and Sport*, 26:240–250, 2006.

[71] J Escher and D Evequoz. Music and heart rate variability. study of the effect of music on heart rate variability in healthy adolescents. *Praxis*, 88(21):951–2, May 1999.

[72] John C. Platt, Christopher J.C. Burges, Steven Swenson, Christopher Weare, and Alice Zheng. Learning a gaussian process prior for automatically generating music playlists. In *Advances in Neural Information Processing Systems 2001 (NIPS '01)*, pages 1425–1432.

[73] Arthur Flexer, Dominik Schnitzer, Martin Gasser, and Gerhard Widmer. Playlist generation using start and end songs. In *9th International Conference on Music Information Retrieval 2008 (ISMIR '08)*, pages 173–178.

[74] X. Zhu, Y.-Y. Shi, H.-G. Kim, and K.-W. Eom. An integrated music recommendation system. *Consumer Electronics*, 52(3):917–925, Aug. 2006.

[75] Ning Han Liu, Shu Ju Hsieh, and Cheng Fa Tsai. An intelligent music playlist generator based on the time parameter with artificial neural networks. *Expert Systems with Applications*, 37(4):2815–2825, April 2010.

[76] Andreja Andric and Goffredo Haus. Automatic playlist generation based on tracking user's listening habits. *Multimedia Tools and Applications*, 29(2):127–151, June 2006.

[77] Nuria Oliver and Lucas Kreger-Stickles. Papa: Physiology and purpose-aware automatic playlist generation. In *7th International Conference on Music Information Retrieval 2006 (ISMIR '06)*, pages 250–253.

[78] Rodrigo de Oliveira and Nuria Oliver. Triplebeat: enhancing exercise performance with persuasion. In *10th International Conference on Human Computer Interaction with Mobile Devices and Services 2008 (MobileHCI '08)*, pages 255–264.

[79] Nuria Oliver and Fernando Flores-Mangas. Mptrain: a mobile, music and physiology-based personal trainer. In *8th International Conference on Human Computer Interaction with Mobile Devices and Services 2006 (MobileHCI '06)*, pages 21–28, Helsinki, Finland.

[80] Jacob T. Biehl, Piotr D. Adamczyk, and Brian P. Bailey. Djogger: a mobile dynamic music device. In *ACM SIGCHI Conference on Human Factors in computing systems 2006 (CHI '06)*, Montreal, Canada.

[81] Stephen D. Bay and Mark Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD '03*, Washington, D.C.

[82] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357 – 366, aug 1980.

[83] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Simple and practical algorithm for sparse fourier transform. In *ACM-SIAM Symposium on Discrete Algorithms 2012 (SODA '12)*, Kyoto, Japan.

[84] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Nearly optimal sparse fourier transform. In *44th ACM Symposium on Theory of Computing 2012 (STOC '12)*, NewYork, NY.

[85] Younghyun Ju, Youngki Lee, Jihyun Yu, Chulhong Min, Insik Shin, and Junehwa Song. SymPhoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In *SenSys '12*, Toronto, Canada.

[86] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. SeeMon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. MobiSys '08, Breckenridge, CO, USA.

[87] Pengfei Zhou, Yuanqing Zheng, Zhenjiang Li, Mo Li, and Guobin Shen. IODetector: A generic service for indoor outdoor detection. In *SenSys' 12*, Toronto, Canada.

[88] J Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, 2002.

[89] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.

[90] Power Monitor. `http://msoon.com/LabEquipment/PowerMonitor/`.

[91] Karlheinz Brandenburg. *Perceptual Coding of High Quality Digital Audio*, volume 437. Springer US, 2002.

[92] John Watkinson. *The Art of Digital Audio*. Newton, MA, USA, 1993.

[93] Wang, He and Lymberopoulos, Dimitrios and Liu, Jie. Local Business Ambience Characterization Through Mobile Audio Sensing. In *Proc. of the 23rd International World Wide Web Conference (WWW '14)*, Seoul, Korea, April 2014.

[94] Cisco Visual Networking Index: Forecast and Methodology 2009–2014. `http://www.cisco.com/`

en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf.

[95] R. Chalmers, K. Almeroth. A Mobility Gateway for Small Device Networks. In *Proc. of IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, Orlando, USA, March 2004.

[96] N. Thompson, G. He, H. Luo. Flow Scheduling for End-Host Multihoming. In *Proc. of INFOCOM*, Barcelona, Spain, April 2006.

[97] S. Kandula, K. Lin, T. Badirkhanli, D. Katabi. FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput. In *Proc. of NSDI*, San Francisco, USA, April 2008.

[98] P. Rodriguez, R. Chakravorty, J. Chesterfield, I. Pratt, S. Banerjee. MAR: a Commuter Router Infrastructure for the Mobile Internet. In *Proc. of MobiSys*, Boston, USA, June 2004.

[99] T. Alperovich, B. Noble. The Case for Elastic Access. In *Proc. of the MobiArch*, Chicago, USA, September 2010.

[100] G. Ananthanarayanan, I. Stoica. Blue-Fi: enhancing Wi-Fi performance using bluetooth signals. In *Proc. of MobiSys*, Wroclaw, Poland, June 2009.

[101] K. Kim, A. Min, D. Gupta, P. Mohapatra, J. Singh. Improving Energy Efficiency of Wi-Fi Sensing on Smartphones. In *Proc. of INFOCOM*, Shanghai, China, April 2011.

[102] Android Developer Phone 2 (ADP2). http://en.wikipedia.org/wiki/Android_Dev_Phone.

[103] Agilent 34410A Digital Multimeter. http://www.home.agilent.com/agilent/product.jspx?pn=34410A.

[104] N. Balasubramanian, A. Balasubramanian, A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proc. of ACM SIGCOMM Internet Measurement Conference (IMC'09)*, Chicago, USA, 2009.

[105] A. Rahmati, L. Zhong. Context-for-Wireless: Context-Sensitive Energy-Efficient Wireless Data Transfer. In *Proc. of MobiSys*, San Juan, Puerto Rico, June 2007.

[106] Ye-Sheng Kuo, Sonal Verma, Thomas Schmid, and Prabal Dutta. Hijacking power and bandwidth from the mobile phone's audio interface. In *1st ACM Symposium on Computing for Development 2010 (DEV '10)*.

[107] G.M. Friesen, T.C. Jannett, M.A. Jadallah, S.L. Yates, S.R. Quint, and H.T. Nagle. A comparison of the noise sensitivity of nine qrs detection algorithms. *Biomedical Engineering*, 37(1):85 –98, Jan 1990.

[108] J Pan and W.J. Tompkins. A real-time qrs detection algorithm. *Biomedical Engineering*, 32(3):230–236, March 1985.

[109] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, Aug 2004.

[110] F. J. Aherne, N. A. Thacker, and P. Rockett. The bhattacharyya metric as an absolute similarity measure for frequency coded data. *Kybernetika*, 34(4):363–368, 1998.

[111] Olivier Lartillot. Mirtoolbox user's manual. *Finnish Centre of Excelence in Interdisciplinary Music Research*, Dec 2011.

[112] Handheld ecg monitor (md100b). http://www.choicemmed.com/.

[113] Fingertip pulse oximeter. http://www.naturespiritproduct.com.

[114] AL Goldberger, LAN Amaral, L Glass, JM Hausdorff, PCh Ivanov, RG Mark, JE Mietus, GB Moody, C-K Peng, and HE Stanley. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):215–220, Jun 2000.

[115] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[116] Suman Nath. ACE: exploiting correlation for energy-efficient and continuous context sensing. In *MobiSys'12*, Lake District, UK.

[117] Haitham Hassanieh, Fadel Adib, Dina Katabi, and Eric Price. Faster gps via the sparse fourier transform. In *18th Annual International Conference on Mobile Computing and Networking 2012 (MobiCom '12)*, Istanbul, Turkey.

[118] Mengda Lin, A. P. Vinod, Chong Meng, and Samson See. A new flexible filter bank for low complexity spectrum sensing in cognitive radios. *Journal of Signal Processing Systems*, 62(2):205–215, 2011.

[119] Mic Bowman, Saumya K. Debray, and Larry L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 1993.

[120] Johannes Braams. Babel, a multilingual style-option system for use with latex's standard document styles. *TUGboat*, 12(2):291–301, June 1991.

[121] Malcolm Clark. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.
[122] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.
[123] Leslie Lamport. *LaTeX User's Guide and Document Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
[124] S.L. Salas and Einar Hille. *Calculus: One and Several Variable*. John Wiley and Sons, New York, 1978.
[125] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP'97), St. Malo, France*.
[126] Brian Noble. System Support for Mobile, Adaptive Applications. In *IEEE Personal Communications*, 2000.
[127] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP'99)*.
[128] Yunsi Fei, Lin Zhong, and Niraj K. Jha. An Energy-aware Framework for Coordinated Dynamic Software Management in Mobile Computers. In *Proc. of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*.
[129] W. Yuan, K. Nahrstedt, and X. Gu. Coordinating Energy-Aware Adaptation of Multimedia Applications and Hardware Resource. In *Proceedings of the 9th ACM Multimedia (Multimedia Middleware Workshop), Ottawa, Canada*, 2001.
[130] S. Adve, A. Harris, C. Hughes, D. Jones, R. Kravets, K. Nahrstedt, D. Sachs, R. Sasanka, J. Srinivasan, and W. Yuan. The Illinois GRACE Project: Global Resource Adaptation through Cooperation. In *Proc. of the Workshop on Self-Healing, Adaptive, and self-MANaged Systems (SHAMAN)*, 2002.
[131] R. Barga, D. Lomet, and G. Weikum. Recovery Guarantees for General Multi-Tier Applications. In *Proc. of the IEEE International Conference on Data Engineering (ICDE'02), San Jose, California*, 2002.
[132] R. Want, B. Schilit, A. Norman, R. Gold, D. Goldberg, K. Petersen, J. Ellis, and M. Weiser. An Overview of the ParcTab Ubiquitous Computing Environment. In *IEEE Personal Communications, 2(6):28-43*, 1995.
[133] Damian Arregui, Francois Pacull, and Jutta Willamowski. Rule-Based Transactional Object Migration over a Reflective Middleware. In *Proc. of the Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*.
[134] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proc. of the Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*.
[135] Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. Caching Strategies for Data-Intensive Web Sites. In *Proc. of 26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt*, 2000.
[136] Progress DataXtend Database Cache. www.progress.com.
[137] Cisco Systems. http://www.cisco.com/.
[138] JBoss Application Server. http://www.jboss.org/products/jbosscache.
[139] Java Caching System. http://jakarta.apache.org/jcs/.
[140] Squid Web Proxy Cache. http://www.squid-cache.org/.
[141] ECMA International. *Standard ECMA-335, Common Language Infrastructure (CLI)*, 4 edition, June 2006.
[142] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular Software Upgrades for Distributed Systems. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, July 2006.
[143] Sameer Ajmani, Barbara Liskov, Liuba Shrira. Scheduling and Simulation: How to Upgrade Distributed Systems. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS IX), Lihue, Hawai*, May 2003.
[144] J. Apostolopoulos, M. Trott. Path Diversity for Enhanced Media Streaming. *IEEE Communications Magazine*, 42(8):80–87, August 2004.
[145] E. Gustafsson, A. Jonsson. Always Best Connected. *IEEE Wireless Communications Magazine*, 10(1):49–55, February 2003.
[146] A. Sgora, D. Vergados. Handoff Prioritization and Decision Schemes in Wireless Cellular Networks: a Survey. *IEEE Communications Surveys & Tutorials*, 11(4):57–77, Fourth Quarter 2009.
[147] Y. Wang, S. Wenger, J. Wen, A. Katsaggelos. Error Resilient Video Coding Techniques. *IEEE Signal Processing Magazine*, 17(4), July 2000.
[148] Web Page of Video Traces Research Group, 2010. `http://trace.eas.asu.edu/tracemain.html`.

[149] Federal Communications Commission, Fourteenth Report May 2010. https://www.fcc.gov/14report.pdf.

[150] K. Tsao, R. Sivakumar. On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts. In *Proc. of ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, Rome, Italy, December 2009.

[151] K. Lee, I. Rhee, J. Lee, Y. Yi, S. Chong. Mobile data offloading: how much can WiFi deliver? In *Proc. of ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, PA, USA, December 2010.

[152] S. Nirjon, A. Nicoara, C. Hsu, J. Singh, J. Stankovic. MultiNets: Policy Oriented Real-Time Switching of Wireless Interfaces on Mobile Devices. *Technical Report CS-2011-08, University of Virginia*, October 2011. http://goo.gl/DDAuj.

[153] Veronique Roger. Heart disease and stroke statistics–2012 update. *American Heart Association*, Dec 2011.

[154] Chio-In Ieong, Mang i Vai, Peng-Un Mak, and Pui-In Mak. Ecg heart beat detection via mathematical morphology and quadratic spline wavelet transform. In *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, Jan 2011.

[155] P Hamilton. Open source ecg analysis. In *Computers in Cardiology, 2002*, sept. 2002.