2018

Mobile Cloud Computing: A Comparison Study of Cuckoo and AIOLOS Offloading Frameworks

Inan Kaddour
University of North Florida, n00669880@ospreys.unf.edu

MOBILE CLOUD COMPUTING: A COMPARISON STUDY OF CUCKOO AND AIOLOS OFFLOADING FRAMEWORKS

by

Inan Kaddour

A Thesis Submitted to the
School of Computing
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April 2018

The Thesis "Mobile Cloud Computing: A Comparison Study of Cuckoo and Aiolos Offloading Frameworks" submitted by Inan Kaddour in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee:	Date
______________________________ Dr. Sanjay P. Ahuja Thesis Advisor and Committee Chairperson	____________________
______________________________ Dr. Roger Eggen	____________________
______________________________ Dr. Behrooz Seyed-Abbassi	____________________

Accepted for the School of Computing:

______________________________ ____________________

Dr. Sherif Elfayoumy
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

______________________________ ____________________

Dr. Mark A. Tumeo
Dean of the College

Accepted for the University:

______________________________ ____________________

Dr. John Kantner
Dean of the Graduate School

ACKNOWLEDGEMENT

I want to thank my parents for always being my greatest motivation and pushing me to get where I am today, this achievement is yours.

I would also like to take this opportunity to thank my Thesis Advisor Dr. Sanjay Ahuja for his valuable guidance, support, and time he dedicated to me through all this process; thank you Dr. Ahuja. I also thank Dr. Roger Eggen and Dr. Behrooz Seyed-Abbassi who agreed to be on my thesis committee. I also want to thank Tim Verbelen for his valuable help and guidance in building the Aiolos based application. I also want to thank Roelof Kemp for helping me build the Cuckoo based application. I also thank School of Computing and the ISA for all the support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Currently, smart mobile devices are used for more than just calling and texting. They can run complex applications such as GPS, antivirus, and photo editor applications. Smart devices today offer mobility, flexibility, and portability, but they have limited resources and a relatively weak battery. As companies began creating mobile resource intensive and power intensive applications, they have realized that cloud computing was one of the solutions that they could utilize to overcome smart device constraints. Cloud computing helps decrease memory usage and improve battery life. Mobile cloud computing is a current and expanding research area focusing on methods that allow smart mobile devices to take full advantage of cloud computing. Code offloading is one of the techniques employed in cloud computing with mobile devices. This research compares two dynamic offloading frameworks to determine which one is better in terms of execution time and battery life improvement.

Chapter 1

INTRODUCTION

Offloading, also called augmented execution, is the method of sending a resource intensive task to a remote server; an old technique that has been rediscovered to reduce power consumption and speed up computation tasks. Since the beginning of mobile computing in the early 1990s, the lack of resources of mobile devices has been identified as a major constraint.

Mobile elements are resource-poor relative to static elements. Regardless of future technological advances, a mobile unit's weight, power, size, and ergonomics will always render it less computationally capable than its static counterpart. While mobile elements will undoubtedly improve in absolute ability, they will always be at a relative disadvantage [Satyanarayanan14]. The data shown in Figure 1 illustrates that the previous statement remains correct even with technological advances in mobile devices, as their resources remain limited when compared to a typical server.

In 1997, to improve execution time, offloading was first introduced in mobile computing by Noble et al. in the Janus speech recognition application [Noble97]. The application was modified to operate in three modes in Odyssey. The latter is a platform for mobile data access, it monitors mobile device resources such as bandwidth, battery power and

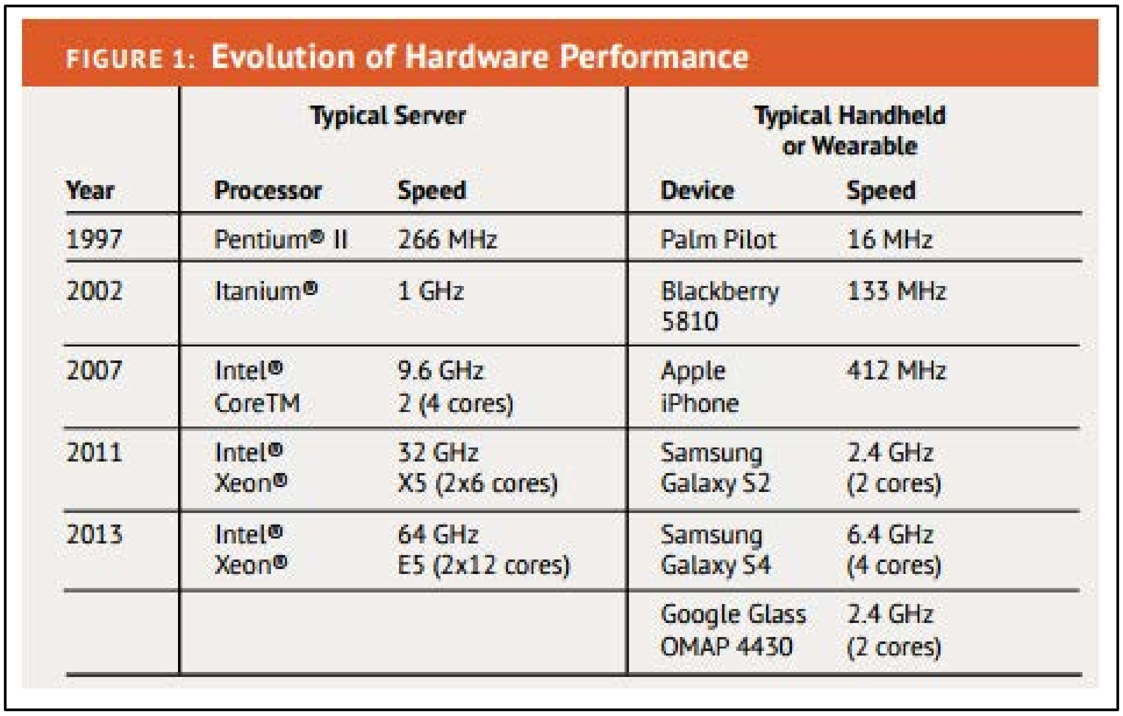

FIGURE 1: Evolution of Hardware Performance

	Typical Server		Typical Handheld or Wearable	
Year	Processor	Speed	Device	Speed
1997	Pentium® II	266 MHz	Palm Pilot	16 MHz
2002	Itanium®	1 GHz	Blackberry 5810	133 MHz
2007	Intel® CoreTM	9.6 GHz 2 (4 cores)	Apple iPhone	412 MHz
2011	Intel® Xeon®	32 GHz X5 (2x6 cores)	Samsung Galaxy S2	2.4 GHz (2 cores)
2013	Intel® Xeon®	64 GHz E5 (2x12 cores)	Samsung Galaxy S4	6.4 GHz (4 cores)
			Google Glass OMAP 4430	2.4 GHz (2 cores)

Figure 1: Hardware Comparison between Servers and Mobile Devices.

CPU cycles, and then notifies running mobile applications when it detects a change to those resources. The first mode was local execution of the application, the second mode was remote execution of the application. The last mode was a hybrid, where the first phase, which is the conversion of raw speech to a more structured representation of the speech processing application was executed locally, and the second phase, which is the reminder of the speech recognition process, was executed on the server. Odyssey had the ability to dynamically decide the optimal execution mode based on many factors such as network bandwidth. Flinn demonstrated that remote execution could save battery energy [Flinn99].

The appearance of cloud computing in 2008 addressed a very important question around offloading which was “where should remote execution take place?” The success of

Apple's cloud-based Siri speech recognition service validates the use of clouds at commercial levels and opened a new era of cloud offloading. Offloading is considered a better option than online applications for two main reasons [Kovachev11]. The first is that users do not always have access to the Internet. The second is that online applications cannot gain access to the phone's features such as camera or motion detection. There are two types of offloading: static and dynamic. Static offloading is when the tasks to be executed on the cloud are identified at compile time or runtime. Dynamic offloading is when an external resource manager determines whether to run a specific task locally or on a remote server to achieve better performance and longer battery life. There are two main offloading approaches. The first approach requires a framework on the top of the existing runtime system, for example Mobile Assistance Using Infrastructure (MAUI), Cuckoo, ThinkAir, Aiolos, and Mobile Cloud Middleware (MCM) framework. The second approach requires a modification to the operating system (OS) or virtual machine on which the process is running [Verbelen12A]. As a result, this modification makes it hard for this approach to be a real-world approach for offloading, due to security concerns associated with modifying the OS. CloneCloud is an example for the second approach [Paramvir12]. Chun and Al developed an architecture that supports five types of augmented execution [Chun09]:

1. Primary functionality outsourcing: offloading computational intensive tasks.
2. Background augmentation: offloading background processes.
3. Mainline augmentation: offloading light-weight computation for heavy weight analysis.

4. Hardware augmentation: offloading of computation because of hardware limitation.
5. Augmentation though multiplicity: parallel execution of offloaded tasks.

Many of today's mobile applications such as augmented reality applications, do expensive computations locally which affects response time and energy consumption. On the other hand, applications should not be fully dependent on the Internet or Wi-Fi connection. Users should be able to run their resource intensive applications regardless of whether the Internet can be accessed or not. The use of a dynamic offloading framework will help to resolve these issues by executing extensive computation tasks on the cloud whenever it is possible, instead of executing them locally. However, if the cloud is unreachable, then a dynamic offloading framework will execute extensive computation tasks locally. Cuckoo and Aiolos are two open source dynamic offloading frameworks that can be used by companies to enhance the performance of heavy computation applications. They follow a client/server model since both frameworks come with client and server components. They support all five execution types indicated above [Kemp10] and they are pioneers in the mobile-cloud offloading domain. This research compares those two frameworks using a commercial cloud provider, Amazon EC2, to determine which one offers better performance in terms of battery life and execution time.

1.1 Android

Android is an open-source operating system that runs on top of Linux and is dedicated to mobile devices. Applications in Android are written in Java and then compiled to Dalvik bytecode. Every application runs on a distinct virtual machine called a Dalvik Virtual Machine (DVM) to avoid interference between applications [Bouzefrane11].

Android has four main components: activities, services, content providers, and broadcast receivers. Activities interact with users through their self- contained user interface. Services are used for CPU or network intensive operations [Kemp12] and they do not have a user interface. Services run in the background where activities and services communicate through inter process communication (IPC) as shown in Figure 2. Content providers handle data access and data sharing between applications. Finally, broadcast receivers are applications that respond to broadcast messages from other applications or other components in the system.

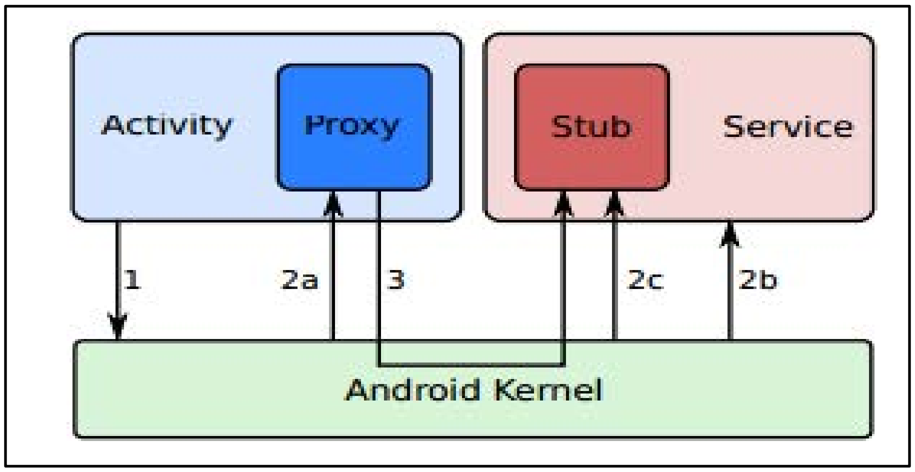

Figure 2: Android Activity Service Communication.

1.2 Android Interface Definition Language (AIDL)

AIDL is an approach used for Inter-Process Communication (IPC). AIDL generates code that enables two android processes to communicate, since one process cannot access the memory of another process [Android18B]. For example, if a developer has a process that needs to call a method in another process (service for example), AIDL is implemented to generate code that allows access to that method [Android18B]. Implementing AIDL requires an update to both processes. AIDL is a light version of COM or Common Object Request Broker Architecture (CORBA) [Aleksandar13] and uses proxy class to pass values between processes. As shown in Figure 3, AIDL is used so that an activity in process A can call methods in process B using an interface defined inside the AIDL file. Eclipse generates a proxy and stub based on the interface. Stub is used to implement all methods defined in AIDL file, and proxy is used in process A to call the remote methods in process B.

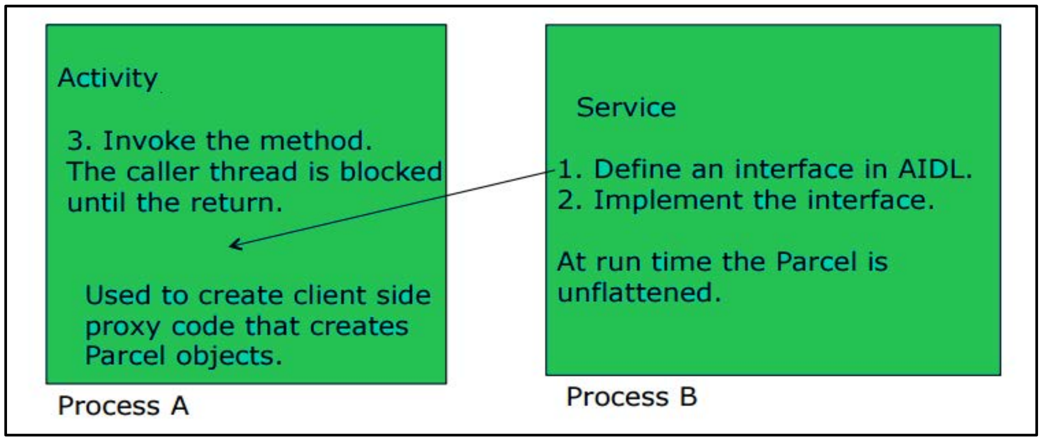
Activity
3. Invoke the method. The caller thread is blocked until the return.
Used to create client side proxy code that creates Parcel objects.
Process A
Service
1. Define an interface in AIDL.
2. Implement the interface.
At run time the Parcel is unflattened.
Process B

Figure 3: Activity Service Communication through AIDL.

IPC uses a Remote Procedure Call (RPCs) mechanism so that an activity can make a direct call to a remote method. Android uses binder as its RPC mechanism [Android18C]. It decomposes method calls and their data to a level that an operating system can understand, transmitting them from the local process to the remote process, and reassembling and reenacting the calls there [Android18C]. As shown in Figure 4, binder kernel driver allows the communication between proxy and stub.

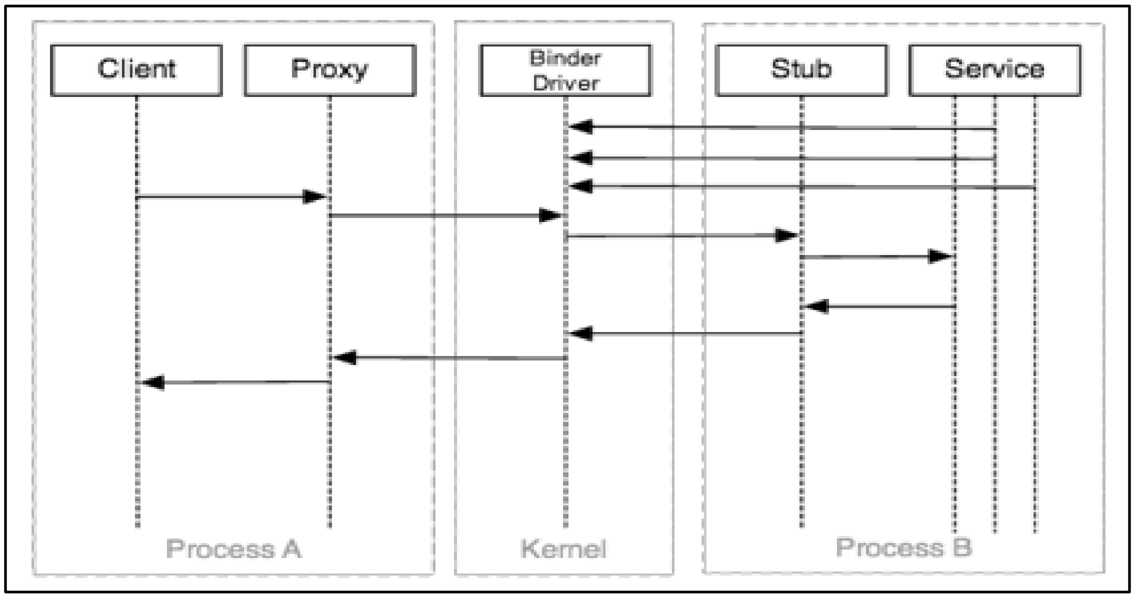

Figure 4: IPC through Proxy-Stub Architecture.

1.3 Open Services Gateway Initiatives (OSGi)

OSGi, which first appeared in 1999, is a framework for a dynamic modular architecture in which an application is composed of multiple reusable components that communicate via services. OSGi has been used in Eclipse Equinox, Apache Felix, GlassFish v3, and other projects [OSGi12].

The purpose of using OSGi is to reduce the complexity of code development. In addition, OSGi fully supports a test-driven development (TDD), which makes it easy to test all components locally. It also enables companies to reuse existing components with minor code modifications [OSGi12]. Additionally, OSGi provides a module cycling/updating capability in order to increase availability and decrease system outages [Hall11]. Finally, OSGi framework comes with an interface that can be used by system administrators or developers to get an insight into current task execution [Aiolos15].

As shown in Figure 5, OSGi consists of three main layers: module, lifecycle, and service layer. The module layer is the core of OSGi because it enables modularity. The OSGi module concept is called a bundle.

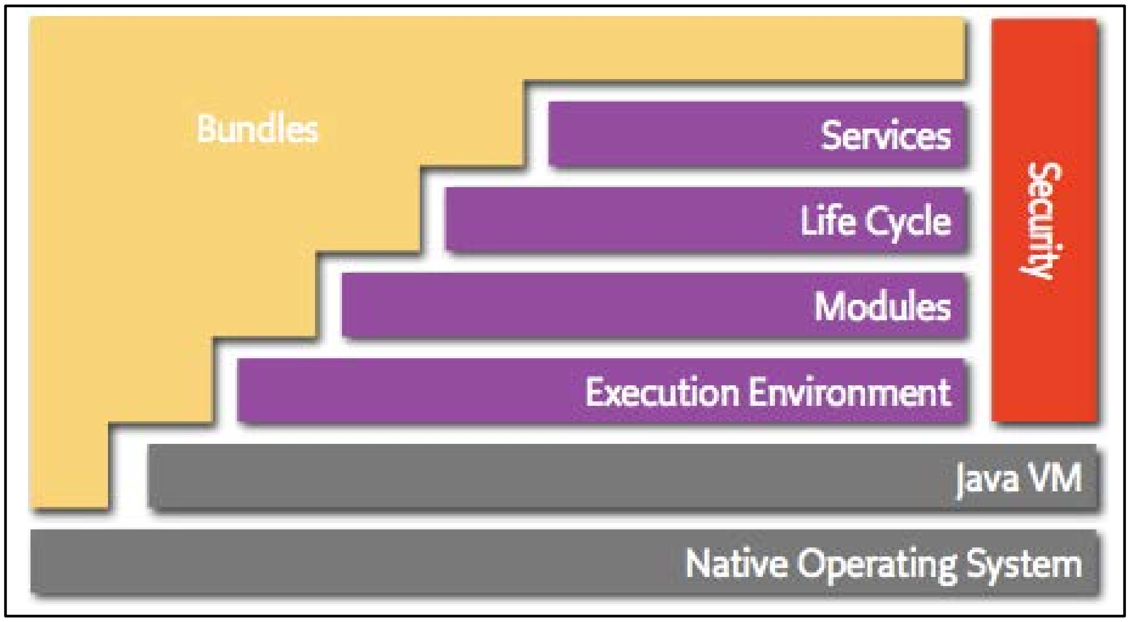

Figure 5: OSGi Model.

OSGi bundle is a JAR file with extra metadata as shown in Figure 6. Unlike a typical Java JAR file, not everything inside a bundle is visible to all other bundles. Embedded metadata contains information about which packages in the bundle are visible to the outside world [OSGi12]. It also contains information about which packages within in the same bundle, and other bundles, it is dependent upon in order to function properly [Aiolos15].

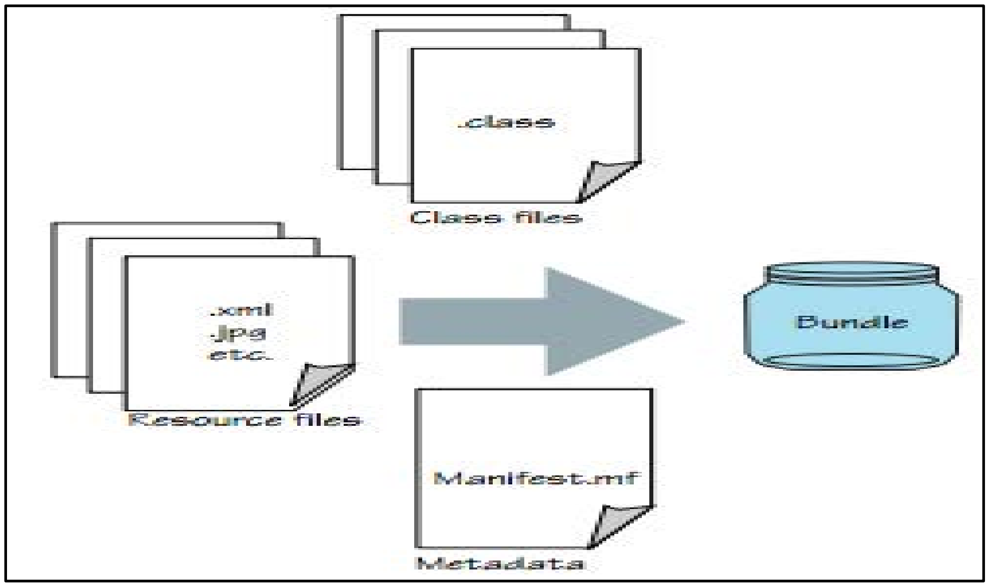

Figure 6: OSGi Module Layer Components

The lifecycle layer provides the ability to dynamically install and manage bundles in the OSGi framework [Hall11]. It also allows bundles to communicate with each other by giving them access to the runtime environment.

The service layer's main goal is to allow communication among modules. It enables a single JVM Service-Oriented Architecture (SOA). OSGi services follow a publish, find, and bind paradigm [Hall11], where service providers publish services to the service registry and service clients search the registry to find available services to use, as shown in Figure 7.

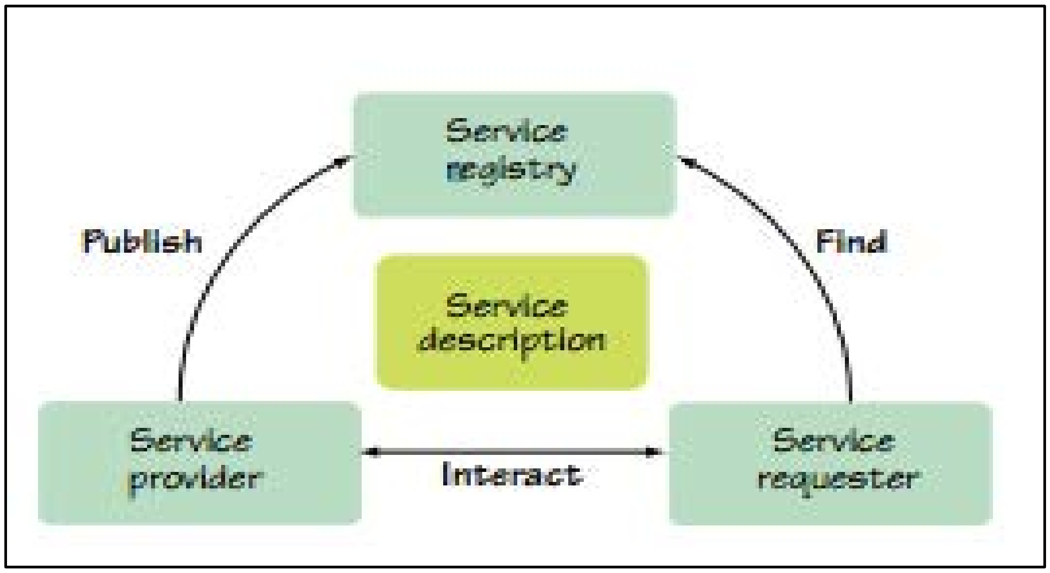

Figure 7: OSGi Service Module - Service Oriented Interaction.

This paper is organized as follows: Chapter 2 defines Aiolos and Cuckoo. It also describes a brief history of existing mobile dynamic offloading research. Chapter 3 explains research methodology, and describes testbed setup. Chapter 4 discusses the results of the experiments. Finally, Chapter 5 concludes the paper.

Chapter 2

LITERATURE REVIEW

2.1 Cuckoo

Cuckoo is a client/server framework for dynamic offloading. It only targets Android devices and takes advantage of how android's main components, activities, and services, communicate. As shown in Figure 8, Cuckoo comes with the following components: a very simple programming model and environment (Eclipse plugin), a runtime, oracle, a resource manage application, and server application.

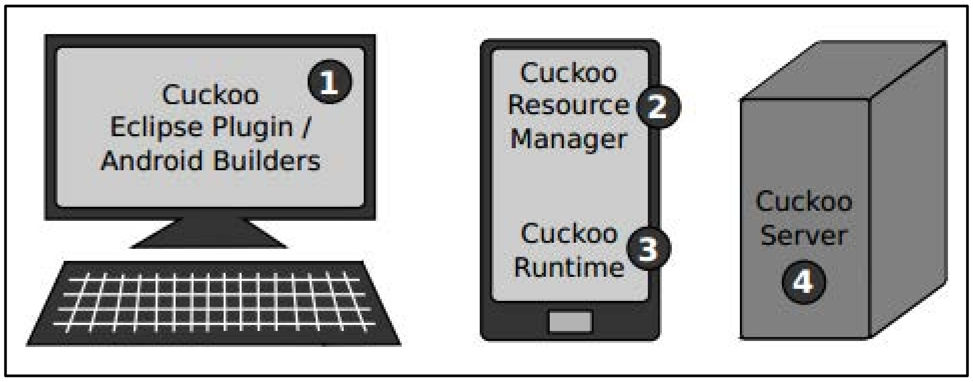

Figure 8: Cuckoo Components.

Cuckoo has many advantages. It bundles local and remote code in the same package so that the offloaded code can be installed from smart devices at runtime. It allows different implementations of local and remote code of the same function to better utilize cloud resources. As shown in Figure 9, Cuckoo comes with an Eclipse plugin to integrate with

Eclipse to facilitate the creation of computation offloading applications. Once an AIDL file is created by developers, Cuckoo Service Rewriter (4) adds code to generated java files so that Cuckoo can intercept service method calls and run an offloading algorithm against each method call to decide whether to execute the method locally or on the server. Cuckoo Remote Service Deriver (2) generates dummy service implementations which need to be overwritten by developers. Ant Compiler (3) is used to create an apk file that will be installed and run on the server.

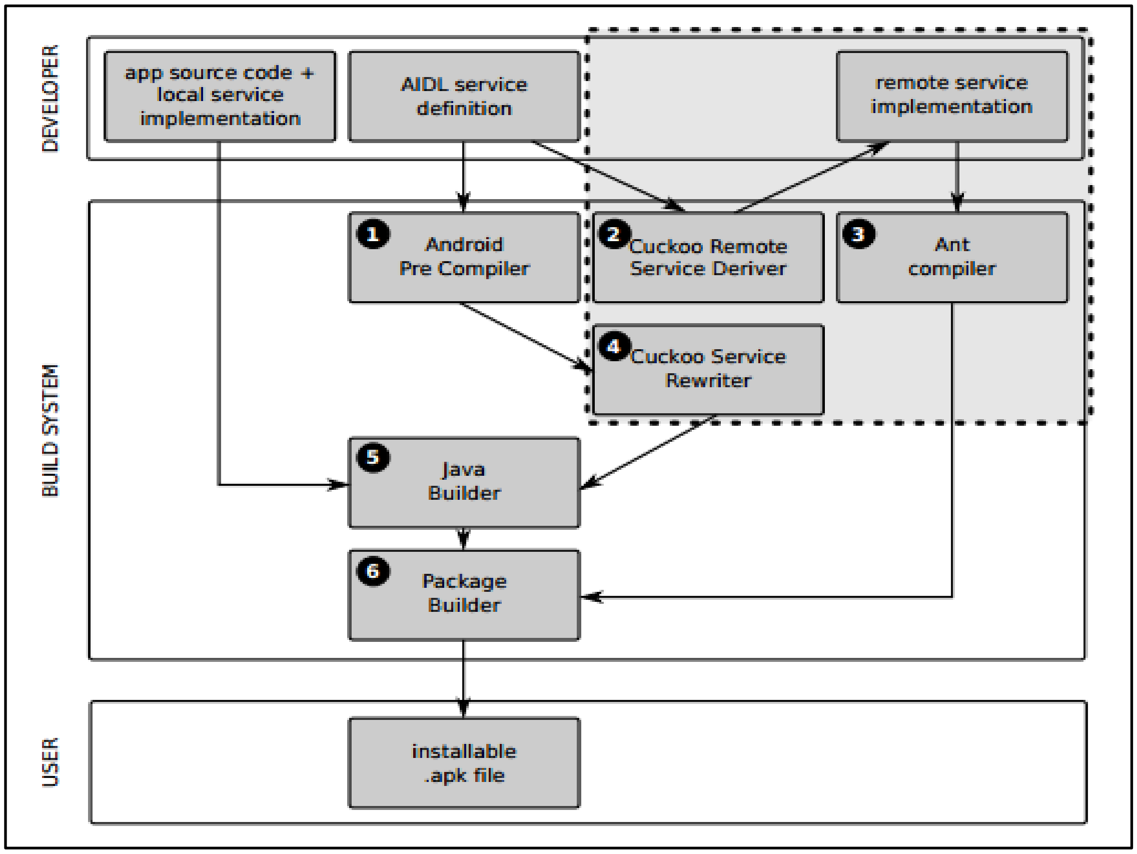

Figure 9: Cuckoo Build Process.

Oracle is the decision maker component of Cuckoo. Decisions are based on the strategy chosen by developers, which can be “local”, “remote”, “energy”, “speed” or “parallel”

[Kemp12]. The strategy can also be a combination of “energy” and “speed”. In this case, Cuckoo offloads the execution of the method if it will save more energy or speed-up the execution time. By default, the strategy is “speed/energy”. Oracle uses an algorithm that combines context information, heuristics, and history to decide whether to a run a method locally or remotely. Based on the developer strategy, Oracle estimates execution time, transfer time, round trip time, connection setup overhead, and power consumption on the local and remote servers to decide where to run the method [Kemp12]. The role of the resource manager component of Cuckoo is to make remote resources known to the smart device.

2.2 Aiolos

Aiolos is client/server model framework that is built on the top of OSGi and R-OSGi [Verbelen12A]. The main purpose of using OSGi is to split up the application into components. Those components are independent from each other, which facilitates the offloading process. Aiolos comes with an Eclipse plugin to help developers build off-loadable mobile applications. They are only required to annotate classes they want to consider for offloading, and the framework will generate OSGi bundles for them and publish them as OSGi services.

To decide whether to run a method locally or remotely, Aiolos uses two optimization models; optimize execution time and optimize energy [Verbelen12B]. To optimize the execution time, the framework calculates the expected execution time locally and remotely based on many factors: speedup factor, network bandwidth, latency, and argument size. A simple decision model is used to optimize the energy consumed. This model always assumes that energy consumed by sending and receiving bytes to and from the server is smaller than the energy saved by offloading the computation [Verbelen12B]. Aiolos also uses a history-based profile for each service method to speed up the decision process. As shown in Figure 10, Aiolos is split up into three layers [Aiolos15]:

- Core: contains Proxy Manager, Remote Service Admin, and Topology Manager.
- Monitoring: collects information about service and node level. It contains a Service Monitor, and Note Monitor.
- Deployment: finds and deploys components to the cloud. It contains Repository, Deployment Manager, and Cloud Manager.

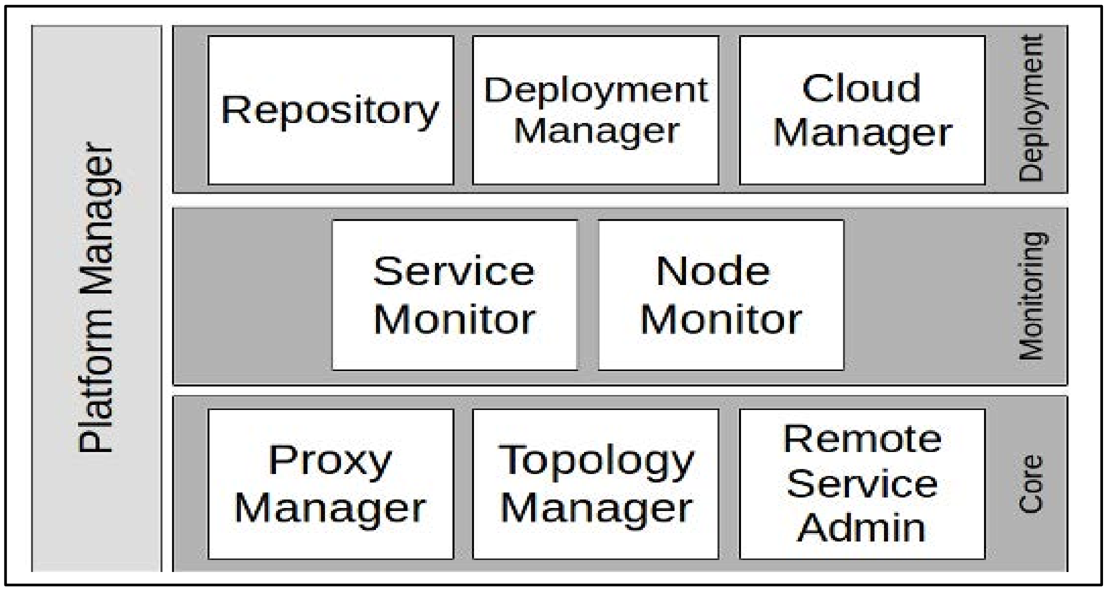

Figure 10: Aiolos Three Main Layers.

2.3 Related Work

The intention of this section is to focus on documenting the contribution of other researchers and to expand the understanding of concepts, models, and patterns of computation on mobile devices. The most important references surveyed are listed below.

There are many studies that talk about the benefits of offloading, but most studies to date only compare available frameworks theoretically without specific examples. A theoretical example is research by Kovachev et al. that compared Alfred O, MAUI, and cloudlets [Kovachev11]. Their work involved comparing various offloading techniques/frameworks in terms of how their architectures work. Research by Kemp et al. in [Kemp10] only discussed the architecture of Cuckoo and its performance using eyeDentify and Photoshoot applications. Their research proved that Cuckoo, as an offloading framework, increases performance of slower phones using an indoor server.

Another framework called Aiolos was introduced by Verbelen et al. [Verbelen12B]. This group's research described the architecture of Aiolos and how it's offloading logic works. It also evaluated Aiolos's performance using Honza's Chess and a photo editor application. They concluded that offloading always improves performance, particularly if the server is local. Also, a user-centric MCC approach was taken by Huang et al. in [Huang13]. In their research, they described context aware applications as the next generation of mobile applications. Those mobile applications are able to collect user's behaviors and attributes [Huang13] in real time to analyze the user's situation and act proactively. It is vital for context aware applications to have an offloading engine to be able to analyze user’s data in a powerful machine. Their research introduced the MobiCloud framework to help developers build context aware applications.

In addition, a few studies explored the possibility of offloading intensive CPU tasks to nearby mobile devices. An example is the work done by Marinelli [Marinelli09], in which the author explored the possibility of executing computational tasks on mobile device networks and heterogeneous networks of phones and servers. Finally, other studies introduced a cloudlet layer between mobile devices and cloud. In a study by Bhatnagar [Bhatnagar13], the author introduced the advantages of using cloudlets by building a face recognition application on the top of the Mobile Cloud Hybrid Architecture (MOCHA) framework.

Web applications today are more complex than ever before, they require more computation resources than mobile devices can supply. To overcome this issue, Wang et al. in [Wang12] developed a JavaScript offloading framework called ExtremeJS (Extensive Transformation and Elastic Migration and Exection of JavaScript). ExtremeJS only works on javascript code. ExtremeJS creates a cloned context of the application on the cloud, and then ships computation intensive functions to it [Wang12]. ExtremeJS comes with three components; profiler, code analyzer, and migrator. Profiler’s job is to identify computation intensive functions by creating a cost model for each function. Then, code analyzer decides which function can be migrated to the cloud. Finally, the migrator is responsible of synchronizing the application contexts and ships the computation intensive functions to the cloud. The framework makes JavaScript websites

10 times faster [Wang12]. This research showed that mobile devices cannot handle modern website so there is a need for a dynamic offloading framework for user to have a flawless experience

Wang et al. in [Wang17] created a cross instruction set architectures (ISAs) offloading framework that is not dependent to any programming language, runtime system or the availability of source code. The framework was built on the top of HQEMU system [Wang17]. It comes with three components; offline profiler, dynamic binary optimizer and dynamic binary translator. Offline profiler resides on the client side, and its job is to analyze binary code to identify which function should be sent to the server. Dynamic binary optimizer resides on the client side. The role of dynamic binary optimizer is to send a function to the server, wait for the results to be returned and then resume the execution of the mobile application. When a request is received, dynamic binary translator, which resides on the server, initializes its internal emulation state according to the received execution state of the target [Wang17]. After the emulation is done, dynamic binary translator sends back the results with the emulation state to the client before entering a wait mode. The framework achieves a 1.93x speedup with 48.66% reduction in energy consumption [Wang17]. That research demonstrated that mobile dynamic offloading is crucial for mobile applications to speedup application's performance and also to reduce energy consumption.

Kim et al. in [Kim16] proposed a dynamic offloading framework for a drone-based mobile system to overcome both limited resources and limited battery power in a drone.

The framework consists of four sub-modules; offloading decision module, image processing, drone positioning & camera control, and remote agency module. The offloading decision module is responsible for determining whether a module should be offloaded or not to reduce response time. The decision is based on mobility information of the target and network conditions. If a module is going to be offloaded, remote agency module sends the input data required for offloading to control center, waiting the execution time for the offloaded computation and then receives the resulting data back [Kim16]. The framework is able to reduce energy consumption and execution time required for recognizing and tracking of moving targets. That research showed that smart machines (such as drone, autonomous cars, and robots) are also in need of dynamic offloading frameworks.

Like the studies mentioned above, this research focuses on calculating energy consumption and execution time when comparing Cuckoo to Aiolos framework. On the other hand, this is the first study that is comparing two open source offloading frameworks that are available for any person or organization to use. The methodology used in this research is adopted from [Wang14] in which a process is executed 30 times on the cloud and locally on the phone. The execution time is measured each time the process is executed. To compare battery consumption, a given task is executed 100 times after the battery is fully charged, and then the remaining percentage of battery energy is checked using KingSoft Battery Doctor application.

Chapter 3

RESEARCH METHODOLOGY

This research evaluates the performance on an LG Leon mobile device with Amazon EC2 as an offloading platform. The study performs and analyzes a series of experiments for Cuckoo and Aiolos frameworks to obtain execution time and power consumption on the mobile device with the mobile device connected to the Internet through either Wi-Fi or 4G or when the phone is offline. Testing involved the device performing two different kinds of computation tasks: heavy computation task, and light computation task. The study also ran several tasks on Cuckoo and Aiolos using different file sizes to find out the impact of file size on performance, and to find the break-even point where both Aiolos and Cuckoo frameworks have the same performance in terms of execution time.

The objectives of using Cuckoo and Aiolos are to shorten the execution time and save the power of mobile devices because computation intensive tasks run quicker on a powerful cloud server. In this study, a resource intensive application and a non-resource intensive application were created using both Cuckoo and Aiolos frameworks. Two key factors were monitored: execution time, and percentage of remaining battery power. For the resource intensive application, we compared the performance of both frameworks when a phone is connected to the Internet through 4G or Wi-Fi, and also when the phone is offline or offloading servers are not available. For the non-resource intensive application, we compared the performance on the cloud versus local. This study also determines the preferred environment for each framework when running each kind of computation task in order to conclude which framework is more efficient.

To compare execution time of the two frameworks, the application ran 50 times on each framework, and each time we captured the time required for the application to finish. As part of this research, a framework was occasionally forced to use a certain environment by making the other environment unavailable. For example, Aiolos prefers to offload a heavy computation task. However, if the offloading server is not available, it forces Aiolos to run the task locally.

To compare the two frameworks in terms of power consumption, the application ran 50 times immediately after the battery was fully charged, then the remaining percentage of battery power was captured. The consumed battery percentage was calculated as follows:
Power Consumed = Initial Power – Remaining Power

3.1 Breadth First Search Algorithm

Breadth first search is a search algorithm where the root node is expanded first and then all successors of the root node are expanded next, then their successors, and so on. Every node is expanded at each depth before moving to the next level. The breadth first search algorithm can be costly in terms of space and time taken to find the target node. If each node generates b more nodes, then to get to a node at depth d, the algorithm must generate $O(b^d)$ nodes [Russell10]. The breadth first search algorithm stores every expanded node, so in a worst-case scenario the space complexity is O(b^d) [Russell10]. In this experiment, breadth first search is used to find the shortest path in terms of number of edges from a given source vertex to every other vertex in an undirected graph.

3.1.1 Light Computation Breadth First Search Task

A “light” task was employed to generate a graph of 250 vertices and 1,273 edges, then find a path from a given source node to every other node in the graph. The source node chosen for this experiment was node number 100.

3.1.2 Heavy Computation Breadth First Search Task

A “heavy” task was employed to generate a graph of 1,000,000 vertices and 7,586,063 edges, then find a path from a given source node to every other node in the graph. The source node chosen for this experiment is node number 200.

3.2 Setting up the Android Development Environment

The Android development environment is composed of six different software components:

Eclipse Kepler 4.3.2 Edition

- Android SDK (Software Development Kit)
- ADT Plugin for Eclipse (Android Development Tool)

- Git (Open Source Version Control)
- Apache Ant (build tool)
- BNDTools (OSGi plugin)

The test development framework uses Eclipse, the Android SDK, and the ADT plug-in. The Android SDK provides API libraries and development tools necessary to build, test and debug apps for Android [Android18A]. The ADT plug-in for Eclipse facilitates setting up Android projects, creating an application UI, adding packages based on the Android Framework API, and providing an emulator to test the Android apps locally in the development machine.

Git is an open source version control that is used to export Cuckoo and Aiolos frameworks locally. Apache Ant is a Java-based build tool from Apache Foundation. Ant files are .xml files that enable developers to compile a set of projects at the same time [Apache12] because OSGi projects contain at least four projects. Finally, Bndtools plugin allows developers to create OSGi applications [BndTools12].

3.3 Creating Virtual Machines on the Amazon EC2 Cloud Service

By using the Amazon Web Services web-based console, it is possible to configure and create a virtual machine on the EC2 platform. Additionally, the JRE 6 or 7 must be installed on each Cuckoo virtual machine in order for the Cuckoo server to run.

3.4 Software Specifications

- Eclipse Kepler 4.3.2 Edition as development framework with Java Runtime Environment JRE 7.
- The Android Software Development Kit (SDK).
- Android Development Tools (ADT).
- Ant build tool to build jar files.
- BndTools to create OSGi components.
- SSH software to connect to Amazon VM.
- KingSoft Battery Doctor to calculate the percentage of remaining power.

3.5 Hardware Specifications

- LG Leon as a mobile client described in Table 1.
- One VM configuration, M3 medium instance, on Amazon cloud provider described in Table 2.
- Wi-Fi and 4G characteristics as described in Table 3.

LG Leon	
Operating System	Android 4.0.1 (Lollipop)
Memory	1 GB
Storage	8 GB
Battery	1820 mAh

Table 1: Mobile client specifications.

Amazon EC2 – M3 Medium Instance	
Number of cores	1 Core
Processor	Intel Xeon E5-2670
Compute Unit	3 C.U
Operative System	Ubuntu Server 14.04 LTS
Memory	3.75 GiB
Internal Storage	8 GiB

Table 2: Amazon EC2 Specifications.

	Wi-Fi	4G HSPA
Service Provider	Comcast	T-Mobile
Download	50 Mbps	10 Mbps
Upload	10 Mbps	1 Mbps

Table 3: Comparison between Wi-Fi and 4G Internet service.

Chapter 4

RESULTS AND DISCUSSION

This study evaluates and compares the performance and power consumption of processing a light computation task and a heavy computation task, both locally on the mobile device and remotely on Amazon EC2 cloud.

The phone application used in this experiment had two parts. The first part was to read a file that contained a set of node pairs locally. The second part, to build a graph, was performed either locally or on the cloud; then using the breadth first search algorithm, the objective was to find a path from a given source node to every other node in the graph. A light task involves building a graph of 250 vertices and 1,273 edges and finding a path from a given node to every other node. A heavy task involves building a graph of 1,000,000 vertices and 7,586,063 edges and finding a path from a given node to every other node. Each task ran 50 times using each offloading framework (Aiolos and Cuckoo). The execution time was recorded when the application finished running. The power consumption was recorded after running the whole application 50 times.

The strategy used in Aiolos favors offloading, which means that an Aiolos application will offload whenever possible. A Cuckoo application is built with a speed/energy strategy which means that the framework will decide at runtime where to run the dynamic part of the application based on many factors. Unlike the conclusions of Tim et al. in

[Verbelen12B], Aiolos does not have an engine that determines at runtime where to run the dynamic part of the phone application. It comes with three strategies. The first strategy prefers offloading, while the second strategy prefers local execution. The third strategy prefers a randomly chosen environment. In addition, Aiolos allows developers to implement their own decision-making strategy, if desired.

4.1 Cuckoo vs Aiolos

4.1.1 Light Computation Task

As shown in Figures 11 and 12, Aiolos tends to faster than Cuckoo when running the application on the cloud for both 4G and Wi-Fi. However, Cuckoo is faster when the application runs locally, as shown in Figure 13. Cuckoo is always slow the first time the application is executed because it needs to send the remote JAR file to the server and get it installed. Both frameworks perform better using a Wi-Fi link than 4G.

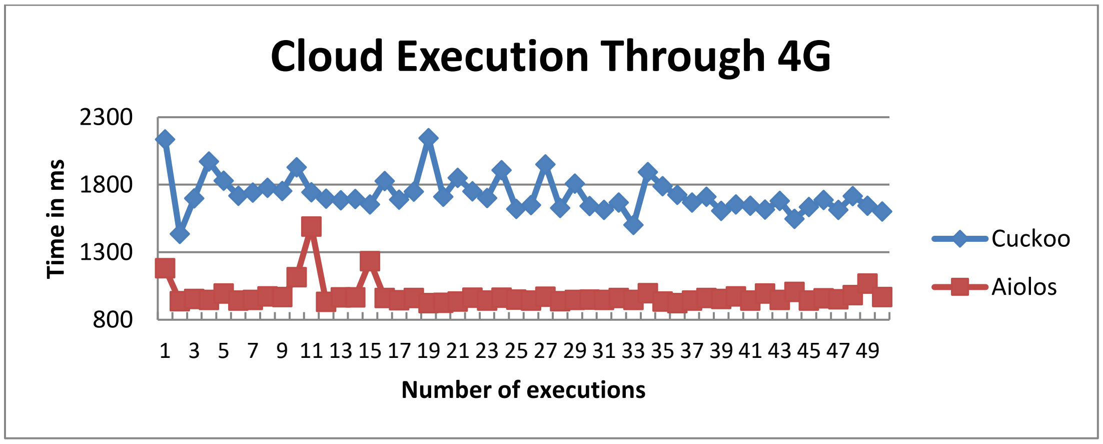

Figure 11: Aiolos vs Cuckoo: Amazon EC2 Execution Time using 4G for Light Computation Task.

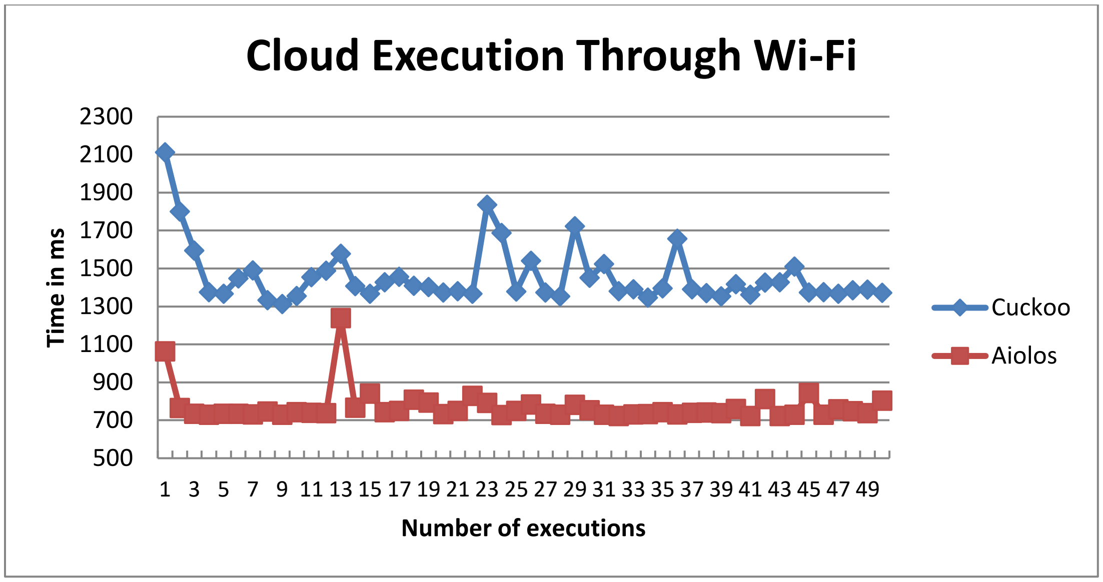

Figure 12: Aiolos vs Cuckoo: Amazon EC2 Execution Time using Wi-Fi for Light Computation Task.

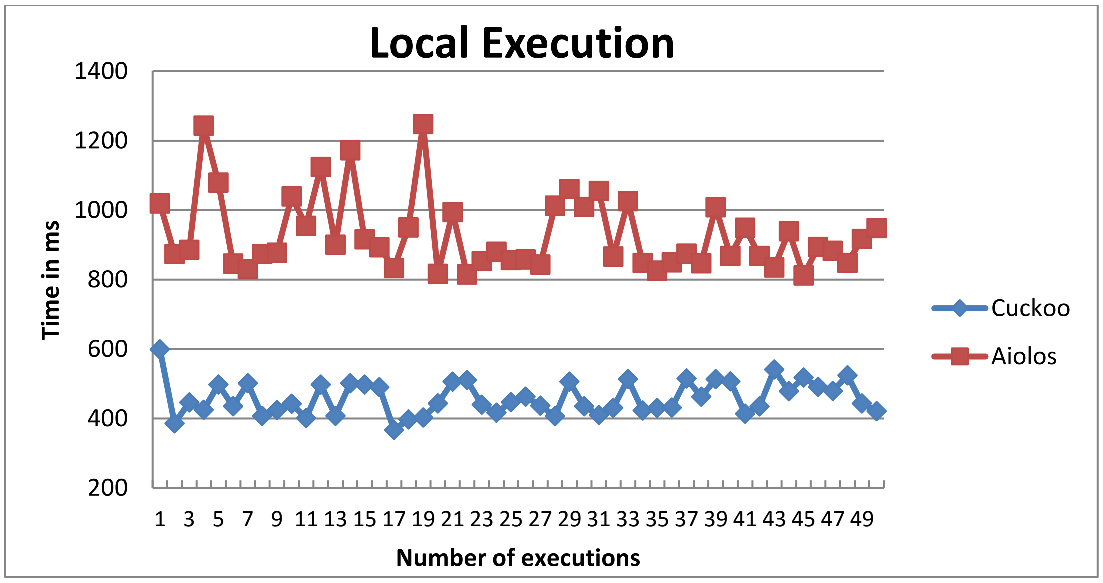

Figure 13: Aiolos vs Cuckoo: Local Execution Time for Light Computation Task.

There is no difference in battery consumption as both frameworks yield the same results (Figures 14, 15 and 16).

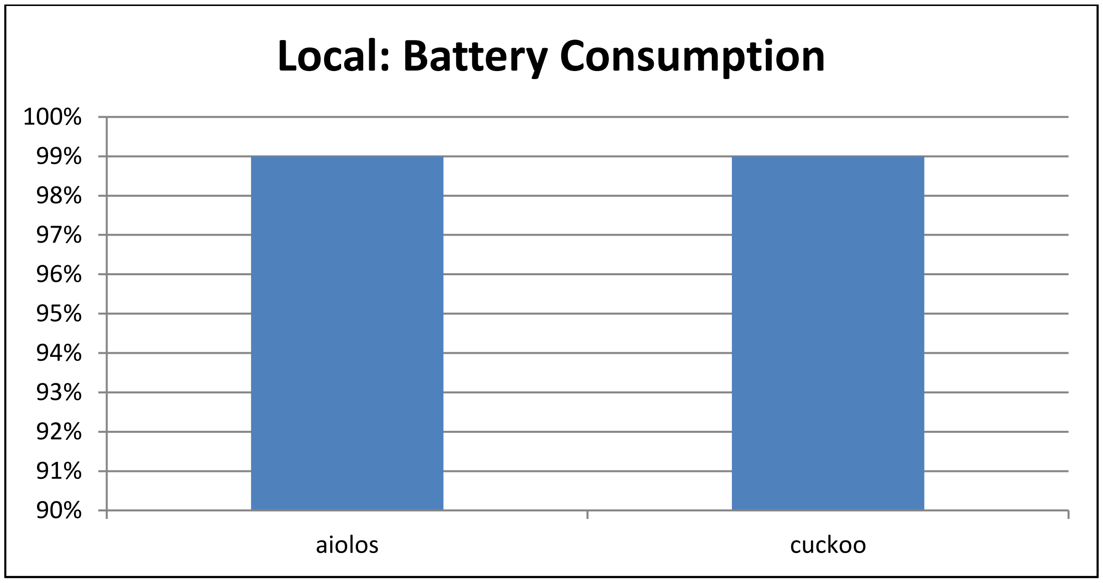

Figure 14: Aiolos vs Cuckoo: Battery Consumption when Light Computation Task is run locally.

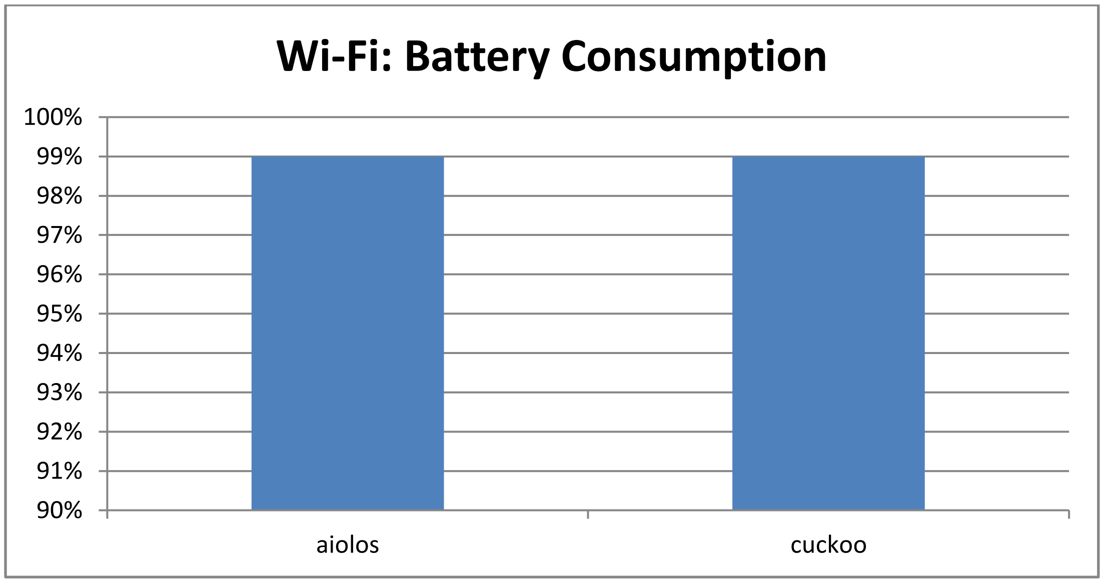

Figure 15: Aiolos vs Cuckoo: Battery Consumption when Offloading Light Computation Task using Wi-Fi.

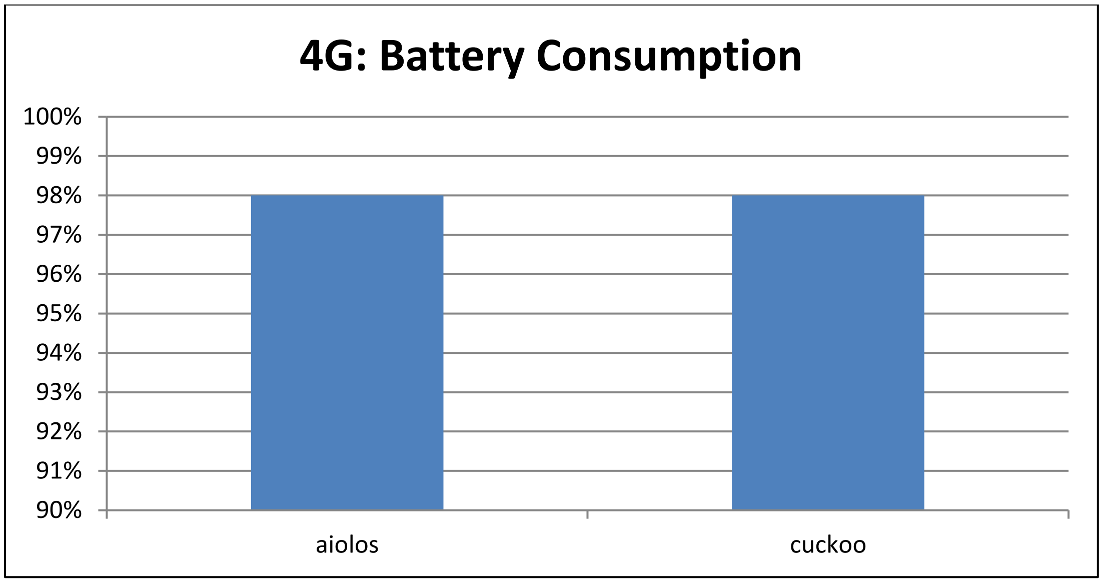

Figure 16: Aiolos vs Cuckoo: Battery Consumption when Offloading Light Computation Task using 4G.

4.1.2 Heavy Computation Task

As shown in Figures 17 and 18, Aiolos tends to be faster than Cuckoo in handling a computation intensive task on the cloud using either both 4G and Wi-Fi. In contrast, there is no difference between the two frameworks when the whole application runs locally, as shown in Figure 19. Although the process must send a large set of data through either a 4G or Wi-Fi link to run the search algorithm on the cloud, due to limited mobile resources it is a lot faster than running the heavy computation task locally using either framework. Again, Cuckoo is always slow the first time the application is executed because it needs to send the remote JAR file to the server and get it installed. In addition, it is worth noting that the performance of either framework is slower when using 4G instead of a Wi-Fi link simply because Wi-Fi is faster than 4G.

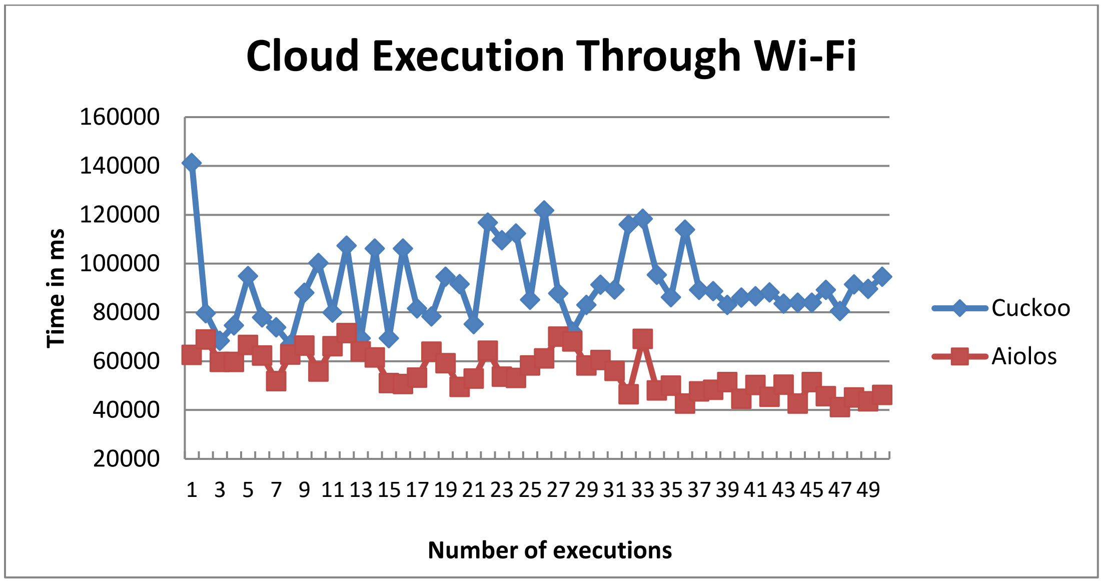

Figure 17: Aiolos vs Cuckoo: Amazon EC2 Execution Time using Wi-Fi for Heavy Computation Task.

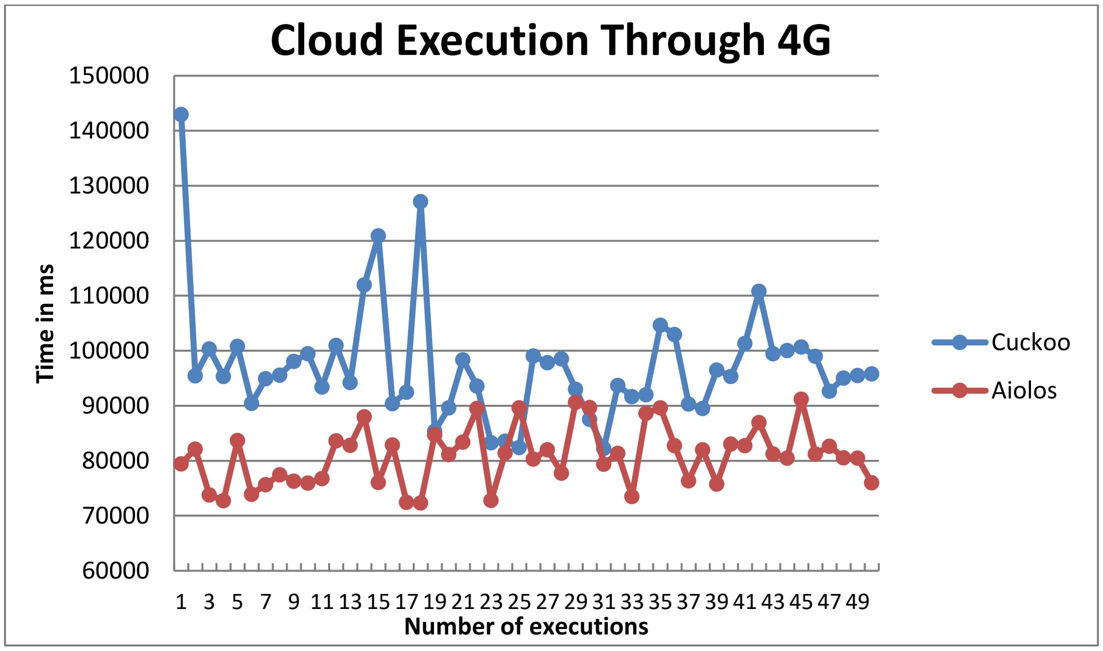

Figure 18: Aiolos vs Cuckoo: Amazon EC2 Execution Time using 4G for Light Computation Task.

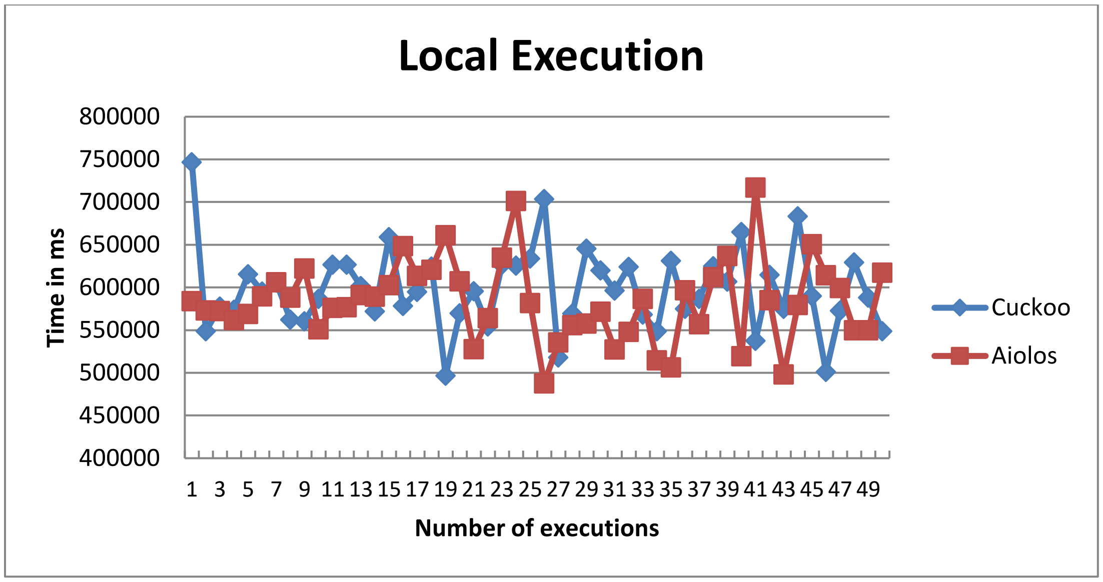

Figure 19: Aiolos vs Cuckoo: Local Execution Time for Heavy Computation Task.

Regarding battery consumption, there is not a significant difference when running the search algorithm on the cloud using a 4G or Wi-Fi link. Using Wi-Fi, Aiolos consumes 4% and Cuckoo consumes 5%, as seen in Figure 20. Using 4G, Aiolos consumes 5% while Cuckoo consumes 7%, as seen in Figure 21. It is a result of Aiolos being faster than Cuckoo in execution time. In addition, running the whole process locally consumes 45% of battery power for both frameworks, as shown in Figure 22.

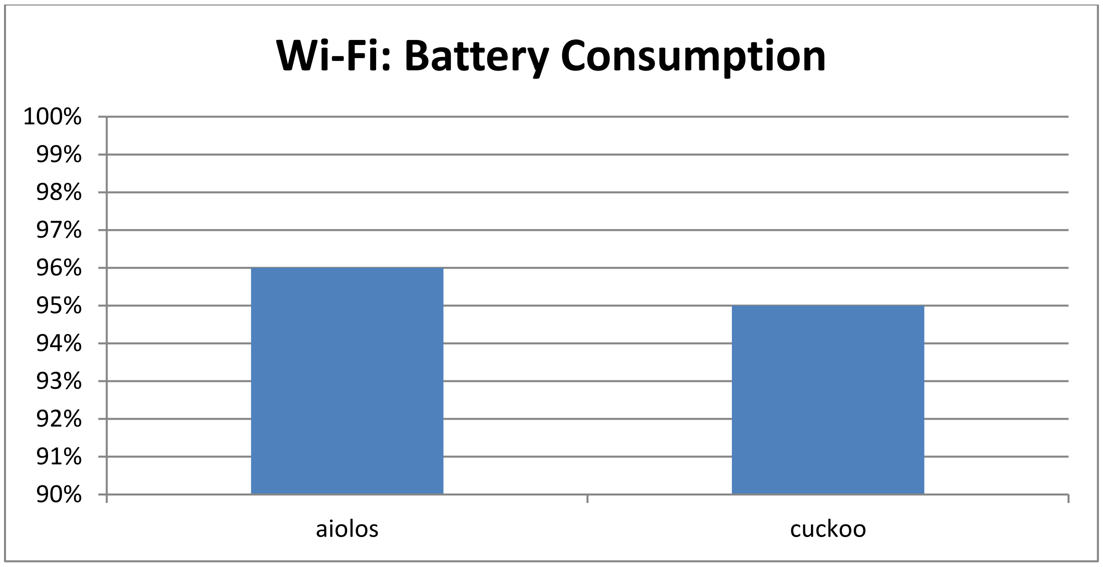

Figure 20: Aiolos vs Cuckoo: Battery Consumption when Offloading a Heavy Computation Task using Wi-Fi.

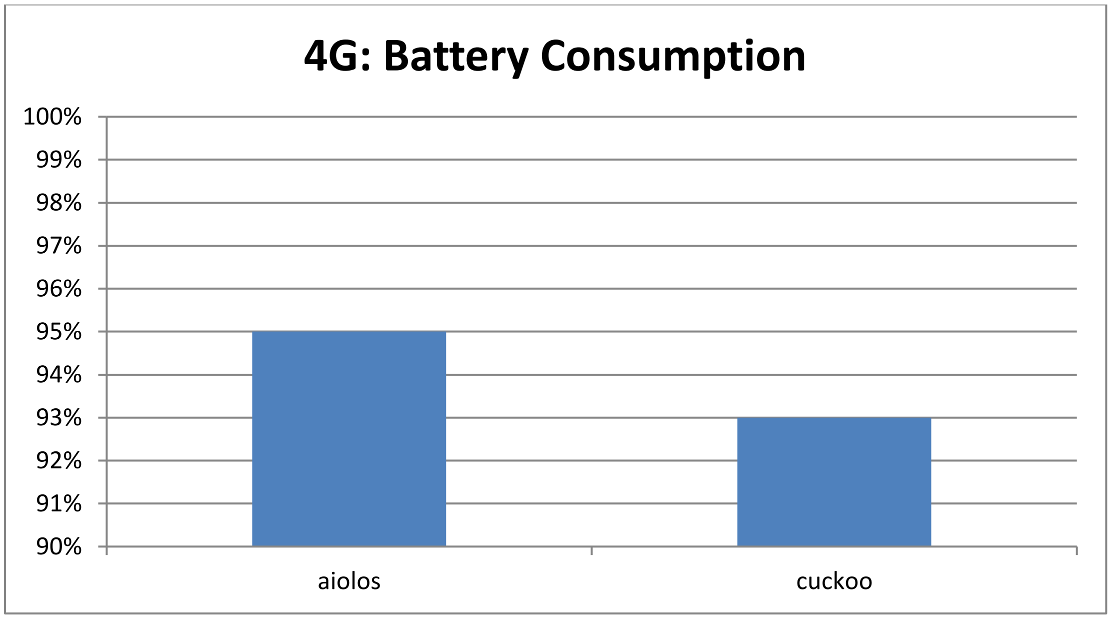

Figure 21: Aiolos vs Cuckoo: Battery Consumption when Offloading Heavy Computation Task using 4G.

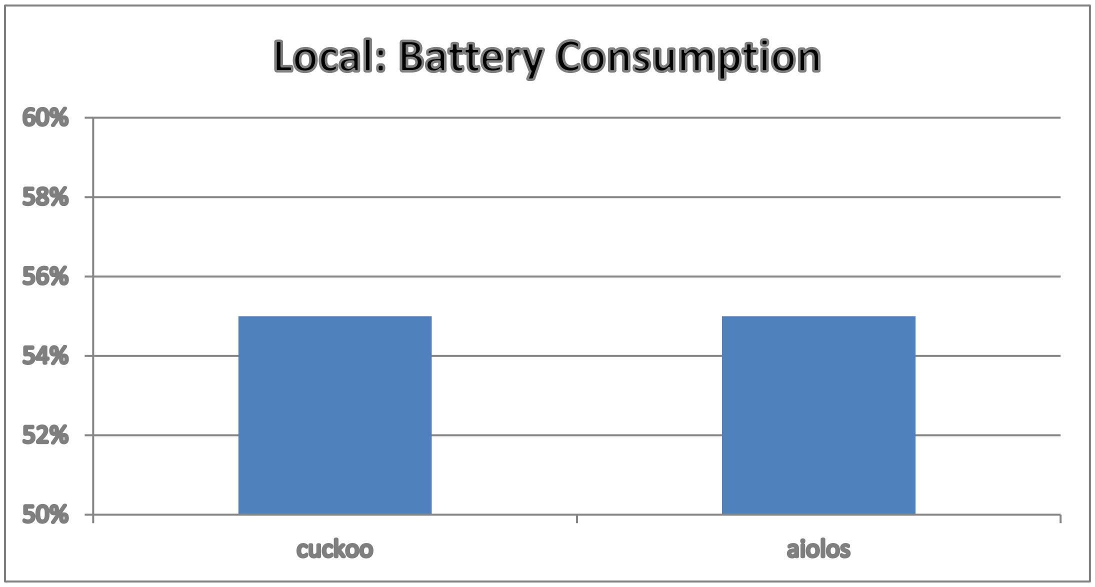

Figure 22: Aiolos vs Cuckoo: Battery Consumption when Heavy Computation Task is run locally.

4.1.3 Performance Comparison for Different File Sizes

Figures 23 and 24 illustrate that Aiolos tends to perform faster than Cuckoo as file size gets bigger using either Wi-Fi or 4G. This experiment did not include the data from the first time we ran a task with each file size using Cuckoo, since it takes longer to install a new service on the EC2 machine. Aiolos performs the same as Cuckoo when running locally, as shown in Figure 25.

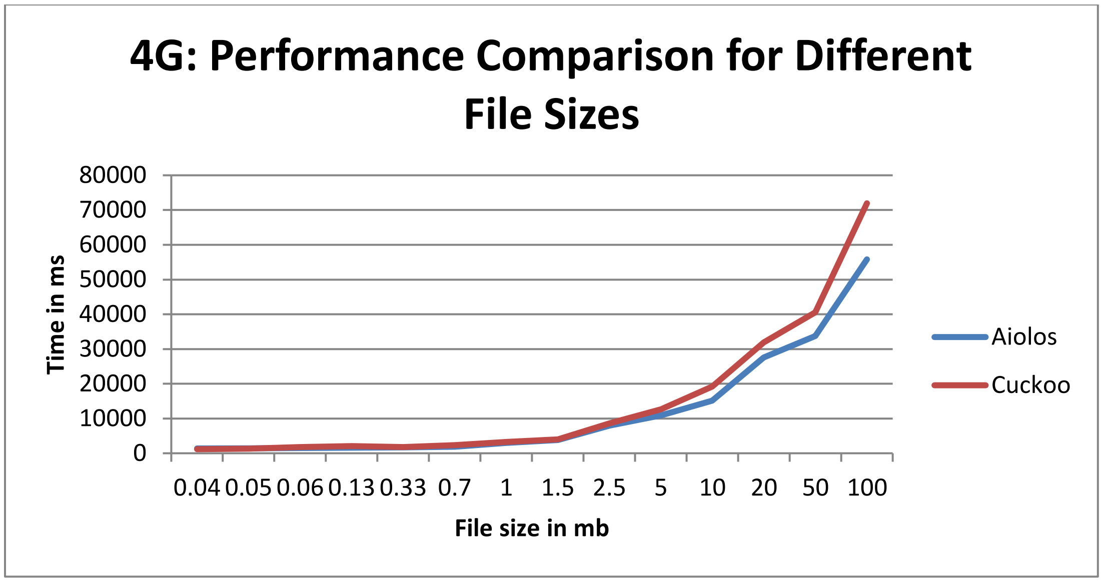

Figure 23: Aiolos vs Cuckoo: Amazon EC2 Execution Time for Different File Sizes using 4G.

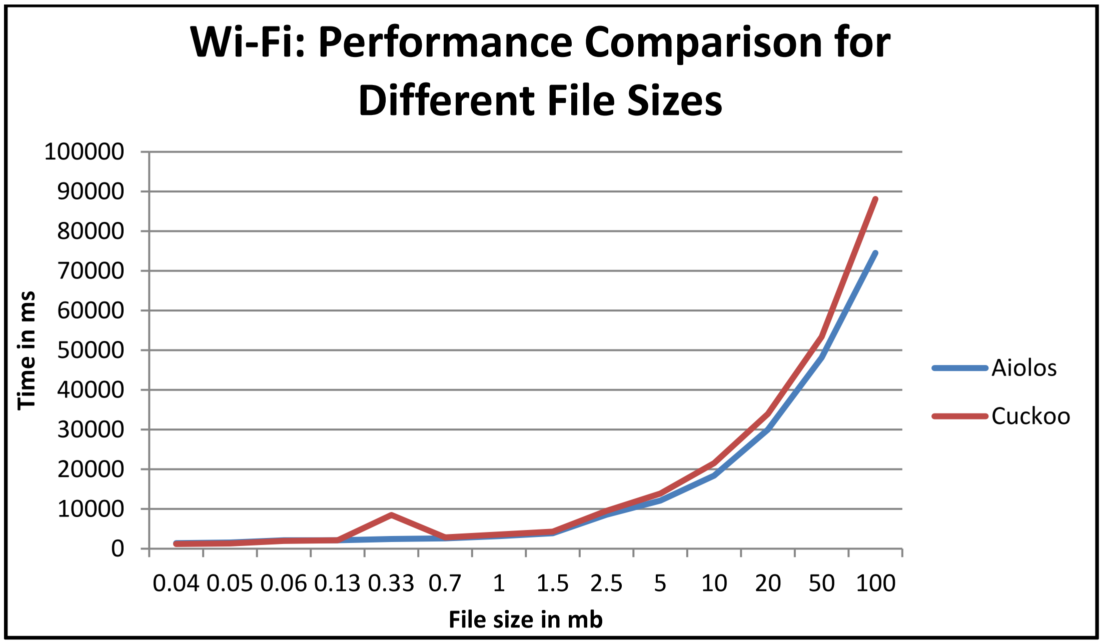

Figure 24: Aiolos vs Cuckoo: Amazon EC2 Execution Time for Different File Sizes using Wi-Fi.

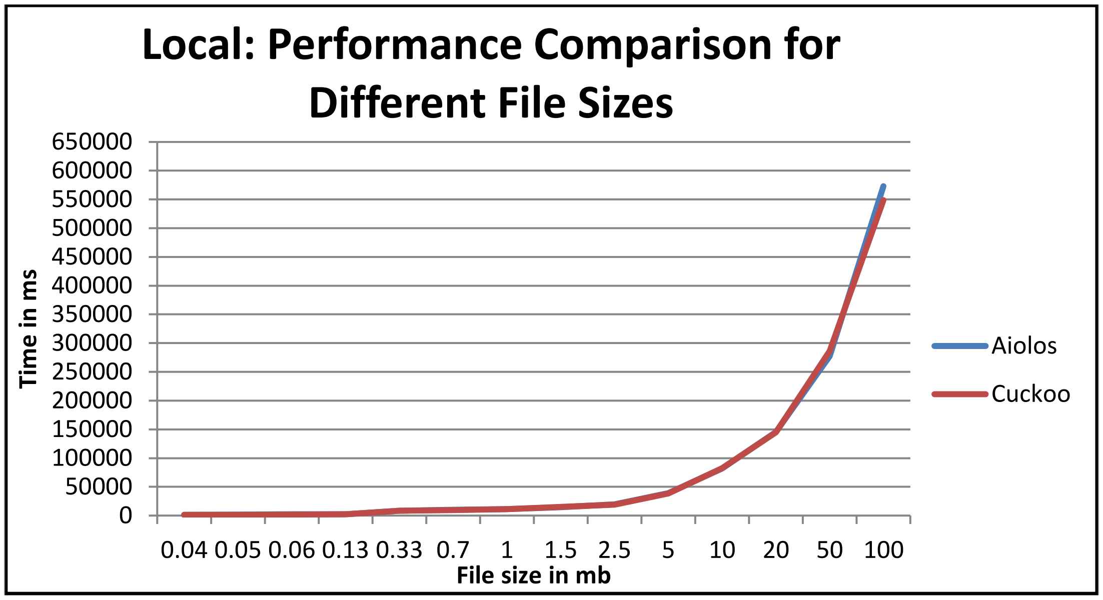

Figure 25: Aiolos vs Cuckoo – Amazon EC2 Execution Time for Different File Sizes using Local.

4.1.4 Break-Even Points

As shown in Figures 26 and 27, there is a break-even point in execution time where Cuckoo and Aiolos perform equally well. When using 4G, it is between file size 0.06 MB and 0.13 MB. When using Wi-Fi, it is between file size 0.05 MB and 0.06 MB.

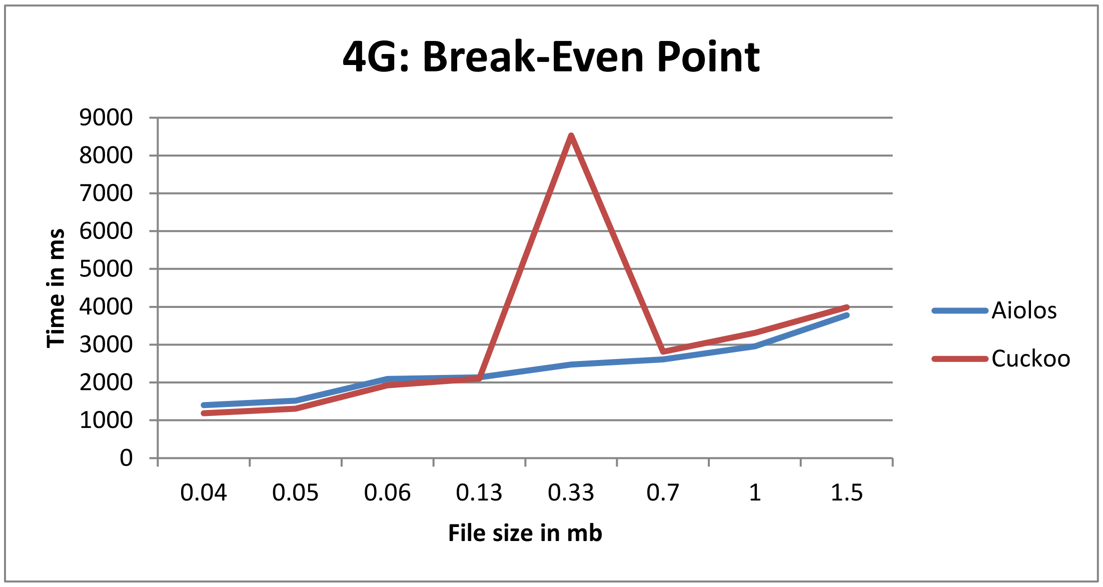

Figure 26: Aiolos vs Cuckoo: Break-Even Point for Amazon EC2 Execution Time for Different File Sizes using 4G.

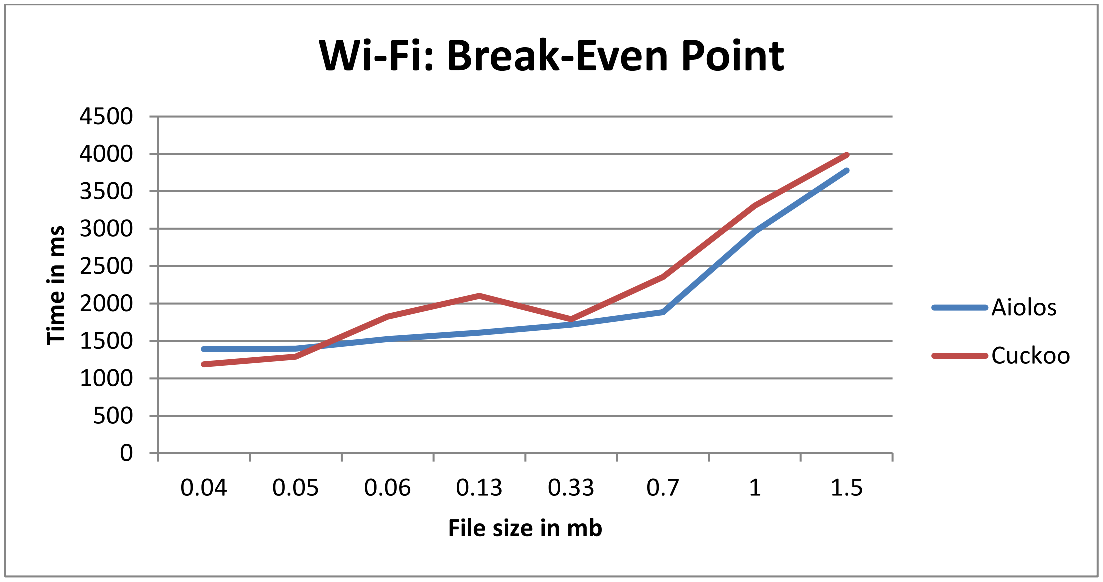

Figure 27: Aiolos vs Cuckoo: Break-Even Point for Amazon EC2 Execution Time for Different File Sizes using 4G.

4.2 Aiolos: Local vs Wi-Fi vs 4G

4.2.1 Light Computation Task

As illustrated in Figure 28, Aiolos tends to be faster using Wi-Fi and slower locally or using a 4G link. The Aiolos framework prefers to run a light computation task on the cloud whenever possible.

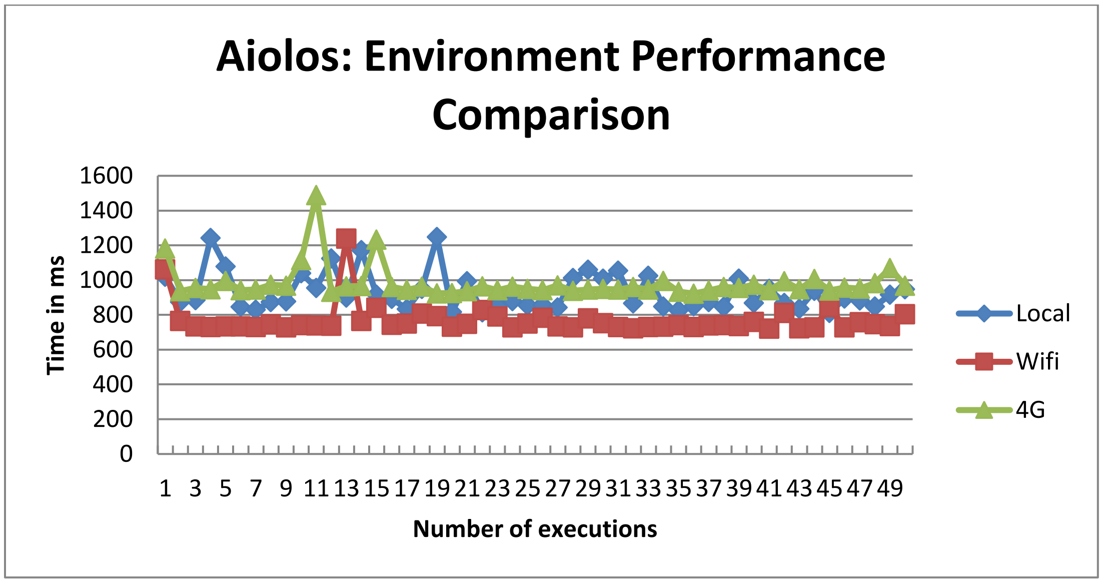

Figure 28: Local vs Wi-Fi vs 4G: Aiolos Execution Time for Light Computation Task.

Battery consumption is almost the same across different environments. As shown in Figures 29, 4G consumes 1% more of battery power compared to local and Wi-Fi.

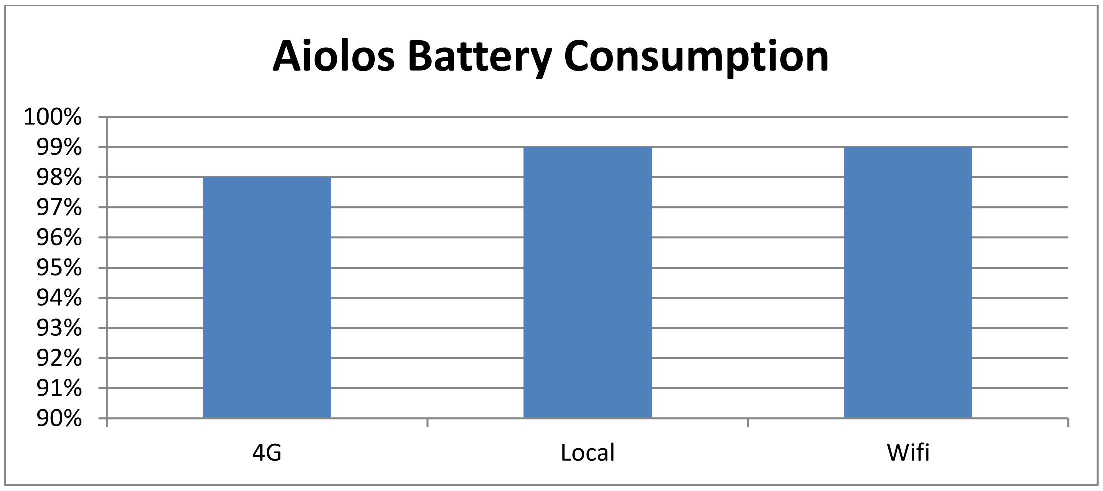

Figure 29: Local vs Wi-Fi vs 4G: Aiolos Battery Consumption for Light Computation Task.

4.2.2 Heavy Computation Task

As shown in Figure 30, Aiolos performance is slower locally compared to either 4G or Wi-Fi due to limited resources in the mobile device. The Aiolos framework prefers to run a heavy task on the cloud whenever possible.

Running the whole process locally drains the battery power compared to the cloud, as seen in Figure 31. Aiolos consumes 45% locally, 4% when using a Wi-Fi link, and 5% when using a 4G link. In this case, it is beneficial to send a large set of data through the Internet and perform the search on the cloud.

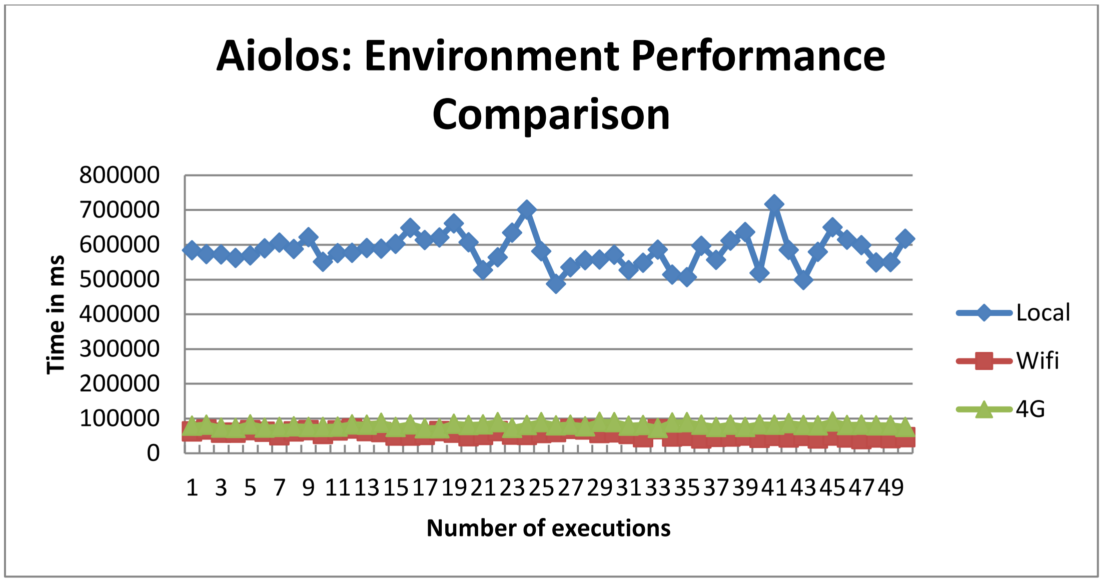

Figure 30: Local vs Wi-Fi vs 4G: Aiolos Execution Time for Heavy Computation Task.

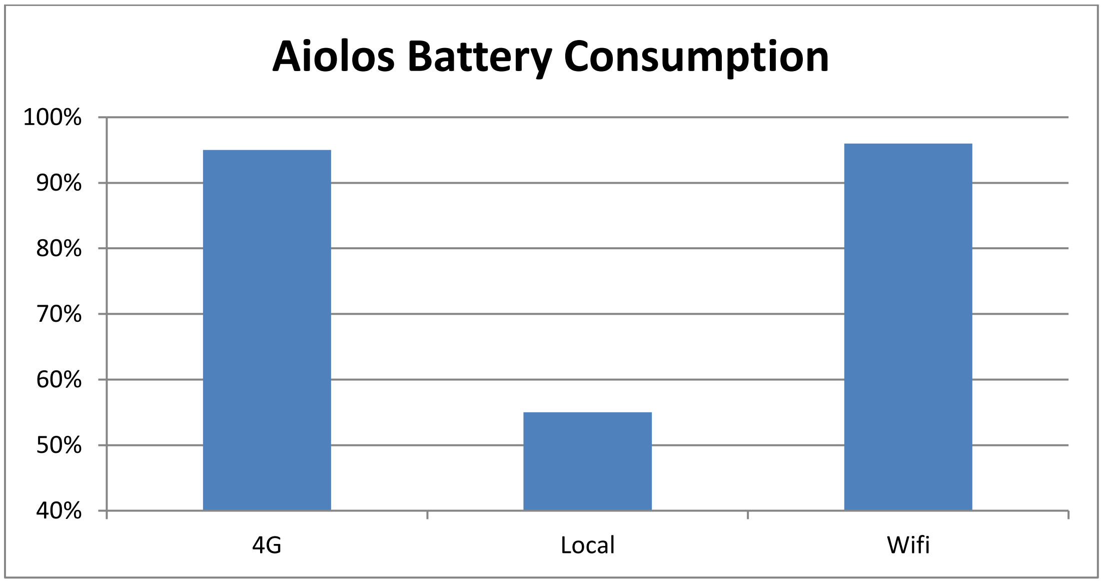

Figure 31: Local vs Wi-Fi vs 4G: Aiolos Battery Consumption for Heavy Computation Task.

4.3 Cuckoo: Local vs Wi-Fi vs 4G

4.3.1 Light Computation Task

As shown in Figure 32, Cuckoo seems to be faster running the search algorithm locally than on the cloud. 4G tends to be the slowest means of communication if the framework decides to run the algorithm on the cloud. Based on this study, the Cuckoo framework prefers to run a light computation task locally.

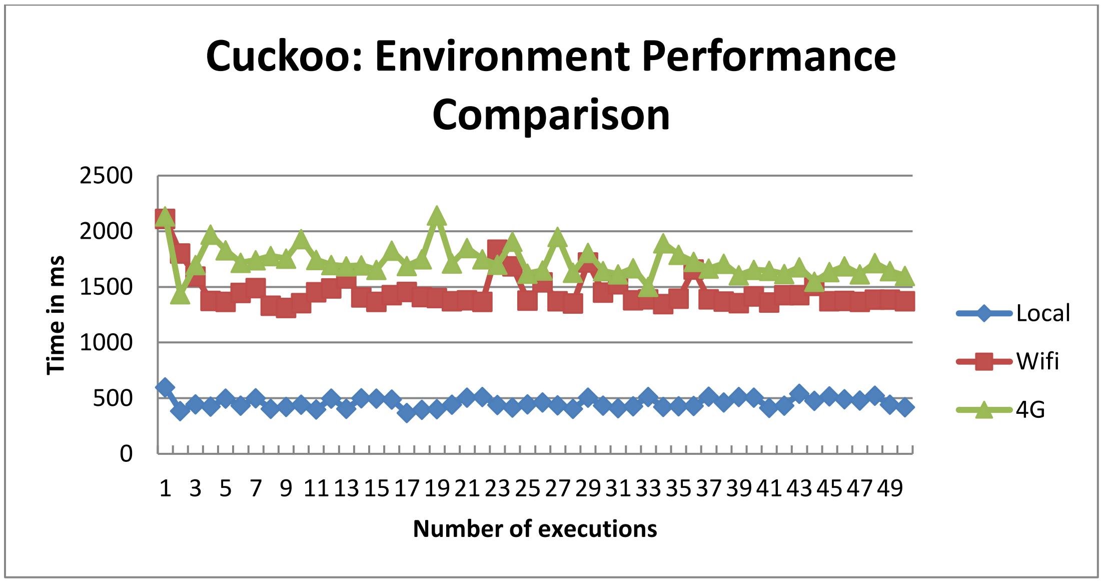

Figure 32: Local vs Wi-Fi vs 4G: Cuckoo Execution Time for Light Computation Task.

As shown in Figure 33, there is not much difference when it comes to power consumption with a slight advantage to local and Wi-Fi link. Cuckoo consumes 2% when using 4G, and 1% locally and when using Wi-Fi.

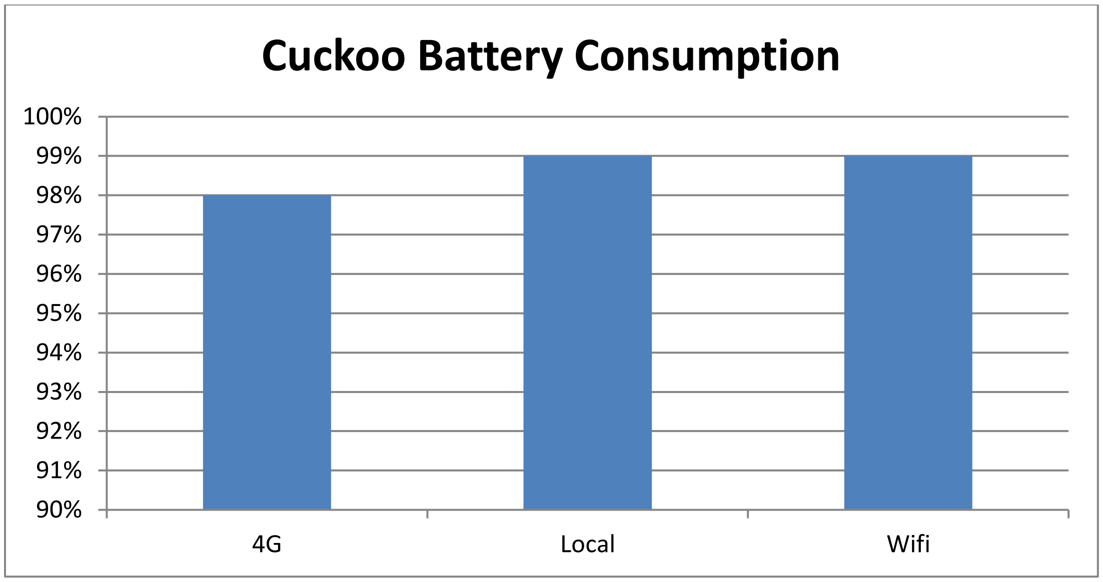

Figure 33: Local vs Wi-Fi vs 4G: Cuckoo Battery Consumption for Light Computation Task.

4.3.2 Heavy Computation Task

Just like Aiolos, Cuckoo is slower when running a heavy computation task locally, due to limited resources in the mobile device, as shown in Figure 34. Running the task on the cloud improves the performance of the application. However, using Wi-Fi as a means of communication with the cloud makes the application even faster than using a 4G link. In this case, Cuckoo prefers to run a heavy computation task on the cloud.

Figure 35 illustrates that running the whole application locally drains the battery by 45%. In this case, offloading a heavy computation task using either Wi-Fi or 4G saves a lot of power. Cuckoo consumes 5% when using Wi-Fi and 7% when using 4G.

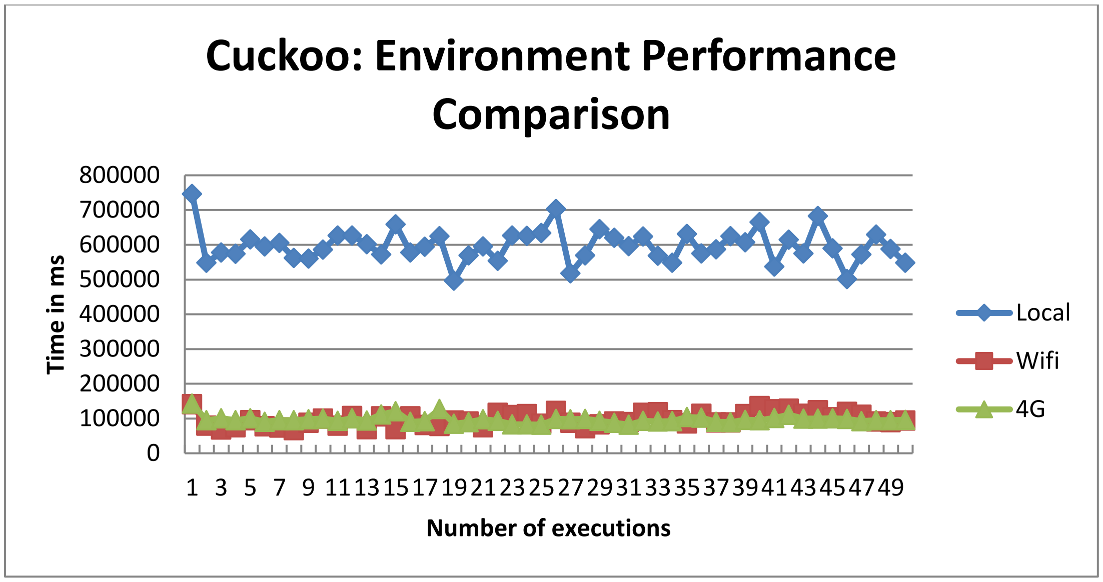

Figure 34: Local vs Wi-Fi vs 4G: Cuckoo Execution Time for Heavy Computation Task.

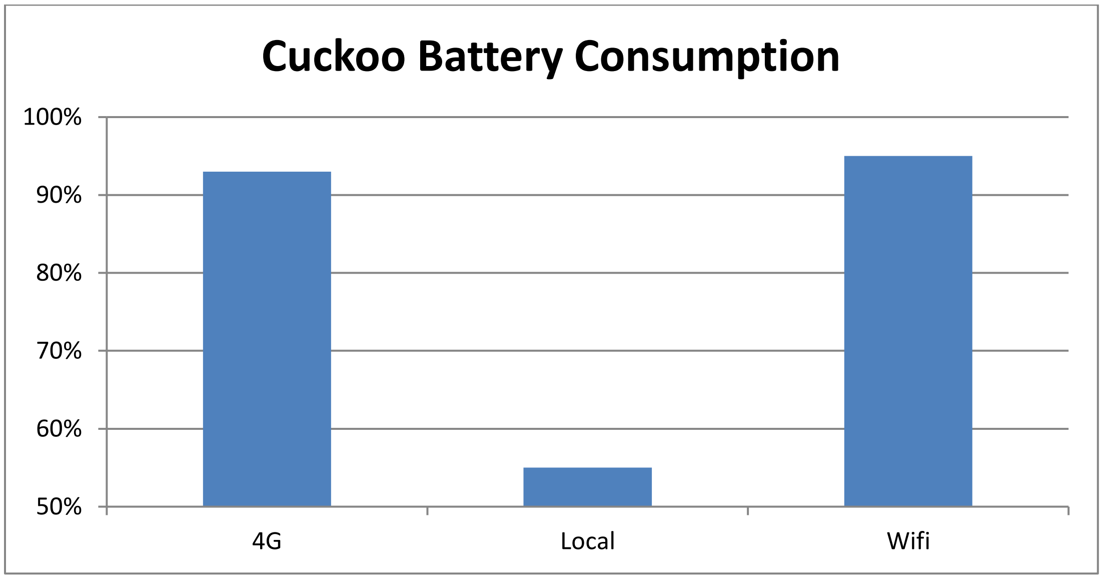

Figure 35: Local vs Wi-Fi vs 4G: Cuckoo Battery Consumption for Heavy Computation Task.

4.3.3 Service Jar Installation

The purpose of this experiment is to confirm that Cuckoo framework installs Jar file only the first time it remotely runs a particular task. To confirm that, a phone application is built using Cuckoo to remotely run the same light computation task every day for six days. Figure 36 shows that a light computation task usually takes longer the first time it runs. The reason for the overhead is that Cuckoo has to send a whole JAR file to the server in order to install and initialize a service. Once the JAR file is installed and is initialized successfully, the client can invoke the service directly from an EC2 machine.

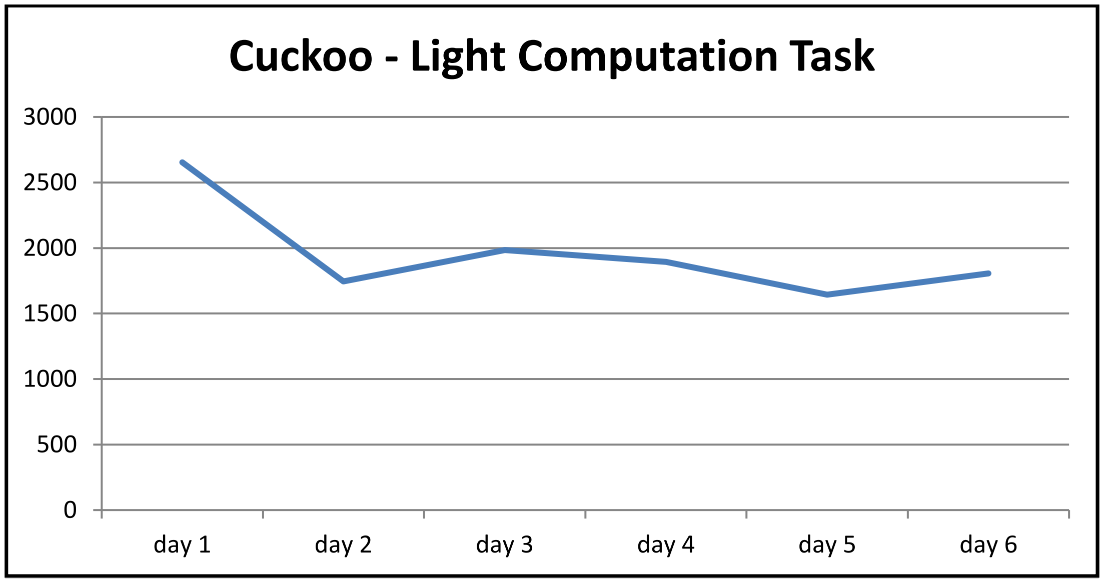

Figure 36: Cuckoo Performance for a Light Computation Task over the Period of 6 Days

Chapter 5

CONCLUSIONS

The following is a discussion of the results presented in Chapter 4 where two dynamic offloading frameworks were used to handle both a light computation task and a heavy computation task, as well as to determine the strength of each framework. Amazon EC2 was chosen to host Aiolos and Cuckoo servers.

5.1 Task Studies

5.1.1 Light Computation Task

The goal of this experiment was to determine which framework, Cuckoo or Aiolos, handles a light computation task better than the other. The task was to build a graph of 250 vertices and 1,273 edges, then employ the breadth first search algorithm to find a path from a given source node (node 100) to every other node. The research included running the task locally as well as on an EC2 instance.

Cuckoo tends to be faster locally while Aiolos performs faster than Cuckoo when offloading the light task to the cloud. The Aiolos strategy prefers offloading, so it tries to offload a task first, but if Aiolos fails to establish a connection to the server then it runs the task locally. As a result, Aiolos is slower when running a light task locally because it wastes time trying to offload. Power consumption tends to be the same using either framework.

Cuckoo tends to be faster locally compared to on the cloud. In addition, if Cuckoo offloads a light task then Wi-Fi is the fastest link. Cuckoo oracle, and specifically the Cuckoo decision maker component, is the main reason behind the slower performance of Cuckoo when it offloads a light task. Oracle makes a decision based on an algorithm that includes bandwidth estimation, execution time estimation, round trip time estimation, and power estimation. Based on the test results, this process introduces some overhead to the execution time of a light task. In a real-world application, where Cuckoo is not forced to utilize a particular environment, Cuckoo oracle runs a light task locally, which is the environment with the fastest execution time. Regarding battery consumption, 4G consumes 1% more power than other environments because of the extra effort needed to transmit all data to the EC2 instance.

Aiolos seems to perform faster when offloading a task through Wi-Fi. Unlike Cuckoo, Aiolos prefers offloading whenever it is possible. As a result, it does not waste time comparing different environments before running a light task. However, Aiolos depends on the method of communication with the offloading server. This research shows that Aiolos is slower when offloading through 4G. In real world application, where Aiolos is not forced to utilize a particular environment, it runs a light task on the cloud. To sum up, in a real-world application Cuckoo performs faster because its local average execution time is 458ms, while Aiolos utilizing Wi-Fi averages 769ms.

5.1.2 Heavy Computation Task

The goal of this experiment was to find which framework, Cuckoo or Aiolos, handles a heavy computation task more efficiently. The task is to build a graph of 1000000 vertices and 7586063 edges, then use the breadth first search algorithm to find a path from a given source node (node 200) to every other node. The experiment includes running the task locally as well as on an EC2 instance.

Due to limited resources in the mobile device, a heavy computation task running locally takes more time to be completed regardless of which framework is used. Conversely, Aiolos performs better than Cuckoo when offloading a heavy computation task to an EC2 instance. This can be due to either of two factors:

- The Cuckoo algorithm uses more time to decide where to run a task, whereas Aiolos just offloads a task whenever it is possible.
- R-OSGi is faster than Ibis middleware

Cuckoo tends to perform faster when offloading through Wi-Fi. In a real world application, where Cuckoo is not forced to utilize a particular environment, it runs a heavy task on the cloud using either Wi-Fi or 4G.

Aiolos seems to be faster when offloading through Wi-Fi. In a real world application, where Aiolos is not forced to utilize a particular environment, it runs a heavy task on the cloud using either Wi-Fi or 4G.

Regarding battery consumption, offloading using either framework saves 23% more power than the local environment. Aiolos consumes less battery power than Cuckoo when offloading a heavy computation task. The increased power consumption is due to Cuckoo execution time which is longer than Aiolos execution time. In conclusion, Aiolos outperforms Cuckoo in handling a heavy computation task.

5.1.3 Different File Sizes

Cuckoo tends to perform faster when file size is less than 0.05 MB when communicating with the EC2 machine through Wi-Fi. In addition, it performs faster when file size is less than 0.13 MB when communicating with the EC2 machine through 4G. As file size gets bigger Aiolos seems to be faster using either 4G or Wi-Fi. When communicating to an EC2 machine through Wi-Fi, Cuckoo performs much slower when file size is 0.33 MB because it decides to run it locally instead of on the EC2 machine.
The break-even points between the two frameworks are the following:

- Between 0.06 MB and 0.13 MB when using 4G.
- Between 0.06 MB and 0.13 MB when using Wi-Fi.

5.2 Finding of the Development Effort

Creating an application using the Cuckoo framework is straightforward. As soon as a developer makes an Android project as an off-loadable project, the Cuckoo framework adds a folder called remote to the project. Then, a developer can add the off-loadable code inside that folder. The main issue with the Cuckoo compiler is that it does not provide developers enough details about compile time errors or where the errors are. Another issue with Cuckoo is the lack of documentation and online support.

Creating an application using the Aiolos framework requires developers to have OSGi experience. An Aiolos application contains four projects that are API, API Implementation, Servlet, and android project. In addition, Aiolos requires a manual update to the Android ".bndrun" file, which is the OSGi environment configuration file. The set up process is complex, especially for developers who do not have much OSGi experience. Additionally, the Aiolos framework only works in a Linux operating system. Finally, Aiolos has insufficient documentation and no online support. Based on experience with both frameworks, Cuckoo applications are easier to build and to set up.

5.3 Future Research

This study is limited to comparing mobile local processing of both light and heavy computation tasks to Amazon EC2 using Wi-Fi and 4G communication links. An extension to this study on mobile offloading could include other Android mobile devices, such as tablets and smart watches, and other cloud providers such as Google Cloud Engine, IBM SmartCloud or others. Additionally, cloudlets could be included in the research to determine their influence on the experiment. The main characteristic of cloudlets is low latency. 4G LTE could also be included to determine its impact on cloud offloading.

Since this study covers two offloading frameworks, it could serve as a reference for future studies involving the development of a new offloading framework that takes advantage of both Cuckoo and Aiolos strengths. It could also help developers and software architects choose either framework based on which one provides the best functionality.

REFERENCES

Print Publications:

[Bhatnagar13]
Bhatnagar, P., R. Jaipur, and I. Rajasthan, “Implementation of Mobile-Cloudlet-Cloud Architecture for Face Recognition in Cloud Computing using Android Mobile,” International Journal of Computer Applications Technology and Research 2, 6 (2013), pp. 671-675.

[Flinn99]
Flinn, J. and M. Satyanarayanan, “Energy Aware Adaptation for Mobile Applications,” Proceedings of the seventeenth ACM Symposium on Operating Systems and Principles 33, 5 (December, 1999), pp. 48-63.

[Hall11]
Hall, R., et al., “OSGi in Action: Creating Modular Applications in Java,” Manning Publications Co., Greenwich, CT, 2011.

[Huang13]
Huang, D., T. Xing, and H. Wu, “Mobile Cloud Computing Service Models: A User-Centric Approach,” IEEE Network 27, 5 (2013), pp. 6-11.

[Kim16]
Kim, B., et al., “Dynamic Offloading Algorithm for Drone Computation,” Proceedings of the International Conference on Research in Adaptive and Convergent Systems (October, 2016), pp. 123-124.

[Noble97]
Noble, B., et al., “Agile Application-Aware Adaptation for Mobility,” Proceedings of the sixteenth ACM Symposium on Operating Systems Principles 31, 5 (October, 1997), pp. 276-287.

[Russell10]
Russell, S. and P. Norving, “Artificial Intelligence: Modern Approach,” Prentice Hall, Upper Saddle River, NJ, 2010.

[Satyanarayanan14]
Satyanarayanan, M., “A Brief History of Cloud Offload: A Personal Journey from Odyssey Through Cyber Foraging to Cloudlets,” GetMobile: Mobile Computing and Communications 18, 4 (2014), pp. 19-23.

[Verbelen12A]
Verbelen, T., et al., "Cloudlets: Bringing the Cloud to the Mobile User," Proceedings of the third ACM Workshop on Mobile Cloud Computing and Services (2012), pp. 29-36.

[Verbelen12B]
Verbelen, T., et al., "AIOLOS: Middleware for Improving Mobile Application Performance through Cyber Foraging," Journal of Systems and Software 85, 11 (2012), pp. 2629-2639.

[Wang12]
Wang, X., et al., "Migration and Execution of JavaScript Applications between Mobile Devices and Cloud," Proceedings of the third Annual Conference on Systems, Programming, and Applications: Software for Humanity (October, 2012), pp. 83-84.

[Wang17]
Wang, W., et al., "Enabling Cross-ISA Offloading for COTS Binaries," Proceedings of the fifteen Annual International Conference on Mobile Systems, Applications, and Services (June, 2017), pp. 319-331.

Electronic Sources:

[AIOLOS15]
"AIOLOS overview", http://aiolos.intec.ugent.be/, 2015, last accessed February 21, 2018.

[Aleksandar13]
Aleksandar, G., "Deep Dive into Android IPC/Binder Framework at Android Builders Summit", https://events.static.linuxfound.org/images/stories/abs2013_gargentas.pdf, 2013, last accessed February 21, 2018.

[Android18A]
"Get the Android SDK", http://developer.android.com/sdk/index.html, last revision January 12, 2018, last accessed February 17, 2018.

[Android18B]
"Android Interface Definition Language (AIDL)", http://developer.android.com/guide/components/aidl.html, last revision February 21, 2018, last accessed February 23, 2018.

[Android18C]
"Processes and Threads", https://developer.android.com/guide/components/processes-and-threads.html, last revision February 21, 2018, last accessed February 23, 2017.

[Apache12]
"Apache Ant 1.9.6 Manual", http://ant.apache.org/manual/index.html, last revision January 21, 2012, last accessed September 20, 2015.

[BndTools12]
"Bndtools Tutorial", http://bndtools.org/tutorial.html, last revision January 21, 2012, last accessed January 15, 2016.

[Bouzefrane11]
Bouzefrane, S., D. Huang, and P. Paradinas, "An OSGI-Based Service Oriented Architecture for Android Software Development Platforms", http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.411.9733&rep=rep1&type=pdf, November 2011, last accessed February 21, 2018.

[Chun09]
Chun, B. and P. Maniatis, "Augmented Smartphone Applications through Clone Cloud Execution", https://www.usenix.org/legacy/event/hotos09/tech/full_papers/chun/chun.pdf, 2009, last accessed February 21, 2018.

[Kemp10]
Kemp, R., et al., "Cuckoo: a Computation Offloading Framework for Smartphones", http://www.asci.tudelft.nl/media/proceedings_asci_conference_2010/asci2010_submission_9.pdf, 2010, last accessed February 21, 2018.

[Kemp12]
Kemp, R., "Programming Frameworks for Distributed Smartphone Computing", http://dare.ubvu.vu.nl/bitstream/handle/1871/50612/dissertation.pdf?sequence=1, 2012, last accessed February 21, 2018.

[Kovachev11]
Kovachev, D., Y. Cao, and R. Klamma, "Mobile Cloud Computing: A Comparison of Application Models", https://arxiv.org/ftp/arxiv/papers/1107/1107.4940.pdf, 2011, last accessed February 21, 2018.

[Marinelli09]
Marinelli, E. E., "Hyrax: Cloud Computing on Mobile Devices using MapReduce", https://pdfs.semanticscholar.org/8cd2/11cc816952f036ed65a7022adba063486008.pdf, 2009, last accessed February 21, 2018.

[OSGi12]
"Architecture", https://www.osgi.org/developer/architecture/, last revision January 21, 2012, last accessed January 15, 2018.

[Paramvir12]
Paramvir, B., et al., "Advancing the State of Mobile Cloud Computing", http://www.cs.columbia.edu/~lierranli/publications/mCloud-MCS12.pdf, 2012, last accessed February 21, 2018.

[Wang14]
Wang, M., “Novel Mobile Computation Offloading Framework for Android Devices,” Washington University in St. Louis (2014), https://pdfs.semanticscholar.org/3070/0b846d78983b93673fb8fcee8388027910de.pdf, last accessed February 23, 2018.

VITA

Inan Kaddour has a Bachelor of Software Engineering from International Institute of Higher Education in Morocco in 2008, and expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida, April 2017. Dr. Sanjay Ahuja of the University of North Florida is Inan's thesis advisor. Inan has been an Application Consultant in web and windows applications at Humana, Jacksonville, Florida for 8 years. Inan's academic work has included use of Java, Microsoft Visual Studio.Net, Oracle, SQL, Javascript, PHP, Jquery, ASP.NET MVC 3, JSON, Bootstrap, RMI, and networking.

Inan likes to incorporate his computer skills in real life situations requiring the use of Web and Windows application along with Cloud Computing. His main goal is to become a respected Software professional who strives to produce excellence in every area useful for society, and to maintain ethical, high quality standards and respectful approaches to working with the vast power of computing.