IMPROVING FAILURE MANAGEMENT THROUGH COOPERATION BETWEEN MOBILE DEVICES AND CELLULAR NETWORK

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Nawanol Theera-Ampornpunt

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2017

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Saurabh Bagchi, Chair

School of Electrical and Computer Engineering

Dr. Sonia Fahmy

Department of Computer Science

Dr. Jennifer Neville

Department of Computer Science

Dr. He Wang

Department of Computer Science

Approved by:

Dr. Voicu Popescu by Dr. William Gorman

Head of the Graduate Program

To my parents and brother, who are always there to support me.

ACKNOWLEDGMENTS

I would like to acknowledge the many people who contributed to this dissertation. First, I would like to thank my advisor, Professor Saurabh Bagchi, whose invaluable feedback is essential in shaping this dissertation. His suggestions enhanced my critical thinking, problem solving, and idea presentation, and helped my grow as a researcher.

I am thankful for my committee members, Professor Sonia Fahmy, Professor Jennifer Neville, and Professor He Wang, whose insightful questions and constructive feedback helped improve this dissertation.

I have been privileged to collaborate with exceptional researchers during the course of this dissertation, both at Purdue and outside. I am especially thankful for Dr. Rajesh Panta and Dr. Kaustubh Joshi from AT&T Labs Research, for the collaboration opportunity, for providing their expertise on cellular networks, and for mentoring me during my internship at AT&T.

Lastly, special thanks to members of the Dependable Computing Systems Laboratory (DCSL), for their feedback and support in all aspects of my dissertation.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . vii

LIST OF FIGURES . . . viii

ABBREVIATIONS . . . x

ABSTRACT . . . xii

1 INTRODUCTION . . . 1
- 1.1 Contributions . . . 1
  - 1.1.1 Using Big Data to Improve Dependability in Cellular Network . . . 1
  - 1.1.2 TANGO: Toward a More Reliable Mobile Streaming through Cooperation between Cellular Network and Mobile Devices . . . 2
  - 1.1.3 AppStreamer: Reducing Storage Requirements of Mobile Games through Predictive Streaming . . . 3
- 1.2 Published Work . . . 3

2 USING BIG DATA TO IMPROVE DEPENDABILITY IN CELLULAR NETWORK . . . 5
- 2.1 Introduction . . . 5
- 2.2 Background . . . 6
  - 2.2.1 Network Architecture . . . 6
  - 2.2.2 Mechanisms that Improve Reliability . . . 7
- 2.3 Case Studies . . . 9
  - 2.3.1 Data Source . . . 9
  - 2.3.2 Methodology . . . 10
  - 2.3.3 Predicting Drops . . . 10
  - 2.3.4 Predicting Drop Duration . . . 14
- 2.4 Related Work . . . 16
- 2.5 Conclusions and Challenges . . . 17

3 TANGO: TOWARD A MORE RELIABLE MOBILE STREAMING THROUGH COOPERATION BETWEEN CELLULAR NETWORK AND MOBILE DEVICES . . . 18
- 3.1 Introduction . . . 18
- 3.2 Background . . . 20
- 3.3 Data Pre-caching Service . . . 21
  - 3.3.1 Overview . . . 21

Page

3.3.2 Location Prediction . . . 24
3.3.3 Deployment . . . 25
3.4 Evaluation . . . 26
3.4.1 Data Source . . . 26
3.4.2 Experiments . . . 27
3.5 Related Work . . . 46
3.6 Conclusions . . . 48
4 APPSTREAMER: REDUCING STORAGE REQUIREMENTS OF MOBILE GAMES THROUGH PREDICTIVE STREAMING . . . 49
4.1 Introduction . . . 49
4.2 Related Work . . . 54
4.3 Design Overview . . . 56
4.4 Components of AppStreamer . . . 58
4.4.1 File Access Capture . . . 58
4.4.2 File Access Prediction Model . . . 59
4.4.3 Block Grouping . . . 60
4.4.4 Markov Chain . . . 65
4.4.5 Data Block Fetcher . . . 68
4.4.6 Initial Files Cached . . . 69
4.4.7 Temporary Storage Limit . . . 69
4.5 Implementation . . . 70
4.5.1 Capturing File Accesses . . . 70
4.5.2 Block Storage . . . 71
4.6 Experimental Evaluation . . . 72
4.6.1 Games Used for Evaluation . . . 72
4.6.2 Training Data . . . 73
4.6.3 User Study – Dead Effect 2 . . . 73
4.6.4 User Study – Fire Emblem Heroes . . . 76
4.6.5 Comparison with Prior Work . . . 78
4.6.6 Microbenchmarks . . . 81
4.7 Discussion . . . 84
4.8 Conclusion . . . 85
5 CONCLUSIONS . . . 86
REFERENCES . . . 87
VITA . . . 92

LIST OF TABLES

Table — Page

2.1 Class distribution of data points from one cell given comparison results of the top two attributes 13

3.1 Different types of simulated congestions used in the experiments. 30

3.2 Some key take-aways from the experimental evaluation. 46

LIST OF FIGURES

Figure Page

2.1 Diagram of the proposed architecture. 6

2.2 Cumulative distribution of cell drop rate. 11

2.3 Drop prediction accuracy as a function of weight parameter. 12

2.4 Drop rate of four randomly selected cells 14

2.5 Drop duration prediction accuracy as a function of weight parameter for SVM and AdaBoost. 16

3.1 Architecture of a UMTS data network. 20

3.2 Overview of pre-caching service in TANGO. 23

3.3 Audio pause time as a function of the number of audio streaming users. . . 32

3.4 Average audio stream quality as a function of the number of audio streaming users. 34

3.5 Average wasted bandwidth per user due to user abandonment as a function of the number of audio streaming users. 35

3.6 Audio pause time as a function of buffer size. 37

3.7 Parameter sensitivity of pre-caching. 39

3.8 Video pause time in the simple case where the user is watching one long video and move from a uncongested cell to a congested cell. 41

3.9 Location prediction accuracy with varying history length. 42

3.10 Location prediction accuracy with varying alert threshold. 43

4.1 Overview diagram of AppStreamer. 52

4.2 Cumulative file read in Dead Effect 2. 53

4.3 Grouping of blocks during training. 61

4.4 Storage requirements for Dead Effect 2. 74

4.5 User study results for Dead Effect 2. 75

4.6 Storage requirements for Fire Emblem Heroes. 76

Figure Page

4.7 User study results for Fire Emblem Heroes. 77

4.8 Comparison of bandwidth consumption between cloud gaming and AppStreamer. 79

4.9 Microbenchmarks for Dead Effect 2. 82

ABBREVIATIONS

3G	3rd Generation
3GPP	The 3rd Generation Partnership Project
4G LTE	4th Generation Long-Term Evolution
APK	Android Package Kit
cell	cell sector
CN	Core Network
CTMC	Continuous-time Markov Chain
FIFO	First-in, First-out
FPS	First-person Shooter
GGSN	Gateway GPRS Support Node
GHz	Gigahertz
GPRS	General Packet Radio Service
GTP	GPRS Tunneling Protocol
HSPA	High-Speed Packet Access
IP	Internet Protocol
Kbps	Kilobits per Second
LRU	Least-recently Used
ms	millisecond
OBB	Opaque Binary Blob
OS	Operating System
PDN-Gateway	Packet Data Network Gateway
PDP	Packet Data Protocol
QoS	Quality of Service
RAB	Radio Access Bearer

RAN	Radio Access Network
RAT	Radio Access Technology
RNC	Radio Network Controller
RRC	Radio Resource Control
S-Gateway	Serving Gateway
SVM	Support Vector Machine
TCP	Transmission Control Protocol
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
VA	Virtual Appliance

ABSTRACT

Theera-Ampornpunt, Nawanol PhD, Purdue University, August 2017. Improving Failure Management through Cooperation between Mobile Devices and Cellular Network. Major Professor: Saurabh Bagchi.

Mobile devices have become an integral part of our lives. As people rely more on them, the traffic demand has increased rapidly, outpacing the growth of the capacity of cellular networks. As a result, connectivity problems such as congestions are becoming more common. In a similar manner, users' increasing demand for storage space on mobile devices leads to major inconvenience when available space runs out. In this dissertation, we present a novel method of mitigating or preventing the negative effects of such connectivity issues in multimedia streaming applications, as well as a technique for reducing storage requirements of mobile applications.

Mobile streaming applications usually limit the download rate in some way, in order to conserve user's bandwidth. However, when connectivity is degraded, playback can easily be disrupted. To prevent this, we propose a novel framework called TANGO, where real-time network conditions combined with the user's location prediction are used to give the application an early notification of network degradation. This allows the application to change its buffering strategy proactively in order to prevent playback disruption.

We next focus on reducing storage requirements of mobile applications, especially games, through predictive streaming. The size of mobile applications and the users' demand for storage have been outpacing the growth of storage capacity of mobile devices. This leads to the users frequently having to uninstall some applications or remove personal files in order to free up storage for new applications. We propose a technique called AppStreamer, which predicts applications' file accesses and use this

information to cache the applications' resource files in a smart way. We implement AppStreamer on Android and evaluate the effects it has on user experience using user studies. The results indicate that most people notice no degradation of user experience, while the storage requirements of the application can be reduced by more than 85%.

1 INTRODUCTION

Mobile devices have become an integrated part of our lives, and more applications are being used by mobile users. As a result, reliability of mobile applications and services has increasingly become more important to the user. In current practices, mobile applications view the cellular network as no more than a "dumb pipe," oblivious to the dynamic conditions and intricacies of the network. On the other hand, the network can see the traffic (if it is not encrypted), but lacks the information about how the data is being consumed and presented to the user. In this dissertation, we explore different ways the mobile device and cellular network can cooperate to improve dependability of mobile applications and services.

1.1 Contributions

This section summarizes the contributions of this dissertation.

1.1.1 Using Big Data to Improve Dependability in Cellular Network

Large infrastructures often employ monitoring mechanisms that instrument many things from system performance to detailed user activities. This results in large amount of data collected. Such data is often used for offline analysis, but rarely online. In this work, we explore how data collected from cellular networks can be used to improve the user experience and dependability of mobile applications. We tackle two specific problems: predicting disconnections and predicting duration of call drops. The former can help applications such as multimedia streaming applications prepare for impending disconnection by pre-caching more content beforehand. The

latter can be used to enable a more seamless reconnection after a voice call drop without the annoyance of reconnection attempt that takes too long or ends in failure.

1.1.2 TANGO: Toward a More Reliable Mobile Streaming through Cooperation between Cellular Network and Mobile Devices

Multimedia streaming applications is an important class of applications that accounts for more than half of mobile traffic. Due to the size of the content, most mobile streaming applications employ some mechanisms that limit the download rate in order to limit energy cost and bandwidth usage such as limited buffer size. However, in current practices, the application treats the cellular network as a "dumb pipe," and knows nothing about the current or future condition of the network other than the current quality of its own network connections. As a result, the application can react to changes, e.g., using adaptive bit-rate streaming, in network condition only *after* the connectivity has deteriorated. Ideally, the buffer size need to be set higher before and during periods of connectivity problems. However, without a way to predict future quality of the network connection, the application can only choose one fixed buffer size, effectively forced to make a tradeoff between resource usage when connectivity is good and chance of playback disruption when connectivity is bad.

We present TANGO, a framework that enables real-time data analysis using data from the cellular network, through which a new class of services can be built. We design and implement a pre-caching service, whereby network condition monitoring is combined with user location prediction to give streaming applications an early notification of connectivity degradation. The application can then initiate a mitigation action, such as pre-caching more content to the buffer. In effect, we turn on its head two operational principles today. First, we enable cooperation between the cellular network and the mobile device through our framework, which current practice does not allow for such cooperation. Second, we use this cooperation to allow proactive handling of network condition changes, while the overriding operation mode today is

through adaptive bit-rate streaming which reacts to the changing network condition in the hope that the condition will change slowly. Network conditions do change quickly in many scenarios, such as, flash crowds or changing spots with poor signal strength.

1.1.3 AppStreamer: Reducing Storage Requirements of Mobile Games through Predictive Streaming

The growth of mobile applications' size has been outpacing the growth of storage available on mobile devices. In addition, as mobile devices become more and more powerful, users do more things on them, and their storage demand increases. In current practices, all applications need to be downloaded, installed, and stored in their entirety in order to be used. However, we observe that large applications often use only a small part of their resource files during a typical usage session. Based on this idea, we propose AppStreamer, a technique for reducing storage requirements of mobile applications. AppStreamer works by predicting file blocks that will be read by the application, and downloading only these necessary blocks from the cloud storage server before they are actually read by the application. Only a small part of the application which is read from the start needs to be stored on the device at all times. AppStreamer is implemented at the file system layer of the operating system, and therefore does not require application modification or source code. Evaluation through user study shows that AppStreamer drastically reduce amount of storage used while keeping the user experience similar to the original unmodified version.

1.2 Published Work

Parts of this dissertation have been published in or submitted to peer-reviewed conferences.

Published works:

- **Using Big Data for More Dependability: A Cellular Network Tale**
  Nawanol Theera-Ampornpunt, Saurabh Bagchi, Kaustubh Joshi, and Rajesh Panta. In *Proceedings of 9th Workshop on Hot Topics in Dependable Systems* (HotDep 2013), Farmington, Pennsylvania, 3 November 2013.

- **TANGO: Toward a More Reliable Mobile Streaming through Cooperation between Cellular Network and Mobile Devices**
  Nawanol Theera-Ampornpunt, Tarun Mangla, Saurabh Bagchi, Rajesh Panta, Kaustubh Joshi, Mostafa Ammar, and Ellen Zegura. In *Proceedings of the 35th Symposium on Reliable Distributed Systems* (SRDS 2016), Budapest, Hungary, 26-29 September 2016.

Work under review:

- **AppStreamer: Reducing Storage Requirements of Mobile Games through Predictive Streaming**
  Nawanol Theera-Ampornpunt, Sameer Manchanda, Saurabh Bagchi, Rajesh Panta, Kaustubh Joshi, Mostafa Ammar. (Under review) *The 26th ACM Symposium on Operating Systems Principles* (SOSP 2017), Submitted April 2017.

2 USING BIG DATA TO IMPROVE DEPENDABILITY IN CELLULAR NETWORK

2.1 Introduction

There are many examples of large infrastructures that instrument everything from regular network performance metrics to detailed user activities. For example, cellular networks collect information about bandwidth usage, handovers, signal strength, connection/disconnection events, etc., to analyze network performance. Web analytics record user clicks, location, page visit time, page dwell time, etc., to help page owners understand their traffic. The collection of these large quantities of analytics from large infrastructures and their use for understanding user behaviors and trends has been called "big data".

Unfortunately, today, these measurements are not used in meaningful ways *online*, if at all. Instead, they are often used to do offline analysis. In this chapter, we explore whether real-time data collected from detailed instrumentation of large scale infrastructures can be used to provide real-time services that improve the dependability of the infrastructures and through this, the experience for users using them.

Specifically, this chapter presents our vision of how big data analytics can be used in real time to enhance the dependability of cellular network services. We demonstrate how big data can help in the design of adaptive techniques to reduce the incidence of voice or data disconnections in cellular networks and mitigate their effect when they do occur. We show that such mitigation mechanisms come with a cost that prohibits them from being turned on all the time. In order for them to be beneficial, real-time data analysis is necessary because network disconnections depend not only on static factors (such as user locations that are prone to bad network connectivity), but also on dynamic factors (such as current level of congestion in the cell, available radio

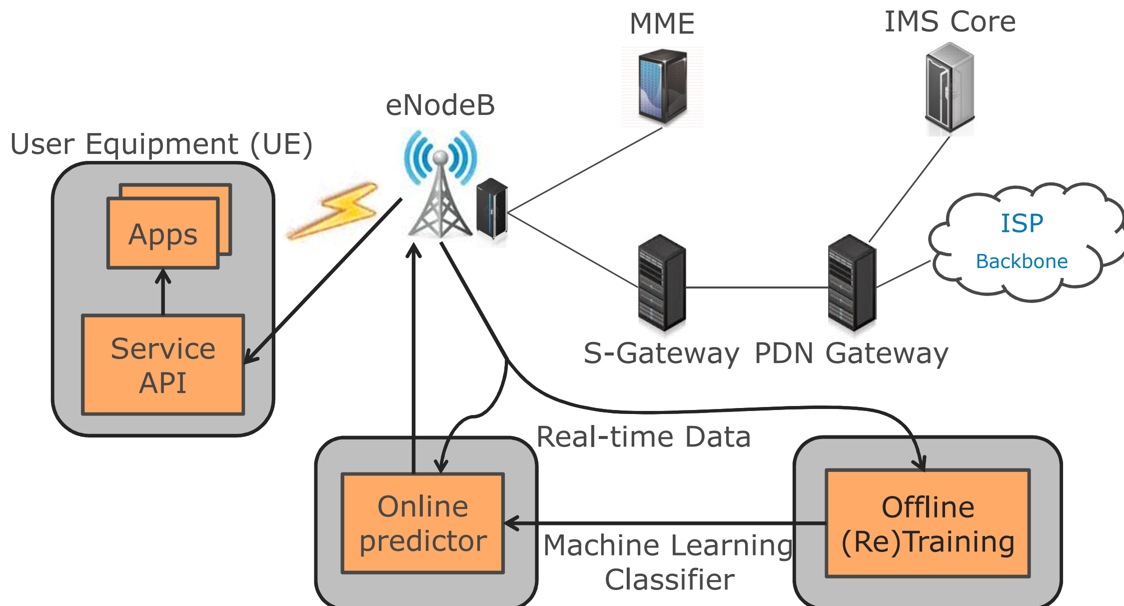

Figure 2.1. Diagram of the proposed architecture.

resources, etc.). We analyze real data collected by a major cellular network and show that we can build a model that identifies conditions that are likely to lead to network disconnections where real time mitigation mechanisms are beneficial.

2.2 Background

2.2.1 Network Architecture

Figure 2.1 shows our proposed architecture for the LTE network. The mobile device, called User Equipment (UE), is connected to a cell sector (henceforth referred to as cell) in a base station, called eNodeB. A physical base station can have multiple sectors, potentially covering different regions. Cellular traffic from the eNodeB passes through Serving Gateway (S-Gateway) and Packet Data Network Gateway (PDN-Gateway) to external network (e.g., the Internet).

In order to identify dynamic conditions that can predict voice or data drops with sufficient confidence, we add two components in the proposed architecture as shown

in Figure 2.1: offline training and online predictor. The offline training component takes data from eNodeBs and construct a machine learning classifier for a specific prediction task. Due to changing network conditions over time, this offline training needs to be repeated often (e.g., every day). The learned classifier becomes the online predictor, which makes predictions using real-time data from the eNodeB. When the event of interest occurs (e.g., probability of a drop exceeds the threshold), the online predictor sends a notification to the component(s) responsible for initiating the mitigation actions. This could be the mobile device itself (e.g., to initiate pre-caching), or a component in the network (e.g., switching to older network technology) or both. In case where the device is involved, the notification is first sent to the Service API in the device, which dispatches it to applications that have registered to receive that particular type of notifications.

In this study, we focus on one type of failures: abnormal voice or data disconnections (also referred to as "drops"). In this work, abnormal disconnections are defined as abnormal release of Radio Access Bearer (RAB) or Radio Resource Control (RRC) connection [1]. Before performing any voice call or data communication, a phone first needs to establish an RRC connection. This is followed by a RAB connection, which assigns essential network resources to the user device based on the quality of service required by the specific application. Since different applications can have different QoS requirements, a single device may have multiple RAB connections, but only a single RRC connection at any given time. Our first case study focuses on predicting the failures, while the second case study focuses on predicting how long the failure will last (referred to as "drop duration"), given that a failure has occurred.

2.2.2 Mechanisms that Improve Reliability

Before describing the offline training and online predictor components, we show that the various techniques that can be used to mitigate the effects of possible data or voice disconnections incur some cost, and thus cannot be used all the time. It is also

important to update offline classifier regularly and perform prediction using real time data to identify conditions that can lead to abnormal connection drops as accurately as possible.

Switching to Older Technology: As newer Radio Access Technologies (RATs) usually provide better performance and/or new features, users tend to prefer them over older technologies and devices are configured to use them whenever possible. This leads to higher congestion, as each RAT uses its own base stations and other infrastructures. Thus, older RATs generally have higher reliability than newer ones. The main drawback of using older technology is the lower connection speed, which could be more important than the reliability of the connection, depending on the application. For example, a video streaming application may be able to mask a disconnection by buffering the video before it occurs, if the download speed is high enough. On the other hand, when making a voice call in a congested region, it may be better to use an older, less congested RAT despite the drop in audio quality.

Prefetching: Prefetching can be done for applications where future content is known in advanced or can be predicted, such as web browsing (link prefetching) as well as audio/video streaming (increasing the buffer size). If used early enough before the connection drop, and the drop does not last very long, it can mask the drop completely. Otherwise, it still allows the user to access more content before being affected by the network problem. This method carries a cost of wasted bandwidth and energy if left on longer than necessary or all the time.

Voice Call Auto-Reconnecting: In the current state of practice, if a disconnection occurs during a voice call, the call is simply dropped. It is straightforward to add a functionality of automatic reconnection to voice calls, which will make reconnection more seamless for both parties. However, because most of the time, a drop lasts rather long, this would inconvenience the user as well as the other party of the call while he or she waits. It it better to use the mechanism only when the drop duration is expected to be short.

While the mechanisms mentioned above do improve reliability and/or user experience, they all come with some cost that prohibits them from being used all the time. What we need is a way to determine the conditions (e.g., how likely a drop is going to occur in the immediate future), the temporal nature of the failure (e.g., how long the drop is expected to last), take into account the device specific information (e.g., type of applications the user is running), and use one of the mechanisms only when the benefits outweigh the costs. In the next section, we will show how big data analytics can be used to identify these conditions.

2.3 Case Studies

In this section we explore how we can identify conditions needed by the mitigation mechanisms described in Section 2.2.2 with enough confidence. We focus on two specific problems: predicting drops and predicting drop duration.

2.3.1 Data Source

We use real-world 3G cellular traffc data from a tier-1 U.S. cellular network collected on or after June 5, 2012. Although the 3G network has slightly different architecture from LTE as described in Section 2.2.1, the prediction mechanism described here can be applied for LTE networks as well. Data are collected at the NobeB (which is analogous to LTE's eNodeB) during its normal operation, and aggregated at the Radio Network Controller (RNC), which manages multiple NodeB's. All devices and user identifiers are anonymized for our analysis, and packet inspection was not used. The dataset contains various low-level performance events, each of which contains common metrics as well as its own set of metrics. Common metrics include timestamp, the hash of user's International Mobile Subscriber Identity (IMSI), cell IDs of the cells the device is connected to. Examples of logged events include connections/disconnections, cell load, primary cell of each UE, and download and upload throughput. Once a model is trained, the original data used to train it are discarded.

2.3.2 Methodology

Because different events have different reporting interval, we need to combine them by computing aggregate functions (e.g., average, count, last report value) of the values within a sliding time window. Specifically, for each time window and each metric, we compute 10 aggregate functions of that metric values within the window. If no value is reported within the window, we report it as a special missing value. Aggregated values across all metrics corresponding to the same window form a data point, used to train and evaluate classifiers. For the task of predicting drops, a data point is assigned the "failure" class label if the window it corresponds to precedes a failure by 20 seconds or less. Otherwise, it is assigned the "normal" class label. For the task of predicting drop durations, since we are only making predictions when a drop has just occurred, we only include windows that lead up to, but not including, the drop. We use window size of 10 seconds. We select 12 events with 250 metrics altogether, leading to 2,500 attributes after computing the aggregates.

We use two machine learning algorithms for our data analysis: AdaBoost and Support Vector Machine (SVM). We use Weka's implementation of AdaBoost, with decision stump as the base classifier [2]. For SVM, we use LIBSVM implementation [3]. All prediction accuracy results are generated using separate train and test datasets, each collected from one day of operation.

2.3.3 Predicting Drops

In this first case study, we look at whether we can identify conditions correlated with disconnections. For this purpose we consider disconnections of both voice calls and data connections. There are many ways to partition the models; we can build a single global model, or train a separate model for each cell sector, or build a personalized model for each user. We found that accuracy of personalized models suffer from the lack of sufficient amount of data, because for most users, there is not enough failure data to build a good model.

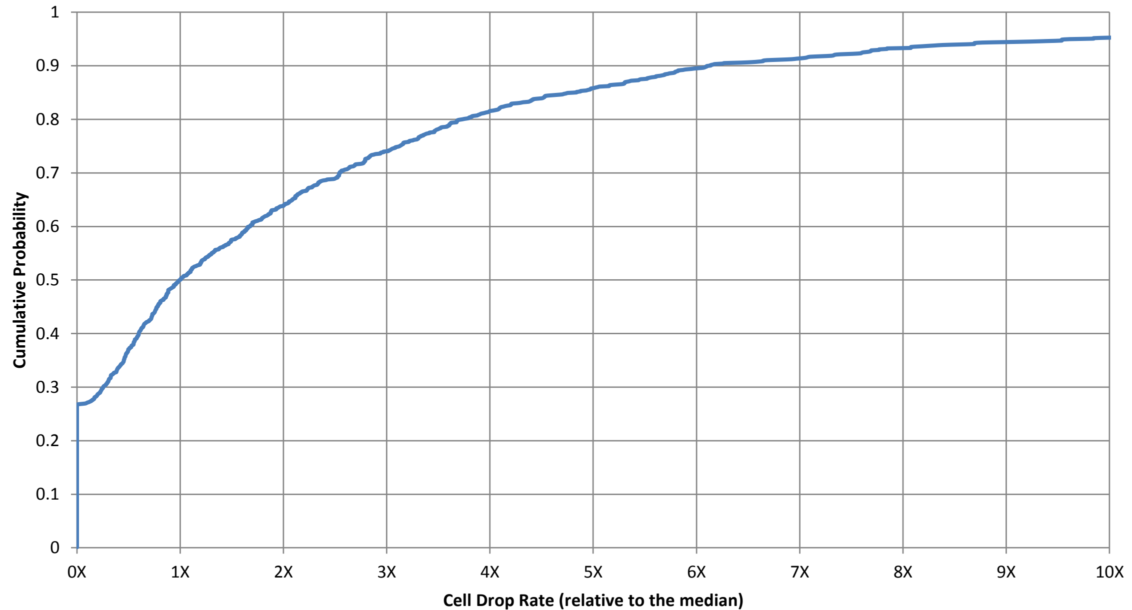

Figure 2.2. Cumulative distribution of cell drop rate.

The question then becomes, are the reliability characteristics of each cell different from each other enough to justify partitioning the models by cell? To answer this question, we look at the drop rate of each cell from the same time window. For this study, we define the drop rate to be the number of drops during a period divided by the amount of data traffic used across all users during the same period.

Figure 2.2 shows the cumulative distribution of cell drop rate from the operational period of April 5 to June 24, 2013. Each cell's drop rate is shown relative to the median drop rate among these cells. The drop rate ranges from zero to 107 times the median. 26% of all cells have no drop at all during this period. From the figure, we can see that there is a high variation among the cells. Thus, it is reasonable to partition the models by cell sector.

We separate the data by cell and use AdaBoost to train a model for each cell. Due to the large amount of data, we need to sample and include only a fraction of users for each cell. Figure 2.3 shows the prediction accuracy in terms of precision and recall. The weight parameter is a tunable parameter that controls the relative

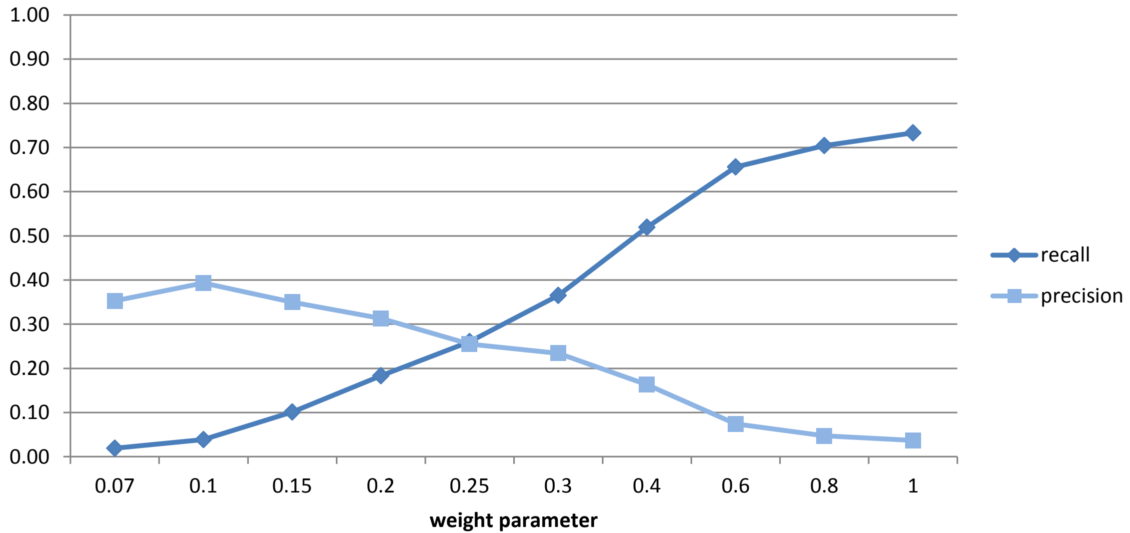

Figure 2.3. Drop prediction accuracy as a function of weight parameter.

cost of a false positive versus the cost of a false negative (higher weights mean false negatives cost more). Recall and precision have their conventional meaning. Recall is the proportion of actual failure cases that the classifier is able to predict. Precision is the fraction of actual failure cases to the number of predicted failures. Thus, recall is the complement of false negative rate and precision is the complement of false positive rate.

While the precision of roughly 25% might seem low, it is enough for applications where maintaining a connection is essential. Specifically, when the classifier predicts that a drop will occur, there is a probability of 25% that a drop will actually occur. If we initiate a mitigation action based on this prediction, 25% of the time it would be the correct course of action, while 75% of the time the costs of the mitigation action are incurred unnecessarily. Depending on the action, the costs may be lower audio quality, wasted energy and bandwidth, or something else. Our experience shows that prediction accuracy varies based on the operational period from which the data used to train and evaluate the models are collected. Also, there is potential for the accuracy to be improved, for example by using a better sampling method, including

Table 2.1.
Class distribution of data points from one cell given comparison results of the top two attributes compared to the overall fraction of failure data points (labeled X). 'greater' and 'not greater' refers to the outcome of the comparison between the value and the learned threshold.

A1	A2	Fraction of failure data points
not greater	not greater	$0.51X$
not greater	greater	$2.04X$
greater	not greater	$1.82X$
greater	greater	$40X$
Any	Any	X

more types of logged events in the data analysis, or extracting better features. Next, we illustrate how the attributes used by the classifier correlate to the failures.

Each classifier is different, but we found that many classifiers have the same top two attributes that influence the prediction decision the most: 1) the number of UE's uplink throughput records whose value is zero, and 2) the sum of the cell's transmit power across all records within a window. We will refer to these attributes as A1 and A2, respectively. Table 2.1 shows the class distribution of data points from the test dataset from one cell for each combination of outcomes of comparisons between the attribute value and the threshold learned by the Decision Stumps corresponding to A1 and A2. The fraction of failure data point for each combination is shown in relative to the overall fraction of failure data points. We can see that when either If either A1 or A2 (but not both) exceeds its threshold, the fraction roughly doubles. However, when both A1 and A2 exceed its threshold, the fraction becomes 40 times higher. Since the throughput is reported every 2 seconds, even if it is zero, the fact that there are many records with upload throughput of zero indicates that there is some problem with the communication. The cell's transmit power is related to the current load on the cell, which is correlated with drops.

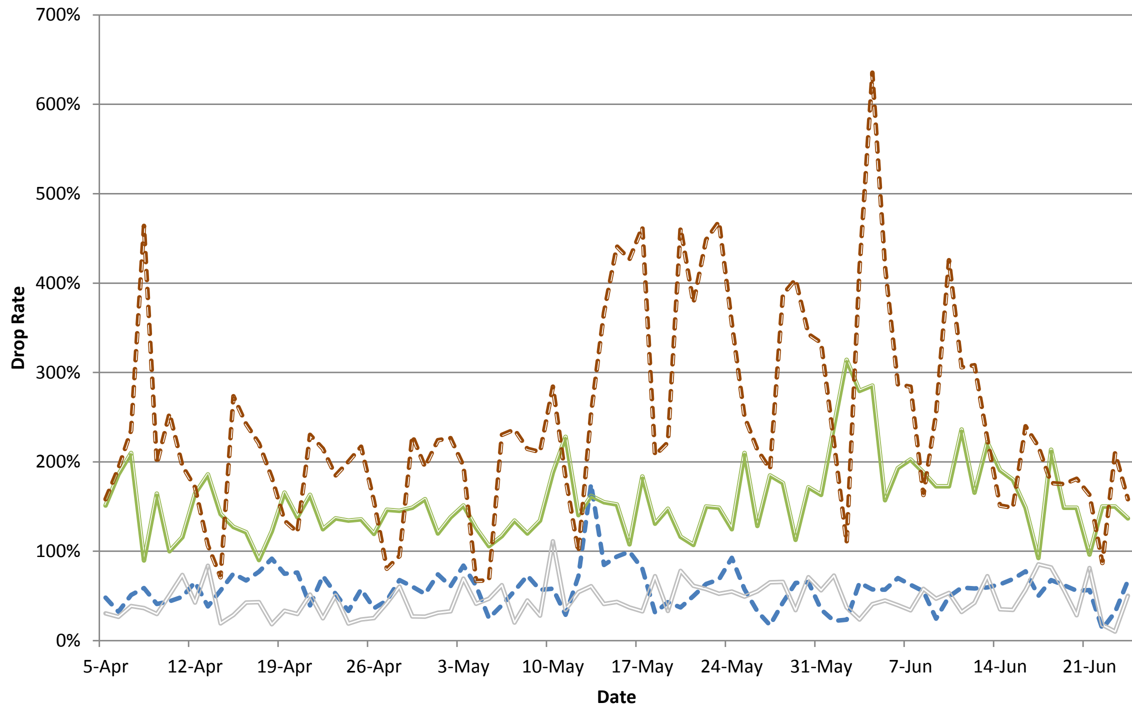

Figure 2.4. Drop rate of four randomly selected cells on each day from April 5 to June 24, 2013, compared to the median drop rate.

Next, we explore how the conditions change over time, as this will determine how often the classifiers need to be retrained. We randomly pick four cells and plot their drop rate relative to the median drop rate during the period from April 5 to June 24, 2013 in Figure 2.4. We can see that the drop rate of some cells change significantly over time, and at different time from other cells. In order to capture these changes in conditions, the models need to be retrained frequently.

2.3.4 Predicting Drop Duration

Although related, predicting drop duration is a separate problem from predicting connection drops. The question we ask here is, given that a drop has occurred, what is the earliest time that the connection can be reestablished. We will refer to the duration between these two events as 'drop duration'. This could depend on the

user's mobility pattern and environmental conditions, among other factors. This is important for guiding if a mitigation action is likely to be useful — a short drop duration would mean that it is acceptable to pause the call and then resume it when the connection can be reestablished, while a long drop duration would mean that it is better to simply drop the call.

Unfortunately, we cannot directly determine the earliest time that a reconnection attempt would be successful from our data source. Specifically, our data source only reports successful reconnections (since the network does not know about unsuccessful reconnections). The 3GPP standard does not require the device to attempt to reconnect after a disconnection. Thus, an absence of successful reconnection does not imply that an attempt would have been unsuccessful. However, this is the best method available to us to estimate the drop duration and we use it with an understanding that this is an upper bound of the true drop duration.

Since the goal of drop duration prediction is to make a decision whether to hold the voice call while attempting to reconnect, instead of predicting the drop duration directly, we simplify the problem by predicting whether the drop will be short or long instead. The threshold is set as $t = 10$ seconds. This decision needs to be agreed on by both the network and the disconnected party. However, no communication between the two is possible once a party is disconnected. Thus, during a voice call, the online predictor needs to keep analyzing real-time data and keep the devices of both parties updated about the decision (whether to drop the call immediately or attempt to reconnect), so that once a drop occurs, both of them may initiate the appropriate action.

Due to variability of drop duration across cells, the classifiers should ideally be partitioned by cell. However, due to time constraints, we only have results from a single global classifier used for all cells. Figure 2.5 shows the accuracy of drop duration prediction for SVM and AdaBoost with different values of weight parameter. Because the two algorithms have different ranges of weight parameter, the actual values are not shown on the axis. Here, recall is the proportion of short drops that

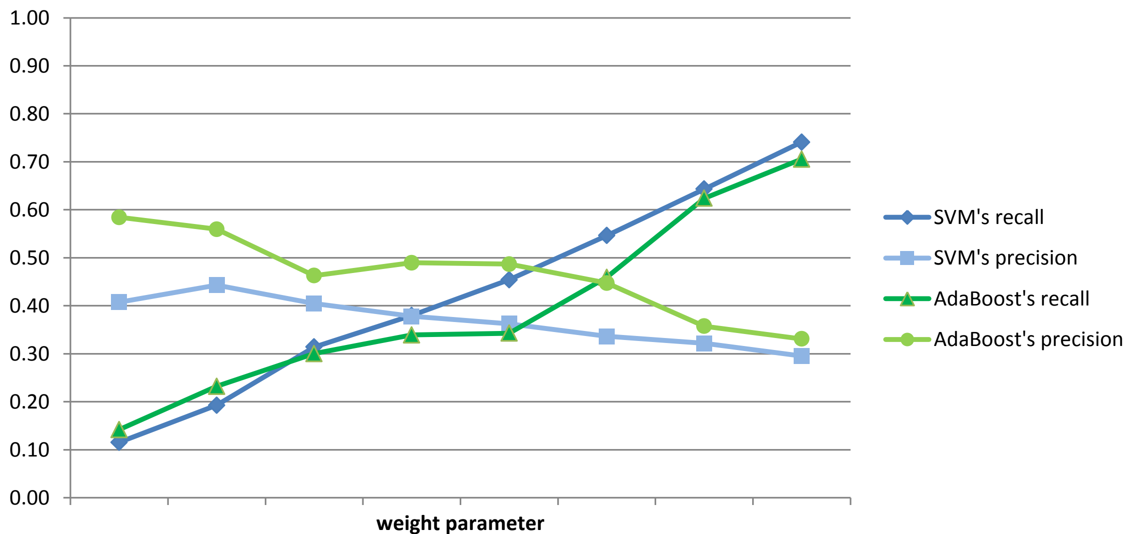

Figure 2.5. Drop duration prediction accuracy as a function of weight parameter for SVM and AdaBoost. Higher weight means false negatives cost more, relative to false positives.

the classifier is able to predict correctly. Precision is the fraction of actual short drops to the number of drops predicted to be short. AdaBoost performs slightly better than SVM, achieving both recall and precision of roughly 45%. We found that the metric that most correlates with short drops is download throughput, with high throughput correlated with short drops.

2.4 Related Work

There are several proposals for managing faults in cellular networks. However, much of the work focuses on detection of failures [4–7], and identification of root cause of the failure [8,9]. Their goal is different from ours, which is to predict failures and proactively try to prevent them, or lessen the effects of failures on user experience in the short term.

Javed et al. propose a machine learning framework for predicting handovers based on data available at the handset [10]. The goal is to notify applications before a short-

term disruption in the service due to the handovers so that they can modify their behavior to counter it. This is similar in spirit to our work, although we are not limiting ourselves to handovers. Furthermore, the wealth of data collected at the network provides much more information than data available at the handset. This enables us to predict events that could not have been predicted otherwise.

2.5 Conclusions and Challenges

This chapter presents our vision of how big data analytics can be used in real time to improve dependability and user experience. We demonstrated this by analyzing real cellular traffic data from a major cellular network and showed that we can construct a model that predicts drops and drop duration in real time with enough accuracy to enable mitigation actions to be used.

Challenges There are several challenges that need to be solved before such real-time data analysis and mitigation actions within the domain of cellular network become feasible:

Real-time Data Access: Real-time data access is not available for the majority of events logged by cellular network due to reasons including storage requirements and the need to protect user privacy.

Data Volume: Due to the amount of data and number of users involved, such real-time data analysis requires efficient data streaming and processing systems close to the data source.

Lack of unified framework: Because many mitigation actions are application-specific, and some must be initiated by the device, there must be a standard way for the network to send notifications to the device, and for applications to express interests in receiving such notifications.

3 TANGO: TOWARD A MORE RELIABLE MOBILE STREAMING THROUGH COOPERATION BETWEEN CELLULAR NETWORK AND MOBILE DEVICES

3.1 Introduction

Multimedia streaming has become a major application for mobile users. Cisco reports that mobile video traffic accounted for 55% of total mobile traffic in 2014 [11]. In 2012, 70% of Pandora's usage was from mobile devices [12]. However, the growth of mobile network capacity is not keeping pace with the growth of traffic. Cisco estimates that mobile traffic will increase 10-fold between 2014 and 2019. On the other hand, the observed growth in network capacity has been much slower, at 4-fold per 5 year on average [13]. Therefore, network congestion will be more and more common in the future, and mobile applications will need better mechanisms to mitigate the problem.

Streaming applications usually limit the download rate by setting a maximum buffer size and/or limiting the download rate. A buffer that is too large would result in unnecessary energy and bandwidth usage due to unpredictable user abandonment. A buffer that is too small would result in interrupted playback in the event of short-term network connectivity problems. Existing systems such as adaptive bit-rate schemes use a "static" buffer and try to keep the buffer full by adapting the bit rate according to current network conditions. In this work, we argue that the static buffer approach has fundamental limitations and is not adequate to address network connectivity problems. If the network condition is good (the common case), the buffer should be small. If the network condition is bad, the buffer should be large—directly proportional to the duration for which the network is going to be in the degraded state. However, the mobile device alone has no way of predicting future network quality. The cellular network, on the other hand, has a global view of the network. It knows which areas in the potential path of the streaming user are currently having network

issues. Thus, for a smooth streaming, we need a buffer of dynamic size. To support dynamic buffer, a cooperation between the mobile device and the cellular network is needed. We propose a novel idea—building a cooperation framework to support dynamic playback buffer for improving the performance of streaming applications.

Specifically, we design TANGO, a service that performs real-time data analysis in order to give streaming applications an early notification of impending connectivity degradation, so that the applications can initiate a mitigation action, such as to pre-cache more content. In effect, we turn on its head two operational principles today. First, we enable cooperation between the cellular network and the mobile device through our framework, while current practice does not allow for such cooperation and the mobile device considers the cellular network to be a "dumb pipe". Second, we use this cooperation to allow proactive handling of network condition changes, while the overriding operation mode today is through adaptive bit-rate streaming which reacts to the changing network condition in the hope that the condition will change slowly. Network conditions do change quickly in many scenarios, such as, flash crowds or changing spots with poor signal strength.

An important component of TANGO is user location prediction. Mobility prediction has been studied in several works, with satisfactory results [14–16]. We implement and evaluate the system using a simple mobility prediction algorithm, based on current and previously connected cell sectors, since this information is readily available, without incurring additional cost, say for GPS measurements. More sophisticated location predictors can be plugged in and potentially result in greater benefits.

We design and implement the two primary components of the system—data analysis and event notification (part of TANGO), and the attendant mitigation action (part of the application). The data analysis considers real data collected by a cellular service provider at the edge elements of its network, the Radio Network Controller (RNC). Information contained in the traces includes actual download/upload rate

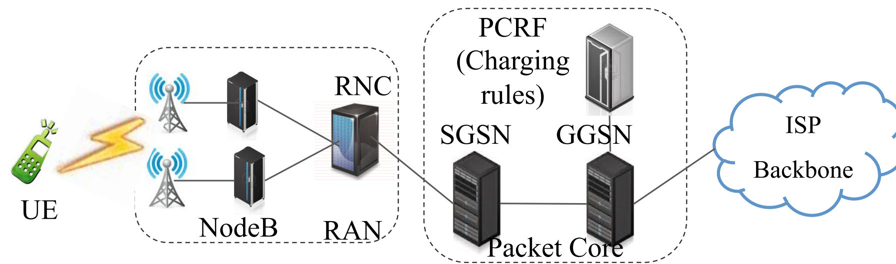

Figure 3.1. Architecture of a UMTS data network.

and current cell sector. This data is finely granular and comprises per mobile device data.

We evaluate TANGO using simulations, using the aforementioned data traces as well as behavior of a real streaming application as the basis. The application used is Pandora, which provides an online radio service, as well a popular javascript-based video streaming client. Effects of stalls are quantified and measured as pause time. We compare TANGO with the baseline approach and DASH [17]. In simulations with audio streaming users, we found that pre-caching reduces pause time by 13–72%, outperforming DASH by a significant margin, while maintaining higher stream quality.

The remainder of this chapter is organized as follows. Section 3.2 provides background on cellular network. Section 3.3 presents the pre-caching service. Section 3.4 presents the experimental evaluation. We follow this with related work and then conclude the chapter.

3.2 Background

In this section, we first describe the basics of 3GPP cellular architecture and then provide information about the data set used in our evaluations. The description here applies to 3G/High-Speed Packet Access (HSPA) network, and with some slight modifications to 4G LTE networks as well.

Figure 3.1 shows the key components of a typical UMTS data network. It consists of 2 major components: the Radio Access Network (RAN) and the Core Network (CN) (or Packet Core). The mobile device, called User Equipment or UE in UMTS terminology, is connected to one or several cell sectors (henceforth referred to as cells) in RAN. This set of cells is called the active set. Only one cell in the active set is actively used for communication at a time. This cell is referred to as the primary cell. A physical base station (called NodeB in 3G and eNodeB in LTE) can have multiple cells, which provide radio resources to UEs for wireless communications. Cellular data traffic from several NodeBs are then passed to the Radio Network Controller (RNC), which manages handovers, and scheduling of wireless resources among the NodeBs under its control. The RNCs connect to Serving GPRS Support Nodes (SGSNs) at the core network. The SGSNs are connected to the external networks, such as the Internet, via Gateway GPRS Support Nodes (GGSNs). When a UE connects to the network, it establishes a Packet Data Protocol (PDP) context which facilitates tunneling of IP traffic from the UE to the peering GGSN using GPRS Tunneling Protocol (GTP) (see [1] for details of the UMTS network).

We evaluate TANGO using real-world cellular traffic data collected at the RNC in a 3G RAN from a tier-1 cellular network carrier. All device and user identifiers are anonymized for our analysis. Details of the dataset are provided in Section 3.4.

3.3 Data Pre-caching Service

In this section we describe the data pre-caching service, which notifies applications when connectivity is predicted to be poor in the near future, based on the user's mobility pattern and current load in the network.

3.3.1 Overview

A common problem in cellular networks, especially in dense regions is network congestion. A cell's available wireless bandwidth is limited and is shared among

users connected to that cell. When the total bandwidth demand exceeds the total available bandwidth, we call that cell *congested*. Adding more capacity in the form of more cell towers, or more number of cells in an existing cell tower, or increasing the transmit/receive power of some antennas are all used today to relieve congestion. However, it is no wonder that these do not completely solve the problem, as seen in the growth of mobile traffic outpacing the growth of network capacity [11, 13].

In addition to congestion, some cells provide persistently lower data rates compared to other cells. This could be due to many reasons such as misconfiguration of antennas, a nearby source of interference, buildings or landscape obstructing the radio signals, or partial hardware failure.

Rather than trying to prevent congestion or the cause of the inferior data rate itself, this service aims to notify the application shortly before the user enters the congested area. Applications can take advantage of this information and respond in various ways—e.g., by pre-caching content that the user is expected to use in the near future (this is the mitigation strategy that we experiment with) or switching to a different carrier or base station within the same carrier. Examples of applications that can use pre-caching as the mitigation action include audio/video streaming (which we use in our experimental evaluation), GPS navigation, and web browsing. These applications have a common property that the future content is known or predictable, to varying extents, and can be downloaded at any time, but in order to conserve bandwidth and/or energy, today's practice is to not predownload content too far in advance. This usual mode of operation works well when network interruptions, if any, are short. However, when the user enters a congested cellular area or a cell with inferior data rate, the amount of buffered content may not be enough to provide uninterrupted user experience. This is known from prior reports, e.g., [18, 19], and we also empirically see this in our experiment where the pause time exceeds 2% of media stream time for a generous fixed buffer size of 10 songs (Figure 3.3).

Instead of relying on such one-size-fits-all strategy, this service enables applications to utilize different strategies in different network conditions. In general, the pre-

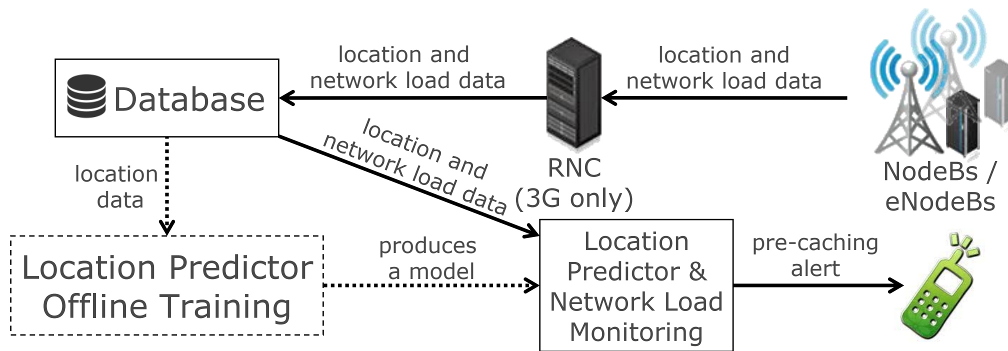

Figure 3.2. Overview of pre-caching service in TANGO using network elements that are present in 3G or LTE networks. Dashed lines and dashed boxes represent workflow that only occur during offline training.

caching service can be used whenever a connectivity degradation can be predicted to occur in the near term, such as with lookaheads of a few minutes. This can include situations where the user is entering a tunnel or moving out of the coverage area of her cellular carrier, for example.

In order to provide this service, TANGO needs to be able to predict user location in the near-term future and know the areas that are currently congested as well as cells identified as providing inferior data rates. Congestion information requires real-time network data, as congestion changes dynamically. As we will show in the Section 3.4.2, the set of cells providing inferior data rates also change over time. Therefore, the service will require real-time network data from network elements.

Figure 3.2 shows the overview of the service. During network operation, device location and network load data are collected by the network elements then stored in a database. During offline training, location data from all users in the database is used to train the location prediction model. In the online phase, the application first registers with the service. Then, the device's past and current location is used to continuously predict future location. If a predicted location is currently congested, a pre-caching alert is sent to the application. The application can then decide how

much data to pre-cache based on current buffer level, battery level, etc. We discuss location prediction in the next section.

3.3.2 Location Prediction

Today, device's location information is available from currently connected cell and GPS measurements. GPS gives more fine-grained location information than current cell does. However, in the common case where GPS is not already being used, using GPS will result in significant energy overhead. Furthermore, for the pre-caching service to be useful, we only need to know coarse-grained information, i.e., which future cells the device will be connected to, since congestion is determined at the granularity of a cell. Therefore, we opt for the current cell as well as past cells as the data source for location prediction.

Both the device and the network are aware of the current primary cell to which the device is currently connected. In our case, we implemented the predictor that uses network data as input, since prediction results will need to be combined with network load data later. Specifically, location predictor's input data consists of the current primary cell as well as the history of past primary cells of the specific device whose location it is trying to predict. Because users with high mobility can move to multiple cells within a short amount of time, instead of predicting a single cell that the user will move to, the predictor estimates the probability that the user will enter a cell C within a specific amount of time, separately for each nearby cell C. As an optimization, the prediction is only needed for cells that are currently congested, or known to be providing inferior data rates.

Symbolically, the predictor works by estimating

$$P(C \in \mathbb{S} | C_t, C_{t-1}, \ldots, C_{t-m}) = \frac{Freq(C \in \mathbb{S}, C_t, C_{t-1}, \ldots, C_{t-m})}{Freq(C_t, C_{t-1}, \ldots, C_{t-m})}$$

from the training data, where C is a future cell, $\mathbb{S}$ is the set of all cells the user will enter in the next u minutes, C_t is the current cell, $C_{t-1}, \ldots, C_{t-m}$ are the history of past cells, and m is the length of cell history. $Freq(C_x, C_{x-1}, \ldots, C_1)$ denotes the

frequency (count) of any user visiting cells $C_1, C_2, \ldots, C_x$, in that order. Note that this prediction is about whether the user will enter the cell C in the next u minutes, and not about whether C will be the next cell, and is thus an easier problem to solve. During training, each observed cell sequence of length $m + 1$ (for the denominator) and $m + 2$ (for the numerator) results in the frequency corresponding to that cell sequence increasing by one. The model's parameters m and u can be set by the user.

Intuitively, given past trajectory, the predictor answers the query "will the user enter cell C in the next u minutes?" This is done by keeping count of how many times a user enters cell C within u minutes after that exact trajectory. The counts are kept separately for each trajectory. Note that the time component of the past trajectory is not considered by the model. This algorithm is extremely simple, and is not designed to improve the state of the art. However, more complex algorithms based on similar input data can be readily plugged in, and the service will potentially benefit from improved accuracy. This benefit is quantified in Section 3.4.2.

As new base stations are added and old ones removed from the network, the location predictor will need to be retrained. Changes in traveling patterns can also cause degraded prediction accuracy. The prediction accuracy should be monitored and another offline training can be triggered once the accuracy falls below a set threshold.

3.3.3 Deployment

In this section we discuss the various components that are needed to provide the service, and how they can be implemented in a real cellular network.

The service runs on dedicated servers separate from the existing cellular network components, so as not to impede the time-critical cellular network operations. These servers need access to real-time data about the device location and network load. In 3G networks, network data from base stations is aggregated at the RNCs, where one RNC serves a set of geographically close by base stations. Therefore, it is natural to

place the database server at each RNC, in order to reduce communication latency. In 4G LTE, there are no RNCs; instead, data are collected by the eNodeB's. In this case, the real-time data from the eNodeB's needs to be sent directly to the database server. Since cellular networks can cover a large area, often a whole country, there should be multiple database servers partitioned by location, in order to distribute the load and reduce latency. Each database server is then responsible for storing data corresponding to network operations around it. Since in the online phase the service needs to access data stream as it is generated, the data needs to be *pushed* from the database server rather than having the service *pull* the data as in traditional database systems. For this task we need a streaming data management system, which can run a continuous query on data in motion until the query is explicitly uninstalled.

Having each server provide service for a specific area means that the device needs a way to select the correct server based on its current location. This can be done by having a central “directory server” which directs the device to the correct server each time it uses the service. This directory information can be cached in the device for later use in order to reduce the amount of communication as well as the load on the directory server.

In Section 3.4, we measure the amount of additional load on the network due to a service in terms of the amount of data involved and the amount of computation needed.

3.4 Evaluation

We evaluate the effectiveness of the data pre-caching service based on real cellular traces from the 3G network of a major US-based cellular network service provider.

3.4.1 Data Source

The traces used for all experiments were collected at an RNC of the service provider over the period from February 2013 to November 2014. All device and

subscriber identifiers from these traces were anonymized. The dataset, with the device ID, timestamp, and event type as the unique identifier (primary key), contains various performance data and events in the tabular format, with each type of data having its own set of metrics as the columns in addition to the common metrics. The common metrics are the device ID, timestamp, and cells that the device is connected to. Examples of recorded data include download and upload throughput and uplink power, reported at 2 seconds interval, and event-based data items such as, connections and disconnections, handover events, as well as other low-level radio protocol events. There is a second kind of data, which is specific to a cell, and for that the cell ID, timestamp, and event type form the primary key. An example of this type of data is the number of active devices in the cell.

3.4.2 Experiments

We designed simulation experiments mimicking various real-world scenarios to evaluate the costs and benefits of data pre-caching in an audio streaming application. The prediction performance as well as overhead of the location predictor is evaluated. Finally, we investigate cells with chronically poor connectivity present in the traces.

Reducing stalls in audio streaming

This experiment aims to evaluate if the pre-caching of TANGO helps reduce stalls in an audio streaming application. This would effectively translate to increased capacity of the network for handling audio streaming users, while keeping the quality level the same. The cutoff is provided by the buffering time as a percentage of the total playback time, which for an acceptable quality of use, needs to be kept below a threshold (typically 1-5%, depending on kind of audio and kind of users). We picked streaming audio because online radio services are becoming increasingly popular with Pandora and Spotify having become household names in the US.

We measure the buffering time by running a simulation, with the trace data coming from a real cellular network. An area in the real network is selected, and the cells in the area are simulated. A varying number of emulated audio streaming clients add traffic in addition to existing background traffic with simulated congestions, and the audio pause time is measured. Audio pause time is defined as the total listening duration (say, A), minus the duration where audio is actually played (say, B), minus the initial buffering time (say, C). The quantity $A - B$ gives how long the user has spent waiting for the player to buffer content, thus a smaller value is better for user experience. The quantity C is subtracted from that because that is the initial buffering when the user initiates the request for the audio stream. Presumably, this delay is less annoying to the user than a delay in the midst of listening.

We selected an area in downtown San Francisco, USA to be the simulation area. The area is 1.66 miles long along north-south and 1.91 miles long along east-west. All cells in the area (>1000) are included. Each cell is simulated by having a fixed capacity, equal to the maximum bandwidth observed in the trace. Existing background traffic as well as users' mobility patterns are taken from the trace between 4-5pm on 17 November, 2014.

The audio streaming users are emulated by mimicking the action of the mobile Pandora app without actually transmitting any data or playing the songs. We experimentally found that Pandora keeps one song in the buffer, in addition to the current song, at all times. When the playback of the current song finishes, it downloads one more song as fast as possible, resulting in a bandwidth spike of a few seconds, and no bandwidth usage afterwards. For each client, the emulator keeps track of how much data has been downloaded for the current song, what the current playback position is, its location (current cell), as well as the audio pause time, and updates this information every second. In place of real traffic, the emulator sends an estimate of how much data the client would request, to the cell proxies. Each cell proxy then sums up the bandwidth demand, and distributes the available bandwidth (after subtracting

the background traffic from the total capacity) among the clients proportional to their demands.

In each experiment, six approaches are compared: 1) baseline, 2) DASH, 3) TANGO w/o bit-rate adaptation, 4) TANGO w/o bit-rate adaptation w/ perfect location predictor, 5) TANGO, and 6) TANGO with perfect location predictor. The same default buffer size is used for all six approaches. In baseline, the client always tries to keep the buffer full at all times, and no additional mechanism for reducing stalls is used. In DASH, the future bandwidth is predicted as the harmonic mean of download speed during the last 10 seconds, as recommended by Jiang et al. [20]. Three bit-rates are available: 128 Kbps (default), 64 Kbps, and 32 Kbps. The client always pick the highest bit-rate that uses no more than the predicted bandwidth. When predicted bandwidth is lower than 32 Kbps, the lowest bit-rate of 32 Kbps will be chosen. In TANGO, actual location prediction and network load monitoring happen in the same way it would be in the real system. TANGO includes the same bit-rate adaptation mechanism used by DASH. For comparison, we also included variants of TANGO without bit-rate adaptation as well as variants with perfect location predictor. In variants with perfect location predictor, the location predictor is replaced by one that always makes correct predictions. This helps quantify the effect the location predictor's accuracy has on the overall benefits of the system.

With TANGO, the location predictor continuously predicts each user's location and monitors cells in the area for congestion. As part of TANGO's input, cells need to be designated as congested or not congested. The thresholds for a cell to be considered congested are parameters that can be set by the user. We empirically found that optimal results are obtained when we consider cells where at least 85% of its capacity has been used for the last 30 seconds to be congested. Recall that TANGO works by sending a pre-caching alert to the client when a user is predicted to be moving to a congested cell in the next u minutes. The optimal value for u was found to be 5 minutes. The client emulator pre-caches content by increasing the buffer size from one song to B minutes and download the stream at he default 128

Table 3.1.
Different types of simulated congestions used in the experiments.

Type	Description
Mobile flash crowds	50 Congestions that move across cells like a user does. Each comes and goes randomly.
Static congestions	Congestions in 2% of the cells that lasts through the whole experiment.
Random congestions	Congestions in 50% of the cells that come randomly and last 0–20 minutes.

Kbps bit-rate as quickly as possible. The default bit-rate is used in order to prevent degradation of user experience. The buffer size controls the tradeoff between audio pause time and bandwidth usage. In our case, we consider lower audio pause time to be more important than saving bandwidth, so we choose a larger buffer size of $B = 30$ minutes.

Each emulated client follows the movement pattern of a real user from the trace. For these experiments, we only simulate users with some movement between cells, since TANGO will not work for stationary users. Each client's arrival time is uniformly random within the length of the experiment of 1 hour. On average, the simulated users move between cells every 145 seconds, so we believe the chosen length of experiment is sufficient. The session length follows the Weibull distribution with $\lambda = 13$ minutes and $k = 0.52$, as found by Zhang et al. to be the best fit of the session lengths of mobile online radio users [21]. This corresponds to the average session length of 24.25 minutes. The relevant characteristics of the songs in the audio stream are the song length and the bit rate. The lengths of the songs come from Spotify's top 50 chart for the week of November 2, 2014 [22].

We mimic congestion in three ways: 1) by adding mobile "flash crowds" to the background traffic, 2) by simulating static congestions in a small fraction of the cells, and 3) by simulating random congestions in a larger fraction of the cells, as shown in Table 3.1.

Each simulated flash crowd follows the movement pattern of a real user randomly picked from the trace. 50 flash crowds are simulated. Their arrival time is uniformly random, and they last as long as the original user stays active. In effect, these flash crowds randomly come and go. While a flash crowd is in a cell, all users connected to that cell cannot download any data. Overall, this causes 1.5% of the cells' operating time to be congested, when summed across all the cells and across all the time points and divided by the total number of time points in the one hour trace.

Static congestions are simulated by randomly picking 20% of all cells as congested. These congested cells do not provide any data to the users connected to them. This effect lasts through the entire 1-hour-long experiment.

Random congestions are similar to static congestions, but the start time and duration are random. This affects 50% of the cells. The start time is picked uniformly at random. The duration ranges from 0 to 20 minutes, also picked uniformly. Overall, this cause 8.3% of the cells' operating time to be congested.

The result of the three simulations is shown in Figure 3.3. Total audio pause time is measured as a percentage of total playback time. For static congestions and random congestions, there is a critical number of audio streaming users beyond which the patterns change. This critical point, at roughly 7,000 users for static congestions, and 7,500 users for random congestions, corresponds to the point where the capacity of most of the network is reached. For flash crowds, the effect is smoother, so there is no clear-cut critical point, although the general trend is similar. Before the critical point, bit-rate adaptation in both DASH and TANGO barely provides any benefit. This is because there is not much opportunity to use a lower bit-rate, as congestions are sudden, and congested cells provide zero bandwidth. However, TANGO significantly reduces audio pause time compared to DASH and baseline, with TANGO variants with perfect location predictor having a slight edge over their counterpart with real (imperfect) location predictor. We believe the reason perfect location predictor only improves the results slightly is because a false positive can help mask a false negative later, since the pre-caching buffer size is 30 minutes and users move between cells

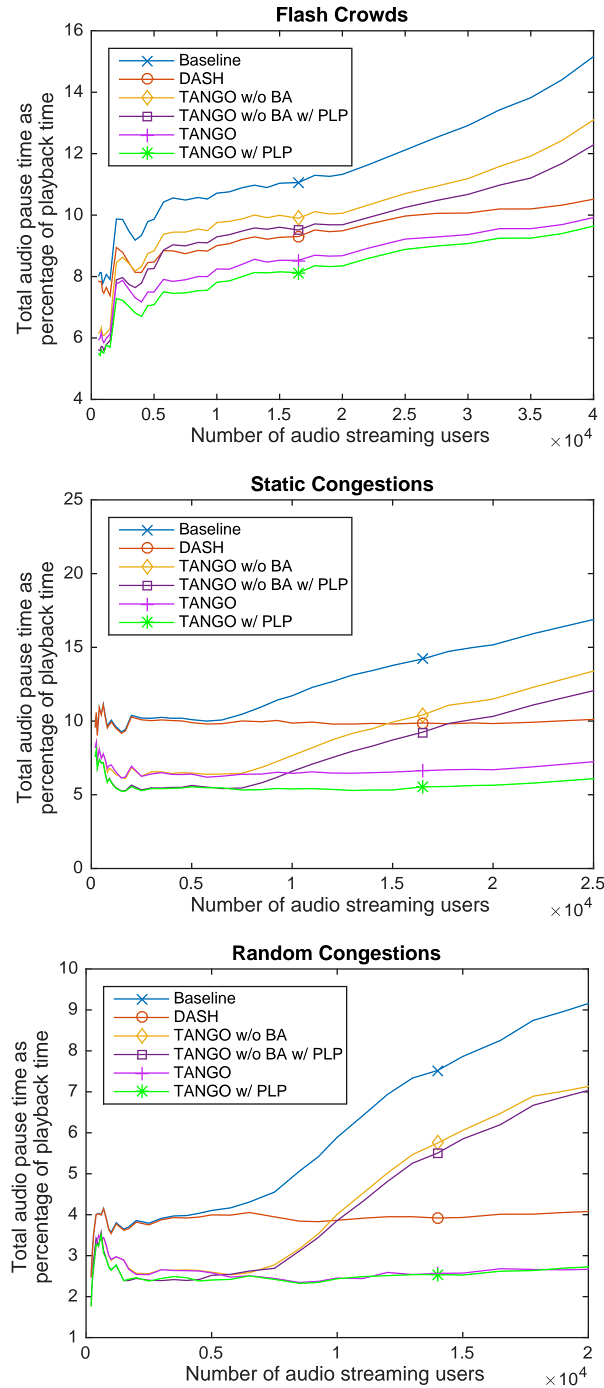

Figure 3.3. Audio pause time (shown as percentage of total playback duration) as a function of the number of audio streaming users, with three types of simulated congestions. The pre-caching due to TANGO allows the network to accommodate significantly more audio streaming users before the disruption due to buffering becomes unacceptable.

every 2.4 minutes on average. In addition, TANGO does not predict an uncongested cell will become congested. Therefore, when congestions come and go often, even the perfect location predictor will miss opportunities for pre-caching.

After the critical point, however, bit-rate adaptation starts to provide significant benefit. Audio pause time is kept nearly constant while more users are added even after the capacity of the network is reached. This is only possible because the quality of the stream decreases proportionally. TANGO variants without bit-rate adaptation perform better than DASH at lower number of audio streaming users, but quickly performs worse as the number of users increases. TANGO with bit-rate adaptation is able to keep the significant advantage over DASH even with high number of users.

While reducing stalls is important, doing so while significantly lowering the stream's bit-rate is undesirable. We measured the average stream's bit-rate across all users in the simulations, shown in Figure 3.4. Approaches that do not include bit-rate adaptation always keep the bit-rate constant at 128 Kbps. The general trend is that, the higher the number of users, the lower the average bit-rate. The lower average bit-rate at the lower end of the number of users is likely due to high variance. TANGO's average bit-rate is significantly higher than DASH in all scenarios, at roughly the midpoint between the highest bit-rate of 128 Kbps and DASH's average bit-rate. Looking at the audio pause time together with average bit-rate, we can see that TANGO significantly reduces audio pause time, while maintaining higher stream quality than DASH.

For mobile users, higher bandwidth usage results in more energy consumed, as well as higher cost of service in most cases. Therefore, it is important to keep the amount of wasted bandwidth small. Wasted bandwidth is a result of having a non-empty buffer at the end of the user's listening session. We measured the average wasted bandwidth across all users in our experiment, shown in Figure 3.5.

Both baseline and DASH have a fixed buffer size of one song, so the wasted bandwidth is almost constant. At higher number of audio streaming users the wasted bandwidth decreases slightly due to the fact that some users do not have enough

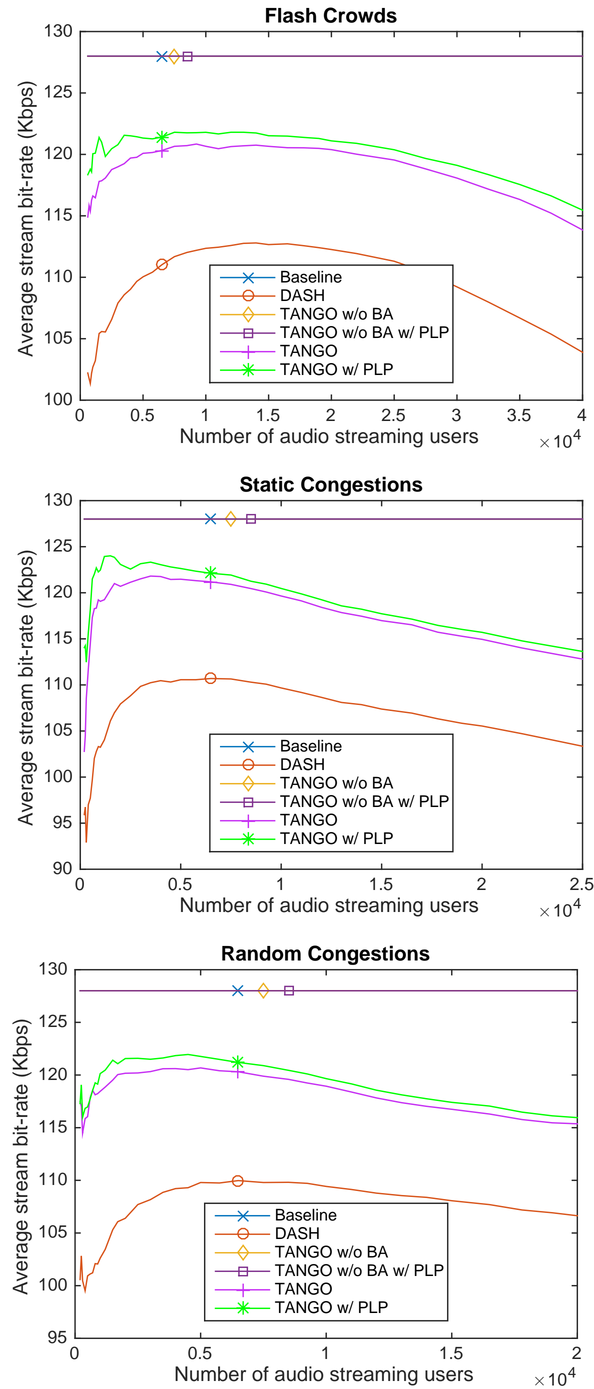

Figure 3.4. Average audio stream quality as a function of the number of audio streaming users, with three types of simulated congestions. BA refers to bit-rate adaptation, while PLP refers to perfect location predictor. Approaches that do not include bit-rate adaptation always use the default bit-rate of 128 Kbps.

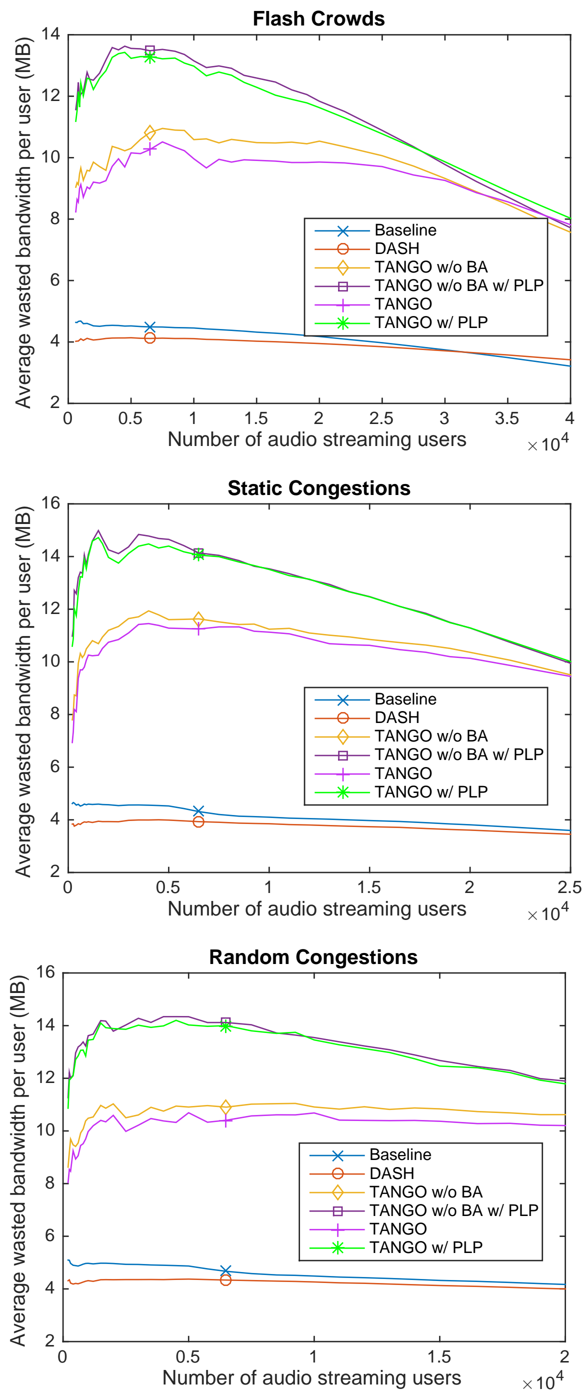

Figure 3.5. Average wasted bandwidth per user due to user abandonment (i.e., end of session) for different approaches as a function of the number of audio streaming users. BA refers to bit-rate adaptation, while PLP refers to perfect location predictor.

bandwidth to keep the buffer full as well as the lower bit-rate for DASH. In TANGO, the buffer size changes dynamically depending on the predicted connectivity for each user. TANGO produces 2.5–3x the amount of wasted bandwidth compared to baseline and DASH. Bit-rate adaptation only has a small effect on wasted bandwidth. TANGO with perfect location predictor produces more wasted bandwidth than TANGO. Overall, the increase in wasted bandwidth is proportional to the opportunity to pre-cache, which results in reduction of audio pause time. While using TANGO results in more wasted bandwidth, we believe this is a worthy tradeoff for less playback disruption. If desired, lower wasted bandwidth can be achieved by reducing the pre-caching buffer size, at the cost of more playback disruption.

Next, we quantify the effects of varying the default buffer size for each approach in the presence of congestions. Here we fix the number of audio streaming users at 10,000, and vary the default buffer size. Note that TANGO has a separate pre-caching buffer size, which is fixed at 30 minutes. The results are shown in Figure 3.6. Buffer sizes are specified in terms of the number of songs, in addition to the current song, which is always buffered in its entirety.

The general trend in all case is that larger buffer results in lower pause time. However, it is clear that when the buffer is large, the pause time only depends on whether bit-rate adaptation is used. This is likely because as buffer becomes larger, overall bandwidth usage increases, and the capacity of the network is reached. Without bit-rate adaptation, the bandwidth required to sustain smooth playback is much higher, so playback is disrupted for users who cannot get the required bandwidth. Nevertheless, such large buffers are not practical, due to the large amount of wasted bandwidth which is inevitable at the end of the user's session. With small buffer, TANGO always perform significantly better than the respective baseline approach (e.g., DASH vs. TANGO). With perfect location predictor, the benefit is larger in flash crowds and static congestions, and almost the same as real location predictor in random congestions.

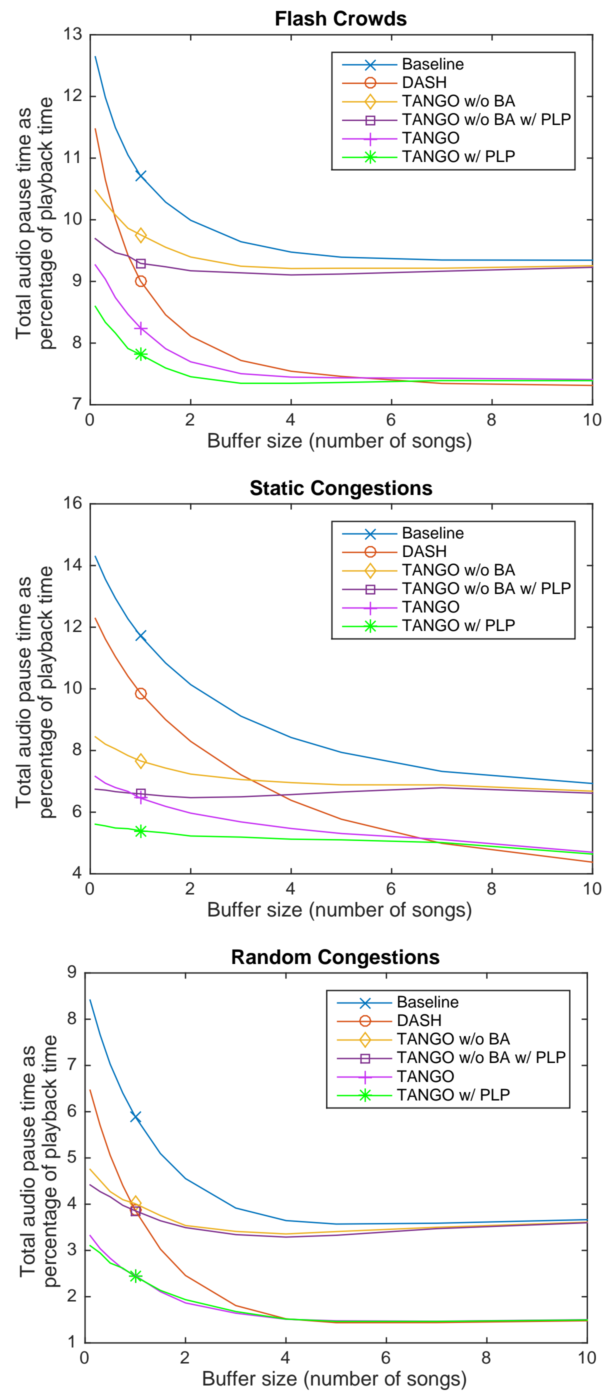

Figure 3.6. Audio pause time (shown as percentage of total playback duration) as a function of buffer size (as number of songs), with three types of simulated congestions. BA refers to bitrate adaptation, while PLP refers to perfect location predictor.

These results show that without cooperation from the cellular network, an audio streaming player needs to pick a buffer size that gives the best trade-off between low pause time in the presence of congestions (bigger buffer is better), and wasting of energy and bandwidth when the user ends the session. With TANGO, the default buffer size can be kept low, and bigger buffer is used only when an impending connection degradation is predicted.

We find experimentally that when 5,000 audio streaming users are emulated, 64% of the cells are never visited by any emulated user. Of the remaining 36%, the average number of emulated users per cell is 2.03. Thus, there is a non-uniform distribution of users among the cells according to the mobility traces.

In these audio streaming experiments, there are three important parameters that influence when and how pre-caching is done, namely congestion capacity threshold, congestion duration threshold, and pre-caching buffer size. Here, we investigate how each of the parameters impacts the results.

The results are shown in Figure 3.7. The number of audio streaming users is fixed at 5,000. For all three parameters, the audio pause time is not highly sensitive to their values. As long as a reasonable value is chosen, the pause time will be close to optimal. Note that the optimal values for these parameters are not the same as the optimal values we use in the audio streaming experiments. This is because the optimal value is slightly different for different number of audio streaming users. For those experiments, we choose the values that produce the best overall results.

Idealized benefit to a single video streaming user

In the previous set of experiments, we showed the benefits of TANGO with audio streaming. In this experiment, we show that TANGO can be equally useful for video streaming applications by reducing the video pause time. The experiment setup consists of a single client moving from uncongested cell to congested cell while playing video that is hosted on an HTTP server. Since there is only one client, we run

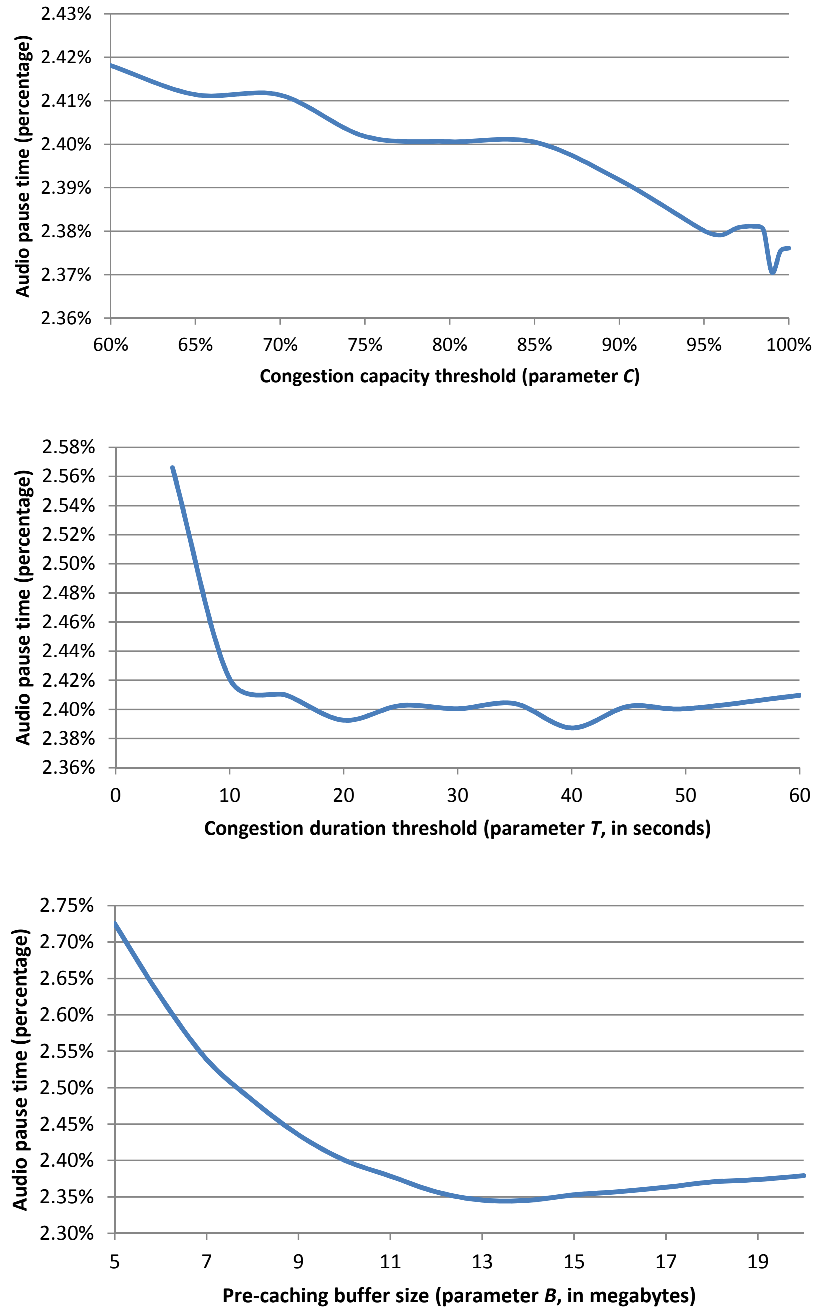

Figure 3.7. Parameter sensitivity of pre-caching. The number of emulated users is fixed at 5,000.

the experiment using a real video streaming application on an Android smartphone instead of relying simulations. The application we use is `dash.js` [23] which is a popular javascript-based implementation of MPEG-DASH standard [17]. We chose this player as this is open-source and is actively backed by many leading industry content providers [24]. We modified the player to include pre-caching that gets triggered whenever a congestion alert is sent to the player. The cells are simulated using an HTTP proxy. The proxy's bandwidth is controlled at the server side using the `tc` tool in Linux. Simulated congested cells have bandwidth capacity of 100 Kbps.

In our simulated situation, the player streams a long video that is obtained from the DASH dataset published by Lederer et al. [25]. The user starts in a non-congested cell and moves to a congested cell after a varied amount of time from 10 seconds to 2 minutes. The time spent in congested cell is fixed at 10 minutes. Thus, this experiment captures an idealized benefit from pre-caching, as a function of lead time.

We compare four approaches in this experiment: 1) baseline, 2) DASH, 3) TANGO w/o bit-rate adaptation, and 4) TANGO. The default buffer size is fixed at 30 seconds for all approaches. In baseline, the video client does not implement bitrate adaptation. In DASH, we follow a typical DASH setting with the video available at four different bit-rates: 220, 440, 895, and 1,340 Kbps. In both variants of TANGO, the client will start pre-caching from the beginning of the experiment. The bit-rate adaptation mechanism in TANGO is exactly the same as in DASH.

Figure 3.8 shows the video pause time as experienced by the client for the four approaches after spending a varied amount of time in the uncongested cell. We can see that both baseline and DASH clients are unable to mitigate the pause time during congestion regardless of time spent in uncongested cell because of the static buffer size. With or without TANGO, bit-rate adaptation helps reduce pause time by lowering the bit-rate while the user is in the congested cell. When lead time is 2 minutes, TANGO is able to pre-cache enough content to last through 10 minutes of congestion. TANGO, which combines both pre-caching and adaptation, is most versatile in reducing the pause time as it can prefetch video content when bandwidth

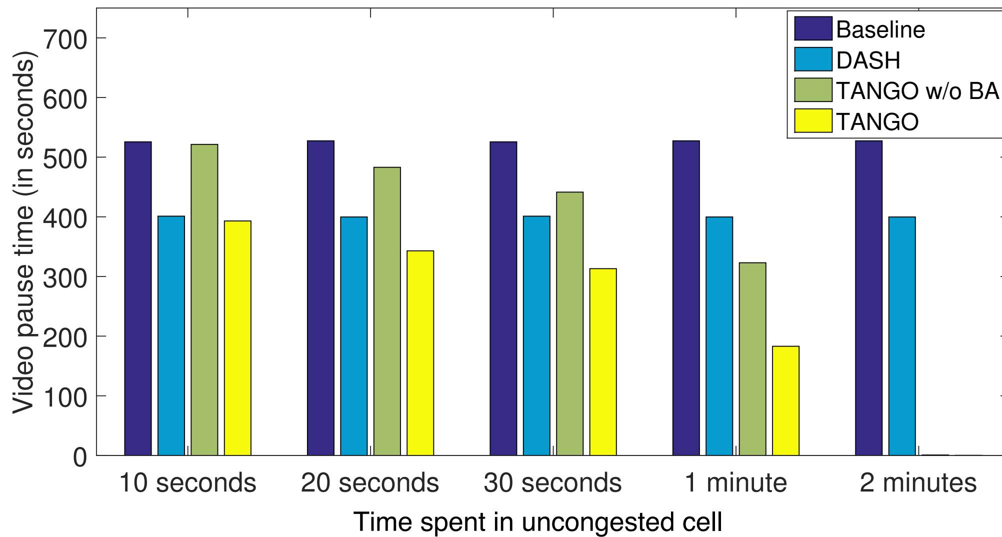

Figure 3.8. Video pause time in the simple case where the user is watching one long video and move from a uncongested cell to a congested cell.

is high and also download more video during congestion by reducing the quality which is needed when lead time is small.

Location Prediction

Section 3.3 describes how the location predictor works. It predicts all the cells that the user will move into in the next 5 minutes, based on the current cell and previously visited cells. There are two important parameters that are known to affect the accuracy of location prediction: the length of history and the probability threshold for generating an alert. This is a micro experiment to investigate how these two parameters affect our location prediction accuracy.

The predictor is trained using data from October 1–20, 2014, and tested on data from October 21–31, 2014. The results for different history lengths are shown in Figure 3.9. Here, the alert threshold is fixed at 0.15 (if the probability of moving to a congested cell is greater than the alert threshold, then pre-caching is initiated), and the prediction is for the next 5 minutes. Precision is the proportion of all predictions

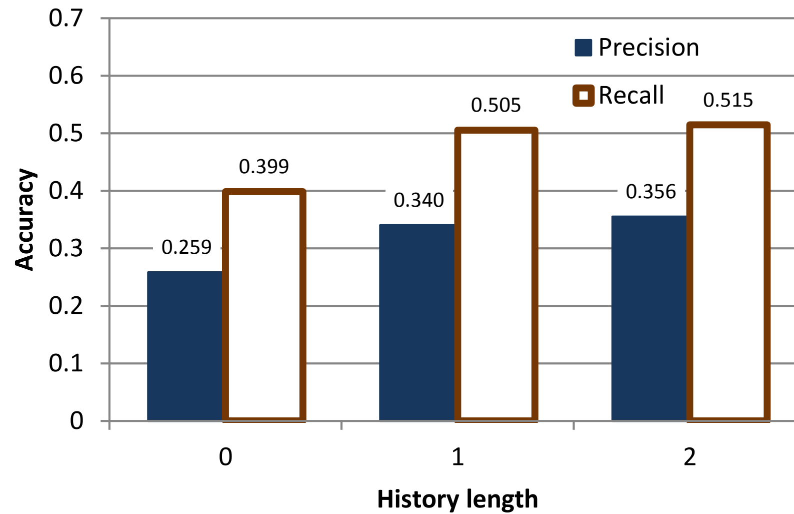

Figure 3.9. Location prediction accuracy with varying history length. History length of 0 indicates that only the current primary cell is used.

the classifier makes that says the user will move from the current cell to another cell C within the next 5 minutes that are correct predictions. Recall is the proportion of actual user movement to another cell that was predicted correctly. The results show that using only the current primary cell to predict future cells is not adequate. Between history length of 1 and 2, however, the accuracy difference is small. As the model stores counts of how many times a user visits cells $C_1, C_2, \ldots, C_{m+1}$ (where m is the history length) separately for each sequence of cells, the number of parameters needed to be estimated increases exponentially with the history length. Thus, simpler models are generally preferred, so we choose the history length of 1 for our experiments.

We also investigate the effect the alert threshold has on prediction accuracy. For this, we fix the history length to 1. The results are shown in Figure 3.10. From the results, selecting the threshold would involve a tradeoff between precision and recall. Therefore, we need to compare the relative cost of false alarm and false negative. A false alarm leads to unnecessary pre-caching, which may lead to wasted bandwidth at the end of a user's listening session. False negative leads to disruption of the stream, which aggravates the user. The relative cost would depend on the user and his service

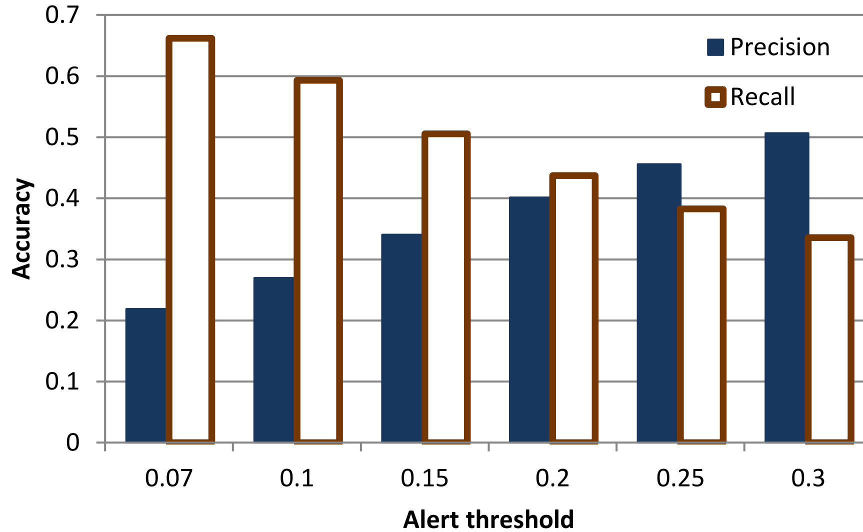

Figure 3.10. Location prediction accuracy with varying alert threshold.

plan (e.g., whether he is on an unlimited data plan). For our experiments, we value less playback disruption higher than bandwidth, and decided to use the threshold of 0.15.

Even with the optimal parameters, the prediction accuracy is still somewhat low. However, there are two specific promising avenues for improvement. The prediction algorithm that we use does not take into account *when* the past cell movement occurred. This information captures the 'speed' of the user, which will let us separate high mobility users from the low mobility users. In addition, in the data traces, there is no direct way of determining cell movements. We use three overlapping events to infer movements of a user from one cell to another — throughput (measured every 2 seconds), handover, and release of a certain radio resource, the last two being event-based. We learn anecdotally that these three put together are still not a complete data source for all cell movements.

Overhead

We measured the overhead of the location predictor with respect to the offline training and online prediction. Using a 2.26 GHz Intel Xeon machine, an individual

prediction for one user takes only 1 millisecond. On a macro level, one core of the CPU provides enough computation for location prediction for 10.3 million users, each triggering when the user moves from one cell to another. At this number of users, the size of input data stream is 667 kilobytes per second on average. This online overhead scales linearly with the number of users, and the computation can be easily distributed among servers to divide the load. Based on these numbers, a single server is more than adequate for providing location prediction to all users in one RNC.

Offline training involves processing information about all movements between cells within the training period, and therefore takes much longer. Building a model based on 20 days of operation in one RNC, which involves processing approximately 9 gigabytes of data, takes 16.7 minutes. This overhead scales linearly with the number of movements between cells in the training dataset, which is a function of the number of users and their degree of mobility. This offline training can be done in bulk periodically at a central location, or in a distributed manner at the RNCs.

Cells with Chronically Poor Connectivity

In addition to cells that are temporarily congested due to a high number of active users, data pre-caching can also be used if we can identify cells that provide chronically inferior data rates compared to other cells. We find from the traces that the occurrence of chronically underperforming cells is *more common* than transient congestion in the cells. Independent of TANGO, this is a useful characterization of a cellular network.

We identify such cells by comparing their average data rate with those of their neighboring cells, which are more likely to have similar bandwidth demands than cells further away. Specifically, we compute the average data rate in each non-overlapping 30-minute window in the past 7 days of each cell. Then, for each pair of neighboring cells, we perform the *paired t-test*, with the average data rate in a 30-minute window making up a sample. The *p-value threshold* is set to 0.05, with Bonferroni correction. The t-test tells us how likely it is for the difference to be due to chance (i.e., the two

cells have the same true distribution of average data rates). We classify cells that have lower data rates (with statistical significance) than 80% of their neighbors as cells with inferior data rate.

We analyze data from period of operation March 1 to March 31, 2014, from larger areas than the one used in earlier experiments, and found that there are 214 cells with inferior data rates, out of approximately 15,000 cells. Within those inferior cells, there are 874,181 unique active users (those in high power state, with dedicated communication channel). On average, each of these users spends 9.4 minutes (per month) in cells with inferior data rates.

Dynamic Changes to Underperforming Cells

In this section, we study how quickly the set of cells identified as providing inferior data rates change over time. The reasoning for this is that, if the cells' quality is static, then there would be no need for real-time communication between the device and the network, since the device can simply pre-download the map of quality of cells and use the location predictor based on local movement information. On the other hand, if the classification changes quickly, it would mean that the models will have to be periodically refreshed.

We analyzed data from multiple RNCs during March 2014 and found that on average, 78% of the cells identified as providing inferior data rates are still identified as such on the next day. This means that in one day, 22% of those cells are no longer providing inferior data rates (while roughly the same amount of cells are newly identified as providing inferior data rates). Thus, if not updated, the classifications will quickly become inaccurate in a matter of a few days. This supports the need for real-time communication between the device and the network, as the device cannot simply cache the database and use it for a long time.

Table 3.2.
Some key take-aways from the experimental evaluation.

Audio streaming—pause time	TANGO reduces pause time significantly compared to DASH and baseline. Bit-rate adaptation can almost nullify the effect of having more users in the network.
Audio streaming—stream quality	TANGO always maintains higher stream quality compared to DASH
Audio streaming—wasted bandwidth	TANGO increases bandwidth usage to 2.5–3x in order to pre-cache content
Audio streaming—buffer size	Larger buffer will reduce the effects of congestions more. Baseline and DASH need larger buffer in order to match TANGO's performance.
Runtime overhead	The cost of predicting location is small enough to be usable in practice
Underperforming cells	In the cellular network, 1.4% of the cells are underperforming on any one day; 78% of these will underperform the next day as well

Summary Take-aways from Evaluation

We summarize the main take-aways from the experimental evaluation in Table 3.2.

3.5 Related Work

Adaptive bit-rate streaming has been studied in many works in various aspects. Past measurement studies [26, 27] have pointed out the problems in bit-rate adaptation. Following that, different improvements to bit-rate adaptation have been suggested that adapt either using past TCP throughtput [20, 28] or buffer occupancy [29] or both [30]. TANGO extends current streaming system fundamentally and introduces the idea of buffer adaptation along with bit-rate adaptation using information from the network. An advantage with our approach is that it can be integrated easily with any existing bit-rate adaptation approach.

There are several proposals for managing faults or performance related issues that work either at the network [6, 7, 31] or the device side [32, 33] without any cooperation between them. On the network side, much of the work on fault management in cellular networks has focused on reactive measures, i.e., detection of the failure [4–7] and then identification of the root cause of the failure [8, 9]. On the device side, Balasubramanian et al. use empirical user behavior data regarding mobile web search to schedule data downloads in bursts and reduce energy consumption on the mobile device [32]. TANGO provides the flexibility to implement both reactive and proactive mechanisms (e.g., data pre-caching presented in this chapter) on top of it. Furthermore, most of the prior studies use statistical machine learning techniques with very limited amount of data [7–9]. In contrast, we use a wealth of information from a tier-1 cellular network provider in US. Also, our data set offers fine device and time level granularities for our data modeling and analysis.

The performance of data pre-caching presented in this chapter can be further improved by utilizing ideas from several prior studies on handover in cellular networks [10, 34, 35]. Javed et al. propose a machine learning framework for predicting handovers [10]. The premise is that handovers cause short-term disruption to application performance, and if an application knows in advanced that a handover is likely to occur in the near future, it can modify its behavior to counter the performance degradation.

User mobility prediction has been studied in several works. Bradley and Rashad propose a prediction technique that take into account different behaviors of users during weekday and weekend [16]. Pathirana et al. propose a prediction algorithm that uses data from GPS measurements in an environment where both users and base stations are mobile [36]. Zhang et al. use connected cells combined with call records to enhance mobility prediction [14]. These location predictors can be used in place of the simple location predictor used in TANGO, as long as the required input data is available, and prediction lookahead can be set to short term such as 5–10

minutes. Our experiments include results obtained when the location predictor has 100% accuracy as a comparison point.

3.6 Conclusions

In this chapter, we propose TANGO, a framework for the cellular network and the mobile device to cooperate to improve both network utilization and user experience. We show an instantiation of the framework's pre-caching service, a mechanism for notifying mobile streaming application before the device enters an area with bad connectivity. The streaming application can then adjusts its buffering strategy by increasing the size of the buffer and pre-downloading content into it, so that when connectivity worsens, there is more buffered content available. We evaluate TANGO using real cellular data collected at the edge network element of a major US-based cellular network service provider. Evaluation done using simulated audio streaming application shows that audio pause time is reduced by 13–72% in the presence of different types of congestions, while introducing only a slight decrease in stream's quality and reasonable amount of extra bandwidth usage. We also identify cells with chronically poor connectivity, which can potentially be used as a trigger for data pre-caching, and see that there is a fair amount of churn in these cells. As ongoing work, we are furthering the vision of cooperation between mobile devices and the cellular network by building more services that tap into knowledge from both sides.

4 APPSTREAMER: REDUCING STORAGE REQUIREMENTS OF MOBILE GAMES THROUGH PREDICTIVE STREAMING

4.1 Introduction

The amount of content that users seem to want to store on their mobile devices has been outstripping the growth of storage available on the devices. For example, a survey of the habits of 30 million consumers worldwide in early 2016 found that the amount of content stored by smartphone users has grown by a steep 55% over the past 10 months [37]. This growth is being spurred by two major factors. The first is the flowering of the app stores for both Android and iOS devices with a profusion of apps, and specifically, content-rich mobile games. Mobile games now account for 37% of the total game market (2016) and its revenues rose 21% since just the year before. These games take up large amounts of storage on our smartphones with Pokemon Go taking about 200 MB and more demanding games, like first person shooter (FPS) games taking 2-3 GB[1] [38]. The second factor is the increasingly powerful processors and graphics cards, high resolution display, and ubiquitous network connectivity on our mobile devices which makes it attractive to play these heavy games on mobile devices such as smartphones and tablets. Games are but one of the culprits, albeit a major one, for using large amounts of storage on our mobile devices; others are media files such as music, photos, and videos.

Contrast the above state of affairs with the relatively slow growth in the amount of storage available on our smartphones. The NAND flash storage technology has given us 32–64 GB of internal storage standard on smartphones today, while the smartphones from 5 years ago had 8–16 GB standard. For concreteness, consider

[1]Some examples are the 2.2 GB *Godfire: Rise of Prometheus*, the 2.5 GB *World of Tanks Blitz*, and the 3.2 GB *Final Fantasy IX*.

the Samsung Galaxy series which follows exactly the numbers given above, for the Galaxy S7 of 2016 compared to the Galaxy SIII of 2012. The NAND flash technology is seen to be reaching the end of its scaling capability [39], and vendors are predicting major advances in 3D NAND, where the basic one-transistor-per-cell architecture of flash memory is stacked into three dimensional arrays within a chip [40]. However, these are in the future and are open to the vagaries of technology development and adoption. Suffice it to say that trends indicate that our desire to store content on smartphones is going to continue to outstrip the amount of inbuilt storage capacity available on these devices.

Available solutions to storage crunch

There are two consumer-grade options available today for alleviating the storage crunch. The first is the expansion to microSD cards through an optional expansion slot on the device, though here too the limit is typically 256–512 GB, which delays the problem rather than solving it, not to mention that these high-capacity microSD cards can cost as much as a midrange mobile device itself. Furthermore, original manufacturers often prevent their built-in apps and core functionality from using the expandable storage, as do many game vendors, for copyright and distribution or performance reasons [41]. The second solution approach is the use of cloud storage. The most popular use case is the storage of photos and static content on cloud storage. For example, the Google Pixel phone has a widely advertised feature that its photos will be backed up to Google Cloud, at full resolution, with no charge to the end user. This second solution approach however is less applicable for mobile applications. If an application's files and data are stored on the cloud rather than on the phone, when the user wants to run the application, the entire application must be downloaded before it can be used. This adds unacceptable amount of delay to application startup.

An alternative approach is running the application on a cloud server, and streaming the contents of the screen to the phone as a data stream. This approach is

widely in use today for gaming and is referred to as *cloud gaming*. Expectedly, it has high bandwidth overhead, and incurs input delay due to network latency and video encoding/decoding overhead, which makes it a challenging fit for highly interactive applications such as games, especially under high-latency network such as cellular network. We quantify this overhead through experiments on our two evaluation games in Section 4.6.5.

Our Solution Approach: AppStreamer

In this chapter, we introduce the design of AppStreamer, which alleviates the storage crunch on mobile devices for applications, especially games. It does this through extending the traditional microarchitectural notions of prefetching and caching from local storage to cloud storage. In our context, the device's local storage is considered the cache, while the cloud storage is the farther-off storage which has high access latency but contains all data. In contrast to prefetching techniques used by modern operating systems, AppStreamer's prefetching is much more aggressive as cache misses can easily ruin the user experience by introducing noticeable delays. AppStreamer predicts the use of data blocks in the near future, based on file usage patterns of the application. The predicted blocks are then prefetched from the cloud storage with the goal of ensuring each data block is present on the device's storage *before* it is needed by the application. A high-level overview of AppStreamer is shown in Figure 4.1, which depicts the client mobile device and the server infrastructure as well as the offline and the online processes.

We develop the concepts in the context of interactive games, because they are typically large in size and require high interactivity and thus low latency, though the concepts as well as the implementation also apply to other types of applications on mobile devices[2]. The traditional notion of prefetching and caching have to be

[2]Note that streaming media applications, such as YouTube client, do not fall within this scope because the media content is typically streamed to memory and removed when the session ends, and thus does not consume storage on the device.

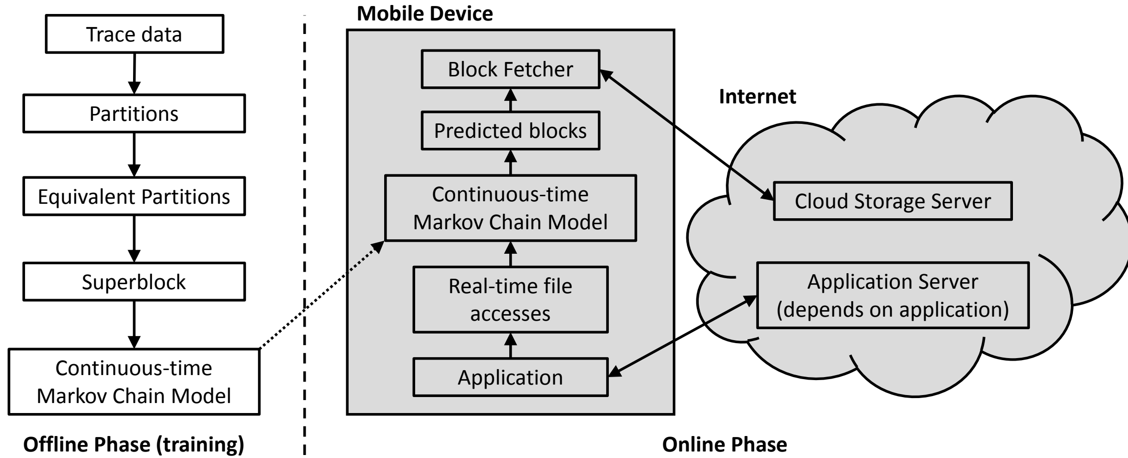

Figure 4.1. Overview diagram of AppStreamer, showing the components on the mobile device and the cloud, as well as the offline training and online phases.

adapted to the stringent requirements of interactive games. Here, constant delays of a few tens of milliseconds are considered intolerable for the gamer [42]. Empirically, we find that file blocks are accessed in granularities of tens of MB for most of the interactive games, as can be seen for the game Dead Effect 2 in Figure 4.2. Hence, AppStreamer has to use prediction with few misses and initiate the prefetching with significant lookahead, considering that typical cellular network bandwidth is low, in the range 10-20 Mbps[3].

Our solution works in a manner that is agnostic to the actual game and thus needs no semantic understanding of the game or access to the source code of the game, either on the client or the server side. AppStreamer achieves its functionality by trapping file system calls in the Android kernel and clustering the data blocks that are accessed. We find that the clustering of the blocks has to be done paying heed to the different control paths that different users can take through a game as well as different rates at which different players play a game. AppStreamer takes care of these considerations by using a modeling formulation called Continuous Time Markov Chain (CTMC) and modeling the states, which represent a cluster of data blocks, as

[3]The US average for LTE bandwidth is 13.95 Mbps [43].

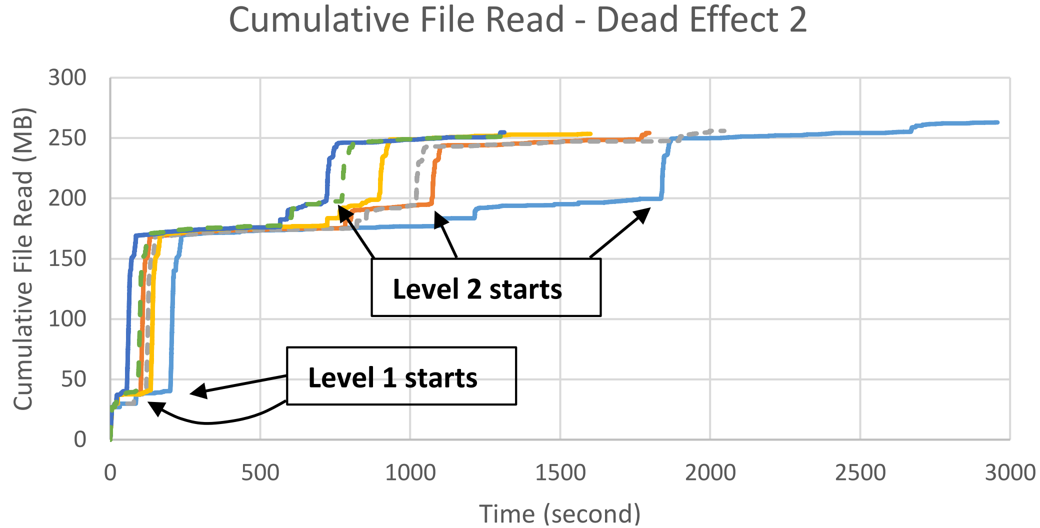

Figure 4.2. Cumulative file read in six playthroughs of the first two levels of Dead Effect 2.

well as the transition time among the states, which captures the rate of game play. The input parameters to AppStreamer allow for an exploration of the two dimensions of the tradeoff—the amount of storage used on the device and the bandwidth used to bring data to the device versus the pause time inflicted on the user.

Evaluation

We evaluate the effectiveness of AppStreamer through two distinct types of games customized for mobile devices—Dead Effect 2, a first-person shooter game and Fire Emblem Heroes, a turn-based strategy role-playing game. We conduct user studies for both of these games and find that in the majority of the cases, the users do not find any significant degradation in quality of experience—95% for Dead Effect 2 and 96% for Fire Emblem Heroes. This is achieved while using respectively 87% and 86% less storage than the baseline. We also conduct microbenchmark studies to evaluate the effect of various configuration parameters on the two dimensions of the tradeoff, which guides choice of the values for the parameters.

In summary, we describe the following contributions in this chapter.

1. We create a mobile gaming middleware AppStreamer that reduces the amount of download needed during installation as well as local storage space needed while playing the game. We do this through expanding the traditional notion of prefetching and caching to include a cloud storage server, as a level beyond local storage on the device.

2. We show that it is possible to record data block accesses and use them to predicting what blocks will be needed in the future and how far ahead in the future. We achieve this in a way that is generalizable across players and across games and does not require source code access.

3. We evaluate the effectiveness of AppStreamer through two popular, closed-source games and microbenchmarks. We find that AppStreamer is effective in reducing the storage requirement on the device, while not degrading the user quality significantly for this highly latency-sensitive workload.

4.2 Related Work

In this section we will discuss alternatives to downloading large games on mobile devices in their entirety, as well as related pieces of work.

Our work is complementary to two closely related classes of work. One class of work [44,45] makes modifications to the game at both the client and the server side and collaboratively gets the right frame to the device at the right time. The second class of work [46, 47] offloads computationally demanding parts of the local application to a cloud server. AppStreamer can use solutions from the first class to assist in the pre-caching decisions at the device, by using the resources at the server. Thus the computational load at the client can be reduced and close to optimal lookahead can be used for pre-caching content. AppStreamer can leverage solutions from the second class to offload some demanding parts of the game computation, such as, computing the rendering of some graphic-rich frame, to a nearby cloud server. Since

we predict some blocks that will be used in the near future, this can help in making the appropriate offloading decision at the appropriate time.

One approach to mobile gaming is the thin client model of gaming, often called "Cloud Gaming". In this model, the mobile application transmits user inputs to a server, which performs the computations and graphics rendering necessary in place of the mobile device. The rendered frames are then streamed to the mobile device as a video. This approach has the advantage of removing the computational as well as storage limitations of mobile hardware.; however, the bandwidth usage is high, and delays in interactivity due to latency can make this approach unappealing, as 100–200 ms of round-trip time is common in mobile networks. We perform experiments to compare AppStreamer with traditional cloud gaming approach and found that our approach has 77% lower bandwidth usage and 98% lower latency. Lee et al. proposed reducing cloud gaming latency by using Markov modeling and Kalman filtering to speculate about future frames [44]; however, this approach requires modifying the source code of the game, which can be unappealing to developers. It also has a limitation on the number of possible user inputs, not to mention that it significantly increases the bandwidth usage. Abe et al. propose the idea of streaming virtual appliance (VA) on mobile network by caching files predicted to be necessary in the near future [48]. The overall idea is similar to AppStreamer, however, we focus on highly interactive mobile applications as opposed to all applications in general. Our goal is to make the user experience indistinguishable from storing all files locally while their approach results in long, 1–3 minutes start-up delay, and multiple interruptions during each session.

File access prediction, the core of AppStreamer, has been studied and employed by most modern operating systems. Linux uses the read-ahead mechanism to prefetch sequential file blocks in order to improve performance [49]. Windows has components Prefetcher and SuperFetch which monitor file usage patterns during the boot process as well as during application launch in order to speed up both processes. However, other than simple sequential access prediction, the prefetching is usually limited to

OS boot and application launch; the usage pattern beyond launch can depend heavily on user or external input and is harder to predict. We tackle the harder of problem of predicting usage pattern during the entire application's usage which is affected by user input, among other things. In our scenario, the cost of a "cache miss" is much higher compared to the cost of prefetching an unnecessary block, thus a more aggressive prefetching approach is needed. Mobile applications are typically not as complex as desktop or server applications, and thus lend more easily toward profiling file usage patterns.

4.3 Design Overview

The goal of AppStreamer is to reduce storage requirements of large mobile applications. This is done by storing file blocks that are needed by the application on the mobile device, and speculatively fetching more blocks from a cloud storage server as the application runs. We focus on files that are part of the application package. This includes required files that some applications (especially games) download on the first run or when new updates are introduced. The key characteristic of these files is that they are read-only and are common across users. Files written by the application are typically small and cannot be shared across users, so we always store these files on the device. AppStreamer will not be as effective for applications that write large files such as video editing software.

We would like the system to work for all applications without any modifications or access to the source code. Thus, AppStreamer is implemented at the operating system level, at the logical file system layer. As long as the blocks the application reads are already on the device, the application will operate normally without any user-perceptible delay, even when some other blocks are not present on the device. Intuitively, past file reads should be a good predictor of future file reads, so this is what we use for our prediction model. It is possible to include other sources of the application's behavior from the perspective of the operating system, such as CPU

usage, memory usage, network usage, and past file writes, but we have found that past file reads alone already provide good prediction accuracy. There is also a desire to minimize the monitoring overhead which contributes to possible slowdown of the application.

AppStreamer's operation can be divided into two phases: offline phase and online phase, as shown in Figure 4.1. The figure shows the cloud storage server, which is a component introduced due to AppStreamer. The cloud storage server is expected to have *all* of the read-only content needed by the application. This content needs to be downloaded and stored at the server prior to the execution of the application on the mobile device. In the offline phase, the file access prediction model (Continuous-time Markov Chain in our case) is trained using a collection of file access traces from multiple users using a specific application as input data. Note that the training dataset does not need to include the specific user who will use the device in the online phase, since there are common trends in user behavior which can be generalized. In the online phase, as the user opens up and uses the application, the model predicts blocks that are needed in the near future, and fetches them from the cloud storage server in real time. Some applications, especially games, may communicate with their own application server as part of normal operation. AppStreamer does not use or modify such network traffic, so it will work regardless of whether the application communicates with an application server (e.g., an online game) or does not (e.g., an offline game).

While AppStreamer is agnostic to the type of application whose storage requirement is to be reduced, most of the large mobile applications today are games, due to their rich media content and large number of control paths. Other large applications such as video editing applications and map applications still generally have the key property that only a portion of their files are accessed at any one time, and thus AppStreamer would still work well for them. However, from this point onward, we will focus on mobile games running on smartphones and tablets as our target application domain.

AppStreamer relies on good connectivity to work well. From the experiments, it uses roughly 100–150 MB of bandwidth during each 20–30 minute session. This can translate to relatively high cost if the cellular connection is used. To alleviate this problem, AppStreamer can keep downloaded file blocks on the device's local storage as long as free space is available. When space runs out, a LRU eviction policy can be used to prioritize keeping blocks that are more likely to be accessed again. Since the free space is used to store only necessary blocks, the amount of storage used is still significantly lower than baseline where the entire application is stored locally. When a non-metered connection such as Wi-Fi is available, this is no longer an issue.

4.4 Components of AppStreamer

AppStreamer is composed of three main components: a component for capturing file blocks accessed by the application, file access prediction model, and data block fetcher that downloads data blocks from the cloud storage server. In this section, we describe the design of each component and how they interact among themselves.

4.4.1 File Access Capture

AppStreamer reduces storage requirement of applications by storing the file blocks that will be read by the application on the mobile device. As long as we can predict the application's file reads and fetch the required blocks before a request from the application, the application will work normally as if all blocks are present on the device. We use past file reads to predict an application's file accesses. To capture the file read information, we create a wrapper function for the file read function at the virtual file system layer of the Android operating system. Information captured includes file path, file name, file pointer position, number of bytes to be read, and timestamp. From the file pointer position and the number of bytes read, we can determine which blocks are being read. In the offline (training) phase, the file reads are logged and written to a file, which will be used to train the prediction model. In

the online phase, the file reads are given to the prediction model as input, in order to make predictions in real time.

4.4.2 File Access Prediction Model

The file access prediction model is trained using file access traces collected from multiple users playing a specific game. The raw input to the prediction model is the file read system calls the game makes. The input features associated with each call are file path, file name, file pointer position, number of bytes to be read, and timestamp. Because Android's file system operates at the block level, where each block is 4 KB, we convert each record of file read system call into individual 4 KB blocks.

There are several requirements the model must meet.

1. *Temporally biased*: The model should put more emphasis on blocks that are needed in the near future, as opposed to the blocks that are needed further in the future.

2. *Low overhead*: Since the model will be running on the mobile device itself, the model must have low computational and memory overhead.

3. *Probabilistic*: The model should provide probabilistic estimates — of the likelihood of a block being needed in the future. A probability threshold can then be set to decide that the blocks whose probabilities exceed this threshold will be prefetched. This gives us the ability to make tradeoff between missing a necessary block and downloading an unnecessary block.

We believe an appropriate model for these challenges is the Continuous-Time Markov Chain (CTMC). CTMC captures the possible transitions between states in a probabilistic manner, as well as the duration spent at each state. The duration information allows us to limit the lookahead when making predictions, thus meeting the temporally grounded requirement. Markov models are probabilistic in nature and the predictions give us the probability associated with each predicted block. The

computational and memory overhead depends on what we define as the states for the model.

The straightforward way to define the state of the CTMC is to use individual blocks read as the states. However, the memory, storage, and computational requirements become too high. Consider that a 1 GB game has 250,000 4 KB blocks. In the worst case, we need to store close to $250{,}000^2 = 62.5$ billion transition probabilities. Making predictions with the CTMC requires a depth-first-search, and having a branching factor of 250,000 makes the computation infeasible even at low depth limit, and the necessary depth limit can be quite high in this case. In practice, the computational cost of making a prediction is approximately k^d, where k is the branching factor and d is the depth; k depends on the transition pattern between states (i.e., how many outgoing states are there on average). In the worst case, k is the number of states, or 250,000 in the above case. d is approximately the lookahead parameter divided by the average duration spent at a state. We empirically show that a model that uses individual blocks as states is infeasible in Section 4.6.5.

Instead of using individual blocks read as the states, we use groups of blocks as the states. From the file access traces in Figure 4.2, we notice that data blocks tend to be accessed in groups, and these groups are similar across different users and different runs. Therefore, we design an algorithm for grouping blocks together, as well as extracting common patterns from different runs. We find that this significantly reduces the memory, storage, and computational requirements of the model.

4.4.3 Block Grouping

As mentioned earlier, grouping blocks is an important step for state space reduction for the CTMC. A group of blocks is simply blocks that are frequently accessed together in multiple different runs. We want each group to be as large as possible (to reduce the state space), while keeping the property that most blocks in the group are accessed together. Our specific algorithm, shown in Algorithm 3 creates such groups,

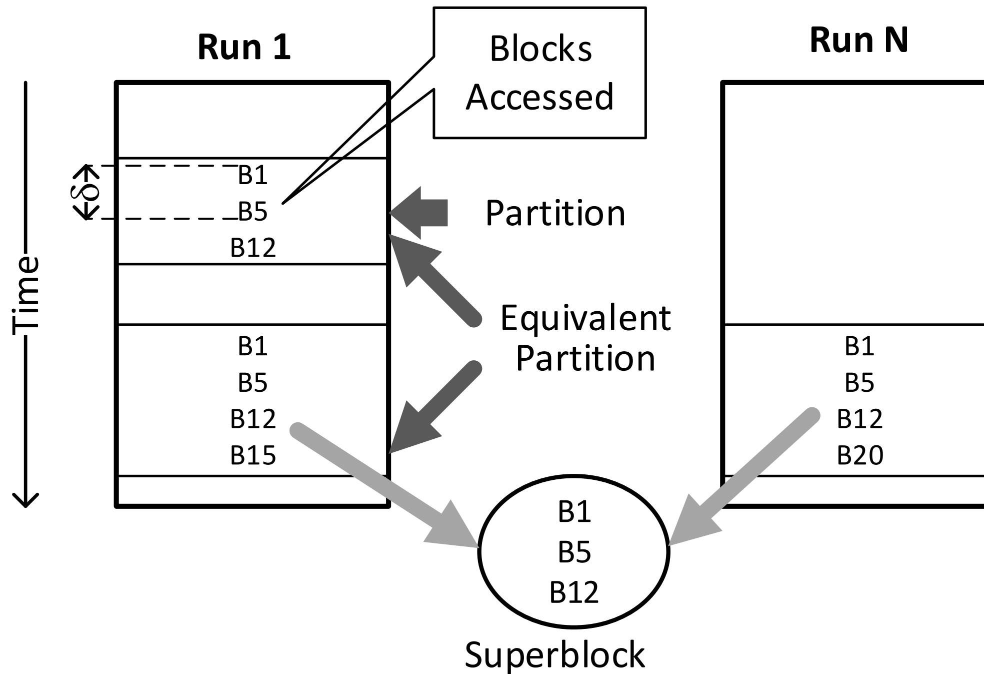

Figure 4.3. Grouping of blocks during training. Concepts of *Partition* and *Equivalent Partition* apply within a single trace while *Superblock* applies across multiple traces.

with the property that the number of blocks in a group multiplied by number of runs accessing the group is sufficiently large, which implies that the group is significant and common. Note that there can be overlap between groups. We propose a method for grouping blocks that meets these requirements and essentially summarizes the raw block accesses into essential information that the CTMC can easily process.

Our block grouping method can be divided into three steps as shown in Figure 4.3. In the first step, we group block reads in one trace that are close together in time into *partitions*. In the second step, we generate *equivalent partitions* from the partitions by merging partitions that are sufficiently similar. In the third step, we extract overlaps in equivalent partitions from different trace runs to create groups of blocks which we call *superblocks*. Superblocks represent groups of blocks that are frequently accessed together, and they are used as input to the CTMC.

```
Function partitionTrace(trace, δ)
    traces ← sortByTimestamp(trace);
    numPartitions ← 1;
    partitions[numPartitions] ← { };
    partitions[numPartitions].add(trace[1]);
    for i ← 2 to length(trace) do
        if trace[i].timestamp - trace[i-1].timestamp < δ then
            partitions[numPartitions].add(trace[i]);
        else
            numPartitions ← numPartitions + 1;
            partitions[numPartitions] ← { };
            partitions[numPartitions].add(trace[i]);
        end
    end
    return partitions;
```

Algorithm 1: Generate partitions from a trace file. Each partition contains successive file reads that are within a parameter (δ) spaced in time.

Partitions

In this step, each trace run in the training data is processed separately to produce *partitions*. Each partition contains blocks that are accessed close together in time, with the threshold of maximum time between successive block reads specified by the user-specified parameter δ. The value for δ should be chosen to separate block reads in different "batches." Empirically, the value should be between 1–1000 milliseconds. The algorithm is shown in Algorithm 1.

Equivalent partitions

Empirically, we found that some partitions within the same trace run are very similar to each other. In order to remove redundant information, we merge near-identical partitions from the same trace into one *equivalent partition*. We use Jaccard index as the similarity metric with each partition considered as a set of data blocks read. The Jaccard index of two sets A and B is defined as

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

A Jaccard index of 1 means that both sets are identical, while a Jaccard index of 0 means there is no overlap between both sets. All pairs of partitions within the same trace with Jaccard index higher or equal to a parameter denoted by τ are merged into one equivalent partition. Equivalent partitions are then continue to be merged with partitions that are highly similar to themselves. No changes are made to partitions that are not sufficiently similar to any other partitions. The algorithm is shown in Algorithm 2. After these steps, we have the equivalent partitions, which are used as input to the next step. We discuss choosing the values for δ and τ further in the microbenchmark section (Section 4.6.6).

Superblock

Equivalent partitions generated in the previous step represent groups of blocks that are frequently accessed together within a single run. At this point, we have essentially summarized the information in each run. In order to utilize information about different playing styles and different execution paths, we need to combine information from different runs into a single model, while removing redundant information. Due to different timings and execution paths, equivalent partitions in multiple trace runs are often not exactly identical. For example, an equivalent partition in one trace run can be made up of three equivalent partitions from another trace run, or may not show up at all in yet another trace run. Intuitively, we want to find a large

```
Function genEquivalentPartitions(partitions, τ)
    equivPartitions ← [ ];
    for i ← 1 to length(partitions) do
        highestSimilarity ← 0.0;
        for j ← 1 to length(equivPartitions) do
            sim ← jaccardIndex(partitions[i], equivPartitions[j]);
            if sim > highestSimilarity then
                highestSimilarity ← sim;
                s ← j;
            end
        end
        if highestSimilarity ≥ τ then
            equivPartitions[s] ← union(partitions[i], equivPartitions[s]);
        else
            equivPartitions.append(partitions[i]);
        end
    end
    return equivPartitions;
```

Algorithm 2: Merge similar partitions into one equivalent partition.

intersection of equivalent partitions across trace runs, consider the intersection as a single unit of blocks, and remove the intersection from the equivalent partitions that contribute to the intersection. We then repeat the process until no large intersection can be found. We call each large intersection a *superblock*. The challenge comes from the fact that the blocks in some equivalent partitions may not be present in all trace runs.

The pseudocode for creating superblocks is shown in Algorithm 3. First, we find the largest overlap between any n equivalent partitions where each belongs to a different trace and n can be lower than the number of traces. The intuition behind

not requiring n to be exactly the number of traces is that different players could go through different paths in the game before reaching a common point, and thus some common patterns may not be present on all traces. For the purpose of this algorithm, the size of an overlap is defined as the number of blocks in the overlap multiplied by the number of contributing traces n. The pseudocode for finding the largest overlap is shown in Algorithm 4. The recursive function should be called with $traceID = 1$. The blocks in this overlap are then removed from all of the contributing equivalent partitions, and put aside as a new superblock. The process repeats until the size of the overlap is lower than a user-specified threshold $minSuperblockSize$. At this point, the remaining equivalent partitions are merged into the superblocks that are closest in time. Recall that a superblock is the intersection of multiple equivalent partitions from different runs. For each contributing equivalent partition, we store the timestamp of the equivalent partition with the superblock. For example, a superblock that comes from three runs will have three timestamps stored, each corresponding to a run. To determine which superblocks are closest in time to each remaining equivalent partition, the timestamp corresponding to the equivalent partition's run is used. At the end, each trace is transformed into a sequence of superblocks, which are then used as input to the Markov model.

4.4.4 Markov Chain

A Markov chain model is specified by a number of states, the transitions among the states, and initial observation distribution. To capture the time duration spent in a state, we add duration which is associated with each possible state transition to the model. However, unlike the usual CTMC where the duration follows an exponential distribution, we simply store the arithmetic mean and standard deviation of the duration seen from the training data. Each superblock corresponds to a state in the CTMC. Time spent in a duration is independent of other parameters of Markov chain, which can be learned through the usual method. Although our model's formulation

```
Function createSuperBlocks(equivPartitions, numTraces, minSuperblockSize)
    superBlocks ← [ ];
    while true do
        overlap ← findLargestOverlap(equivPartitions, numTraces, 1, null);
        if overlap.size < minSuperblockSize then
            break;
        end
        remove blocks in overlap from the equivalent partitions it originated from;
        superBlocks.append(overlap);
    end
    /* Merge each remaining equivalent partitions into the closest superblock
       */
    for i ← 1 to numTraces do
        for j ← 1 to length(equivPartitions[i]) do
            if equivPartitions[i][j].size > 0 then
                merge equivPartitions[i][j] into the closest superblock, based on timestamp
                  of original equivalent partition from trace i;
            end
        end
    end
    return superBlocks;
```

Algorithm 3: Create superblocks from equivalent partitions.

is slightly different from the original CTMC, for simplicity we will continue to refer to our model as CTMC.

To determine the current state, recent file reads are partitioned in the same way partitioning is done for training. With the most recent partition, we find the superblock(s) that are a subset of this partition. If there are multiple superblocks, any arbitrary choice works well in practice because the lookahead is long enough that small differences in the starting state do not lead to widely divergent paths. The chosen superblock becomes the current state of the CTMC. In order to make predictions, from the current state, we do a depth-first search to compute the probability of each possible following sequence of future states. However, the number of possible

```
Function findLargestOverlap(equivPartitions, numTraces, traceID,
 curOverlap)
  largestOverlap ← [ ];
  if traceID > numTraces then
    return curOverlap;
  else
    for index ← 1 to length(equivPartitions[traceID]) do
      if traceID = 1 then
        newOverlap ← equivPartitions[traceID][index].blocks;
      else
        newOverlap ← curOverlap ∩
         equivPartitions[traceID][index].blocks);
      end
      if newOverlap.size ≤ largestOverlap.size then
        continue;
      end
      result ← findLargestOverlap(equivPartitions, numTraces, traceID +
       1, newOverlap);
      if result.size > largestOverlap.size then
        largestOverlap = result;
      end
    end
    return largestOverlap;
  end
```

Algorithm 4: Find largest overlap between one equivalent partition from each trace.

sequences grows exponentially with the length of the sequence. To keep this computation feasible, we prune the search in two ways: by stopping when the probability

of reaching that particular state through a specific path is lower than a threshold p_{stop} or when the cumulative duration exceeds the lookahead time L. At the end, we have the probability of each superblock being read within the lookahead time. If this probability is higher than the block fetching threshold, denoted by $p_{download}$, then the blocks in the superblock that are not already on the device are added to the download queue.

Since gamers with different skill levels often play games at different speeds, while the sequence of file block accesses may be the same, we take into account the user's playing speed when making prediction. As the user plays the game, we observe time duration spent at each state, and compare it to the average duration learned by the model. Because playing speed only affects the duration between state transition, the effect of shorter duration can be simulated by increasing the lookahead time L. Therefore, when we make predictions, the lookahead time is adjusted, with a faster player getting higher lookahead, proportional to how fast she plays compared to the average.

Similarly, different players may play in different ways, and this can result in different control paths being taken. Such information can also be captured by a *single* Markov model, by learning the probability of transitioning to state j given that the current state is i for all pairs of (i, j). This lets us utilize a single model for everyone, instead of having multiple models for each class of users, which would then require a classifier that determines the class of current user, as well as more training data.

4.4.5 Data Block Fetcher

The data block fetcher is responsible for downloading data blocks from the cloud storage server. There are two types of data blocks that need to be downloaded—predicted data blocks for future needs and application-requested data blocks for current needs that are not already on the device. The predicted blocks are stored in a download queue in first-in, first-out (FIFO) order. The speculative block fetcher

runs as a kernel thread and downloads predicted blocks in the download queue, using lowest network priority in order to not affect the responsiveness of other processes. The download queue is organized as a circular buffer to minimize memory overhead. Application-requested blocks that are not already on the device need to be fetched urgently because the application, and by extension the user, is waiting for its response and so a request is sent immediately to the cloud storage server. The original read system call will block until the data is obtained. To reduce latency, a TCP connection to the cloud storage server for both types of block fetching is set up at the beginning and kept alive until the application terminates.

4.4.6 Initial Files Cached

On Android, an application's executable and core files are packaged in a file in the format called Android Package Kit (APK) and stored on the device after installation. An application's APK file is always immediately read in its entirety when the application is launched, so we always store the entire APK file on the device to avoid delays. Some other resource files may also be accessed at launch, or soon after. Since we do not have enough time to fetch these files during the short duration between application launch and when they are accessed, they must also be stored on the device, in order to not introduce delays. We denote the amount of file blocks that are stored on the device at all times (excluding the APK file) by $B_{initial}$. This is a parameter that can be changed to make a tradeoff between delay and amount of storage saved. An alternative design is to store blocks that are accessed within the first x seconds in any of the training trace runs. However, with our design, we can control exactly how much permanent storage is used by AppStreamer.

4.4.7 Temporary Storage Limit

In the current state of practice, *all* of a game's files are *always* stored on the device. With AppStreamer, only the APK file and a small portion of the game's files

(specified by parameter $B_{initial}$) are always stored on the device. These files that are always stored on the device take up *permanent storage* (until the game is uninstalled). With AppStreamer, predicted blocks that are fetched speculatively are also stored on the device. However, they are deleted when the game's process terminates at the end of the user's session. Thus, we can say that these fetched blocks take up *temporary storage* on the device. As long as the user does not play multiple games at the same time (which is uncommon), this temporary storage can be shared among all games that use AppStreamer.

In the case where available free space is extremely limited, a hard limit on the temporary storage can be imposed. In this case, when the limit is reached and more storage space is required for storing fetched blocks, we evict old blocks using the least recently used (LRU) eviction policy. Blocks that are predicted (by AppStreamer) to be read in the near future are never evicted. This hard limit can have an effect on bandwidth usage and may introduce extra delays compared to when there is no limit. Such effects are quantified in our microbenchmarks (Figure 4.9 "Temporary storage limit").

4.5 Implementation

In this section, we describe how different components of AppStreamer are implemented in the Android operating system. Although different operating systems may use different file systems, AppStreamer only requires that files be organized in blocks, so it can be implemented in most operating systems.

4.5.1 Capturing File Accesses

File read is the input to AppStreamer's file access prediction model. In the offline phase, the reads are recorded and used as training data. In the online phase, the reads are given to the prediction model as input data, in order to make predictions. To collect the file reads in both phases, we create a wrapper function for the file read

system call `vfs_read` at the virtual file system layer of the Android kernel. In the offline phase, the wrapper function records the file read if it is a relevant file (i.e., a file used by that game and is not a user-specific file like her profile), and writes it to a log file, before doing the actual file read and returning the requested data as usual. Information recorded includes file name and path, current file pointer, number of bytes requested, and timestamp. Although every relevant file read results in a file write, empirically, the overhead of read logging is not noticeable to the user, so it does not affect the behavior of the user or the application. From our experiments, the average size of uncompressed log is 2.93 MB for 30 minutes of gameplay. It can be easily compressed if necessary, e.g., compressing repeated path and file names. In the online phase, the wrapper function passes on the information about the file read to the prediction model. If the requested blocks are already on the device, they are returned immediately to the caller. If the requested blocks are not on the device (i.e., a "cache miss"), the urgent block fetcher immediately requests the data from the cloud storage server. The urgent blocks are directly consumed from memory whereas the speculative blocks are stored in storage, to save the precious memory resource at the expense of slightly higher delay.

4.5.2 Block Storage

Most file systems organize files as a collection of blocks. It is therefore natural for AppStreamer to also operate at the granularity of blocks. In Android, each block is 4 KB in size. To simplify organization of blocks, we first create a file which acts as a container for all blocks ("block container file") and AppStreamer creates a mapping from the game's accessed file name and block offset to the location in the block container file, the latter being at 4 KB block boundary. Blocks can be stored in the block container file in any order. A hash table in memory stores the mapping and since a game uses a limited number of blocks in its working set, the size of this hash

table is small–with a 1 GB game and a liberal 25% of its blocks in the working set, the hash table is only 1 MB in size.

4.6 Experimental Evaluation

We evaluate AppStreamer in two ways: with user studies and with microbenchmarks. The goal of the user studies is to evaluate AppStreamer in a realistic setting, and measure the effect of delays introduced by AppStreamer by having users rate the user experience. The goal of the microbenchmarks is to evaluate how different parameters of the model affect the overall performance of AppStreamer. Due to the large number of parameter values explored, the microbenchmarks are done using trace-based simulation.

4.6.1 Games Used for Evaluation

We use two Android games in our evaluation: Dead Effect 2 and Fire Emblem Heroes. Dead Effect 2 is a 3D single-player first-person shooter game. Gameplay is divided into multiple levels, where the blocks needed for each level are loaded at the beginning of the level. The levels are linear (level 1, 2, 3, and so on), but different collected traces show diversity among users of the blocks accessed during gameplay. Its APK is 22.91 MB and all of its resources are stored in a single OBB (opaque binary blob) file which is 1.09 GB.

Fire Emblem Heroes is a 2D strategy role-playing game. Gameplay is divided into multiple small levels in several game modes. At first, only the main story mode can be played. As the player completes a few levels, paralogue mode, training mode, special maps mode, and player-versus-player mode are unlocked. These modes can be switched to and from at any time and in any order. The players choosing levels affects which blocks are needed, and also makes prediction in Fire Emblem Heroes nontrivial. The game has roughly 5,200 data files, whose sizes sum up to 577 MB, including the 41.01 MB APK. We chose these two games as they represent two dominant classes

of games on mobile devices, with differing characteristics in terms of themes, player interaction, and control flow. Both have the common characteristics of heavy graphics and low latency interactions.

4.6.2 Training Data

For Dead Effect 2, the trace data for the user study consists of 6 runs collected from two players. For the microbenchmarks, the trace data consists of 12 runs collected from four players. Each run is from start of the game to the end of level 2, which takes roughly 30 minutes.

The file read pattern of Dead Effect 2 is shown in Figure 4.2. As soon as the game is launched, the entire APK is read and a small part of the OBB file is read. When each level starts, the resources needed for that level are loaded at the beginning, resulting in a jump in the cumulative file read. During each level, there are several small jumps dispersed without any easily perceptible pattern. For Fire Emblem Heroes, the trace data for the user study consists of 7 runs collected from one player. For the microbenchmarks, the trace data consists of 14 runs collected from four players. For this game, each run consists of 20 minutes of gameplay from the beginning where the player is free to choose which level to play.

4.6.3 User Study – Dead Effect 2

Performance of AppStreamer depends on the network speed. For the user study, we set up the storage server on a local machine with network speed limited to 17.4 Mbps, which is the average worldwide LTE speed reported by Open Signal as of November 2016 [43]. The phones used are Nexus 6P running Android 6.0.1. Each participant plays the first two levels of the game, once on a phone with AppStreamer, and once on an unmodified phone. Each user then filled a questionnaire with four questions: 1) user's skill level (in that category of games, such as FPS), 2) quality

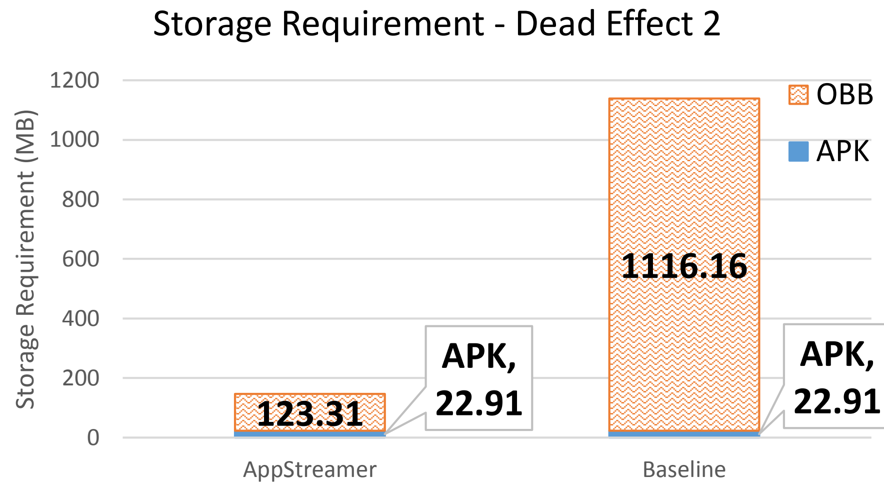

Figure 4.4. Storage requirements for Dead Effect 2.

of user experience, 3) delays during menu and loading screens, and 4) delays during gameplay inside a level.

The first user study is done with Dead Effect 2, with 23 users participating in the study. The amount of storage used by Dead Effect 2 is shown in Figure 4.4. Baseline refers to the current state of practice which is storing the whole game on the phone. The amount shown for AppStreamer corresponds to the permanent storage used by files that are always stored on the phone. As the user plays the game, more blocks are downloaded on the fly. These blocks are stored in the temporary space and not shown in the figure. On average, each run takes 146.51 MB of temporary space. Overall, AppStreamer uses 146.22 MB of permanent storage, compared to 1,139.07 MB for baseline. This represents a 87% saving of storage space.

The summarized responses to each question on the questionnaire are shown in Figure 4.5. 70% of the participants rate the overall user experience of playing with AppStreamer the same as playing on an unmodified phone. The remaining 30% rate the run with AppStreamer as marginally worse than baseline. There were no

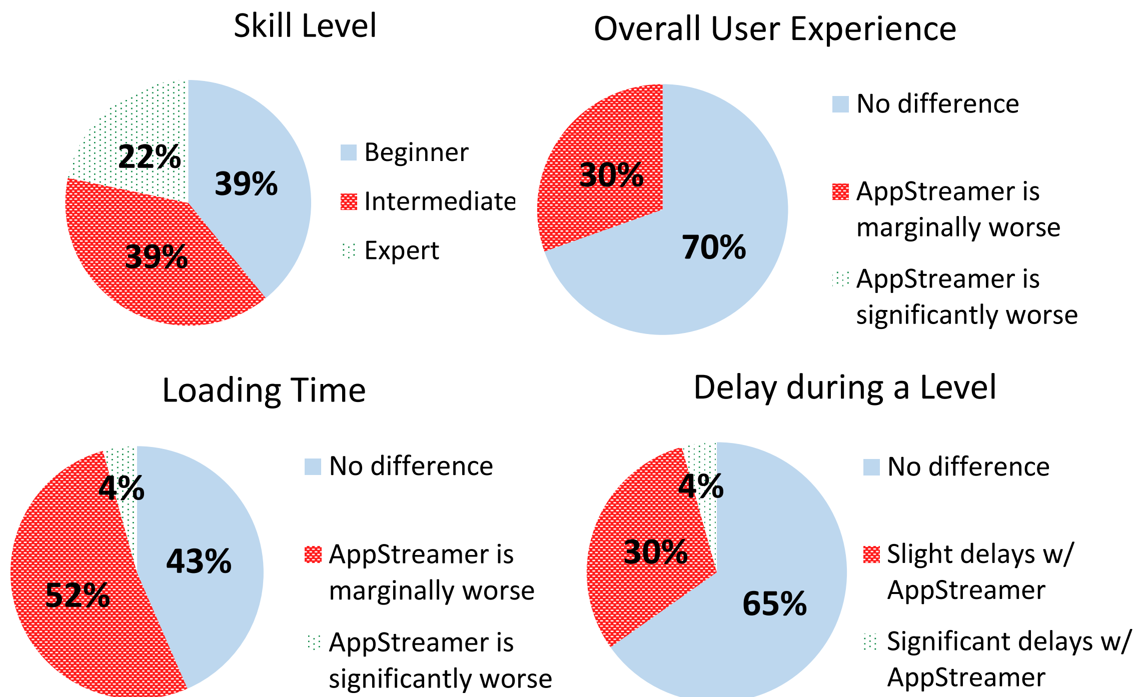

Figure 4.5. User study results for Dead Effect 2 with 23 participants.

disruptions other than pauses, such as, crashes, hangs, or visual glitches, during gameplay with our technique in place.

On average, there were 336.2 KB of "cache miss" — blocks that the game tries to read but are not present on the phone, during each run with AppStreamer. This translates to 0.15 second of delay, for approximately 28 minutes of gameplay, giving a 0.009% delay. The cache hit rate is 99.87%. The run that has highest amount of cache misses is affected by 1.52 seconds of delay. Compared to each level's loading time of roughly 20 seconds, this extra delay is barely noticeable. This shows that AppStreamer is able to predict and cache most of the necessary blocks before they are accessed for Dead Effect 2.

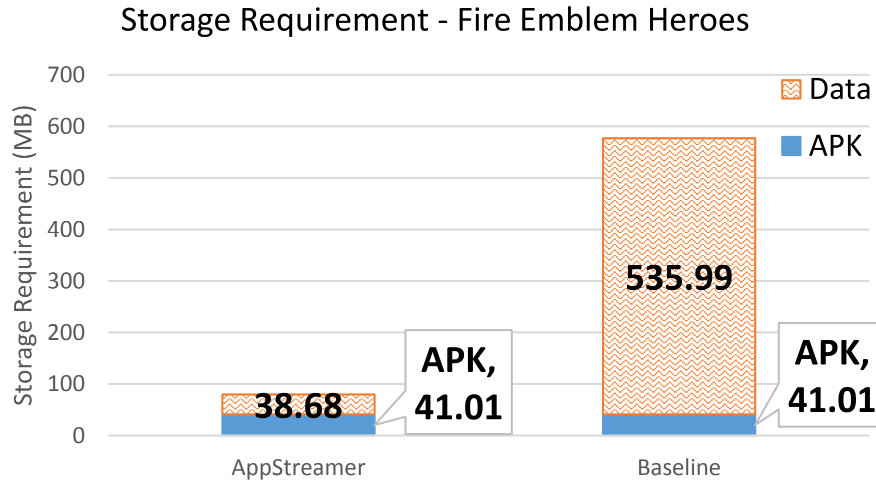

Figure 4.6. Storage requirements for Fire Emblem Heroes.

4.6.4 User Study – Fire Emblem Heroes

The second user study is done using Fire Emblem Heroes, with 26 users participating in the study. The study's setup is the same as the first user study with Dead Effect 2, except that the user is free to play any level for 20 minutes on each phone. The game has several different modes which are unlocked as the first few levels are completed, and some of these modes have multiple levels which the player can choose to play. Before playing, the participants are informed of these modes, and are instructed that they can switch to any mode and play any level as desired.

The amount of storage used by Fire Emblem Heroes is shown in Figure 4.6. Baseline refers to the current state of practice which is storing the whole game on the phone. The amount shown for AppStreamer corresponds to the permanent storage used by files that are always stored on the phone. As the user plays the game, more blocks are downloaded on the fly. These blocks are stored in the temporary space and not shown in the figure. On average, each run takes 104.29 MB of temporary space. Overall, AppStreamer uses 79.69 MB of permanent storage, compared to 577 MB for baseline. This represents a 86% saving of storage space.

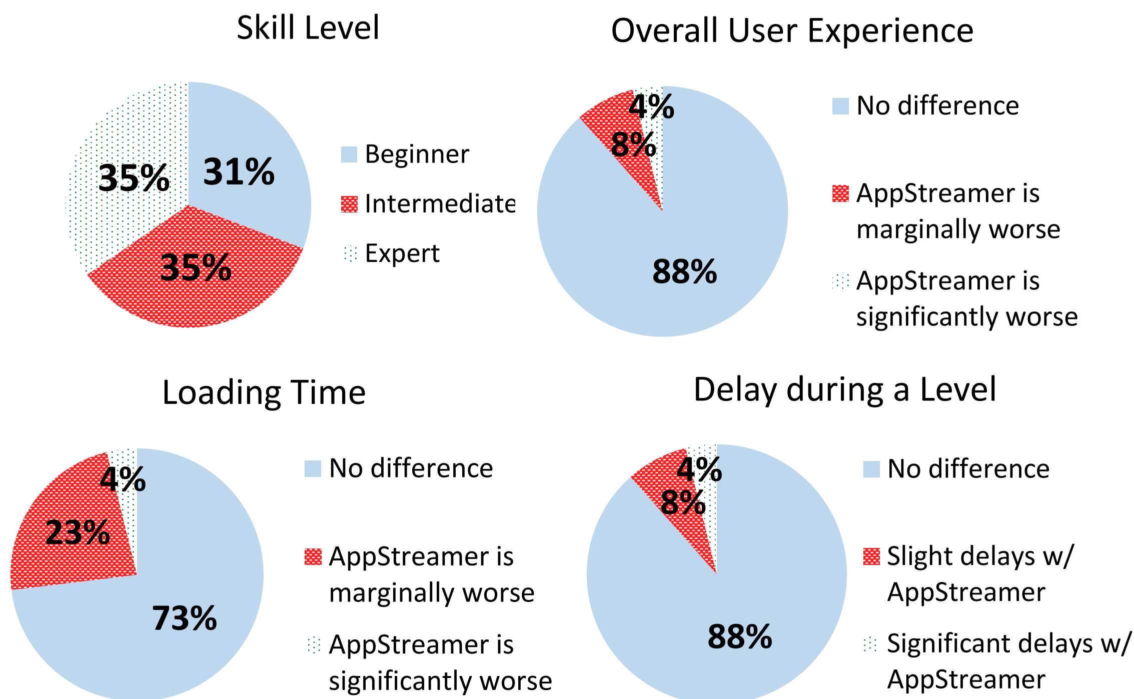

Figure 4.7. User study results for Fire Emblem Heroes with 26 participants.

The summarized responses to each question on the questionnaire is shown in Figure 4.7. 88% of the participants rate the overall user experience of playing with AppStreamer the same as playing on an unmodified phone. Again, there were no disruptions other than longer loading time and delays during gameplay. Interestingly, two users comment that the run with AppStreamer was actually smoother. This might be explained by the fact that the delays before and after each level are dominated by network communication with the game server, rather than file reads, and the delay may depend on the current load of the game server.

On average, there were 1.63 MB of cache miss during each run with AppStreamer, which translates to 0.75 second of delay in 20 minutes of gameplay, giving a 0.0625% delay. The cache hit rate is 97.65%. The run that has highest amount of cache misses is affected by 5.12 seconds of delay. Nevertheless, the user still rated the overall

user experience as no difference from the unmodified version. One user rates the run with AppStreamer as significantly worse due to significant delays. However, the communication log on the cloud storage server indicates that only 0.97 MB of blocks were missed and needed to be fetched urgently. This translates to 0.45 second of delay, which should be barely noticeable. Overall, the results of this user study show that AppStreamer is able to predict and cache most of the necessary blocks before they are accessed, even when there are different branches for different users in the gameplay.

4.6.5 Comparison with Prior Work

In this section we compare the bandwidth consumption and latency of AppStreamer to state-of-the art cloud gaming systems. One challenge in thin client gaming is that users are disturbed by latencies higher than 60 ms [50]. Lee et al. addresses this problem by using speculative execution, which can mask up to 128 ms of latency, at the cost of using between 1.51 and 4.54 times as much bandwidth as standard cloud gaming [44]. Using speculative execution requires access to and modification of the source code of the game, so we could not directly compare AppStreamer to speculative execution. We did, however, test the performance of a thin client using GamingAnywhere, an open source cloud gaming system [51].

Cloud gaming

In order to determine the bandwidth usage of a thin client model, we ran Nox, an Android emulator, and GamingAnywhere on a server and the GamingAnywhere client on a smartphone. We tested both Dead Effect 2 and Fire Emblem Heroes, and recorded the download bandwidth usage of the GamingAnywhere application on the smartphone. Data uploaded from the smartphone consists of encoded input events (such as swipes and taps), and data downloaded consists of audio and video of the game being streamed. The bandwidth usage of cloud gaming and AppStreamer

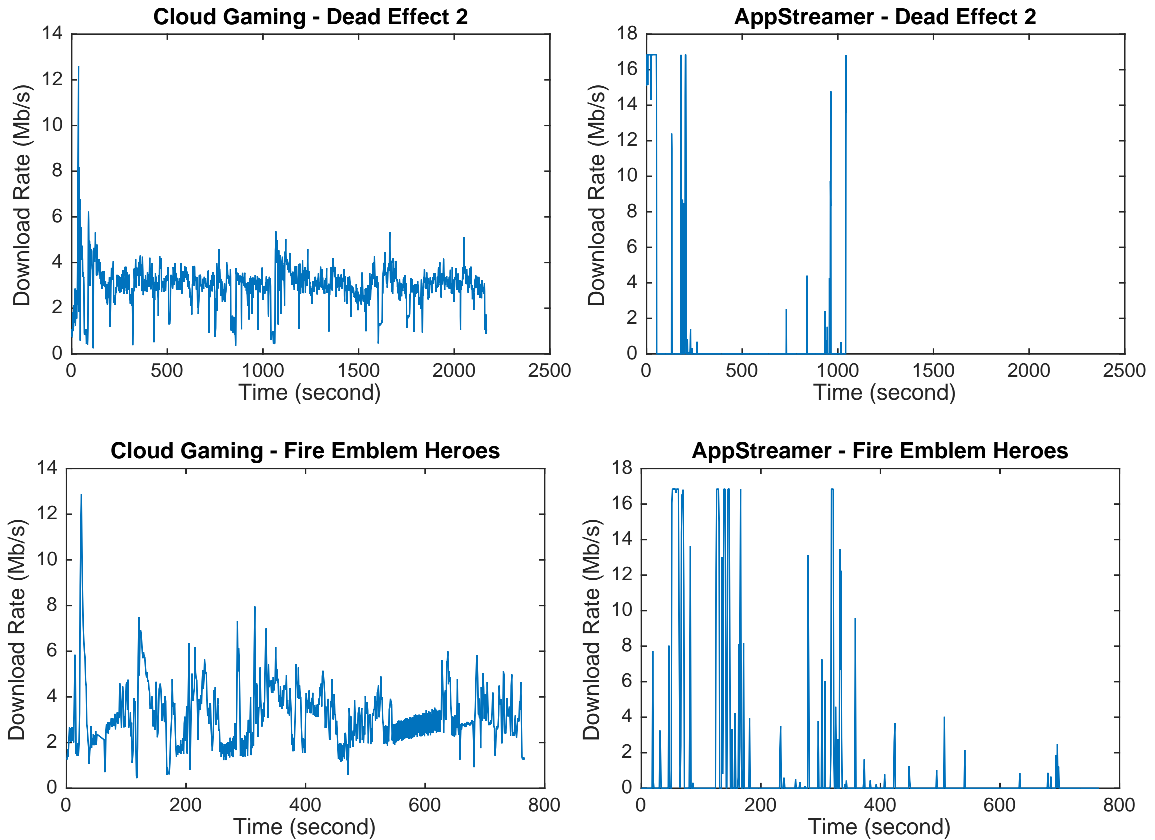

Figure 4.8. Comparison of bandwidth consumption between cloud gaming (left) and AppStreamer (right) for both games.

are shown in Figure 4.8. We found that for Dead Effect 2, cloud gaming uses 3.01 Mb/s while AppStreamer only uses 706 Kb/s on average. For Fire Emblem Heroes, cloud gaming uses 3.20 Mb/s while AppStreamer only uses 745 Kb/s on average. This shows that traditional cloud gaming is a lot more bandwidth intensive than our file block streaming approach, with 4.3X higher bandwidth requirement for our two target games.

As mentioned earlier, latency is a key measure of usability of cloud gaming tools. Latency can be very visible and annoying to users, as the time between every user input (e.g., a screen tap) and the frame that shows the effect of that input is the network round-trip time, plus video/audio encoding and decoding delay. The network round-trip time depends largely on the distance between the cloud gaming server and

the client. Based on typical placement of gaming servers, a typical round-trip time is 100 ms [52]. On the other hand, AppStreamer is not as heavily affected by latency as much as cloud gaming approaches, since speculative block fetches are mostly in batches and can be easily pipelined. The urgent block fetches are affected by the latency, but the amount of urgent block fetches is typically small. As a back-of-the-envelope calculation, for Dead Effect 2, on average 84.05 out of 64,858 blocks accessed are fetched urgently. Fetching each block requires 100 ms + 4 KB / 17.4 Mbps = 101.8 ms. Thus, the overall delay is $\frac{84.05}{64858} \times 101.8 = 0.13$ ms, which is much smaller than the constant 100 ms in the cloud gaming approach. For Fire Emblem Heroes, AppStreamer's overall delay is $\frac{416.96}{17731} \times 101.8 = 2.39$ ms.

Block-wise predictor

In addition to the cloud gaming approach, we also compare AppStreamer to a simple file access prediction algorithm that operates at the block granularity, which we call *BlockPairLookup*. In the training phase, it stores all pairs of blocks (B_i, B_j) such that B_j is read within the lookahead time L after B_i is read. In the online phase, when a block B_i is accessed, it predicts all B_j's where (B_i, B_j) is in its memory.

We run the BlockPairLookup algorithm for both games and compute the delay and amount of unnecessary blocks downloaded using our simulator. We find that it has excessive memory utilization—20.5 GB with 30 seconds lookahead with Dead Effect 2 and 4.6 GB with 60 seconds lookahead with Fire Emblem Heroes. Both would be infeasible on today's mobile devices. For Dead Effect 2, with BlockPairLookup, average delay per run is 6.32 seconds (8.4X of AppStreamer), and 74.39 MB of unnecessary blocks are downloaded (1.1X of AppStreamer). For Fire Emblem Heroes, average delay per run is 7.32 seconds (18.8X), and 64.10 MB of unnecessary blocks downloaded (1.1X). Because BlockPairLookup's predictions are always a superset of AppStreamer's predictions, the higher delay is likely due to the unnecessary blocks that are put in the download queue delaying the download of necessary blocks and

the inefficiency of requesting a single block at a time. This shows that models that operate on single block granularity incur too much memory and delay and are thus impractical.

4.6.6 Microbenchmarks

In this section, we evaluate how different parameters affect the results. The parameters studied are δ, τ, p_{stop}, L, $minSuperblockSize$, $p_{download}$, and $B_{initial}$, described in Section 4.3, as well as buffer size and network connection speed. The results are generated based on trace-based simulation. In the simulation, first training data is used to train a Markov model. Then, file reads from the test data is replayed and given as input to the Markov model. Blocks predicted by the model that are not already present on the phone are logically fetched from the storage server, with network speed fixed to a certain value to simulate real network conditions. In the case where buffer size is limited, we employ the LRU policy to evict blocks from the limited storage available.

Since there are many parameters, we conduct the microbenchmarks by varying one parameter at a time, and fixing the rest of the parameters to the optimal value. Optimal values are chosen by carefully weighing the tradeoff between delay and false positives, with higher weight given to delay, as it has a direct impact on the user experience. The values are $\delta = 0.1$ second, $\tau = 0.9$, $minSuperblockSize = 17$, $B_{initial}$ $= 122$ MB (excluding APK), $p_{stop} = 0.01$, $L = 60$ seconds, and connection speed = 17.4 Mbps. By default, we do not set a limit on temporary storage used to store fetched blocks. The average length of each run is 1,653 seconds. Due to limited space and the fact that the results show the same trends, we omit the microbenchmark results for Fire Emblem Heroes, and show only results for Dead Effect 2. The output metrics are delay and false positives, defined as predicted blocks that are not read by the game within 8 minutes of being predicted. Delays that are long or frequent enough

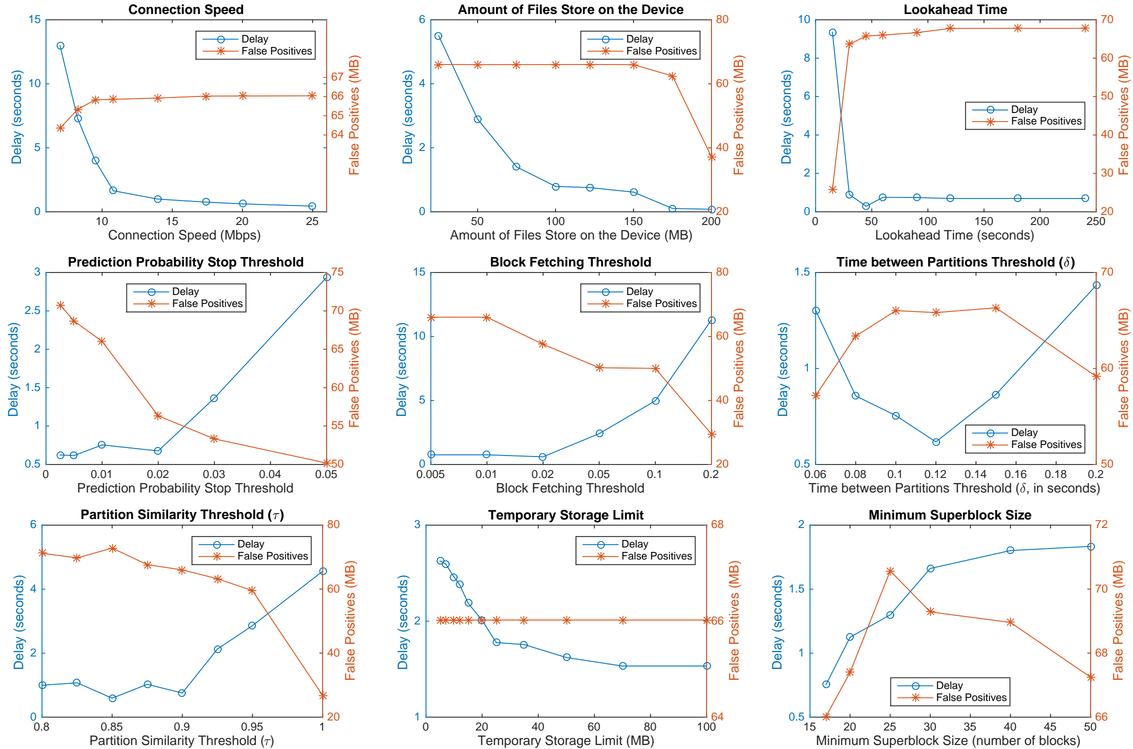

Figure 4.9. Microbenchmarks for Dead Effect 2.

can ruin the user experience, while false positives incur extra network bandwidth and energy cost.

The results are shown in Figure 4.9. First, we look at the effect of connection speed. In addition to average worldwide LTE speed of 17.4 Mbps, we also include average U.S. LTE speed of 13.95 Mbps and average worldwide WiFi speed of 10.8 Mbps. As expected, higher connection speed leads to lower delay. Even the lower WiFi speed of 10.8 Mbps is enough to keep the delay small, but speed lower than that will result in large delay. Connection speed has a negligible effect on the false positives. Next, we look at the amount of initial files cached on the phone, denoted $B_{initial}$. Higher value gives lower delay and false positives, at the cost of higher storage requirements. Delays are virtually eliminated at $B_{initial} \geq 175$ MB. This represents a storage savings of 84%. At the higher end of 200 MB, the amount of false positives is also reduced.

Recall that when making predictions, our Markov model relies on two stopping criteria to keep the computation tractable: lookahead time, denoted by L, and probability stop threshold, denoted by p_{stop}. From the results, as long as the lookahead time is at least 30 seconds, the delay remains constantly low and the amount false positives is largely constant. When the lookahead time is too low, delay increases significantly. Probability stop threshold is somewhat similar. As long as the value is 0.02 or lower, delay remains relatively constant. Higher value leads to higher delay. The amount of false positives is lower when p_{stop} is higher, as early stop means fewer blocks get predicted. The block fetching threshold, denoted by $p_{download}$, affects the final decision of whether or not to download blocks in the predicted merged cluster, based on predicted probability. It directly influences the amount of false positives, with higher threshold resulting in lower false positives. However, the delays are kept at an acceptable level only when $p_{download}$ is 0.02 or lower.

Time between partitions threshold, denoted δ, controls how consecutive blocks are merged into the same partition. Lower value leads to more partitions that are smaller. The results clearly show that 0.12 is the optimal value. This amount represents the upper limit of the amount of computation (e.g., image decoding) the application does between chunks of data in the same batch of read. Partition similarity threshold, denoted τ, controls merging of two similar partitions within the same trace. A value of 1 means the two clusters need to contain the exact same blocks in order to be merged. The results show that values between 0.8 and 0.9 produce similarly low delay, while higher values result in higher delay.

Temporary storage limit sets a hard storage limit for storing blocks fetched speculatively. This does not include the APK and files that are always stored on the phone. In reality, this buffer can be shared by all applications as long as they do not run at the same time, because as soon as an application terminates, all files associated with the application in the buffer can be removed. The results show that a 75 MB buffer is enough to produce the same result as when there is no limit. Since it is fair to assume that 75 MB of free space is available on the phone, this result shows that

AppStreamer is not bottlenecked by the amount of temporary free space. Temporary storage limit does not affect false positives, so they remain constant.

The minimium superblock size, denoted *minSuper- blockSize*, serves as the stopping criterion of the first step of the process of generating superblocks. Lower value leads to more precise model and predictions, but incur longer training time. The results confirm that lower values are always better than higher values. However, we could not complete the benchmark using values lower than 17, as the training time suddenly jumps from a few minutes to several hours.

4.7 Discussion

Here we discuss some logical next steps that can be used to augment AppStreamer. The model for prediction can be made more personalized considering that the style of gameplay can be very individualistic. This could involve using bits of information from the user profile to aid in the prediction task. Our solution performs all computation on the mobile device relying on the cloud storage server only for storage. Cloud gaming, on the other hand, performs all computation on the server and streams rendered content to the device, at the expense of heavy bandwidth usage. There is likely a middle ground whereby computation can be shared between the device and the cloud server. This can benefit from the long line of prior work in computation offloading from resource-constrained devices [46, 47, 53] but will have to be repurposed due to some domain-specific requirements, including stringent low latency, high degree of dynamism in gameplay, and the role of heavy graphics. Finally, our solution exposes a rich set of parameters that control the tradeoff between local storage and bandwidth used on the one hand and the pauses experienced by a user on the other. We have explored the effects of these parameters only individually, while there likely exist interdependent relationships that would suggest a combinatorial search through the space of parameter choices. The complexity for such a search has to be tamed.

In this work, we assumed that the connectivity is always good. However, this is often not the case with cellular connections. In Chapter 3, we introduced methods to predict connectivity degradation and discussed mitigation actions that a media streaming application can take. With AppStreamer, the application is essentially streamed from the cloud storage to the device, and the same idea can be applied. When the connection quality is expected to go down in the near future, the model should make predictions that go further in advance by increasing the lookahead time and reducing the probability threshold. This will increase the amount of data to be downloaded, and the prediction accuracy will be lower, but it is necessary, as the cost of not being able to download the necessary data when it is needed is much higher than the bandwidth and energy cost of downloading more than necessary. The analogy of bit-rate adaptation here is transcoding multimedia files such as images, videos, 3D textures to a reduced quality version. However, some care needs to be taken in order to ensure that the application still functions normally with the file content modified.

4.8 Conclusion

We set out to see how to reduce the storage pressure caused by resource-hungry applications on mobile devices. We found that mobile games were a significant contributor to the problem. We had the insight that mobile games do not need all the resources all the time. So if it were possible to predict which resources would be needed with enough of a lookahead, then they can be prefetched from a cloud storage server and cached at the device. We achieve this goal through the design of AppStreamer, which uses a Markov Chain to predict which file blocks will be needed in the near future and parametrizes it such that the model can be personalized to different speeds and gameplay styles. We show that for two popular third-party games, AppStreamer reduces the storage requirement significantly (more than 85%) without significantly increasing the delay for the end user.

5 CONCLUSIONS

In this dissertation, we explored different ways for mobile device and cellular network to cooperate in order to improve user experience and dependability of mobile applications and services. In Chapter 2, we showed that data collected by cellular network can be used to predict drop and drop duration in real time as well as how such information can be leveraged by the application. In Chapter 3, we proposed a novel framework that enables cooperation between mobile device and cellular network. We introduced the pre-caching service, which utilizes the framework to improve buffering strategy of multimedia streaming applications. This is done by coupling network condition monitoring with mobility prediction, in order to send advance notification of network degradation to the application. In Chapter 4, we proposed a technique for reducing storage requirements of mobile applications. This is based on the insight that an application usually only accesses a small portion of its files during a session.

REFERENCES

REFERENCES

[1] Harri Holma and Antti Toskala. *HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications*. John Wiley & Sons, 2007.

[2] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The Weka data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[3] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[4] Benjamin Cheung, Gopal Kumar, and Sudarshan A Rao. Statistical algorithms in fault detection and prediction: Toward a healthier network. *Bell Labs Technical Journal*, 9(4):171–185, 2005.

[5] Chi-Yao Hong, Matthew Caesar, Nick Duffield, and Jia Wang. Tiresias: Online anomaly detection for hierarchical operational network data. In *Proceedings of the 32nd International Conference on Distributed Computing Systems, IEEE*, pages 173–182. IEEE, 2012.

[6] Yan Liu, Jing Zhang, Michael Jiang, David Raymer, and John Strassner. A model-based approach to adding autonomic capabilities to network fault management system. In *Network Operations and Management Symposium, IEEE*, pages 859–862. IEEE, 2008.

[7] Sudarshan Rao. Operational fault detection in cellular wireless base-stations. *Network and Service Management, IEEE Transactions on*, 3(2):1–11, 2006.

[8] Raquel Barco, Volker Wille, Luis Díez, and Matías Toril. Learning of model parameters for fault diagnosis in wireless networks. *Wireless Networks*, 16(1):255–271, 2010.

[9] Yoshinori Watanabe, Yasuhiko Matsunaga, Kosei Kobayashi, Toshio Tonouc, Tomohiro Igakura, Shinji Nakadai, and Kenichirou Kamachi. UTRAN O&M support system with statistical fault identification and customizable rule sets. In *Network Operations and Management Symposium, IEEE*, pages 560–573. IEEE, 2008.

[10] Umar Javed, Dongsu Han, Ramon Caceres, Jeffrey Pang, Srinivasan Seshan, and Alexander Varshavsky. Predicting handoffs in 3G networks. *ACM SIGOPS Operating Systems Review*, 45(3):65–70, 2012.

[11] Cisco visual networking index: Global mobile data traffic forecast update 2014 – 2019 white paper. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html`. Accessed: Apr. 10, 2016.

[12] Why Pandora is a mobile-first company. `http://digiday.com/mobile/why-pandora-is-a-mobile-first-company/`. Accessed: Apr. 10, 2016.

[13] Cooper's law. `http://www.arraycomm.com/technology/coopers-law/`. Accessed: Apr. 10, 2016.

[14] Daqiang Zhang, Daqing Zhang, Haoyi Xiong, Laurence T Yang, and Vincent Gauthier. Nextcell: Predicting location using social interplay from cell phone traces. *IEEE Transactions on Computers*, 64(2):452–463, 2015.

[15] Haoyi Xiong, Daqing Zhang, Daqiang Zhang, Vincent Gauthier, Kun Yang, and Monique Becker. MPaaS: Mobility prediction as a service in telecom cloud. *Information Systems Frontiers*, 16(1):59–75, 2014.

[16] Joshua Bradley and Sherif Rashad. Time-based location prediction technique for wireless cellular networks. In *Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering*, pages 937–947. Springer, 2013.

[17] Iraj Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[18] Ozgur Oyman and Sarabjot Singh. Quality of experience for HTTP adaptive streaming services. *Communications Magazine, IEEE*, 50(4):20–27, 2012.

[19] Kristian Evensen, Tomas Kupka, Haakon Riiser, Pengpeng Ni, Ragnhild Eg, Carsten Griwodz, and Pål Halvorsen. Adaptive media streaming to mobile devices: Challenges, enhancements, and recommendations. *Advances in Multimedia*, 2014:10, 2014.

[20] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE. In *Proceedings of the Eighth International Conference on Emerging Networking Experiments and Technologies*, pages 97–108. ACM, 2012.

[21] Boxun Zhang, Gunnar Kreitz, Marcus Isaksson, Javier Ubillos, Guido Urdaneta, Johan A Pouwelse, and Dick Epema. Understanding user behavior in Spotify. In *Proceedings of the 32nd IEEE International Conference on Computer Communications*, pages 220–224. IEEE.

[22] Spotify charts. `http://charts.spotify.com/`. Accessed: Nov. 28, 2014.

[23] DASH Industry Forum dash.js player. `https://github.com/Dash-Industry-Forum/dash.js/wiki`. Accessed: Apr. 14, 2016.

[24] DASH Industry Forum members. `http://dashif.org/members/`. Accessed: Apr. 14, 2016.

[25] Stefan Lederer, Christopher Müller, and Christian Timmerer. Dynamic adaptive streaming over HTTP dataset. In *Proceedings of the Third Multimedia Systems Conference*, pages 89–94. ACM, 2012.

[26] Saamer Akhshabi, Ali C Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, pages 157–168. ACM, 2011.

[27] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, timid, and unstable: Picking a video streaming rate is hard. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, pages 225–238. ACM, 2012.

[28] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C Begen, and David Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE Journal on Selected Areas in Communications*, 32(4):719–733, 2014.

[29] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.

[30] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. *ACM SIGCOMM Computer Communication Review*, 45(4):325–338, 2015.

[31] He Yan, Ashley Flavel, Zihui Ge, Alexandre Gerber, Dan Massey, Christos Papadopoulos, Hiren Shah, and Jennifer Yates. Argus: End-to-end service anomaly detection and localization from an ISP's point of view. In *Proceedings of the 31st IEEE International Conference on Computer Communications*, pages 2756–2760. IEEE, 2012.

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

[33] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N Padmanabhan. Bartendr: A practical approach to energy-aware cellular data scheduling. In *Proceedings of the 16th Annual International Conference on Mobile Computing and Networking*, pages 85–96. ACM, 2010.

[34] Srinivasan Seshan, Hari Balakrishnan, and Randy H Katz. Handoffs in cellular wireless networks: The Daedalus implementation and experience. *Wireless Personal Communications*, 4(2):141–162, 1997.

[35] Timothy Sohn, Alex Varshavsky, Anthony LaMarca, Mike Y Chen, Tanzeem Choudhury, Ian Smith, Sunny Consolvo, Jeffrey Hightower, William G Griswold, and Eyal De Lara. Mobility detection using everyday GSM traces. In *Proceedings of the Eighth International Conference of Ubiquitous Computing*, pages 212–224. Springer, 2006.

[36] Pubudu N Pathirana, Andrey V Savkin, and Sanjay Jha. Mobility modelling and trajectory prediction for cellular networks with mobile base stations. In *Proceedings of the Fourth ACM International Symposium on Mobile Ad Hoc Networking & Computing*, pages 213–221. ACM, 2003.

[37] Mary Lenninghan. Consumer demand for content outstrips smartphone capabilities. `http://www.totaltele.com/view.aspx?ID=492839&G=1&C=1&Page=0`, 2016. Accessed: 2017-3-14.

[38] XDA Developers. Biggest Android games & apps. http://forum.xda-developers.com/showthread.php?t=1766439, 2016. Accessed: 2016-12-2.

[39] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 2–2. USENIX Association, 2012.

[40] Yi-Hsuan Hsiao, Hang-Ting Lue, Tzu-Hsuan Hsu, Kuang-Yeu Hsieh, and Chih-Yuan Lu. A critical examination of 3D stackable NAND flash memory architectures by simulation study of the scaling capability. In *Proceedings of the 2010 IEEE International Memory Workshop*, pages 1–4. IEEE, 2010.

[41] Android Guides. How to use micro SD card in Android Marshmallow? http://gadgetguideonline.com/android/android-marshmallow-guide/how-to-use-micro-sd-card-in-android-marshmallow/, 2016. Accessed: 2017-04-21.

[42] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 181–186. ACM, 1991.

[43] Open Signal. The state of LTE. https://opensignal.com/reports/2016/11/state-of-lte, 2016. Accessed: 2016-12-2.

[44] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165. ACM, 2015.

[45] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai: High-quality mobile gaming using GPU offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135. ACM, 2015.

[46] Mark S Gordon, Davoud Anoushe Jamshidi, Scott A Mahlke, Zhuoqing Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, volume 12, pages 93–106, 2012.

[47] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, and Ellen Zegura. COSMOS: Computation offloading as a service for mobile devices. In *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 287–296. ACM, 2014.

[48] Yoshihisa Abe, Roxana Geambasu, Kaustubh Joshi, H Andrés Lagar-Cavilla, and Mahadev Satyanarayanan. vTube: Efficient streaming of virtual appliances over last-mile networks. In *Proceedings of the Fourth Annual Symposium on Cloud Computing*, page 16. ACM, 2013.

[49] Fengguang Wu. Sequential file prefetching in Linux. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*, pages 218–261, 2009.

[50] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of Third ACM SIGCOMM Workshop on Network and System Support for Games*, pages 152–156. ACM, 2004.

[51] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. GamingAnywhere: An open cloud gaming system. In *Proceedings of the Fourth ACM Multimedia Systems Conference*, pages 36–47. ACM, 2013.

[52] Srikanth Sundaresan, Walter De Donato, Nick Feamster, Renata Teixeira, Sam Crawford, and Antonio Pescapè. Broadband internet performance: A view from the gateway. In *ACM SIGCOMM Computer Communication Review*, volume 41:4, pages 134–145. ACM, 2011.

[53] Yi-Hsuan Kao, Bhaskar Krishnamachari, Moo-Ryong Ra, and Fan Bai. Hermes: Latency optimal task assignment for resource-constrained mobile computing. *IEEE Transactions on Mobile Computing*, 2017.

VITA

VITA

Nawanol Theera-Ampornpunt received the Bachelor of Science degree in computer science from Carnegie Mellon University in 2009. In the same year, he joined Purdue's Department of Computer Science to work on his Ph.D. under the supervision of Prof. Saurabh Bagchi. His research interests include applied machine learning, artificial intelligence, mobile computing, and dependable distributed systems.