UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Information Technology specialty

Kristiina Ritso

Scaling Virtualized Smartphone Images in the Cloud

Bachelor Thesis (6 EAP)

Supervisor: Satish Narayana Srirama, PhD

Co-Supervisor: Huber Flores, MSc

Author:	“.....” May 2013
Supervisor:	“.....” May 2013
Supervisor:	“.....” May 2013

Allowed to defend

Professor: .... “.....” May 2013

TARTU 2013

Table of Contents

Illustration Index

Index of Tables

1. Introduction

Smartphones are getting more and more popular each day. Along with it, user requirements rise too. Last year, in the third quarter, smartphones running on Android operating system had 75% of the market share [1]. People have the need for more powerful phones, but we have certain restrictions, for example storage, battery and memory usage. This is where we could make use of cloud computing. It would reduce cost and reliance on hardware and software, because the computations are run in the cloud [2]. Cloud computing for mobile users would bring benefits. Data and tasks would be kept in the cloud rather than in a physical device, providing on-demand access [3]. This would save storage space, decrease memory usage and save battery life. We will be discussing code offloading as this is an important part of this thesis, because we can take the tests results, that will be carried out in this thesis, into consideration when doing code offloading.

We will be setting up an Android image with Dalvik Virtual Machine (later referred to as Dalvik VM) in Amazon Elastic Compute Cloud (later EC2) instance, test its performance compared to a physical device and try to scale the instance using Amazon Elastic Load Balancing tool and Auto Scaling. We will run ten different algorithms in the cloud and on a physical android device then compare and analyze the result and later stress test a single Android x86 image with Tsung. The tests that we will be carrying out will give us an understanding how much load can one instance bear and how scalable our solution can be. The result of this thesis are useful for the Cloud Computing Laboratory, institute of Computer Science, Faculty of Mathematics and Computer Science, University of Tartu.

Chapter 2 consists of state of the art. It describes related work and the purpose of this thesis. And describes a commercial implementation.
Chapter 3 describes background information about scaling, Android, Dalvik virtual machine, Tsung, Amazon Elastic Compute Cloud and Amazon Elastic Load Balancing tool that are used to solve the problem of this thesis.
Chapter 4 describes the problem statement and research problem.

Chapter 5 describes how to set up the environment towards solving the problem and tests that were carried out, also an analyzis of the results we got.

Chapter 6 concludes the work and describes what could be done in the future.

Chapter 7 is an abstract in Estonian

2. State of The Art

2.1. Code Offloading

Taking cloud computing context as an example, code offloading means that large scale applications can load balance their computations during heavy workload, it means sending selected code to cloud for computations. Code offloading can be beneficial in applications where there are little dependencies. We have to take into account that sending code to cloud, getting back the response and merging takes time and when a part of code, that other methods depend on, is sent for execution in the cloud, it may be inexpedient. It is recommended to send large processing tasks for remote execution. Figure 1. shows the application's flow with code offloading to a cloud instance.

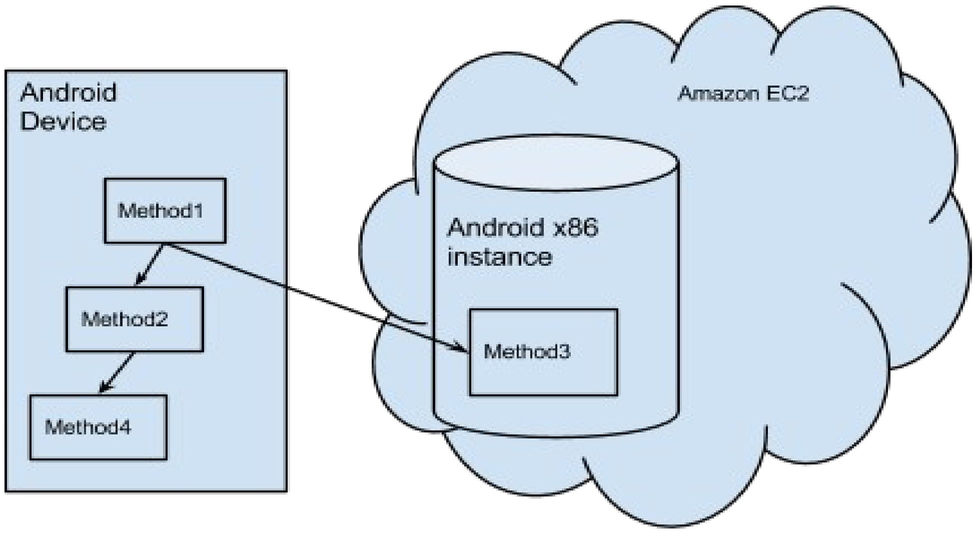

Figure 1*: Application's flow with code offloading to Cloud. Method 1, 2 and 4 are executed in device but method 3 is sent for remote execution*

To get a better understanding of code offloading architecture we will give an example. Let us assume we are doing some sort of computing and use Amazon Elastic Cloud Compute instance as a physical mobile device's clone.

1. We send a method to the cloud and invoke processing on Amazon’s instance
2. We continue tasks on device until we receive a response that remote process is finished and pushed back to device
3. We will merge the remotely executed code with device’s code.

If we cannot continue with our tasks in the device then it would take up time due to network latency and therefore it would make the process slower. Doing code offloading we have to consider that, as well as deciding what methods are worth sending for remote execution and which ones are not.

An architecture introduced by COFA [4], a client-server architecture where the server lies in a powerful machine hosted over hight bandwidth line. They configured Dalvik so that it has three modes, server, client and normal. When Dalvik is in normal mode the code will not be offloaded and it will operate like an unmodified virtual machine. In client mode, it can make requests to server and transfer off-loadable methods. They use Transmission Control Protocol for communication between the client and the server. If a thread is called for remote execution, it is passed to the server with description of its thread index and class descriptor. The thread index helps the server to find the code in the apk file located in server. Figure 2. describes its architecture.

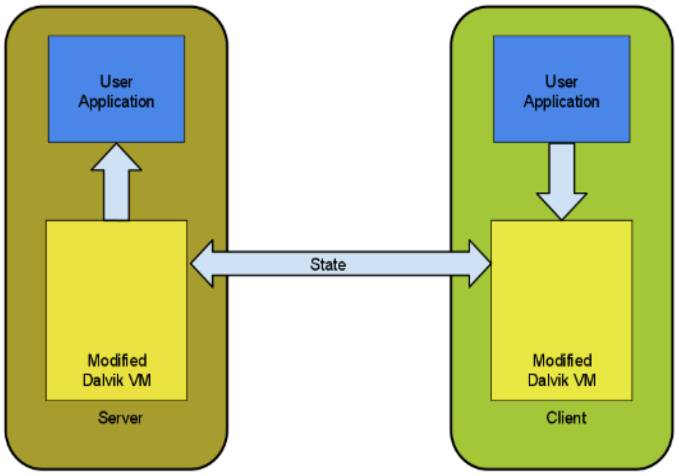

***Figure 2**: COFA architecture. On the server side, runs a modified Dalvik with server mode. On the client side, Dalvik runs in client mode. Taken from [4].*

In our implementation, we will send a Java file for execution in the Dalvik situated in Amazon instance. We will be introducing COFA and other work done on code offloading in the next subsection. We will also analyze and compare other related work.

2.1.1. Research

Previous work that can be mentioned is MAUI [5], CloneCloud [6], ThinkAir [7], Cuckoo [8], COMET [9], ECOS [10] and COFA[4]. MAUI's approach is using code annotations to determine which methods to offload. Annotations are set within the source code during development. The programmer annotates methods that should be taken into account for code offloading. MAUI uses its optimization framework to decide which method to send for remote execution. It measures network connectivity to the infrastructure and estimates its latency and bandwidth. All that is taken into consideration when dealing with optimization problem. If there is a disconnection, MAUI resumes code execution locally. However, MAUI did not consider scaling. Each time a new application was developed a new server proxy also had to be configured. This makes scaling hard because we cannot seamlessly do it. In this thesis, we will try to automatically scale out, this means we will use multiple independent instances together and pass out the workload between them. Unlike MAUI, we do not have to configure a new server proxy each time we add a new instance. Amazon Elastic Load Balancing tool will spread the workload between identical instances that Auto Scaling tool creates.

Another notable paper is about CloneCloud. Its essence primarily relies on the application-layer virtual machine (VM) side. [6]. CloneCloud is process based, each time a new application is made, a new process has to be configured. Unlike MAUI, CloneCloud's one goal is taking programmer out of the application partitioning business, it aims to make this procedure automatic. A chosen point of an independent thread is taken out of the program and migrated to a device's clone in the cloud. Eventually the thread returns with its remotely created state and is migrated back to the original state. Like MAUI, mathematical optimizer chooses these migration points. The partitioning part of CloneCloud is offline, it picks which parts to migrate for remote execution and which are executed on mobile device. In this thesis, like CloneCloud, we rely on application-layer level, we will be using Dalvik VM. In this thesis we will be using CyanogenMod7 that is based on Android version 2.3 [13], unlike CloneCloud that uses Android 1.5 Cupcake (application programming interface level 3). Android 1.5 was released 30 April, 2009, and now, in the year 2013, has less than 0.1% devices distributed with that version [11]. We will not be focusing on code offloading part as CloneCloud does, we will test how much load can an Amazon instance bear and how scalable can it be.

Third considerable work is ThinkAir. It is a framework to increase smartphone's capabilities with cloud computing. ThinkAir addresses scalability issues and removes environmental conditions that CloneCloud engenders by adapting online method of offloading. ThinkAir provides a library to make it easy for developers to exploit the framework with minimal effort. Like MAUI, ThinkAir uses code annotation. A part of the code is to be considered for remote execution, a programmer simply annotates it with *@Remote*. This gives the developer more control over what he wants or does not want to send for remote execution. ThinkAir code generator takes this annotated part of code and generates wrappers and utility methods. ThinkAir's execution controller starts the profiler to provide data. It also decides whether to execute the invocation remotely or not. If not, then execution is continued in the phone [6]. ThinkAir explores dynamic scaling of the computational power as well as the server side. ThinkAir’s VM automatically scales computational power and depending on the user’s requirements, it can allocate more than one VM for tasks. In this thesis, unlike ThinkAir that uses Oracle’s Java VM, we will be using Dalvik VM and test out how scalable and time consuming can that be.

Cuckoo is another framework which is implemented for better code offloading and is targeted at Android. It provides a dynamic runtime system which decides whether to execute a piece of code remotely or locally. Cuckoo also provides a programming model for mobile environments which supports both, local and remote execution. This is useful because smartphones may not always be connected to network, thus making cloud resources unavailable. Cuckoo has two services, first one is Service Rewriter. Cuckoo builder is called and has to be invoked before Java Builder and after Android Precompiler. It will rewrite generated stub, so Cuckoo could decide at runtime if a method should be called for remote or local execution. Second Cuckoo service is Remote Service Deriver. It generates an Ant build file for building a Java Archive File (.jar) for remote execution. The jar file contains the remote implementation [8]. Cuckoo differs from previously mentioned work because of its services. Although sending a .jar file for execution can be less time consuming, still each time a change is made in the code, the generation of Ant build file has to be repeated. Cuckoo generates dummy methods that the programmer can change. This is similar to MAUI and ThinkAir, where the methods that are annotated are sent for remote execution, in Cuckoo's case they are the dummy methods. Figure 3. shows the schematic overview of the Cuckoo's build process.

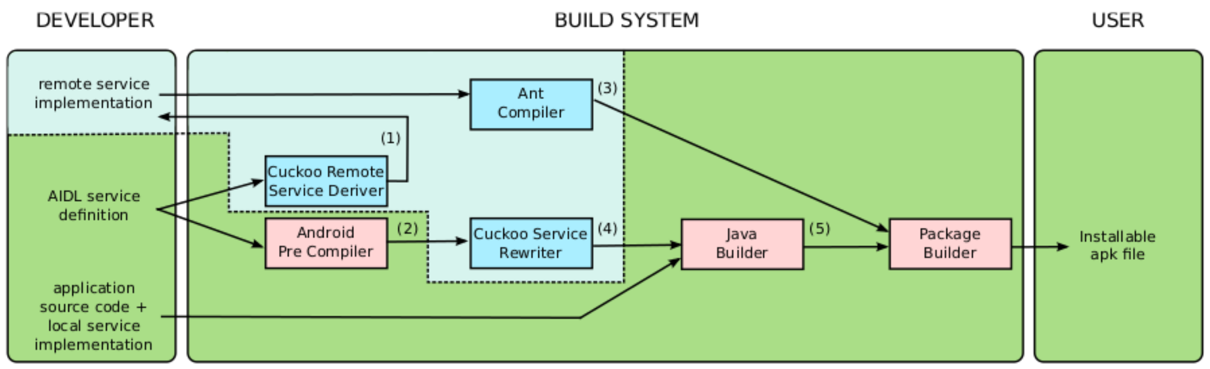

Figure 3*: The schematic overview of the Cuckoo's build process. The developer has to implement remote service, and local service implementation. They generate Ant build and dummy methods for remote execution. After building the installable apk file is generated. Taken from [8].*

Cuckoo has a Resource Manager that runs on smartphone. The remote resource has to be registered with the Resource Manager in order to the phone to recognize it. Once the resource is registered it can be used by any application that uses Cuckoo framework. Cuckoo also allows to collect remote resources from laptops and personal computers. Unlike ECOS, that will be introduced later, Cuckoo is not secure, meaning anyone can access the remote code and install any code to the system. This can be a problem when using Cuckoo framework, for example for some kind of image manipulation in the remote location. If someone with bad intentions has installed code onto the system that downloads that picture without the owner's permission, it would be a serious issue.

COMET, Code Offload by Migrating Execution Transparently, is a runtime system for code offloading. It is built on top of Dalvik VM and implements distributed shared memory (DSM), that is applied for code offloading. COMET's approach offers same benefits as specialized offloading systems. COMET is similar to MAUI which enabled automated offloading. When a connection is lost, COMET allows, like Cuckoo, computations to resume on a device. Unlike previously mentioned work, the developer does not have to write the logic of sending the code for remote execution. This sets a limitation to COMET, because it may decide to send code for remote execution that is actually not necessary for computations. COMET handles multiple-threaded environment by virtualizing threads across multiple endpoints. Similar to MAUI and CloneCloud, they assign IDs for objects in the tracked sets which allows endpoints to talk about shared objects [9].

Another framework, ECOS, Practical Mobile Application Offloading for Enterprises, uses a controller to leverage diverse compute resources. ECOS is focused on privacy and this is one of the measures it takes into account when deciding where to offload code. ECOS is focused on privacy, but its drawbacks are that it limits the resources used for remote execution. Encrypted code can cause latency overhead. They transferred 15KB of unencrypted and encrypted data for remote execution. With unencrypted data, they received the response in 0.3 seconds whereas with encrypted data, they received the response in 0.8 seconds. The decision of securing has to be done by the contents of data that will be sent for remote execution. In ECOS they statistically assigned security levels to applications and mobile devices. the Since ECOS is enterprise-centric, it is understandable, because enterprise applications operate on data which has strict privacy requirements. [5], [6], [7], [8], [8] Gave the understanding when and how to do code offloading but they did not describe the impact that sending code for offloading on a single remote instance may have [10]. ECOS introduced an idea that an offload is assigned to a dedicated resource, for ensuring increased performance. In this paper they do not mention scaling and issues it may cause.

An implementation based on MAUI was COFA, its difference was that instead annotating the code like MAUI and ThinkAir, they took programmer out of the application partitioning part. COFA's architecture was discussed in section 2.1.. They operate with application's binaries, and their implementation makes repacking existing binaries on Google Play easily off-loadable. Like MAUI, they have a profiler. On device's side it monitors battery consumption as well as thread executions. On the application's side it monitors thread execution times, if one takes up too much time, it is considered for remote execution. The profiler also monitors network, when its quality is poor then the code is not offloaded.

Annotating code that is sent for remote execution gives developers the power to decide what to execute locally and remotely. Like mentioned in [9] when the developer does not have to write the offloading logic, some unnecessary computations can be sent for remote execution. The profiler introduced by COFA could bring benefits when the annotation should be automatic. When the network quality is poor or a thread will not process too long, then a local execution should be considered. When doing code offloading, one should also think about security. Cuckoo framework pointed out that this can have some consequences when malicious users can access the remote system. ECOS tries to decrease these issues by assigning security levels. Another topic to think about is how to decide how strong should the encryption be, because it may cause latency overhead as described by ECOS. [8] and [9] only deal with code offloading, but in our thesis we will be carrying out tests to get more information about instance's load tolerance and scalability, these issues are not pointed out in previously mentioned papers. In the next subsection we will be discussing one commercial implementations that uses cloud computing capabilities to increase smartphone application’s performance.

2.1.2. Commercial Implementations

One of commercial implementations was introduced by Danihelka and Kencl [14], interactive 3D services over Windows Azure. Although Windows Azure is ideal for building rapidly scalable 3D environments, it is still hard to synchronize between several clients, due to cloud operations and network delays. The idea behind this implementation was to manipulate a 3D teapot in the cloud. Its architecture consists of several layers, 3D environment that is placed in the cloud and accessible by multiple devices. They use Silverlight 5 technology because it provides hardware-accelerated 3D graphics using Microsoft's toolset XNA. For cloud service they use Windows Azure. For rendering code reuse between the web browser with Silverlight and Windows Phone 7 they created a library to encapsulate difference between those two platforms. Tests involved scalability, reliability and response latency by manipulating 3D teapot with 24 different clients. Figure 4. shows the architecture of the experiment.

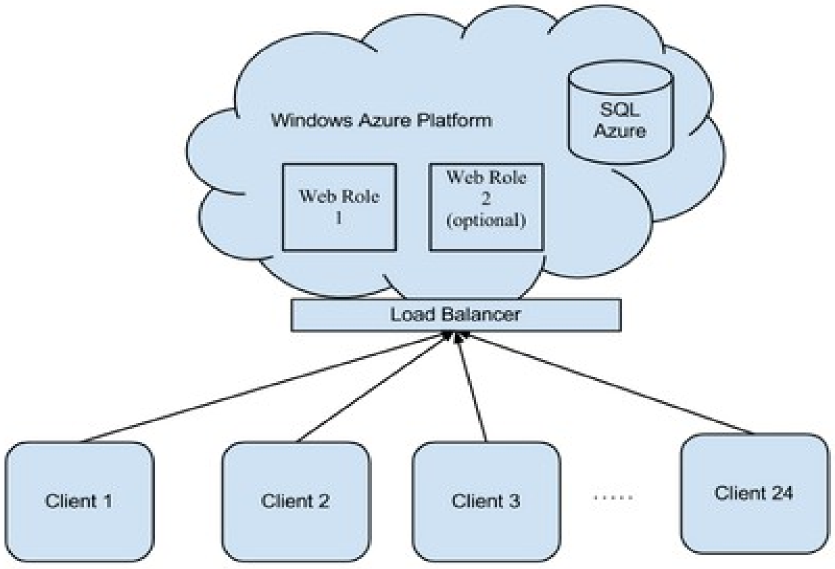

Figure 4*: Architecture of the experiment. Up to 24 clients send load to the load balancer which then divides the load between the web roles. Taken from [14]*

The clients were connected using 1 Mbps Internet. Simulation ran for 1 hour, starting with only one teapot instance and later incrementally adding instances, always 2 per client. The client instances were trying to connect to the web service every 100ms to synchronize the state of a single object. By the time 6 instances were manipulating the synchronization got slower, this means they could not see the image manipulations instantly. With 24 instances, there was no trace of synchronization. With growing instances, latency becomes a bigger problem, because it becomes less predictable and higher. [14]. In this experiment, they did not try scaling but this may be a solution to make the synchronization process faster.

3. Background

3.1 Scaling And Scalability

Scalability is a desirable attribute that has the ability to make applications or products to function well when computing power or volume’s size is changed. Scaling is the procedure of increasing the computing power. There are two types of scaling, horizontal and vertical. Vertical scaling means that when the load increases, more processors and volumes are added to extend processing capabilities. Horizontal scaling means that more processing nodes, multiple independent computers, are added for extending the computing power [15] [16]. In this thesis, we use horizontal scaling, adding more identical instances as the load increases. Application is not scalable if it encounters performance problems after scaling. [17]. In the next section we will describe Android operating system to get an overview of its architecture and nature.

3.2. Android

Android is a mobile operating system that runs on top of Linux kernel relying on its core system services like memory management, security, driver model etc. On top of kernel it has customized Dalvik Virtual Machine [11]. We will explain the nature of Dalvik VM in a later section. Kernel is a software that provides a layer between hardware and applications [18]. Android provides, with its core libraries, functionalities of Java. Figure 5. shows the major components of Android operating system.

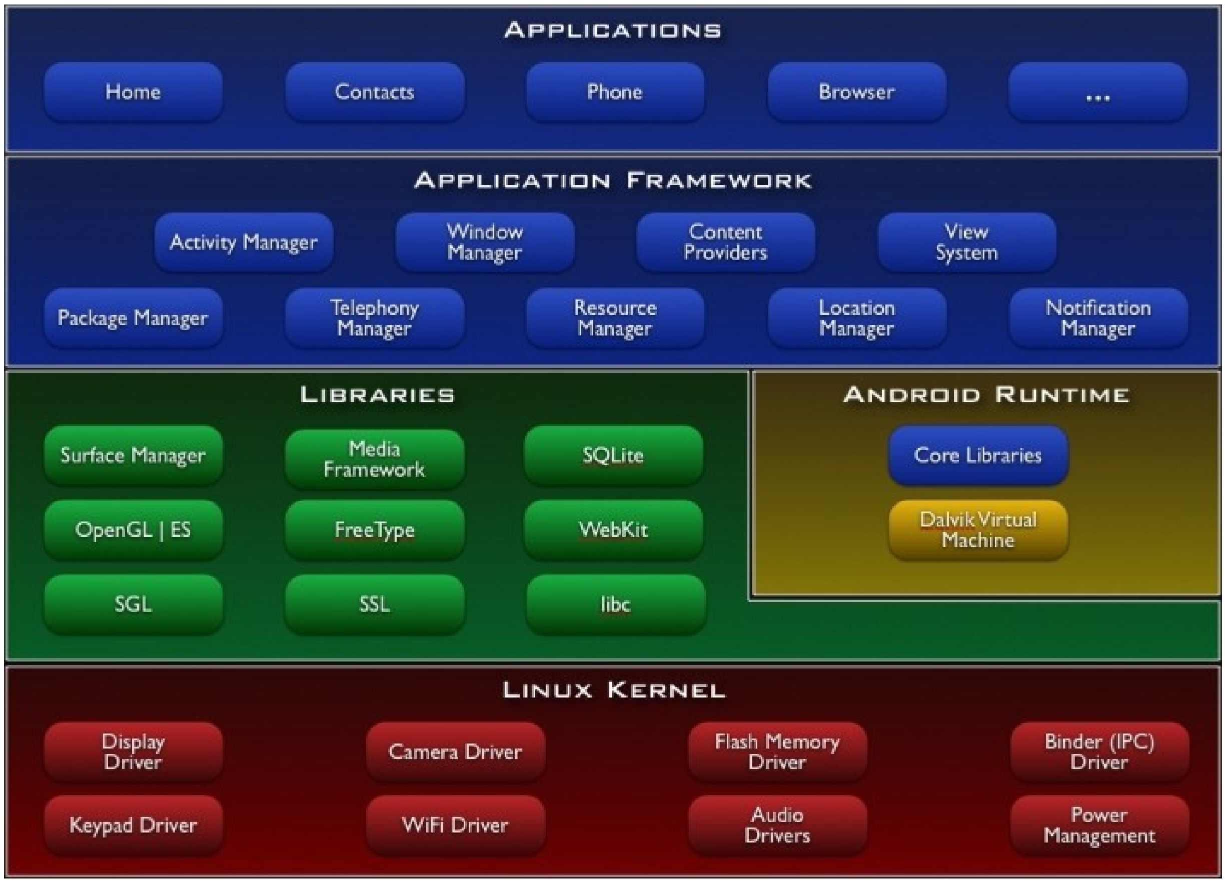

Figure 5*: Android operating system major components. The Linux Kernel lies in the bottom. Taken from [11].*

Android uses its own virtual machine, its runtime, which runtime manages memory. The last one also manages process lifetimes. Android applications that are running in the background will close when system needs memory. This is a bottleneck because sometimes there are too many applications running in the background and this causes slowness in everyday performance[12]. In this thesis we will use CyanogenMod that is a firmware based Android Open Source Project. We will briefly look into CyanogenMod in the next subsection.

3.2.1 CyanogenMod

In this thesis we are using CyanogenMod, it is a customized open source aftermarket firmware for Android devices and it is based on Android operating system. [13]. The difference between Android and CyanogenMod is that it is developed by a community. After Android version is released, it will be customized, added some new features and ported into several, new and old, devices. CyanogenMod is an alternative operating system for Android devices [19]. Like Android, CyanogenMod also has Dalvik VM,we will discuss its nature in the next section.

3.3. Dalvik Virtual Machine

Instead of Oracle's JVM, Android uses its own virtual machine named Dalvik. It is designed to run on devices that have low memory. Every application runs in its own Dalvik virtual machine instance, this requires Dalvik to be small. Figure 6. shows that Dalvik VM lies between Linux process and Android application.

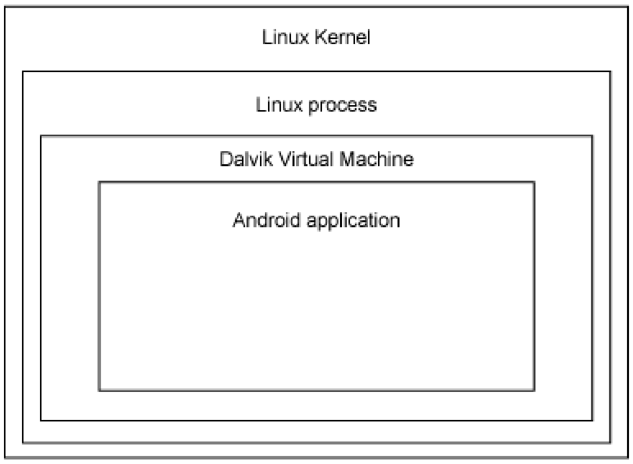

***Figure 6**: Layers of Android . Taken from [20].*

Dalvik executes .dex files (Dalvik executable) which are optimized for minimal memory [11]. Android applications are stored as .apk-s (application package file) that include dex bytecode and also resources. Apk file format is used to install middleware and application software on Android operating system. [21]. Dalvik bytecode is designed so that it would make less memory reads and writes and it has increased code density compared to Java bytecode [22].

It is possible to run Dalvik, just like a virtual machine, from a desktop system. First we have to compile our Java language resources, then convert and combine .class files into DEX files[1].
To run the code you need a bootclasspath script[2] for Dalvik. The script is also available in Appendix A. We will talk about how to setup the environment, including Dalvik VM, in chapter 5. We will deploy Dalvik in Amazon Elastic Compute Cloud which will be introduced it in the next subsection.

3.4. Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (later EC2) is a web service that provides compute capacity in the cloud. It allows to configure capacity with minimal effort and provides a complete control over computing resources. EC2 has a virtual environment that allows us to create instances and use web services. Its instances have variety of operating systems [23]. Amazon EC2 has several features [23]:

1. Amazon Elastic Block Store - Allows storage of Amazon EC2 instances. Elastic Block Store volumes are attached to the instances and persist independently from the life of an instance. Volumes are automatically replicated on the backend. It also provides snapshots that can be used as a starting point for a new instance. In this thesis we have made a 60 GB Elastic Block Store volume for our instance.

2. Elastic IP Address - Static IP addresses for dynamic cloud computing. It allows to mask instances and Availability Zone failures.

3. Amazon CloudWatch - Provides monitoring cloud applications and resources, visibility of operational performance, CPU Utilization, disk read and writes and network traffic.

[1] https://github.com/kohviuba/scalingAndroidImages/blob/master/Wraping%20class%20file%20to%20dex
[2] https://github.com/kohviuba/scalingAndroidImages/blob/master/rund.sh

4. Auto Scaling - Allows us to automatically scale EC2 instances according to the conditions defined. Instances are seamlessly scaled up to maintain performance and scaled down when the demand drops.

5. Elastic Load Balancing - Distributes incoming traffic across multiple EC2 instances. It also detects unhealthy instances and automatically reroutes traffic from them.

We will be discussing about the last two features in the later section. Elastic Compute Cloud has several different instance types, marked M1 and M3, with dissimilar specifications. First generation M1 instances are low cost and with a balanced set of resources that are suitable for many applications. Second generation M3 instances are for higher level of processing performance. In this thesis we will use M1 Small instance, because this is with the least we need, to deploy a smartphone image in the cloud. Its efficiency to handle workload will be tested and discussed in a later chapter. In the next subsection we will talk about Amazon Elastic Load Balancing tool that we will be using in this thesis to solve the scalability issues.

3.5. Amazon Elastic Load Balancing Tool

Amazon Elastic Load Balancing (ELB) Tool distributes incoming traffic and can detect the health of an instance. It automatically reroutes traffic from unhealthy instances, it will resume routing traffic in them when the instance becomes healthy again. Amazon Elastic Load Balancing Tool can also be used in Amazon Private Virtual Cloud. This tool will be useful for managing the scaling problem we have in this thesis. User makes a requests from the client computer to ELB, and the load balancing tool seamlessly spreads the load between the instances. Using this tool we can achieve more tolerant applications. In this thesis we will use this tool and stress test the instance with Tsung, this will be discussed in the next subsection. [23]

3.6. Tsung

Tsung is a load testing tool used to test scalability and performance. It is developed in Erlang. Erlang is a functional programming language, it is made to support hundred thousands lightweight processes in a single VM [24]. Erlang is concurrency-related language, writing parallel programs is easy because there are no shared resources and parallel processes do not have mutual exclusion [25]. Tsung can simulate huge number of users, we will use that to simulate offloading code to cloud. We will deploy Tsung on 3 instances, 1 controller and 2 worker nodes. Tutorial how to deploy Tsung in a cluster is available in Github[1].

[1] https://gist.github.com/huberflores/2827890

4. Problem Statement

Smartphones have become more popular over the years, partly because we can do so much more with them than just call and send text messages. We can use social networking services like Facebook, Twitter and Foursquare, read news, listen to music and watch videos. Smartphones are also replacing cameras and netbooks. People have the need to be in touch with everything and everyone and use their smartphones for that, when their personal computer or laptop is not around. With this increasing popularity people also have the need for more powerful smartphones, but with increased power some problems arise, like battery life. When we intensively use our smartphones we have to charge them daily. By offloading some work to cloud we could increase battery life indicators and also make computing process faster. The solution for increasing the computing capabilities, we could find in cloud computing. Next we will be discussing about this thesis' research problem.

4.1. Research Problem

Based on previous work we cannot make conclusions which instances to use for cloud offloading and what kind of architecture the implementation should have and weather we could scale an Android instance with x86 architecture. We are trying to figure out how much time would we spare when doing code offloading and how scaling the instance could make the process even faster. There are also drawbacks, for example network latency and cloud instances' load tolerance. We have to figure if our chosen instance will be sufficient in order to achieve a more powerful performance. In this thesis we will be using Amazon EC2 instance to test scaling and code offloading. The instance we use is M1 Small, it is sufficient enough for deploying our test environment, its parameters are discussed in more detail in the next chapter. Later we would change the instance type to M1 Medium and carry out the tests again. Another aspect of the research problem is whether a virtualized smartphone machine is scalable enough on the cloud when the load increases. In the next chapter we will set up a testing environment and carry out several tests.

5. Towards A Scalable Infrastructure For Code Offloading

This chapter describes how to set up the environment for solving the problem stated in chapter 4. It also describes the tests that were made and analysis of their results. We will be setting up a single Android instance and a server on it. With Tsung tests we will send a calculation method to the Android instance's server that will send the file for execution. Tsung will simulate several users to test our instance's workload tolerance. Later we will use Amazon's Load Balancing tool for scaling and managing the load. We will add two more instances that are identical to our Android instance and have the same server on them.

5.1 Setting Up An Instance In The Cloud

In this section we describe the setup of Android image with x86 architecture. We will use this instance to test how much faster is it to run the code in cloud. The code is first executed locally in the instance, because this thesis does not focus on code offloading, so we will not be fully implementing it in the boundaries of this thesis. First we had to create an instance in Amazon. It is a rather simple procedure that can be carried out in EC2 Management Console. We made a 64-bit Linux instance with 60GB of storage. The instance type is M1 Small, it has 1.7GiB memory and 1 EC2 Compute unit. [23]. For creating an instance see the Amazon Elastic Compute Cloud tutorial [1] . After we had launched the instance, we deployed CyanogenMod. The tutorial is available in Github[2]. We also had to get Android SDK for the instance, it is available on Android Developers Page.[3] There is a list of dependencies that had to be installed before we could set up CyanogenMod7 [26]. Installing ia32-libs dependencies caused some problems on Cloud. The solution to that was to enable installation of i386 packages. When deploying Android x86 on Cloud, we used Oracle's Java 1.6.. Another important point is that we need a certain hierarchy of the environment in order it to work. Figure 7. shows the structure.

[1] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html#EC2_LaunchInstance_Linux

[2]https://gist.github.com/huberflores/4687766

[3] http://developer.android.com/sdk/index.html

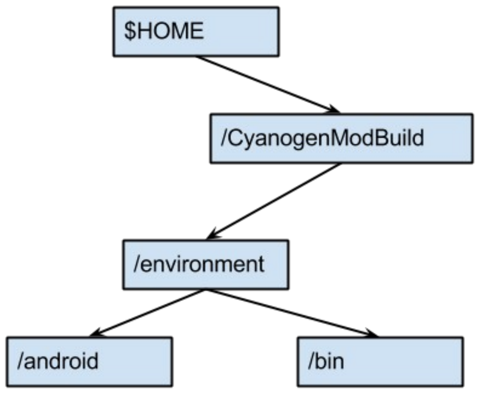

***Figure 7**: Directory's hierarchy that is necessary for our solution to work.*

The tutorial in the appendix has a mistake, two directories *android* and *bin* must be made under */environment* not in the root folder. After initializing the repository and synchronizing it we can move on to building an isolated Dalvik machine. We will describe it in the next subsection.

5.1.2. Building Isolated Dalvik Machine

After setting up the CyanogenMod environment, we were able to build an isolated Dalvik machine, so we could run tests in the cloud. The full tutorial is available in Github[1]. For building Dalvik, we needed to be in the *$HOME/CyanogenModBuild/android/system* directory. We chose *full_x86-eng* image for Dalvik. Once the environment was ready and built, we had to create a bootclasspath script for launching Dalvik VM. It is available in the Git repository[2] and Appendix A. Now we had our basic test environment ready, we were able to carry out tests on them that will be described and discussed in the next section. We added a server to our instance after we had carried out the simple benchmarking tests on our instance

[1] https://gist.github.com/huberflores/4714824

[2] https://github.com/kohviuba/scalingAndroidImages/blob/master/rund.sh

and in our smartphone. The server is necessary for stress testing with Tsung. Its implementation will be described in the next section along side with the tests carried out on the phone and on the instance.

5.2. Benchmark Testing Android x86

In this subsection we will compare and contrast the results of the tests that were ran on Android-x86 image and on a physical device. The device we used was Samsung Galaxy S II with Android 4.1 Jelly Bean. It has 1 GB of RAM, Dual-core 1.2 GHz Cortex-A9 CPU. [27]. After testing the M1 Small instance, we changed its type to M1 Medium and carried out the tests again. M1 Medium instance has 3.75Gib memory, 2 EC2 Compute Units and supports both, 32-bit and 64-bit platforms. We had 10 methods, we ran each of them ten times in the cloud and on the device and marked down their execution time in both cases. In these tests we did not consider the time the method is sent to cloud. The code of the methods can be seen in a Github repository[1] and Appendix A, here are short descriptions of the methods:

1. IsPrime - This method takes an integer array and iterates through it. If an element of the array is a prime number, then it prints out " [number] is prime", otherwise "[number] is not prime".

2. PrimeFactors - This method takes an integer array with 12 elements in them and iterates through it, calculates every element's prime factors and prints them out.

3. BubbleSort - This method takes an integer array with a length of 10 000. Each element is a random number between 1 and 1 000 000. It sorts the array using bubble sort algorithm.

4. MultiplyMatrices - This method takes 16x16 matrices and multiplies them and after that prints out the new matrix.

[1] https://github.com/kohviuba/scalingAndroidImages/tree/master/methods

5. Quicksort - This method takes an integer array with 1 000 000 elements. Each element is a random number between 1 and 100 000 000. The array is sorted using Quicksort algorithm

6. FloydWarshall - This method takes a 16x16 matrix and finds a shortest path in a weighted graph using Floyd-Warshall algorithm

7. Fibo - this method calculates Fibonacci numbers from 0 to 200000.

8. NQueens - This method is an implementation of the n-Queens problem on 8x8 chessboard.

9. TowersOfHanoi - This method solves the Towers of Hanoi problem.

We intentionally use big numbers for calculations so the difference would come out clearly. The full tables with the test results are in Github[1] and Appendix A. Here we bring out the average result of each test. Table 1. represents the results.

[1] https://github.com/kohviuba/scalingAndroidImages/tree/master/methods

Method Name	Result in M1 Small (milliseconds)	Result in M1 Medium (milliseconds)	Result in the device(milliseconds)
IsPrime	4776.0844	2395.7121	5433.225
PrimeFactors	29943.0607	15950.0059	7145.987
BubbleSort	6846.0144	3989.3402	2895.652
MultiplyMatrices	0.1931	0.1818	0.7382
Quicksort	2518.9266	1393.3065	1616.7306
FloydWarshall	3.8956	3.3951	94.4131
Fibo	198.2832	98.8443	81.4402
NQueens	88949.3576	77818.2227	11509.5300
TowersOfHanoi	124.0893	22.981	211.9660
GreatestCommon Divisor	0.0144	0.0128	0.0389

Table 1. Average execution times of the tested methods. Methods were ran in the device and then in the cloud Android instances.

Although some methods ran faster on a device than on a cloud instance, compared to M1 Medium instance, it would be beneficial to run computations in the cloud. M1 Small instance stayed slow for our calculations. For detecting how much load can each instance bear we will do some stress and performance testing using Tsung, we also added a Apache Tomcat server on our Android instance, this test case is described in the next subsection.

5.3. Performance Testing Using Tsung

This section describes the tests we ran with Tsung and their results. For stress testing we set up a cluster with three Tsung images. Tutorial on how to deploy Tsung in a cluster can be found in Github[1]. We had to configure them so, that there would be one controller and two worker instances. Each Tsung instance had to have other instances public keys in order to communicate via SSH. The controller node has a configuration file[2], which is available in Appendix A, that makes a connection with the worker nodes, sets the server, the request and the number of clients. In the Dalvik instance, we set up an Apache Tomcat server. Its servlet takes Tsung's uploaded Java file, compiles it and wraps the class file into a dex file and then executes it with Dalvik. The code for the servlet can be seen in a Git repository[3] and Appendix A. When setting up the environment we had to bare in mind that in order to write and execute files we have to give certain permissions for the catalog where the uploaded files will be written and executed. Figure 8. shows the architecture of our implementation.

[1] https://gist.github.com/huberflores/2827890
[2] https://github.com/kohviuba/scalingAndroidImages/blob/master/Tsung%20tests/example.xml
[3] https://github.com/kohviuba/scalingAndroidImages/tree/master/Tsung%20tests/webandroid

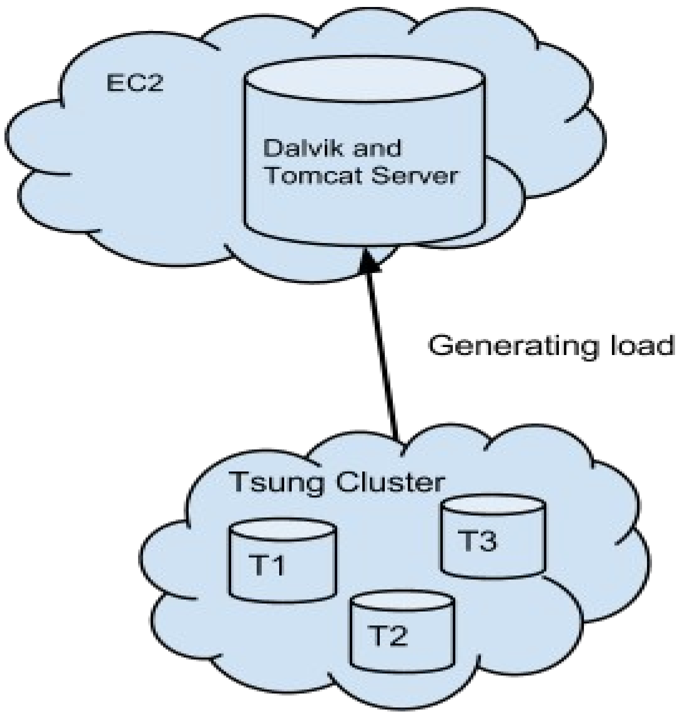

Figure 8: *Tsung cluster generating load(T1, T2 and T3 are Tsung instances) and sending it to Tomcat server in our Dalvik instance.*

We used Tsung recorder to write a user scenario. We put Tsung recorder listening to a port, opened a web browser and configured a new proxy and went to a page we needed to record. [24]. We uploaded a file, on our local computer's browser, in that page and sent it to the server. After recording the scenario we used the output in our configuration file. The usage of this file can be seen in the Github configuration file [7]. In our first test we had one arrival phase, that lasted for 2 minutes. Each second we added 2 users, so in total there was 240 users added. The count of finished users was 239. Figure 9. shows that the instance can handle that amount of users simultaneously. The duration of user's session was 16 minutes and 55 seconds which is clearly a lot of time. Full reports and graphs can be seen in the Github repository[1] and Appendix A and the meaning behind the statistics can be found out from Tsung's user manual[2].

[1] https://github.com/kohviuba/scalingAndroidImages/tree/master/Tsung%20tests/stats
[2] http://tsung.erlang-projects.org/user_manual.html#htoc72

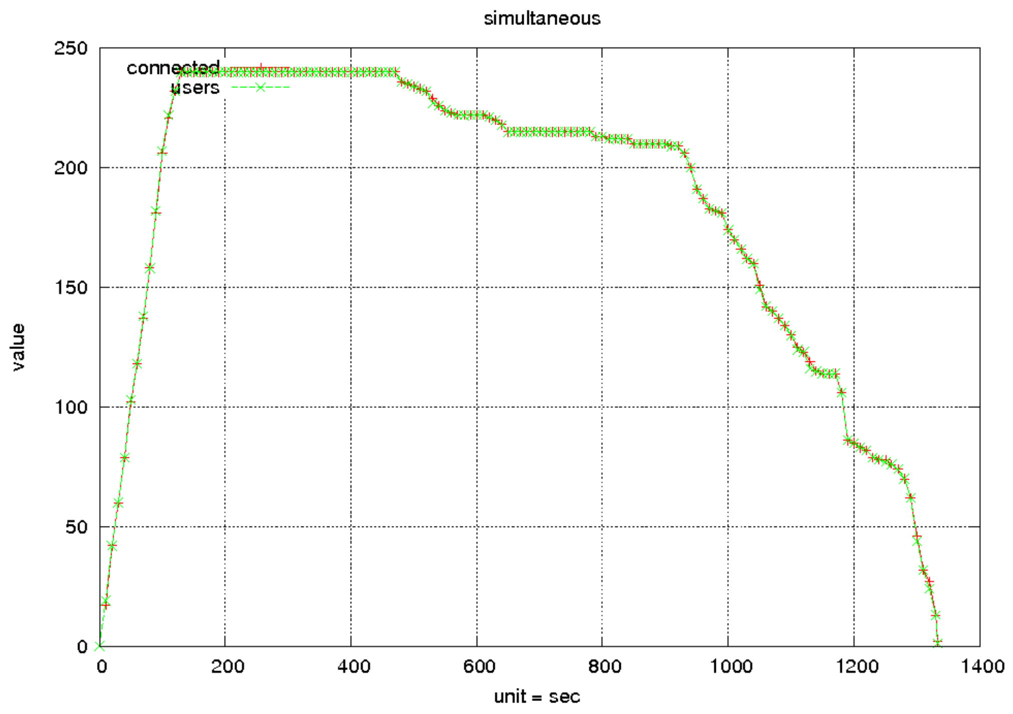

Figure 9: *Simultaneous users. Maximum user count was 240. The red line describes connected users and the green line describes overall count of users.*

Since this load seemed to be tolerable for the instance, despite the time spent on the session, we decided to increase it by adding more users. In this test, we added 3 users every second. 25 of the users got the response that request timed out. The user's session lasted for 18 minutes and 44 seconds. Figure 10. shows the simultaneous users graph.

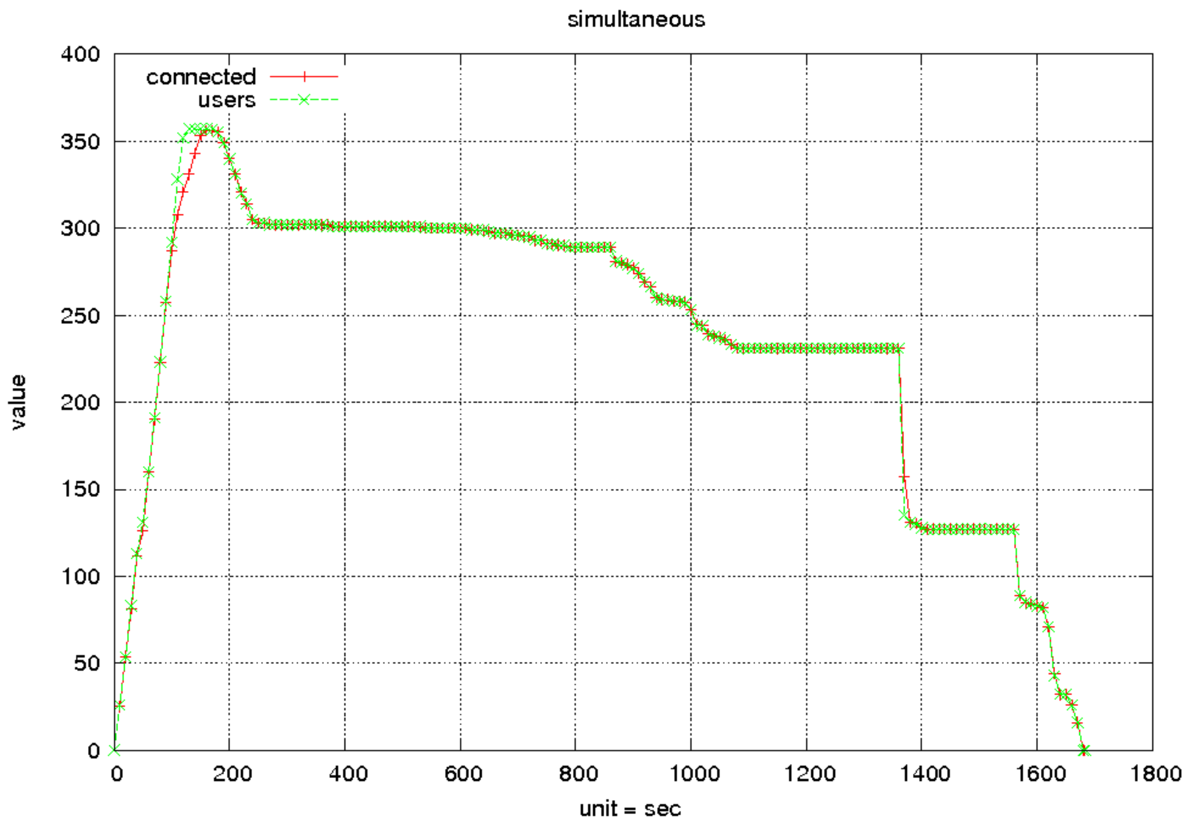

Figure 10: *Simultaneous users graph. Red line describes connected users and green line overall count of user.*

As seen from the Figure 10. since the 80 second the load becomes too big, there are less connected users than overall users. The connected users count starts to drop around 300 users, so we assume, based on previous graphs and tests, that one Android instance can bear maximum 300 users. When carrying out the same tests with M1 Medium instance, we noticed that it could only bear 220 users. This may be caused by changing the instance's type. We were unfortunately unable to trace back the problem. Figure 11. shows the simultaneous users graph. Every second there was added 2 users and the test lasted for 120 seconds.

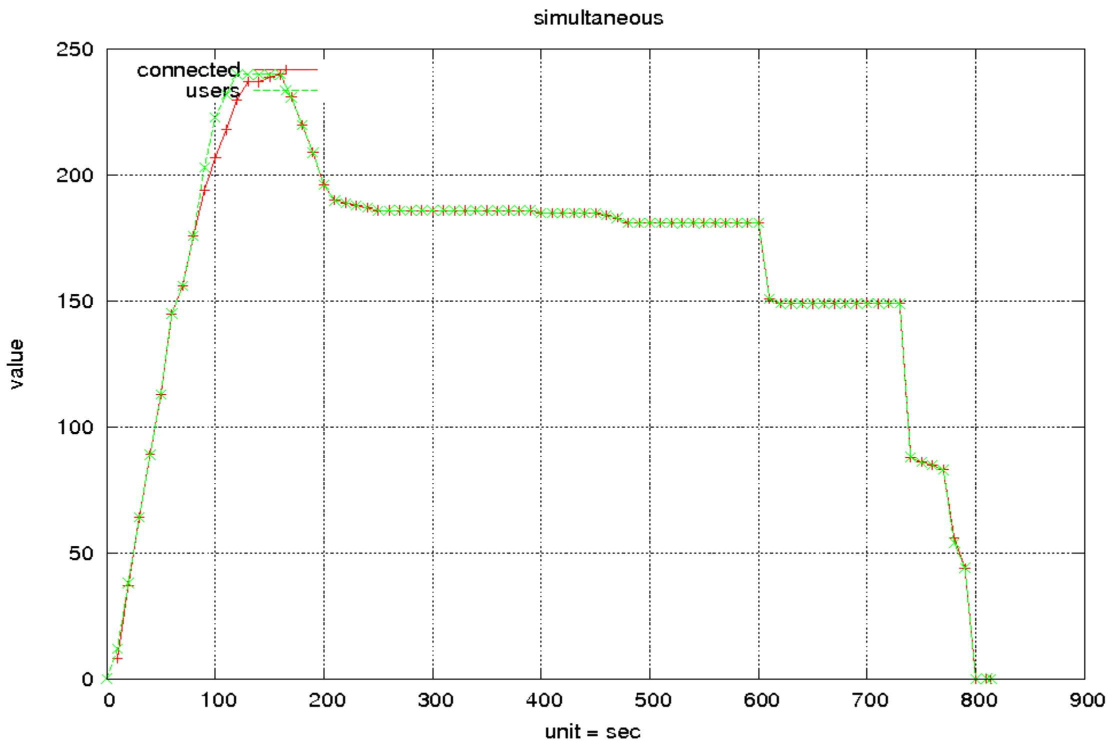

Figure 11: M1 Medium instance's simultaneous users graph. Around 80th second the instance is overloaded

When we were carrying out this test in M1 Medium instance, the server loaded itself until the tests finished, when entered via web browser. During the test, instance's CPU was 100% . This may have been one of the problems why there is little load tolerance. We will try to increase the tolerance by scaling the instance and then carrying out the tests again. We set up Elastic Load Balancing tool, and then run the tests again. These will be discussed in the next section.

5.4. Scaling the Android x86 EC2 Instance

In this subsection we will set up two identical Amazon instances, add an Elastic Load Balancer, scale the instance with Auto Scaling tool when load gets bigger and run Tsung tests, that were described in section 5.3., again. For ELB we had to add at least one instance more in addition to our Android-x86 because this tool can be used with two or more instances. For making identical instances we first need to create a new Amazon Machine Image (AMI) of our Android instance. This is a rather simple procedure and can be carried out in the EC2 Management Console. By right-clicking on the selected image, we can choose "*Create Image (EBS AMI)*" from the menu. Once the procedure is finished, the new AMI will be available under "*Images*" section in the left side panel of the Management Console. We will create new instances in the same availability zone as our main instance, but Elastic Load Balancer tool can manage instances in different availability zones, so it is not compulsory to create them in the same zone.

5.4.1. Adding Elastic Load Balancer With Auto Scaling

Once we have our instances running, we can add Elastic Load Balancer tool for load managing. A guide for setting up Elastic Load Balancer can be found in Amazon documentation[1]. For right communication between the ELB and back-end server, we need to set server timeout for 60 seconds [28]. Once the ELB is created, we can verify that the description matches our specifications by clicking on the load balancer. When instances has been registered and are in service, we can test our load balancer to check that it works as needed by copying the DNS name of the load balancer on our browser address field. If the balancer is working correctly one should see the back-end server's default page.

Next we set up Auto Scaling tool. It will automatically scale our instances when the load increases and the ELB will divide it between the new instances. Figure 12. shows our

1 http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/UserScenariosForEC2.html

improved implementation that includes in the section 5.3. described Tsung cluster and Dalvik instance with Tomcat server.

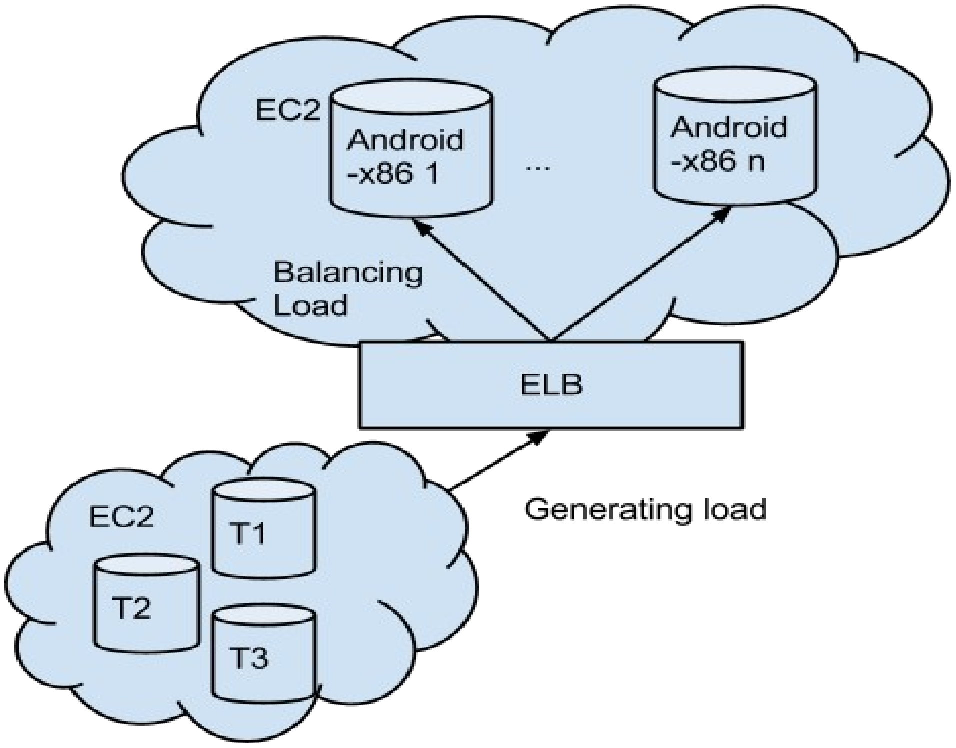

***Figure 12**: Tsung Cluster generating load, sending it to Elastic Load Balancer (ELB) that is balancing the load between instances. When the load increases, another Android-x86 instance is created.*

For setting up Auto Scaling tool, see the AWS documentation[1]. In order this tool to work properly we had to add paths to Java JDK and Auto Scaling tool directory respectively. The guide to add them is also in the AWS documentation [1]. Once the setup has been done, we can create a launch configuration. The launch configuration's instance id in our case is *ami-978d9ae3* and instance type is M1 Small. After we had created the launch configuration, we also had to create an Auto Scaling group. We put the minimum size of instances 2 and maximum 10. While creating the group, we also had to describe the load balancer that we wanted to attach. For scaling out we had to define a scaling policy that will add one instance with a cool-down of 30 seconds. When the policy is created, we will add an alarm, so that

[1] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-auto-scaling-elb.html

when after 60 seconds, CPU is higher than 60%, our scaling policy will be fired. For scaling down we also had to make a policy which removes an instance when its CPU is less than 40%. Alarms can be created from the EC2 Management console with those defined policies as well as from the command line interface. While executing the Tsung tests again, now with a Amazon Load Balancing tool dividing load between instances, we ran into a problem. Like seen from the results of the tests that were ran in section 5.3., getting response back takes time. Our problem was, that the Amazon Load Balancing tool has a session timeout in 60 seconds and it cannot be configured. We were able to see that when the load increased, the scaling policy was fired and new instances added. The load was divided between the healthy instances. Because of the timeout, we could not get relative statistics to conclude how much more load could the instances bear and how scalable the virtualized smartphone image is. Drawbacks of our implementation, these will be discussed in the next subsection.

5.4.2 Implementation’s Drawbacks

In this section we will talk about the drawbacks we had with our implementation, that could not give us proper test results for making conclusion. In our servlet implementation, we wrote the user’s uploaded Java file to disk for later execution. After simultaneously writing files on the disk we also had to compile them with javac. This operation raised the CPU usage to 100%, thus making it slow. When we ran these load tests against a single Android-x86 instance, without ELB balancing the load, it took time to compile, wrap the .class file in dex and execute it with Dalvik and get the response back. We measured the time how much each command took time. We used for this test our MultiplyMatrices method and ran them in M1 small and M1 Medium instance. We ran each command 10 times, here we bring out the results' average. tests results are available in Github[1] and in Appendix A. Table 2 shows the results of our test.

[1] https://github.com/kohviuba/scalingAndroidImages/blob/master/Execution%20time%20test%20-%20Sheet1.pdf

Command	M1 Small	M1 Medium
javac	3.1558	1.3574
dx	1.5415	0.6489
rund.sh	0.105	0.2279

Table 2. Commands execution times in instances M1 Small and M1 Medium. Javac compiles the .java file. Dx wraps the generated .class file into .dex and rund.sh executes the .dex file

Like we can see, compiling Java takes a little over 3 seconds in instance M1 small, and when running multiple compilations at the same time, makes it even slower. Although the process got faster when we deployed the Elastic Load Balancer and Auto Scaling, but then we had another drawback that was the ELB's timeout. It cut down our responses after 60 seconds because within this time, we had not got back responses to our requests. We were able to see that the Android images were scaled nicely as the load increased, but we could not get the result, how much users could the instances bare, when instances are scaled horizontally. Because of this drawback we did not try scaling with M1 Medium instance.

6. Conclusions And Future Possible Work

One of the goals of this thesis was to deploy an Android-x86 image to an Amazon M1 Small instance and to find out whether this instance is sufficient enough for code offloading, how much load can it bear. Another goal was to find out how scalable is an Android-x86 image. In conclusion we are able to say that an Amazon instance M1 Small is sufficient to deploy an Android-x86 image, but our tests that were carried out showed that it is slower than a physical device for running computations and bearing only 300 simultaneous users.

Tsung load tests were carried out and we were able to see, how many concurrent users can use the instance. A server for uploading a file and sending it for execution to our Dalvik instance was also implemented. An Amazon Elastic Load Balancer tool was used together with Auto Scaling. The load balancer divided workload between instances. If one instance's CPU raised to 60% another instance was made and rest of the load was sent there. This procedure was repeated if necessary and up to 10 instances made. Although we were able to see that with our implementation, when the workload increased, more working nodes were added, it is not sufficient for making a conclusion whether a smartphone image is scalable, because we had several drawbacks. The Elastic Load balancer tool had a timeout of 60 seconds and that cut off our requests, so although we were able to get some responses back, we did not get enough relevant information to make any conclusions. These drawbacks will be taken into account and are adequate for future work. When doing code offloading we should think how to handle storing the code, that was sent for remote execution, while we compile and execute it. The Amazon tools that we used in this thesis made the implementation of scaling simple and understandable.

For future work the implementation of a servlet we used could be improved, for example instead writing the incoming file to disk we could handle it by writing it into memory. Another possibility is to think maybe simply compiling and running the classes with Java instead of Dalvik. Because we did not mainly focus on the code offloading side, the implementation needs optimization. This could lead to better test results. Right now, when the server receives a file, it writes it to the disk, then compiles it, wraps the generated .class file into dex and executes it with Dalvik. Compiling Java classes is rather slow, especially when there are several Java compilers running. For better offloading results, in the future we could use Java Compiler API [29] to compile parts of the code. Another approach would be sending a file that is already wrapped in dex for remote execution.

Virtualiseeritud nutitelefoni platvormi skaleerimine pilvekeskkonnas

Bakalaureusetöö (6 EAP)

Kristiina Ritso

Resümee

Üks selle Bakalaureuse töö eesmärkidest oli Android-x86 nutitelefoni platvormi juurutamine pilvekeskkonda ja välja selgitamine, kas valitud *instance* on piisav virtualiseeritud nutitelefoni platvormi juurutamiseks ning kui palju koormust see talub. Töös kasutati Amazoni *instance*'i M1 S*mall*, mis oli piisav, et juurutada Androidi virtualiseeritud platvormi, kuid jäi kesisemaks kui mobiiltelefon, millel teste läbi viidi. M1 *Medium instance*'i tüüp oli sobivam ja näitas paremaid tulemusi võrreldes telefoniga.

Teostati koormusteste selleks vastava tööriistaga Tsung, et näha, kui palju üheaegseid kasutajaid *instance* talub. Testi läbiviimiseks paigaldasime Dalviku *instance*'ile Tomcat serveri. Pärast teste ühe eksemplariga, juurutasime külge *Elastic Load Balancing* ja automaatse skaleerimise *Amazon Auto Scaling* tööriista. Esimene neist jaotas koormust *instance*'ide vahel. Automaatse skaleerimise tööriista kasutasime, et rakendada horisontaalset skaleerimist meie Android-x86 *instance*'le. Kui CPU tõusis üle 60% kauemaks kui üks minut, siis tehti eelmisele identne *instance* ja koormust saadeti edaspidi sinna. Seda protseduuri vajadusel korrati maksimum kümne *instance*'ini. Meie teostusel olid tagasilöögid, sest *Elastic Load Balancer* aegus 60 sekundi pärast ning me ei saanud kõikide välja saadetud päringutele vastuseid. Serverisse saadetud faili kirjutamine ja kompileerimine olid kulukad tegevused ja seega ei lõppenud kõik 60 sekundi jooksul. Me ei saanud koos *Load Balancer*'iga läbiviidud testidest piisavalt andmeid, et teha järeldusi, kas virtualiseeritud nutitelefoni platvorm Android on hästi või halvasti skaleeruv.

Licence

Non-exclusive licence to reproduce thesis and make thesis public

Bibliography

[1] - “Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter, According to IDC”
URL: http://www.idc.com/getdoc.jsp?containerId=prUS23771812#.UREd6eninz4
last visited: 05.01.2013

[2] Cloud Computing and Smartphones
URL:
http://cloudtimes.org/2011/03/01/cloud-computing-and-smartphones/ last visited: 06.01.2013

[3] Mobile cloud computing URL: http://en.wikipedia.org/wiki/Mobile_cloud_computing] last visited: 02.02.2013

[4] Deepak Shivarudrappa, MingLung Chen, Shashank Bharadwaj URL: https://bitbucket.org/shashank/shashank.bitbucket.org/src/851f08735551/projects/cu/COFA.pdf last visited: 06.05.2013

[5] Eduardo Cuervo , Aruna Balasubramanian, Dae-ki Cho,
Alec Wolman , Stefan Saroiu , Ranveer Chandra , Paramvir Bahl.
MAUI: making smartphones last longer with code offload. *Duke University, University of Massachusetts Amherst, UCLA, Microsoft Research* ACM, 2010

[6] Byung-Gon Chun, Mayur Naiak, Sunghwan Ihm, Petros Maniatis, Ashwin Patti . CloneCloud: Elastic Execution between Mobile Device and Cloud. *Proceeding EuroSys '11 Proceeding of the sixth conference on Computer systems* Pages 301-314

[7] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier and Xinwen Zhang
ThinkAir: Dynamic resource allocation and parallel execution in cloud for mobile code offloading

[8] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a Computation Offloading Framework
for Smartphones. URL:
www.cs.vu.nl/~rkemp/papers/kemp-mobicase2010.pdf last visited: 03.05.2013

[9] COMET COMET: Code Offload by Migrating Execution Transparently, Mark S. Gordon†,D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, Xu Chen
OSDI '12 Proceeding of the 10th USENIX conference on Operating Systems Design and Implementation Pages 93-106

[10] Aaron Gember, Chris Dragga, Aditya Akella. E COS: Practical Mobile Application Offloading for Enterprises *Hot-ICE '12 Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet,*

Cloud, and Enterprise Networks and Services Pages 4-4

[11] Android Developers URL: http://developer.android.com/ last visited: 07.04.2013

[12] Reto Meier Android 4 Application Development Page 8

[13] CyanogenMod7 URL: http://www.cyanogenmod.org/about last visited: 07.04.2013

[14] Jiri Danihelka Lukas Kencl . Interactive 3D services over Windows Azure URL: www.rdc.cz/download/publications/danihelka12cloudfutures.pdf last visited: 05.05.2013

[15]André B. Bondi Characteristics of Scalability and Their Impact on Performance *WOSP '00 Proceedings of the 2nd International workshop on Software and performance* Pages 195-203, ACM 2000

[16]Vertical Scalability URL: http://www.techopedia.com/ last visited: 05.05.2013

[17] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve and Alla Segal. Dynamic Scaling of Web Applications
in a Virtualized Cloud Computing Environment. e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on 21-23 Oct. 2009. Pages 281- 286 IEEE Xplorer

[18] How Linux Kernel Works URL: [http://www.tuxradar.com/content/how-linux-kernel-works last visited: 01.04.2013

[19] About CyanogenMod7. URL: http://wiki.cyanogenmod.org/ last visited: 07.04.2013

[20] Figure of Dalvik architecture URL: http://www.ibm.com/developerworks/opensource/library/os-android-devel/fig02.gif last visited: 07.04.2013

[21] APK file extension URL: http://en.wikipedia.org/wiki/APK_(file_format) last visited: 07.04.2013

[22] Stefan Brähler Analysis of the Android Architecture. Karlsruhe institute for technology, 2010

[23] Amazon Elastic Compute Cloud URL: http://aws.amazon.com/ec2/ last visited: 01.05.2013

[24] Tsung's Users Manual URL: http://tsung.erlang-projects.org/user_manual.html last visited: 29.04.2013

[25] Erlang. URL: http://www.macs.hw.ac.uk/~ag275/tutorial/erlang/ last visited: 28.04.2013

[26] Initializing Build Environment. URL: http://source.android.com/source/initializing.html last visited: 07.04.2013

[27] Samsung Galaxy S II and Galaxy Y Specifications. URL: http://www.samsung.com/ last visited: 25.04.2013

[28] Deploy Elastic Load Balancing in Amazon EC2-Classic URL: http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/US_SettingUpLoadBalancerHTTPS.html last visited: 01.05.2013

[29] Java Compiler API. URL: http://www.javabeat.net/2007/04/the-java-6-0-compiler-api/ last visited: 05.05.2013

Appendix A

Tutorials how to deploy CyanogenMod7, Dalvik VM and Tsung can be accessed https://gist.github.com/huberflores . Methods used for testing and their results alongside Tsung test results are available at https://github.com/kohviuba/scalingAndroidImages and on a CD attached to this thesis.