Winter 12-15-2014

Novel Mobile Computation Offloading Framework for Android Devices

Meng Wang
Washington University in St Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Electrical and System Engineering

Thesis Examination Committee

Paul Min, Chair

Hiro Mukai

Heinz Schaettler

Novel Mobile Computation Offloading Framework for Android Devices

by

Meng Wang

A thesis presented to the School of Engineering

of Washington University in St. Louis in partial fulfillment of the

requirements for the degree of

Master of Science

December 2014

Saint Louis, Missouri

Contents

List of Figures

Acknowledgments

Special thanks to Dr. Min and Dr. Hung.

Meng Wang

Washington University in Saint Louis

December 2014

Dedicated to my parents.

ABSTRACT OF THE THESIS

Novel Mobile Computation Offloading Framework for Android Devices

by

Meng Wang

Master of Science in Electrical Engineering

Washington University in St. Louis, December 2014

Researcher Advisor: Professor Paul Min

The thesis implements an offloading framework for Google™ Android™ based on mobile devices. Today, the full potential for smartphones may be constrained by certain technical limits such as battery endurance and computational performance. Modern mobile applications own more powerful functions but need larger computation and faster frame rate, which consume more battery energy. Using the proposed offloading framework, mobile devices can offload computational intensive workload to servers to save battery energy consumption and reduce the execution time. The framework can also enable software developers to easily build and deploy services on the servers to support mobile devices to run computationally intensive jobs. Compared with other offloading schemes for android cell phones, the scheme enables developers to choose which parts of the codes are potentially offloading. As developers fully understand the data flow models of the apps, they are considered most capable of making offloading decisions. Developers can minimize communication overhead brought by offloading by carefully partitioning source code by data dependency. Experiment results and data showed that the proposed offloading scheme could significantly reduce computational time and battery energy consumption.

Chapter 1

Motivation and System Overview

The main motivation of the research is implementing a novel offloading mobile framework that improves the battery life and computational performance of Android smartphones. Smartphones have brought much convenience to people's life. People can use smartphones to stay in touch with people they care about, schedule their daily events, send emails, and read news. Offloading heavy computational workload can improve battery life and computational performance, which could enhance user experience significantly.

1.1 Background Introduction

Smartphones have played a more and more significant role in people's daily life due to the convenience and entertainment they bring. People can talk with their family members, trade stocks and play games wherever they are.

1.1.1 History of Mobile Phones

In 1947 AT&T introduced Mobile Telephone Service to many towns in America [22]. In 1973, Motorola created the first handheld mobile. Martin Cooper, who was a Motorola researcher at the time, made the world's first mobile phone call [22]. The prototype had to charge for 10 hours

and only had a 30 minutes talk time [22]. The 2G mobile phone system was widely deployed in 1990s. With increasing demand on data communication, 2G technology was replaced by 3G technology in mobile phone from 2001. Now, equipment manufacturers and carriers pay more attention on offering 4G and LTE technologies to accommodate bandwidth-intensive applications [22].

Today, emerging smartphone and tablet PC technologies have redefined personal computing in our everyday lives. Some mobile devices, such as smart phones, iPad and Kindle, have replaced PC as the most widely used computing devices in terms of Internet usage statistics. These mobile devices have become so popular due to the better ergonomic interface and portability over conventional laptops. We can expect that more and more applications, which were originally developed for traditional computers, will be ported to smartphones and tablet PCs. Furthermore, more and more novel mobile applications are developed and become popular in users' daily life. They bring not only convenience but also entertainment to users. For example, an app, Google maps, is based on many map-based services, such as Google Maps website, Google Ride Finder, Google Transit [1]. The app can provide users a planning route for traveling. It is considered as one of the world's most popular mobile apps [1]. Flappy birds, a smartphone game, that a player can control a bird to fly between pipes without collision [2], was released in May 2013 and suddenly become a global phenomenon in early 2014.

1.1.2 Limitations of Smartphones

However, the full potential for smartphones and tablet PCs may be constrained by certain technical limits such as battery endurance, computational performance, and portability. Modern

mobile applications own more powerful functions but need larger computation and faster frame rate, which consume more battery energy. Over the years, battery's energy density has not improved as significantly as semiconductor technologies. Unlike conventional cell phones, for which a single charge may last for several days, today's smartphones and other mobile devices barely sustain normal usage for a day without being charged. Hence, there are two ways to prolong battery endurance: one that increases the energy capacity, i.e., increasing the battery size, and one that reduces power consumption rate. Increasing the battery size leads to the increase in manufacturing cost as well. Also the device size is virtually fixed for ergonomic consideration, we are very likely to trade battery endurance for computational performance, or vice versa, when designing a smartphone or a tablet PC. That is, we either make our device superior in performance but running out of battery sooner, or make them withstand longer at potentially lower performance. With the device size assumed to be fixed, we may tradeoff battery endurance for computational performance as part of designing a smartphone or a tablet PC.

1.2 Potential Solutions

There are many ways to decrease battery consumption, such as turning off unused apps and components. However, background services, which the users may not even be aware, routinely take up the CPU and/or the communication module in mobile devices. These services affect the power consumption significantly even when mobile devices are not in use.

Considering the computational performance, we expect that more advanced system on chips, which provide higher performance per watt, will be made available for mobile devices. However,

there is still a long way to go before any mobile system on chips can be on part with its contemporary x86 CPUs in performance while keeping the power consumption low enough to handheld applications. Therefore, PC software titles, even the ones that are generally considered light-weight and widely adopted on contemporary PCs, may become “processor hogs” when ported onto mobile platforms. This is a significant concern since PC software developers (except those who work on 3D games and scientific computing) have been dealing for many years with regard to overpowered PC processors. At the same time, underpowered mobile system on chips may eventually suppress the creativity and possibility of mobile applications. We are thus seeking ways to increase computational capability on mobile devices without sacrificing battery life.

1.2.1 Mobile Operating System

There are many mobile operating systems in the market, such as Android, iOS, Blackberry and Windows Phone. According to the statistical data, Android tops 81.3% of smartphone market share, iOS owns 13.4% of smartphone market share, Windows Phone grows to 4.1% of smartphone market share and Blackberry has only 1.0% of the market share [3]. In this thesis, we mainly focus on Android operating system and just provide some comparisons between Android and iOS.

Android operating system is based on Linux kernel. It is based on direct manipulation, which uses touch inputs, such as swiping, tapping and pinching to manipulate on-screen objects [6]. It is applied on smartphones, tablets, smart TVs and cameras. iOS operating system is developed by Apple. It is derived from Mac OS X, which is based on a series of Unix-based graphical

interface operating systems [4]. iOS is also based on direct manipulation, which use multiple touch gestures, such as swiping, tapping and pinching to manipulate on-screen objects [4].

Android smart phone often gives an impression of shorter battery endurance than Apple's iPhone family with larger battery packs and tighter third-party software control. Therefore, reducing the power consumption rate through background services is a critical issue for Android based mobile computing devices. One of its popular reasons is that Google publishes most of the source code for Android, including network and telephony stacks, under free and open-source software licenses [6]. It allows users to change their system according to their preference.

At user interface part, iOS does not allow users to change its user interface, which frustrates for those who want to personalize their smartphones [5]. In contrast, Android devices are more open to users to build their personalized user interface. In terms of stability, iOS is more stable than Android, because iOS only gives users or apps low priorities to control the mobile operating system and does not allow users to change the operating system in order to ensure the integrity. However, Android gives developers and enthusiasts Android Open Source Project source to develop their personalized versions of operating system [6].

However, Android offers an open platform and users can customize their operating system, it is possible to expose their private information. Additionally, manufacturers can choose different hardware, models and screen size, which can cause a compatible problem. Also, some updated apps won't have an excellent performance in every Android phones.

1.2.2 Mobile Framework

Today, the Internet connectivity is essential for mobile computing devices. People can use iPhone or Kindle to search information on the Internet, communicate with friends and share photos with family members. Advanced wireless communication allows mobile users to access the Internet as a natural part of computing device. Therefore, we can regard the Internet as a virtual bus and offload some performance demanding workload tasks to the cloud servers. The cloud servers at data center are thus performance accelerators in the local analogy. Since the servers located at data centers are highly upgradable and scalable, and more importantly, the capabilities of the servers are not limited by the battery energy. Thus, offloading the workload from the mobile devices to the servers can significantly reduce power consumption rate in the mobile devices and improve performance significantly.

However, most third-party developers have little incentive to build their software products under cloud computing paradigm and hosting the services associated with them. It is obvious that most third-party software developers are not concerned about the power consumption caused by their products. Since consumers generally do not associate the battery life to a particular software product, attracting consumers with functionalities and features is always software developers' top priority no matter how much energy the software consumes. Moreover, setting up dedicated servers to host the offloaded services is a high-profile investment and brings low competitive advantage in return. Device vendor (e.g., Samsung), on the other hand, may realize tremendous benefits by providing a reliable mobile computation offloading framework which improves the performance (e.g., batter endurance, timeliness, features, reliability) of their products significantly.

1.2.3 Offloading

Offloading has become a promising technique to solve this problem by allowing smartphones to offload computationally intensive workload to servers. Although computational offloading, which emerged around 1970s, is not a brand new concept, its potential has never been so extensively explored until advanced wireless communication and high-speed Internet can sufficiently support it without significantly degrading the user experience [7]. Cloud computing, which was a different approach to explore the potential of ubiquitous Internet connectivity, facilitates and inspires innovations on computational offloading scheme [8][9]. Various efforts have been made to offload Java applications to the servers to take advantage of the cross-platform capability of Java bytecode [10]-[12]. Following the huge success of all-touch smartphone first introduced by Apple iPhone, Google came up with Android, which is an open-source and royalty-free mobile operating system, to compete against it. Google's strategy eventually pays off as Android has obtained the highest market share among all mobile operation systems as of the first quarter of 2013 according to IDC's latest statistic [15]. In addition to the commercial success, Android's open-source nature also invites researchers to innovate on mobile computing technologies, including offloading schemes [13][14], without building a mobile computing platform from scratch. Like many other researchers, we decide to implement our offload framework on Android due to its popularity and openness.

We can image that some tasks must keep working even when the Internet connection is unavailable, such as an alarm clock or a calendar reminder. These tasks may be offloaded but we should also implement a mechanism on the framework to launch backup tasks locally in case the

Internet connection is lost. Also not every computational workload on a mobile device can be offloaded. For instance, if a background service is location sensitive, offloading it might be unwise since the device has to continuously send updated coordinates to the server.

Decisions on what to offload are challenging. For example, data synchronization overhead, which depends on data volume and transmission bandwidth, the execution context of workload, and the differences in execution capability between the mobile SoC and the server's processor, can affect the advantage of offloading. Profiling the workload and monitoring the network quality-of-service (QoS) are essential in this framework. Furthermore, since the computation and communication characteristics are dynamic rather than static, a daemon, which periodically measures computational capability, communication performance, power consumption and dynamically makes offloading decisions for each offloadable task, is required in the framework.

In this thesis, we propose a simple and novel mobile offloading framework for Android devices. According to our experiment result based on two sample apps, the proposed framework enables app developers to easily take advantage of computational offloading to reduce significant amount of energy consumption and execution time.

Chapter 2

Proposed Mobile Computation Offloading Framework

2.1 Background Introduction

Before we present the proposed offloading framework, we would like to briefly review how the internal structure of a conventional app looks like. As we can see, the conventional app comprises multiple functional blocks communicating with each other. Some of the functional blocks perform numerical calculation or data process, while some of them are responsible for interacting with users. The app can also employ the communication abstract provided by the operating system to access a third party server over the Internet. Note we deliberately avoid using object-oriented programming terms in order to provide a more generalized view of software structure.

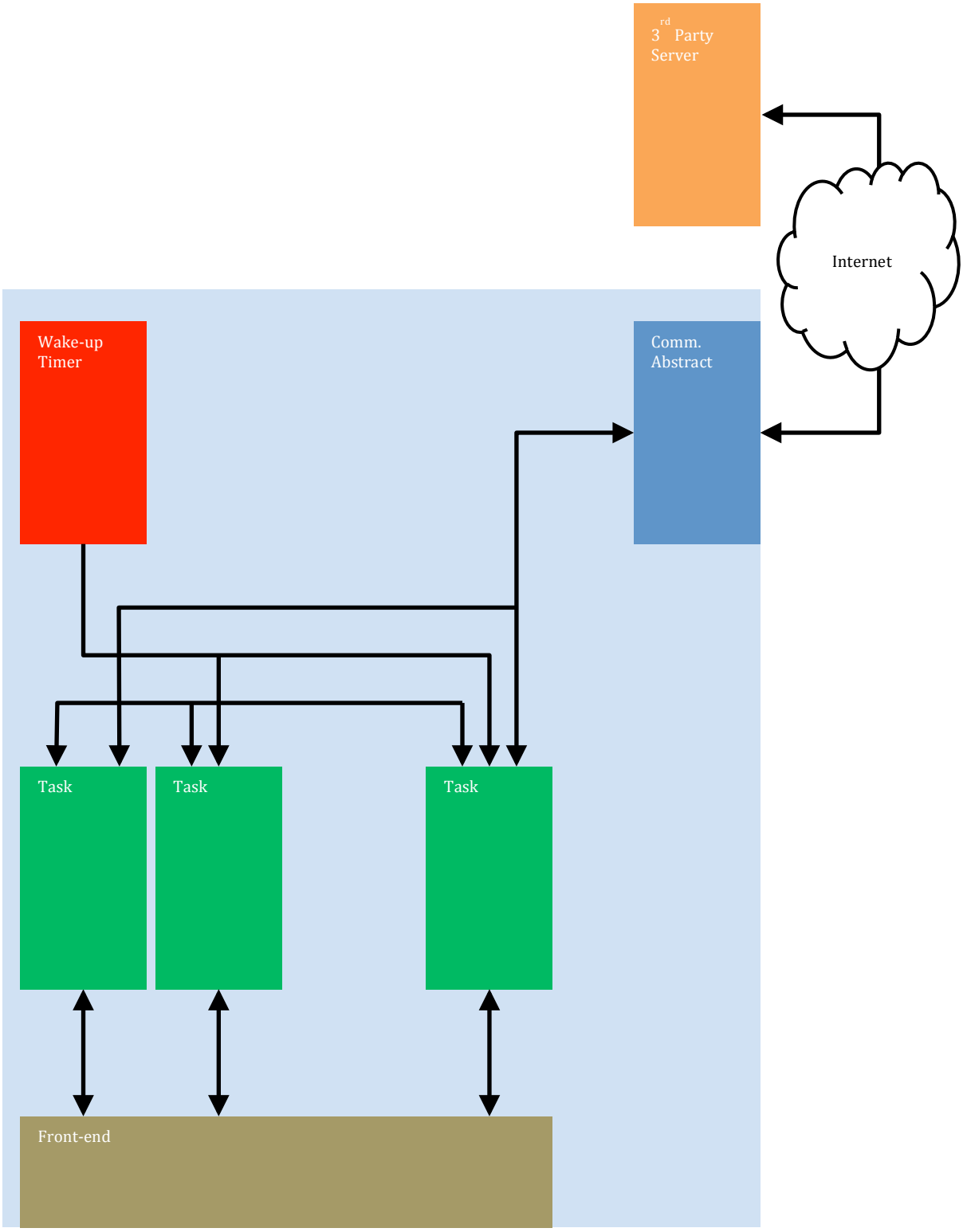

Figure 2.1: Conventional Software Resource Configuration

A conventional way for configuring software resource on a smart phone or a tablet PC is shown in Figure **2**.**1**. There are a wake-up timer abstract, some tasks which are periodically woken up by the timer abstract, some tasks which are not driven by the timer abstract, multiple front-end abstracts which are associated to the tasks, a communication abstract which handles the Internet connection, and one or more third-party servers which provide proprietary information over the Internet. The tasks here are defined by functions and can be in any granularity. However, we prefer to take functions as the basic elements of computation to better isolate them by their characteristics. In this configuration, the device side is fully responsible for providing the CPU time and energy required by the processes.

2.2 Related Work of Offloading Framework

In [13], the authors emulated an Android cell phone on the server to execute offload code. The server also emulates all embedded sensors, such as GPS, in order to recreate the complete runtime environment. By cloning the lower level abstraction, app developers do not need to change any code. However, if an app heavily depends on a built-in sensor, the smartphone has to constantly report the sensor reading to the server over the Internet, which spends significant amount of communication bandwidth and battery energy. For apps with relatively low computational complexity, the additional communication cost can easily defeat the purpose of offloading. For example, when a user is using a location-based information app, the mobile device has to send its coordinates to the server constantly and drains up the battery quickly. In [14], the authors are aware of the potential high communication cost of cloning a complete smartphone runtime environment remotely and thus limit the offloading target to pure functions,

which require only small and simple data structures. However, pure functions are not necessarily computationally intensive and thus offloading those functions might not save battery energy either.

2.3 Proposed Offloading Framework

To address these shortcomings, we propose a new mobile offloading framework. The proposed framework gives each app developer, who is most familiar with the app's computational and data flow model, explicit control of assigning potentially offloadable sections of the source code. Although the proposed framework requires app developers to add some lines to pack offloadable sections and call communication subroutines to interact with the server asynchronously, the additional effort is minimized.

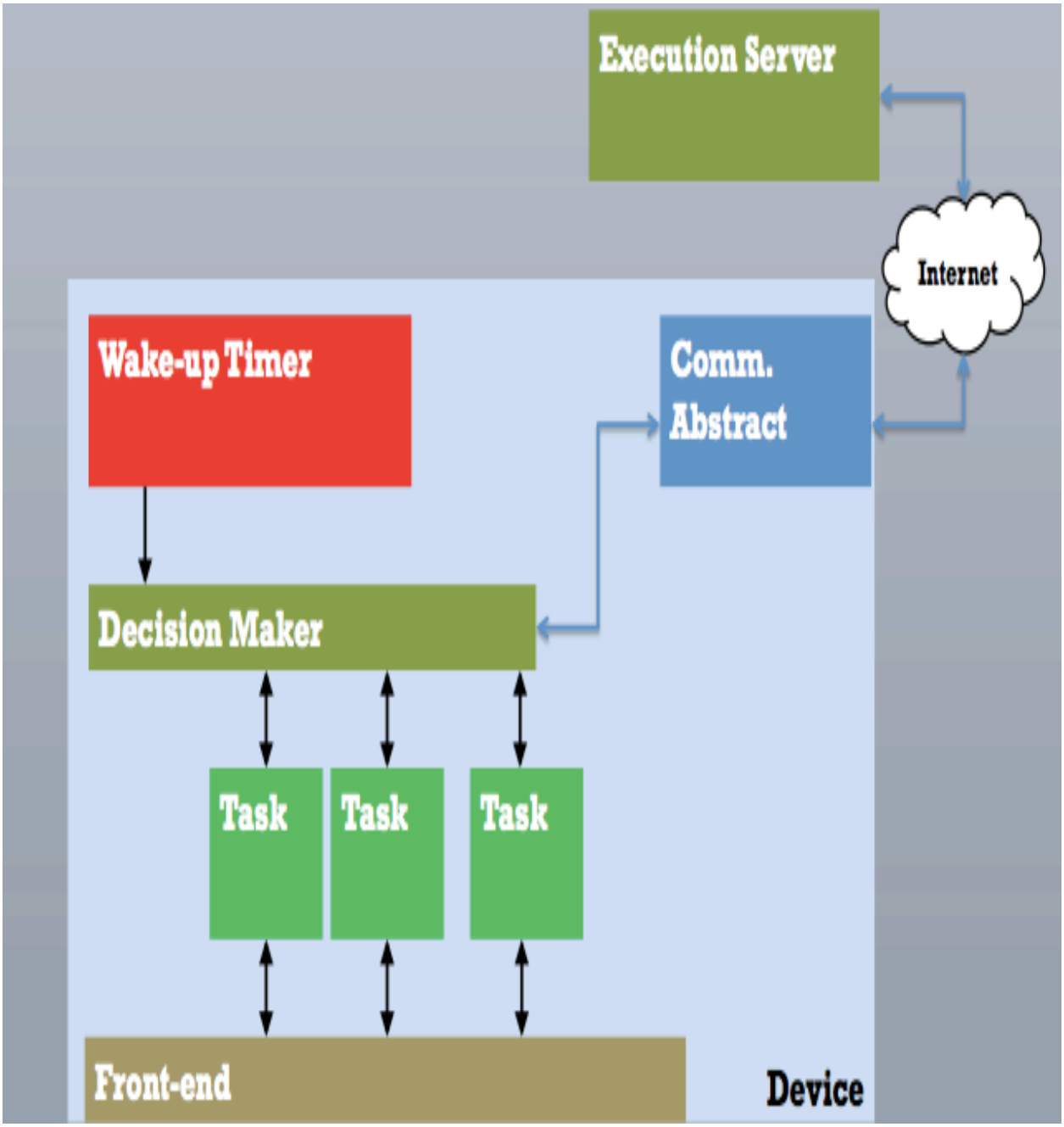

Figure 2.2: Proposed Mobile Framework

2.3.1 Decision Maker

Figure **2**.**2** shows the proposed framework for Android platform. On the device side, in addition to what we have in the conventional configuration, we place a decision maker as an inter-task communication gateway between the offloadable functional blocks and conventional ones to decide whether a called offlodable function should be performed by its local Dalvik bytecode or its remote Java applet. In the future, each offloading decision will be made based on the information provided by various components, including a device status register file, a profiler, a service status register file, and the manifest XML file explicitly defines the offloading parameters, such as service level agreement (SLA), for each offloadable block. A server side service status monitor, which keeps track of real-time workload of the server will be also proposed to assist the decision maker.

The decision maker is a simple subroutine which is called at the beginning of each offloadable block. An external data structure is accessed for historic performance record by the decision maker to estimate and compare the potential cost of offloading before deciding whether the offloading is beneficial. The effectiveness of an offloading decision maker relies on the accuracy on cost estimation. If we want to optimize the responsiveness of the proposed framework, the decision maker should estimate the end-to-end execution time for both cases and pick the shorter one. The end-to-end execution time is defined as the time lapse from when a task is called to when its result returned to the caller.

In the proposed mobile offloading framework, app developers are responsible for determining which sections of the source code could be offloaded. Obviously, the decision should be made

carefully as not every functional block can be offloaded to the server. For instance, offloading a location service daemon, which constantly accesses the GPS abstraction, might not be a wise decision, since it requires the device to continuously post its coordinates to the server over the Internet. It is expected that only computationally intensive blocks with low data dependency on the rest of the app could be offloaded. As app developers are most familiar with the computational and data flow models used in their app, we believe it is a better solution than online or offline machine analysis. Additionally, if the network condition is too bad, such as slow upload speed, the maker should not call the offloaded function.

2.3.2 Execution Server

An execution server is the key component which schedules and dispatches offloaded tasks. It is implemented on the server side. It is a highly customized HTTP server written in Java. It talks with device side HttpPipe object via HTTP connections. The server is responsible of checking the availability of an offloadable applet, receiving and storing an offloadable applet, launching the offloadable applet, retrieving the output, and sending it back to the device. The action performed by the server is selected by interpreting each incoming HTTP message. Besides the generic HTTP header fields, the client-server primitive is based on XML-style messages carried in HTTP request and response bodies. The instruction indicates the properties of a request is a series of flags and key-value pairs between <inst> and </inst>, while the optional data field is inserted between <data> and </data>. Flags are written as <flag/>, while key-value pairs are parsed as <key> value </key>. The XML-style message scheme is easier to be read and interpreted automatically, and highly extensible.

Each uploaded offloadable applet file is stored in <package_name>/<applet _id>/ folder of the server's file system, where the applet ID is the absolute value of the file's hash code. Each offloadable applet file is renamed as "offload.*" to simplify the subsequent launch operation. The isolation among different apps is achieved by storing the offloadable applet files in different folders in the file system. It is still possible, though very unlikely, that different offloadable applet files have the same package name and generate identical hash code. In that case, the unlucky developer should put some garbage code in one of the offloadable applet to solve the hash collision. We do not implement any collision resolution mechanism to handle this extremely rare case.

Launching an offloadable applet is done by creating a runtime process with shell command. The output string is subsequently retrieved from the process with getInputStream method and sent back to the client side. By performing the launch of each offloadable applet with system shell command, process lifecycle is managed by the platform. Therefore, it is much easier to integrate the server with off-the-shelf high availability server tools such as Linux Virtual Server than container approaches.

Two additional classes are created to help the main server class. The first one is ShellThread, which is responsible of performing shell command in a separated thread. Since the AVD should be running in the background and its boot-up time is incredibly long, we must launch it with the help of a ShellThread object. ShellThread objects are also used in launch applets in JVM or ADB shell. However, we call the launch method directly without creating new thread because 1) each offload request is already handled in a separate server thread, 2) the server thread should be

blocked anyway until the result is ready to be picked up from System.out.

The other helper class is ConstDef, which is static class to aggregate all the constants, mainly the location of Android SDK, used by the server. It is the only class an operator would need to edit when configuring a computer as an offload server.

2.4 Implementation

As the programming syntax and API support are almost identical, except some platform specific calls, we expect an app developer can reuse the entire offloadable code block to compile the Java applet. The only required change is inserting an input stream deserializer and an output object serializer at the input and output ends, respectively, thus each offloadable functional block only uses streams to communicate with outside. It is mandatory in order to unify the communication interface as API, which will be further explained in the next section.

In current implementation, we assume each offloadable applet is available on the server and ready to be launched by the execution server by its unique identifier. We propose a static applet upload system open to all app developers to submit up-to-date offloadable applets. Consequently, mobile devices do not have to deliver them to the server during runtime.

Once the execution of an offloadable applet is completed, the server returns the output stream to the mobile device and kills the process along with the underlying JVM to realease the computational resource. The mobile device can continue performing subsequent tasks since then.

Chapter 3

Experiments Setup and Results

In this proposed mobile offloading framework, we expect the framework can realize two goals. The first one is shortening the time of processing data and the second one is saving much more energy of mobile devices. Hence, we create an Android app, which is compute-intensive, to test this framework. Two parameters, calculation time and percentage of battery energy drop, are significant in this test. We run the experiment 100 times on different frameworks in order to observe their difference based on the two parameters mentioned above.

3.1 The framework of Sample App and Offloading System

Today, Android apps have become more and more powerful. They are small, easy to install, and easy to use. There are over one million Android apps in Google play, Google's official digital distribution platform. Students can communicate with their friends, post their daily photos and share their important moments through social media apps. Children can play mobile games online and watch cartoons or funny videos in YouTube. Photographers can use apps to manage and edit their raw photos. Mobile apps have brought a whole new world beyond human's imagination and got way more sophisticated than simple personal information management tools.

Many mobile app developers are seeking new ways to unleash the capability of mobile devices. Although recently mobile processors have performance pretty close to contemporary desktop ones, battery drainage concerns is still holding the potential of mobile computing devices back. The proposed mobile framework is positioned in mitigate this problem and grant more freedom for mobile apps.

3.1.1 Background Noise Spectrum Android App

In order to evaluate the proposed mobile offloading framework, we create an Android app, which performs background noise spectrum analysis. First we assume that most background noise is location dependent, time-invariant for small window, but time-variant for long term. That is, background noise in each location at certain time has its unique characteristics based on our assumption. If we can sample the background noise and analyze it at each location and time frame, we would be able to establish a time variant map of background noise signature. With this map, we could

1. Perform a noise cancellation service within a carrier's infrastructure,
2. Take advantage of the geographic information retrieved from the background noise,
3. Identify the probable source location from an unknown audio data by comparing the characteristic of its background noise, which could be helpful in forensic investigation.

In the sample app, the mobile device records background noise for a short period of time as a WAV file from the microphone. The WAV audio format is an uncompressed audio format. It can ensure a high quality of the sound file and be an ideal format for drawing background noise spectrum. The app then sample and collect data from the wave file and call Fast-Fourier

transform (FFT) function to transform the audio sample to frequency domain and displays the spectrum on the mobile device in the end. Obviously, the most computationally intensive block of the app is FFT, hence we can observe the difference of calculation time and consumed percentage of battery energy between the original mobile framework and the new proposed mobile framework.

3.1.2 The Flow Charts of Original and Proposed Apps

The flow chart of the app is shown in Figure **3**.**1**. In this figure, background noise is recorded and sampled based on FFT rules. The sample data is stored in an array. The FFT function processes data and plot the data on the phone screen. As mentioned above, the block of FFT needs large computation. Consequently, we fork the app and create another version, which utilize the proposed framework by offloading the FFT block. The remaining tasks, such as background noise recording, sampling the data and spectrum plotting, still take place on the mobile device. The block diagram of the app with offloaded FFT block is shown in Figure **3**.**2**.

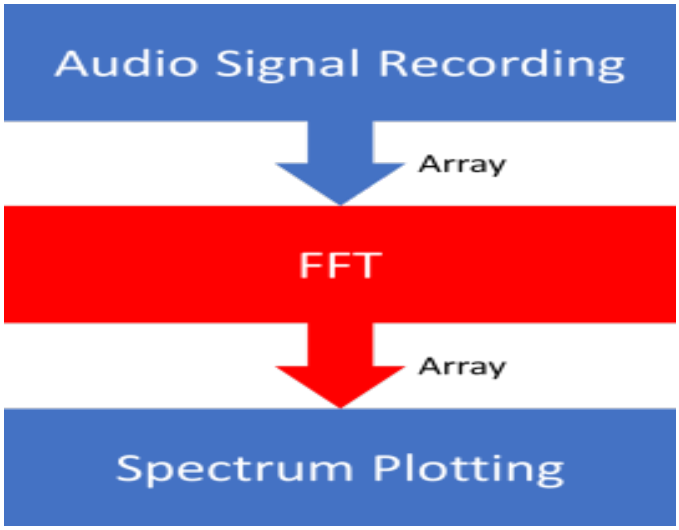

Figure 3.1: Flow Chart of Spectrum Analysis App without Offloading

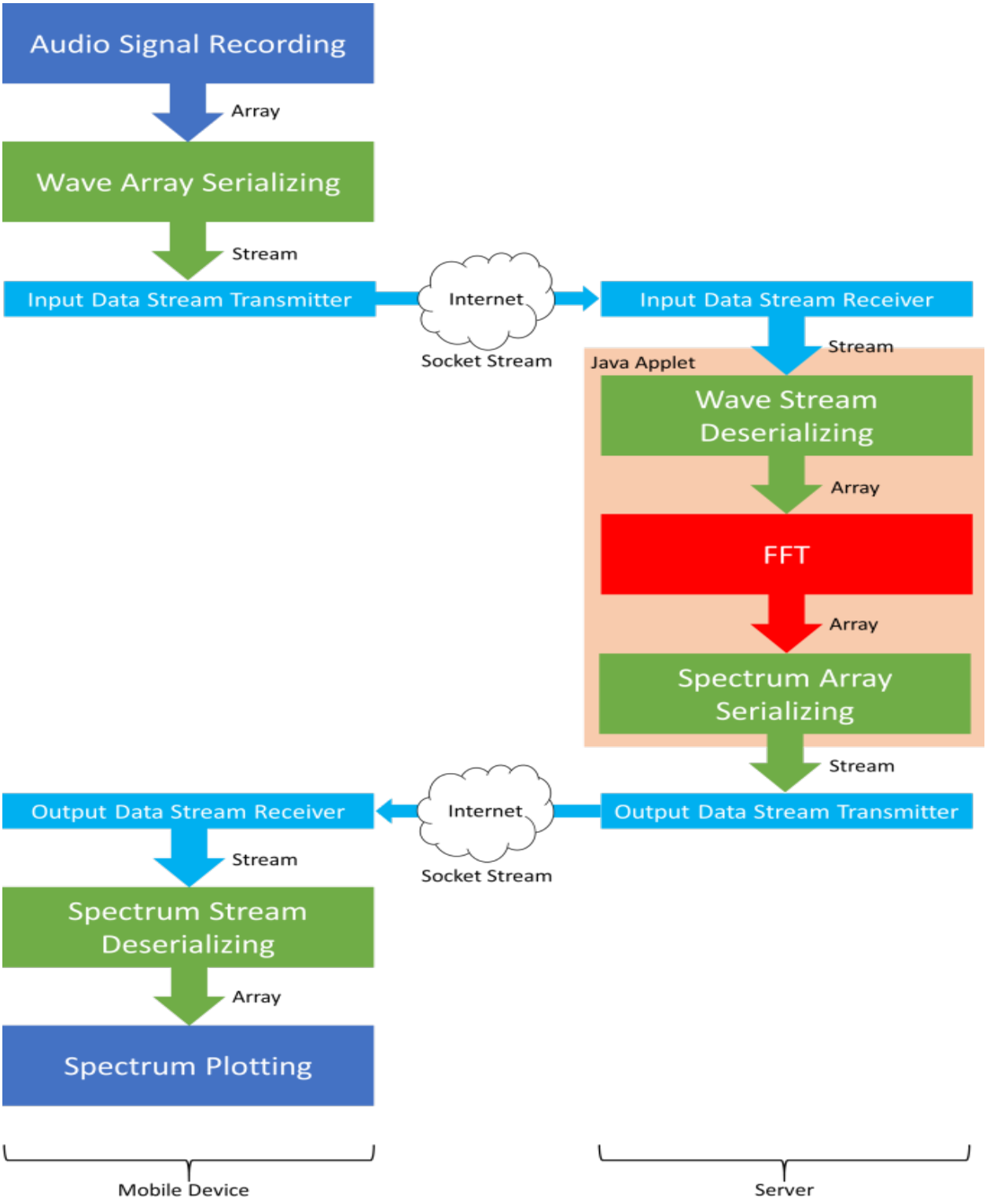

Figure 3.2: Flow Chart of Spectrum Analysis App with Offloading

As we can see in Figure **3**.**2**, the original flow chart is slightly modified. The app still keeps some functions, such as recording and sampling background noise and plotting spectrum on the screen, which are not specifically computationally intensive. The FFT processing block, however, is separated and moved to a server. In order to send recorded audio sample to and receive the spectrum data from the FFT processing block, which resides on the server side now, we insert several additional interface blocks to the flow diagram. In the proposed offloading framework, data transmissions of both directions are serialized as streams (casted as socket streams) to simplify and unify the communication interface. In this application, at first the recorded audio data is sampled and stored as an array object. Then it is serialized into a stream before transmitted over the Internet, and deserialized back to an array object after received on the server side to feed the FFT processing block. The output data generated by the FFT processing block, i.e., the spectrum array, goes through the same conversion to traverse back and be plotted on the mobile device.

Comparing to Figure **3**.**1**, the extra blocks converting data object back and forth twice in Figure **3**.**2** make the app with offloaded FFT processing look very inefficient. However, the data conversion is a relatively low cost practice as 1-dimensional arrays can be easily serialized to and deserialized from stream objects. It is believed that the saving from offloading the FFT process outweighs the overhead in data conversion. Additionally, a server usually has more advanced hardware, such as more powerful CPU and larger memory, and its capability is not bounded by battery consumption. The overall performance, in terms of the average execution time and battery consumption rate, of some apps will be improved in the proposed framework. The hypothesis will be proved by the data in the following part.

3.2 Experiments

In order to observe the performance improvement and battery usage, we suspend the dynamical offloading mechanism, which automatically decides whether an offloadable block should be offloaded or running locally according to the communication cost and service availability information available at the time. Therefore, the decision marker and the service status monitor are omitted in our experiment. We used the two aforementioned apps, the original background noise analyzer app, which exclusively runs on the mobile device, and the forked one, which offloads the FFT processing block to the server. Furthermore, we assume the FFT applet is always available on the server. Therefore, only input and output data streams traverse between the mobile device and the server.

3.2.1 Information of Test Equipment

We use a Samsung Galaxy S III mini 8GB as the test mobile device. This smartphone is equipped with a dual-core ARM Cortex-A9 processor running at 1 GHz, 1 GB RAM, and 8 GB flash memory as secondary storage. It runs Android 4.1.2 (Jelly Bean) out of box without any firmware update or tweak. Although it supports GSM and HSDPA connectivity as most contemporary smartphones in US market, we only use Wi-Fi connection in the experiment. Its 1500 mAh batter pack is relatively small, which helps us to easily observe energy consumption from the battery bar.

On the server side, we used a MacBook Pro to host our server program. It is equipped with a dual-core Intel Core i7 at 2.9 GHz, which can be overclocked up to 3.6 GHz with Turbo Boost technology, 8 GB PC3-12800 DDR3 SDRAM, and a 750 GB 5400-rpm hard drive. The server program runs on Java SE Runtime Environment build 1.7.0_07-b10 on top of OSX 10.8.4.

3.2.2 Experiment Result and Analysis

To observe the performance difference, we measure the time lapses for both apps to analyze a 2-second wave file by performing 32768-point FFT and display the spectrum plotting for 30 times. The execution time comparison is shown in Figure **3.3**. In average, the original app takes 15.52 seconds to complete the analysis while the forked app without offloaded FFT block takes only 2.27 seconds. Therefore, in spite of the overhead on data conversions and transmissions, the proposed offloading framework still boost the performance of the sample app by 6.85 times.

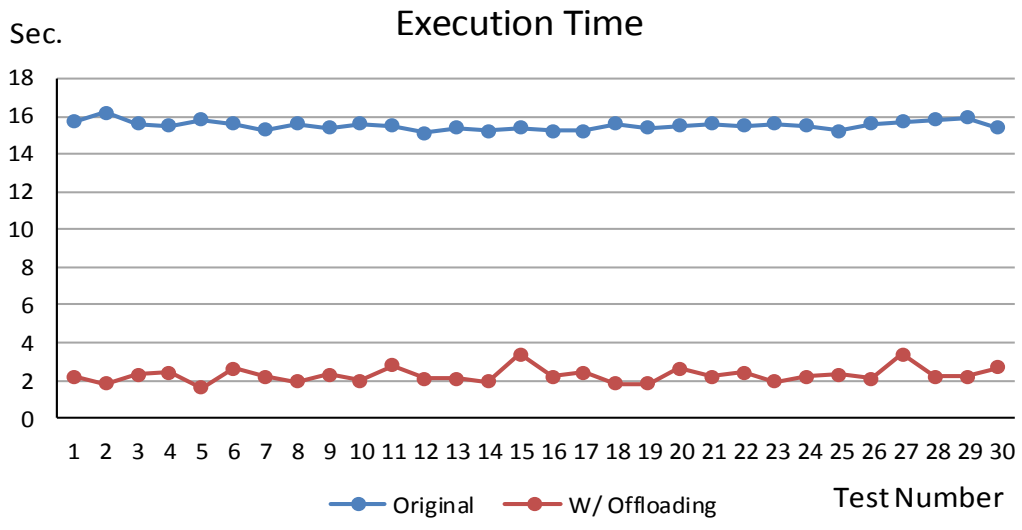

Figure 3.3: Execution Time of the Sample App with and without Offloading Support

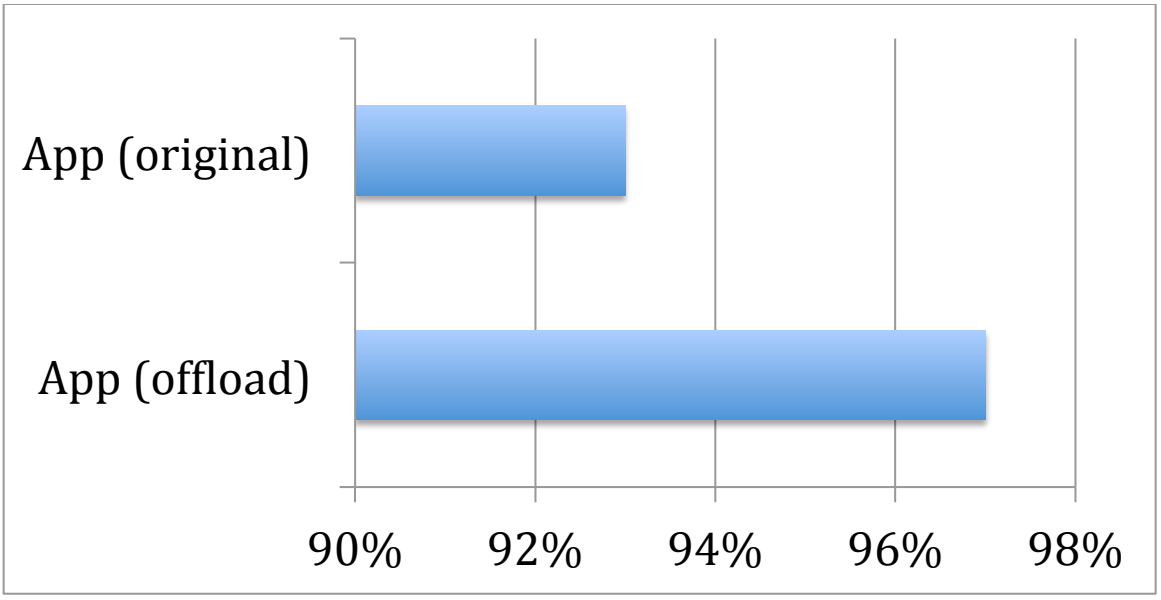

Figure 3.4: Remained Battery Energy

To compare the battery energy consumption, we run each app 100 times right after the battery is fully charged then use KingSoft Battery Doctor to check the remaining percentage of battery energy. The mobile device retains only 93% of the batter energy after running the original app 100 times. In Comparison, it retains 97% of the battery energy after running the forked app with offloaded FFT 100 times. The result is shown in **Figure 3.4**.

According to Figure **3.3** and **Figure 3.4**, we can conclude that both speed and energy consumption of processing computationally intensive tasks are improved significantly with the proposed mobile framework. The server, which is hosted on a general personal computer, is not only equipped with a higher performance processor and larger memory than contemporary smart phone, but also is highly upgradable and scalable. In the proposed mobile framework, the relatively high performance server computer can significantly reduce the processing time of the

FFT block. Even though penalized by the data conversion and communication overhead, the execution time is still shorter than it is in the original arrangement. More importantly, the server's computational capability is not bounded by battery energy (although the energy consumption of data centers has become a critical issue). Therefore, app developers can employ excessively complex algorithms without worrying about the battery drain as long as the execution of computational intensive blocks is carefully offloaded.

Chapter 4

Distributing System

We believe, in the future, more and more tasks will be offloaded to servers and eventually become overwhelming. Grid computing is an excellent approach to accommodate the new load. Grid computing is a family of distributed computing. It can enable organizations to utilize either well-organized computer cluster, or remaining CPU time of their existing computer to perform computational intensive tasks [17]. In grid computing, each computational project is partitioned into numerous independent tasks. A server supervises many computers, i.e., client nodes. When the server detects one of the computers is idle, it pushed a new task to the idle node. The server is also responsible of retrieving and aggregating the results from the nodes.

In addition to utilizing dedicated computer cluster, grid computing technology also enables utilizing unused computational resource on personal computers. Today, personal computers have become overly powerful for most users. When performing some simple activities, such as browsing the Internet, chatting with others or editing an office document, computers spend most of the time on waiting for the next keyboard or mouse event and the CPU time is wasted. Therefore, some researchers developed volunteer computing technology to utilize the CPU time by harnessing personal computers with client software. [18] Among the volunteer computing technologies, Berkeley Open Infrastructure for Network Computing (BOINC) is arguably the most prominent framework. With BOINC, users can create and deploy their own grid computing applications.

There are some advantages to implement the server in the proposed framework with grid computing technology. First of all, most offloadable tasks are independent and can be processed simultaneously. Secondly, the server can be implemented and scaled up with virtually no cost by aggregating unused computational resources on current operational computers rather than piling up dedicated blade server computers on racks. Finally, since the server only monitors participating computers and forwards tasks to the idling ones, it is less likely to be overwhelmed by high quantity of incoming offloading requests.

4.1 Framework of BOINC

Among the volunteer computing technologies, Berkeley Open Infrastructure for Network Computing (BOINC) is arguably the most prominent framework. BOINC is an open source middleware system for grid computing created in 2002 [24]. It provides a platform for organizations to publish their own computational applications and for volunteers to participate in some scientific projects.

Figure **4.1** shows the basic infrastructure of BOINC. BOINC consists two mainly parts: a server system and client computers. A server system can monitor its client computers' status. If a client is idle, it will be issued at least one new work unit, which do not require more RAM than it has. The clients run the work unit, each comprises of an executable application and input files. Once the computation is completed, the output data is saved as a file and uploaded back to the server. The server can subsequently issue new work unit to the client to keep it from idling [19].

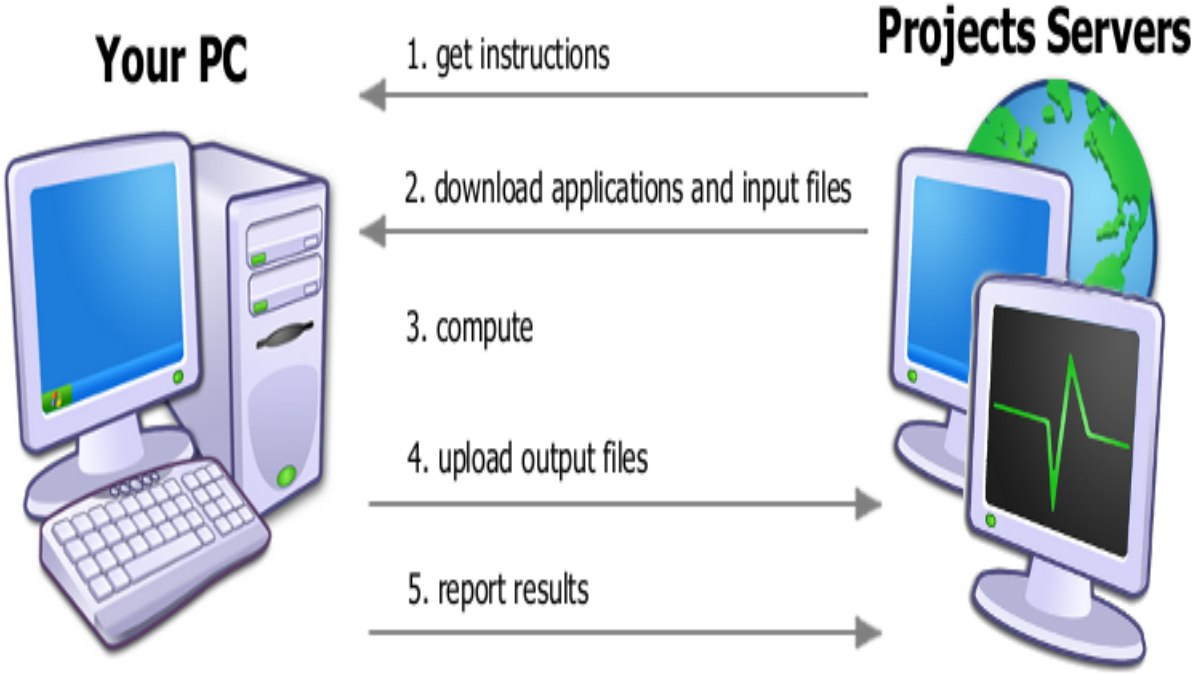

Figure 4.1: Basic Infrastructure of BOINC [19]

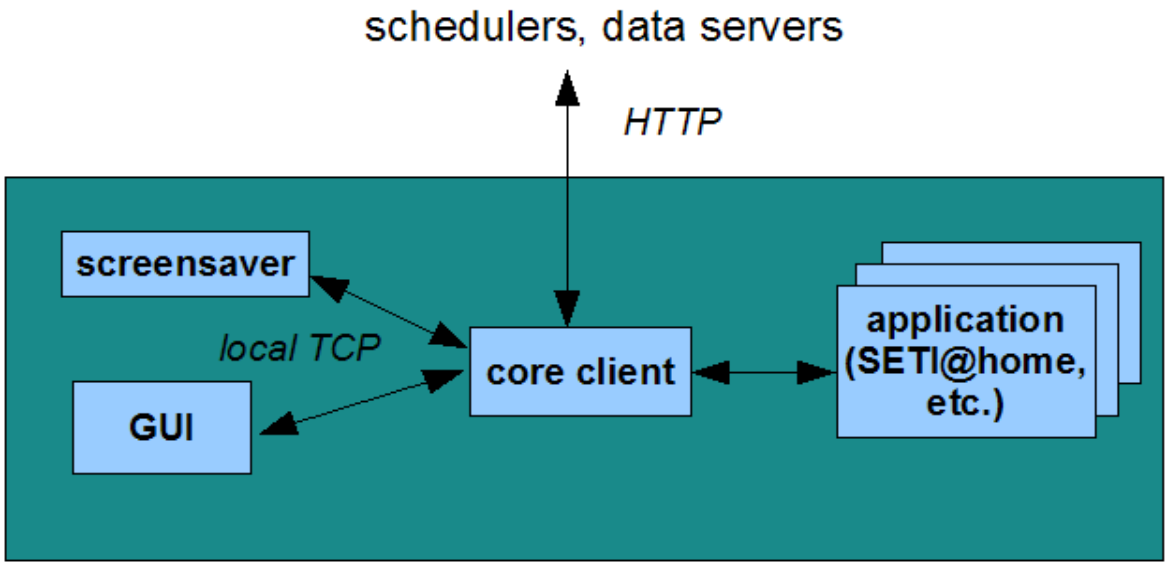

Figure 4.2: Flow Chart of BOINC Software [19]

The Figure **4.2** shows how the BOINC software works. The green box is installed in a user's computer. There are four components in the green box: Application, Core Client, Screensaver, and GUI. Each application is a scientific computing program. A work unit can run several applications simultaneously if it has more than one processor core. The result of each application is sent to Core Client when available. Core Client then subsequently exchanges works with scheduler and data sever through an HTTP session. GUI part uses TCP to control Core Client and interact with users to start, suspend and resume an application. Screensaver part is not essential, which only informs an application to generate screensaver graphics.

If we can make use of BOINC to process mobile offloading tasks, offloading blocks will be saved as Applications in the server. When a mobile device sends a request along with the input data to the server, the server recognizes the associated Application and sends both the Application and input data to an idle Core Client as a work unit. The Core Client then uploads the result back to the server with through HTTP session.

The operation loop is technically sensible. However, we still need to evaluate the feasibility of using BOINC to implement the server in the proposed framework. In our test, we use BOINC Server Debian Virtual Machine, which can be run upon VirtualBox, to setup up a BOINC Server and create our own BOINC project. It provides the necessary directories to run a project and MySQL, Apache2, and PHP, which are necessary software, are installed in it.

4.2 Test

We used a MacBook Air as the test tool. It is equipped with a dual-core Intel Core i5 at 1.3 GHz, 4 GB 1600 MHz DDR3, and a 128 GB Flash-Storage hard drive. The operating system is OS X Version 10.9.5. The VirtualBox version is 4.2.18 r88780. The BOINC Server Debian Virtual Machine is equipped with Debian System based on UNIX. Its base memory is 384 MB. We set up a BOINC server by using a BOINC server VM, whose image can be downloaded in the official website, with all the necessary software already installed. This virtual machine can be run in VirtualBox on this Mac OS X. Also we use VirtualBox to create another VM instance to simulate a client computer. The user Operating System is Ubuntu. Its base 2048 MB memory, two network adapters: Intel PRO/1000 MT Desktop (NAT) and Intel PRO/1000 MT Desktop (Internal Network, ‘intnet’) and BOINC client software installed. Then we create an application, which can be executed by the client VM instance, and add it on the server. In this test, we use the uppercase application, which can transform all the characters of an ordinary ASCII file into upper case, as the example project.

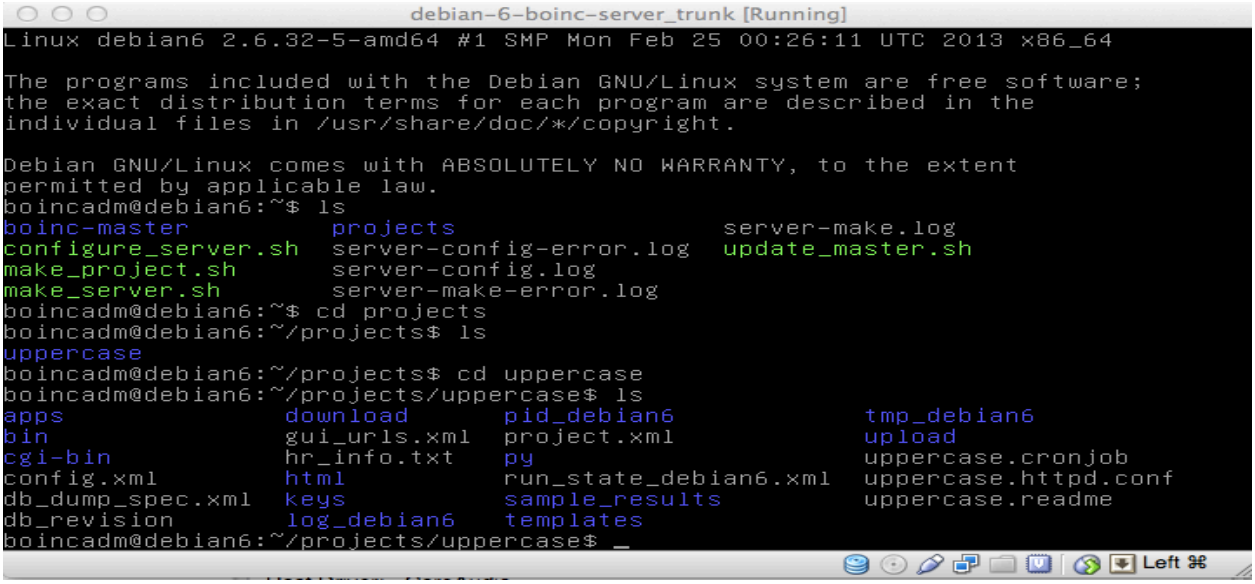

Figure 4.3: Debian BOINC Server

In Figure **4.3**, the apps folder contains an executable file named uppercase. We place input text files in the download folder. And when the client computer is idle, it downloads the executable file and input file from the server and uploads an output file to the upload folder, when it has completed the computation task.

Figure 4.4: Input File

Figure 4.5: Output File

From the Figure **4**.**4** and Figure **4**.**5**, there are 20 ‘a’ in an input file and 20 ‘A’ in an output file. Based on this hand-on test, we learn that BOINC is driven by scripts to automate the data input, execution, and data output as files. That is the execution control has a very transparent interface to work unit issuer. Therefore, BOINC allows users to easily build their own project and use other computers to do computation tasks with a proper script.

In the future, Android developers can offload computationally intensive blocks as executable files along with input data and script files to the server. The whole offload request is subsequently refactored as a BOINC work unit. When the data is processed, the result is written into a file and sent back to mobile devices from the BOINC server.

Chapter 5

Discussion and Conclusion

Google Android provides an open-source platform for developers to freely customize their phones and develop fantastic mobile apps. The proposed mobile framework uses offloading and grid computing to improve their performance: execution time and consumption of battery energy.

5.1 Discussion

In the proposed framework, the server can help mobile devices to process complex computational tasks, which potentially take long time and drain much battery energy on mobile devices. The computational tasks on the server can further be offloaded to other computers with surplus resources using BOINC client-server architecture to scale up. Backed by scalable servers in data centers, mobile computing devices with the proposed offloading scheme are able to provide more computation cycles per battery charge without compromising their portability, performance, software features, and hardware cost. Additionally, the framework provides a simple and friendly interface for Android app developers. Developers only need to extract offloading part from their apps, upload relevant offloading applet to the execution server and insert a data conversion function. They are not required to modify most parts of their code. We also believe that the proposed framework will enable new generation of applications on mobile computing devices, leveraging virtually unlimited computing resources in the cloud.

It is easy for developers to know the time and space complexities of each offloading task based computer science analysis. However, estimating the server execution time is very difficult. There are a lot of factors, which need to be considered. For example, the network delay will be longer if the server and a mobile device has a long geographical distance or the networking technology is too low. Hence, the estimation of the processing time in the proposed framework is a mixture of analytical, empirical, and statistical approaches, rather than pure analytical.

The background noise analyzer app used to demonstrate the advantages of offloading is an extreme sample. We will develop more sample apps, which can better present the real world apps and improve the generality and completeness of the API of the proposed framework.

5.2 Related Work

Kwon and Tilevich have proposed a distributed mobile execution framework [20], which is also based on the offloading concept. The approach, however, is quite different from the proposed framework.

Although both frameworks require source code to be altered and recompiled, the approach taken in [20] is very different from the proposed one. In [20], the authors focus on the programming model to enable offloading and exception handling when network is unavailable, while the proposed framework is essentially an API to unify the interaction between mobile devices and servers. Furthermore, in [20], the authors do not define a unified interface for the server to host

offloaded tasks. The server-side processing is ad-hoc to each offloadable task.

In contrary, the proposed framework provides a unified, intuitive, and transparent interface for both offloadable classes and corresponding servers. More importantly, using serialized single-input, single output (SISO) data interface for offloadable tasks significantly simplify the potential integration with BOINC.

5.3 Future Work

In the future, mobile apps firstly communicate with the server, when they are ready to execute offloading task. The server should look up whether a relevant executable applet is already existed in it. If it cannot find the relevant executable applet, mobile apps have to upload offloading blocks to the server and save them as applet files in the server. It is convenient for other users, when they use same apps to process the data.

Also, information security is very important in the society. Sending data and receiving data may become a potential threat to the information security. In the future, we would like to develop a more secured protocol for the communication between servers and mobile devices.

Finally, we expect to test the connection between the BOINC server and mobile devices. It is an important part because when the BOINC server gets results from clients, results should be sent back to mobile devices.

5.4 Conclusion

This thesis describes a new proposed mobile framework, which can accelerate the data processing and save battery energy. By offloading proper functional blocks, which are computational intensive and have low data dependency to the rest of the app, to the server, a mobile device can save significant amount of battery energy and provide more responsive user experience. App developers only need to put a little extra effort on adapting an existing Android app to the proposed offloading framework and significantly improve the battery life and user experience.

To evaluate the performance and efficiency of the proposed framework, we developed a sample app with conventional and offloading configurations. The experiment results fully demonstrate that the advantages of the proposed framework. In the future, some ideas can be added in the framework and make the framework more advanced, intelligent and efficient.

Bibliography

[1] http://en.wikipedia.org/wiki/Google_Maps#cite_note-5

[2] http://en.wikipedia.org/wiki/Flappy_Bird

[3] http://www.engadget.com/2013/10/31/strategy-analytics-q3-2013-phone-share/

[4] http://en.wikipedia.org/wiki/OS_X

[5] http://tech.co/ios-vs-android-app-development-consumer-experience-comparison-2014-02

[6] http://en.wikipedia.org/wiki/Android_(operating_system)#cite_note-AndroidOverview-12

[7] K. Yang, S. Ou, and H. H. Chen, "On effective offloading services for resource-constrained mobile devices running heavier mobile Internet applications," IEEE Communications Magazine, vol. 46, no. 1, pp. 56-63, Jan. 2008.

[8] T. Erl, R. Puttini, and Z. Mahmood, Cloud Computing: Concepts, Technology & Architecture. 1st ed., Prentice Hall, N. J, 2013.

[9] C. Baun, M. Kunze, J. Nimis, and S. Tai, Cloud Computing: Web-Based Dynamic IT Services. 2nd ed., Springer-Verlag Berlin Heidelberg, N.Y, 2011.

[10] D. Kovachev, and R. Klamma, "Beyond the clinet-server architectures: A survey of mobile cloud techniques," IEEE int. Conf. on Communications in China Workshops (ICCC), Beijing, China, Aug.2012, pp. 20-25.

[11] G. Chen, B. T. Kang, M. Kandemir, N. Vijakrishnan, M. J. Irwin, and R.Chandramouli, "Studying energy trade offs in offloading computation/compilation in Java-enabled

mobile devices," IEEE Trans. Parallel and Distrib. Sys., vol. 15, no. 9, pp. 795-809, Sept. 2004.

[12] S. Park, Y. Choi, Q. Chen, and H. Y. Yeom, "Some: selective offloading for a mobile computing environment," IEEE int.

[13] E. Y. Chen, S. Ogata, and K. Horikawa, "Offloading android applications to the cloud without customizing Android," in IEEE int. Conf. on Pervasive Computing and Commun Workshops (PERCOM Workshops), Lugano, Switzerland, Mar. 2012, pp. 788-793.

[14] H. Y. Chen, Y. H. Lin, and C. M. Cheng, "COCA: Computing offload to clouds using AOP," IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 2012, pp. 466-473.

[15] K. Restivo, R. Llamas, and M. Shirer. (2013, May 16). Android and iOS Combine for 92.3% of All Smartphone Operating System Shipments in the First Quarter While Windows Phone Leapfrogs BlackBerry, According to IDC [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS24108913

[16] "Java SE Embedded Performance Versus Android 2.2" [Online] Available: https://blogs.oracle.com/javaseembedded/entry/how_does_android_22s_performance_stack_up_against_java_se_embedded

[17] http://boinc.berkeley.edu/trac/wiki/DesktopGrid

[18] http://w3.linux-magazine.com/issue/71/Distributed_Applications_With_BOINC.pdf

[19] http://boinc.berkeley.edu/wiki/How_BOINC_works

[20] Young-Woo Kwon and Eli Tilevich, "Power-Efficient and Fault-Tolerant Distributed Mobile Execution," in 32nd International Conference on Distributed Computing Systems (ICDCS 2012), 2012.

[21] http://en.wikipedia.org/wiki/Mobile_phone

[22] http://en.wikipedia.org/wiki/History_of_mobile_phones

[23] Distributed Applications with BOINC

[24] http://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing