8-31-2017

Programming frameworks for mobile sensing

Hillol Debnath
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

PROGRAMMING FRAMEWORKS FOR MOBILE SENSING

by
Hillol Debnath

The proliferation of smart mobile devices in people's daily lives is making context-aware computing a reality. A plethora of sensors available in these devices can be utilized to understand users' context better. Apps can provide more relevant data or services to the user based on improved understanding of user's context. With the advent of cloud-assisted mobile platforms, apps can also perform collaborative computation over the sensing data collected from a group of users. However, there are still two main issues: (1) A lack of simple and effective personal sensing frameworks: existing frameworks do not provide support for real-time fusing of data from motion and visual sensors in a simple manner, and no existing framework collectively utilizes sensors from multiple personal devices and personal IoT sensors, and (2) a lack of collaborative/distributed computing frameworks for mobile users. This dissertation presents solutions for these two issues. The first issue is addressed by TagPix and Sentio, two frameworks for mobile sensing. The second issue is addressed by Moitree, a middleware for mobile distributed computing, and CASINO, a collaborative sensor-driven offloading system.

TagPix is a real-time, privacy preserving photo tagging framework, which works locally on the phones and consumes little resources (e.g., battery). It generates relevant tags for landscape photos by utilizing sensors of a mobile device and it does not require any previous training or indexing. When a user aims the mobile camera to a particular landmark, the framework uses accelerometer and geomagnetic field sensor to identify in which direction the user is aiming the camera at. It then uses a landmark database and employs a smart distance estimation algorithm to identify

which landmark(s) is targeted by the user. The framework then generates relevant tags for the captured photo using these information.

A more versatile sensing framework can be developed using sensors from multiple devices possessed by a user. Sentio is such a framework which enables apps to seamlessly utilize the collective sensing capabilities of the user's personal devices and of the IoT sensors located in the proximity of the user. With Sentio, an app running on any personal mobile/wearable device can access any sensor of the user in real-time using the same API, can selectively switch to the most suitable sensor of a particular type when multiple sensors of this type are available at different devices, and can build composite sensors. Sentio offers seamless connectivity to sensors even if the sensor-accessing code is offloaded to the cloud. Sentio provides these functionalities with a high-level API and a distributed middleware that handles all low-level communication and sensor management tasks.

This dissertation also proposes Moitree, a middleware for the mobile cloud platforms where each mobile device is augmented by an avatar, a per-user always-on software entity that resides in the cloud. Mobile-avatar pairs participate in distributed computing as a unified computing entity. Moitree provides a common programming and execution framework for mobile distributed apps. Moitree allows the components of a distributed app to execute seamlessly over a set of mobile/avatar pairs, with the provision of offloading computation and communication to the cloud. The programming framework has two key features: user collaborations are modeled using group semantics - groups are created dynamically based on context and are hierarchical; data communication among group members is offloaded to the cloud through high-level communication channels.

Finally, this dissertation presents and discusses CASINO, a collaborative sensor-driven computation offloading framework which can be used alongside Moitree. This framework includes a new scheduling algorithm which minimizes the total completion

time of a collaborative computation that executes over a set of mobile/avatar pairs. Using the CASINO API, the programmers can mark their classes and functions as "offloadable". The framework collects profiling information (network, CPU, battery, etc.) from participating users' mobile devices and avatars, and then schedules "offloadable" tasks in mobiles and avatars in a way that reduces the total completion time. The scheduling problem is proven to be NP-Hard and there is no polynomial time optimization algorithm for it. The proposed algorithm can generate a schedule in polynomial time using a topological sorting and greedy technique.

PROGRAMMING FRAMEWORKS FOR MOBILE SENSING

by
Hillol Debnath

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2017

APPROVAL PAGE

PROGRAMMING FRAMEWORKS FOR MOBILE SENSING

Hillol Debnath

Cristian Borcea, PhD, Dissertation Advisor — Date
Professor, Computer Science, New Jersey Institute of Technology

---

Narain Gehani, PhD, Committee Member — Date
Professor, Computer Science, New Jersey Institute of Technology

---

Reza Curtmola, PhD, Committee Member — Date
Associate Professor, Computer Science, New Jersey Institute of Technology

---

Xiaoning Ding, PhD, Committee Member — Date
Assistant Professor, Computer Science, New Jersey Institute of Technology

---

Luca Foschini, PhD, Committee Member — Date
Assistant Professor, Department of Electronics, Computer Science and Systems, University of Bologna

BIOGRAPHICAL SKETCH

Author: Hillol Debnath

Degree: Doctor of Philosophy

Date: August 2017

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2017
- Bachelor of Science in Computer Science and Engineering,
  Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, 2008

Major: Computer Science

Presentations and Publications:

H. Debnath, M. A. Khan, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "The Moitree Middleware for Distributed Mobile Cloud Computing: Design, Implementation, and Evaluation" *(under submission)*

H. Debnath, C. Borcea, N. Gehani, R. Curtmola, and X. Ding, "Sentio: A Personal Virtual Sensor System for Mobile Devices" *(under submission)*

H. Debnath, G. Gezzi, A. Corradi, N. Gehani, R. Curtmola, and X. Ding, C. Borcea "CASINO: A Collaborative Sensor-driven Offloading System" *(under preparation)*

P. Neog, H. Debnath, J. Shan, N. Paiker, N. Gehani, R. Curtmola, X. Ding, and C. Borcea, "FaceDate: A Mobile Cloud Computing App for People Matching", *in 7th International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (Mobilware 2016)*, December 2016

M. A. Khan, H. Debnath and C. Borcea, "Balanced Content Replication in Peer-to-Peer Online Social Networks", *in IEEE International Conferences on Social Computing and Networking (SocialCom 2016)*, October 2016

M. A. Khan, H. Debnath, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "Moitree: A middleware for cloud-assisted mobile distributed apps", *in 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud 2016)*, March 2016

C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, "Avatar: Mobile distributed computing in the cloud" *in 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud 15)*, March 2015

H. Debnath, C. Borcea, "TagPix: Automatic Real-time Landscape Photo Tagging For Smartphones" *in 6th International Conference on Mobile Wireless Middleware, Operating Systems, and Applications (Mobilware 2013)*, November 2013

I dedicate this thesis to my family: my loving Mom, Mina Rani Nath; my Dad, Ratish Kumar Debnath; my sister, Niti Nova Debnath; and my wife, Anju Biswas who have always kept faith in my abilities and helped me to dream bigger.

ACKNOWLEDGMENT

I would like to convey my wholehearted gratitude to my dissertation advisor, Dr. Cristian Borcea for the immense support he has provided me throughout my Ph.D study and research. It would be impossible to describe the patience he has shown to improve my scientific thinking and technical writing. He always greeted me with enthusiasm and curiosity whenever I came up with any research idea. He always helped me to get motivated during my difficult and depressing period of research. Without his persistent guidance and support, it would not be possible to complete this dissertation. Above all, his advice on life in general helped me to grow as a better professional and person. I consider myself truly fortunate to have him as my advisor.

I would also like to thank Dr. Reza Curtmola, Dr. Narain Gehani, and Dr. Xiaoning Ding for helping me with valuable suggestions and feedback throughout my research. I cannot thank them enough for the patience they have shown to improve the writing and presentation of my research papers.

I would like to thank Dr. Luca Foschini from University of Bologna, Italy for being a part of my dissertation committee. His very thoughtful comments and suggestions have helped me to improve the presentation of this dissertation.

I thank my fellow lab-mates Manoop Talasila, Susan Pan, Daniel Boston, Nafize Paiker, Giacomo Gezzi, Mohammad Ashraf Khan, Pradyumna Neog for their support during my research. The constructive discussions with them have always helped me to clarify confusion and improve my ideas. I would also like to thank my friend Ratan Dey, Shoubhik Mondal, and Anjana Grandhi for helping me to get comfortable in a new country and culture.

Above all, I could not possibly arrive this stage of my life without the dedication, love, and support my parents have provided me. I am forever indebted to my mom, Mina Rani Nath and my dad, Ratish Kumar Debnath for all the sacrifices they have

made to provide me the best education and support. I would like to thank my sister Niti Nova Debnath for her loving support throughout my life. I thank my wife, Anju Biswas for her unconditional love and support to me. Without her inspiration and positive attitude during many frustrating and hopeless situations, it would be impossible for me to complete this dissertation. Finally, I would like to thank my daughter, Aria Debnath for bringing happiness, inspiration, and a new meaning of life.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

The proliferation of smart mobile devices in people's daily lives is making context-aware computing a reality. While smart phones and tablets are already ubiquitous, wearable sensors and devices (e.g., smart clothing, smart glasses, activity trackers, 3D motion sensors, smart-watches) represent the next wave of mobile/sensing technology that are enabling context-aware computing. It is predicted that the annual shipment of wearable computing devices will exceed 485 million by 2018, and the global revenue of smart wearable sales will rise from $4.5bn in 2014 to $53bn in 2019 [1, 10].

Smartphones and smart-wearable devices are embedded with various types of sensors such as motion sensors (e.g., accelerometers, gravity sensors, gyroscopes, and rotational vector sensors), environmental sensors (e.g., ambient air temperature and pressure, illumination, and humidity), position sensors (e.g., orientation sensors, magnetometers), health monitoring sensors (e.g., heart-rate monitor) and so on. Other different types of sensors [21, 11] are also emerging. Many apps and services use sensors on smart devices for understanding user-context, ranging from photo utilities [97] to health monitoring [34] and well-being [101, 72, 82, 7]. For example, UNICEF's Wearable For Good challenge [25] received hundreds of innovative ideas and applications of wearable sensors for well-beings of women and children. Context-aware computing based on IoT has also become a hot area of research [96].

However, there is a surprising lack of efficient, high level sensing API and frameworks for smart devices. For example, with current available sensing frameworks [3], it is difficult to combine motion sensors (accelerometer, gyroscope, etc) with visual sensors such as the camera in order to provide meaningful suggestions to users based on the sensed context and visual data (e.g., captured photo). For this

reason, programmers have to write significant amount of manual code for writing a smart photo tagging app. A sensing framework capable of combining motion and visual sensors in real-time is needed to provide meaningful, context-aware feedback on visual data.

Existing sensing frameworks are also typically limited to allowing an app to use sensors only from a single device. This contradicts a new trend that shows users having access to sensors on multiple personal devices and in IoT-enabled environments. No existing framework provides support for accessing sensors collectively from all devices owned by a user. Hence, it is necessary to find a personal sensing framework to utilize the collective sensing capability of user's devices.

Programmers face significant challenges for performing collaborative computations over a group of users' data in mobile platforms. There is a lack of a fast, scalable, and energy efficient distributed computing framework for mobile devices. Resource limitation of mobile devices can be alleviated by using cloud resources. But, it also introduces a huge burden to programmers to manage distributing data and computation among participating users, communication, etc. Code becomes fragmented as a consequence and programming becomes very complex in such a model. Therefore, it is essential to design and develop an efficient collaborative computation framework accompanied by an easy-to-use, high level API set.

Computation offloading [42, 67] has been successfully used in mobile-cloud platforms to reduce both computation latency and mobile device's energy consumption. However, there is a lack of dynamic offloading frameworks in a mobile distributed computation model used in platforms such as Avatar [31]. Existing frameworks [42, 39, 67, 63, 69] are typically limited to considering resource consumption of a single mobile device. When a group of mobile devices collaborate with each other, the offloading decision can be dynamically made by considering the global resource conditions of participating devices. This can decrease the overall

completion time of the whole computation. A dynamic collaborative computation offloading framework is needed to be designed and implemented.

The rest of this chapter is organized as follows. Section 1.1 presents the issues of current sensing-based photo tagging frameworks. Section 1.2 discusses the issues and overview of current sensing frameworks for personal sensing. Section 1.3 provides an overview of collaborative computation in cloud-assisted mobile platform. An overview of collaborative computation offloading is presented in Section 1.4. Section 1.5 presents the thesis of this dissertation. The contributions of this dissertation are presented in Section 1.6. Partial contribution done by other researchers in Moitree and CASINO is discussed in Section 1.7. Finally, Section 1.8 provides the structure of this dissertation.

1.1 Sensing-based Automatic Landscape Photo Tagging Framework

The necessity of an automatic photo tagging framework has long been felt, but it has become more acute in the last few years with the widespread availability of smartphones and tablets equipped with cameras. There are more than one billion smartphones and tablets in the world [106] today, and they are used to generate a huge number of photos. Many of these photos are tagged and shared with our social networks. For example, 300 million photos are uploaded daily on Facebook [114]. A huge number of these photos are captured using smartphone camera. Tagging all these photos manually is becoming almost impossible.

Using motion and position sensors available in a smartphone, it is possible to identify in which direction the user is aiming the camera at. A framework can utilize such sensed context and further use a landmarks database to automatically tag a significant number of photos: photos containing landmarks. Four goals should be met by such a framework designed for apps running on smartphones:

High accuracy automatic tagging: To tag each photo with just one tag: the name of the landmark in the camera focus. The tagging should require no user actions.

User privacy protection: The framework must not upload the photos to Internet tagging services such as Google Goggles. Thus, all tagging processing should be done locally on the phone.

Real-time tagging: To help users who do not know the name of the landmark they just photographed, the app should display the tag as soon as a photo is taken. Therefore, the processing should be fast.

Modest phone resource usage: The app should be lightweight and consume a modest amount of resources, with battery power and network bandwidth being the main concerns.

To the best of our knowledge, currently, there is no photo tagging framework for smartphones that can satisfy all these goals. Most of the existing automatic tagging systems are content-based, pattern matching, and classification-based systems [77, 80]. These systems either work on servers (do not protect user privacy, and may not be able to provide real-time tags) or they consume a lot of CPU and energy if ported on the phones. Furthermore, they demand photos of moderately good quality (i.e., do not work well for night photos) and taken from specific angles. Finally, they require extensive training.

1.2 Distributed Sensor Virtualization for Mobile Apps

Many new context-aware apps could benefit from a framework that allows them to leverage the collective power of all the sensors available to a user, which are termed personal sensors in this dissertation. A few examples include apps for fitness and health tracking, gaming, activity recognition, and disaster recovery [38, 24]. For instance, a mobile game running on a powerful mobile device (e.g., tablet) may access the accelerometer on a device worn by the player (e.g., smart watch) to offer a more

user-friendly and immersive experience. A very different example is a real-time health monitoring app, whose code is offloaded to the cloud to speed up execution and save battery power on the smart phone [42, 67, 31, 70, 71]. The code running in the cloud will need to access sensors such as accelerometer and heart rate monitor that are embedded in the user's phone and in her smart watch, respectively.

With current solutions, it would be difficult to develop such applications because programmers would have to write application-specific code to manage groups of devices and to access remote sensors on these devices in real-time. Dealing with remote sensors, particularly sensors from multiple devices, can be challenging. This is not only because sensors in different devices have different features and different hardware APIs, but also because apps must execute certain tasks, such as sensor exploration, sensor management, and data aggregation/fusion, with low latency.

To provide an efficient framework for personal sensors, we need to overcome a set of design and implementation issues: *API Design issues:* How to abstract local and remote sensors (or a combination of both) via an easy to use API set, which will work across devices? How to decouple application logic from low level sensor management and framework level logic? *Implementation issues:* How to correlate (i.e., temporal alignment) and merge data points when they are from different sensors, located on different devices? How to minimize the effect of communication delay and jitter between data points? *Optimality issues:* How to select the "best" sensor for a certain sensor type from several available sensors located on different devices, while considering latency, accuracy, and energy constraints? How to seamlessly switch from one sensor to another, where the context changes and a new sensor becomes the "best"?

1.3 Collaborative Computation in Cloud-assisted Mobile Platform

Sensing data can be used to generate more useful and interesting end-results by doing collaborative computation among users. Unfortunately, the limitation of computing resources, storage, bandwidth, and, energy on mobile devices makes it challenging to do any significant computation on them. Cloud-assisted mobile computing gained popularity by augmenting resources to mobile devices. Offloading computation and communication from mobile devices to software surrogates in the cloud has been proven to improve app response latency, reduce wireless communication overhead and energy consumption at the mobiles [107, 39, 42, 68, 118]. These surrogates can be instantiated as virtual machines (VMs), containers, or even processes. Microprocessor manufacturers have recently started providing shielded application execution over untrusted cloud platforms [9], thus offering guarantees that the surrogates are truly personal and protected from the cloud providers [29].

Executing distributed computation over mobile-surrogate pairs has become an attractive possibility. Users (via apps) can collaborate by forming groups defined by friendship, common interests, geography, etc. Examples of such collaborative and distributed apps include discovering alternative routes to avoid traffic jam/congestion, finding people of interest in a crowd using face recognition techniques (e.g., a lost child), monitoring and stopping the spread of epidemic diseases, mobile multi-player gaming, etc.

However, this approach incurs additional problems to app developers as the code becomes more fragmented. They are burdened with developing multiple components of the app, and maintaining these complex partitioning of codes (e.g., client and cloud components, communication among various components and participating devices, etc.). Moreover, the cloud part generally has a different execution environment than the mobile and it makes programmers' tasks even more complicated.

Platforms such as Avatar [31] can alleviate such complexity in developing collaborative and distributed mobile apps. In such a platform, a virtual machine (an avatar[1]) is instantiated as a surrogate in the cloud for each mobile device. The execution environment is same on both mobile and avatar (e.g., runs on same OS) and the same app can run on both. This allows apps to run seamlessly utilizing various resources from the mobile-avatar pair (e.g., sensors from smartphone or smartwatches and huge computational resources from avatar). In this setup, programmers no longer need to partition their apps into separate client and cloud components. However, it still involves writing significant amount of code for the distributed execution of the app which in turn requires communications among lots of computing entities under the hood (e.g., distributing data and computation to various devices, surrogates/avatars, cloud services, etc and to collect the result back from them). Therefore, a framework is needed which would take care of these low-level tasks and facilitate the execution of collaborative apps on distributed mobile computing platform such as Avatar. These distributed apps need to be executed with groups representing a collection of mobile device-surrogate pairs. In addition to providing the aforementioned run-time support, the framework should provide an easy to use, high-level API set for developing distributed apps for such platforms.

1.4 Collaborative Sensor-driven Offloading System

Apps running on a cloud-assisted mobile platform are generally partitioned into two parts based on expensiveness of computations. The heavier part is executed (i.e., offloaded) to the cloud. While this basic and static form of computation offloading sounds a viable solution for finishing computation faster and saving battery power, it has drawbacks. When computation is offloaded to the cloud, data accompanying the computation is also needed to be migrated to the cloud. If the data is large, it might

[1]We use the word "Avatar" to indicate the distributed platform and the word "avatar" to denote the surrogate virtual machine in the cloud

consume lots of battery and time due to long network communications. Consequently, the advantage of the offloading mechanism can get nullified. This is only one side of the problem. There can be other considerations (e.g., network condition) which can make offloading based on static partitioning ineffective.

It is evident that an effective computation offloading mechanism has to be based on dynamic partitioning considering the resource conditions of devices. In other words, it is crucial to decide *when* and *what* to offload to the cloud entity. For example, it might be efficient to run most computations on the mobile and offload very few when the battery is more than 80% full and the network condition is bad. But, the same app can be run by executing only the mandatory parts on the mobile and offloading most of the parts to the cloud if the battery is less than 40% and the network is very good. As a result, the decision making has to be dynamic and adaptive to the current situation.

Existing works [42, 67, 39, 69] on offloading are designed for a single device-cloud scenario. The offloading decision maker in these works only considers the local device's resource conditions. None of them considers a distributed computation scenario where a group of users are executing an app collaboratively. A distributed app is comprised of many smaller tasks/jobs which are executed by participating users' mobiles and avatars. A job can have dependencies on other jobs and it can be dependent on sensors or data (e.g., private data) residing on a particular user device. A dependent job needs to wait for its predecessor job to finish and generate the output data before it can start its own execution. If such two jobs are executed in two devices which have long communication delay in between them, it would hurt the whole computation. Due to the dependency on devices and other jobs, it is not possible to offload all the jobs to the cloud or execute all of them on mobile devices. In order to complete the whole distributed app as fast as possible, it is essential to have a job scheduler which considers such dependencies and utilizes all available computing resources to

execute jobs as parallelly as possible. An API set is needed by the programmers so that they can mark/annotate parts of their code as offloadable. The scheduler will only consider these annotated code components (jobs) for offloading to the cloud.

1.5 Thesis

The thesis of my dissertation is *context-awareness can be comprehensively utilized if supported by efficient and high-level programming frameworks for personal sensing and collaborative computation over mobile devices.*

1.6 Contributions of Dissertation

1.6.1 TagPix - A Real-time Photo Tagging Framework

We designed and developed TagPix, a sensing-based real-time landscape photo tagging framework for smartphones. The main novelties of TagPix are: (1) An algorithm that leverages the GPS and the orientation sensors on the phone to compute the angular distance between the object in the camera focus and the landmarks in the camera's angle of view, which then allows TagPix to accurately select the tag(s) for the object; and (2) Three usable methods, based on simple user actions and trigonometric calculations, to estimate the Euclidean distance between the user and the object in the camera focus, which allow TagPix to improve its tag selection accuracy.

We designed a new real-time, privacy preserving photo tagging framework, which works locally on the phones and consumes little resources (e.g., battery). TagPix provides high tagging accuracy without requiring any previous training or indexing. Since it does not use computer vision techniques, it works well independent of the photo quality. We devised three new and usable methods for estimating for estimating the Euclidean distance between the user and the landmark to improve tagging accuracy and remove false positive tags. Estimating this distance is difficult because phone cameras have digital zoom which cannot be used in classic optical

physics formulas. Our methods require minimal effort from users (e.g., move a few steps and point the camera to the object again) and employ lightweight trigonometric calculations. We developed a lightweight, low energy consuming prototype for Android. The prototype works on several types of Android-based phones and uses the Google Places API [54] to retrieve landmarks in the proximity of the user. This prototype was successfully tested with 6 users across 8 cities in the USA. Only using tagging based on angular distance calculations (i.e., the best method in terms of usability), TagPix achieves an average accuracy of 86%. After applying the Euclidean distance estimation in conjunction with angular distance, the accuracy increased to 93%.

1.6.2 Sentio - Distributed Sensor Virtualization for Mobile Apps

The second contribution of the dissertation is the design and development of Sentio, a middleware and sensing framework for collectively utilizing all sensors available to a user from various smart devices and IoT sensors. Sentio, a personal virtual sensor system (*PVSS*), enables apps to seamlessly utilize the collective sensing capabilities of the user's personal sensors. Sentio presents apps with a PVSS abstraction — a collection of virtual sensors formed on top of smart devices owned by the user and IoT devices located in the proximity of the user. Virtual sensors provide a unified interface to local and remote sensors for app development. Multiple virtual sensors could be leveraged to provide composite virtual sensors, which perform sensor aggregation and fusion.

Sentio is designed and implemented as a distributed middleware, which exposes a high-level API to apps running on any smart device owned by the user. Thus, it shields the programmers from all the low-level communication and sensor management details. We implemented a prototype of Sentio using Android-based smart phones and smart watches. We used Sentio to build two proof-of-concept

mobile apps: SentioApp and SentioFit. We also modified two open source games to use Sentio API: Tilt Control and Space Shooter. We ran multiple experiments using these four apps. For example, we ran the Space Shooter game on a mobile device and used a virtual accelerometer mapped to a smart watch as a game controller. Our experimental results demonstrate that Sentio can guarantee real-time delivery of sensor data and impose a minimal overhead.

1.6.3 Moitree - A Collaborative Computation Framework

For supporting collaborative computation over cloud-assisted distributed mobile platform (e.g., Avatar), we designed and implemented Moitree[2], the middleware that facilitates the execution of distributed/collaborative apps. Moitree also provides a high-level API set for developing distributed apps for such platform. Programmers can use the API to access resources, without any assumption of where the code is running (on mobile or avatar). Moitree communication API offloads the user-to-user communication from mobile devices to avatars. To keep the whole platform fast, efficient, and loosely-coupled, the API works with an event driven mechanism. And, the middleware is implemented following a message-oriented middleware(MOM) design to comply with the API framework architecture. While the concepts of Moitree are general and applicable to any distributed mobile cloud platform, we have designed it and implemented it for the Avatar platform.

We have implemented a prototype of **Moitree** middleware and programming framework for Android platform. It runs on Android devices and OpenStack-based cloud running Android x86 avatars. We have developed two proof-of-concept apps in order to evaluate the programming effort minimized by Moitree. One of the apps finds a lost child at a crowded event by performing face recognition on the photos taken by people attending the event; the other is a dating app based on users' preferences

[2]The word "moitree" is taken from Bengali, which means alliance/collaboration.

about partners' faces. Although both apps use face recognition to achieve their goals, the work-flow and their use of Moitree API is different from each other. The number of LoC of Moitree-based app implementation is reduced significantly when compared to their implementations done without Moitree. We also performed experiments with macro and micro-benchmarks on top of our prototype to evaluate the efficiency, scalability, and overhead of the middleware.

1.6.4 CASINO - A Collaborative Computation Offloading Framework

We have designed and implemented CASINO, a dynamic and collaborative offloading framework for distributed apps running on mobile-cloud computing platforms such as Avatar [31]. This framework provides an easy programming framework which can be used by programmers to partition their apps both statically and dynamically. For static partitioning, programmers can annotate their function/classes with "local" or "remote" to request the framework for executing them in the mobile or cloud respectively. Programmers can also mark parts of their code as "offloadable" for utilizing dynamic partitioning. The framework collects profiling information (e.g., CPU, battery level, the bandwidth, data communication cost, etc) from participating devices. It is worth noting that there are a large number of variables to consider when there is a long list of jobs and devices. This makes it difficult to find an optimal schedule which will minimize the total completion time of the whole computation. Lawler et al[74] showed that this problem is NP-hard and there is no polynomial time scheduling algorithm for such scenario.

We have designed a greedy scheduling algorithm which can generate a scheduling solution. We have achieved polynomial running time by reducing the exponential search space (e.g., all <device,job> combinations) to a polynomial range. The search space reduction is done by a combination of topological sorting and greedy method. First, an ordered sequence of jobs is generated by topological sorting which works

based on the job dependencies. After that, the algorithm only tries the (device,job) combinations in this ordered sequence. The greedy method is applied for each job to select the best device available during that step which minimizes its completion time. This step works in a greedy manner because it does not try all the combinations of device and jobs in each step. It only chooses the current best for each job.

In this way, the algorithm schedules the *offloadable* jobs to specific mobiles and avatars such that both the computation and communication time for a job are minimized. Hence, the total completion time of the distributed app is minimized. After the scheduling is done, the execution engine facilitates the actual execution of particular jobs in the scheduled device/avatar.

Although the concept of CASINO is generic and can be implemented for any mobile-cloud distributed platform, we have implemented it to work on top of the Avatar[31] platform and Moitree[64], its middleware. The scheduler is designed as a centralized cloud service and implemented using Java. The rest of the framework is implemented using AspectJ and Android SDK. The framework is evaluated in two parts. First, the job scheduler is evaluated using a simulated data set. The data set is modeled using realistic device and network conditions. The evaluation shows that the scheduling algorithm results in better schedules and completion time comparing to scenarios where : 1) all jobs are executed in the cloud, and 2) all jobs are executed on mobile devices. The second part of the evaluation is done to show that the execution engine carries out the mobile-to-avatar offloading mechanism in a fast and efficient manner. We have run micro-benchmarks to verify this claim. The micro-benchmark shows that the offloading framework is capable of executing computation offloading from mobile to avatar with very small energy usage and latency.

1.7 Contributors to This Dissertation

The Moitree platform was designed and implemented collaboratively with my colleague Mohammad Ashraf Khan. The author's contribution is the design and implementation of Moitree API, the SDK, the middleware (with the exception of the group management system), and a proof-of-concept app named LostChild. Ashraf designed the programming model and implemented the group management system. In order to understand my contribution, the whole platform is presented in this dissertation, including Ashraf's part.

The collaborative computation offloading framework, CASINO is designed and implemented in collaboration with Giacomo Gezzi. Giacomo designed and implemented the offloading mechanism for a single device-cloud scenario. In CASINO, the contribution of the author is the collaborative/distributed scheduling and execution part. He has designed the job scheduler and implemented the middleware support needed for data communication among mobile-avatar pairs.

1.8 Structure of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses about related works. Chapter 3 presents TagPix, a sensing-based automatic photo tagging framework. Chapter 4 describes Sentio, a sensing framework for personal collective sensing. Chapter 5 discusses Moitree, a middleware and programming framework for collaborative computation in cloud-assisted mobile platforms. Chapter 6 presents CASINO, a collaborative computation offloading framework. Finally, the dissertation concludes in Chapter 7.

CHAPTER 2

RELATED WORK

2.1 Mobile Sensing

With the advent of sensor-rich smart mobile devices, the mobile sensing research has become very prolific. Mobile sensing can be categorized as various types: individual, participatory, crowd, social, etc. An individual sensing system [40] generally runs on a single mobile device and uses a background service to collect sensor data from the mobile. Participatory [33, 84, 17, 16] and crowd sensing [49, 113] systems consist of a group of participant users who contributes (voluntarily or with monetary incentive) sensor data to form a larger database of sensed data. Social sensing [27, 46, 105, 90] uses a fusion of mobile, social and sensor data to fully enable context-aware computing in a social level. This dissertation discusses about sensing systems which can be categorized as individual, participatory as well as crowdsensing.

Apart from smart mobile devices, IoT devices and sensors have also become a viable tool for understanding users' context. IoT sensors are connected to communication networks. They can be both independent entities or part of a larger sensor network. IoT sensors are placed in various environments ranging from smart homes, smart cities, smart cars to power-grids. In this dissertation, we have considered sensors embedded in mobile devices as well as IoT sensors.

2.2 Related Work on Sensing-based Automatic Photo Tagging Framework

Besides the computer vision systems mentioned in Section 1.1, geotagging for multimedia content has been proposed in the literature as well [83]. Most of these projects are based on 3D modeling and object matching. Similarly, the work in [103] combines the 2D appearance of landmarks in photos with 3D geometric constraints

to extract scene summaries and construct 3D models. This system uses "iconic scene graphs for clustering based on geometric constraints, and then uses structure from motion techniques to generate the 3D model of the landmark. Thus, they are similar in nature to the computer vision techniques already discussed, which are not suitable for deployment on phones. TagPix provides a lightweight, real-time, and privacy preserving alternative.

A number of other projects have addressed photo tagging or used techniques similar to those in TagPix. In [92], the system expects the user to tag a set of photos manually, and then tries to correlate the other photos with this set using the time and location of the photos. It suggests existing tags based on these correlations. This system is lightweight, but it has limitations: it requires significant actions from users, and may have low accuracy in places with many landmarks. TagPix is able to overcome these limitations.

TagSense [97] is a smartphone automatic tagging system, which explored the opportunity of using the phone sensors to get a clue about what is being shot. The accelerometer, GPS, gyroscope, light sensor, and microphone are used to infer the context. The phones also collaborate to learn who may be in the captured photo. The main focus was to tag people and include keywords about context, such as "outdoor", "noisy", or "inside XYZ museum". While this system proposed a novel way of providing tags automatically based on sensor data collected from the user's phone, it still differs from TagPix in a very important aspect. This system did not try to identify the landmarks in the camera focus, and implicitly did not include a method to do so. As we explained already, this is not a straightforward problem: the landmarks in the photos may not be nearby, and selecting the most appropriate tag is difficult.

Argon [4] and Wikitude [51] are two augmented reality browsers which overlay tags for nearby landmarks or places of interest over the live camera screen. The main

similarity with TagPix is that they use a place-tag database to find nearby tags. Unlike TagPix, they do not attempt to place the tags precisely over the landmarks they represent. They place info bubbles on the camera screen to let the user know what landmarks are nearby. Furthermore, Argon does not overlay tags for farther away landmarks.

A location-driven tag suggestion system is proposed in [60]. This system use sources such as a public Geographic Names Information System (GNIS) database, community tags from Flickr pictures, and personal tags shared through users' photo collections. Bags of place-name tags are first retrieved, clustered, and then re-ranked using spatial distance criteria. The community tags from photos taken in the vicinity of the input geotagged photos are ranked according to the distance and visual similarity to the input photo. Compared with this system, TagPix achieves much higher tagging accuracy because it leverages the orientation sensors to eliminate the tags that are not in the camera's angle of view. By using the Euclidean distance estimation, TagPix is able to further improve the tagging accuracy.

Similar with this system, TagPix could use multiple tag-place databases, especially community and personal tags. For example, folksonomies such as the one presented in the MarkIt game [95] could be added to TagPix. In the literature, there are also tagging systems based on visual folksonomies, generate tags for new visually similar photos. Such systems utilize collaboratively annotated image databases, and then analyze the images to find visually similar photos and propagate the tags for them. TagPix is different in nature from this system because it works on the phone and requires no (or minimal) user actions.

The work in [87] focused on accurate localization of distant objects, and used similar methods with TagPix for distance estimation. The user needs to focus the target object in the camera viewfinder and then take multiple photos from different nearby positions. The distance is then estimated based on 3D models of the captured

photos. The major difference between this work and TagPix is in terms of usability. TagPix provides high tagging accuracy even without asking the users to move or take multiple photos. Furthermore, even when it does ask the users to move, it is only for a short distance and one additional photo as opposed to larger distances and many photos.

SmartMeasure [23] is a popular app in Google Play, which uses the camera to estimate the distance to a target object. It first asks the user to input the estimated height of the object. Then, it uses a simple trigonometric formula to estimate the distance. This method has some similarities with the methods used by TagPix to estimate the distance to an object. However, we do not require the user to input any value, which ultimately improves the usability and could potentially avoid errors due to wrong user input.

Table 2.1 Summary of Related Work Discussed for TagPix

Work	Goal	Used Techniques	Use of Sensors
Luo et al. [83]	Finding identity in photo albums	GeoTagging	No
Raguram et al. [103]	Identify landmarks in photo	3D modeling and object matching	No
Naaman et al. [92]	Generate tags for a photo	Manual tagging and Location-time based context matching	Yes
TagSense [97]	Tag people and location context	Uses sensor based context recognition	Yes
Argon [4], Wikitude [51]	Show places and direction on augmented reality browser	Uses location data and GPS	Yes
Satellites in our pockets [87]	Positioning system using mobile devices	3D modeling based distance estimation	Partially
Google Goggles [53]	Identify objects in a photo	Computer vision and machine learning	Yes

Google Goggles [53] is a visual search tool which require the user to upload the image to a server, which then returns relevant information. The servers supporting Goggles are trained with more than a billion photos [112]. While this system works relatively well for images containing text or logos, it does not work well with landmarks missing such features. It also needs very specific levels of light and

resolution to work well. We tested it, and it failed to identify all the landmarks that TagPix tagged correctly by using the phone sensors and our algorithm. Unlike Goggles, TagPix works independent of the photo quality and protects the user privacy because photos are not uploaded to a server to be tagged.

In the Table 2.1, we present a summary of the important related work discussed in this section.

2.3 Related Work on Personal Sensing Framework

Work that facilitates the development of mobile apps utilizing sensors has similarities with Sentio. SeeMon [62] is a framework for the development of context-aware applications, which monitors user context through sensors in a scalable and energy-efficient manner. Similar to Sentio, SeeMon provides sensor control policies. On the other hand, it does not provide a general-purpose interface or support for accessing sensors across devices or fusing data from multiple sensors.

Beetle [76] presents a system which supports simultaneous access to the same BLE-enabled sensor/device from different apps running on a mobile device. Due to the nature of BLE protocol (e.g., star-topology), a single smart phone/PC acts as a central node and other small wearable sensors/devices act as peripheral nodes. This solution does not work for offloaded mobile apps and does not work for non-BLE sensors. Furthermore, this setup does not work for multiple smart phones/tablets trying to access sensors from each other because Android devices cannot work as BLE peripherals. Rio [28] presents a system where an app running on a mobile device can access remote I/O devices by using Unix-like device files. Rio uses a client-server architecture for communication. However, it was not designed to handle a multi-device setup, with multiple clients and multiple servers interacting concurrently. Similar with Beetle, Rio does not provide support for offloaded mobile apps. In

addition, the deployment of both works is difficult because they need mobile devices to be rooted.

Code in the air (CITA) [104] simplifies the development of mobile applications that keep monitoring sensors to respond to user activities or context changes. With CITA, programmers specify "context-action" tasks, and the framework automatically distributes, executes, and coordinates tasks. PRISM [44] is a platform for developing and deploying community sensing applications. It automatically distributes and coordinates sensing tasks. CITA and PRISM focus on binding tasks and sensing data as well as on managing tasks. Sentio, on the other hand, focuses on providing seamless access to sensors. Sentio may be used together with CITA or PRISM to further reduce programming complexity of mobile context-aware app.

MobileHub [109] is a system that rewrites applications to leverage a sensor hub for power efficiency, without additional programming effort. Unlike Sentio, MobileHub only supports access to local sensors through the sensor hub. Another framework, AFV [66], is designed for developing context-aware applications that utilize the wearable devices in personal area networks. It provides an API for monitoring context at low costs. It differs from Sentio in that it does not support sensor data fusion from different devices and cannot effectively support real-time apps such as mobile games. It also lacks support for code offloading, which is provided in Sentio.

Sentio is also related to research on efficiently utilizing sensors on smart phones. RTDroid [116] studied how to reduce the latency of data collection from local sensors. Sentio can use a similar technique as RTDroid in its middleware to further improve performance. Offloading sensing tasks from mobile devices to the cloud has been studied to save energy on mobile devices [102]. However, the techniques are designed for sensors on an individual mobile device. Unlike Sentio, they do not use sensors from multiple devices collectively.

Table 2.2 Summary of Related Work Discussed for Sentio

Work	Type	Goal	Provides API?
SeeMon [62]	Mobile sensing	Providing scalable context monitoring in a single mobile	Yes
Code in the air [104]	Mobile sensing	Support for developing "context-action" tasks	Yes
PRISM [44]	Remote sensing	Executing sensing applications on remote mobile devices	Yes
MobileHub [109]	Mobile Sensing	Rewrites applications to utilize a sensor hub	No
AFV [66]	Personal Area Network	Monitoring context at low costs	Yes
BodyCloud [48], Fortino et al. [47]	Body Sensor Network	Collect sensor data from body sensors and store them in the cloud	Yes
BuildingDepot [26], BOSS [45]	Building Management Systems	Providing support for accessing sensors and actuators placed in buildings	Yes

Sentio targets wearable and IoT devices together with smart phones. A wearable IoT concept is presented in [59], and the placement of wearable sensors based on accelerometers is also studied [86]. These works have a different focus from Sentio, which aims to leverage the collective power of personal sensors from multiple devices.

Sentio is remotely related with wireless sensor network (WSN) management, particularly the management of Body Sensor Network (BSN), where various sensors are placed on human body to form a wireless sensor network. Many works in WSN and BSN areas have contributed on coordinating sensors and efficiently collecting data from sensors. For example, the work presented in [117] facilitates data collection from the cloud by virtualizing physical sensors. BodyCloud [48] provides XML based programming constructs for developing community BSN applications. It is proposed to utilize the cloud to store and process data streams from the sensors in a body area network [47]. Different from these works, Sentio concentrates on using personal sensors in smart devices. These devices are fundamentally different from the sensors in WSN and BSN from many perspectives, such as architecture, use cases, and sensor types. These fundamental differences make it difficult to use the designs for WSN or BSN management for personal sensor systems.

Another form of WSN is used by Building Management Systems (BMS), where sensors/actuators are placed in buildings and are connected by a wireless network.

BuildingDepot [26] and BuildingDepot 2.0 [115] provide sensor data storage for sensors such as energy meters, heating, ventilation and air conditioning (HVAC). These works are not designed to run on mobile devices, and they need dedicated servers to run Apache, MySQL, etc. Sentio can run entirely on mobile devices (even without the cloud entity). Thus, these works differ from Sentio in terms of use cases and they do not provide features such as multi-device sensor fusion, code offloading, seamless switching of sensors due to system status change, etc. Another work, BOSS [45] provides an API for sensors/actuators placed in a building to simplify application development. However, programmers need to write device/sensor-specific code (e.g., $set\_min\_airflow$). Sentio, on the other hand provides a high-level API, designed specifically for mobile apps. BOSS also needs to run several services (Hardware Presentation Layer, Transaction Manager) on a dedicated computer collocated with BMS. Thus, it cannot run on a setup with only mobile devices.

Table 2.2 shows a summary of important related work discussed in this section.

2.4 Related Work on Collaborative Computation Framework

While assisting mobile devices with cloud resources is a very active research area [107, 39, 42, 68, 118], Moitree is the first middleware for cloud-assisted mobile *distributed* apps. Very recently, a few works have investigated cloud support for mobile distributed computing [68, 118]. Clone2Clone [68] offloads peer-to-peer networking to the cloud, thus enabling more efficient communication among mobile users. Moitree, on the other hand, provides full system support for the execution of mobile distributed apps and a high-level API for programming distributed apps over mobile/avatar pairs. Sapphire [118] is a distributed programming platform for mobile cloud applications that separates the application logic from the deployment logic. Thus, programmers can modify distributed application deployments without changing the application code (e.g., change the caching behavior). This work is complementary to Moitree

and could be leveraged by the Avatar platform to allow for dynamic management of non-functional app features. It should be noted that Moitree is not just for offloading of computation and communication to the cloud; rather, it is a programming model to build mobile distributed apps based on dynamic context such as social groups, time, and location, while providing computation and communication offloading to improve efficiency.

Moitree has clear advantages in terms of latency, energy-efficiency, and availability over middleware platforms for programming distributed apps designed for purely mobile environments [79, 81, 85, 91]. Among the middleware for distributed programming over mobile ad hoc networks (MANET), LIME [91] and TMACS [79] propose group abstractions similar to Moitree. LIME [91] provides a framework in which mobile agents can form groups based on context-awareness. Moitree's programming model has two main advantages over LIME: more flexible communication abstractions, and its supporting middleware performs transparent dynamic group management. TMACS [79] proposes an object-oriented distributed middleware framework for MANET. In Moitree, groups are defined based on users and their activities activities rather than the types and scopes of objects as in TMACS. This makes mobile distributed programming simpler and more natural.

MELON [56] is a general purpose coordination language for MANET that supports asynchronous exchange of persistent messages. Although MELON provides an API similar to Moitree, it does not support group management or different types of communication between group members.

Pogo [32] and MobiSoC [58] are closer to Moitree because they use server-side resources to provide middleware platforms for specific areas of mobile computing. Pogo [32] proposes a middleware for distributed mobile phone sensing. Unlike Pogo which focuses on sensing, Moitree provides a general programming model for mobile distributed computing. Pogo does not explicitly use group abstractions such as

Moitree. Also, the assignment of mobile sensing devices to a particular researcher is done by an administrator in Pogo, while Moitree groups are handled dynamically by the middleware. MobiSoC [58] supports mobile social computing and provides a high-level API based on people and places, similar in nature with the one provided by Moitree. Both platforms use groups as main abstractions. But unlike MobiSoC which maintains global state about communities at the server-side, Moitree provides a distributed architecture in which apps work in peer-to-peer fashion. Furthermore, MobiSoC focuses on mobile social apps, while Moitree enables general-purpose mobile distributed apps.

2.5 Related Work on Collaborative Offloading Framework

Computation offloading has been used in many apps and frameworks for various purposes: face recognition [41, 78], image search [39], gesture and object recognition [36, 100, 63], data compression [75], video encoding & transcoding [75, 119], speech recognition [110], indoor map reconstruction [35], virus scan [67], etc. Although they work well for cloud-assisted mobile apps in an one-to-one situation (offloading between only a mobile-cloud pair), they are not suitable for distributed apps. For example, a distributed app is run by many pairs of computing entities (mobiles and their cloud surrogates). In such cases, offloading decision made in one mobile device can influence the whole computation process. It can affect the completion time and the battery life of other users' mobile devices. Works described in [89, 37, 57] considered multi-device setup where the devices are connected to the same base station or access point. [89, 37] employed game theory based approaches to minimize the energy usage in mobile devices and at the same time optimize the use of the shared communication channel. However, these works did not consider a distributed computation scenario where multiple devices are running related/dependent computations to collaboratively produce an end-result.

2.6 Chapter Summary

In this chapter, we have discussed about existing photo tagging frameworks for smartphones and their incapability of exploiting sensing potential in an efficient and real-time manner. Next, we have discussed existing works on sensing which attempted to utilize multiple sensors collectively. We have also discussed previous works related to collaborative computation over mobile devices. Finally, we presented existing works on computation offloading.

CHAPTER 3

TAGPIX: AUTOMATIC REAL-TIME LANDSCAPE PHOTO TAGGING FOR SMARTPHONES

TagPix utilizes motion sensors such as orientation sensor (i.e., accelerometer and magnetic field sensor) embedded in smartphones along with a landmark database to generate relevant and meaningful tags for landscape photos captured by the mobile camera.

3.1 TagPix Overview

To illustrate how TagPix works, let's consider the scenario from Figure 3.1, in which the user zooms in and takes a photo of Tour Eiffel. TagPix's goal is to determine one tag that best represents the photo. We assume a place-tag database stored on the phone; alternately, this database can be accessed over the Internet. This database provides landmark tags and their associated locations.

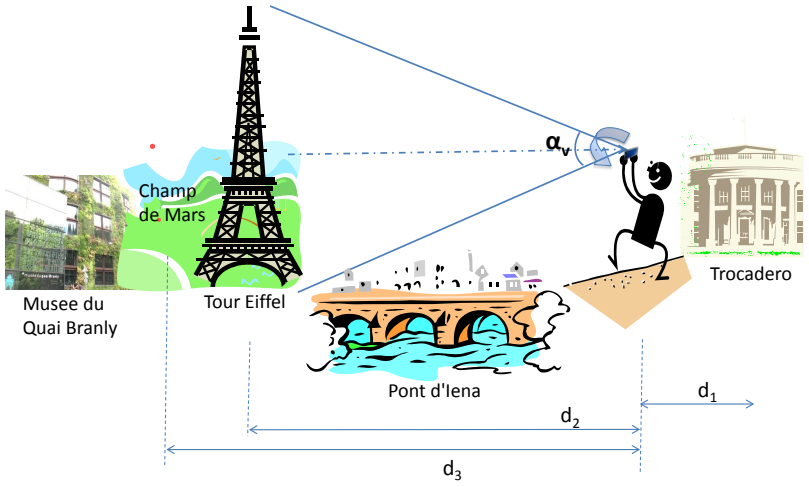


Figure 3.1 How to select the correct tag for a photo?

First, TagPix reads the GPS location of the user and retrieves the nearby landmark tags. If it just considers the distance between these landmarks and the user, the app would end up with the wrong tag (i.e., Trocadero which is the closest, but it's behind the user). Therefore, TagPix uses the orientation sensors on the phone to determine the angle of view for the photo (only the vertical angle of view, α_V, is shown in the figure). This operation helps to eliminate the Trocadero and Pont d'Iena tags; the first is at the back of the user, and the second is below the angle of view as the user is on a hill and zooms in the Eiffel Tour.

Next, we would like to compute the distance to the object in the camera focus and select the closest tag. Unfortunately, distance estimation based on standard optics formulas can be done only for objects within a few feet of the camera. This is because the phones come with cameras and lenses that have minimal ability in sensor size and focal length (e.g., the digital zoom value is unusable in optics formulas, and the lens generally has a fixed and small focal length). Instead of Euclidean distance, TagPix computes the angular distance between the direction of the camera focus and each of the remaining tag-identified landmarks. This eliminates Musee du Quai Branly, and keeps Tour Eiffel and Champ de Mars which both have 0 angular distance in this example. In our basic solution, TagPix chooses Tour Eiffel because it's closer to the user position. In many situations this method provides very good accuracy, but it's still possible to tag incorrectly in a few situations (e.g., the landmark with the lowest angular distance is not the one in focus).

Additionally, if the user is willing to do a bit of extra work, TagPix is able to estimate the distance to the object in focus with good accuracy and solve this problem. We devised two methods to do it: in one, the user is required to first point to the base of the landmark being photographed, and then TagPix uses simple trigonometric formulas to compute the distance. In the other, the user is required to take two photos of the landmark from two positions, which can be very close to each

other (e.g., 10 feet). With these methods, we can guarantee the correct tag in our example, e.g., Tour Eiffel.

As a final observation, let's note that determining the tag through this distance estimation method can work with or without the angular distance. However, by combining both of them TagPix achieves the highest accuracy. For example, two tags could be at the same Euclidean distance, but one could be eliminated using the angular distance.

3.2 TagPix System Design and Implementation

Figure 3.2 shows the system architecture for TagPix, which consists of three layers. The first is the Sensors layer, which is responsible for managing the sensors, e.g., when to turn them on and off, how to collect data from them, etc. The second is the Data Aggregator and Manager layer, which is responsible to do pre-processing and adjustment work (e.g., angular adjustment for magnetic declination) as well as merging operations on the collected sensor data. The third layer is the Tag Suggestion Generator, which is responsible for tagging decisions and providing suggestions to users. In other words, it computes the distances (angular and Euclidean) between the tagged landmarks and the object in the photo, and then selects the best matched tag (or tags if the user configures the app for multiple tags).

Algorithm 1 presents the pseudo-code for TagPix. In the following, we describe each of the instructions in the pseudo-code based on our Android implementation; we used Android 2.3.3 (Gingerbread), but TagPix works on newer Android versions as well.

3.2.1 Acquiring Location Data

Once the user pushes the camera button, TagPix queries Android for the best location provider based on the accuracy of existing sensor hardware, environment, energy level,

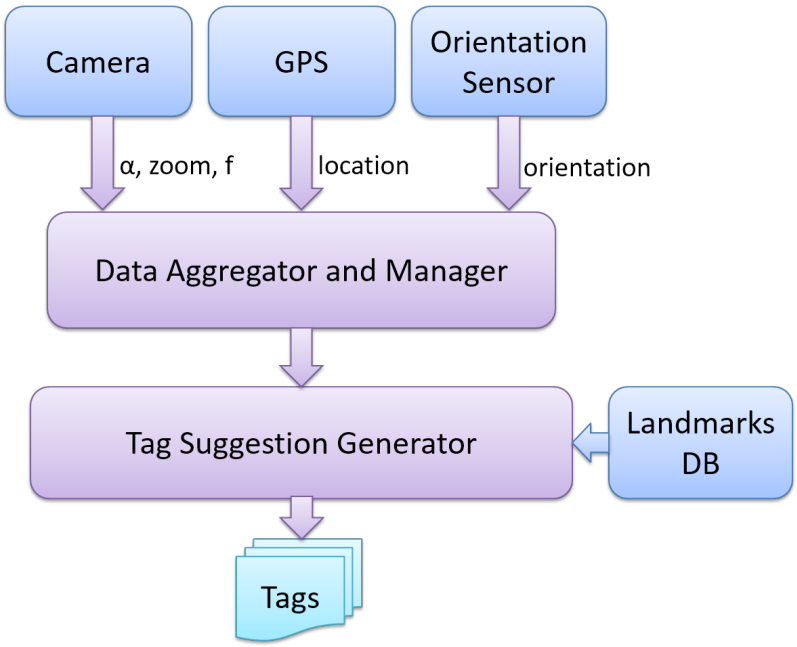


Figure 3.2 TagPix system architecture.

etc. We find that most of the time it provides GPS as the best provider. But, in case the GPS signal is not available with a decent accuracy or the battery level is too low to access the GPS sensor, Android provides other location sensing options such as WiFi, 3G, etc. TagPix registers a location listener to that location provider, listens for a "location change" event, and requests updates for a change in location of at least 1 meter. Whenever the "location change" event fires, TagPix fetches the longitude, latitude, and altitude for the current location. The location API also provides a way to cache the previously known location data in case it is unable to resolve the current location. As long as the camera is open, the location provider is updated once the user moves 20m from the previous place.

3.2.2 Fetching the Landmark List

Once the current location data is acquired, TagPix inquires the place-tag database about the nearby landmarks list. TagPix uses Google Places API [54] to retrieve tags associated with places and landmarks. In this way, TagPix is able to access a

Algorithm 1 TagPix Pseudo-code

```
1: gps = getGPSData()
2: landmarkList = callGoogleLocationAPI(gps, tagsNumber)
3: orientData = getOrientationData()
4: cameraState = getCameraInfo()
5: aov = angleOfView(cameraState)
6: coordinateRange = calculateTargetedArea(aov)
7: tags = angularDistFilter(landmarkList, coordinateRange, cameraState,
   orientData, AngThreshold)
8: if useEuclideanDist == true then
9:     angles = getUserLandmarkAngles()
10:    euclDistance = getEuclideanDist(angles, UserHeight, UserMovementDist)
11:    tags = euclideanDistFilter(tags, euclDistance)
12: end if
13: return tags
```

relatively up-to-date place-tag database. While TagPix sends the user location to the database, it does not send the photos; thus, it protects user privacy to a significant extent and reduces the bandwidth usage. To avoid sending the user location to the server, TagPix could store such a database on the phone and update it periodically. For example, the GeoNames project [50] provides a database of 8 million named places from all over the world and is available for off-line use.

Resolving the location from GPS and then acquiring the places list by calling the Google API is a fairly slow process compared to the other calculations, which may degrade the user experience. Therefore, we create an asynchronous task to invoke the Google Places API before TagPix starts any other computation. Thus, the places list is fetched in the background while TagPix calculates the phone orientation.

Since it is not clear what radius value to set for "nearby", we choose to ask for a fixed number of tags ordered by proximity to user location. The call returns a JSON array containing the information of interest (i.e., name/tag and location) about nearby landmarks and popular places. The JSON array is then converted to Java objects to be used in the app.

3.2.3 Calculating the Orientation

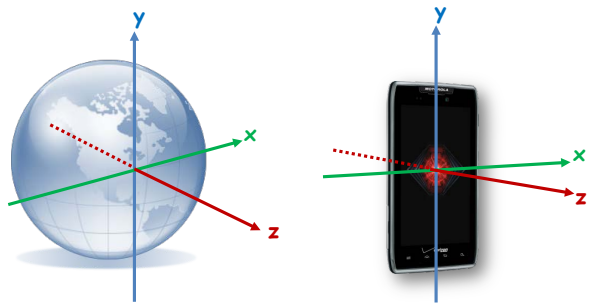


Figure 3.3 The world geomagnetic coordinate system vs. the phones coordinate system.

To determine the phone orientation, TagPix uses the software-based orientation sensor in Android, which derives its data from the accelerometer and the geomagnetic field sensor. It provides geomagnetic field strength values for each of the three coordinate axes. TagPix calculates the rotation and inclination matrices that allow us to compute the rotation around the three axes. However, there is one additional step before this computation: the app has to align the phone's coordinate system to the World Coordinate System (as shown in Figure 3.3). In the phone coordinate system, X and Y axes are tangents to the ground location of the user and Z is perpendicular on X-Y plane. TagPix uses the two matrices to find the azimuth angle (angle around the Z axis) and the tilt angle which are used to filter landmarks in a later stage.

3.2.4 Calculating the Angle of View

Calculating the angle of view (AoV) α is a standard procedure used in Optics and Photography. As shown in Figure 3.4, the AoV is defined as the angular extent of a given scene that is imaged by a camera and is computed using the following equation:

$$\tan\frac{\alpha}{2} = \frac{\frac{d}{2}}{S_2}; \tag{3.1}$$

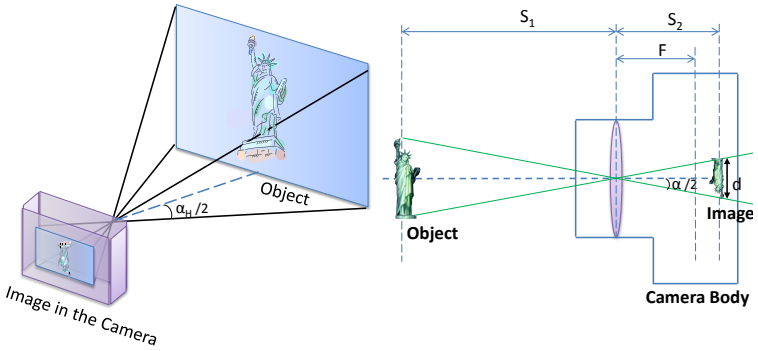


Figure 3.4 Angle of view.

For a well-focused photo, S_2 equals the focal length of the lens (F). When calculating the horizontal AoV (α_H in the figure), d is the camera sensor frame width (w); when calculating the vertical AoV, d is the camera sensor frame height (h). These sensor values are retrieved from the Android Sensor API. For example, the horizontal AoV will be:

$$\alpha_H = 2 \arctan \frac{w}{2F}; \tag{3.2}$$

3.2.5 Calculating the Angular Distance and Selecting Tags

Using the AoV and the orientation angles, TagPix computes the angular distance between the object in focus and the retrieved landmarks. This operation allows TagPix to filter out many tags which are not located within the region targeted by the camera. Then, TagPix orders the remaining tags as a function of their angular distance (i.e., the closest to the object in focus is the best).

Figure 3.5 shows the angles used to calculate the angular distance γ. The equation is:

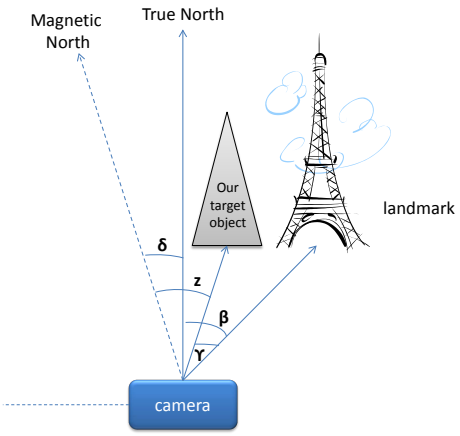


Figure 3.5 Angular distance.

$$\gamma = \beta - z + \delta \tag{3.3}$$

The angle z is the azimuth angle calculated from the orientation data. The angle β is the GPS bearing between two coordinates, which is defined as the horizontal direction of travel of the phone from the first coordinate to the second one. We can consider the bearing as the angle created by the connecting line of the two coordinates and the geographic Y axis. The GPS bearing is calculated using true North Pole as reference. However, for Android devices, the orientation sensor works with the magnetic north pole as a reference.

The difference among the true north and the magnetic north is known as the magnetic declination, δ, and varies from place to place; it also varies with time. The National GeoPhysical Data Center [94] provides a calculator to know the exact value of δ for a particular place. Android's SDK offers a specific class, GeoMagneticField, which provides utility methods to adjust the magnetic declination with orientation sensor data.

In the ideal case, γ is 0 (i.e., our targeted object is aligned with the landmark). But, it is difficult to achieve such accuracy in many situations. Therefore, we use a threshold angle value θ to select the most appropriate tags: all the landmarks which have an angular distance $\gamma < \theta$ are returned as the suggested tag list. Keeping the threshold angle very low would return only the most accurate tags.

3.2.6 Estimating Euclidean Distance

As we will show in Section 3.3, using only angular distance produces good results in practice with the major advantage of not requiring any action from the users. However, if the user is willing to do some simple actions, TagPix can further improve the tagging accuracy by estimating the Euclidean distance to the targeted object. For example, if two landmarks A and B fall within the angle θ and both have the same angular distance, the estimated Euclidean distance can be used to choose one of them (i.e., the one closer to the targeted object).

We devised three methods for estimating the distance: one requires the user to just point to the base of the landmark after taking the photo; the other two require the user to move short distances and take two photos. The first method is more convenient for users, but it works well only for objects located in the immediate vicinity of the user. At the cost of a slight inconvenience, the other methods achieve better accuracy for objects located farther away.

Method 1 This method does not require the user to move or take multiple photos. All it needs is user's height (which is inputted by the user when the app is configured) and the camera to be aimed at the base of the landmark after taking the photo. In this way, TagPix obtains the angle x (as shown in Figure 3.6) from the orientation sensor data, and then it computes the distance:

$$d = h * tanx \tag{3.4}$$

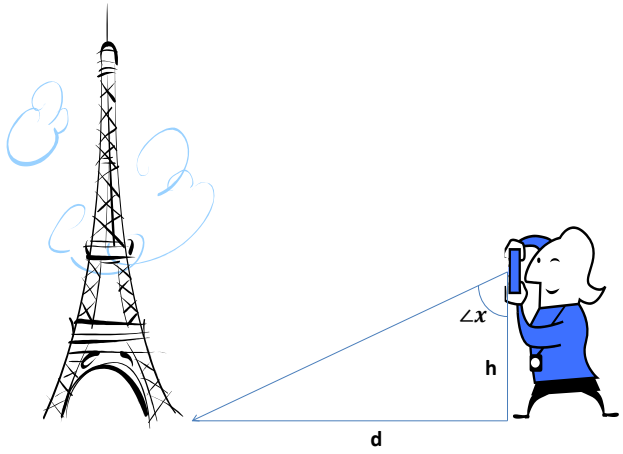


Figure 3.6 Method 1 to estimate the object Euclidean distance.

Method 2 This method needs two photos taken from two different locations, as shown in Figure 3.7. First, the user needs to aim the camera at the top of the landmark, and TagPix measures the angle of inclination β from the orientation sensor data. Then the user needs to move forward for a short distance (s) and repeat the procedure; TagPix measures the angle α. The longer the distance, the higher the accuracy of the method will be. From [88], we know that there is a high correlation between step length and height of a person. As we already know the height of the user, the app will just ask the user to move a fixed number of steps forward (e.g., 10). The distance is then calculated using this equation:

$$d = \frac{s * \tan\beta}{\tan\alpha - \tan\beta} \tag{3.5}$$

Method 3 This method can be used instead of method 2 if the user cannot move forward (e.g., she would end up in road traffic). In this method, the user moves sideways, and TagPix measures the angles α and β as illustrated in Figure 3.8. The distance is calculated as follows:

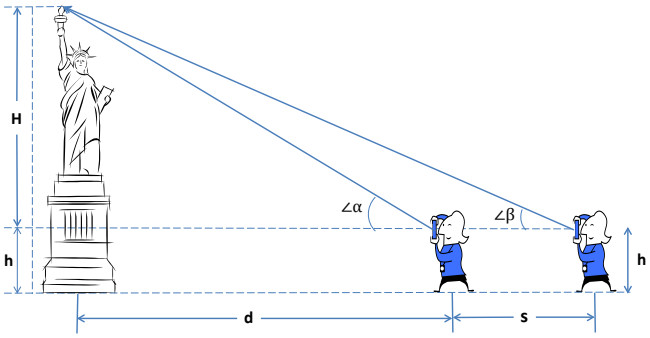


Figure 3.7 Method 2 to estimate the object Euclidean distance.

$$d = \frac{s}{\tan\alpha + \tan\beta} \tag{3.6}$$

We found that measuring α and β accurately is problematic due to the practical limitation of GPS and orientation sensor. A better solution is to calculate the angle created at the target by s (let us call it θ, measured in radians) from the two bearing values corresponding to the two locations of the user, which is comparatively more accurate. The user only needs to move about 10 meters to achieve good accuracy. The distance is then computed as follows:

$$d = \frac{s}{\theta} \tag{3.7}$$

We have tested all three methods by measuring the distance of real world objects. Method 1 has the lowest accuracy among them (about 20% average error), and is usable only for distances less than 40 meters. Method 1 also has problems with landmarks situated in elevated places. For method 2, the average error is 15%. For method 3, it is 14%. Both these methods maintain this level of accuracy for

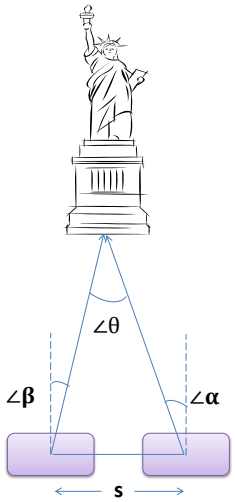


Figure 3.8 Method 3 to estimate the object Euclidean distance.

any practical distance measurement. Finally, methods 2 and 3 can adjust their calculations for elevated landmarks if we know the elevation of the landmark.

3.3 Evaluation

TagPix was tested with 6 users and evaluated for many places and landmarks in 8 cities in USA: New York City; Newark, Harrison, Kearny, Jersey City, Morristown, and Parsippany-Troy Hills in New Jersey; and Columbus in Ohio. For ground truth, we used user's feedback in real time to verify the generated tags. For incorrectly generated tags, the users entered the right tags. We have tested the app for both very famous landmarks and more obscure ones, and it performed well in both cases. For example, we tested the app with university buildings, hospitals, churches, museums, libraries, schools, restaurants, coffee shops, etc.

For visual illustration, Figures 3.9 and 3.10 show examples of photos correctly identified by TagPix. Figure 3.9 shows two of the test photos where there were more than one landmarks in close proximity and also in close angular distance. In the top photo, the user's target was to capture the restaurant Rio Rodizio, while two other landmarks, namely The Berkeley College (A in the figure) and The IDT Energy (B in

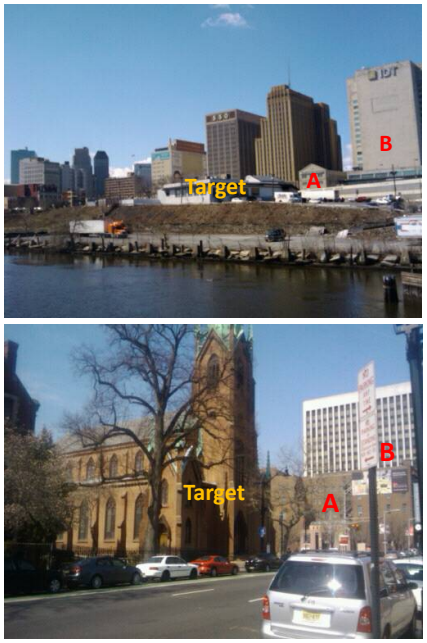


Figure 3.9 Photos taken during the test phase. Despite having other landmarks in the photos, the first tag correctly identified both of them: Rio Rodizio restaurant (top) and Saint Patrick's Pro-Cathedral (bottom).

the figure), were present in the frame. Our angular distance based calculation ranked the target landmark (Rio Rodizio) as the top suggestion, and TagPix also filtered out a false positive tag (a tag which has small angular distance, but is incorrect) which was lying behind the target object by using the Euclidean distance estimation. The photo shown at the bottom of the figure illustrates a similar scenario: TagPix ranked the Saint Patrick's Pro-Cathedral as the top suggestion, while the Newark Museum (A in the figure) and Horizon Blue Cross Blue Shield (B in the figure) are ranked second and third, respectively.

Figure 3.10 shows two of the test photos taken during night when there was very little available light to shoot. But, as TagPix does not rely on visual contents of the image, it worked perfectly for such scenarios.

Table 3.1 shows the tagging accuracy of TagPix for each user when only angular distance (angular distance threshold is set to 20 degrees) is used to select the tags.

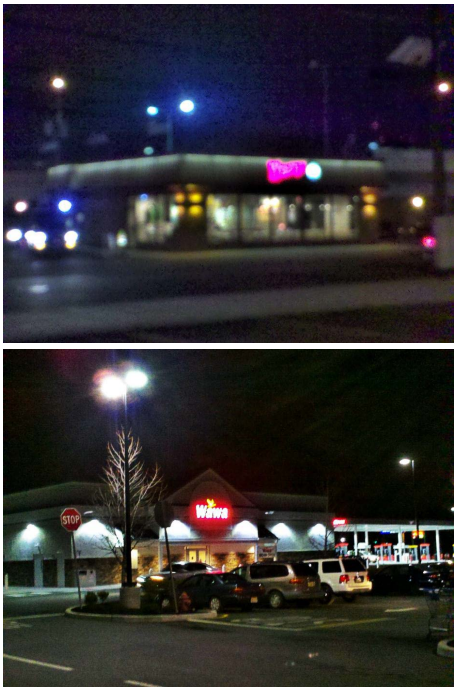


Figure 3.10 TagPix works well with blurred photos or photos taken in low light. The first tag correctly identified both of them.

This is the method with the best usability as it does not require any action from the user. The table shows the accuracy when the first suggested tag is the correct one and when the second suggested tag is the correct one. For few photos, the landmark list was not fetched in time due to technical problems (e.g., GPS signal taking too much time or lack of Internet connectivity). We do not consider these photos as usable in our experiment. Overall, the results indicate 86.52% accuracy. Among the incorrect tags, 10.11% were just outside of our angular distance threshold.

To understand the extent of improvement due to TagPix, we also compare our results with the tag list returned by Google Places, ranked by distance to the user (i.e., the closest landmark is ranked first). We calculated the accuracy of Google Places by considering any of the first two tags in the list. The last two columns in Table 3.1 show that TagPix obtains a much higher accuracy due to our algorithm that leverages the phone sensors in the process of tag selection.

Table 3.1 TagPix Accuracy and Comparison with Google Places API

User #	# of photos	1st Tag Correct (%)	2nd Tag Correct (%)	Incorrect (%)	Accuracy %	Google Places Accuracy (%)
1	7	100	0	0	100	42.86
2	67	86.57	1.49	11.94	88.06	14.93
3	7	42.85	0	57.143	42.85	28.57
4	4	75	25	0	100	50
5	1	100	0	0	100	100
6	3	100	0	0	100	33.33
Total	89	84.27	2.25	13.48	86.52	21.35

Figure 3.11 shows the overall distribution of tags. Each bar represents how many tags were in that class (e.g., first suggested tag, second suggested tag, etc.) For this graph, we considered the first four tags, while any other tags were deemed incorrect. The results show excellent accuracy for the first tag: 84.2%. Many users may actually accept multiple tags for their photos. In such a situation, if we consider all four tags, the tagging accuracy becomes 96%.

Table 3.2 TagPix Results with Euclidean Distance Estimation

Photo #	Actual Dist.	Estimated Dist.	Dist. Accuracy	Correct Tag?	False Positive	False Positive Removed
1	59	52.1	88.3	Y	1	1
2	47	50.9	91.6	Y	0	0
3	74	77.9	94.7	Y	2	2
4	57	63.7	88.3	Y	2	2
5	24	26	91.5	Y	1	0
6	41	38.2	93.2	Y	0	0
7	39	44.3	86.5	Y	2	1
8	41	36.7	89.4	N	N/A	N/A
9	101	122	79.5	Y	2	1
10	75	64.5	85.9	Y	1	1
11	62	74.1	80.4	Y	0	0
12	17	19.8	83.3	Y	1	1
13	170	120	70.5	Y	3	3
14	21	19.1	90.9	Y	0	0
15	14	15.6	88.4	Y	1	1

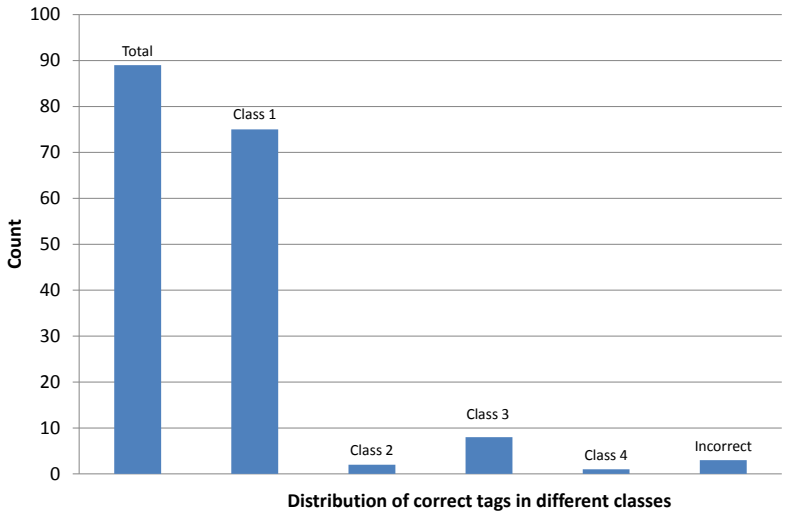


Figure 3.11 Distribution of tags in different classes.

Next, we performed experiments to quantify the accuracy benefits of using Euclidean distance estimation together with angular distance. First, we have applied the distance estimation for some of the unrecognized photos from the dataset used in the previous experiments. We found that applying this method minimizes the number of false positive tags. A false positive tag is defined as a tag that falls within the angular distance, but it is not the correct one. For example, one of our photos contains a landmark named "Burger King", but the landmark was not in the middle of the frame. It has an angular distance of 47 degrees. Whereas two other nearby landmarks (not visible in the photo due to the distance, and thus false positives), "Cosmos Bar" and "Manor Bar & Grill", had angular distance of 21 and 22 degrees, respectively. If only angular distance is considered, then both Cosmos Bar and Manor Bar come before the Burger King in the suggested list; in fact, Burger King is not even considered because we selected only the first two tags. The ground truth distance between the user and Burger King is 47 meters, whereas Cosmos Bar and Manor Bar are 188 and 135 meters away, respectively. The best of our distance estimation

methods has 86% accuracy. If we apply it to this scenario, TagPix can easily select Burger King as the best tag

Second, we have used TagPix with Euclidean distance estimation (method 3 in particular) for a new set of photos. TagPix adjusts the spatial distance +/-15% to take into account the observed average error. Specifically, we apply it for the tags returned after the filtering done using the angular distance in order to select the best one. Table 3.3 presents the results. The accuracy increases to 93%, compared to 86% when using only angular distance. Additionally, TagPix is able to remove 81% of the false positive tags.

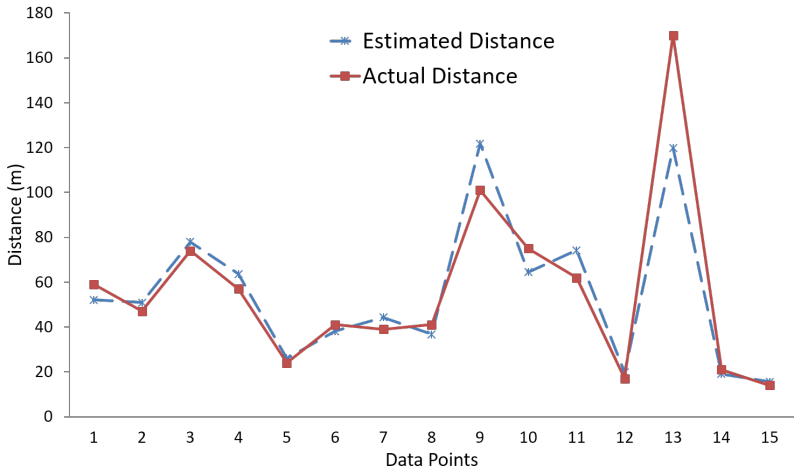


Figure 3.12 Comparison between actual and estimated distance using method 3.

For better illustration, Figure 3.12 shows the comparison between the actual distance and the estimated distance by method 3. As we can see, our estimated distances are good enough to be used in TagPix for most practical situations.

Finally, we tried to find the reason why some landmarks cannot be recognized by TagPix in a few occasions. We determined that this is mainly due to the place-tag database, which did not contain information for these landmarks. In a few other cases, the reason was the noisy nature of the geomagnetic orientation sensors. If two landmarks are very close, their correct order can be inversed due to this reason.

One of our four main goals from the beginning was to keep TagPix's resource usage minimal. We evaluated both bandwidth and energy consumption to find out whether they satisfy this goal. First, we measured the average bandwidth consumption for tagging. We found that TagPix needs 25.3KB of data to be fetched on average. After fetching the landmark list, it can be used for all pictures taken in that vicinity.

Second, we measured the energy consumption of Tagpix in its major phases (shown in Figure 3.13 and Table 3.3). We have also calculated how many pictures can be captured and tagged with a fully charged battery. A widely used power measurement tool for Android, PowerTutor [19], is used for this analysis.

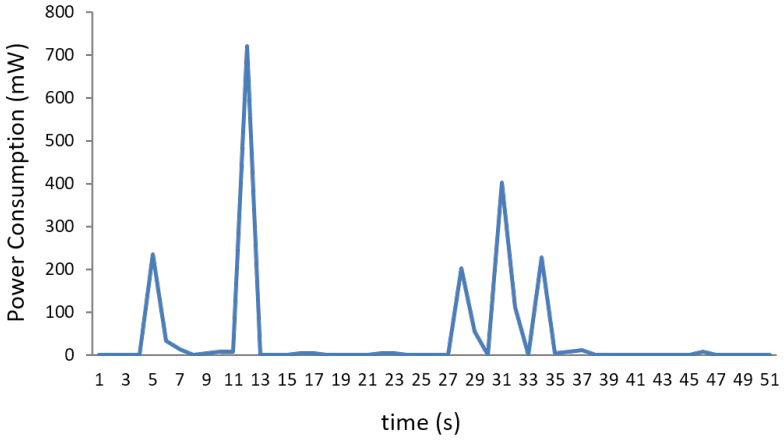


Figure 3.13 Power consumption of TagPix during different phases of a tag generation task.

Table 3.3 Energy Consumption of TagPix's Different Phases

	Acquiring location + fetching list	**Capture the photo**	**Calculating orientation + Processing**	**Tag generation + Saving photo**	**Total**
Energy consumed (mJ)	1242	319	399	223	2183
Number of photos can be tagged with full battery	15014	58457	46736	83623	8542

We have run the energy consumption tests on Motorola Droid 2 which has a 1400mAh battery, roughly equivalent to 18.65 kJ. We find that 8542 photos can be captured and tagged by TagPix with this battery. This calculation was done excluding the energy consumption by LCD which varies a lot depending on how long the user is keeping the screen on and also on the screen brightness. We have found that the LCD itself consumes 360 mW, 533 mW or 900 mW per second when used with 1%, 50% or 100% screen brightness respectively. Next, we calculated the energy consumption of TagPix including the LCD power. The user keeps the screen on for 15 seconds on average to capture a photo with auto generated tag(s). If the user uses the minimum brightness(1%), 2459 photos can be taken with TagPix with a fully charged battery. Similarly, 1832 or 1189 photos can be taken if the user uses average (50%) or the maximum (100%) brightness respectively. These numbers demonstrate that TagPix is energy efficient, and, even in the worst scenario, it can tag over 1000 photos on one charge. It is worth noting that modern smartphones such as Nexus 6 has battery capacity of 3220 mAh which is more than double of Motorola Droid 2's battery capacity. On such a device, more than 2300 photos can be tagged by TagPix on one charge.

3.4 Chapter Summary

This chapter presented a smartphone app for real-time landscape photo tagging. This framework, TagPix, leverages the phone sensors and a place-tag database to achieve very good tagging accuracy. At the same time, TagPix is lightweight as it consumes a modest amount of resources on the phones and protects users' privacy as the photos are not uploaded to servers for tagging. In the basic design, TagPix is completely automatic, thus offering very good usability. For even higher accuracy, users are asked to do simple actions such as moving a short distance and taking an additional

photo. TagPix was implemented in Android and successfully tested for many photos in different cities.

CHAPTER 4

SENTIO: DISTRIBUTED SENSOR VIRTUALIZATION FOR MOBILE APPS

This chapter presents Sentio, a personal virtual sensor system (*PVSS*), which enables apps to seamlessly utilize the collective sensing capabilities of the user's personal sensors. As shown in Figure 4.1, Sentio presents apps with a PVSS abstraction — a collection of virtual sensors formed on top of smart devices owned by the user and IoT devices located in the proximity of the user. Virtual sensors provide a unified interface to local and remote sensors for app development. Multiple virtual sensors could be leveraged to provide composite virtual sensors, which perform sensor aggregation and fusion.

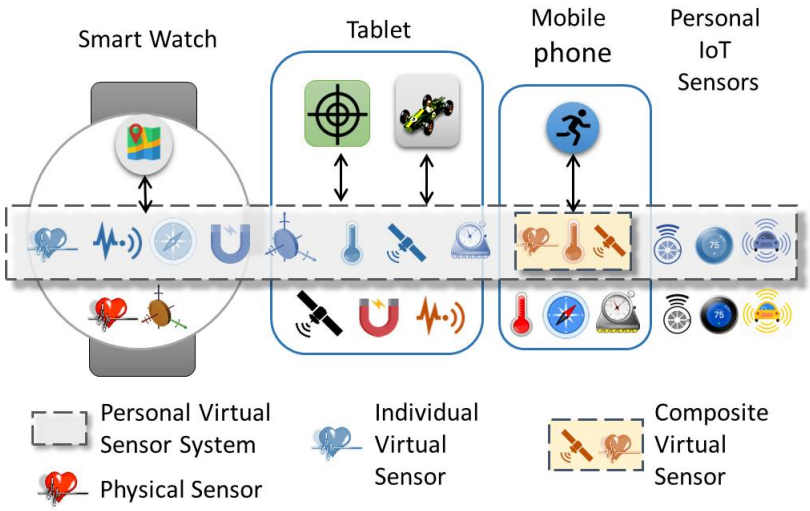


Figure 4.1 Sentio creates a personal virtual sensor system across different devices. A health monitoring app on the phone uses Sentio to access a composite sensor built on top of three virtual sensors that abstract physical sensors: a heart rate monitor in the smart watch, a GPS in the tablet, and a temperature sensor in the phone.

4.1 Overview

Sentio is motivated by several use cases of collective personal sensing. First, apps running on a mobile device may need to access personal sensors located on other devices because these sensors are not locally available. For example, a health monitoring app running on the phone may need to access a heart monitoring sensor from a smart watch.

Second, apps may need to access the same type of sensors from several devices to cross-validate the data. For instance, many complex and overlapping human activities are difficult to recognize [73]. An activity recognition app running on the phone may use accelerometers from both the phone and the smart watch to improve the recognition accuracy.

Third, apps may want to choose the "best" sensor for a certain sensor type from several available sensors located on different devices. For example, in order to save battery power on the phone, an app running on the phone may access a GPS sensor from a tablet in the backpack carried by the user. However, if the user leaves the backpack in the car and continues on a hike, the app should transparently switch from the tablet GPS to the phone GPS.

Our aim is to simplify app development by providing seamless access to personal sensors, no matter on which personal device an app runs. Therefore, Sentio creates a personal virtual sensor system (PVSS) to facilitate app access to physical sensors located on different devices. A virtual sensor behaves as a local and universal sensor abstraction. A virtual sensor has a type (e.g., accelerometer, temperature) and can be mapped to physical sensors of the same type located on different devices. Composite sensors can be built on top of several virtual sensors to perform sensor fusion or aggregation. Sentio provides default sensor data fusion methods, but the programmers can specify their own methods. A simple example of a composite sensor is an accurate accelerometer that uses data from all the virtual accelerometers

currently available in PVSS. A more complex example is a "climbing" sensor that combines readings from a heart rate monitoring sensor and a barometric pressure sensor to warn climbers when they should rest or drink more water at high altitude.

If an app needs only one sensor of a certain type, Sentio uses a heuristic algorithm to select or switch to the "best" available sensor of that type, as a function of energy consumption, latency, or sensor accuracy.

Let us note that physical sensors available in PVSS change over time: the user may not carry certain devices at a given time and, thus, the physical sensors embedded in these devices are removed from PVSS. As long as there is at least one physical sensor of a certain type, the virtual sensor of that type will be available.

Sentio also supports code offloading to the cloud. Let us consider, an activity recognition task running on the smart phone and using the virtual accelerometer mapped to the local accelerometer in the phone. At a later time, the app offloads the activity recognition task to the cloud. Since, this task needs to access the accelerometer sensor, Sentio creates a virtual accelerometer in the cloud and seamlessly maps it to the physical accelerometer of the phone. The offloaded code in the cloud can access the accelerometer sensor in the same way it was doing from the mobile. Hence, the mapping between the virtual sensors and the physical sensors remain transparent to the app, creating the illusion that these sensors are always attached to the local device.

Sentio is designed and implemented as a distributed middleware, with components on all personal smart devices of a user. An instance of the middleware on a device manages the physical sensors of the device and executes tasks such as sensor registration, sensor mapping, and data collection. More importantly, each instance is in charge of fulfilling the requests from instances on other devices to retrieve sensor data. Sentio exposes the same high-level API to apps running on any smart device owned by the user; the same API is available for apps offloaded to the

cloud. Sentio shields the programmers from all the low-level communication and sensor management details, and is able to select the sensor that provides the best latency, accuracy or power savings for a given app and context.

Sentio uses a computing entity (CE) [31] residing in the cloud for sensor management and for enhancing the computational and storage resources available to mobile devices. Each user has her own CE, which can be a virtual machine or a container. CE maintains the PVSS sensor registry and the current state of all virtual sensors exposed by Sentio to apps. In addition, CE is used to provide support for computation offloading to the cloud. App code offloaded to the cloud can access virtual sensors through a Sentio instance running in the CE.

In Sentio, one device (generally, the smart phone) is considered the primary device and all the other devices are considered as secondary devices. If the CE becomes unreachable due to network problems, the primary device uses ad hoc networking to perform PVSS management. To be able to do this switch, CE and the primary device periodically synchronize the PVSS state with each other.

Sentio can use state-of-the art solutions for security and privacy. However, we do not address these issues in this paper, as our current design is focused on system and middleware challenges.

4.2 System Design

As shown in Figure 4.2, Sentio consists of a set of middleware instances running on smart devices and the cloud. To create and use virtual sensors, apps interact with their local middleware instances by calling the API functions implemented in the Sentio library. The virtual sensors created on a device and the physical sensors embedded in the device are managed by its local middleware instance. The access to virtual sensors is also handled by the middleware instance, which either forwards the requests to the corresponding physical sensors if they are located on the device or

forwards them to the middleware instances on the devices where the physical sensors are located.

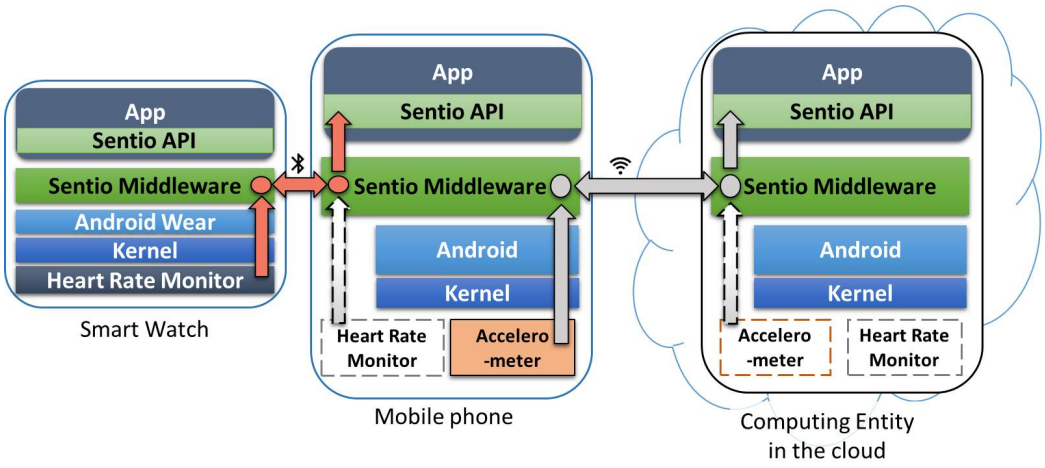


Figure 4.2 Sentio's distributed middleware works on top of device OSs and provides a unified view of PVSS via Sentio API. Solid arrows show the accesses to physical sensors, and dotted boxes show accesses to virtual sensors.

4.2.1 Sentio API

Sentio API follows an event-driven and callback-based asynchronous design. For each Sentio API call, a program needs to provide a callback function. The API call sends a request to the Sentio middleware about the desired operation, and returns immediately when the request is sent. The middleware handles the request and returns the necessary data to the app by invoking the callback function. This asynchronous API design fits well with the architecture of Sentio, in which apps and the Sentio middleware run in different processes and exchange information using events and messages.

As shown in Tables 4.1 and 4.2, Sentio API methods are mainly designed to 1) query about available sensors in the PVSS, 2) specify callback functions for receiving data from the middleware, 3) create virtual sensors (individual or composite), and

Table 4.1 List of Sentio API Methods

API	Description
getAvailableSensorList (SensorListListener listListener)	Fetches the list of available sensors in the sensor registry. The list is received by the callback function of the "listListener" interface.
registerSensor (SentioSensor sensor, SensorListener listener)	Registers the "sensor" from the PVSS. Sensor data will be received by the "listener".
unregisterSensor (SentioSensor sensor)	Unregisters the "sensor".
SensorListListener callback interface (example shown in code listing 4.1)	
onSensorListAvailable (Map<SentioDevice, List<SensorType≫ sensors)	This callback function receives the mapping between the device and its list of sensors from the middleware.
SensorListener callback interface (example shown in code listing 4.2)	
onSensorDataAvailable (SensorDataBundle data)	This callback function receives sensor data from the middleware.
FuseAction interface (used for writing custom fusing functions)	
SensorDataBundle fuseData (SensorDataBundle... data)	Programmers write custom fusing functions inside this interface method. This method receives a SensorDataBundle object from each sensor in the current composite sensor. Programmers apply the fusing function to these data, and it returns a single SensorDataBundle object.

4) register and unregister sensors. We use five examples to explain how an app uses these APIs to interact with the Sentio middleware.

The first example (Code listing 4.1) shows how to query the middleware for the list of available sensors in the PVSS. The main API call for this purpose is *getAvailableSensorList* (line 2), after the program gets an instance of the *SentioManager* (line 1). It requires an instance of *SensorListListener* callback interface established to handle the results, which is passed as a parameter to the API. In the *SensorListListener* instance, the callback function *onSensorListAvailable()* (line 4) receives a map containing the mappings of each device and a list of sensors on the device (e.g., <device1, sensor list>, <device2, sensor list>, ...).

The second example (Code listing 4.2) explains how a program receives data from a virtual sensor. To access a virtual sensor, a program must create an instance of

Table 4.2 SentioSensor Builder API

API	Description
SentioSensor builder API (example shown in code listing 4.3)	
addSensor (SentioDevice device, SensorType type)	Adds the sensor of type "type" from the device "device" to the virtual sensor.
addSensor(SentioSensor sensor)	Adds the virtual (composite/individual) sensor "sensor" to the current virtual sensor.
addSensor (SensorType type, SensingMode mode)	Depending on the "mode" parameter, this API call picks the best sensor of "type" type from the sensor registry. The mode can be POWER_SAVING, REAL_TIME, or ACCURACY. If no mode is specified, a default mode is used.
samplingRate (SamplingRate rate)	Defines the sampling rate or frequency. The value of "rate" can be one of NORMAL, UI, GAME, FASTEST.
addListener (SensorListener listener)	Specifies the listener which will receive the sensor data.
fuse (FusingMode mode)	Specifies the fusing mode. The predefined modes are: Combine, Average, Competitive, Complementary.
fuse (FuseAction fuseAction)	Takes a FuseAction interface as parameter which enables the programmer to define custom fuse functions.

SensorListener callback interface (line 1) and implement the *onSensorDataAvailable()* callback function in the instance (line 3). After the program registers for the virtual sensor with the *SensorListener* instance, the Sentio middleware will automatically pass sensor data as *SensorDataBundle* objects to the callback function. Information about the sensor (e.g., source device, sensor type, etc) is also included in the *SensorDataBundle* object.

The third example (Code listing 4.3) shows how to build a composite sensor. This composite sensor is for environment monitoring. It consists of a magnetic field sensor in the ACCURACY sensing mode (added on line 2) and a GPS sensor in the $POWER\_SAVING$ sensing mode (added on line 3). Since our sensor building API follows the Java builder design pattern [30], programmers can add an arbitrary number of sensors by repeatedly calling *addSensor()*. An individual virtual sensor can be built by calling *addSensor()* only once. At the same time, another polymorphic

```
1 SentioManager sentio = SentioManager.getInstance();
2 sentio.getAvailableSensorList(new SensorListListener() {
3    @Override
4    public void onSensorListAvailable(Map<SentioDevice, List<SensorType>>
         sensors) {
5       //TODO: use sensor list
6    }
7 });
```

Listing 4.1 Using the *getAvailableSensorList()* function.

```
1 SensorListener sensorListener = new SensorListener() {
2    @Override
3    public void onSensorDataAvailable(SensorDataBundle bundle) {
4       //TODO: process sensor data
5       // Sensor type: bundle.getSensorType()
6       // Sensor source: bundle.getSourceDevice()
7       // Sensor data: bundle.getSensorData();
8    }
9 };
```

Listing 4.2 Code for a sensor listener.

version of *addSensor()* can accept a composite sensor as the parameter. This can be leveraged to build a composite sensor from both individual and composite virtual sensors.

The sensing modes specified in *addSensor()* calls can direct Sentio to select appropriate sensors to satisfy the request. If a program does not specify a mode, Sentio will balance accuracy, speed, and energy consumption and will select sensors based on a method to be described in Section 4.2.2. Line 4 specifies the sampling rate of the virtual sensor. *SamplingRate* is an enum structure with four possible predefined values: NORMAL, UI, GAME, FASTEST. Android's sensor framework also uses these same four sampling rates. The NORMAL sampling rate indicates a period of 200 ms between two sampled data points. UI and GAME correspond to 60 ms and 20 ms sampling periods, respectively. The FASTEST sampling rate directs the middleware to sample data as frequently as possible.

```
1 SentioSensor sensor = SentioSensor.newBuilder()
2             .addSensor(SensorType.MAGNETIC_FIELD, SensingMode.ACCURACY)
3             .addSensor(SensorType.GPS, SensingMode.POWER_SAVING)
4             .samplingRate(SamplingRate.NORMAL)
5             .addListener(sensorListener)
6             .fuse(FusingMode.COMBINE)
7             .build();
```

Listing 4.3 Code for building a composite sensor.

```
1 SentioSensor sensor = SentioSensor.newBuilder()
2             .addSensor(device1, SensorType.ACCELEROMETER)
3             .samplingRate(SamplingRate.GAME)
4             .addListener(sensorListener)
5             .build();
```

Listing 4.4 Code for building an individual virtual sensor by manually specifying the source device.

Line 5 specifies the *sensorListener* callback for receiving sensor data, and line 6 specifies the fuse function to be used for fusing data. A few predefined fusing functions are provided by Sentio, such as Combine, which combines data points together and delivers the results in an array. Programmers can also apply a custom fuse function.

In the fourth example (Code listing 4.4), a game running on a tablet uses the accelerometer on a smart watch as the sensor to control the game. On line 2, the game builds a virtual sensor by specifically selecting an accelerometer in the smart watch *device1*. This overrides the automatic selection in Sentio.

A program must register a sensor in order to access it and unregister the sensor after it stops using the sensor. The last example shows how *registerSensor()* and *unregisterSensor()* are used to register and unregister a sensor on line 2 and line 5 of Code listing 4.5, respectively.

```
1 //register a virtualized gyroscope
2 sentio.registerSensor(sensor, sensorListener);
3 ...
4 //unregister the gyroscope from device1
5 sentio.unregisterSensor(sensor);
```

Listing 4.5 Code for registering and unregistering sensors.

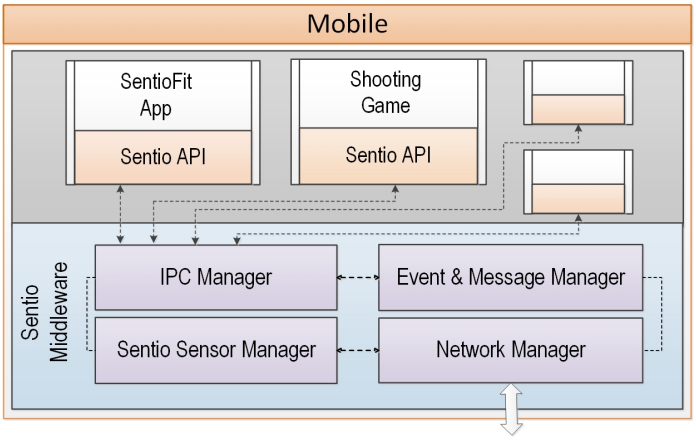


Figure 4.3 The Sentio middleware architecture.

4.2.2 Sentio Middleware Design

The Sentio middleware resides on each device as an independent process, referred to as a Sentio middleware instance. The middleware has four major components, as shown in Figure 4.3.

IPCManager is in charge of the inter-process communication (IPC) with Sentio apps. When an app initiates a SentioManager object (e.g., line 1 of Code listing 4.1), an IPC channel is establish between the app and the middleware instance. The middleware keeps record of the channel and uses it later for routing data to this app. When the app stops and unbinds itself from the middleware, the channel and the channel information are released.

The execution of the Sentio middleware is driven by various events and messages. For example, every Sentio API call is translated to a corresponding low level event, and sensor data is sent in the form of messages. The **Event and Message**

Management component ensures that both events and messages are queued after they are received by the middleware, and are dispatched later by queue managers.

The **NetworkManager** handles network communication between the middleware instances running on different devices. By default, Sentio uses the internet for connectivity. However, if devices are disconnected from the internet but are physically located in nearby proximity, The NetworkManager works in a P2P fashion using techniques such as WiFi-direct or Bluetooth.

The core of the Sentio middleware is the **SentioSensorManager** component, which manages the local physical and virtual sensors on a device. When Sentio is initialized, SentioSensorManager runs a sensor discovery process to form a list of physical sensors on the device (local sensor registry), which includes the specifications of the sensors, such as the precision/resolution, the lowest and highest sampling rates supported, and power consumption rating. It sends the list to the cloud entity, where a global list of sensors in the whole Sentio system (global sensor registry) is compiled. The global sensor registry will used to find appropriate sensors to build virtual sensors. The SentioSensorManager component that runs on the cloud entity manages the global sensor registry, and is responsible for discovering and managing personal IoT sensors placed in cars, houses, appliances, etc.

During execution, when a request is received to register a virtual sensor, SentioSensorManager needs to identify the physical sensors that meet the required criteria and map them to the virtual sensor. It does so by searching the global sensor registry fetched from the cloud. To accelerate the searching, a copy of the global sensor registry is maintained on each device and synchronized with the cloud. If the desired sensors can be found locally, the SentioSensorManager will finish the mapping from the sensors to the virtual sensor, and deliver sensor data to the requesting app. If the desired sensors are located on other devices, the local SentioSensorManager will contact the corresponding SentioSensorManagers on those devices in order to

get access to those sensors. It will also forward the mapping information to these SentioSensorManagers, which will be used to route and deliver sensor data from the physical sensor to the requesting app. The sensor selection method is described in detail in Section 4.2.2.

SentioSensorManager also monitors context changes in the device, such as battery level dropping below a threshold and network connection changes. If there are virtual sensors on other devices built based on the physical sensors on the device, the SentioSensorManager on the device will also notify the SentioSensorManagers managing the virtual sensors about the status change. Based on the changes, they may switch the physical sensors used for the virtual sensors. The switches take place transparently to apps. It is worth noting that some apps may want to get data only from a physical sensor which matches the requested sensing criteria. In such cases, if the second best sensor (selected for switching) does not meet the specification (e.g., sampling rate, accuracy, etc.), then this automatic switching can be disabled and an alert can be sent to the app mentioning that a suitable sensor is unavailable. This type of behavior can be specified by the programmer during the sensor building phase. It can be supported with a small change in the current API.

Sensor Selection When there are multiple sensors available for building a virtual sensor, the SentioSensorManager needs to select a sensor. It makes the selection based first on the sensing mode specified by the program (Section 4.2.1). Specifically, if the sensing mode is $REAL\_TIME$, Sentio considers two factors for each sensor. One is the highest sampling frequency that the sensor can support. The other is the latency for the sensing data to arrive to the app, which mostly consists of network communication delay. If the sensing mode is $POWER\_SAVING$, Sentio considers the energy consumption of the sensor and the energy for transferring the sensing data over the network. It also considers the battery level of the host device of the sensor

to avoid selecting a sensor on a device with a low battery level. If the sensing mode is $ACCURACY$, Sentio picks the sensor with the highest resolution.

If the program does not specify a sensing mode, Sentio makes the selection by balancing a few factors, including speed, accuracy, and power saving. Specifically, Sentio computes a score for each sensor based on the formula shown in Equation 4.1, and selects the sensor with the highest score.

$$score = w_a * \frac{1}{precision} + w_d * \frac{1}{min_{delay} + latency_{comm}} + w_p * \frac{battery_d}{power_s + power_{comm}} \tag{4.1}$$

In Equation 4.1, *precision* represents the smallest change a sensor can detect. For example, the accelerometers in Nexus 6 and Samsung Gear Live watch are 0.039 m/s^2 and 0.15 m/s^2, respectively. min_{delay} and $latency_{comm}$ refer to the supported minimum sampling period of the sensor and the communication latency to receive data from the sensor; $power_s$ indicates the power consumption rating of a sensor; and $battery_d$ indicates the remaining battery of the host device. $power_{comm}$ refers to the power consumption for transporting data from this sensor to the requesting device. w_a, w_d, and w_p are the weights assigned to the accuracy, delay, and power consumption factors, respectively. In our implementation, we give equal weights to these factors.

Sensor Data Rate Control In many apps, it is important that sensor data arrives at a steady rate. Sentio uses sensor data rate control to ensure that virtual sensors are able to report data at steady rates. A physical sensor cannot report data at a steady rate. Various mechanisms, such as filtering, are used to make the data rates steady for some sensors. For a virtual sensor backed by a physical sensor on another device, the data transfers over network can make the data rate even more unsteady. To get

steady data rates, Sentio uses a rate controller, which buffers data points arriving early and uses extrapolation to project data points that arrive late.

Managing Composite Sensors In Sentio, building a composite sensor starts from creating component virtual sensors based on the requirements specified in *addSensor* calls (refer to Code Listing 4.3). Then, the mapping between the composite sensor and the virtual sensors are established and maintained by the SentioSensorManager. To keep feeding the app with the fused data, a buffer is used to stage the data from component virtual sensors, since they may have different data rates. Every time when enough data is accumulated to calculated a fused data point, the fusing function is invoked. Since the data rate of a composite sensor is affected by the data rate of each component virtual sensor, it can be more unsteady than that of an individual virtual sensor. Therefore, even if we have applied data rate control to each component virtual sensor, we still use a data rate controller to regulate the fused data points to further stabilize the rate.

4.3 Implementation

We have implemented a prototype of Sentio on top of the standard Android (for mobiles, tablets, and the cloud entity) and Android Wear (for smart watches). The prototype Sentio SDK has 3,127 lines of code (LoC). The Sentio middleware has 3,726 LoC for the mobile module (i.e., standard Android OS) and 561 additional LoC for the wearable module (i.e., for Android Wear OS). The cloud entity is an Android x86 virtual machine hosted and managed by the Avatar system [31]. Although the implementation is Android-based, the implementation techniques are generic and can be used in implementations on top of other systems.

The implementation follows a Message-Oriented Middleware (MOM) architecture, where different components communicate with each other with events and messages. Since apps and the local Sentio middleware instance are different

processes, we use Android's binder IPC interface for the communication among them. Each Sentio middleware instance contains a queuing mechanism to manage events and messages, using the observer design pattern (dispatchers observe on queues). This pattern is widely used to keep the communicating components decoupled and maintain high efficiency at the same time. The communication between the Sentio middleware instances on different devices is done through a TCP library named kryonet [12]. The NetworkManager in Android Wear uses a different communication mechanism. Therefore, we use MessageApi[8] provided by Google Play Services for communication between a smart watch and another device. MessageAPI uses BlueTooth and WiFi for low level communication.

The Sentio middleware uses Android's sensor SDK to access and manage physical sensors. To respond to the status changes of the devices, we use Android's BroadcastReceiver mechanism to monitor status changes (e.g., $BATTERY\_LOW$) and register the intended actions, which are invoked by the Android automatically on status changes (e.g., switching a supporting physical sensor when the battery level on its host device is low).

4.3.1 Anatomy of a Sensor Registration Process

Let us now take a detailed look into how a sensor registration process works beginning from the API call to the receiving of data. For example, let us consider that a game running on the tablet needs to access the accelerometer from the smartwatch.

Calling APIs The programmer first builds a virtual accelerometer sensor using code similar to shown in the code listing 4.4. Then, the SensorListener callback function needs to be implemented (as shown in code listing 4.2). Inside this callback function, the programmer defines what to do with the sensor data (changing game graphics, taking specific action, etc). Once the sensor is built and the listener callback is implemented, the programmer can call the registerSensor() API.

Translating API Call to Event Sentio API library keeps track of this <sensor, listener> mapping. Next, the library translates this API call to a event of type REG (AvatarEventType.REG) and attaches the required meta-data (requesting app's id, requested sensor type, specified criteria, etc). This event is sent to the middleware by the library using binder IPC.

Processing the Event by the Middleware The IPCManager component of the middleware receives this event. It enqueues it in the EventQueue. The queue dispatcher processes the event based on it's meta-data. The dispatcher figures out that it is a sensor registration event and it needs to be sent to the sensor manager. The event is then dispatched from the queue and accepted by the sensor manager.

Resolving Physical Mapping of Sensor The sensor manager analyzes any requested criteria of the sensor registration (e.g., sensing mode $REAL\_TIME$). It then resolves the physical mapping of the sensor. The example shown in code listing 4.2 does not specify any criteria, rather it asked for a sensor specifically from a device. The mapping is explicitly provided. After the mapping is resolved, the current sensor manager will create an instance of *SensorRegInfo* which contains two mappings: requesting <device, app>, source <device, sensor>. The REG event will be modified by embedding this registration info. For our current example, requesting device is the tablet and source device is the smartwatch. Since the sensor source device is a different device, the REG event will be sent to the NetworkManager which will transmit it to the target device.

Accessing the Physical Sensor Once the middleware in the sensor source device (smartwatch) receives the event, it enqueues the event in it's own EventQueue. The event queue dispatcher handles the event and sends it to its sensor manager. The sensor manager uses the meta-data and *SensorRegInfo* to know which sensor to access.

It then uses Android's sensor API to access the sensor. The sensor data follows the reverse path from the smartwatch to the app in the mobile phone.

4.4 Application Case Study

We validated the Sentio prototype with four mobile apps: two proof-of-concept apps developed from scratch (i.e., SentioApp and SentioFit) and two open source mobile games that have been adapted to work with Sentio (i.e., Tilt Control and Space Shooting).

SentioApp demonstrates the ease of using Sentio API to access sensors from the PVSS. The app visually shows sensors available in the PVSS. Users can access any of these sensors and build composite sensors by visually selecting individual sensors. Once a sensor is built and registered, the app shows sensed data from this sensor on the screen. The code for this app is 339 lines of code (LoC) including the UI and the Sentio API calls. Invoking Sentio APIs need only six LoC. We ran this app on various combinations of smart devices. For example, we ran the Sentio middleware on several phones (Nexus 6, Nexus 5X, Moto X) and a watch (Samsung Gear Live) to form the PVSS and, then, we ran the app on Nexus 6.

SentioFit uses a composite sensor built using the heart rate monitor(HRM), the step detector, and the barometric pressure sensor. We applied a custom fusion function which: i) checks the heart rate (normal/resting heart rate is within [60,100], but a more robust formula can be used [2]), ii) keeps count of the number of steps per minute, and iii) converts air pressure to altitude. Based on these data, the fusion function generates the final result: KEEP_GOING, SLOW_DOWN, RUN_FASTER. In this way, the app uses the composite sensor to train the user, while making sure his heart rate is not too high (e.g., the heart rate can rise rapidly in higher altitude). We ran this app on a Nexus 5X mobile. The composite sensor used virtual sensors

physically located in Samsung Gear Live watch (HRM) and Nexus 5X (barometric pressure sensor and step detector).

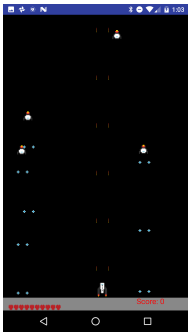
We have also used two open source mobile games from Github to demonstrate the simplicity of using Sentio in existing real-time apps. Both games use the accelerometer for controlling the play. Screenshots of these two games are shown in Figure 4.4. In the Tilt Control game, the users tilt the mobile to try to put all the balls in the central circle. In the the Space Shooting game, the user also tilts the phone to control the space ship and to shoot or shield. We integrated a virtual accelerometer using Sentio by only modifying 6 LoC. We ran the game on a Nexus 5X phone and the virtual accelerometer was mapped to three devices: the local Nexu5X, a Samsung Gear Live watch, and a Nexus 6 phone. The gaming experience was smooth and without any lag for all three scenarios when used with Android's sampling rates $GAME$ or $FASTEST$.

Developing and running these apps demonstrate the ease and robustness of Sentio API. For example, new or existing apps can use virtual sensors from any smart device by writing just a few LoC. Also, programmers can build composite sensors easily without worrying about data collection, communication, and aggregation.

4.5 Performance Evaluation

The main goal of our experimental evaluation is to assess the impact of Sentio on sensor data delivery, with a special focus on the performance achived by apps that have real-time constraints (e.g., mobile games). In addition, we evaluated the costs of Sentio API calls, sensor composition, and sensor switching. We have used Android-based smart phones, tablets, and smart watches, as well as Android x86 based virtual machines running in an OpenStack-based cloud. Specifically, the smart devices were: Nexus 5, Nexus 6, Nexus 5X, Moto X Pure, and Samsung Gear Live watches. Each cloud entity runs Android 6.0 x86 64 bit OS and is configured with

(a) Tilt control game



(b) Space shooting game

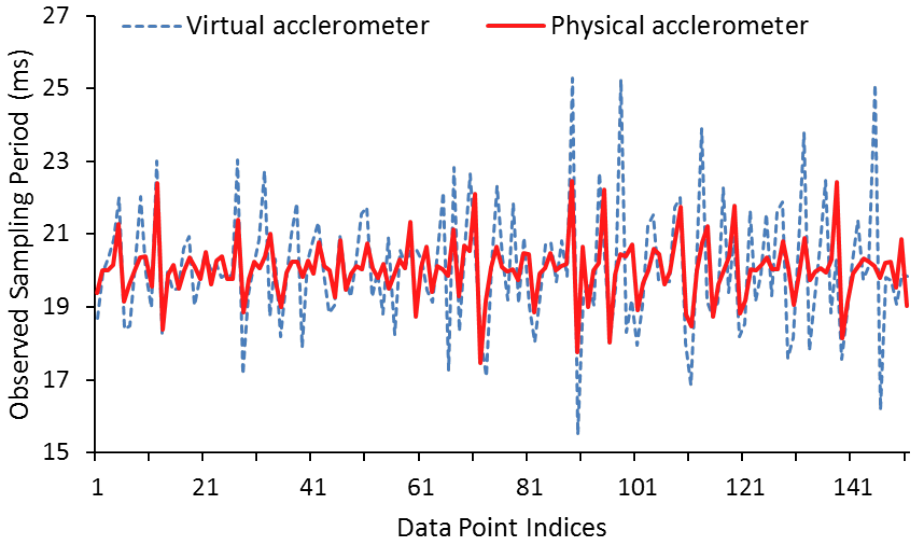
Figure 4.4 Screenshots of two open source mobile games modified to work with Sentio.

four virtual CPUs and 3GB of memory. Mobiles communicate with the cloud using our institution's secure WiFi network. For device-to-device communication, a mix of WiFi and Bluetooth is used.

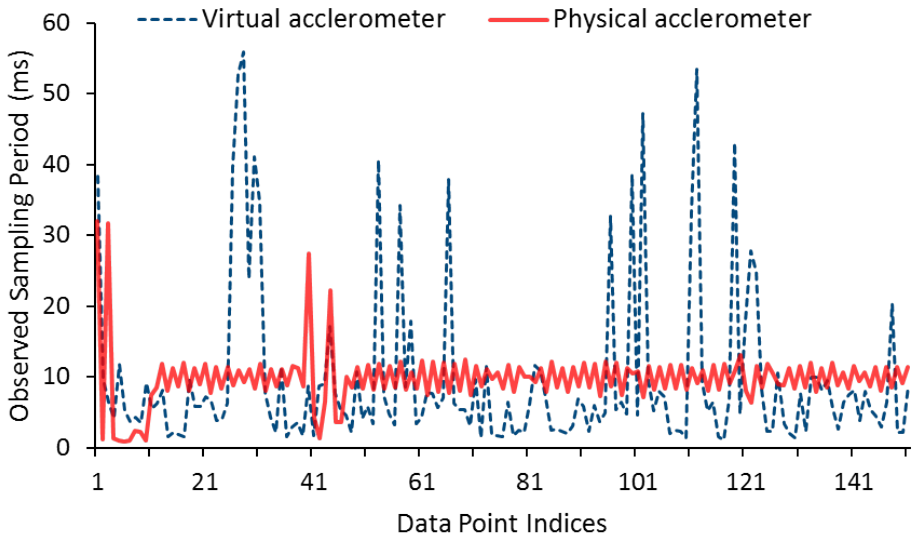
4.5.1 Real-time Performance for Accelerometer Sensor

We ran several experiments to verify that Sentio does not impose significant latency on sensing data as observed by the apps. For these experiments, we used the Space Shooting game. This game originally uses the accelerometer from the smart phone that runs the app to control the game. We used Sentio's API that exposes a virtual accelerometer to the app, and this virtual sensor can be mapped to the local smart phone accelerometer or to the accelerometer from a smart watch. In the experiments, the sensors continuously stream data to the app at the requested rate.

To understand the impact of Sentio on this real-time app, we measure the observed sampling period of the virtual accelerometer in the two scenarios: (1) the physical accelerometer is on the phone that runs the app; (2) the physical accelerometer is on the smart watch. Figure 4.5 shows the results. We measured the



(a) Physical accelerometer on phone



(b) Physical accelerometer on watch

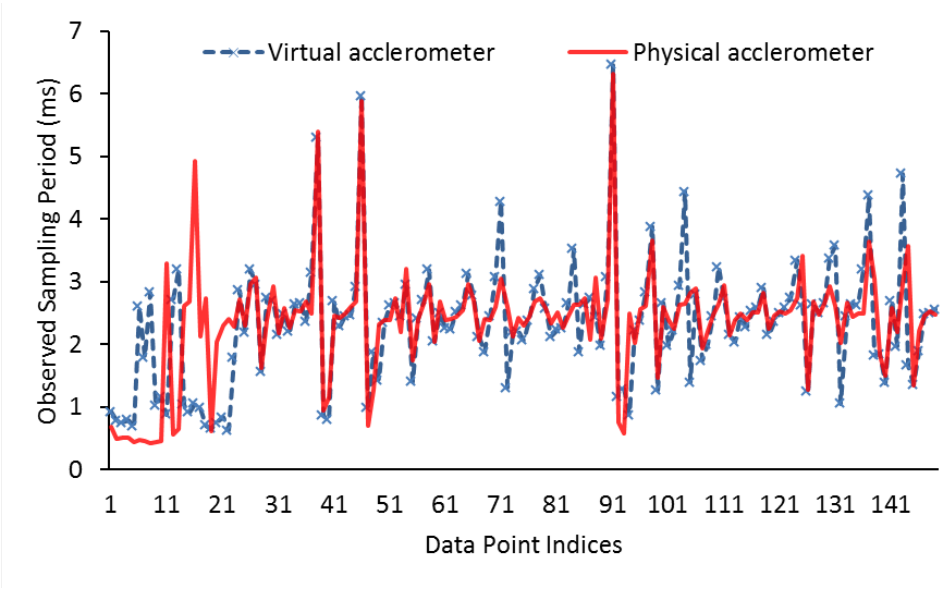
Figure 4.5 Comparison between the observed sampling period of a virtual accelerometer and a physical accelerometer when the physical accelerometer is on the phone and on the watch. The app runs on the phone and the sampling rate is set to GAME — 50 samples per second.

observed sampling period for the virtual sensor and directly for the physical sensor. The variation between the two curves is due to the overhead introduced by Sentio.

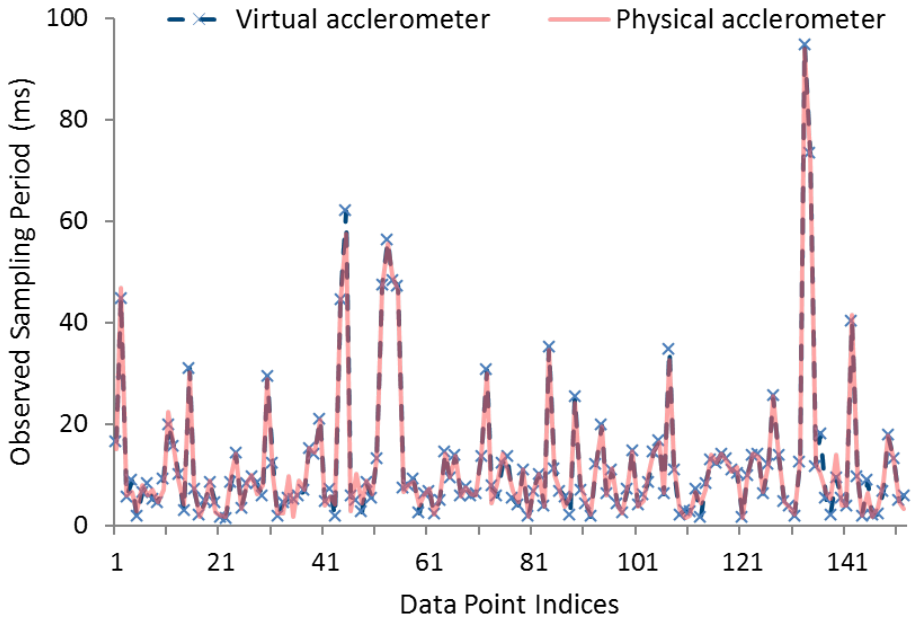
This metric does not include the network delay, but it includes the jitter. Similar to real-time multimedia streaming, Sentio will introduce a one time network delay. Then, the apps will be influenced only by the jitter, which is captured by our metric.

Figure 4.5(a) shows the results when the physical sensor is on the phone. The requested sampling rate for this case is "$GAME$" in Android, which requires the data to be delivered every 20 ms (50 data points per second). The average observed sampling period of Sentio's virtual sensor is 20.063 ms and that of the physical sensor is 20.054 mS. Both are very close to the requested value of 20ms. This means that Sentio incurs minimal overhead if the app runs on the same device with the sensor. The results show a higher variability for the virtual sensor, but the difference is in the order of a few ms — this difference is not perceived by users. Let us also point out that the observed sampling period for the virtual sensor is sometimes lower that the one for the physical sensor due to queueing effects in Sentio (e.g., a sample is queued for a longer time and delivered shortly before the next one).

Figure 4.5(b) shows the comparison when the physical sensor is on a the watch. In this scenario, the game is hosted on the phone and is played using the smart watch. Bluetooth (BT) is used for direct phone-to-watch communication. It can be observed that there is significant amount of variability introduced by the BT communication, but still under the limits that would impact playing experience. The high jitter can be reduced by using the rate controller which we will discuss and evaluate in a later part of this section. The difference between the average observed sampling rate on the virtual sensor and the physycal sensor is just 0.227 ms. Therefore, on average, the overhead of Sentio is still very low. We were surprised to observe that the average observed sampling period is about 10 ms instead of the requested 20 ms. We checked the hardware specification of the accelerometer on the watch and discovered that it delivers data at a faster rate if possible.



(a) Physical accelerometer on phone



(b) Physical accelerometer on watch

Figure 4.6 Comparison between the observed sampling period of a virtual accelerometer and a physical accelerometer when the physical accelerometer is on the phone and on the watch. The app runs on the phone and the sampling rate is set to FASTEST.

Next, we repeated the same experiment with the Android's sampling rate "*FASTEST*" to verify how Sentio performs under the most constrained real-time demands. This sampling rate requires to retrieve and deliver sensed data as fast as the sensor hardware supports. Figure 4.6 shows the results. The difference between the average observed sampling period is 27μs when the physical sensor is on the phone and 10μs when the physical sensor is on the watch. We also observe that the absolute values are substantially lower on the phone compared with the previous experiment. This could improve the performance of certain apps. When the physical sensor is on the watch, we observe a higher variability for the observed sampling rate of the physical sensor, and this translates in a similar variability for the virtual sensor. Overall, these results demonstrate that Sentio impose a negligible overhead, and it works well even under strict real-time constraints.

In the first two experiments, we used BT communication between the two devices. In the following experiment, we use WiFi communication between two phones: the app runs on one phone, and the physical sensor is on the other phone. We varied the sampling rate to see its impact on Sentio's performance. Table 4.3 shows the result. The average, standard deviation, and median values are shown for each pair of virtual and physical sensors. We observe that the difference in average observed sampling rate for the virtual sensor and the physical sensor is very low (¡1ms) for GAME, UI, and NORMAL sampling rates. For the FASTEST sampling rate, the difference is 6.6ms. These differences are sligtly larger than in the case of BT communication. We attribute this result to the contention in our public WiFi network, which leads to queuing effects in Sentio, especially for high sampling rates. Nevertheless, the results demonstrate that Sentio also works well with WiFi communication.

Table 4.3 Observed Sampling Period (in ms) for Different Sampling Rates when using WiFi Communication. The Game Runs on One Phone, and the Physical Sensor is on the Other Phone

	NORMAL		*UI*		*GAME*		*FASTEST*	
	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy
Avg	199.48	199.39	39.82	39.69	19.16	19.09	10.4	3.98
StDev	43.7	4.47	21.42	1.86	77.01	1.76	8.22	0.7
Median	199	200	39	40	12	19	9	4

4.5.2 Performance for Other Sensors

So far, we have shown that Sentio works efficiently for real-time apps (e.g., games). We have used the accelerometer for these experiments because this is the sensor most commonly used in games. Next, we evaluate performance of Sentio for other types of sensors, such as light, magnetic field, gyroscope, barometric pressure sensor, etc. For this experiment, we have used SentioApp, running on a Nexus 6 phone. The app uses various virtual sensors, which are physically mapped to sensors from a Moto X mobile phone. Android's *NORMAL* sampling rate (five samples per second) is used in the experiment. The results in Table 4.4 show that the maximum difference in the observed average sampling period is 4.69 ms. This is a small value when compared to the expected period of 200ms. Therefore, Sentio is expected to work efficently for all the sensors we tested.

4.5.3 Performance for Offloaded Code

Sentio simplifies app offloading because the code running in the cloud can seamlessly access sensors embedded in smart devices. We evaluated how Sentio performs once the code is offloaded to the cloud using the Tilt Control game. The game is hosted in the cloud, and it uses a virtual accelerometer physically mapped to Moto X's

Table 4.4 Observed Sampling Period (in ms) for Different Sensor Types When SentioApp Runs on One Phone, and the Physical Sensor is on the Other Phone. WiFi is Used for Communication. *NORMAL* Sampling Rate

	Light		Magnetic Field		Gyroscope		Orientation		Pressure	
	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy
Average	395.53	394.96	200.6	195.91	203.71	201.02	194.69	194.29	198.11	196.16
StDev	518.16	430.35	63.73	10.48	33.01	6.2	49.87	1.06	99.56	144.39
Median	197	200	200	196	199	201	191	194	247	185

accelerometer. Table 4.5 shows the results. The highest difference in the average observed sampling period (5.32 ms) is observed for the *FASTEST* sampling rate. Sentio worked very well for the other sampling rates. These results prove that code offloading with Sentio is practical for context-aware apps.

Table 4.5 Observed Sampling Period (in ms) when the Tilt Control Game is Offloaded to the Cloud for Different Sampling Rates

	NORMAL		*UI*		*GAME*		*FASTEST*	
	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy
Average	200.7	200.5	46.79	46.73	19.18	19.13	9.32	4
StDev	8.12	0.99	12.28	9.9	5.39	0.93	3.44	0.81
Median	201	201	43	40	19	19	9	4

4.5.4 Jitter Reduction Using a Rate Controller

The results we presented so far use sensor data as soon as they become available. As virtual sensors may be mapped to different devices(i.e., not the one running the app), network delay will cause data to arrive at the app with irregular frequency. This explains the zig-zag pattern in some of the previous graphs. It might not be ideal to

directly deliver sensor data to the apps based on this pattern (e.g., rendering sensor data might result in a jerky motion). As described in Section 4.2.2, Sentio uses a rate controller to reduce this kind of jitter and smoothen the sensor data over time.

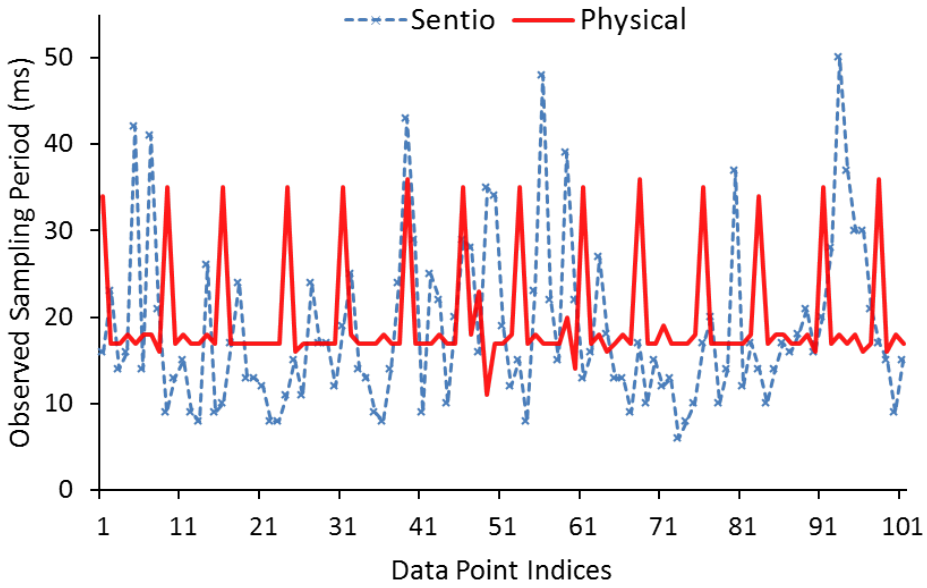
Figure 4.7 shows the improvement achieved by using a rate controller for the SentioApp running on the Nexus 6 phone, with the virtual accelerometer mapped to the Moto X phone, when the $GAME$ sampling rate is used. We observe that the jitter is substantially reduced. Similar results, omitted for brevity, have been obtained for virtual magnetic field sensor and the $NORMAL$ sampling rate.

4.5.5 Cost of Sentio Initialization and API Calls

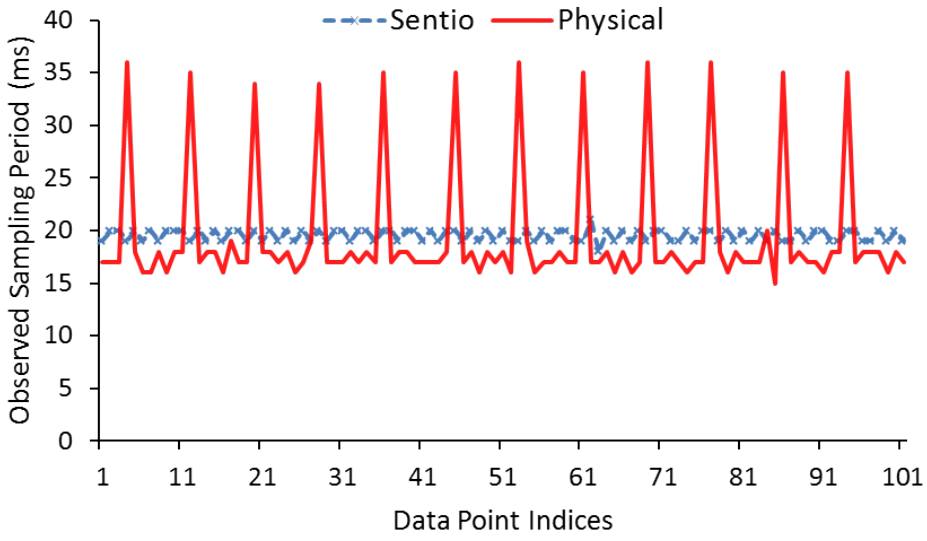
Sentio initialization is a one-time operation that occurs when the middleware is installed on a device. At that time, the sensors in the device are discovered and a sensor registry is built. The time it takes to build the sensor registry varies from device to device: 2,566 μS, 392 μS, and 6,319 μS for Nexus 6, Moto X, and Samsung Gear Live, respectively. The sensor registry is a custom data structure which has a base size of 531 bytes (without any sensor entries). Initializing the sensor list takes 67 additional bytes. After that, the sensor list increases by 16 bytes for each sensor.

The API methods *registerSensor()* and *unregisterSensor()* need 649 bytes of data to be exchanged between Sentio components. Each sensor data bundle has 924 bytes, including header information and the sensor data itself.

The *getAvailableSensorList()* method, used to retrieve the sensor registry from the cloud entity, takes 60ms on average. This latency includes sending a request to the cloud entity and receiving the sensor registry. In contrast, fetching a local copy of the sensor registry only takes 800μS.



(a) Without Rate Controller



(b) With Rate Controller

Figure 4.7 Observed sampling period comparison for accelerometer with and without the rate controller. The sampling rate is $GAME$ — 50 sampples per second.

4.5.6 Performance of Composite Sensors

Composite sensors add an additional layer of software abstraction over the individual virtual sensors. The overhead incurred by this software abstraction is very small

compared to the cost of accessing the physical sensor and data communication. Our experimental results confirm that composing (i.e., fusing) sensors does not add significant latency to the final delivery of data. First, we evaluated a composite sensor built on top of a magnetic field sensor and a barometric pressure sensor which are physically located in the same device (Nexus 6). The average delay caused by fusing the data and delivering it to the app (on the same device) is only 162.02μS. Next, we evaluated a composite sensor built on top of a light sensor physically located in a Nexus 6 phone and a magnetic field sensor physically located in a Samsung Gear Live smart watch. In this case, the average delay caused by the composite sensor's software abstraction was 199.53μS.

4.5.7 Cost for Seamless Switching of Sensors During Context Change

Sentio switches sensors from one device to another seamlessly during context changes on the devices embedding the sensors. For example, if the battery on a device is low, Sentio re-assigns the physical mapping of sensors from this device to another device. We have measured the latency experienced by the app during this re-mapping. We have run SentioApp in Nexus 6 using a virtual magnetic field sensor, which is physically mapped to the magnetic field sensor of the smart watch. In our scenario, Sentio senses when the battery level in the smart watch drops below a specific threshold and re-maps the physical mapping of Nexus 6's virtual sensor to the physical magnetic field sensor of the local device (i.e., Nexus 6). The latency for this re-mapping consists of several components: the delay to pass an event from the smart watch to Nexus 6, with the low-battery context change information; the delay to find another suitable sensor from the registry; and the delay to register the newly chosen sensor and receive data from it.

Figure 4.8 shows that sensor switching causes a clear increase in the observed sampling rate: 375ms as opposed to the average value of 184ms before and after

the switch. We note that this increase only affects the immediately next data point after switching. For all practical purposes, skipping one data point is an acceptable compromise.

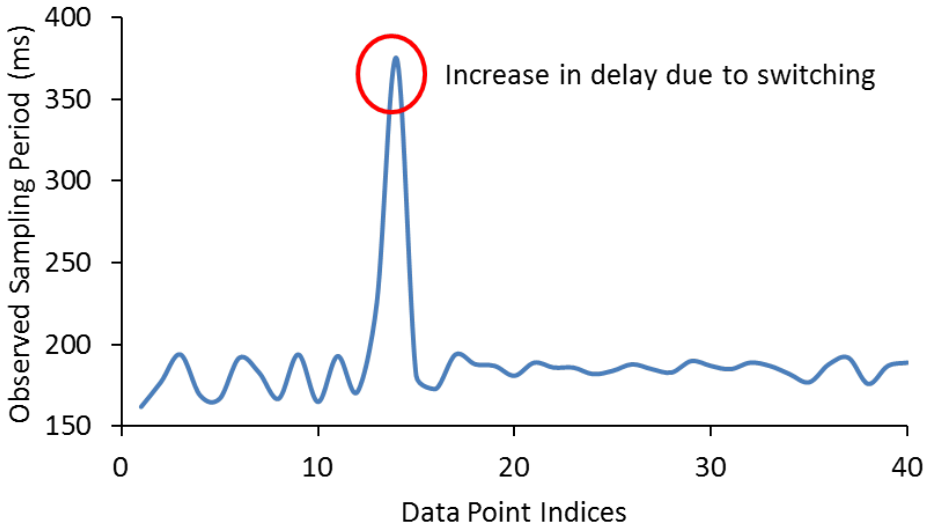


Figure 4.8 Delay observed by SentioApp during a sensor switching from a watch to the local phone. NORMAL sampling rate is used. The increase indicates the switching latency.

4.6 Chapter Summary

This chapter presented Sentio, a personal virtual sensor system, which uses the collective sensing capabilities of all sensors in all devices possessed by a user. Sentio provides a unified view of a personal sensing ecosystem. Apps can use Sentio API to access any virtual sensor in real-time, no matter on which perosnal device its associated physical sensor is located. They can also selectively access the most suitable sensor of a particular type and can build composite virtual sensors using the same API. The Sentio middleware can transparently re-map virtual sensors to physical sensors in response to context changes. We have implemented a protoype of Sentio and two proof-of-concept apps. In addition, we adapted two open source apps

to work with Sentio. We evaluated Sentio using these four apps, and the results show that Sentio does not introduce significant overhead in sensing. Therefore, it can be used for many types of context-aware apps, including apps that have tight real-time constraints such as mobile games.

CHAPTER 5

MOITREE: A MIDDLEWARE FOR CLOUD ASSISTED MOBILE COLLABORATIVE COMPUTING

Collaborative computation over collections of mobile devices can be performed on sensor data collected by a group of users in order to produce useful end-results. However, due to resource (e.g., energy) limitation in mobile devices, running such distributed computation on mobile devices is difficult. Cloud-assisted mobile platforms overcome this limitation by employing a surrogate entity in the cloud. For supporting distributed computation over such cloud-assisted mobile platform, a framework is needed to provide both the distributed execution environment and the programming support for app development. This chapter presents Moitree, a middleware that facilitates the execution of collaborative apps within groups of mobile users (each group being represented by a collection of mobile device/surrogate pairs). In addition to run-time support, Moitree provides an API set for developing mobile distributed apps. Moitree is designed and implemented for the Avatar platform [31] which provides the distributed architecture needed to perform collaborative computation over mobile devices.

5.1 Overview of Avatar Platform

The Avatar platform is motivated by the strong demand for fast, scalable, and energy efficient distributed computing over mobile devices. Several research studies [68, 65, 98] have investigated cloud support for mobile distributed computing. These projects focus mostly on enabling communication among mobile users. Other efforts have investigated offloading mobile code execution to the cloud in the context of stand-alone apps [42, 100, 39, 67]. While Avatar leverages high-level ideas from these efforts, its goal is fundamentally different - to provide a novel programming model

and execution environment for developing, running distributed apps on mobiles by integrating the benefits of modern mobile devices and cloud computing.

5.1.1 Avatar Architecture Overview

In Avatar, a mobile user is represented by one mobile device and its associated "avatar" hosted in the cloud. An avatar is a per-user software entity which acts as a surrogate for the user's mobile device. This enhances the computing power of the mobile and helps to reduce the workload and the demand for storage and bandwidth on mobiles. Each avatar can be hosted as a container or a virtual machine in the cloud in order to provide resource isolation and to simplify per-user resource management. Although containers can be used as the surrogate entity, our prototype is implemented using virtual machines (Android x86) for the flexibility and ease of prototyping. The operating system and the runtime (Dalvik or ART) is the same in avatars and mobiles. Exactly the same app or app components (e.g., functions, threads, etc.) can thus run on both of them. Implicitly, avatars save energy on the mobiles and improve the response time for many apps by executing certain tasks on behalf of the mobiles. Avatars are always available, even when their associated mobile devices are offline because of poor network connectivity or simply turned off. Each avatar coordinates with its mobile device to synchronize data and schedule the computation of avatar apps on the avatar and/or mobile device. A mobile device does not interact directly with the avatars of other mobile users. For reducing bandwidth usage in mobile devices, all user-to-user communication is offloaded to the cloud (i.e., always goes through the avatars).

To summarize, a user in the Avatar architecture is represented by a mobile/avatar pair which ensures: i) the availability of users' content such as app-data is increased (even if the phone dies, the avatar can still participate in collaborative computations), ii) it has rich computational resources, and iii) a

seamless computation and communication offloading mechanism (between mobile and avatar) will save a significant amount of energy (of the mobile device) as well as money (by reducing bandwidth usage).

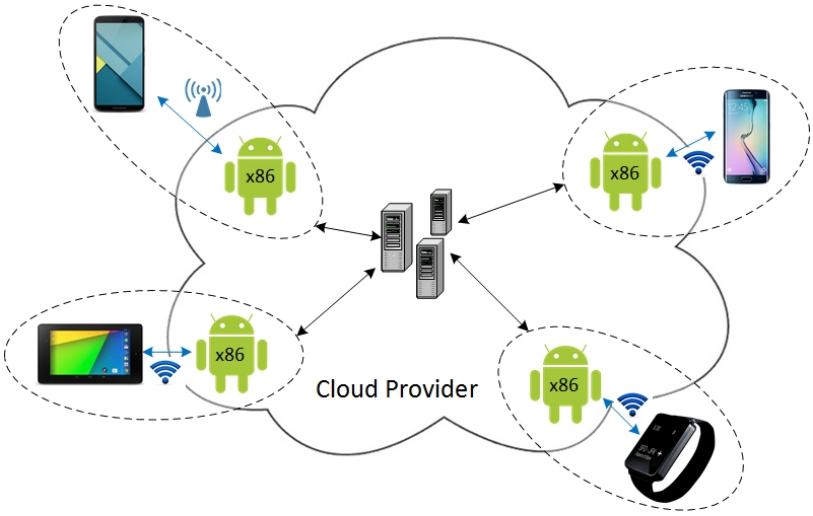


Figure 5.1 The conceptual view of the Avatar platform.

Figure 5.1 shows a simple conceptual view of the Avatar platform. For each user, a new Android x86 VM is instantiated on the cloud. A mobile-avatar pair is established and maintained after this instantiation. When a user installs an app running over Avatar (called "Avatar app" in the the rest of the dissertation), the same app gets installed on both mobile and avatar. Users can install as many Avatar apps as they want on their device. Standard Android apps can co-exist with Avatar apps on the mobile without conflicting each others' running mechanism.

5.1.2 Avatar App Example

There can be different types of collaborative and distributed apps designed to utilize the features of a cloud-assisted distributed mobile computing platform like Avatar. Among those various types of apps, finding people of interest in a crowded area is used as an example in our prototyping effort to demonstrate the usefulness and strength of the platform. There is a real-life motivating example [93] where this type of app

could help the authority. This news tells a story how police caught a murderer in San Francisco with the help of tourists' photo taken in the crime spot. Although, finding people of interest can be applicable in crime-fighting situations like this, the one used in our prototype (LostChild) is rather altruistic in nature.

Alice was visiting Times Square with her mother in a busy Saturday afternoon. They were roaming around. But suddenly, the mom noticed that Alice was lost and she failed to locate her nearby. She observed that many nearby visitors were taking photos of themselves. She wondered if Alice could have appeared in many of those photos. Suppose, there existed an Avatar app (e.g., the LostChild app) which was used by many people in that region. The app would be running on their mobile-avatar pairs. Now, whenever someone takes a photo, it is synchronized to their avatar. When the mom starts the app and searches by submitting Alice's photo, the app forwards the request to the avatar. The app running on her avatar contacts a cloud service and forms a group with available participants (who were present in Times Square during that time period and willing to participate). The app initiates the search in that group. The app running on a participant's avatar will execute the face recognition task to find whether Alice appears on any of their recently synchronized photos. If a match is found, it will send back the result to Alice's avatar. The LostChild app on the mom's avatar will merge all the results received and forward it back to the mobile app. Finally, the app will show the trajectory of Alice's movement on a map so that the mom can follow the direction and find her child.

5.2 Motivation and Challenges

Collaborative apps similar to the one described in Section 5.1.2 can provide useful results very quickly by utilizing distributed computation. Valuable energy on users' mobile devices can also be saved by using such approach. If the avatars use shielded

execution in the cloud using [29, 9]), users' privacy can also be improved to certain extent.

It is evident from the aforementioned example that such distributed execution model involves communication among many different devices and components (mobiles, avatars, cloud services, etc). A programmer has to invest a significant amount of time and effort to develop components for tasks such as discovering potential participants for a distributed computation, forming groups among selected participants, managing group membership, distributing computation and data among participants, managing communications between mobile-surrogate (avatar) & among such pairs inside a group, etc.

All these tedious tasks may not even be directly related to the general goal or business logic of the app. Therefore, providing abstractions and a programming model for groups and group communication will help programmers to minimize their programming effort to a significant extent. App developers should not be aware of many low-level details and architecture of the system. They should be able to call APIs to form a group among relevant users by specifying simple criteria of the group, distribute the data and computation among participants, and get back the results. The first challenge is to define a programmer friendly, high level programming abstraction of the distributed computation model supported by Avatar. There are additional challenges for designing a simple API set using such programming model and yet to keep it powerful in terms of exploiting the strength of the underlying distributed platform.

There are many networking details involved in a distributed system such as Avatar. There might be delays between making an API call and getting its response. Programs should not get blocked after making API calls. The unexpected delay for getting responses can break the semantics of a program. For this reason, responses

need to be delivered in an asynchronous fashion. The API design should be based on asynchronous callbacks.

In order to support such high-level API and provide the distributed run-time support, a middleware is needed which provides the execution environment for collaborative computation on top of platforms such as Avatar. This platform consists of different types of mobile devices with various resource and capabilities. This implicitly brings lots of challenges to the middleware due to the resource constraints and heterogeneity.

The middleware has to be able to translate the high-level API calls to low level operations in a very fast and efficient way. For this reason, it is essential to keep the middleware components loosely-coupled. For supporting asynchronous APIs, the middleware also needs to be compatible with handling delayed responses. An event-driven or message-oriented middleware design is generally employed for keeping the system loosely-coupled and also for handling asynchronous callbacks. This brings us another challenge, how to design events/messages so that they can support fast communication and execution without introducing significant overhead to the system. To overcome this challenge, the middleware has to i) use lightweight and efficient data structures for events/messages, ii) reduce the time for serializing and deserializing data, iii) reduce the amount of processing needed for making event/message routing decisions.

5.3 Moitree Overview

In order to facilitate development and execution of collaborative apps on top of Avatar, we have designed Moitree, an API set and a middleware. The middleware provides supports for Moitree APIs by hiding the underlying complex heterogeneous nature of the platform. In other words, Avatar platform provides a specification for running apps on a cloud-assisted distributed computing platform for mobiles. Moitree

provides an API set and the middleware for the Avatar platform. These two together enable writing distributed mobile apps easily and executing them according to the execution environment specified by the Avatar platform.

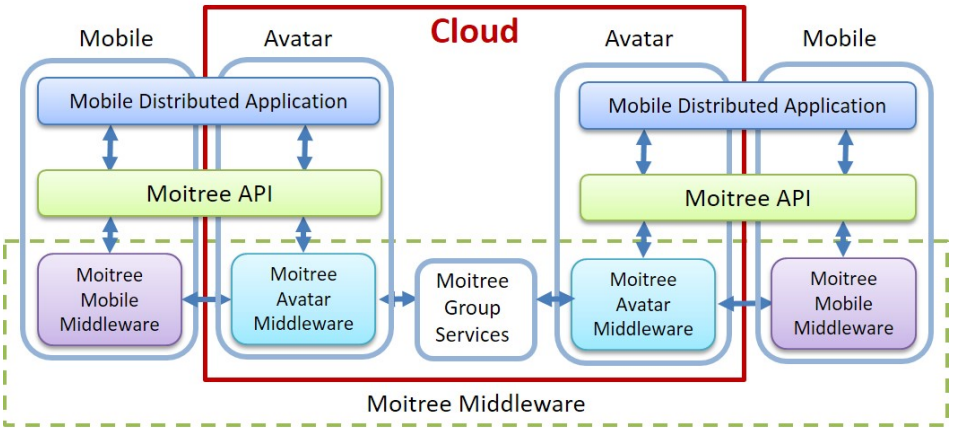


Figure 5.2 Execution of distributed apps in Avatar facilitated by the Moitree API and middleware.

Figure 5.2 shows an overview of Moitree where two users' mobile-avatar pairs are running a distributed app. The app is using Moitree API for performing collaborative computation on top of Avatar platform. Moitree middleware translates API calls and executes it on the platform.

Moitree middleware consists of two major components - Moitree Mobile Middleware (MMM) which runs on mobile devices and Moitree Avatar Middleware (MAM) which runs on avatars. These two components present a unified view of the mobile-avatar pair. The middleware has other cloud services such as the Group Management Service, Discovery Service, etc which will be described later in this chapter.

5.3.1 Moitree Key Ideas

Moitree API set is based on a high-level programming model. It has several key ideas for providing the high-level abstraction of the collaborative/distributed computation model.

Groups and Group Hierarchies A *group* is a fundamental unit over which mobile distributed computation is done. A group is a set of users selected and organized based on some criteria or properties (e.g., location, time, social context, etc.). Each group is app-specific, and each app can create as many groups as it needs. All members of a specific group, by default, run the same app. A user can be part of multiple groups at the same time.

Groups in the same app can form hierarchies with the groups at lower levels being subgroups of the ones at upper levels, and are maintained in a tree structure. This helps programmers to structure the distributed computation for certain apps. For example, subgroups can be recursively created to solve location-based computations using the divide-and-conquer strategy (e.g., finding a free parking spot in a city).

Communication Channels A communication channel provides high level messaging support and makes it easy to offload communication to the cloud. There are four types of channels: (i) *broadcast* for sending messages to all members of a group; (ii) *anycast:* for sending messages to a random member of a group; (iii) *pointToPoint:* for sending messages to a specific member; and (iv) *scatterGather:* for sending messages to all members of a group and then receiving answers from some group members as a function of their computation results. The *broadcast* channel is unidirectional, while the other three are bidirectional.

When a function invokes a communication channel on a mobile, the call is intercepted in the middleware and always forwarded to the cloud (avatars) to carry on. User-to-user communication always occurs via the avatars.

Messages can be designated as *persistent* by the programmers. These messages are useful for forwarding data to new comers to the group (e.g., a person who entered the region that defines the group after the group was formed). Persistent messages for a group are stored in the cloud and distributed to new members when they join the group. Messages can also be locally persisted in mobiles and avatars. Since the mobile-avatar network connection can be intermittent, these persisted messages are used to ensure reliable data transfer between mobile and avatar.

Dynamic Group Membership Since people move and their context changes over time, group membership also changes. The dynamic group membership in Moitree shields the programmers from handling group dynamics. The middleware selects and maintains group members automatically based on properties specified by the programmers. For example, a group can be formed for a given geographic region during a specific time interval. The middleware will add/remove group members based on who enters/leaves the region during that interval.

5.4 Moitree API

The Moitree API is based on an asynchronous and event-driven design in order to be compatible with the loosely-coupled, distributed, and message-oriented middleware design. Programmers have to register listeners/callbacks for particular events/messages. It can be noted that many parts of the Android SDK follows the event-driven programming paradigm. Java Swing GUI toolkit is also famous for using such API architecture. Naturally, choosing an event-driven design for Moitree API will ensure that it is providing an already known programming style to Java and Android programmers.

5.4.1 Moitree API Description

Moitree API for group management functionality such as creating a group, getting the parent group, getting the leader of a group, deleting a group, etc are listed in Table 5.1. Group membership API such as adding/removing a member to/from a group, defining the context of the group (e.g., location, time), etc are shown in Table 5.2. Table 5.3 enlists the communication APIs of Moitree.

The Moitree API set uses three main classes. The *Avatar* class provides methods for group creation and joining. The *AvatarGroup* class offers methods for group management (e.g., leave/delete the group, create subgroups) and group communication. The *MembershipProperties* class is a utility class that has methods for specifying the group properties.

5.4.2 Group Creation, Membership, and Deletion

The *createGroup* API takes four parameters: *parent*, *prop*, *enableLeader*, and *groupLifetime*. The *parent* parameter is used for group hierarchies. If it set to *null*, it means the group is the potential root of a group hierarchy; however, groups are not required to be part of hierarchies. More details on group hierarchies are presented in Section 5.4.3.

The *prop* parameter is an instance of the *MembershipProperties* class. It specifies the context properties that must be satisfied by group members (i.e., region and time interval). Specifying the context can be done using membership APIs shown in Table 5.2.

Some groups may need leaders to implement functions such as consensus or scheduling among their members. If the *enableLeader* is set to *true*, then the user who creates the group becomes the leader. If the leader leaves the group, the middleware selects a new leader. Currently, Moitree selects a random user as a new leader,

Table 5.1 Moitree API - Avatar and AvatarGroup Class

Method	Description
createGroup(AvatarGroup parent, MembershipProperties prop, boolean enableLeader, double groupLifetime)	Creates a group with members selected based on membership properties *prop*; if *enableLeader* is true, the group has a special member with leader role. *groupLifetime* specifies how long the group should exist without receiving any messages from the members.
changeParentGroup (AvatarGroup newParent)	This method is used to re-organize the group tree.
onCreateGroup(AvatarGroup group)	Callback method registered to Moitree for delivering newly created AvatarGroup object. Moitree pushes this group parameter to callbacks registered by initiator's and participants' apps.
joinGroup(AvatarUser user, AvatarGroup group, Credential c)	Joins an already existing group. The credential ensures that the user has appropriate permissions to join the group. Credentials are generated when a group is created and distributed to the members as part of group creation.
deleteGroup(Credential c)	Deletes an existing group. Credentials are used to ensure that the callee has permission to delete the group.
getMembers()	Returns the list of group members.
getLeader()	Returns the group leader.
getRoot()	Returns a reference to the root of the group.
getParent()	Returns a reference to the parent of the group.
removeFromGroup(AvatarUser user)	Removes *user* from a group.
getChildGroups()	Returns the list of children groups of the group.

Table 5.2 Moitree Group Membership API - MembershipProperties Class

Method	Description
setTimeBound(Time from, Time to)	Used to set the time property for identifying users active in the given time interval (typically used in conjunction with the location property).
setLocationBound(LatLng center, double radius)	Used to specify a circular region where a user is/has been/will be (typically used in conjunction with the time property).
setLocationBound(LatLngBounds rectRegion)	Used to specify a rectangular region where a user is/has been/will be (typically used in conjunction with the time property).
setSocialNetwork(SocialNetwork network, Activity a)	Used to identify group members who are part of the user's social network based on activities such as friendship, work, sports, etc.
setList(List⟨Users⟩ users)	Used to add specific users to a group.

Table 5.3 Moitree Group Communication API - AvatarGroup Class

Method	Description
setReadCallBack(ReadCallBack callBack)	Registers callback methods for incoming messages. *ReadCallBack* is an interface with four callback methods corresponding to broadcast, anycast, scatter-gather, and point2point.
broadcast(byte[] message)	Used to broadcast messages to a group.
anycast(byte[] message)	Used to send a message to a random member of the group.
scatterGather(ChannelID cid, byte[] message)	Used to broadcast messages to a group and get responses from group members back to the broadcaster. An app can use as many scatterGather channels as required by using different *ChannelID* for different channels
pointToPoint(byte[] message, AvatarUser to)	Used for user-to-user communication.
sendToLeader(byte[] message)	Used for sending a message to the group leader.

but other leader selection policies could be implemented. The method *sendToLeader* allows any user to send messages to the leader, without the need to use the ID of the leader, which improves fault-tolerance. The method *getLeader* is normally used to determine if the local user is the leader of the group; if yes, the app needs to run leader-specific functionality.

The *groupLifetime* parameter specifies that a group has to be deleted by the middleware in the absence of any group communication for the *groupLifetime* duration. In this way, Moitree de-allocates the resources associated with a group when the group is not active. The apps receive an exception and terminate.

Groups are dynamic in that members can come and go. Users can join a group using the *joinGroup* method. The user invoking this method must know the group ID and have the right credentials. For example, a new user is invited to a multi-player mobile game and is provided the group ID and the credentials. A user can leave a group by calling *removeFromGroup*. Currently, this method is used only to remove the user invoking it. However, we plan to explore if this method should be allowed

to remove other users; such functionality could be useful for group creators and/or leaders.

5.4.3 Group Hierarchies

By default, if a user belongs to a group, it is also a member of the parent group. For simplicity, we do not allow overlapping sibling groups or overlapping groups on different branches of the hierarchy tree.

A user in a subgroup can get a reference to the parent group using the *getParent* method or to the root group using *getRoot*. Similarly, a user in a group can get a references to the child subgroups (i.e., one level down in the hierarchy) using the *getChildGroups* method.

5.4.4 Group Communication

Apps may use any combination of communication channels as needed. Each channel is instantiated by the middleware upon its first invocation in the app. Each app instance can use its own scatter-gather channel. To distinguish these channels, they have unique ChannelIDs. The communication on all channels is asynchronous. Anyone can send a message anytime and receive messages through callback methods. Each sending communication channel is paired with a receiving callback method.

5.4.5 Using Moitree API

To use Moitree API, an Avatar object has to be instantiated first. Avatar object is a singleton representing the mobile-avatar pair. This object is used to call any subsequent Moitree API. To get the instance of the Avatar object, the programmer needs to call an API along with two parameters (shown in the following code sample). The first parameter is the fully qualified name of the application's main class (e.g., "edu.njit.lostchild.MainActivity"). This parameter is used by the middleware to find the app instance and deliver any incoming data destined for it. The second parameter

is the context object of the application. The context object is used to register and use Android's IPC mechanism.

```
String appId = MainActivity.this.getClass().getName();
Context context = MainActivity.this;
Avatar avatar = Avatar.getInstance(appId, context);
```

Group Management API The group management API starts with a createGroup() call. Since groups in Moitree are formed and maintained based on dynamic membership, programmers have to specify the criteria group membership. For example, let us assume that the programmer wants to create a group with users who are present in NJIT campus within a particular time range. At first, she has to define these criteria by creating a *MembershipProperties* object.

```
MembershipProperties properties = new MembershipProperties();
LatLng location = new LatLng(40.7435032,-74.1785025);
double radius = 500.0; //500m radius
prop.setLocationBound(location, radius);
prop.setTimeBound(startTime, endTime);
avatar.createGroup(properties);
```

startTime and *endTime* are developer-defined time range. LatLng class is defined in Google's API for Android. Here, *location* and *radius* are used to represent a circular region. Instead of this, programmers can also use a rectangular region by using:

```
LatLng southwest = new LatLng(40.743750, -74.178589);
LatLng northeast = new LatLng(40.744628, -74.179786);
LatLngBounds rectRegion = new LatLngBounds(southwest, northeast);
prop.setLocationBound(rectRegion);
```

LatLngBounds is also defined in Google's API for Android. It represents a latitude/longitude aligned rectangle.

If programmers want to use hierarchical groups and set an existing group as the parent of a group, they can use a polymorphic version of the createGroup() API and pass the parent group's reference.

```
avatar.createGroup(parentGroup, properties);
```

Additionally, two more parameters can be passed to this API. One of them is a boolean to denote whether there would be a leader in the group or not. The last parameter is a double value to define a lifetime of the group (e.g., how long the group should exist without any interaction).

It can be noted that an app can create any number of groups it needs. The middleware uses membership properties and other parameters to find eligible participants for creating a group.

To receive the group information from the middleware, programmers have to register a callback function shown below:

```
avatar.setOnCreateGroupCallback(new OnCreateGroupCallback() {
   @Override
   public void onCreateGroup(AvatarGroup avatarGroup) {
      //use this group object to communicate with members
   }
});
```

Once the group is formed based on programmer-specified properties, an AvatarGroup object is pushed by the middleware to this registered callback function. It is worth noting that the group object is pushed by the middleware to both the initiator and participants of the group. This group object embodies different information about the created group such as the id, channel information, etc. Programmers can use this object for communicating with group members.

Group Communication API After the group formation, the initiator can send data to any members/participants using different kinds of communication channels. Moitree supports *broadcast*, *pointToPoint*, *anycast* and additionally *scatterGather* channels. Moitree manages all these channels on top of a physical channel. It uses multiplexing and demultiplexing by attaching meta-data with the data. The first three channels are self-explanatory. The *scatterGather* channel is an specialized channel combining OneToMany and ManyToOne topology. The initiator can send

data to all participants and then participants can send back the result to the initiator by using this channel. For example, the LostChild app is an ideal example of using the *scatterGather* channel. The mother's LostChild app will act as an initiator of the group and send Alice's photo to participants using *scatterGather* channel. Then participants will run the face recognition task and send back the result with the same *scatterGather* channel. To send the data, programmers can pass a byte[] data to any specific channel and tag it with $AvatarMessageType.DATA$.

```
1 avatarGroup.scatterGather(data, AvatarMessageType.DATA);
```

Programmers can mark a data as persistent. This will tell the middleware to cache the data for any participants who will join the group later. The middleware will subsequently deliver any cached persistent data to participants who joined late.

```
1 boolean isPersistent = true;
2 avatarGroup.scatterGather(data, isPersistent, AvatarMessageType.DATA);
```

The second parameter indicates that this is a persistent message. Additionally, programmers can use multiple channels of the same type. For example, she can create 2 scatterGather channels. For this, the programmer can mark a channel with a channelID. The middleware will deliver the data to the particular channel using the channelID.

In order to receive data using channels, developers have to implement callbacks and register them with the middleware. When participants want to send back the result they can tag the result data with $AvatarMessageType.RESULT$.

```
1  avatarGroup.setCallBack(new ReadCallBack() {
2     @Override
3      public void broadcast(AvatarMessage avatarMessage) {}
4
5     @Override
6     public void anycast(AvatarMessage avatarMessage) {}
7
8     @Override
9     public void pointToPoint(AvatarMessage avatarMessage) {}
10
```

```
11    @Override
12    public void scatterGather(AvatarMessage avatarMessage) {
13       if (avatarMessage.getMessageType() == AvatarMessageType.DATA) {
14          //generally used to run computation in a participant's app
15          //use avatarMessage.getData() to run computation, then send back
                the result
16          avatarGroup.mapReduce(result, AvatarMessageType.RESULT);
17       } else if (avatarMessage.getMessageType() == AvatarMessageType.RESULT)
             {
18          //This is a result received by the initiator
19       }
20    }
21
22    @Override
23    public void getPersistentData(AvatarMessage avatarMessage) {}
24 });
```

If there are multiple persistent messages cached for a group, the middleware pushes them one after another to any participant who joins the group late.

Additional APIs There are two additional APIs provided by Moitree to facilitate Data synchronization and computation offloading.

Data Synchronization API: Moitree has a sync API for synchronizing data between mobile and avatar. Developers can use this API to add any specific directory for synchronization. For example, there is a directory named "FaceData" on the storage of the mobile device, the developer want to synchronize this with avatar, she can use this API:

```
1 //get the full directory name first
2 String directory = Environment.getExternalStorageDirectory() + File.
     separator + "FaceData";
3 avatar.addSyncableDir(directory);
```

Android's FileObserver API is used to watch a directory for any new file creation or modification of a file. Once a file is created or modified, the Sync Service of the middleware starts to transfer the new file. By default, any file removed from

the mobile device is not removed from the avatar side. If the developer wants this behavior, then they have to use:

```
avatar.addSyncableDir(directory, SyncMode.PROPAGATE_DELETE);
```

Note that the Sync API provides basic synchronization feature for Avatar. An advanced file system [108] can be used for getting more sophisticated synchronization features on top of Avatar.

Offloading Communication API: This private API can only be used by a Moitree module developer. For example, a programmer can use this to write a pluggable computation offloading framework for Moitree. This is essentially a communication API which provides support for sending objects between the mobile and avatar pair. The developer can use multiple channels (for the same app) to send and receive objects. For example, in order to receive data on channel 1, one has to register:

```
int channel = 1;
avatar.setOffLoadingDataListener(channel, new OffloadingDataListener(){
  @Override
  public void onDataAvailable(Serializable data){
    //Use data
  }
});
```

The Avatar class maintains a map of channels and their listeners. To deliver an incoming object, it first finds the reference of the listener from the map and then delivers the data to it.

And, a programmer can use the following code for sending an object data to channel 1:

```
int channel = 1;
//To send a Serializable object named dataBundle via channel 1
avatar.sendOffloadingBundle(dataBundle, channel);
```

We can notice that the developer can send and receive any Serializable object as data. There is an alternative to Serializable interface in Android named Parcelable which

is proven to be faster than Serializable. But, it is more complicated to implement the Parcelable interface comparing to Serializable. We implemented and tested using both interfaces. But, we don't want to enforce our developers to use only Parcelable. We have used the more popular Serializable interface for this reason.

It's worth mentioning that the Offloading Communication Manager uses only one physical channel per app. The developer defined channels are managed by multiplexing and demultiplexing the data. A meta-data is attached to the data to indicate it's channel. This meta-data is again used for demultiplexing.

5.5 Moitree Middleware Design

5.5.1 Design Goals and Implementation Challenges

Moitree middleware has several important goals which are vital to keep the Avatar platform usable and efficient: (i) maintain a stable and seamless mobile-avatar pair, (ii) hide low level details of the underlying system, (iii) translate high-level Moitree API calls to low level instructions (i.e., provide execution environment), (iv) manage provision for computation offloading, and finally (v) glue all the participating entities together by taking care of the communications among them.

To achieve these goals, we have to overcome few challenges - some of them are inherent to any distributed systems, some of them are unique to the Avatar platform. We are discussing both types of challenges from Moitree's design point of view.

Loosely Coupled Design Avatar platform consists of different independently running components (e.g., apps, avatars, mobile devices, cloud services, etc.). They can run irrespective of other components' states. For example, a user's mobile phone should be able to function normally and run standard Android apps independent of the fact that the phone is currently unable to reach its avatar. Similarly, if the user's avatar is participating in a computation and it's mobile counterpart is unreachable, it should continue to run independently. Thus, one of the major requirement of

Moitree middleware design is to keep all its components loosely-coupled. This ensures the transparency of each components' responsibilities in a way that one component doesn't have to know how the emitted message/event will be handled by the other components. By keeping one component decoupled from others, they remain lightweight and the whole system works very fast and in an efficient way.

Heterogeneity Different components of Avatar platform have different execution environments - some of them runs on physical Android devices, some of them run on x86 VMs, some of them might run as plain Java service in the cloud. Even worse, some of the mobile devices even might be wearables (e.g., smartwatches) which has a different and very limited execution environment comparing to standard Android smartphones. The midddleware faces a major challenge to operate on top of such a complex heterogeneous system and to hide it from the high level API set used by programmers.

Resource Constraints Smartphones' resource limitation which is something Avatar platform wants to alleviate, is also a limitation for the Moitree. Parts of the middleware which will run on users' mobiles, cannot consume huge battery power, memory or CPU cycles for its management works. This restriction precludes the use of any heavyweight libraries in the middleware. This means, many of its building blocks has to be built from scratch.

Routing of Messages Users can install many Avatar apps on their devices. Each app can create many groups if it needs so. Groups can theoretically have unlimited members (although, there should be a practical limit). And, each group can create as many communication channels as it requires. To scale and support such huge amount of app-to-app communication is a great challenge. The middleware has to deliver the correct data to the exact channel of a specific group of a particular app

in a target device. There is device-to-device routing concern and then there is object level routing concern (e.g., objects for channels,groups). Moreover, everything has to be done in fast and energy-efficient way so that the users' mobile doesn't get affected after running few computations.

Fast Serialization and Communication Since Moitree provides high level API, programmers interact with it with high level objects. It is worth noting that the middleware has to meet real-time communication goal needed by games, real-time sensor based apps, etc. For this reason, the middleware must serialize/deserialize objects to/from low-level data in a fast and very efficient manner. Java serialization is known to be slow due to its use of reflection. Along with serialization, data transfer also has to be very fast. For this reason, very stable and widely used communication libraries such as Volley [55], Retrofit [111] cannot be used which are based on application layer protocols.

5.5.2 Message-Oriented Middleware

Considering the design goals and challenges, we have designed Moitree as a Message-Oriented Middleware (MOM) [43]. Other middleware design choice such as RPC-based middleware does not suite our design goals. For example, RPC-based middleware uses blocking calls and it expects different components of the system to be strongly coupled with each other.

In an MOM, messages/events are the driving force of the whole system. They work as system wide signals in the middleware. They carry information from one component to another about changes in state of the system. Based on the nature of the received message/event, the receiving component may change it's own state and emit another message/event if necessary. In MOM, messages are exchanged in asynchronous fashion. It means, the sender doesn't wait for the response after sending

the message; it can carry on its other tasks. This asynchronous working mechanism of an MOM makes it a natural fit for Moitree middleware design.

Events and Messages In a message-oriented middleware design, the term message refers to both event and message. However, we use event and message for two different purposes. In Moitree, events are used to denote commands (e.g., create a group, delete a member from the group, etc.), whereas messages represent data (e.g., data required for a computation, results produced by a computation, etc.). Although, the handling mechanism for events and messages are similar, they are handled by separate sub-components of Moitree. Both event and messages have embedded meta-data for routing them from one component of the middleware to another.

5.5.3 The Mobile-avatar Pair

A distributed app in Moitree starts execution in a mobile-avatar pair. This pair can be considered as the atomic computational unit of Moitree execution environment. Therefor, one of the major responsibilities of Moitree middleware is to maintain a stable and efficient mobile-avatar pair so that it works as a single computational unit.

Figure 5.3 shows how the mobile-avatar pair works in Moitree. The component named Moitree Mobile Middleware (MMM) runs on each mobile device and similarly, another component named Moitree Avatar Middleware (MAM) runs on each avatar. These two components manage event/message handling, sensor management, data synchronization, the network communication, etc. This component has several sub-components as shown in Figure 5.3.

5.5.4 Middleware Components

When a collaborative computation is performed by Moitree, many mobile-avatar pairs participate in the group computation to produce the end-result. Mobile-avatar pairs collaborate with each other using various other Moitree middleware shown in the

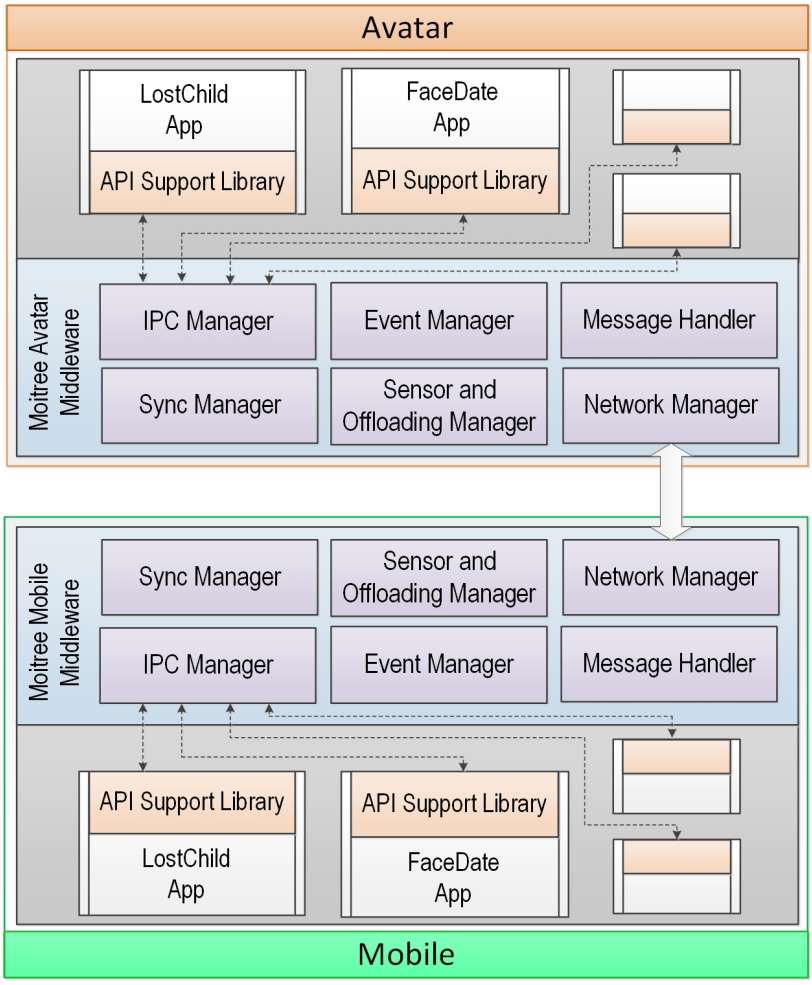


Figure 5.3 Mobile-Avatar pair.

Figure 5.4. Components of the mobile-avatar pair are shown in compact categories in this figure to show other new components in details.

API Support Library (ASL) This component presents the Moitree API functionality to an app. Programmers make API calls using this library. An API call is translated to a corresponding event/message and then sent to either the component MMM or MAM depending on where the API call is made. If the API call is made in the mobile, the recipient component is MMM, and for avatar, it is sent to the MAM. For example, when an app makes a createGroup() API call from

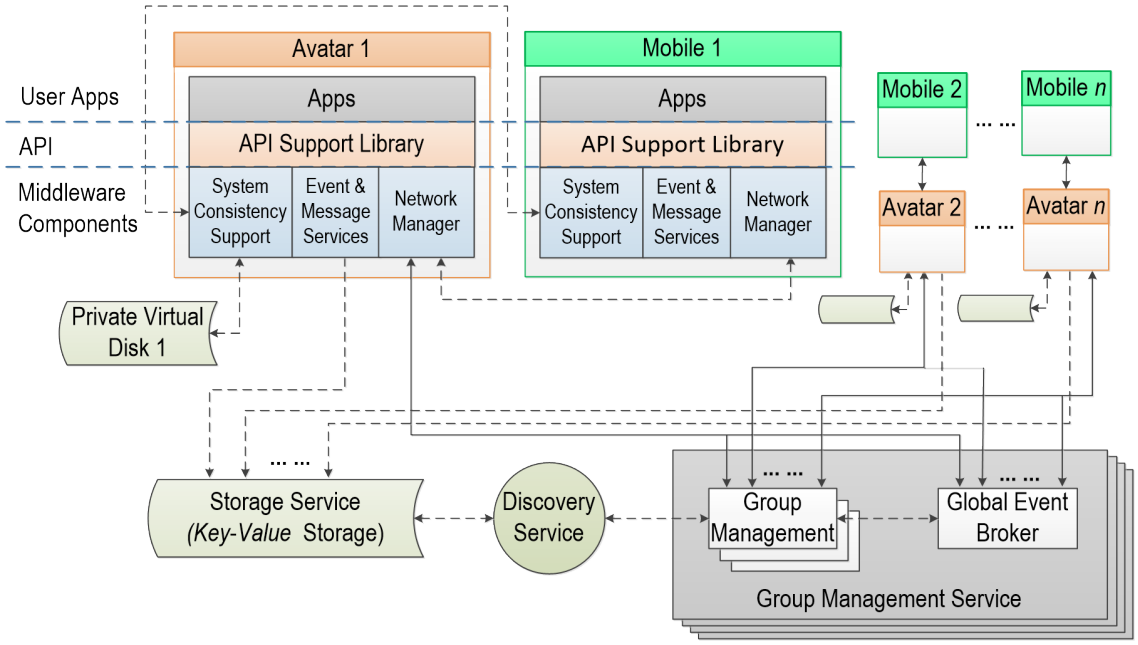


Figure 5.4 Components of the Moitree middleware.

the mobile, the ASL translates it to an event named *CREATE_GROUP* and sends it to the MMM. Since Moitree APIs are asynchronous and callback based, ASL is also used to register callbacks for a particular API call. ASL keeps record of this registration and whenever a response for the API call comes to the middleware, it uses the registered callback to deliver the response to the app. The response can vary depending on the API call. For example, it can be a data (e.g., a group ID for a createGroup API call), or it can be a result of a computation. Hence, the ASL works as a bridge between an app and the middleware. Three major tasks of the ASL are: (i) translating APIs to to internal events/messages and sending them to the MMM or the MAM, (ii) registering listeners for API calls, and (iii) delivering responses (data/results) to registered callbacks.

IPC Manager (IPCMan) Avatar apps and the middleware run as separate processes in a device. For example, the LostChild app shown in Figure 5.3 runs in one process and the MMM runs in another process in the mobile. Since the ASL

component is embedded in the app, it runs in the same process as the app is running. Inter-process communication (IPC) is needed to send and receive data between an app (i.e., the ASL) and the middleware. IPCMan takes care of any IPC communication made between the ASL and the middleware. When apps generate events and messages via ASL, this component acts as a gateway to the middleware and receives the emitted events and messages. On the other hand, when some incoming event/ message needs to be delivered to an app, IPCMan delivers them to the particular app via the ASL.

Event Manager (EM) An event queue and event dispatcher run as a part of the middleware in each mobile device and avatar. Any event received from apps by the IPCMan (e.g., $CREATE\_GROUP$), are subsequently enqueued in the event queue. The queue dispatcher listens for any incoming events in the event queue. Each event has meta-data embedded in it's header. The dispatcher uses this information to dispatch the event and forward it to the next relevant component. The Event Manager follows the observer pattern for achieving a loosely coupled design.

Message Handler (MH) This component is responsible for handling and delivering messages from the app to the middleware and vice versa. It is also designed to have a message queue and a message distributor. The Message Handler component works in a similar way to the EM. Once a message is received by the ASL, there is an extra demultiplexing involved to deliver the message to a particular data communication callback of a particular app (e.g., LostChild). ASL uses Intra-Device Routing header fields in the message to find the target channel and group of an app where the message should be delivered. The Message Handler Component also follows the observer pattern for maintaining the loosely coupled design.

Network Manager (NM) This component takes care of network communication between mobile-avatar and avatar-cloud services. Essentially this works as the

network gateway of the MMM and MAM. Any incoming event/ message from other devices comes to this component first. Mobile's NM listens for incoming packets from avatar. Avatar's NM listens for any incoming packets from wither mobile or any cloud service. The NM in the mobile listens to two different ports for incoming message and event. These two ports for event and message are system-wide known. NM in the avatar listens to two additional ports (4 ports in total) for any incoming packets. These two ports are opened for receiving event and message from cloud services.

Communication Manager for Real-time Apps and Computation Offloading Real-time apps (e.g., games, video conferencing) and computation offloading both require real-time communication between mobile and avatar. For this reason, these two components are separated from rest of the similar components. Essentially this is an extension of the Event Manager, Message Handler and Network Manager. Two separate ports are allocated for real-time apps and computation offloading. They also have their own message queues and dispatchers.

Sync Manager (SM) This component takes care of data synchronization between the mobile and avatar. Specific directories can be configured for this service. There is an API to add any programmer specified directory to be watched and synchronized.

Group Management Service (GMS) GMS is designed to handle group operations, events, and communication. GMS runs as cloud service on a group of dedicated servers. GMS supports hierarchical groups. Developers can use API for create a new group as a child of an existing group. For each group hierarchy, a group manager is used to maintain its tree structure, and a membership manager is used to maintain a list of the current members of each group or subgroup. These two modules are also responsible for handling requests, such as creating and managing

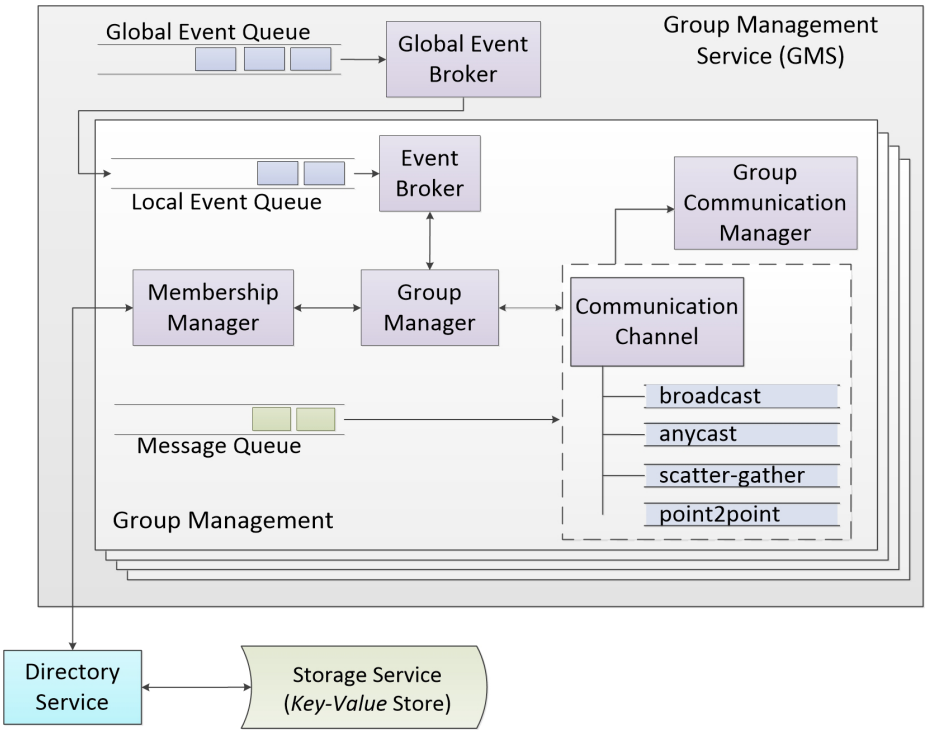


Figure 5.5 Group management service.

groups, changing membership, etc. Figure 5.5 shows the internal structure of the GMS.

To keep the data structures up-to-date, they use keep-alive messages to find out failures of the members; members are removed from groups when failures are detected. For each group, there is an event broker in charge of delivering events within the group, and a group communication manager to maintain communication channels and to forward messages to recipients. Thus, the handling of events and messages is separated to prevent a large number of messages to delay a few important events. The middleware gives higher priority to event handling compared to message handling because events are associated with important group/system state changes that must be reflected in real-time in apps.

Design optimizations may reduce the event/message forwarding workload on GMS servers. For example, forwarding can be offloaded to group leader avatars.

However, this requires that group and channel information is duplicated to these avatars. This may lead to privacy concerns and additional overhead to maintain the information consistent. Another optimization aiming to reduce the load on GMS servers is to save large messages into a shared key-value store. Instead of forwarding complete messages, the GMS servers just forward the keys of the messages. When an avatar receives a message key, it reads out the message from the shared storage. But, this method increases the workload of the storage service. Thus, we have not included these optimizations in our current implementation because they need additional experimental evaluation.

Storage Service (SS) SS provides a shared and permanent storage space for the middleware and is implemented as a key-value store. SS maintains an app registry, which serves the purpose of finding which app is installed on which user's device and avatar. An entry in the app registry is created when an Avatar app is installed on a user's mobile device and avatar. The registry entry contains the avatar ID and the app's name. Information about the users' location and time is also stored in SS to assist the Directory Service component. Finally, SS could be used for sharing large messages among participants in an optimized way. Each avatar has a virtual disk directly attached to it; these disks are not part of SS. Each disk serves as the private and primary storage for the avatar and the apps running on it.

Directory Service (DS) Directory Service is used by GMS to select appropriate candidates for a group. For example, DS provides answers for queries such as "which users have the LostChild app installed and were present in Times Square between 5PM and 6PM today?". The directory service uses Storage Service (SS) as its data repository. The mobile carriers can provide user's location and time data for to serve these queries.

5.6 Moitree Implementation

A prototype of Moitree is implemented using Java and Android. Although, the concept is generic and can be implemented using other programming languages and frameworks.

API Support Library (ASL) This is implemented and distributed as an Android library. It has to be included in an Android project to be able to call any Moitree API (described in Section 5.4). Moitree API calls can be started with instantiating an object of the Avatar class which is defined in the ASL. The instantiation can be done in the main Activity class for an app with single Activity or in the Application class for multi-activity app (and share this instance among activities). Additionally, it can be instantiated in a background service.

IPC Manager (IPCMan) Android's Binder mechanism is used for IPCMan. Each Avatar app works as a Binder client and IPCMan in ADM works as a Binder service. When an app runs, it uses ASL to find the IPCMan Binder service. Using ASL, it binds to the service and sends an initiating message along with it's appId. IPCMan keeps track of the appId and the reply messenger reference (obtained from the initiating message). IPCMan uses this information to deliver any event/message sent to that particular app. Earlier versions of Moitree used Android's intent-based BroadcastReceiver mechanism for IPC. But, it was found to be slow for real-time apps which send huge number of messages within very short interval. However, it suffices for infrequent event transmission. Events are very small objects and there is no significant performance difference between Binder and BroadcastReceiver for event communication. For this reason, Moitree still uses the intent-based BroadcastReceiver mechanism for transferring events.

Network Manager (NM) The network manager in the mobile listens to two different ports for incoming message and event. These two ports for event and message are system-wide known. The NM in an avatar listens to two additional ports (4 ports

in total) for any incoming packets. These two ports are opened for receiving events and messages from the GMS cloud service. Events such as JoinGroup come from the GMS to the avatar (to request an app to join a group computation). And, messages (data) distributed to a group also comes from the GMS to the avatar. The NM uses TCP clients to send data. As the event and message ports are known to all other components of the middleware, the client connects to the target devices' specific port and delivers the packet. Moitree uses a TCP library named Kryonet [12] in the NM.

Cloud Services The Group Management Service (GMS) is implemented using Java and Kryonet library. The Storage Service (SS) is implemented using the Redis [20] key-value database because it is fast, reliable, and can work on both a single node and a multi-node configuration.

5.6.1 Message Routing in Moitree

Moitree manages two types of message routing inside the middleware. The Inter-Device routing manages how a message from one device or cloud service (e.g., mobile, avatar, GMS, etc.) will be routed to another. Whereas, the Intra-Device routing defines and manages how a message will be delivered to a particular callback function of a group inside an app. Meta-data attached with events/messages is used to manage these two routing mechanisms. These meta-data are embedded in the message header of events/messages. Table 5.4 shows a typical message header object used by Moitree internally. The table summarizes when a header field is set and used by a component.

Field *messageType* is an object of AvatarMessageType enum. This field along with other two fields *source* and *channelType* defines the Inter-Device routing behavior of the message. *messageType* can be set by the programmer. For example, a programmer can tag a message with $AvatarMessageType.DATA$ during the scatterGather channel API call. This will define the destination of this message to be participant users' avatars. Whereas, using $AvatarMessageType.DATA\_MOB$

Table 5.4 Avatar Message Header

Header Field	**Set By**	**When**
appId	App	instantiation
groupId	GMS	group creation
channelId	App	data communication
channelType	App	data communication
source	App, MMM, MAM, GMS	during routing
messageType	App	data communication
isPersistentData	App	globally persist
isLocallyPersistent	MMM, MAM	locally persist
receiverID	App	send P2P data

will define the flow to send the data to participants' mobiles. Table 5.5 shows the summarized view of this flow-based destination defined by the *messageType* field.

Table 5.5 Inter-Device Routing Flow

messageType	**Destination**
$DATA$	avatar
$DATA\_MOB$	mobile
$RESULT$	avatar
$RESULT\_MOB$	mobile
$SYNC$	user's avatar (for data sync)

The other two fields *source* and *channelType* in Table 5.4 define the intermediate device/components of the device-to-device route.

First 3 fields (appId, groupId, channelId) of the message header shown in Table 5.4 defines the Intra-Device Routing. Once the event/message is routed to the

correct device by the Inter-Device routing logic shown above, these three fields defines which app's specific channel of a particular group is the recipient of the message.

5.6.2 Example of Event Handling in Moitree

To explain how an event is handled by multiple components of Moitree, let us consider an example of creating a group. Figure 5.6 shows a sequence diagram for this group creation example.

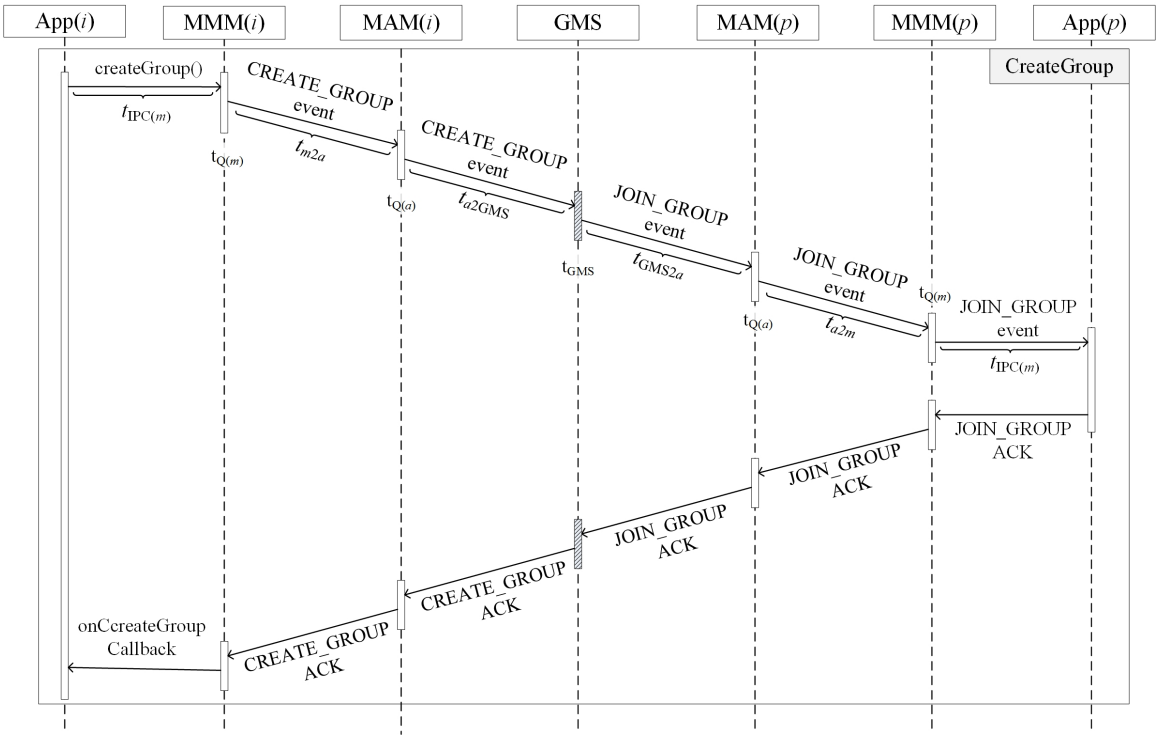


Figure 5.6 Sequence diagram of a group creation using Moitree API.

When the LostChild app in the mobile calls an API for creating a group, the ASL generates an event *CREATE_GROUP* for it and uses IPC to send it to the MMM. The IPCMan in MMM receives the event and then pushes it to the event queue. The event queue dispatcher then reads the routing header of the event, and figures out that it has to be delivered to the avatar. It forwards the event to the next destination, the Network Manager. The opposite happens in the Avatar for

this event. The Network Manager of MAM receives the event, and then the event is pushed to the event queue. The event queue dispatcher will figure out that this has to go to the GMS cloud service, so it forwards the event to the Network Manager which takes care of sending it to the GMS. GMS retrieves the criteria of the group formation and contacts the directory service to get a list of eligible participants. It then generates another event named $JOIN\_GROUP$ to the participants. Let's assume one participant has agreed to join the group by sending an ACK event in response to the $JOIN\_GROUP$ event. This ACK will follow a reverse path to reach the GMS which will in turn send an ACK for the $CREATE\_GROUP$ event to the initiator. This ACK will finally reach the LostChild app in the intiator's mobile.

From the Figure 5.6, we can approximate the total time needed to create a group,

$$\begin{aligned} t_{CG} = 2 * t_{ASL(mi)} + 2 * (t_{IPC(m)} + t_{Q(m)} + t_{m2a} + t_{Q(a)} + \\ t_{a2GMS} + t_{GMS} + t_{GMS2a} + t_{Q(a)} + t_{a2m} + \\ t_{Q(m)} + t_{IPC(m)}) + t_{ASL(mp)} \end{aligned} \tag{5.1}$$

where $t_{ASL(mi)}$ is the time needed to convert the createGroup() API call by the ASL in intiator's mobile. $t_{IPC(m)}$ is the time needed for the $CREATE\_GROUP$ event to reach MMM from ASL via IPC. $t_{Q(m)}$ is the queuing time needed in the MMM. t_{m2a} is the network transmission time from the mobile to avatar. $t_{Q(a)}$ is the total queuing delay in the MAM. t_{a2GMS} is the network transmission time for the event to go to GMS from the avatar. t_{GMS} is the event processing time by the GMS. t_{GMS2a}, $t_{Q(a)}$, t_{a2m}, $t_{Q(m)}$, $t_{IPC(m)}$, $t_{ASL(mp)}$ are similar and self-explanatory notations to the components of first part of the equation.

Although, mobile and avatar of the initiator are different than participants', for estimation purpose, we can assume $t_{IPC(m)}$, $t_{Q(m)}$, $t_{Q(a)}$ are almost equal. And, t_{m2a}, t_{a2GMS} should be same as t_{a2m}, t_{GMS2a} respectively. Hence, equation 5.1 can be

simplified to:

$$t_{CG} = 2 * t_{ASL(mi)} + 4 * t_{IPC(m)} + 4 * t_{Q(m)} + 4 * t_{m2a} + \\ 4 * t_{Q(a)} + 4 * t_{a2GMS} + 2 * t_{GMS} + t_{ASL(mp)} \tag{5.2}$$

5.6.3 App Execution in Moitree

Figure 5.7 shows different states of an Avatar app and their transitions from one another. There are essentially six states of an app which are directly involved with Moitree. There are some implicit Android life-cycle states (onPause, onResume, etc.) that can be handled by the programmers for making their app more user-friendly and efficient. However, we will only discuss the relevant six Moitree related states.

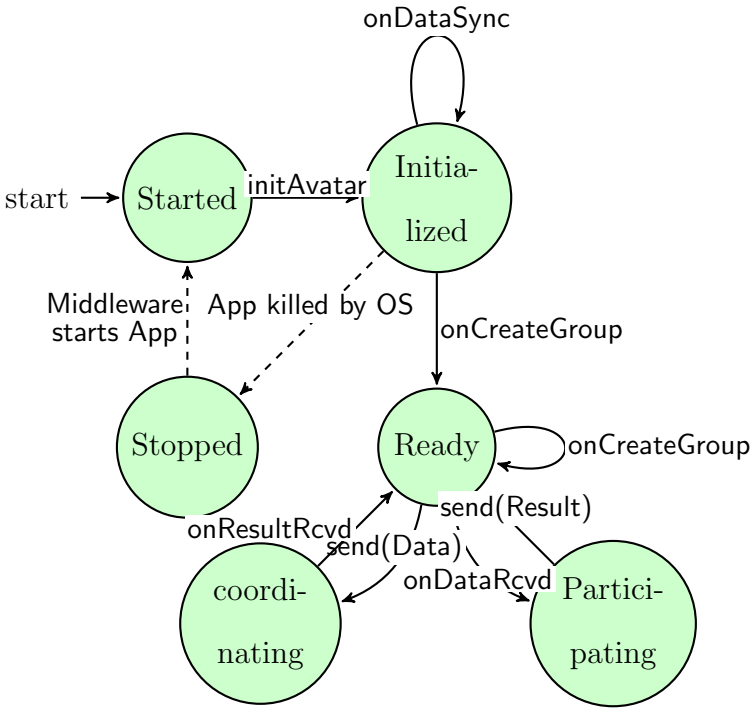


Figure 5.7 State diagram of an Avatar app.

When the user runs an Avatar app on her mobile, the app enters the "Started" state which is equivalent to Android's onStart life-cycle state. Once the avatar instance gets initialized, and callbacks are registered by calling Moitree APIs, the app enters "Initialized" state. In this state, it can react to events such as data

synchronization (onDataSync). For LostChild app, the app can retrain/update its face recognizer with newly synced data.

Once the app receives a notification event "onCreateGroup" in the group callback function, it means it is part of a group computation now. It enters the "Ready" state. It can be noted that one app can be part of many groups. In the "Ready" state, it can receive more "onCreateGroup" event and become part of other groups. All the groups are maintained in separate thread and separate data structure, so the same app can contribute to multiple group computation.

In the "Ready" state, group initiator can send data via communication channels which takes the app to the "Coordinating" state. It can receive the result from participants in this state and go back to the ready state again. In the "Ready" state, participants can receive data in their communication channel callbacks, which will take the app to the "Participating" state. It will compute on the data and can send back the result which will take the app back to the "Ready" state again. It can be noted that app can have multiple active groups and those groups can be in "Coordinating" and "Participating" states. For this reason, "Coordinating" and "Participating" can be called group states rather than app states.

Android can kill the app if the mobile runs low on memory. In that case, the app enters the "Stopped" state. Then, the middleware can start the app again if necessary. The state transition are shown with dashed lines are therefore optional states. This means, these two transitions can happen only if the OS interrupts (i.e., kills) the normal flow of the app-states.

5.7 Deployment

The Avatar platform along with a Moitree prototype is deployed in our private cloud comprised of 8 server machines each running on Intel Xeon E5-2620 CPU and 80GB of RAM.

5.7.1 Deployment Setup

GMS Deployed as a cloud service. It is a collection of java programs running on few physical servers in the cloud. Multiple instances of GMS are deployed to balance load. If multiple cloud is deployed in different geographic locations (e.g., in New York, San Francisco, London, etc.) in the future, even more GMS instances will be run in those locations to balance the load.

Other Cloud Services Directory Service is another cloud service deployed in our private cluster. It is developed using Java and runs on top of a Redis server.

Hosting VMs Avatar VMs are hosted in the OpenStack cloud in our private cluster. We deployed VMs using different versions of Android x86. For most recent testing, we have used Android Marshmallow (version 6.0) based VMs. Although, we have both Android Kitkat (v 4.4) and Android Lollipop (v 5.0) based VMs deployed in the cloud.

Distributing The API Support Library The API Support Library is uploaded to our private maven repository. Programmers can use a single line in their gradle build script to get this library imported in their Android project: *compile "edu.njit.avatar.sharedlib:edu.njit.avatar.sharedlib:1.3.14"*. Both stable and well-tested version and nightly builds are uploaded to the repository. Programmers can import and try either of them.

5.8 Performance Evaluation

We evaluated Moitree middleware and the API set by using various macro-benchmark and micro-benchmark tests. The experimental evaluation has some goals: (1) validate the concept of Avatar platform by showing improvement in running-time and resource usage, (2) verify the effectiveness of the programming model to reduce programming

complexity and efforts by implementing two apps, (3) test the efficiency of the API support library, and (4) measure the scalability, efficiency and overhead of the middleware.

For the experiments, we used Nexus 5, Nexus 6, Nexus 5X, and Moto X Pure mobile devices. Avatar VMs used for the experiment are configured with 6 virtual CPU cores and 3GB of RAM.

5.8.1 Macrobenchmarks

We have done some macro-benchmark tests to ensure Moitree will not cause any issues to users' mobile devices by slowing it down or consuming huge resources by incurring significant overhead. The goal of these macro-benchmark tests is to prove that: (1) Moitree API does not introduce significant delay when users launch apps (i.e., the app loading time remains almost the same even after using Moitree), (2) Moitree middleware does not leave large footprint in the phone memory, uses very little CPU, and consumes a small amount of power, (3) Avatar platform and Moitree reduce apps' running time significantly.

Application Loading Time For developing distributed apps using Moitree, programmers include API Support Library (ASL) in their apps. To call any Moitree API function, an Avatar object instance is required to be obtained. This is considered to be the Moitree API initialization step. This incurs a little delay in the app loading time. To measure this delay, we instrumented an Android test app with and without Moitree initialization. First, we measured how long it takes to fully load the standard Android app. It is worth noting that once the user clicks on an app's launcher icon, Android's ActivityManager starts to load the app and eventually makes the main Activity visible. This step includes loading standard Android support framework and initializing (i.e., making visible) the app's UI. We measured both by using ActivityManager's log. Next, we measured how long it takes to fully load the same

app with Moitree initialization. The Moitree API initialization step includes tasks such as registering IPC mechanism (e.g., connecting to the IPC service), initializing necessary data structures, etc.

Table 5.6 shows the average time for initializing aforementioned phases. Android's support framework and UI initialization take 228.07 mS and 41.87 mS respectively. Moitree API takes a mere 4.07 mS to get initialized. Notice that Moitree API initialization takes significantly small amount of time comparing the time taken by Android's library and UI initialization. This proves that users will not notice any significant delay during the app loading.

Table 5.6 Moitree API Initialization Time Compared to Android App Loading Time

	Android Framework Initialization (mS)	Moitree API Initialization (mS)	Android App UI Initialization (mS)
Average	228.07	4.07	41.87
St Dev	23.55	0.46	4.97

Resource Usage Moitree middleware itself consumes very little resources on the mobile device. The Moitree Mobile Middleware (MMM) component running on the mobile consumes 22.60 MB of memory. And, the MAM running on the VM consumes 6.10 MB of memory. This shows that the memory footprint of Moitree middleware is insignificant comparing to the available memory in popular Android phones which generally ranges from 1GB to 3GB.

We also measured the CPU usage by the Moitree middleware during both initialization and idle state. Figure 5.8 shows the MMM component running on the mobile uses a small amount of CPU during initialization. It's worth stating that this step includes starting various background services such as the NetworkManager, IPC Manager, Queue Managers, etc (components shown in Figure 5.3). Once initialized,

the MMM registers almost no CPU usage in the idle state. CPU usage is measured using Qualcomm's Trepn profiler [99] on a Nexus 6 phone.

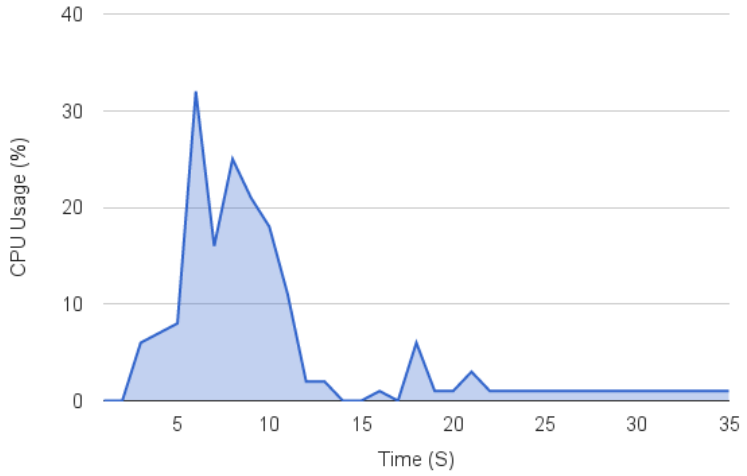


Figure 5.8 CPU usage of the Moitree middleware (MMM) in mobile during initialization and idle condition.

Next, we measured how much power Moitree consumes in the mobile during the initialization. The MMM running on mobile device consumes only 0.16 mAh power during the initialization step. This measurement is done using Qualcomm's Trepn profiler [99] on a Nexus 5X device which has a battery capacity of 2700 mAh. This means, MMM can be started and initialized about 16875 times with a fully charged battery.

Running-time Improvement To understand the real-time performance of Moitree apps and the effect of avatars on latency, we measured the response time for the LostChild app in two scenarios: (1) the major workload in the app, including face detection and recognition, is handled on the mobiles, and (2) the major workload is executed at the avatars. Let us note that mobile to mobile communication in the first scenario is mediated by Moitree services in the cloud. Figure 5.10 shows the end-to-end response time from the time of submitting the initial request until the time of receiving the final results. The figure also shows the breakdown of the latency

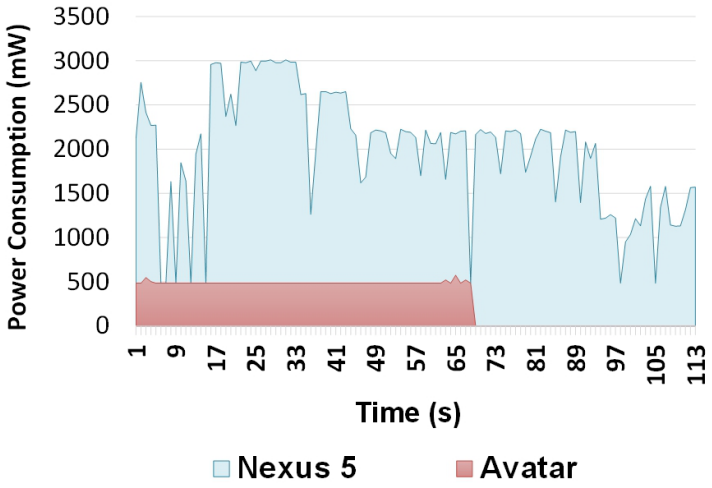


Figure 5.9 Power consumption comparison for participant *LostChild* app phone with and without avatar help.

between the face detection/recognition operations and the networking/middleware operations. In these experiments, we used one mobile device as initiator and three other mobile devices as participants. All the avatars were instantiated on the same server. In this experiment, each avatar VM runs Android 6.0 and is configured with 6 virtual CPUs. Each participant has a database of 47 images containing 60 faces stored in her avatar. In addition, each participant returns a result because all participants have photos of the lost child. The training process for face recognition is done before the app starts.

The results demonstrate that avatars help reduce the end-to-end response time to half when compared to the scenario where the mobiles handle the major workload. A substantial part of this improvement is due to offloading the computation for face detection and recognition to the avatars. We also observe that the latency incurred by Moitree and networking is reduced by 14.4%; this is due to offloading the communication part in these workloads to the cloud.

Figure 5.11 shows that the response time when we varied the number of participants from 2 to 7. In this experiment, the face detection and recognition

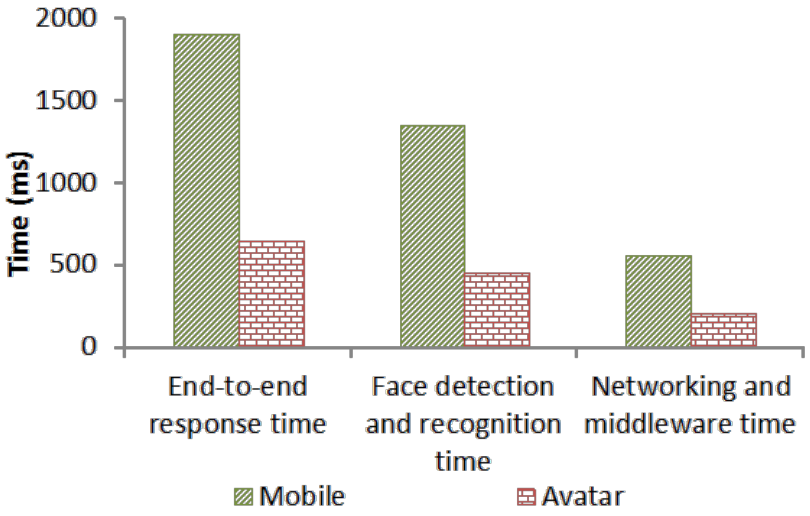


Figure 5.10 End-to-end response time for *LostChild* app when the workload is at the mobile and avatar, respectively.

are executed at the avatars. The experiment was conducted for two scenarios: (1) all the participant avatars run on a single server; and (2) each participant avatar runs on a different server. All the other parameters are the same with those on the previous experiment. The figure plots the median response latency and the latency of the last received response as experienced by the initiator. Let us recall that in this experiment each participant sends a response. In a real-life situation, most participants are not expected to send responses. Therefore, the curves for the latency of the last received response represent the worst case scenario.

The results show that the absolute values are reasonable (generally, between 500 ms and 700 ms). In addition, the application scales well with the number of participants for this experiment. The number of participants has almost no effect on the median latency values. However, the latency of the last received response is affected by the number of participants. We found two reasons for this problem. First, our current Moitree implementation sequentializes the communication among the members and adds a few of milliseconds to every message transmission. Second,

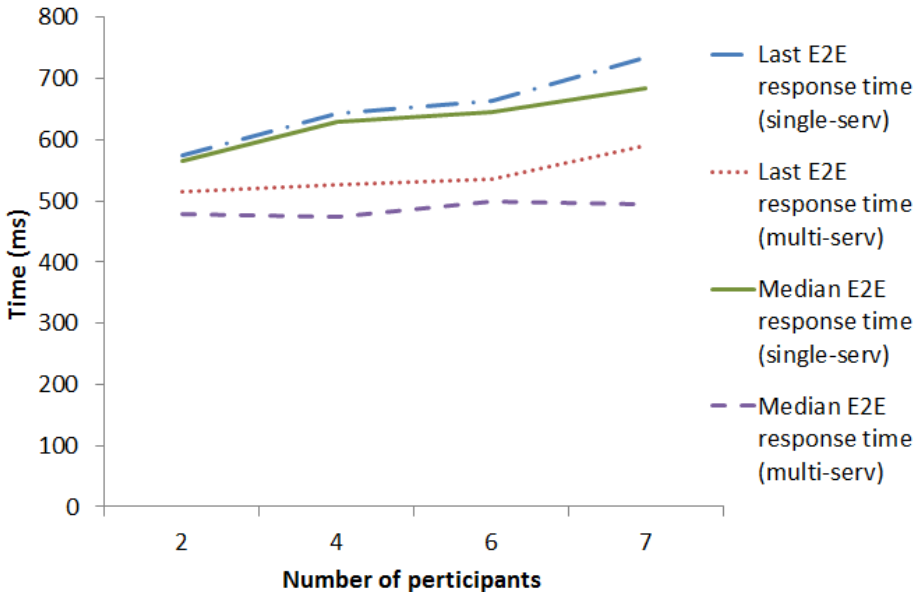


Figure 5.11 End-to-end response time for *LostChild* app vs. number of participants: (i) single-serv: all participant avatars run on one server; (ii) multi-serv: each participant avatar runs on a different server.

avatars do not send responses at the same time; in most experiments, we noticed one or two stragglers. We are working on improving the message delivery system of the middleware and planning more experiments to better understand the second problem. Finally, let us note that running one avatar per server improves the response time and, as expected, is relatively constant. Running all avatars on one server, on the other hand, leads to higher latency and this latency increases with the number of participants. This is caused by the resource contention incurred by the increasing number of avatars on the same machine.

5.8.2 Microbenchmarks

We ran some micro-benchmark tests on Moitree to evaluate its efficiency and scalability.

IPC Performance For every event/message communication, several IPC calls take place throughout the system (e.g., between ASL and the middleware). This

can potentially be a major bottleneck for real-time apps and apps which needs a huge amount of data communication. For this reason, we have measured the IPC performance in Moitree.

Early version of Moitree prototype used a lightweight Android mechanism called BroadcastReceiver (BR). It was essentially designed for sending infrequent notifications or updates from one app to another. It couldn't cope with the demands of real-time apps. Hence, we use only BroadcastReceiver (BR) for transmitting events which are infrequent. For data communication, we use a stable and efficient IPC mechanism called Binder in the current prototype. Android uses Binder driver from kernel for faster IPC. We evaluated both version of IPC mechanism used in our Moitree prototype.

We ran two different experiments to evaluate the IPC mechanism. First, we used a test app to send a single data packet from the app to the ADM component of the middleware. Once the data packet is received by the ADM, it sends back the exact same packet. In the app side, we count the total round-trip time and divide it by two to get the time for one way IPC. We varied the data size of the packet from 0.5KB to 500KB to simulate different sized data. We did the same experiment by using both BR and Binder. Results shown in Figure 5.12 shows: (1) IPC mechanism built on top of Binder is almost two times faster than the version using BroadcastReceiver, (2) IPC mechanism in Moitree is scalable for data size variation.

Second, we did a similar test keeping the data size constant and varying number of concurrent packets submitted to the middleware. We used data packet of size 1 KB in this experiment. Results in Figure 5.13 shows that IPC mechanism of Moitree also scales well for concurrent data communication.

Serialization and De-serialization Performance Since Moitree provides a high level API set, it always receives and returns high level objects (e.g., Java objects).

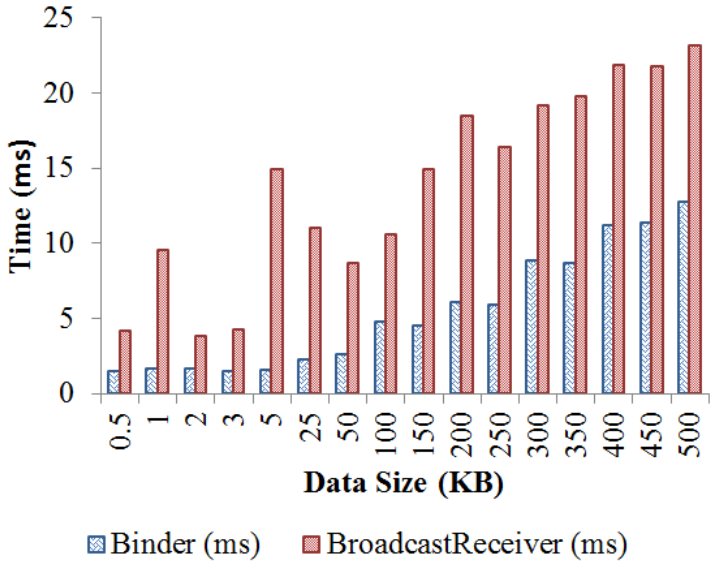


Figure 5.12 Comparison of IPC mechanisms used in the middleware. Time measured for transferring data with varied size.

But, the middleware has to use IPC and network for transmitting these objects from one component/device to another. For this reason, there are a huge number of serialization and deserialization take place inside Moitree middleware. Thus, serialization can be another potential bottleneck for the whole platform. Java serialization process is known to be slower due to the use of run-time reflection mechanism. We avoided using Java serialization to the greatest extent possible. Instead, we used Android's Parcelable interface which uses some boiler-plate codes to specify how the serialization/deserialization should be done on that object. This mechanism makes Parcelable to work much faster than the standard Java Serializable interface. All the objects used inside Moitree which are transferred from one component from another are written using the Parcelable interface.

For network transmission, we have used Kryo serializer which comes with the Kryonet communication library. Kryo serializer is also much faster than the standard Java Serializable interface [22]. We ran an experiment where a test app sends data packets of different sizes from the mobile to the same app running on the

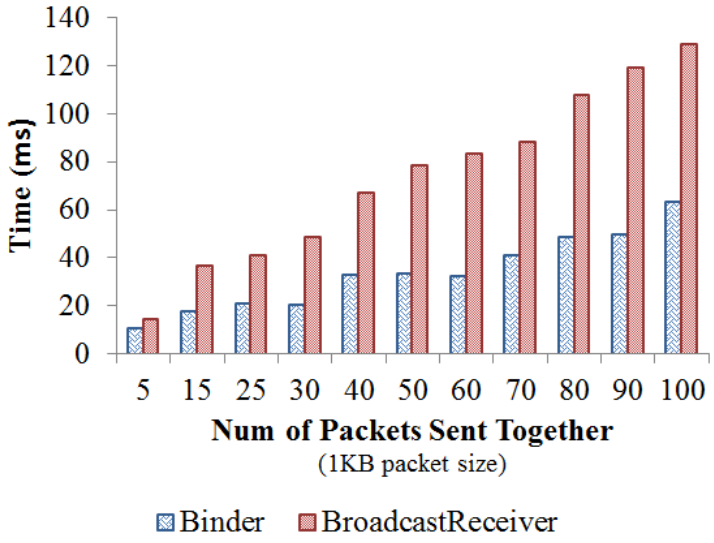


Figure 5.13 Comparison of IPC mechanisms used in the middleware. Time measured against number of packets sent at a time. Packet size is fixed at 1KB.

avatar. We measured the serialization time in the NetworkManager of Moitree Mobile Middleware (MMM) and deserialization time in the NetworkManager of Moitree Avatar Middleware (MAM). Figure 5.14 shows the result which suggests that the Kryo serializer used in Moitree is significantly fast.

Performance of API Support Library For the API Support Library's scalability tests, we measured end-to-end latency for concurrent API calls. A test app is used to call random APIs from mobiles, and then measured the latency to complete these calls. The API calls start at the App layer in Figure 5.4, pass through the API and middleware layers in mobile and avatar, reach the GMS service layer, and then the results are returned via the reverse path.

Figure 5.15 shows the latency of the concurrent API calls. We used up to 100 API calls at the same time, which is a significant load. The aim of the experiment is to assess the delay imposed by the middleware to process the calls. The results show that even 100 concurrent API calls can be resolved in 2.4ms. Next, we instrumented

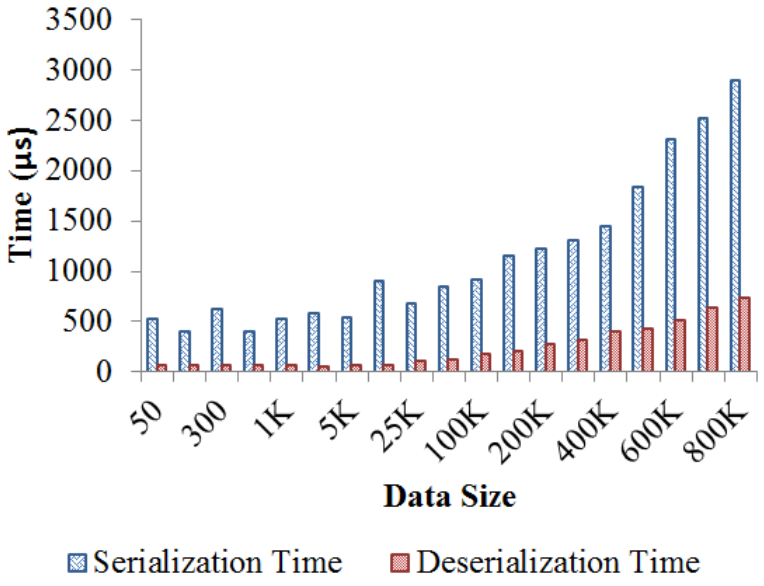


Figure 5.14 Serialization and deserialization delay.

the middleware and discarded the communication time (used by the kryonet [12] library). The results are shown in Figure 5.16. We see that the execution of the API calls remains approximately constant at about 320 s. This demonstrates that Moitree scales well at a load of up to 100 concurrent API calls.

Power Consumption Overhead Our earlier macro-benchmark result showed that Moitree middleware uses very little CPU cycles during idle state. In order to maintain the mobile-avatar pairing, the two ADM components periodically checks for data synchronization and keeps waiting for communication requests. That is why we see the low energy usage in Table 5.7. This result shows that energy consumption on mobiles in the idle state is negligible for all practical purposes. Similarly, Table 5.7 shows that the energy consumed per average API call is very low (a full battery charge allows one and a half million calls). In terms of data transmission, Table 5.7 shows that Moitree introduces a relatively high overhead when compared with plain TCP.

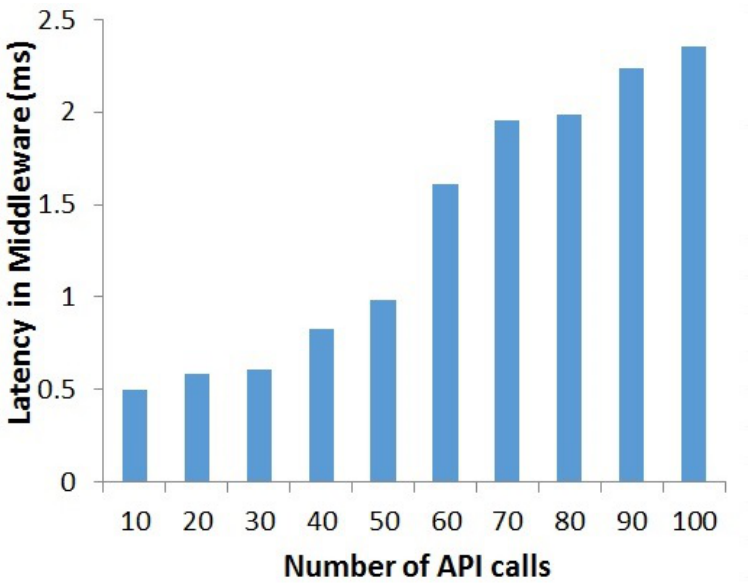


Figure 5.15 Average end-to-end latency for concurrent API calls in Moitree (including network communication).

This is due to the Kryonet [12] communication library, which simplifies programming at the cost of overhead. Nevertheless, the absolute values are still low.

Table 5.7 Moitree's Energy Consumption on Phones

Component	**Energy/Power Consumed**	**Comment**
MMM in idle state	5.5 mJ/sec	Middleware could run for two and half month before draining the battery
Moitree API calls	2.3 mJ/call	One and half million API calls with a full battery
Data transfer by middleware & Plain TCP	0.5 mJ/KB & 0.15 mJ/KB	Energy consumed in addition to WiFi being ON for the transfer

Next, We measured the power consumption overhead on mobile caused by Moitree during data communication. We sent many data packets concurrently by an Avatar app to simulate unusual and large load on the middleware. We used data packet of size 50KB. We measured the power consumed by the mobile middleware

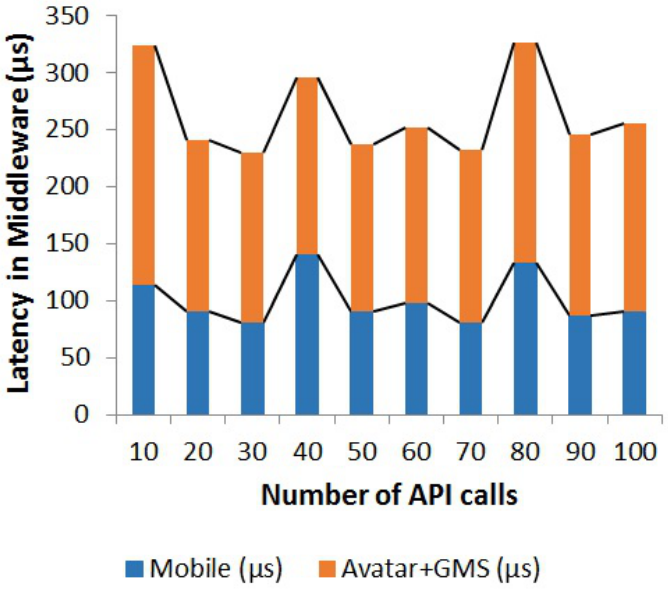


Figure 5.16 Average processing time for API calls in Moitree on mobile and avatar (no network communication).

(MMM) using the Trepn profiler. Next, we used another app which we call a Naive app as it does not use Moitree, rather uses only the Kryonet library to send/receive data. We sent data using this app in the similar way and measured its power consumption. The difference in these two app's power consumption can be considered as the overhead caused by the middleware. Figure 5.18 shows the result of this experiment. We can see that MMM consumes a little more power than the naive app. This is reasonable considering that the naive app only uses a single app to app network communication. Whereas, MMM supports many app to many app data communication by making routing decisions. For this reason, MMM uses queues, dispatchers, routing mechanism, etc which incurs a little overhead seen in the Figure 5.18. Although, the absolute difference is significantly low.

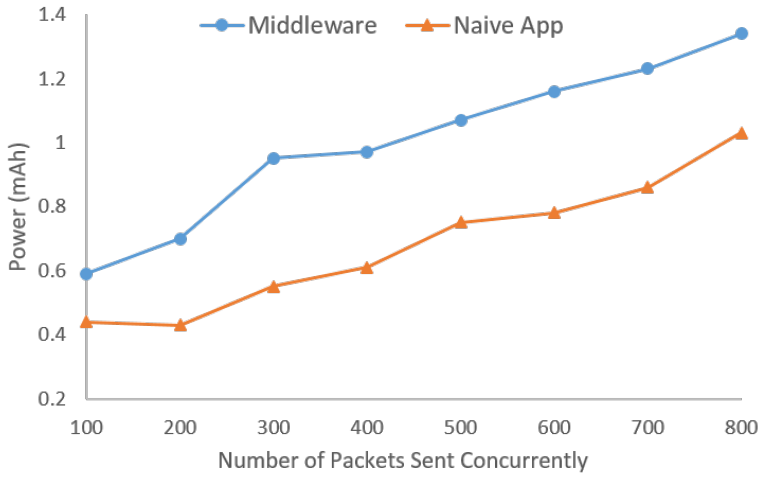


Figure 5.17 Comparison between power consumption caused by sending data concurrently using the middleware and using an app without the middleware. The packet size is 50KB.

Table 5.8 Number of Lines of Code for Our Apps Using Moitree and JXTA

Application	Moitree	JXTA
Lost Child	85	178
Video Conferencing	100	219

5.8.3 Programming Effort Comparison

Although the API Support Library (ASL) along with MMM works very effectively and it scales well to large loads, it is necessary to evaluate the programming effort reduced by Moitree API.

To quantify the benefits of the Moitree programming model, we used the LostChild app and a video conferencing app. In the video conferencing app, users share real-time video streams with friends, family or acquaintance. These three types of groups have different levels of permissions. Friends and family have permissions to see all the streams, while acquaintance can view only selected streams. Both apps

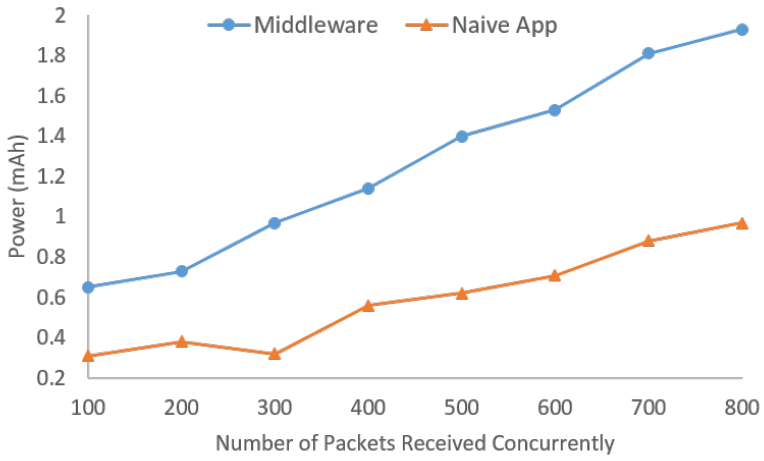


Figure 5.18 Power consumption overhead caused by receiving data using the middleware compared to an app which does not use the middleware. The packet size is 50KB.

are typical mobile distributed apps that may involve a large number of mobile users. Their implementations must deal with the common issues that are usually faced in mobile distributed apps (e.g., identifying and coordinating groups of participants). Therefore, they can act as a good test for Moitree.

We compared two implementations of each app: one done using Java and JXTA [52], and one done in Java and Moitree. JXTA is selected to compareto Moitree for two reasons: (1) JXTA is designed for peer-to-peer systems, in which peers are conceptually similar to sets of autonomous avatar/mobile pairs, and (2) it also has group concepts, although different from those used in Moitree.

For the two implementations of each app, we compared the sizes of their source code. In this comparison, we only counted the lines written by our programmers. The code in other libraries (e.g., OpenCV [15] for face recognition and Kryonet [12] for network communication) is not counted toward the effort to develop the app. The app implementations include mostly group management and group communication features; the rest is done through library function invocations.

We show the numbers of lines of code (LOC) in Table 5.8. We found that Moitree decreases LOC by a factor higher than 2. This is a promising result that illustrates how Moitree can simplify the programming of mobile distributed apps. Currently, we are implementing several additional apps for a more thorough evaluation.

5.9 Chapter Summary

This chapter presented Moitree, a middleware and a high-level programming framework for collaborative computation in cloud-assisted mobile platform. We implemented Moitree on top of our Avatar platform and tested it with two apps. The results of our evaluation are promising. Moitree is able to reduce the number of lines of code to less than half when compared to an existing solution. In addition, Moitree scales well when multiple APIs are invoked concurrently and helps users save energy on mobile devices at the cost of a reasonable latency overhead.

CHAPTER 6

CASINO: COLLABORATIVE SENSOR-DRIVEN OFFLOADING SYSTEM

Computation offloading has been successfully used by many research studies for a long time. Heavier computation is generally offloaded to the cloud. It is proven to reduce both the power consumption in mobile devices and the latency for completing the computation. While many studies [42, 67, 69] were able to provide efficient frameworks for offloading computation from mobile to cloud, they are limited to using only a single mobile-cloud entity pair. These frameworks are not designed to work with a collaborative/distributed computation scenario.

6.1 CASINO Overview

A distributed app in a mobile-cloud platform is executed on the mobile devices and cloud entities (avatars) of a group of users. The app can be divided into many smaller tasks/jobs. The overall completion time of the distributed computation depends on how efficiently the smaller jobs are executed by utilizing both local and cloud resources. In other words, a well coordinated usage of computation offloading can significantly improve the total completion time of the distributed computation.

This chapter presents CASINO, a collaborative sensor-driven offloading framework, which optimizes the total completion time of a distributed app by scheduling and executing smaller jobs in an efficient manner. CASINO provides a simple programming framework which can be used by programmers to partition their apps both statically and dynamically. Although static partitioning is generally less effective, it can be helpful in some situations (e.g., user-interaction tasks should be statically partitioned to the mobile devices). On the other hand, programmers can annotate parts of their code as "offloadable" to utilize dynamic partitioning.

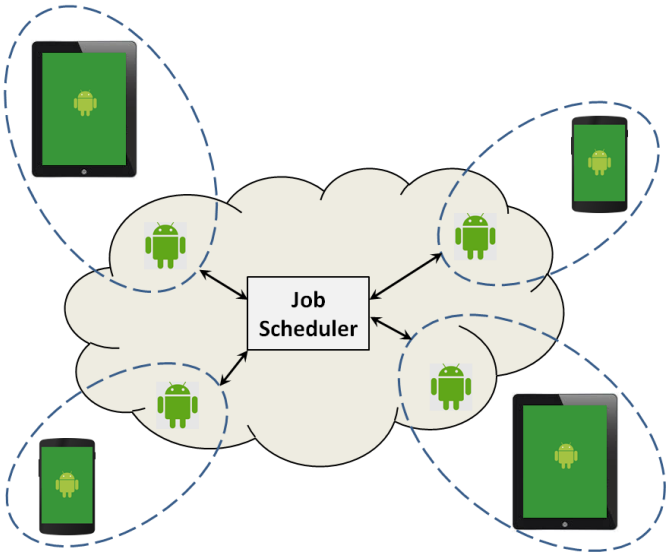


Figure 6.1 CASINO's job scheduler is serving a distributed app running on four mobile-avatar pairs.

CASINO profiles user devices for collecting networking, CPU, battery status. This profiling information is sent to a cloud service named job scheduler (shown in Figure 6.3). The job scheduler does not handle a job directly. Rather it handles a representation of a job. The actual jobs are the code components residing on user devices (mobile or avatar). Whereas, a representation of a job is just an object-oriented instance which holds the reference and information about the actual job. When a distributed app starts execution, CASINO enqueues representations of all the *offloadable* jobs in a job queue. The scheduler then resolves the dependencies between jobs and calculates an ordered sequence of jobs so that any job starts its execution only after its predecessor job has finished. Some jobs can also be dependent on a specific user device. For example, a user can have private photos in her mobile which are not synchronized with her cloud entity. Thus, any image processing job on those photos is dependent on her mobile device. The scheduler then estimates the best execution time of all jobs considering both device and job dependencies. The execution time estimation also includes any data communication delay involved with

the job execution. For example, jobs J_1, J_2 are estimated to run fastest on $mobile_1$ and $avatar_2$. J_2 depends on the output of J_1 and there is a communication cost (i.e., delay) to deliver the output data from $mobile_1$ to $avatar_2$. The job scheduler schedules jobs considering both these computation and communication cost. Hence, CASINO provides dynamic partitioning of a distributed app which minimizes the total completion time.

6.2 Problem Formulation

Let us consider that there is a set of users participating in a distributed app. The app runs on two phases. In the first phase, faces are detected on photos from a publicly available data set. This means that there is no privacy related issue on running these tasks on the cloud. In the second phase, face recognition is done on the detected faces. These recognition jobs are dependent on the output of previous detection jobs. In the recognition step, each user checks whether faces of her friends can be found on the detected faces. Interestingly, some users can decide not to share their friends' faces to the cloud. In such a situation, the recognition step would have to be done in that user's mobile device. Hence, this step enforces device dependency on some jobs.

For formulating our problem, let us define a few variables. We can consider that there are m users and n jobs (i.e. annotated with "offloadable"). Each user has a device D and an avatar Av. The set of users is $U = \{U_1, U_2, ...U_m\}$. The set of machines is $M = \{D_1, D_2, ..., D_m, Av_1, Av_2, ...Av_m\}$. And, the set of jobs is $J = \{J_1, J_2, J_n\}$.

Jobs can have dependency on a particular machine/device. The device dependency matrix can be represented as $R = \{R_{ij} | i \epsilon M, j \epsilon J\}$ where $R_{ij} = 1$ indicates job j is dependent on machine i (e.g., machine i has private data and this data is the input of job j). It is worth noting that if job j does not have dependency on any

device, it can be executed on different mobiles and avatars, even though they do not belong to the user that initiated the application that contains j.

There is another type of dependency where a job depends on the output of another job. The job dependency is represented as a directed acyclic graph, $G < V, E >$ as shown in Figure 6.2. The set of vertices, $V = \{J_1, J_2, J_n\}$ and the set of edges, $E = \{e(j,k)|$ job J_k is dependent on job $J_j\}$.

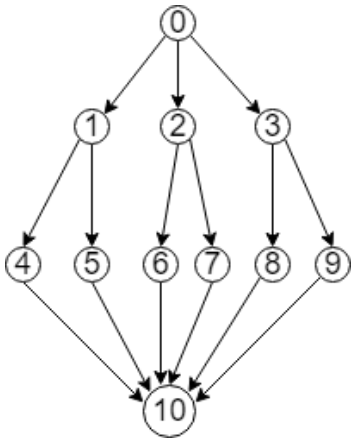


Figure 6.2 Job dependency represented in a directed acyclic graph.

The computation cost matrix is $COMP = \{COMP_{ij}|i\epsilon M, j\epsilon J\}$ where $COMP_{ij}$ indicates the cost (time) of computing job J_j on machine M_i. The communication cost matrix is $COMM = \{COMM_{xy}|x\epsilon M, y\epsilon M\}$. For example, machine M_0 and M_2 have 35 ms of communication cost in between them (as shown in the matrix): $\begin{pmatrix} 0 & 2 & 35 \\ 2 & 0 & 10 \\ 35 & 10 & 0 \end{pmatrix}$

The data Input/Output matrix is $IN = \{IN_{xy}|x\epsilon M, y\epsilon M\}$ where IN_{xy} indicates the size of input data needed by job J_y from J_x. This matrix contains estimated data which are gathered and improved based on historical usage patterns.

The values of these input variables are processed by CASINO from the profiling information of devices.

The goal of CASINO is to produce a schedule, s[][] = $\{s_{ij}|i\epsilon M, j\epsilon J\}$, where $s_{ij} = 1$ indicates job j is scheduled on machine i. CASINO also produces a completion

time matrix, T[][] = $\{T_{ij}|i\epsilon M, j\epsilon J\}$, Where T_{ij} = the time needed to execute job j is on machine i.

6.3 Job Scheduling

Lawler et al. [74] proposed the $\alpha\,|\beta|\,\gamma$ model to define the characteristics of jobs and the machine environment. To explain this model, we define P_{ij} as the processing time of job J_j on machine M_i. α defines the machine environment which is defined as:

$$\alpha = \begin{cases} o & \text{single machine: } p_{1j} = p_j \\ P & \text{identical parllel machine: } p_{ij} = p_j \text{ for all } M_i \\ Q & \text{uniform parallel machines: } p_{ij} = \frac{p_j}{s_i} \text{ for a given speed } s_i \text{ of } M_i \\ R & \text{unrelated parallel machines: } p_{ij} = \frac{p_j}{s_ij} \text{ for job dependent speeds } s_{ij} \text{ of } M_i \end{cases} \tag{6.1}$$

In our case, mobile and avatars can run jobs parallelly and they have different speeds which do not depend on the job types. This matches with the machine environment Q.

Next, β defines the job characteristics which can be any of $\{pmtn, prec, tree, r_j, p_j\}$. These values define job characteristics such as whether a job can be preemptive, has precedence requirement, etc. Our scenario follows the *prec* characteristics, because a job can be dependent on others (i.e., has precedence constraints).

The third parameter of the model is γ which defines the optimality criteria. The values of it can be tardiness, lateness, total completion time, etc. In our case, we want to minimize the total completion time $\sum_{j=1}^{n} C_j$ for all job j.

Hence, our scheduling problem can be defined as a $Q_m|prec|\sum_{j=1}^{n} C_j$ optimization problem. As shown in [74], this class of scheduling problems are NP-hard and do not have any polynomial time algorithm. However, we have devised an approximation algorithm based on a topological sort and Greedy technique

to produce a near-optimal scheduling. A topological sort is used to find a linear ordering of jobs respecting their precedence relation. After having the ordering, we apply a greedy technique based on the minimum sum of computation time and communication time for various device and job combinations.

6.3.1 Scheduling in Batches

The algorithm 2 shows the batch-mode operation of the job scheduler. In each batch, the scheduler receives a list of jobs from the job queue as shown in Line 3. The scheduler then resolves the dependencies among jobs (Line 4) and devices (Line 5). A code analysis tool [6] can be used for the job dependency resolution step. At this step, the job dependency graph $G < V, E >$ is generated. The machine dependency can be resolved by analyzing the *@local* and *@remote* annotations used by the programmer. The scheduler then uses the profile data (Line 6) which are periodically collected from user devices.

In the current design, the scheduler schedules one app at a time. Although,

Algorithm 2 Job Scheduling

1: $Queue \leftarrow$ list of jobs
2: **repeat**
3: $J \leftarrow$ jobs from Queue
4: $G \leftarrow resolveDependency(J)$
5: $R \leftarrow$ machine dependency data
6: $prof \leftarrow$ profile data
7: SCHEDULE(J, G, R)
8: **until** Queue is Empty

6.3.2 The Main Scheduling Algorithm

Algorithm 3 shows how CASINO makes scheduling decisions. Line 2 calls Algorithm 5 (topological sort) to get an ordered list of jobs. This ordering preserves the job dependency on each other. Next, the scheduler builds the computational cost matrix COMP (shown in Line 5) by estimating the completion time of each job on all devices. This means that COMP[d][j] is the estimated execution time of job j on device d. This is done for all job and device combinations. The computation cost estimation can be done using the instruction count of a job and a device's CPU capability. In Android, instruction counts of a function can be calculated by using the Android Debug API. It is worth noting that we can improve the estimation over time by learning from past executions of an app.

After that, the scheduler iterates through all jobs in the ordered list L. First, it checks whether a job has dependency on a device or not (Line 10). If yes, then it is scheduled on that device (s[d][j] entry is updated on the schedule matrix). The total cost of scheduling job j on this device d is calculated in Line 13. The previously estimated computation cost is the first part of the cost calculation. In the second part, the scheduler estimates the communication cost associated with this current schedule. A call to the $communicationCost(device, job)$ function is made where it calculates how much time it would take to transfer any output data from devices where job j's predecessor jobs are scheduled. This is due to the dependency on other jobs' output. Since this current schedule is finalized, the scheduler updates the total cost variable with the current job's cost (Line 14).

If a job does not have any device dependency, the scheduler moves ahead with trying to assign it to an available device and estimates the cost. This step is done in a greedy manner. The device giving the current minimum time for completion is selected for scheduling the job. This step is done using a min heap (shown in Line 18). As shown in Line 20, the device providing the minimum cost is obtained from the

Algorithm 3 The Main Job Scheduling Algorithm

```
1: procedure SCHEDULE(J, M, G, R)                          ▷ m users, n tasks
2:     L ← topologicalSort(G)
3:     for each job j in L do
4:         for each device d in M do
5:             estimate COMP[d][j]
6:         end for
7:     end for
8:     cost ← 0
9:     for each job j in L do
10:        if j.isDeviceDependent() then
11:            d ← j.deviceDependency()
12:            s[d][j] ← 1
13:            T[d][j] ← COMP[d][j] + communicationCost(d, j)
14:            cost ← cost + T[d][j]
15:            continue
16:        end if
17:        for each device d in M do
18:            minHeap ← (COMP[d][j] + communicationCost(d, j))
19:        end for
20:        T[d][j] ← minHeap.poll()
21:        s[d][j] ← 1                          ▷ device d is optimal to schedule j
22:        cost ← cost + T[d][j]
23:    end for
24:    print cost                                ▷ the total completion time
25:    return s                                  ▷ Return the schedule
26: end procedure
```

heap by polling. The schedule matrix and the cost matrix are updated with this selection. The total cost variable is also updated in Line 22. After iterating through all jobs in the ordered list L, the schedule matrix has the updated schedule which is returned in Line 25. The time complexity of the scheduling algorithm is O(mn^2logm) where n is the number of jobs and m is the number of devices.

Algorithm 4 Calculate Communication Cost for State Synchronization

```
1: procedure COMMUNICATIONCOST(d_1, j)
2:     cost ← 0
3:     L_pred ← predecessor(j)
4:     for each job k in L_pred do
5:         d_2 ← scheduled_device(j)
6:         cost ← cost + COMM[d_2][d_1] * IN[k][j]
7:     end for
8: end procedure
```

6.3.3 Communication Cost Estimation

Algorithm 4 shows how CASINO estimates the communication cost of scheduling job j on device d. This algorithm utilizes two matrices named COMM and IN (introduced in Section 6.2). An entry COMM[d_1][d_2] refers to the time it takes to send one unit of data from device d_1 to device d_2. An entry IN[k][j] refers to the amount of data needed by job j from its predecessor job k. The function *communicationCost*(d, j) first finds the list of predecessor jobs of job j (shown in Line 3). It iterates through each job k from the list and find where they are scheduled. As shown in Line 6, it then estimates what is the cost to transfer IN[k][j] amount of data from that device to the current device d. The function sums up all such costs from all the predecessor jobs and returns it.

Algorithm 5 Topological Sort Algorithm

```
1: procedure TOPOLOGICALSORT(G)
2:     add vertices with no incoming edge to queue Q
3:     while (!Q.empty) do
4:         u ← Q.poll()
5:         list.addFront(u)
6:         for each neighbor v of u do
7:             delete edge (u,v)
8:             if v has no incoming edge then
9:                 Q.offer(v)
10:            end if
11:        end for
12:    end whilereturn list
13: end procedure
```

6.3.4 Topological Sort

The algorithm 5 shows how CASINO generates an ordered list of jobs considering their dependency on each other. The *topologicalSort*() function takes a directed acyclic graph (DAG) $G < V, E >$ as input. V is a set of vertices. Each job is a vertex in this graph. E is a set of edges which contains edge e(j,k) such that job k is dependent on job j. This topological sorting is done following Kahn's algorithm [61] which runs in linear time. In this algorithm, a queue is used to traverse through the list of vertices. The queue is initialized with vertices without any incoming edges (Line 2). A vertex with 0 in-degree means that it does not depend on any other jobs. The algorithm first considers such vertex u and adds them to the front of the ordered list (Line 5). Then, it explores each of its neighbor v after deleting the edge (u,v) and checks whether v's in-degree is now 0. If yes, then it enqueues v (Line 9) and

carries on in this fashion. At some point, it reaches to a state when all the vertices (i.e., jobs) are added to the ordered list.

6.4 CASINO Design and Implementation

CASINO has four main components : CASINO API Library, Device Profiler, Execution Manager, and Job Scheduler. The architecture of CASINO and its components are shown in Figure 6.3.

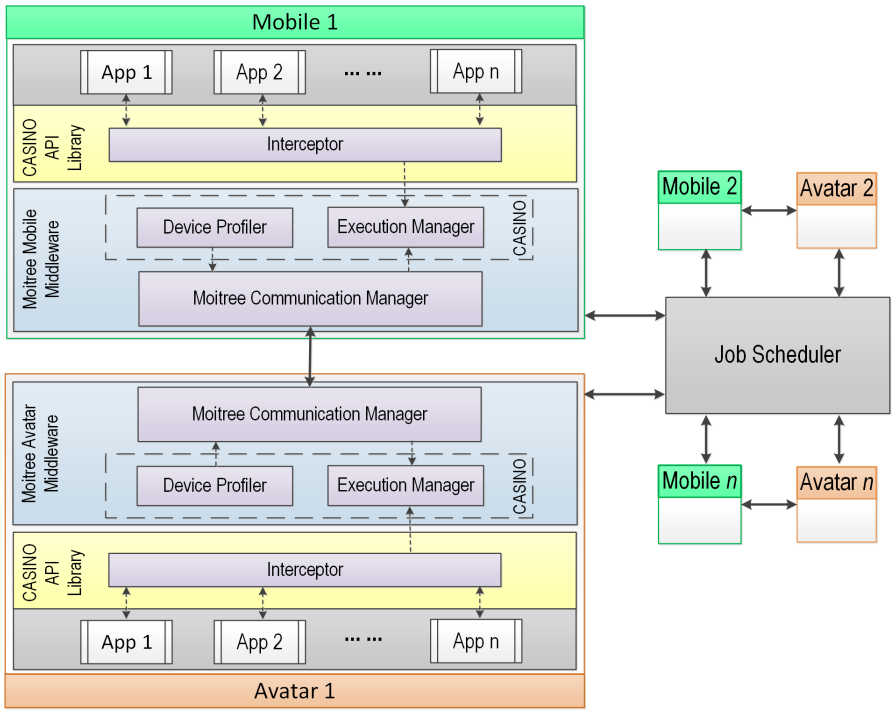


Figure 6.3 CASINO architecture.

6.4.1 CASINO API Library

CASINO has a simple API set for partitioning code statically and dynamically. Programmers can annotate code components using *@Mobile* or *@Avatar* to partition code statically. This will tell CASINO to execute the code component on mobile or avatar respectively. Dynamic partitioning can be achieved by annotating classes/functions with the *offloadable* tag. The API library has a sub component

named Code Interceptor which catches the annotated part of the code and forwards the list of *offloadable* code components to the job scheduler.

6.4.2 Device Profiler

CASINO runs a set of profilers which monitor network bandwidth and latency, battery level, CPU capability and usage, memory in mobile devices. It also monitors the CPU and memory load/usage in the cloud entity. Tools such as Battery Historian [5], Network Monitor [14], HPROF Analyzer [13] can be used for profiling battery, network, and memory. Profiling is done each time a distributed app is executed. Profilers sends profiled information to the job scheduler.

6.4.3 Job Scheduler

This component is designed as a cloud service (similar to the Group Management Service in Moitree). All user devices (both mobile and avatars) communicate with it to submit the profiling information periodically. When a user initiates the distributed app, the scheduler receives a list of *offloadbale* jobs in the job queue. It first resolves the dependency on devices and jobs by running a dependency analyzer [6] tool. By applying topological sort, the scheduler first produces an ordered sequence of jobs which can be executed one after another respecting the job dependency. After that, it uses the profiling information to estimate the time needed to execute each job in different devices. It applies a greedy algorithm (explained in Section 6.3.2) to select the job-device schedule which produces the minimum completion time considering both computation and communication cost (delay). After finishing the scheduling, the job scheduler forwards a list of jobs to each device for executing. It should be noted that the job scheduler works in a batch mode. This means that, after receiving a scheduling request, it freezes the job queue and schedules all the jobs currently

stored in the queue. Once the scheduling is done for the first batch, it can collect the profiling information again and start scheduling jobs for the next batch.

6.4.4 Execution Manager

The execution manager receives scheduling instruction from the job scheduler. This instruction contains a list of jobs to execute and a list of recipients of the output data (if other jobs are dependent on these jobs). The execution manager in a mobile device or avatar executes jobs from this list one after another and sends the state/data to the specified device if needed.

6.5 Evaluation

The goals of the evaluation are to: 1) validate the job scheduler, 2) test the performance of the computation offloading executor. The first part is done by validating the job scheduler with simulated but realistic data. The second part of the evaluation is done by using two real application named PhotoFilter and FaceDate.

6.5.1 Validation of the Job Scheduler

The job scheduler works based on profiling information collected from user devices and avatars. Ideally, this part should be evaluated with real profiling data collected from users' mobile devices and avatars. However, due to the difficulty of such data collection, we have validated the job scheduler with a simulated albeit realistic data set. While this validation is very simple, it illustrates the benefits of our algorithm. We have modeled the data set based on the app described in Section 6.2.

First, let us provide the list of parameters used in this evaluation: Number of jobs, n = 8; number of machines, m = 6. There are three users in the group who have total of three mobile devices and three avatars. In the following matrices, devices with index 0-2 and 3-5 are considered mobile devices and avatars, respectively.

The device dependency matrix is

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Each row represents a machine and a column represents a job. Entry (i,j) indicates whether job j depends on device i or not. A value 1 indicates j is dependent on i (i.e., j should be scheduled on machine i) and a value 0 means it is not. Entries in the first row of matrix D indicates that only Job J_0 and J_7 depends on machine M_0. J_0 and J_7 are modeled as the initialization and result merging jobs respectively. Both these jobs are done by an initiator of the distributed computation. That is why these two jobs are modeled as dependent on initiator user's mobile device M_0. Similarly, we can observe from matrix D that J_5 and J_6 are dependent on mobile devices M_1 and M_2 respectively.

The job dependency is represented as a graph G (as shown in Figure 6.4).

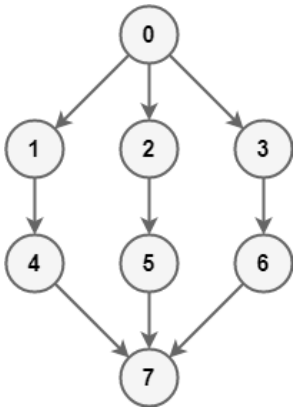


Figure 6.4 The job dependency graph, G.

We are using the adjacency matrix representation of the graph G. The adjacency matrix is

$$\text{adj} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The *adj* matrix is 8x8 in dimension (since there are 8 jobs in our dataset). Each entry (j,k) in this matrix represents whether job k is dependent on job j or not. For example, jobs J_1, J_2, J_3 are dependent on J_0.

The communication cost matrix is

$$\text{COMM} = \begin{bmatrix} 0.0 & 250.0 & 300.0 & 30.0 & 32.0 & 30.0 \\ 250.0 & 0.0 & 290.0 & 38.0 & 30.0 & 31.0 \\ 300.0 & 290.0 & 0.0 & 42.0 & 33.0 & 38.0 \\ 30.0 & 38.0 & 42.0 & 0.0 & 5.0 & 3.0 \\ 32.0 & 30.0 & 33.0 & 5.0 & 0.0 & 7.0 \\ 30.0 & 31.0 & 38.0 & 3.0 & 7.0 & 0.0 \end{bmatrix}$$

The dimension of the matrix $COMM$ is 6x6 (since there are 6 machines in our example dataset). An entry (x,y) indicates the cost (time in ms) of delivering one unit of data from machine x to machine y. Mobile to mobile communication is modeled as most costly (250 - 300 ms). Mobile to cloud is modeled to be within 30 - 42 ms. The cost of avatar to avatar communication is very low and modeled to be within 3 - 7 ms (considering they are physically located in the same cluster). These values are modeled based on NJIT's wireless network.

The speed of computation cost matrix is estimated as:

$$\text{COMP} = \begin{bmatrix} 50.0 & 400.0 & 400.0 & 250.0 & 250.0 & 250.0 & 250.0 & 100.0 \\ 400.0 & 400.0 & 400.0 & 250.0 & 250.0 & 250.0 & 250.0 & 250.0 \\ 400.0 & 400.0 & 400.0 & 250.0 & 250.0 & 250.0 & 250.0 & 250.0 \\ 70.0 & 70.0 & 70.0 & 40.0 & 40.0 & 40.0 & 40.0 & 150.0 \\ 70.0 & 70.0 & 70.0 & 40.0 & 40.0 & 40.0 & 40.0 & 150.0 \\ 70.0 & 70.0 & 70.0 & 40.0 & 40.0 & 40.0 & 40.0 & 150.0 \end{bmatrix}$$

The COMP matrix has a dimension of 6x8. Each row represents a machine and each column represents a job. An entry (i,j) refers to the estimated computation time (in ms) of job j if it is executed on machine i. For example, the entry (0,2) indicates that it will take 400 ms if job J_2 is executed on machine M_0, However, the same job will take 70 ms if it is executed on machine M_3 (it's the avatar of user U_0) as shown in the (3,2) entry. It can be noted that J_0 and J_7 are dependent on machine M_0. Hence, entries (0,0) and (0,7) are the only relevant costs for J_0 and J_7 respectively. All other entries for these two jobs can be considered to be irrelevant and they can well be replaced with infinity values in the matrix.

The topological sort step generates the ordered list : Job0, Job1, Job2, Job3, Job4, Job5, Job6, Job7. This list is generated based on the job dependency graph G shown in Figure 6.4. It is worth noting that there can be multiple valid ordering of the same set of jobs. Any valid ordering will work for our scheduling algorithm.

Based on the aforementioned input values, the job scheduler produced the output schedule shown below:

$$\text{s} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

The schedule indicates that Job J_0, J_7 are scheduled on machine M_0 (user U_0's mobile). J_5 and J_6 are scheduled on M_1 (user U_1's mobile) and M_2 (user U_2's mobile) respectively. J_1, J_2 are scheduled on machine M_3 (user U_0's avatar). J_3 and J_4 are scheduled on M_4 (user U_1's avatar) and M_5 (user U_2's avatar). The total completion time of the whole distributed computation is 1614 ms. Table 6.1 shows the comparison with two other scheduling scenarios: all jobs are scheduled on mobiles, and all jobs scheduled on avatars.

Table 6.1 Comparison of Total Completion Time (in ms)

Sheduled by CASINO	Everything Scheduled On Mobiles	Everything Scheduled On Avatars
1614	3416	1837

It can be noted that the scheduling algorithm generates the schedule in polynomial time by applying a greedy technique. It is well known that greedy based approximation algorithms can get settled in local minima. Due to this nature of the greedy algorithms, it is possible (for some cases) that scheduling everything on the cloud can produce smaller completion time comparing to the one produced by CASINO. Since we are solving an NP-hard problem, this is a reasonable compromise. For a dynamic offloading scheduler, achieving a near-optimal result in realistic time frame is much more desired than producing a perfectly optimal result achieved after running the scheduler for a long time. This evaluation also shows the benefits of dynamic partitioning of codes in a distributed computation.

6.5.2 Efficiency of the Computation Offloading Executor

Next, we have used two real apps named PhotoFilter to evaluate the performance of the single mobile-to-cloud computation offloading. This evaluation will help us to verify that the actual computation offloading executor works efficiently and without much overhead.

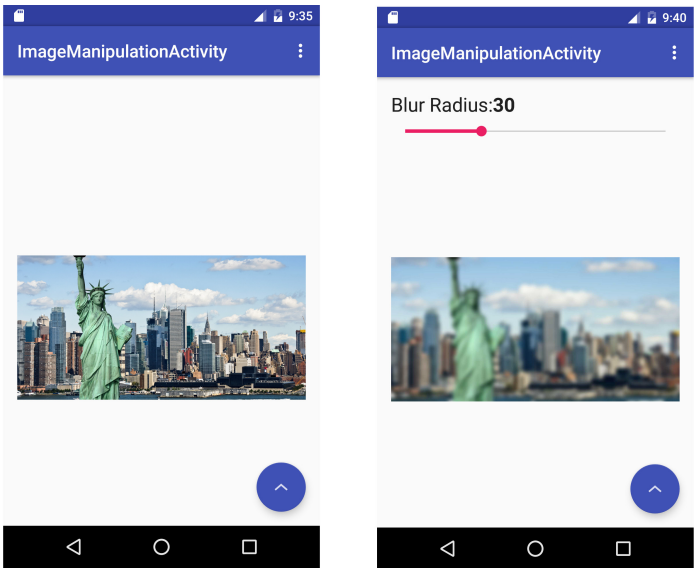


Figure 6.5 Using the blur filter in the PhotoFilter app.

The PhotoFilter app allows a user to select an image and apply different types of graphic filters such as blurring, gray-scale rendering or inverting the colors. This computation could be done locally on the mobile device or using the avatar's support by offloading the computation to it. The filtering jobs are characterized by heavy computation and the size of the target pictures is large enough to check the communication overhead. The blur filter is implemented by a Gaussian function with a user defined radius. CASINO provides support for both standard Android code (written in Java) and native C++ code. To demonstrate this, we implemented two different versions of the same Gaussian implementation, one in plain Java code and one in C++. The code used for the Java and the C++ implementation are relatively the same, considering the language syntax and characteristics. Figure 6.5 shows the result of the Gaussian blur filter applied to an image.

The goal of this evaluation is to verify the improvement in execution time introduced by the offloading. We want to verify that the contribution of computation offloading increases with the increase of the computation complexity. In addition, another parameter to check is the communication overhead introduced to move the

picture from the local device to the avatar machine and to move back the blurred image.

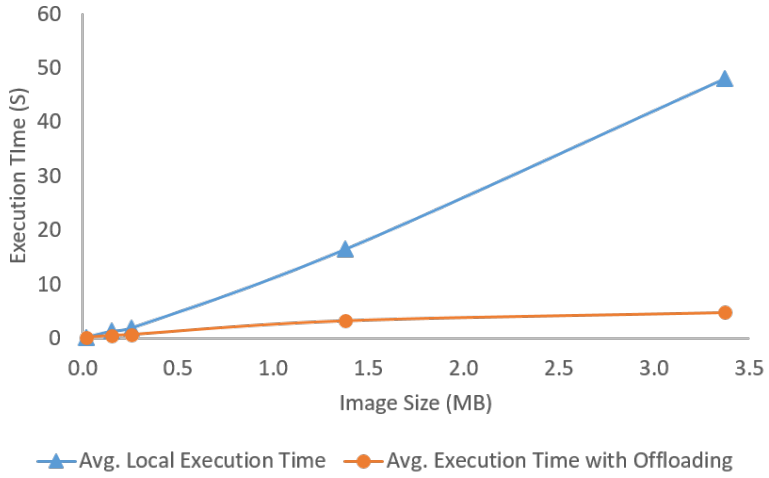


Figure 6.6 Results collected executing a Gaussian Blur filter with and without offloading support. In this case the code filter is implemented in Java.

Figure 6.6 shows the results collected during the experiments. The Gaussian filter computation varies based on the image size. If the picture is relatively small, the offloading contribution is almost zero. However, when the size of the picture increases and consequently the computation increases, the contribution of offloading becomes higher. The benefits of the offloading mechanism are particularly visible in the last measurement.

We have repeated the same experiment but, this time with the native C++ implementation of the Gaussian filter. Figure 6.7 shows the collected results for this native code offloading. It can be noted that some computation such as image manipulation, signal processing, etc have a huge number of array access and manipulation. This is inherently slower in Java comparing to C/C++ code [18]. For this reason, C++ implementation of the Gaussian blur filter runs faster than standard Java based implementation on both Android devices and VMs. This is reflected in Figure 6.7.

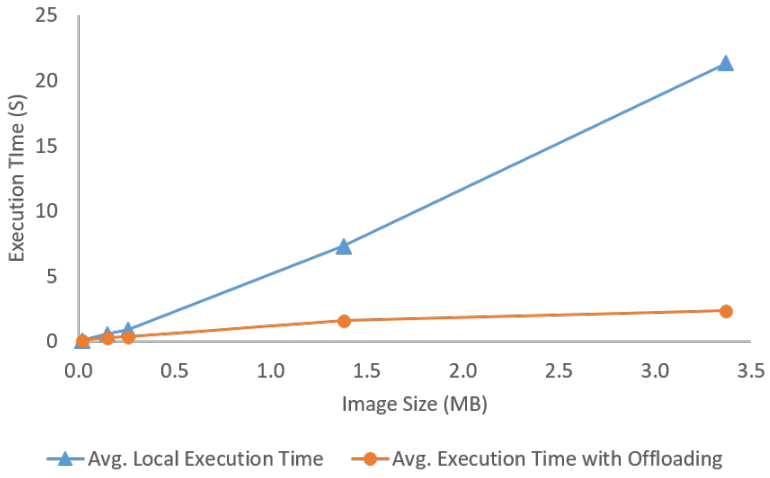


Figure 6.7 Results collected for a C++ implementation of the Gaussian Filter function.

From both the results shown in Figure 6.6 and Figure 6.7, we can observe that the computation offloading becomes significantly effective if the computation is heavy.

6.6 Chapter Summary

This chapter presented CASINO, a sensor driven computation offloading system, which can be used to minimize the total completion time of a distributed computation. Each distributed computation comprises of many jobs which can be executed on users' mobile devices and avatars. CASINO uses a job scheduler which schedules jobs on various devices such that the computation time and the communication time needed to execute the job becomes minimized. CASINO then executes the scheduled jobs on the target devices (mobile or avatar). As a result, CASINO minimizes the total completion time of the whole distributed app. We have evaluated both the job scheduler and the computation offloading executor. The evaluation result shows that both of these parts run efficiently.

CHAPTER 7

CONCLUSION

Ubiquity of sensor rich smart-devices such as smartphones, tablets, and smart wearable devices have inspired us to investigate the feasibility of developing high-level programming frameworks for context-aware computing. Our approach is based on two observations: the lack of programming frameworks for i) personal sensing on smart mobile devices, ii) collaborative computation over mobile devices. We have designed and developed two sensing frameworks which help programmers to utilize sensors from personal smart devices in a simple manner and hence, accelerate the development of context-aware apps. Additionally, we have also proposed and developed two frameworks for efficiently performing collaborative computation over sensor data collected by a group of users.

The performance evaluation of two sensing frameworks show that they significantly reduce the programming effort of programmers for writing context-aware apps for smart devices by utilizing various sensors. Using our API, programmers are able to combine various sensors in a real-time and efficient way. It was also verified that the frameworks manage low-level sensor management, and communication tasks very efficiently.

Our frameworks for collaborative computation empower programmers to explore context-awareness in a distributed manner. Programmers can utilize the collective power of collections of mobile devices in a simple manner using these frameworks. The results show that programmers have to give significantly less effort for developing distributed apps for mobile devices using our frameworks. The frameworks also hides all low-level details of the distributed system and manage the device-to-device communication, distributed execution very efficiently.

We believe that many useful context-aware apps and services can be very easily developed using our programming frameworks. Society can widely benefit by apps developed for health monitoring, disaster recovery, environment monitoring, emergency response, etc. By saving valuable programming effort from programmers, our frameworks will empower them to write even more useful apps for the society and the welfare of mankind.

BIBLIOGRAPHY

[1] ABI Research: Wearable Computing Devices, Like Apple's iWatch, Will Exceed 485 Million Annual Shipments by 2018. https://www.abiresearch.com/press/wearable-computing-devices-like-apples-iwatch-will/. [Online; accessed July-05-2017].

[2] American heart association: Target heart rates. http://www.heart.org/HEARTORG/HealthyLiving/PhysicalActivity/FitnessBasics/Target-Heart-Rates_UCM_434341_Article.jsp. [Online; accessed July-05-2017].

[3] Android - Sensor SDK. https://developer.android.com/guide/topics/sensors/sensors_overview.html. [Online; accessed July-05-2017].

[4] Argon browser. http://argon.gatech.edu/. [Online; accessed July-05-2017].

[5] Battery historian : Android battery profiling tool. https://github.com/google/battery-historian. [Online; accessed July-05-2017].

[6] Dependency analyzer. http://www.dependency-analyzer.org/. [Online; accessed July-05-2017].

[7] Google fit. https://www.google.com/fit/. [Online; accessed July-05-2017].

[8] Google Play Services - MessageAPI. https://developers.google.com/android/reference/com/google/android/gms/wearable/MessageApi. [Online; accessed July-05-2017].

[9] Intel SGX: Intel Software Guard Extensions. https://software.intel.com/en-us/sgx. [Online; accessed November-23-2015].

[10] Juniper research: Smart wearables market to generate $53bn hardware revenues by 2019. https://www.juniperresearch.com/press/press-releases/smart-wearables-market-to-generate-$53bn-hardware. [Online; accessed July-05-2017].

[11] Kickstarter Project - SensorDrone. https://www.kickstarter.com/projects/453951341/sensordrone-the-6th-sense-of-your-smartphoneand-be/rewards. [Online; accessed July-05-2017].

[12] Kryonet. https://github.com/EsotericSoftware/kryonet. [Online; accessed July-05-2017].

[13] Memory profiler : Android memory analyzing tool and hprof viewer. https://developer.android.com/studio/profile/am-hprof.html. [Online; accessed July-05-2017].

[14] Network monitor : Android network monitoring tool. https://developer.android.com/studio/profile/am-network.html. [Online; accessed July-05-2017].

[15] OpenCV. http://opencv.org/. [Online; accessed July-05-2017].

[16] Participatory sensing (smart santander). http://www.smartsantander.eu/index.php/blog/item/181-participatory-sensing-application. [Online; accessed July-05-2017].

[17] Participatory urbanism. http://www.urban-atmospheres.net/ParticipatoryUrbanism/index.html. [Online; accessed July-05-2017].

[18] Performance comparison of java and native c code in android. http://www.learnopengles.com/a-performance-comparison-between-java-and-c-on-the-nexus-5/. [Online; accessed July-05-2017].

[19] Power Tutor. http://ziyang.eecs.umich.edu/projects/powertutor/. [Online; accessed November-23-2015].

[20] Redis. http://redis.io/. [Online; accessed July-05-2017].

[21] SciTech Blogs - The Mobile Phone That Breathes. http://scitech.blogs.cnn.com/2010/04/22/the-mobile-phone-that-breathes/. [Online; accessed July-05-2017].

[22] Serialization performance evaluation. https://github.com/eishay/jvm-serializers/wiki. [Online; accessed November-23-2015].

[23] Smart measure. https://play.google.com/store/apps/details?id=kr.sira.measure&hl=en. [Online; accessed July-05-2017].

[24] The role of ICT during the disaster. http://wsms1.intgovforum.org/sites/default/files/EarthquakeICT0825.pdf. [Online; accessed July-05-2017].

[25] UNICEF: Wearables For Good. http://wearablesforgood.com/the-challenge/. [Online; accessed July-05-2017].

[26] Yuvraj Agarwal, Rajesh Gupta, Daisuke Komaki, and Thomas Weng. BuildingDepot: An Extensible and Distributed Architecture for Building Data Storage, Access and Sharing. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, pages 64–71, New York, NY, USA, 2012. ACM.

[27] Charu C. Aggarwal and Tarek Abdelzaher. *Social Sensing*, pages 237–297. Springer US, Boston, MA, 2013.

[28] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. Rio: A System Solution for Sharing I/O Between Mobile Systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 259–272, New York, NY, USA, 2014. ACM.

[29] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 267–283. USENIX Association, 2014.

[30] Joshua Bloch. *Effective Java*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[31] Cristian Borcea, Xiaoning Ding, Narain Gehani, Reza Curtmola, Mohammad A. Khan, and Hillol Debnath. Avatar: Mobile Distributed Computing in the Cloud. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*, pages 151–156, mar 2015.

[32] Niels Brouwers and Koen Langendoen. Pogo, a middleware for mobile phone sensing. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*, pages 21–40, December 2012.

[33] Jeff Burke, Deborah Estrin, Mark Hansen, Andrew Parker, Nithya Ramanthan, Sasank Reddy, and Mani B. Srivastava. Participatory sensing. In *World Sensor Web Workshop, ACM Sensys*. ACM, 2006.

[34] Juan Luis Carús Candás, Víctor Peláez, Gloria López, Miguel Ángel Fernández, Eduardo Álvarez, and Gabriel Díaz. An automatic data mining method to detect abnormal human behaviour using physical activity measurements. *Pervasive Mob. Comput.*, 15(C):228–241, December 2014.

[35] Si Chen, Muyuan Li, Kui Ren, Xinwen Fu, and Chunming Qiao. Rise of the indoor crowd: Reconstruction of building interior view via mobile crowdsourcing. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 59–71, New York, NY, USA, 2015. ACM.

[36] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems - SenSys '15*, pages 155–168, New York, New York, USA, 2015. ACM Press.

[37] Xu Chen, Lei Jiao, Wenzhong Li, and Xiaoming Fu. Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing. *IEEE/ACM Transactions on Networking*, 24(5):2795–2808, oct 2016.

[38] John W. Cheng and Hitoshi Mitomo. The underlying factors of the perceived usefulness of using smart wearable devices for disaster applications. *Telematics and Informatics*, 34(2):528 – 539, 2017.

[39] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[40] Sunny Consolvo, David W. McDonald, Tammy Toscos, Mike Y. Chen, Jon Froehlich, Beverly Harrison, Predrag Klasnja, Anthony LaMarca, Louis LeGrand, Ryan Libby, Ian Smith, and James A. Landay. Activity sensing in the wild: A field trial of ubifit garden. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1797–1806, New York, NY, USA, 2008. ACM.

[41] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10*, page 49, New York, New York, USA, 2010. ACM Press.

[42] Eduardo Cuervo, Aruna Balasubramanian, Dae ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys '10)*, pages 49–62, June 2010.

[43] Edward Curry. Message-Oriented Middleware. In *Middleware for Communications*, pages 1–28. 2005.

[44] Tathagata Das, Prashanth Mohan, Venkata N. Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. Prism: Platform for remote sensing using smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 63–76, New York, NY, USA, 2010. ACM.

[45] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. BOSS: Building Operating System Services. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 443–458, Berkeley, CA, USA, 2013. USENIX Association.

[46] Shane B. Eisenman, Emiliano Miluzzo, Nicholas D. Lane, Ronald A. Peterson, Gahng-Seop Ahn, and Andrew T. Campbell. Bikenet: A mobile sensing system for cyclist experience mapping. *ACM Trans. Sen. Netw.*, 6(1):6:1–6:39, January 2010.

[47] Giancarlo Fortino, Giuseppe Di Fatta, Mukaddim Pathan, and Athanasios V. Vasilakos. Cloud-assisted body area networks: state-of-the-art and future challenges. *Wireless Networks*, 20(7):1925–1938, 2014.

[48] Giancarlo Fortino, Daniele Parisi, Vincenzo Pirrone, and Giuseppe Di Fatta. Bodycloud: A saas approach for community body sensor networks. *Future Generation Computer Systems*, 35:62 – 79, 2014. Special Section: Integration of Cloud Computing and Body Sensor Networks; Guest Editors: Giancarlo Fortino and Mukaddim Pathan.

[49] Raghu K. Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: Current state and future challenges. *IEEE Communications Magazine*, 49(11):32–39, 2011.

[50] GeoNames. The geonames project. `http://www.geonames.org/`. [Online; accessed July-05-2017].

[51] Wikitude GmbH. Wikitude. `http://www.wikitude.com/app/`. [Online; accessed July-05-2017].

[52] Li Gong. JXTA: A network programming environment. *Internet Computing, IEEE*, 5(3):88–95, 2001.

[53] Google. Google goggles. `http://www.google.com/mobile/goggles`. [Online; accessed July-05-2017].

[54] Google. Google places api. `https://developers.google.com/places/`. [Online; accessed July-05-2017].

[55] Google. Volley. `http://developer.android.com/training/volley/index.html`. [Online; accessed November-23-2015].

[56] Sam Guinea and Panteha Saeedi. Mobile application development with MELON. In *Proceedings of 13th International Conference ADHOC-NOW 2014*, pages 265–278, June 2014.

[57] Songtao Guo, Bin Xiao, Yuanyuan Yang Yang, and Yuanyuan Yang Yang. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In *Proceedings - IEEE INFOCOM*, volume 2016-July, 2016.

[58] Ankur Gupta, Achir Kalra, Daniel Boston, and Cristian Borcea. Mobisoc: A middleware for mobile social computing applications. *Springer MONET*, 14(1):35–52, January 2009.

[59] Shivayogi Hiremath, Geng Yang, and Kunal Mankodiya. Wearable internet of things: Concept, architectural components and promises for person-centered healthcare. In *Wireless Mobile Communication and Healthcare (Mobihealth), 2014 EAI 4th International Conference on*, pages 304–307, Nov 2014.

[60] Dhiraj Joshi, Jiebo Luo, Jie Yu, Phoury Lei, and Andrew Gallagher. *Using Geotags to Derive Rich Tag-Clouds for Image Annotation*, page 239. 2011.

[61] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

[62] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 267–280, New York, NY, USA, 2008. ACM.

[63] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2010.

[64] Mohammad A. Khan, Hillol Debnath, Nafize R. Paiker, Narain Gehani, Xiaoning Ding, Reza Curtmola, and Cristian Borcea. Moitree: A middleware for cloud-assisted mobile distributed apps. In *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 21–30, March 2016.

[65] Kyu-Han Kim, Sung-Ju Lee, and Paul Congdon. On cloud-centric network architecture for multi-dimensional mobility. *ACM SIGCOMM Computer Communication Review*, 42(4):509, 2012.

[66] Harini Kolamunna, Yining Hu, Diego Perino, Kanchana Thilakarathna, Dwight Makaroff, Xinlong Guan, and Aruna Seneviratne. AFV: Enabling Application Function Virtualization and Scheduling in Wearable Networks. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, pages 981–991, New York, NY, USA, 2016. ACM.

[67] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the IEEE Infocom 2012*, pages 945–953, March 2012.

[68] Sokol Kosta, Vasile Claudiu Perta, Julinda Stefa, Pan Hui, and Alessandro Mei. Clone2Clone (C2C): Peer-to-Peer Networking of Smartphones on the Cloud. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2013. USENIX.

[69] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[70] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mob. Netw. Appl.*, 18(1):129–140, February 2013.

[71] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, April 2010.

[72] Nicholas D. Lane, Mu Lin, Mashfiqui Mohammod, Xiaochao Yang, Hong Lu, Giuseppe Cardone, Shahid Ali, Afsaneh Doryab, Ethan Berke, Andrew T. Campbell, and Tanzeem Choudhury. Bewell: Sensing sleep, physical activities and social interactions to promote wellbeing. *Mob. Netw. Appl.*, 19(3):345–359, June 2014.

[73] Oscar D. Lara and Miguel A. Labrador. A survey on human activity recognition using wearable sensors. *IEEE Communications Surveys Tutorials*, 15(3):1192–1209, Third 2013.

[74] Eugene L. Lawler, Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, and David B. Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in Operations Research and Management Science*, 4:445 – 522, 1993. Logistics of Production and Inventory.

[75] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, pages 521–532, New York, New York, USA, 2015. ACM Press.

[76] Amit A. Levy, James Hong, Laurynas Riliskis, Philip Levis, and Keith Winstein. Beetle: Flexible communication for bluetooth low energy. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 111–122, New York, NY, USA, 2016. ACM.

[77] Jia Li and James Z. Wang. Real-time computerized annotation of pictures. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(6):985–1002, 2008.

[78] Jiwei Li, Zhe Peng, Bin Xiao, and Yu Hua. Make smartphones last a day: Pre-processing based computer vision application offloading. In *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 462–470, June 2015.

[79] Jinsong Lin, Eusden Shing, Wing-Kai Chanand, and Rajive Bagrodia. TMACS: Type-based distributed middleware for mobile ad-hoc networks. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, July 2008.

[80] Stefanie Lindstaedt, Roland Mrzinger, Robert Sorschag, Viktoria Pammer, and Georg Thallinger. Automatic image annotation using visual content and folksonomies. *Multimedia Tools and Applications*, 42:97–113, 2009. 10.1007/s11042-008-0247-7.

[81] Jinshan Liu, Daniele Sacchetti, Franoise Sailhan, and Valrie Issarny. Group management for mobile ad hoc networks: design, implementation and experiment. In *Proceedings of the 6th international conference on Mobile data management (MDM '05)*, page 192199, May 2005.

[82] Hong Lu, Denise Frauendorfer, Mashfiqui Rabbi, Marianne Schmid Mast, Gokul T. Chittaranjan, Andrew T. Campbell, Daniel Gatica-Perez, and Tanzeem Choudhury. Stresssense: Detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference*

on Ubiquitous Computing, UbiComp '12, pages 351–360, New York, NY, USA, 2012. ACM.

[83] Jiebo Luo, Dhiraj Joshi, Jie Yu, and Andrew Gallagher. Geotagging in multimedia and computer vision–a survey. *Multimedia Tools Appl.*, 51(1):187–211, January 2011.

[84] Nicolas Maisonneuve, Matthias Stevens, Maria E. Niessen, and Luc Steels. *NoiseTube: Measuring and mapping noise pollution with mobile phones*, pages 215–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[85] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. In *Proceedings of of the second IEEE Annual Conference on Pervasive Computing and Communications (PERCOM 2004)*, pages 263–273, March 2004.

[86] Andrea Mannini, Angelo M. Sabatini, and Stephen S. Intille. Accelerometry-based recognition of the placement sites of a wearable sensor. *Pervasive and Mobile Computing*, 21:62 – 74, 2015.

[87] Justin Gregory Manweiler, Puneet Jain, and Romit Roy Choudhury. Satellites in our pockets: an object positioning system using smartphones. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 211–224, New York, NY, USA, 2012. ACM.

[88] Menno Merlijn. Gait analysis. `http://www.geradts.com/html/Documents/gait.htm`. [Online; accessed July-05-2017].

[89] Erfan Meskar, Terence D. Todd, Dongmei Zhao, and George Karakostas. Energy Aware Offloading for Competing Users on a Shared Communication Channel. *IEEE Transactions on Mobile Computing*, 16(1):87–96, 2017.

[90] Emiliano Miluzzo, Nicholas D. Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. Sensing meets mobile social networks: The design, implementation and evaluation of the cenceme application. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 337–350, New York, NY, USA, 2008. ACM.

[91] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, July 2006.

[92] Mor Naaman, Ron B. Yeh, Hector Garcia-Molina, and Andreas Paepcke. Leveraging context to resolve identity in photo albums. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '05, pages 178–187, New York, NY, USA, 2005. ACM.

[93] CBS News. Police caught murder suspect with the help of tourist photos. http://www.cbsnews.com/news/police-tourists-took-photos-of-san-francisco-pier-murder-suspect/. [Online; accessed November-23-2015].

[94] NOAA.gov. The magnetic declination calculator. http://www.ngdc.noaa.gov/geomagmodels/struts/calcDeclination. [Online; accessed July-05-2017].

[95] Ketan Patel, Mohamed Ismail, Sara Motahari, David J. Rosenbaum, Stephen T. Ricken, Sukeshini A. Grandhi, Richard P. Schuler, and Quentin Jones. Markit: Community play and computation to generate rich location descriptions through a mobile phone game. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10, 2010.

[96] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE Communications Surveys Tutorials*, 16(1):414–454, First 2014.

[97] Chuan Qin, Xuan Bao, Romit Roy Choudhury, and Srihari Nelakuditi. Tagsense: a smartphone-based approach to automatic image tagging. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 1–14, New York, NY, USA, 2011. ACM.

[98] Yang Qin, Dijiang Huang, and Xinwen Zhang. VehiCloud: Cloud Computing Facilitating Routing in Vehicular Networks. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1438–1445, June 2012.

[99] Qualcomm. Trepn power profiler. https://developer.qualcomm.com/software/trepn-power-profiler. [Online; accessed November-23-2015].

[100] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[101] Mashfiqui Rabbi, Min Hane Aung, Mi Zhang, and Tanzeem Choudhury. Mybehavior: Automatic personalized health feedback from user behaviors and preferences using smartphones. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 707–718, New York, NY, USA, 2015. ACM.

[102] Kiran K. Rachuri, Christos Efstratiou, Ilias Leontiadis, Cecilia Mascolo, and Peter J. Rentfrow. Smartphone sensing offloading for efficiently supporting social sensing applications. *Pervasive Mob. Comput.*, 10:3–21, February 2014.

[103] Rahul Raguram, Changchang Wu, Jan-Michael Frahm, and Svetlana Lazebnik. Modeling and recognition of landmark image collections using iconic scene graphs. *International Journal of Computer Vision*, 95(3):213–239, 2011.

[104] Lenin Ravindranath, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. Code in the air: Simplifying sensing and coordination tasks on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 4:1–4:6, New York, NY, USA, 2012. ACM.

[105] Sasank Reddy, Katie Shilton, Gleb Denisov, Christian Cenizal, Deborah Estrin, and Mani Srivastava. Biketastic: Sensing and mapping for better biking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1817–1820, New York, NY, USA, 2010. ACM.

[106] MIT Technology Review. Smart phones are eating the world. `http://www.technologyreview.com/photoessay/511791/smartphones-are-eating-the-world/`, 2013. [Online; accessed July-05-2017].

[107] Mahadev Satyanarayanan, Paramvir Bahl, Ramn Cceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.

[108] Jianchen Shan, Nafize R Paiker, Xiaoning Ding, Narain Gehani, Reza Curtmola, and Cristian Borcea. An overlay file system for cloud-assisted mobile applications. In *The 32nd International Conference on Massive Storage Systems and Technology (MASS 2016)*, 2016.

[109] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. Enhancing mobile apps to use sensor hubs without programmer effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 227–238, New York, NY, USA, 2015. ACM.

[110] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, and Ellen Zegura. COSMOS : Computation Offloading as a Service for Mobile Devices. *Proceedings of MobiHoc*, 2014.

[111] Square. Retrofit. `http://square.github.io/retrofit/`. [Online; accessed November-23-2015].

[112] Dean Takahashi. How google goggles works to deliver visual search results for mobile phones. `http://venturebeat.com/2010/08/23/how-google-goggles-works-to-deliver-visual-search-results-for-mobile-phones/`, 2010. [Online; accessed July-05-2017].

[113] Manoop Talasila, Reza Curtmola, and Cristian Borcea. Mobile crowd sensing. In *Chapter in the Handbook of Sensor Networking: Advanced Technologies and Applications*. CRC Press, 2014.

[114] Donna Tam. Facebook processes more than 500 TB of data daily. `http://news.cnet.com/8301-1023_3-57498531-93/facebook-processes-more-than-500-tb-of-data-daily/`. [Online; accessed July-05-2017].

[115] Thomas Weng, Anthony Nwokafor, and Yuvraj Agarwal. Buildingdepot 2.0: An integrated management system for building analysis and control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, BuildSys'13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[116] Yin Yan, Shaun Cosgrove, Ethan Blantont, Steven Y. Ko, and Lukasz Ziarek. Real-time sensing on android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 67:67–67:75, New York, NY, USA, 2014. ACM.

[117] Madoka Yuriyama and Takayuki Kushida. Sensor-cloud infrastructure - physical sensor management with virtualized sensors on cloud computing. In *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pages 1–8, Sept 2010.

[118] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, Broomfield, CO, October 2014. USENIX Association.

[119] Weiwen Zhang, Yonggang Wen, and Hsiao-Hwa Chen. Toward transcoding as a service: energy-efficient offloading policy for green mobile cloud. *IEEE Network*, 28(6):67–73, nov 2014.