UiO : Department of Informatics
University of Oslo

Distributed Objects at the Edge of the Cloud using Emerald

Test evaluation of compiling and running Emerald programs on Smartphones

Khiem-Kim Ho Xuan - kkho@ifi.uio.no
master thesis spring 2014

Distributed Objects at the Edge of the Cloud using Emerald

Khiem-Kim Ho Xuan - kkho@ifi.uio.no

January 26, 2014

Abstract

This thesis is about research and implementation on a distributed object-oriented programming language called *Emerald* on smartphones. Our goal is to couple smartphones seamlessly to the cloud where we span the gap between apps and cloud services. We therefore want to use a distributed object-oriented programming language that has object mobility to allow moving both programs and data seamlessly across the spectrum of smartphones, near and far clouds. We have ported the Emerald language on Android thereafter we have experimented with various use cases and performed an evaluation.

The evaluation includes benchmarking of the Android implementation of *Emerald* and more subjective evaluation of the chosen use cases. We also present some limitations of what Android can provide when it comes to distributing objects and how far we are willing to go to make it work.

Contents

List of Figures

List of Tables

Preface

I would like to thank my supervisor Eric Bartley Jul for a really good guidance, stories, spending money on dinner with me and lecturing the course **INF5510 - Distributed Objects**.

I want to set my appreciation to Norm Hutchinson for helping me understanding and debugging the source code of the *Emerald* compiler.

I would like to thank my fellow master students Thiseas Mengos, Endri Hysenaj, William Almnes and Georgios Patounas for working together on courses and the simulation of our project during our studies together. I would also want to say thank you to my friends, family and especially my old soccer coach Kjell Dybvik, my brother Tony Ho Xuan Huy and my mother Ai Thi Nguyen for being there for me during my studies,work and my social life.

I want to thank Olga Voronkova, Eva Andritsopoulou, Georgios Patounas William Almnes, Ivar Tryti, Henrik Kjus Alstad, Tony Ho Xuan, Nils Peder Korsveien, Anes Mulic and Endri Hysenaj for proof reading my thesis.

And to not forget Arne Maus, Ellen Munthe-Kaas, Tor Skeie, Stein Michael Storleer, Thomas Plagemann, Vera Goebel and Barbara Hecker(Hacker) for all the lectures and for helping me to become a better developer.

I can not and I will never forget my grandmother, may you rest in peace.

To anyone that I might have forgotten to mention, then I am truly sorry.

Oslo, February 3rd, 2014
Khiem-Kim Ho Xuan

Part I

Introduction

Chapter 1

Introduction

The motivation for our project is to move cloud computing across the spectrum from a near to far cloud. That is why we have moved Emerald to the edge device to make it happen. In other words, we want to distribute objects as far as we can to any devices spread over the spectrum where we use the Emerald programming language. The apps today can not do such computation and does not move to the very edge of the cloud which they are useless in doing so. That is why we want to build a layer on top of *Emerald* in a smart device environment where the *Emerald* language does the distribution all the way down to the handset. Our interest for porting the Emerald language to a smartphone is because Emerald is an *on-the-fly fine-grained mobility* language where the Emerald objects can be small data objects or process objects that can be moved to any machine at any time[17]. We want our smartdevices to receive, compute and move the object to another device or even cloud servers.

1.1 Motivation

In the 1980s, Eric Bartley Jul, Norman Hutchinson, Andrew Black and Henry Levy created a distributed object-oriented programming language known as Emerald[18]. The Emerald language is a way to move objects to another machine in a distributed way where it is lightweight and fine-grained mobile. We only need to write one line of code to move an object to a specific area we have chosen. Their solution to the Emerald language was built in C and the compiler was built in Emerald language. Emerald can run seamlessly on any machines (MAC, Windows, Linux etc.) but we want to extend Emerald to smartphones so that we have the possibility to distribute objects at the edge of the cloud.

Android smartphones are not a full linux as each phone has rooting privilege security making it not possible to execute any binary files at all. That is why we build an app that acts as a layer on top of Emerald to prevent us from rooting any phones at all because we can then add any privileges for the user to send commands through the app to the Emerald compiler[13].

In figure 1.1, we illustrate how the seamless computation works. Seamless means that any data can move freely everywhere without worrying to

see what is happening in the background. Seamless is similar to transparency in distributed systems where we do not need to know what is happening in the background at the users point of view[27]. Seamless makes it possible to do a Near-Far cloud modeling as inside the cloud infrastrucutre we have several different machines that could either be a near or a far cloud and at the edge of the cloud we have either an Android smartphone or a Data Center.

The advantage of a Near-Far cloud based on figure 1.1, is that we can take the computation to move closer for lower latency from the Data Center while on the other side where the smartphone is, we can take the very near cloud and move it further to access cheap and compute services that has lower latency. We want to avoid doing any computation on the smartphone because we could use a lot of battery power when computing data[34].

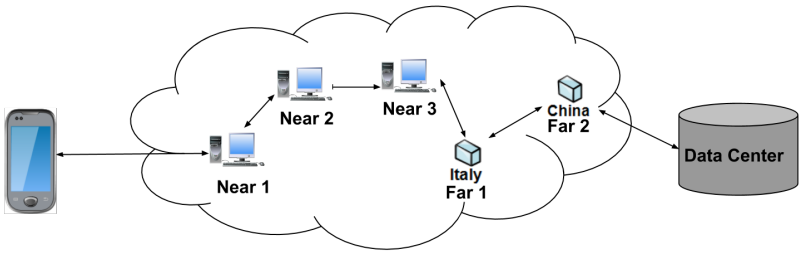

Figure 1.1: A seamless computation model

We demonstrate such model from figure 1.1 using *P2P* approach and the statement is explained on the result section.

1.2 Goal

The goal of our research to is to enable seamless computation using Emerald. Emerald does not go on to the edge and our main goal is to get it to do it.

1.3 Approach

We take an existing language and port it to smartphones, where this language is fine-grained, and enable it across the spectrum to enable mobility. Finally we evaluate how well it worked with some case studies we have done.

1.4 Work done

We have ported the Emerald compiler source to Android and measured how good it is to run any Emerald programs on the phone and distributing across the spectrum. To actually run the compiler, we have built an app which acts as a layer on top of the Emerald compiler where with this app, we demonstrate the seamlessly mobility model.

1.5 Results

We have ported the Emerald to the Android phones and have proven that the concept of near-far cloud by using Emerald as our approach.

- **Performance results:** Using Emerald to move data back and forth on the thin client from near to far cloud proved to be more efficient when having a powerful machine available nearby rather than distributing directly to the data center. The advantage is the low latency between a near cloud that can do the computation or moving data onwards to other clouds that are further away.

- **Seamless results:** Using Emerald to move data from the thin client to a near cloud for computation and storage is more efficient and gives less latency than distributing to far clouds directly. It is more optimal to have a closer server than having to ship data directly to a further cloud server.

1.6 Contributions

We have contributed on the following points:

- Porting Emerald on Android Smartphones
- Created an app that would use Emerald
- Evaluate the performance on Smartphones
- Limitations in the Emerald Implementation

1.6.1 Porting Emerald on Android Smartphones

We have ported the Emerald compiler to Android phones. However, there are many different types of CPU architecture that each smartphone has. The first step is to cross-compile the Emerald compiler to the specific architecture that the phone uses so that it can be executed on the phone. However, a major challenge when it comes to porting any executable binaries on Android is that the phone has a tight security privileges which makes it impossible to execute any binary files at all.

1.6.2 Created an app that would use Emerald

We have created an app for Android smartphones where the implementation is specialized for the Emerald compiler. The app acts as a terminal emulator where the operation command that is typed at the application level sends it to the lower level so that the binaries can be executed.

1.6.3 Evaluate the performance on Smartphones

We have evaluated the performance of distributing objects to near clouds using smartphone to be efficient. The smartphones do not need to do heavy loads of computation, but rather let a more powerful cloud server do its work and provide the result back to the phone.

1.6.4 Limitations in the Emerald Implementation

We have looked, tested and discovered limitations on the Emerald compiler on the Android phones.

1.7 Conclusion

The research we did with porting Emerald on Android phones worked as creating a Near-Far cloud model can actually be done with a programming language that supports mobility. We discovered that most smartphones have different CPU architectures that we need to consider when porting Emerald to any smartphones and also we have to build an app that can run the Emerald compiler so we avoid rooting the phones.

1.8 Outline

The thesis is divided into Chapters and at the end there is an appendix. The four first Chapters describes more of what background material we should know for this thesis.

Chapter 1: Introduction Gives a brief introduction of what we have done.

Chapter 2: Background Explains about the general concept of traditional cloud computing, Near-Far cloud model, Peer-to-Peer (P2P) model, distributed systems, why distributed systems are popular research topic and how this created the paradigms of distributed objects and the history of how smartphone became what it is today.

Chapter 3: Distributed Objects Is an overview of what is Distributed Objects and explains what Emerald Programming Language is.

Chapter 4: Android - the Smartphone OS Explains the general Android Stack, Android NDK and Google cloud messaging.

Chapter 5: Porting Emerald Covers the general steps of testing of porting Emerald into Android phones.

Chapter 6: The Application - Emerald-Lite Explains how the app is implemented and what classes are important to form the app. The most notable with our implemented app is that it is specialized for the Emerald compiler.

Chapter 7: Performance & Evaluation Covers the result from different testbeds done and if it was a good solution for each of them.

Chapter 8: Conclusion This Chapter is the conclusion of the project and the contributions to the work that has been done.

Chapter A: Future Works Explains what should be improved and what should be changed with the app or even the Emerald implementation.

Chapter B: Code Shows where you can find the source code, APK file and full version of the relevant codes that has been described from the previous Chapters.

Part II

Background

Chapter 2

Background

This Chapter describes how a Traditional cloud server is, the Near-Far cloud model, a small history of distributed systems, distributed objects, Planetlab and how smartphones became what they are today. We list up the actual smartphones that we are using for the research, and what kind of performance we can expect each phone has.

2.1 Traditional Cloud Computing

Traditional Cloud computing is where we have a thin client, which can be any machine or device that is accessing a large data center that is far away from the location of where the thin client is, like Google or Amazon[24]. Such cloud computing uses a typical client/server model where the cloud uses HTTP protocol to access the server. The main concept of the cloud computing is that from the thin client, we push an IP packet and pop it to a cloud server to handle the data containing in the packet. Afterwards the server itself pushes the result into another IP packet back to the thin client. In such model there should be no computation or storage on the thin client or the data center. For an illustration, the concept of such computing, look at figure 2.1 where we have a smartphone sending packets through the cloud to a data center where it sends the result to the thin client through the cloud infrastructure.

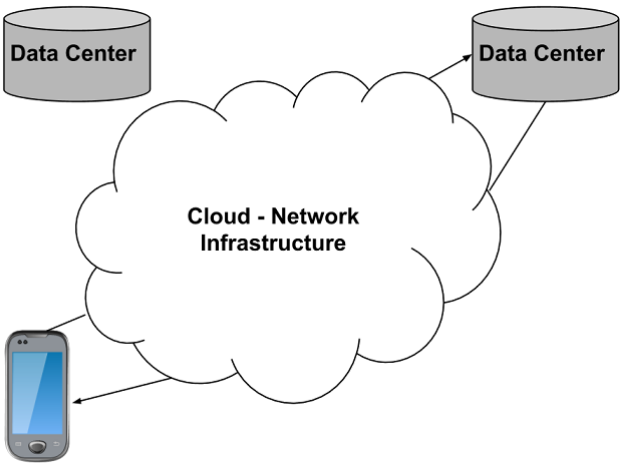

Figure 2.1: The Traditional Cloud Computing Model

The advantage is the economy of scalability as each available service is spread over the continent. The disadvantages are latency, jitter, unstable performance in the data center, and the cost of transferring data to a service that is located far away from the thin client. Such model does not have any computation or storage in between the thin client or the data center.

2.2 Near-Far Cloud Model

Near-Far cloud is a new Cloud Computing Model where there exists a device or a machine that is closer to the user that has storage and computation capabilities at the edge of the cloud. These machines are not far from the data center and the main purpose of having this Near-Far model is to avoid high latency between the data center and the thin client, so why not have a near cloud that can do the computation? Each near clouds are placed between the thin client and the data center to make sure that it is close enough to the edge of the cloud infrastructure. With such model, we have now more availability of services that are spread across the spectrum where the thin client communicates with the near cloud and the near cloud can communicate with a far cloud or the data center.

An example of a Near-Far Cloud model is on figure 2.2 where the smartphone can communicate directly to the near cloud and the near cloud can communcate to the data center.

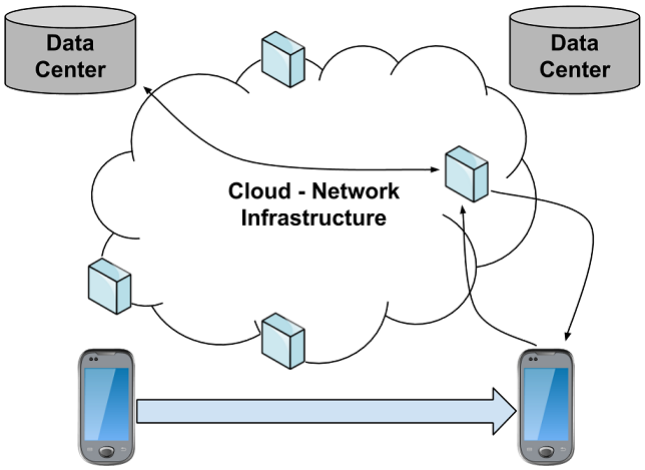

Figure 2.2: The Near-Far Cloud Model

The advantages for such cloud model are low latency where the near cloud can communicate to the data center rather than the thin client doing it directly, good performance and efficiency as the thin client do not need to use a lot of battery for the connection to the near cloud. The challenge for having such a model is that we need to consider the cost of deployment when it comes to computation and storage resources.

2.3 Peer-to-Peer Model (P2P)

For our project, we have built a *P2P* model to illustrate a Near-Far description. A peer communicates to another peer through the spectrum[6]. The communication link between each peer must be setup according to the closest to the furthest location of where they reside. We use the Emerald programming language to make the model because the language itself is specialized for mobility[17]. Emerald is a distributed object-oriented language and is much easier to program and setup than any other languages that exist, when it comes to mobility[14].

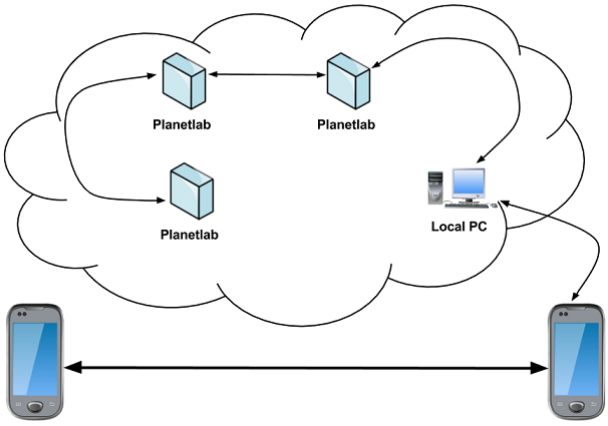

Figure 2.3: A simple P2P model

We illustrate the *P2P* model by looking at figure 2.3 where the thin client can move either programs or data across the spectrum to any near or far cloud. The near clouds are the local machines (IFI) and, while far clouds are the Planetlab hosts. Planetlab is a network testbed where we can experiment shipping data across the world.

2.4 Distributed Systems

Distributed systems is an old and most popular research topic since the late 1970s. A system that is distributed has several definitions where one of them, from George Coulouris[6], defined distributed system as a system where the hardware or the software components are located at computers that have network capability for these computers can communicate and coordinate their actions by using message passing. Our definition is that a distributed system is where you have several machines connected with each other, forming a network where each machines are independent,do their own work and have the functionality to communicate with each other or share resources.

The reason why we are so interested in distributed systems till now is because of the advantages we get from having such system integrated. Examples of distributed systems are massively multiplayer online games (MMOGs), Financial systems, Web search, health care systems, smart homes, bus/train ticket systems and many more. The goals of distributed systems are according to[1]:

- **To share resources:** Every machine must have the function to share and use resources anywhere at any place and almost at any time.
- **To be more open:** If a machine follows a standard protocol for accessing an open interface, then the open distributed system can be extended and improved.

- **To be scalable:** The system should allow to be increased in number of users without much loss of response time due to increased data traffic.
- **To tolerate failures:** Even though machines fail, the system has to make sure that it is still available and tolerate that failures occurs.
- **To be transparent:** Distributed systems are complex to learn. For the users, the system should be viewed as a single system and the application where all the technical distributed technologies are well hidden.
- **To be heterogenous:** In a distributed system, there are many different machines that have been produced by other companies, we might want to have to find a communication between these machines for example an *Apple Mac OS X* should communicate with a Packard Bell *Windows 8*.

Although distributed systems sounds like a really good idea to deploy and use, there are many challenges such system faces as unreliablility, security, heterogeneous, topology issues, latency, bandwidth, transportation cost and administration.

The concept of distributed systems, object-oriented programming language and software engineering altogether led to the motivation of creating a technology to make an object-oriented distributed programming language which is now called *Distributed objects*[6]. Distributed objects are based on objects where in a distributed system environment, each machine has the ability to send objects to another machine where it does the workload for the one that sent the object. In other words, we want the code to be processed remotely through remote procedure call (RPC) or just provide resource sharing in an easy and abstract way[7].

2.5 Planetlab

Planetlab is a collection of computers that we are using for network testbed or distributed system research. The machines are distributed all over the world where we can get an account and by having an account, we hava access to all machines and get a slice allocated. The machines are uniformed, which means if you compile on one machine then you can run on every machine in Planetlab. Planetlab has slices where each is a part of a project where it has a collection of nodes that we can use for our benchmark. The nodes can reside either in USA, France, Poland, China, New Zealand etc. A node represents a host that you can use the host by remotely logging to it. Planetlab provides distributed nodes and modeling facilities[10].

Planetlab is a global research network that supports the development of new network services. Since the beginning of 2003, more than 1,000 researchers at top academic institutions and industrial research labs have used PlanetLab to develop new technologies for distributed storage,

network mapping, *Peer-to-Peer* systems, distributed hash tables, and query processing[26].

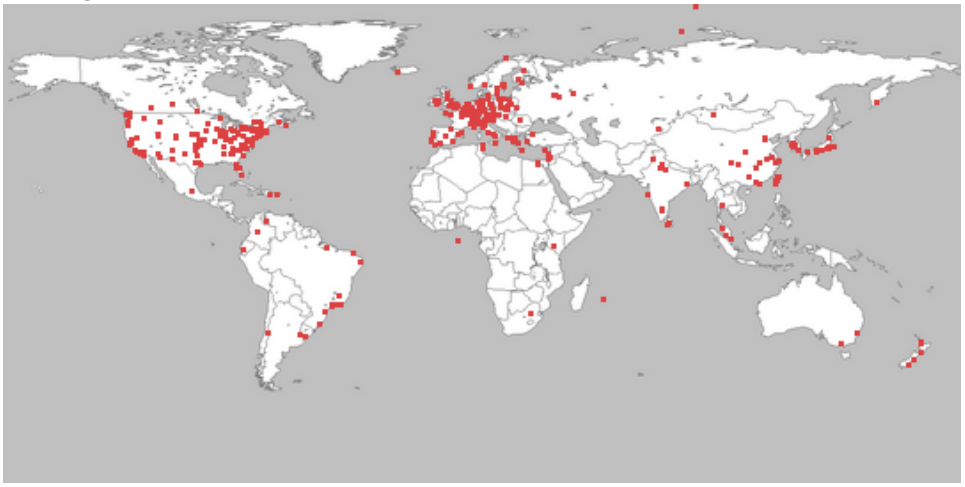

Figure 2.4: The nodes spread across the world[26]

2.6 The Smartphones

Nowadays in year 2013-2014, we are in an era where we have small mobile and ubiquitous devices that are communicating through wireless connectivity. Which means that we have now for example devices that are so called 'smart' phones, this type of phone is acting as a small pc you can have in your pocket[6].

The very first smartphone that was invented stems from the date 16th August 1994 where this phone was named International Machines Corporation (IBM) Simon. the reason why this phone was considered "smart" was that it could not only call and receive calls from other phones, but it hade the functionality of creating and sending fax, e-mails and cellular pages. Not only did it have these functionalities, the phone itself also had address book, calendar, world time clock, electronic notepad and keyboard screen like we have on our smartphones as of today[11].

Because of the concept of IBM Simon, many cellphone companies such as Nokia, Apple, etc. began to make such smartphones where each of these devices would now have an *operating system (OS)* integrated in the phones. The one that is most known and popular mobile OS is *Android*[21]. The idea behind Android was invented by Andy Rubin where the main goal was to have an operating system for cellphones according to[16]. The idea of Andy became a hit target for Google, so in 2005 they bought the Android Inc where they made Andy the head director of mobile platforms for Google[16]. In 5th November 2007, Android was now a part of the *Open Handset Alliance*. In 2008-2010 it became a more dominant mobile platform worldwide.

Android has now become the most popular mobile operating system platform where the system has now increasingly been updated and has a huge benefit for the consumer as the phone itself is strong enough to run

small applications which are now called *apps*. Some applications do not necessarily process data on the phone itself, but might send the information to the cloud server so the phone itself acts as a thin client. That is because phones might use a lot of battery consumption to get the processing of the data to be complete, so why not process the data on a more powerful server or another smartphone that does not use much battery and gets the result back? Why not distribute an object to another smartphone and just process it at that machine locally?

2.7 The Actual Smartphone Usage

The actual Android smartphones that we use are:

- Samsung Galaxy SIII : 8.11 s
- Samsung Galaxy Mini S 2 : 17.12 s
- Samsung Galaxy Fame : 12.15 s

The time shown on each phone describes the performance result which is taken from table 7.1.

2.8 Summary

We described how a typical traditional cloud computation works, Near-Far cloud, *P2P* model, distributed systems, Planetlab and smartphones. There are no computation on either the thin client or the data center. What we want to focus on is the Near-Far cloud model. We have a computation and storage facility in between or further away. To illustrate an example of such model, we proposed a *P2P* model that would explain in detail how it works. In the next Chapter, we describe more in detail the distributed objects.

Chapter 3

Distributed Objects

In this Chapter, we describe what a distributed object is, the concept of how it works and we give an introduction of the Emerald programming language.

The term object means that it encapsulates the data. The operations for extracting the data are called methods and these methods are available on an interface. An interface is a collection of methods where an object is said to invoke methods that are available on the interface. Objects may implement many interfaces and several objects may also offer an implementation to it[1].

Objects are commonly used everywhere because we want to illustrate and model the real world with a combination of the behavior and state of how this object would act in the environment. To distribute objects makes the programming easier, easier to maintain and reduces the complexity of knowing low-level in detail like in C. Where objects have a set of description (known as attributes) and the functionality of these objects is later described (known as methods). Machines and software are complex in general so that is why we use an object-oriented programming paradigm to build a simple environment of the real world for simplification.

The overall functionality of a distributed object should do each of the following steps: Referencing remote objects, provide remote interfaces, distribute actions, handling exceptions in a distributed environment and perform distributed garbage collection in the background[6].

A general model of a distributed object concept is based on the client-server model, where we can place an interface at one machine and the object resides on another machine. Figure 3.1 shows a typical model of how distribution of objects works with the general client-server model concept. The client has a proxy which is the interface it binds to. The proxy acts as a stub for the client, and the main goal is to marshal the remote method invocation into a message packet and unmarshal it when the client gets a reply which contains the result of the invocation the client made. Marshalling is a way of taking the object and convert it into a data stream which has to correspond to the same as a network packet. Unmarshalling is the opposite. It takes the data stream and tries to convert it back to an object[6].

The object resides on the server-side where it has the same interface as the client-side. The stub that is known as skeleton on the server-side does the same operation as the proxy at the client-side. The skeleton unmarshals the message that it receives and does the invocation at the object's interface. When done the skeleton marshals the reply back to the client-side.

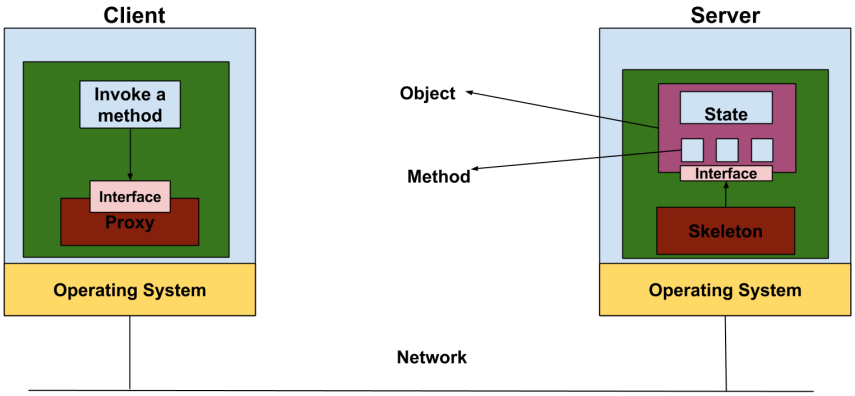

Figure 3.1: The client-server model example of how distribution of objects are set up

The motivation behind using distributed objects when we want to transmit data from a smartphone to another, is that we want to do a remote procedure call to another machine without any large latency. *Common Object Request Broker Architecture (CORBA)*[1] is one of the most known architectures for setting up a distributed object technology where there are many programming languages running on many machines, and follows an *Interface Definition Language (IDL)*[1] which describes how the objects are presented. CORBA solves a portion of the heterogeneity problem where it makes sure that it ignores the lower layer of the machines, as to communicate, it only takes for the two interfaces to look alike[6]. Setting up CORBA is really complex as we have to hard-code who we want to connect to, which representation and methods we need to use and to setup the IDL.

In this Chapter, we present a distributed object-oriented programming language which is more lightweight than CORBA. The language is called *Emerald*.

3.1 Emerald Programming Language

Emerald is a distributed object-oriented programming language that was developed by Eric Bartley Jul, Norman Hutchinson, Andrew Black and Henry Levy. The concept of Emerald was influenced by Algol, Simula and Smalltalk.

The ideas of Emerald were to[17]:

- Try to get a uniform object model for both local and distributed computation.
- Fine-grained mobility.

- Support language for mobility to achieve an efficient implementation.

Another goal of Emerald is to efficiently execute node-local. When it comes to efficiency, Emerald obtained it by[17]:

- integration of the object concept including mobility
- Relying on cooperation between the compiler and the runtime system
- Implementing all objects on each node within a single address space.

Emerald is not an enterprise programming language, this means that Emerald is not a complete programming language as it does not consideration large amount of data representations for primitives over 32 bits of data or different functions the ones Java has (import java.util.*, import java.rmi.* etc). We can do a workaround to build such functionalities and data representations however, some of the *built-in objects* do miss some functions as there is a problem to convert a *Character* to an *Integer*. This limitation does not prevent Emerald to be usable because it was meant to support mobility.

So why mobility? The reasons are[17]:

Load sharing: We have the possibility to move a process to another machine and use the resources on that machine.

Communication performance: In Emerald, there is a possibility to move one entity to a node for the duration of an ongoing interaction for reducing the communication cost. Making the object local increases performance rather than having remote communication.

Availability: The language provides a way to move data to different locations to provide availability where we can find the data elsewhere rather than just one machine.

Reliability: If a machine crashes or is under maintenance, we have replicas available data to that it can be moved to other machines.

User mobility: When the data has been moved, the program can easily migrate to the new machine it has moved to.

Utilize special capabilities: An object that is moved can use the resources available on the hardware or the software of the node it has integrated to.

3.1.1 Objects and Types in Emerald

An Emerald object has a *location* where it currently resides, may be *immutable* and usually has[17]:

- A unique network-wide identity.
- A representation which can be data local to the object.

- A set of operations. If the operations are exported, then they can be used outside of the object (think of it as *public* in Java).
- Optionally, it can include the *initially* clause which initializes variables before the object is invoked.
- Optionally, it can include the process clause which executes specific instructions in the clause.

An example of creating an Emerald object is shown in figure 3.2

```
1  const PizzaBoy <- object PizzaBoy
2    var name: String
3    var id: Integer
4    var orders: Array.of[String]
5
6    export operation GivePizza[s:String]
7
8      for i : Integer <- 0 while i < orders.upperbound + 1 by i <- i
           + 1
9          if s = orders.getElement[i] then
10           stdout.putstring["Pizza delivered to: " ||s||"\n"]
11         end if
12     end for
13   end GivePizza
14
15    initially
16     name <- "Endri Hysenaj"
17     id <- 2000
18     orders <- Array.of[String].empty
19     orders.addupper["Khiem-Kim Ho Xuan"]
20     orders.addupper["Stig Halvorsen"]
21   end initially
22
23   process
24     PizzaBoy.GivePizza["Khiem-Kim Ho Xuan"]
25     PizzaBoy.GivePizza["Stig Halvorsen"]
26   end process
27 end PizzaBoy
```

Figure 3.2: A simple Emerald object creation

The Emerald object has a *const* which is the object constructor that defines the content and behavior of an object and we need *const* to help us create and execute our object[15]. Arrays in Emerald are dynamically expandable[17]. To get the specific element, we call upon the *getElement[i]* to get the element on the index of the array. We should also note that clause is where you add items like a queue. Here you add the element, and the latest element that arrives to the array is added as the currently last element.

Types are more like an interface for the objects. This means that if every object follows every operations in a type, then the object *conforms* to the

type[17]. Conformity uses the symbol "*>" and we have to stress the point that conformity is not symmetric, but transitive and reflexive.

```
1  const Bank <- typeobject Bank
2    operation deposit[i:Integer]
3    operation withdraw-> [i:Integer]
4  end Bank
5
6  const DepositBank <- typeobject Bank
7    operation deposit[i:Integer]
8  end Bank
9
10 const WithdrawBank <- typeobject Bank
11   operation withdraw -> [i:Integer]
12 end Bank
13
14 const ConformTest <- object ConformTest
15   process
16
17     if Bank *> DepositBank then
18       stdout.putstring["Bank conforms to DepositBank\n"]
19     end if
20
21     if Bank *> WithdrawBank then
22       stdout.putstring["Bank conforms to WithdrawBank\n"]
23     end if
24   end process
25 end ConformTest
```

Figure 3.3: A type conformity example

Figure 3.3 shows that we have three types, Bank, DepositBank and WithdrawBank where we check if either DepositBank or WithdrawBank has any of the methods from the Bank. We can now say that the Bank conforms to either DepositBank or WithdrawBank, but not the opposite. If a type T is going to conform to another type S, then T must have all operations that S has and for each operation in T must correspond to the operation in S. Conformity is explained in detail by[14].

With this Type checking, we would solve the heterogeneity problem for example in a network we can think of having different objects with different meanings and if they conform to a specific interface, they can still understand and communicate with each other. We can view an object during run time using the *view as*. This might widen or narrow the operations the object has itself. Before we view the object, we have to use conformity check to confirm if it is possible to use view[17].

3.1.2 Distribution Of Objects

When we have distributed an object, we can get the objects location by using the *locate* operator which returns the current location of the object. It is

more as a reference to the node object for the node[17]. For each node objects, you can get the information about the node like the id, what host it is in or even the date/time of day of the machine it resides in.

The primitives that are used to move objects in Emerald are described in table 3.1

Table 3.1: The primitives used for mobility in Emerald

Primitive	**Usage**	**Description**
move	move X to Y	move the object X to where the object Y is
fix	fix X at Y	move the object X atomically to where Y is. After that you can not move X
unfix	unfix X	make object X mobile again after a fix
refix	refix X at Z	use unfix and then fix to the new destination Z

Figure 3.4 shows that we move the object *Kilroy* to nodes around the network and the program measures how long it took to travel to several nodes and come back again to where the object originally resided.

```
1  const Kilroy <- object Kilroy
2    process
3      const home <- locate self
4      var there :     Node
5      var startTime, diff : Time
6      var all : NodeList
7     . . .
8      for i : Integer <- 1 while i <= all.upperbound by i <- i + 1
9        there <- all[i]$theNode
10       move Kilroy to there
11       . . .
12     end for
13     move Kilroy to home
14   . . .
15   end process
16 end Kilroy
```

Figure 3.4: Kilroy program - moving itself to all Emerald nodes that are available

3.1.2.1 The Call Parameter Passing Semantics

A motivation for moving the object physically to another node and do local call on that machine is that we would reduce the number of call-backs back and forth. Figure 3.5 from [17] illustrates the parameter call-back problem where we have two nodes. The object *A* is going to do a call where the object *X* resides in another node. The parameter contains another object B that also has another operation. When doing a remote procedure call, we see that *B* still resides on node 1 and what happens is that the node 2 must do

a call-back to do operations on object *B* which is inefficient because we do not want uneccessary back and forth calls.

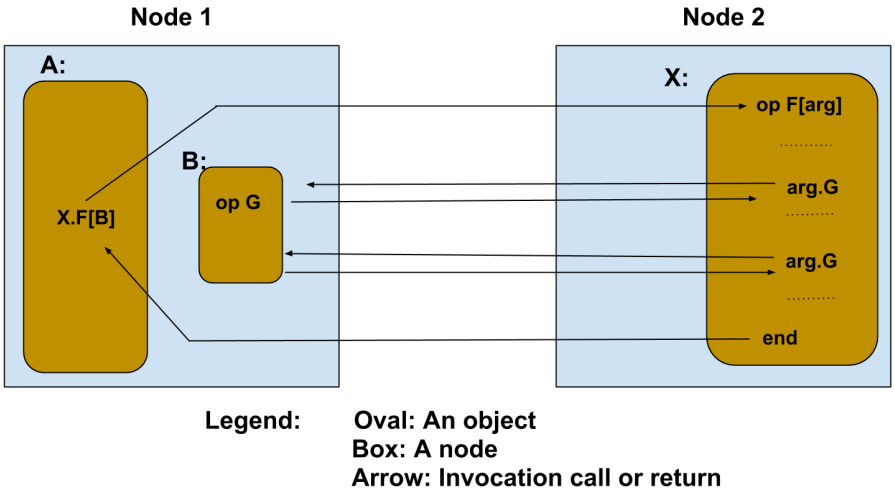

Figure 3.5: The Call-back problem[17]

The main goal of mobility in Emerald is to reduce these uneccessary call-backs. Figure 3.6 from [17] is an example to avoid such problems. The object *B* is moved along when object *A* does a remote call. This is to avoid the call-back problem where *B* is moved to node 2 as a local object. Emerald introduces two parameter passing semantics for avoiding call-back problems, *call-by-move* and *call-by-visit*[17]. *Call-by-move* the object is moved inside the parameter to the remote node that treats the object as a local. The object however does not return to the caller. If you want the object to return after moving the object, we use *call-by-visit* where the object gets moved along to the remote node and after the operation is done, the result is returned along with the object that was moved. The object just "visits" the remote machine.

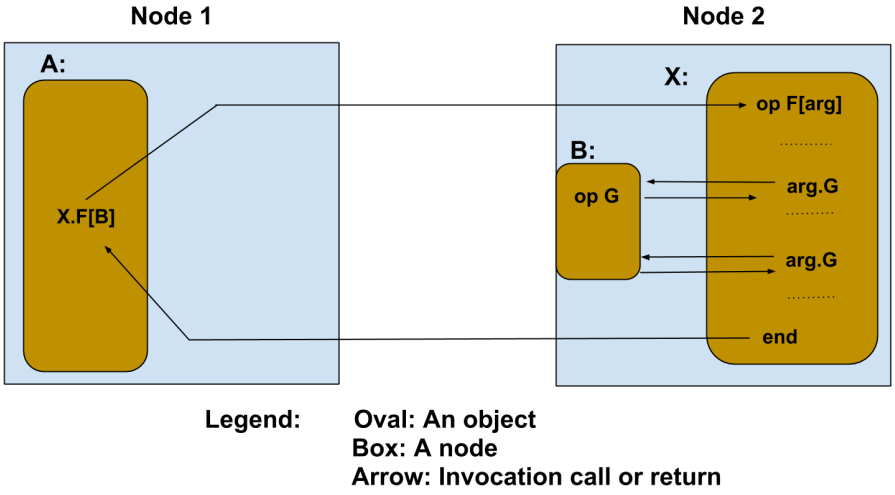

Figure 3.6: Avoiding Call-back problem[17]

3.2 Discussion

Emerald was created at the University of Washington by four PhD students back in the 80s. Emerald is still a prototype which has not been updated, but the concept is still a great advantage. In Emerald you write *move X to Y* to specify where you want to move the object to, and this node does the local procedure call. Types is to find out if an object conforms to a type or a type conforms to a type. With this functionality, we can make sure that the object can communicate with another machine that might have a different representation of the object, but follow the same type, solving the heterogeneity problem.

3.3 Summary

The focus of this Chapter is the concept of distributed objects. Distributed objects need some sort of client/server model to send objects back and forth to each other. Where we have to setup interfaces, proxies, skeleten etc. to let each of the end machines understand each other. However, using Emerald avoids such complex setup. We also look into how Emerald works, how we move objects, what semantics exist for moving objects, what is conformity, the call-back problem and why Emerald is much more different and simpler than setting up CORBA or using Java RMI.

Chapter 4

Android - the Smartphone OS

This Chapter introduces the Android OS where we discuss how Android is built up, how the activities lifecycle is and how it acts when we interact with more than one activity on our everyday smartphones. We also go into depths of how Android NDK works and a small introduction of Google Cloud Messaging (GCM)[8].
Android is a mobile OS where the technology is based on open source platform for mobile devices such as Linux distros and Java. Detailed specification of android is shown in table 4.1

Table 4.1: Table of specs for Android smartphones

Company/developer	Google, Open Handset Alliance, Android Open Source Project
Developed in	C(core), C++, Java(UI)
Operating System	Unix
Working state	Current
Source model	Open Source
Initial release	September 23th 2008
Latest stable release	4.4 KitKat/October 30th 2013
Marketing target	Smartphones, Tablets
Package manager	Google Play/APK
Supported Platforms	ARM, MIPS, x86
Kernel type	Monolithic (modified Linux kernel)
Default user interface	Graphical
License	Apache License 2.0, Linux kernel patches under GNU GPL v2
Official website	www.android.com

So what are the difference between Android SDK and Java SDK? Figure 4.1 shows what happens when we compile from Java and what happens when we compile from android. In Android, you write the same Java code, compile it to the same compiler and get class files where you recreate the class files to .dex which is our runnable files that the Dalvik VM runs[23].

Dalvik VM is covered in the next section where we describe the software stack in Android.

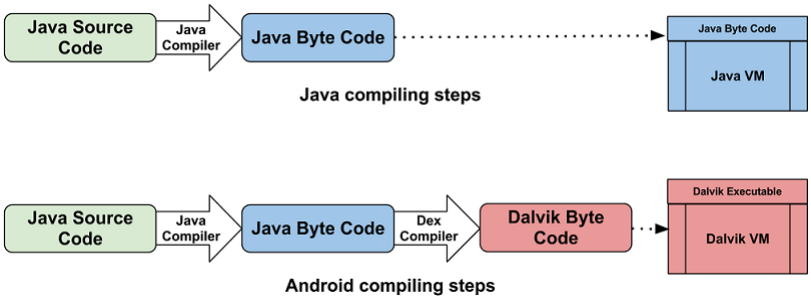

Figure 4.1: Java vs. Android

The motivation for knowing and using Android for this thesis is to find out if there is any way to distribute data objects from a near to a far cloud through smartphone devices.

4.1 Software Stack

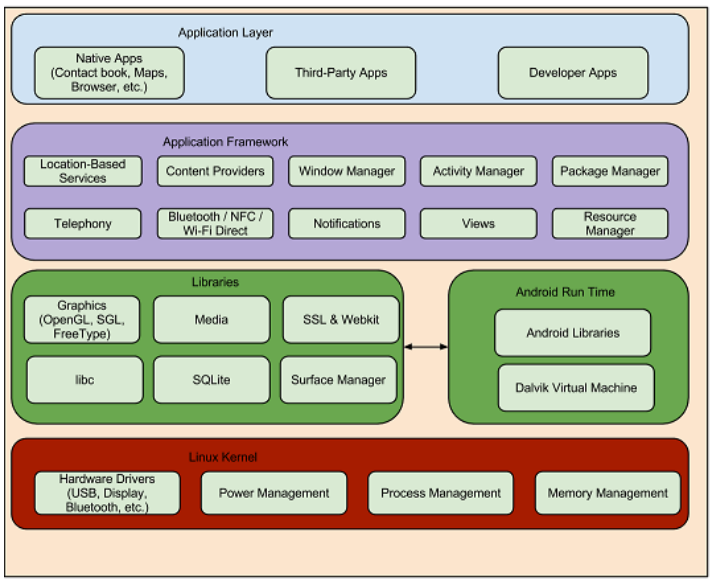

Figure 4.2: The Android Software Stack

The Android Software Stack is composed of a collection of elements that we can form into a stack[22]. Each element from the stack(from bottom to top) is *the Linux Kernel*, *Libraries*, *Android Run Time*, *Application Framework* and *Application Layer*. Figure 4.2 shows how the Android Software Stack is built.

4.1.1 Linux Kernel

The lowest part of the Android Software stack is the kernel. Figure 4.3 shows what components are available at the kernel stack. The kernel

provides the hardware services like drivers, managing memory, network, power and security as well as it provides an abstraction between the hardware and the higher level of the stack[22]. The abstraction makes sure that the developer does not know that the android is actually running on Linux and how complex it is because they only have the view of the application on the top element of the stack.

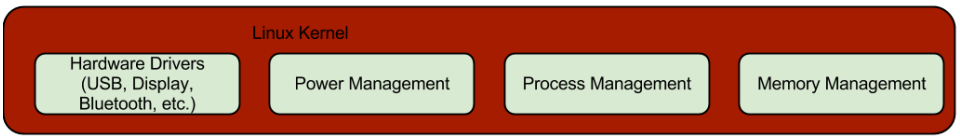

Figure 4.3: The Linux Kernel in the Android Software Stack

The main difference with the linux kernel is that when developing an app, there restrictions have been set where the user can not go and reconfigure the hardware. There is also no virtual memory swapping to the application so the kernel has a killer that reaps out and cleans the application that has been not used for awhile[36].
Another thing we should point out is that the kernel has a C library called *Bionic*[36]. *Bionic* is a license free, light-weighted GNU C library (glibc) that is more suitable to less powerful devices[23].

4.1.2 Libraries

This layer runs on top of the Linux kernel stack element. Where here, we can use C/C++ libraries to make our applications more efficient.

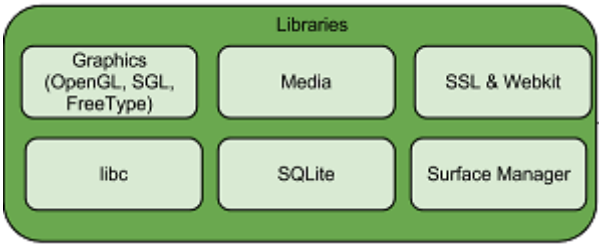

Figure 4.4: The Libraries in the Android Software Stack

4.1.3 Android runtime

The runtime layer forms the basis for the application framework and makes sure that the Android device is not a mobile versioon of the Linux kernel[9, 22].

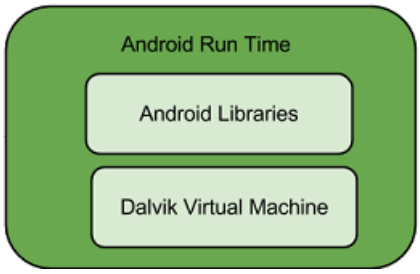

Figure 4.5: The Android run time in the Android Software Stack

4.1.3.1 Core Libraries

The core libraries gives most of the functions in the core Java libraries and also the Android specific libraries[22].

4.1.3.2 The Dalvik Virtual Machine

Even though the coding looks like Java, have similiar creation of objects, structures etc, Android does not support Java Virtual Machine (VM). It has instead the Dalvik VM where this VM is more optimized for mobile devices when it comes to battery consumption and the CPU. Dalvik is dependent on the Linux Kernel to do the threading and manage memory[22]. Dalvik has stripped off most of the libraries because of the license conflict with oracle[19].

4.1.4 Application Framework

This layer creates the classes that are needed for the Android application. This layer also provides an abstraction for accessing hardware or even manage the user interface and the application resources[22].

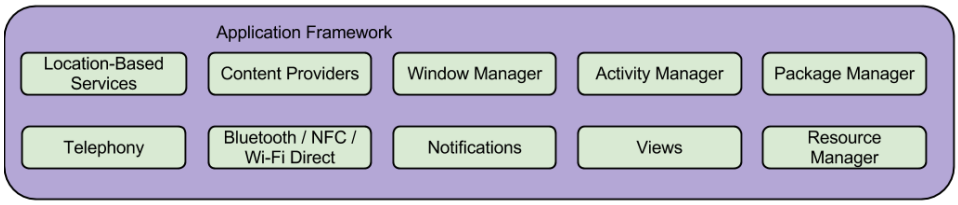

Figure 4.6: The Application Framwork in the Android Software Stack

4.1.5 Application Layer

All our application runs through this layer and uses the services and classes that are given by the application framework[22].

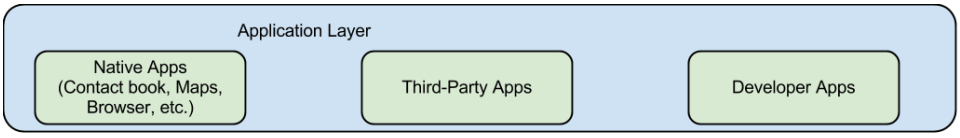

Figure 4.7: The Application Layer in the Android Software Stack

4.2 Application Components

The application components are the building blocks of an Android application[3]. An android application consist of components that are loosely coupled. They are bounded in an xml file known as the *AndroidManifest.xml*[22] The binding makes it possible for each components to communicate or invoke each other when necessary. The event that are invoked is called an *Intent*[22].
The components are *Activity*, *Service*, *Broadcast receiver* and *Content provider*.

4.2.1 Activity

The activity is the graphical user interface of the application itself[3]. An example of an activity could be *Instagram* where we have the user interface that shows us your album, pictures, videos and how many likes and comments you have on each item. An example of implementing a subclass of Activity looks like this:

```
1 public class ExampleActivity extends Activity {
2   //Code not inserted
3 }
```

Figure 4.8: Implementing an Activity class

4.2.2 Service

Service is the component that runs in the background where they either perform a long running tasks, broadcast intents or notifies events to other components. The service can also provide work for remote processes and other components can start the service to either bind or interact with it[3]. An example of a service could be when your device plays music in the background while you are interacting with another app.
An example of implementing a subclass of a Service looks like this:

```
1 public class ExampleService extends Service {
2   //Code not inserted
3 }
```

Figure 4.9: Implementing a Service class

4.2.3 Content provider

The content providers are the components that manage persistent application data between the components. The data could be stored in a file system, the SQLite database,the web, or any other persistent storage location where your application can access it. An example of a content provider is the contact information where you could read,query or edit the information of your contacts[3]. An example of implementing a subclass of a content provider looks like this:

```
1 public class ExampleContentProvider extends ContentProvider {
2   //Code not inserted
3 }
```

Figure 4.10: Implementing a ContentProvider class

4.2.4 Broadcast receiver

Broadcast receiver is the component that has the ability to send or receive broadcasted data. The announcement does not display any user interface and does not do any big work as it is only a gateway for the other components[3]. An example of a broadcast is when an application broadcasts to other applications that there might be a newly updated data that can be downloaded to their devices or even a broadcast can also be an announcement to tell your device that you battery is low.

```
1 public class ExampleBroadcastReceiver extends BroadcastReceiver {
2   //Code not inserted
3 }
```

Figure 4.11: Implementing a BroadcastReceiver class

4.2.5 Additional components

The additional components are just helping to bind and construct the four main components that has been mentioned[33]. The given additional components are:

Fragments This is just a piece of the applications user interface or behavior of the Activity

Views The user interface elements that are displayed for example, buttons, forms, textfield etc.

Layouts This gives the appearance of the view

Intents Intents are meant to bind the components together where each component can interact with each other

Resources resources are elements that are external for example, pictures, strings or context.

Manifest Manifest is the configuration file that, as mentioned above, binds the components and also specifies the hardware requirements, what platform requires to support the application or even permission setup.

4.3 The Activity Lifecycle

An activity has three states it can be on; either it is active or running, it has been paused or it has stopped.
Figure 4.12 from[2] illustrates how the life cycle looks like. When the application is active or running, it has to be in the foreground of the screen which is the top of the activity stack. When the application is paused, it has lost focus but is still visible for the user. This means that there is another activity that sits on top of the paused one and the paused activity might get killed if there is no more memory space. An application is stopped if it is obscured by another activity and is no longer visible to the user.

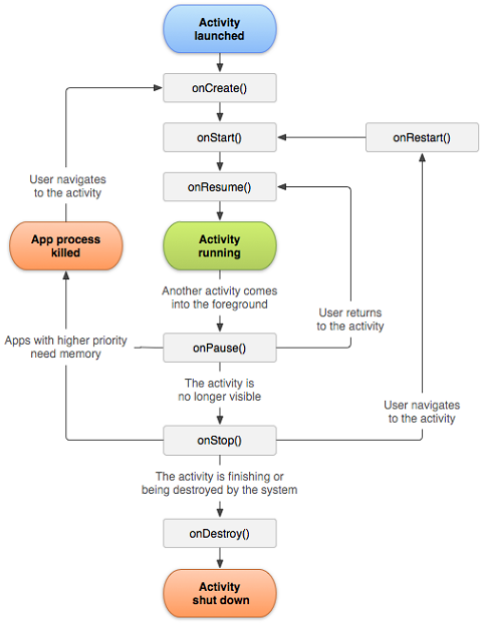

Figure 4.12: The activity lifecycle[2]

The Android OS must notify the application if there is going to go from one state to the other. The events to control an applications state changes are these methods:

OnCreate() This method is needed to create the Activity itself and is called if the activity needs to be started again.

OnStart() When the Activity has been created, the method makes sure that the Activity is about to become visible for the user.

OnResume() This is the method that makes sure the Activity is going to be visible for the user.

OnPause() The current activity is going to be paused, so another Activity is triggered.

OnStop() The Activity is no longer visible for the user, it has stopped, but is still alive in memory.

OnDestroy() The Activity is going to get destroyed. This means the activity will ble cleaned from memory and will not be restored by the system.

4.3.1 The Activity Stack

Now we just mentioned one activity lifecycle, but how does the system set up for many activities? Well, all activities are scheduled in an activity stack. Figure 4.13 illustrates what the Activity stack might look like based on[4].

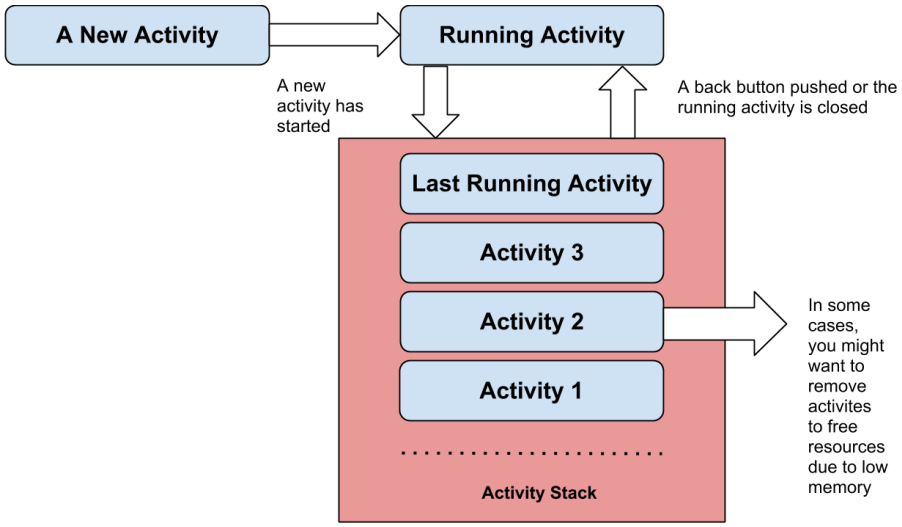

Figure 4.13: The activity stack[4]

An Activity that is *running* is usually on top of the stack while the onces that are not active are below the running activity. The current Activity that is running can be terminated if the user for example pushes the "back" button or closes the activity so the next element on the stack becomes the active one.

4.4 Android Native Development Kit (NDK)

Android NDK is about integrating C/C++ files with Java. We could call it a *hybrid* application where the Java code is run on the Dalvik VM while the C/C++ code is compiled and run directly on the hardware. The reason why a *hybrid* implemented app is feasible is because we might want to increase performance when it comes to processing large amounts of data. We need an interface to somehow to connect these two different languages together, the one that does it is the *Java Native Interface (JNI)*[20].
To illustrate the relationship, Figure 4.14 from[20] shows an example of the relationship where the *JNI* is integrated into the *Dalvik VM*, where it is possible for the *Native code* to invoke and access methods or fields in the *Java Code* and visa versa forming a two-way communication[20].

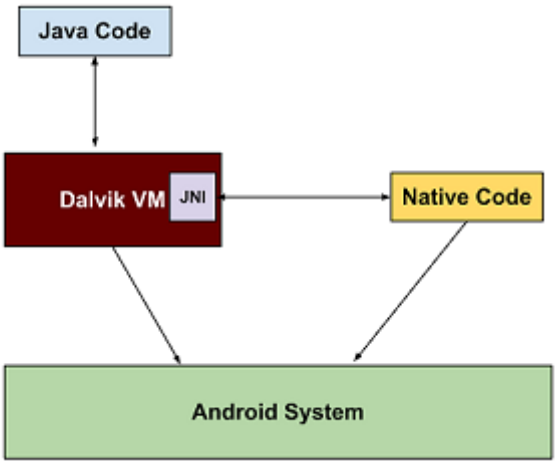

Figure 4.14: The relationship of Java and native code through JNI[20]

When creating a native application, we can classify the the application itself shown in figure 4.15[30].

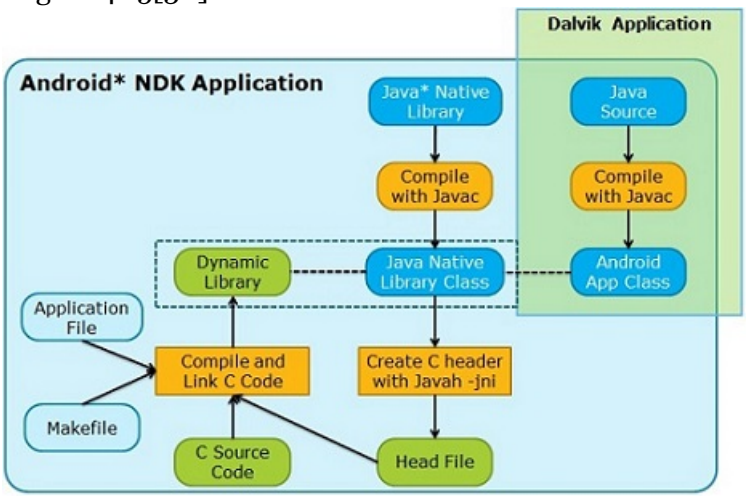

Figure 4.15: Hybrid application of Dalvik and Android NDK app, two types of Android application fused[30]

The application classifications from figure 4.15 are:

Dalvik Application Where this application does all the standard Android SDK development and creates an APK file.

Android NDK Application Where all the C/C++ or assembly codes are compiled into a dynamic link library and is used by the Dalvik Application source code through the JNI interface

When we want to run an executable binary that works on linux distros, we have to recompile again. This is known as *cross-compiling*. In Android, we have something called *standalone-toolchain* which has the ability to *cross-compile* sources to Android embedded systems. *Cross-compiling* means that you create an executable file for another platform rather than itself. We have to be cautious over what type of *Application Binary Interface (ABI)* we cross compile to. ABI defines type of CPU architecture which are[20]:

- ARM.
- Intel x86.
- MIPS.

To cross compile *C* files into an executable and push them directly to an embedded device requires that you have rooted your phone to run executable files and that you **must** not use any *gcc* at all. You have to for example, use *arm-linux-androideabi-gcc*. A detailed explanation of standalone toolchain can be found in *docs/standalone-toolchain.html* from the android-ndk directory when you have extracted it onto your pc.
To avoid permission restriction, the only way to get your executable binaries to work on non-rooted phones is to make an app and use the functions of the binary file. This is more challenging as you have to create Java threads for the user interface and the progress of the data processing on the app.

4.5 Google Cloud Messaging (GCM)

The main idea of the cloud server from Google is to send message (push message) to the cloud and either distribute the message to other phones or just receive the results back. The model of GCM follows a typical client-server model. The previous version was called *Android Cloud to Device Messaging (C2DM)*, but is now deprecated[12]. To begin communicating with the GCM server, the application client must register a Sender ID (project number) where with this ID, the client can receive the Registration ID back from the server as this ID is used for telling the GCM where to send messages to. The messages are more or less preferred to use to notify clients that an updated date is available. Figure shows 4.16 how the interaction between the phone, GCM server and our server works, each arrow shows how the interaction works stepwise.

Step 1 The client that runs the app sends an activation request

Step 2 The client receives a response with the Registration ID.

Step 3 Client sends the Registration ID to the app server where the ID is used to send data to the phone

Step 4 When the server has any data that has been updated, it sends a request to the GCM server using the Registration ID.

Step 5 The GCM server "pushes" updated notification of data to the client.

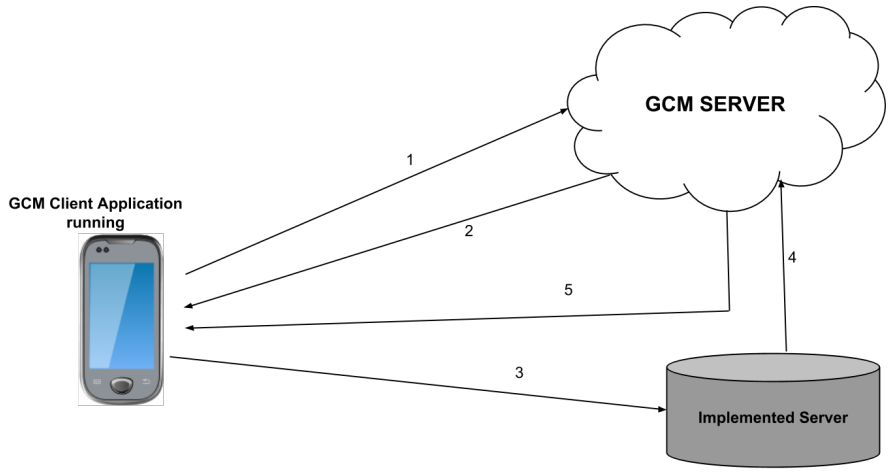

Figure 4.16: The interaction in GCM model

4.6 Discussion

it is not always a beneficial to do everything on the application. Sometimes we might want to process data directly and efficiently on the hardware. *Android NDK* gives the opportunity to use *C/C++* with the usual *Java* known as *hybrid*. For these two programming languages to communicate, there is a need for an interface known as *JNI* which makes sure that either the Java side sends calls to the C/C++ to process data, or that C/C++ side sendscommands to the Java side either requesting data or give the result output to the Java side.
Sometimes there is a need to cross compile C/C++ program so that the binary can be executed on Android OS. These are known as *standalone-toolchain*. There is a way to make an executable binary file and run it on the phone, you need to have root access to do that. That is why we have to build a *wrapper* around the executable binaries which is the app that gives the privileges which are described on the manifest xml file rather than having to root the phone.

4.7 Summary

This chapter is a brief overview of the Android OS on the smartphone where most of the topics that has been described are relevant for implementing Emerald on the smartphone. Android is based on linux for mobile devices and is developed by *Open Handset Alliance*. What was most notable is how the software stack is built up and what each element represents and how they interact with each other to prioritize the layers on top of them. The Activity components was also covered to see how each component interact and what their functionalities were when we ran our application. The Activity lifecycle and Activity stack was also covered to see how each Activity behaved during their lifecycle and how all Activities are stored and what happens when the current running Activity is terminated or paused. We also described how Android NDK works where we have some way to do a *hybrid* application of Java and C/C++ communicating with each other through an interface known as JNI. The last topic that was presented was about GCM where the most important part is how the phones can push the message up in the cloud and distribute the message to other phones or just get the result back.

Part III

Porting Emerald

Chapter 5

Porting Emerald

In this chapter, we introduce how we first came into the idea of why we have to port Emerald to our smartphones rather than just try to execute the already executable source of Emerald to begin with. We first investigate, test and make the Emerald compiler run as a service on the smartphones.
The first test was to download a *terminal emulator* on a non-rooted phone because with this app, it makes the phone act as a linux shell where we wanted to just download the emerald-0.99 version to run the commands that were *emc* which compiles the source code and emx that would execute our Emerald program. The second test was to use *standalone-toolchain* to cross compile and port the Emerald compiler sources to the Android phone and find out and conclude if it worked or not. The last test was to try to run Emerald programs on smartphones with the specific architectures that are used on the device.
However, we faced many challenges such as NAT issues, bandwidth latency, Emerald nodes not recognizing other Emerald nodes and that the Emerald nodes crashed due to segmentation fault and synchronization problems related to much simulatenous network communcation.

5.1 Running Emerald on the Terminal Emulator IDE app

Android smartphones do not usually have a terminal available, but we can use a shell by connecting it to a PC and use the *Android Debugging Bridge (adb)* to start the shell terminal on PC or just download a terminal emulator app.
The first attempt to test out Emerald on Android was to install the *Terminal Emulator IDE* which made it possible to run a terminal shell like we have in our linux distros. The reason why we use Terminal IDE rather than *adb shell* is that we can add permissions on the app so the Emerald compiler can receive packets and make sure it can listen to multiple machines. What the Terminal IDE offered was many features we have in our linux, for example vim, bash, git, gcc and even Java compilers. But unfortunately, they did not offer us root permissions. Since our smart phones are implemented in the

linux kernel, most of the functionalities where there but that depended on which type of terminal emulator we installed on our phones.

5.1.1 Results

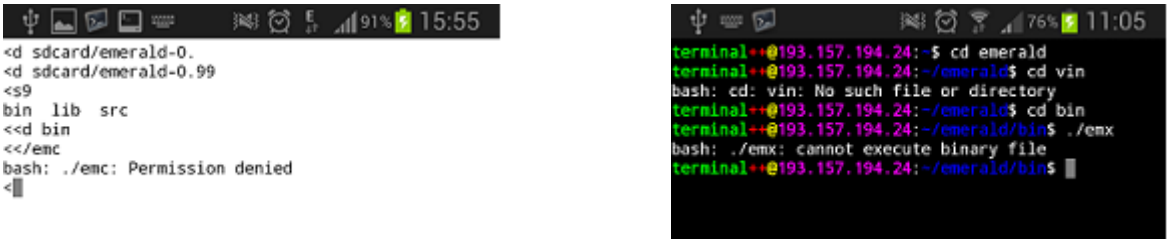

Figure 5.1: The painful limitation of our smart phones

This was a failed attempt to test the Emerald compiler by simply moving the directory containing the compiler and executable binaries. We wanted to test if it was possible to not just use the Emerald compiler as it is now rather than cross compiling.
The reason why every smartphone has set a restriction that the phone itself is because each user should not have the possibility to manage the memory, CPU etc. They might even delete the OS itself!
Figure 5.1 was several hours of testing to see if there was a way to make sure we run the Emerald without cross compiling and using the command *su* to give permission to run the Emerald compiler, or the the one that is going to execute the program. That was not successful and what we came to is that, to make our phones support and run Emerald on terminal, we need to have *root* privileges to get full access to the device itself as each device is a PC and should act as one. With the *Terminal IDE* we do not need root access to execute binary files as long as busybox is installed. However, for an Android phone to execute any binaries at all, the source code must be cross compiled into a specific CPU architecture. The most common CPU Architecture for Android is *ARM*[35]. It would be an advantage for future OS that there was an option to make the user to have root access from a developers point of view. Running the *Emerald* with just the terminal IDE app would make it easier to distribute objects around each device so that it could form a network or even pushing data to the cloud.
The advantage of rooting the phone is that you get the possibility to move apps to the sd card instead of the internal memory, this saves some space for the internal memory of the phone. Another advantage is that you can customize the *Read-Only Memory* to speed up the process of the phone or even change the user interface of the phone to your liking[25].
That does not mean that rooting would solve our problem, as we still need to re-compile the Emerald compiler to the CPU architecture of the smartphone. What that means is that we need to *cross compile* the Emerald compiler to make it executable on the Android phone. Even if we root our phones,we still get permission denied, or that the Android itself can not execute the binary file.

We developers find this annoying and not the huge benefit when we want to gain full access and do whatever we want. The only thing we can do to not brick the phone, is to make an app which acts as a layer on top for the Emerald binary resources.

5.1.2 Discussion

Having an Android Terminal which acts like a Linux shell terminal would be a huge advantage for us computer scientists as it gives us the opportunity to test different programming languages in small constraint devices such as smartphones. By cross compiling the Emerald binary file, we can still use it on the Terminal app, but not on *adb* where we need root access to execute binary files at all. Although we could modify and remove the restriction, it is not a big problem to give our smart phones root access, but that would not guarantee that our phones are in a good condition which means that it might be jailbreaked, bricked or we would break the rule of the warranty for our phones as well as there is also security issues that needs to be considered[25]. Even though we root our phones we still need to *cross compile* the Emerald compiler to make it executable on our phones. We can not just copy a binary executable file on Android because of the architecture differences and resource constraints. These specified app compilers mentioned are implemented with *Android NDK* and have used *standalone-toolchain*.

5.2 Android NDK and Standalone-toolchain

The developers that made the *Terminal IDE* and *C4Droid* where suggesting me to use *standalone-toolchain* on the Emerald compiler:

Khiem-Kim (Me): I am trying to add the Emerald compiler so that it can compile Emerald programs on Android. It seems however that I am restricted due to licensing problems as Android uses *bionic*

Spartacus Rex99: Yes, bionic definitely causes a few problems.

I would try and compile emerald with the latest NDK. Build a stand alone toolchain , shows how in the NDK docs, and use that.

it is quite powerful now, 8 incarnations later.

n0n3m4: Just use arm-linux-gnueabi toolchain (you can build it with crosstool-ng) with "-static" flag for GCC and your program builds properly

From the two app developers point of view, *Spartacus Rex99* has a point! We need to use standalone toolchain to cross compile and try running the binary executable on Android. *n0n3m4* tips was unfortunately not useful because, when using arm-linux-gnueabi, the binary file does not execute at all on Android phones. The Android OS itself has restricted us to actually use it as a fully distributed device, so in this case, we try to go "native"

and cross compile the Emerald source. Running Emerald with Terminal Emulator was a failed attempt, but we moved on and in this chapter we are going to go in depth of how we can make a "hybrid" application run a compiler and still give a relatively good user interface with slightly better performance due to the fact that we have to use native calls to retrieve results from the commands or executing a binary and display the output to the user. The most interesting about NDK is what are the limitations when we already know that Android itself does not fully support glibc, which means some of the source code of the compiler must be changed to support the specific CPU architecture the smartphone has.
We first describe how the user interface interact with the native to get information from it. Then afterwards we test out the standalone-toolchain if it went well to cross compile the Emerald source and execute the binaries on the phone.

5.2.1 The User interface for the Emerald compiler

We have to consider that we want the compiler to run when executed by normal users, and even though we have cross compiled the Emerald compiler, it can not be executed on our phones because of the restrictions which has been mentioned previously. Implementing a wrapper for the Emerald compiler such as a feasible user interface would then make it possible to use the compiler and execute programs[20].
One of the main purposes of creating a user interface that uses the Emerald compiler is that the app would then grant permissions which are specified on the *manifest* xml file. The most notable permissions that we should consider are the *CHANGE* network states, where it makes it possible to receive data from other phones.
A typical manifest file that are needed for using such an app is:

```
1   <uses-sdk
2         android:minSdkVersion="8"
3         android:targetSdkVersion="17" />
4
5    <uses-permission android:name="android.permission.INTERNET" />
6    <uses-permission android:name="android.permission.
         WRITE_EXTERNAL_STORAGE" />
7    <uses-permission android:name="android.permission.WAKE_LOCK"
         />
8    <uses-permission android:name="android.permission.
         ACCESS_SUPERUSER" />
9    <uses-permission android:name="android.permission.
         ACCESS_NETWORK_STATE" />
10   <uses-permission android:name="android.permission.
         ACCESS_WIFI_STATE" />
11   <uses-permission android:name="android.permission.
         CHANGE_WIFI_MULTICAST_STATE" />
12   <uses-permission android:name="android.permission.
         CHANGE_NETWORK_STATE"/>
13   <uses-permission android:name="android.permission.
         CHANGE_WIFI_STATE"/>
```

We need to specify what version of Android can run the app where *minSdkVersion* describes the early version we can support while *targetSdkVersion* says the highest version of the Android OS we can support, as long as any phones have an OS that is between or like the version specified, they can then run the app. We also specify typical permissions we can give the app, this makes it possible to connect to the network and retrieve the *address* used to initiate a node in Emerald. When we install the app on our phones, we create a *UI thread* where all activities are run inside this thread. A single thread inside an activity can not access UI elements and to access UI elements, we need to define a handler[20].
The application itself must have a background processing thread which can process and give the result of the execution to the *UI thread*[20].
So to illustrate the concept, look at figure 5.2 where we have the *UI/Worker thread model*. The Application layer has a *handler* that is either processing or updating messages to the user interface. The *Worker thread* is where we can execute any shell commands or binaries. The UI thread sends the given commands so that the worker thread runs the specific execution of command that is prompted from the UI. While the *Worker thread* is executing, the UI should still respond even though there is an execution in the background and when the *Worker thread* is done, it flushes out the updated data to the *handler* in the *UI thread* so it can display the result on the UI screen.

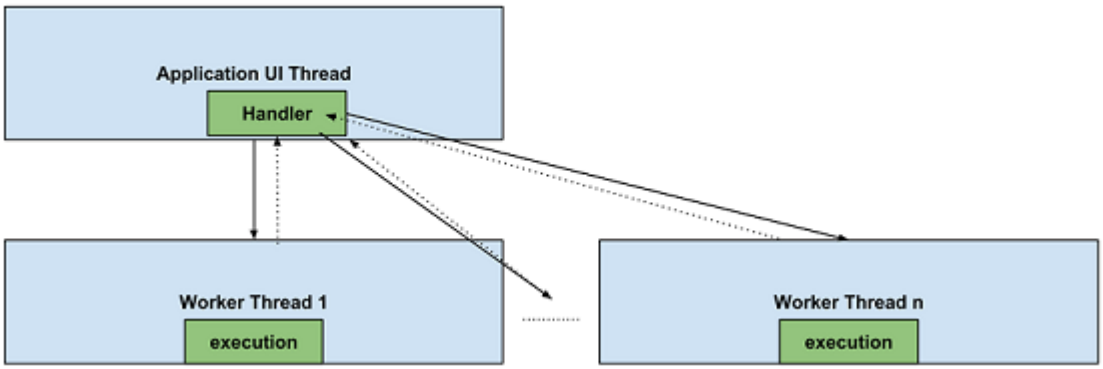

Figure 5.2: The UI/Worker thread model

5.2.1.1 Sending messages back and forth

An example of interacting with the worker thread in the background could be for example like this:

```
1       @Override
2       public void onClick(View v) {
3         if (!typeCommand.getText().toString().matches("")) {
4
5           Thread WorkerThread = new Thread(new Runnable() {
6
7             public void run() {
```

```
8                 threadMsg(execCommands(typeCommand.getText().
                      toString()));
9               }
10
11              private void threadMsg(String msg) {
12                //create a message and pass it over to the handler
13              }
14
15              public String execCommands(String command) {
16                StringBuilder buildString = new StringBuilder();
17                try {
18                  //Can use ProcessBuilder and Runtime in Java
                        library to execute commands
19                  //Best solution is to use native calls, creating
                        process and output the result
20
21                } catch (Exception e) {}
22                return buildString.toString();
23
24              }
25
26              // Define the Handler that receives messages from the
27              // thread and update the progress
28              private final Handler handler = new Handler() {
29
30                public void handleMessage(Message msg) {
31                 //Handle the message that is received and do some
                       action to it
32
33                }
34              };
35
36            });
37            // Start the worker Thread
38            WorkerThread.start();
```

What happens is that, when we type in a specific button, we create a workerthread where this thread executes the command given, create a message and pass it onwards to the handler where this handler makes sure that the output of the result is displayed to the user.
This is not so efficient and can lead to complexity in the native code[20], the best way to execute any binaries or linux commands is to go native where we have to use *p_threads* or creating processes by using *fork()* in C.
In Java, we could call *sendMessage()* method to send messages to the *handler*. When it comes to native, we need to create a method that calls from native code through the *JNI* so from JNI, the message is sent to the handler at the *UI thread*.
The UI shows only the output of the running program. The user only needs to consider which node(ip address and port number) to communicate with and what program it wants to run to distribute the object. In our app, we try to avoid that the UI "freezes" by calling native functions where these creates processes that runs the execution of Emerald or any other shell commands in the background. The process acts as a worker thread and when they are finished, the result is shown to the UI thread.

5.2.2 Cross compiling the Emerald source

First of all, using *standalone-toolchain* require us to setup the correct environment path of what kind of CPU architecture should be used. Usually, Android phones has *ARM* as their CPU. To cross compile the Emerald source, I chose to use the *arm-linux-androideabi-gcc* which is available when you download the Android NDK tools.
An example of a script that can be run on the Emerald compiler source code:

```
1  NDK=~/adt-bundle-linux-x86-20130917/android-ndk-r9/
2  SYSROOT=$NDK/platforms/android-9/arch-arm/
3  CROSS_PATH=$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/
       linux-x86/bin
4
5  export CFLAGS="-fpic -fno-builtin-memcpy -fno-builtin-memset -fno-
       builtin-memchr -fno-builtin-strlen \
6      -ffunction-sections \
7      -fsigned-char"
8
9  export RANLIB="$CROSS_PATH/arm-linux-androideabi-ranlib"
10 export CC="$CROSS_PATH/arm-linux-androideabi-gcc-4.8  --sysroot=
       $SYSROOT -static"
11 ./configure --target=arm-linux-androideabi --host=i686-linux --
       enable-static=true --prefix=/home/kkho/em_output
```

Before configuring and creating makefiles, we specify what type of CPU architecture we want to use as our compiler instead of using the classical gcc. For our app, we use the *arm-linux-androideabi-gcc-4.8* and has a path that is specified in the *CROSS_PATH* variable. Although we could also use other types of CPU architecture compilers that are available on the android ndk toolchain directory like MIPS or x86. It is also appropriate to define the sysroot so that the compiler looks for libraries and headers inside the *sysroot* directory. We also specify flags where these are actually an environment variable such that *CFLAGS* specifies the compiler flag. We can specify the location of where the binaries are stored, to do that we include the *–prefix* to indicate the location.
To run the executable binary itself (which is the first step to do to find out if it works on Android or not), we need root permissions. We rooted the emulator itself to find out if *emx* works. In Android, running a shell script does not work, so we can not just type *ec* and *emc*, we have to include *sh* before typing in *emc* or *ec* for example: *sh emc*. We also have to make sure to change the script so that it can run correctly, but we do not want the user of the app to type in *sh* each time to compile a program file. Therefore we made an ARM binary executable version of *emc*.

5.2.3 Compiling and executing Emerald files

First of all, the binary file *emx* works when porting to Android. There are no errors when executing a binary file.

After cross compiling and trying to run to compile an Emerald file, we notice that an error occured where we got a yacc stack overflow. This exception appears when the compiler complains about <EOF> of the source file, which is shown on the example above. To avoid that, we add *"-fsigned-char"* on the *CFLAGS* variable. This variable makes sure that the characters are signed. Before cross compiling the Emerald sources, the chars are represented as signed, but might be represented as unsigned on the ARM architecture[29]. The *yyps* usually starts in -1, but in ARM it starts with 255 which causes the yacc stack overflow problem.
An Emerald object does not necessarily represent the same object information but as long as it conforms to a type, it can still use the operation where each machines would understand its operation.

5.2.4 Result

Porting Emerald to Android was successful, you have to type in *./emx* to execute the binary, and to run *emc* you have to type in *sh emc*. Using *ec* does not work, the shell can not find the command *tr* from the ec script. To execute a binary on the Android phone, we need root access unless we implement an *app* which acts like a layer on top of the Emerald sources. To avoid typing "./" everytime you want to execute a binary, we modified the *Terminal IDE* where this app offers bashrc so that we can add environment paths and avoid the root permission issue. Adding a path where Emerald is would then make sure that we can simply run emx or emc (as long as both are binary files).
We have to keep in mind that in the ARM architecture they translate char as unsigned automatically, and for chars to be signed, they need to be forced as signed with the variable "-fsigned-char" so that the compiler can be used.

5.2.5 Discussion

Having a framework ready to build the app for Android is a good thing, as having UI thread run and let the user type in different buttons rather than the UI has to wait for respond of the executing commands for the Emerald. That is why we need to have a Worker thread that runs in the background and provide the execution load off from the UI and we have to implement a handler which can receive several thread messages if it was a successful operation or not.
Android NDK provides us with standalone toolchain that makes it possible to cross compile an already implemented C/C++ code so that we can port this to our phones. Our achievement of porting Emerald to Android made sure that we can execute an Emerald binary file where it is understood by the phone. Executing *emx* directly through *adb shell* requires root access which is why we need to have the app as a top layer for the Emerald binaries (*emx* and *emc*) to make the Emerald work for non rooted phones.
Porting the Emerald compiler was a success, using standalone-toolchain to port Emerald on Android works perfectly, but there had to be some changes with paths and signed vs unsigned chars. The Emerald can now be used to

run on most smartphones that has the Android but as long as they have the ARM as their CPU. You have to either root the phone and use *adb shell* to run Emerald, or you have to build a UI with permissions and utilities to run Emerald on non-rooted phones.

5.3 Summary

This chapter explains in detail of what we do to port the Emerald by first copy the sources and try to run the compiler on the phone and cross-compile the Emerald compiler and what challenges we face when it comes to recompiling it to a specific architecture. We also discovered that we have to make an app that needs to be a top layer for the Emerald compiler because any executable binaries can not be executed by any users than those that have the root privileges on the phone.
There were also major challenges such as NAT, bandwidth and latency issues as well as Emerald nodes not recognizing other Emerald nodes at all.
The next chapter describes our app, the Emerald-Lite which is specialized for compiling or running any Emerald programs.

Chapter 6

The Application - Emerald-Lite

In this chapter, we describe how the app is implemented. The app itself acts as a layer on top of the Emerald compiler where the user types in the commands at the application level and sends it to the lower level where we have the privileges to execute any C/C++ binaries. The app is an emulation of a terminal and since a phone that is not rooted can not directly execute any binaries, we have to create a terminal emulator that simulates a real terminal as in any Linux distros.
We want to make sure that any smartphones have the possibility to use Emerald to go on to the edge without rooting the device itself.
We make the code and the UML model as abstract as possible and include the method, variables and classes that are the most important to propose. The app is a modified version of the *Terminal IDE*[31] which is an open source developed app. This app offers many utilities like *.bashrc* file where we can add environment paths for where the Emerald sources reside. The apps name is *Emerald-Lite* which is just a name to indicate that this app run Emerald programs on Android phones.

6.1 Main menu

The main menu displays buttons where we either run the Terminal, install the utilities that includes the Emerald source or shutdown the program so you actually kill the Activity. The UML diagram below which is figure 6.1 describes how the Main menu acts and what occurs when you want to run the Terminal.
When the app is launched, the Start object initializes and acts as the Activity and use the same functionalities, methods, etc. like Introscreen has. Although the Start object is a subclass of Introscreen, for simplification we use the object rather than the superclass. The Introscreen object is dependent on TermService and Term object because it uses both of them to either create an Intent or start an Activity as it is an Activity itself. The *Term* object needs to create and instantiate the *TermService* and *TermViewFlipper* object where the TermService acts as a *Service*

compontent. The *TermViewFlipper* is a subclass of *ViewFlipper* where this subclass acts as an iterator for view objects. The view objects that we use for our app is the *EmulatorView*. The EmulatorView displays the text which is outputted to the screen

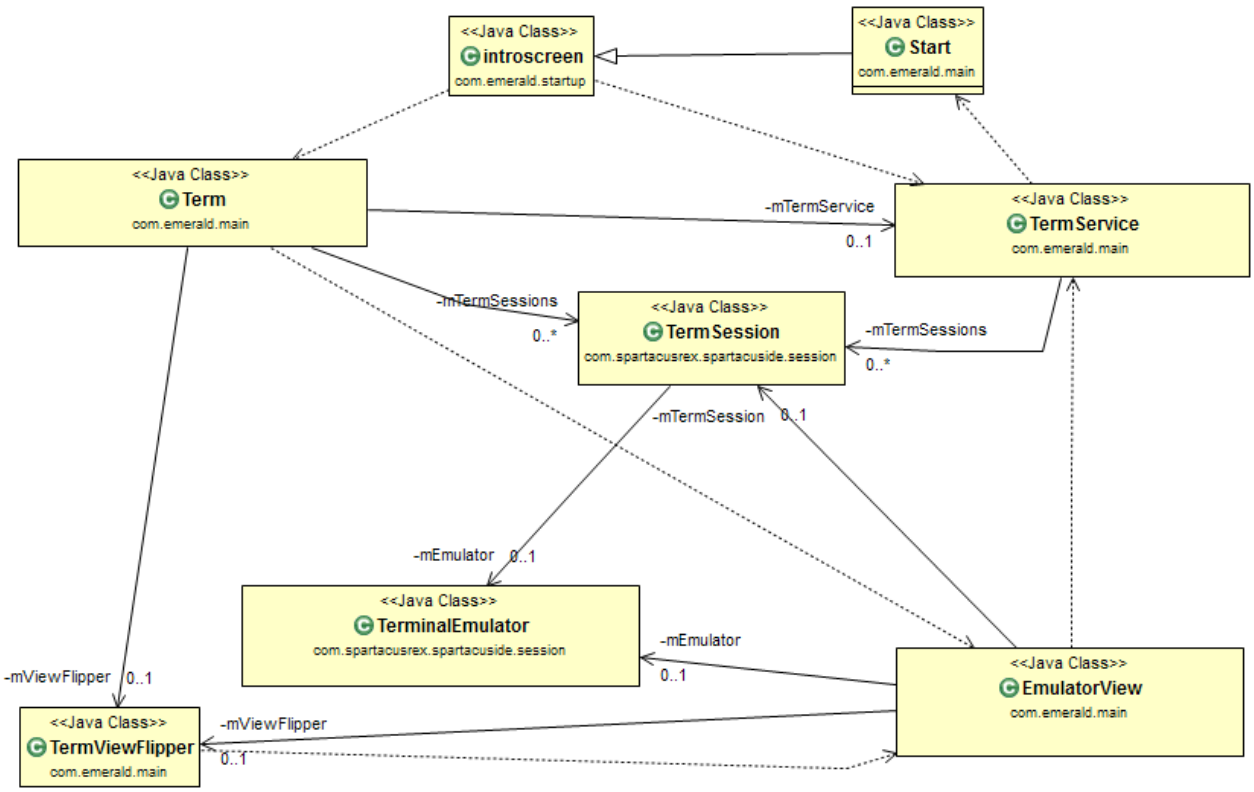

Figure 6.1: UI Main menu class diagram

and the current cursor position of the *TerminalEmulator* object which is our main function of the app to run Emerald on an emulated terminal. The TerminalEmulator object contains the most appropriate functionalities and states a typical terminal it usually has. TermService is dependent on Start object because it creates a service and makes sure that the app is running in the background with the same state as it was before. The TermService also intializes amount of TermSessions where a TermSession consist of TerminalEmulator.

6.1.1 Implementation in Android

To implement the Main menu in Android, we need to consider all the objects presented in the UML diagram plus other features which has been abstracted away. We can ignore showing how Start object is created. We are more interested to show what each object does. We introduce most of the objects from the UML diagram which is: *Introscreen*,*TermService*,*TermSession*, *Term*, *EmulatorView*, and *TerminalEmulator*.

6.1.1.1 Introscreen

The Introscreen uses the TermService as a service and the object also have buttons that is used to choose either to start the Terminal, start the installer utilties or close the app and we do all this on the onCreate() method. Since the introscreen implements an *OnClickListener()*, we define the *onClick()* on the activity itself to start the Term object.

```
1  public class Introscreen extends Activity implements
       OnClickListener {
2    . . .
3    @Override
4    public void onCreate ( Bundle icicle ) {
5      super.onCreate (icicle);
6      // Start the Service and initialize the intent for the service
7      . . .
8      }
9      . . .
10     public void onClick (View zButton ) {
11       if( zButton == findViewById (R.id.main_start)) {
12         //Check if system is installed
13       } else {
14         // Start the Terminal
15         startActivity(newIntent(Introscreen.this,Term.class));
16       }
17         . . .
18     }
19         . . .
20   }
```

6.1.2 TermService

The TermService goal is to create a Service for the Introscreen Activity where not only makes sure to run in the background, but also initiates bashscript and a list of TermSessions. When we want to "destroy" the Service, we have to make sure to clean up the Session on the *onDestroy()* method. To create a TermSession, we can define that in the *initSessions()* method that would call on the method *createTermSession()*.

```
1  public class TermService extends Service implements
       SharedPreferences.OnSharedPreferenceChangeListener {
2
3      . . .
4      @Override
5      public void onCreate( ) {
6      //initialize termsessions and put the service in the
           foreground,
7      //start a notification intent and initialized sessions at the
           internal memory path of the app
8      . . .
9      File home = getFilesDir();
10     if(home!= null) {
```

```
11        initSessions(home);
12      }
13      . . .
14      }
15
16      private void initSessions(File zHome) {
17        . . .
18        // Create initial Terminals
19      }
20        . . .
21      @Override
22      public void onDestroy() {
23        //clear out the termsessions and stop the foreground
24        . . .
25      }
26      . . .
27  }
```

6.1.3 TermSession

TermSession creates a TerminalEmulator object to actually flush out the result of the input and output to it. We use FileOutputStream and FileInputStream to make it possible to flush out. We also need to initialize a handler and threads which read any messages from process childs. To create a process, we use the methods in the Exec object which is static and native (Exec is covered in the next section).

```
1   public class TermSession {
2     private UpdateCallback mNotify;
3     private int mProcId;
4     private FileDescriptor mTermFd;
5     private FileOutputStream mTermOut;
6     private FileInputStream mTermIn;
7     private TerminalEmulator mEmulator;
8     private ByteQueue mByteQueue;
9     private byte[]mReceiveBuffer;
10    private String mHomeFilesDir;
11    private boolean mIsRunning = false;
12
13    private Handler mMsgHandler = new Handler() {
14
15      @Override
16      public void handleMessage(Message msg) {
17        if(!mIsRunning) {
18          return;
19        }
20
21        if(msg.what == NEW_INPUT) {
22          readFromProcess();
23        }
24      }
25    };
26
27    public TermSession(String zHomeFilesDir,TermSettings settings,
        UpdateCallback notify,String initialCommand ) {
```

```
28        mSettings = settings;
29        mNotify = notify;
30        mHomeFilesDir = zHomeFilesDir;
31        int[] processId = new int[1];
32
33        createSubprocess(processId);
34        mProcId = processId[0];
35        mTermOut = new FileOutputStream(mTermFd);
36        mTermIn = new FileInputStream(mTermFd);
37        mEmulator = new TerminalEmulator ( . . . );
38
39        mIsRunning = true;
40        . . .
41      }
42
43
44      public void write(String data) {
45        //flush out the terminal ourtput you get from the terminal
              file descriptor
46        . . .
47      }
48
49      private void createSubprocess(int[] processId) {
50        . . .
51        mTermFd = Exec.createSubprocess(arg0,arg1,arg2,processId);
52      }
53
54      public void updateSize (int columns , int rows) {
55        Inform the attached pty of our new size, update the emulator
              size
56      }
57      . . .
58      /* *
59      * Look for new input from the ptty , send it to the terminal
            emulator.
60      */
61      private void readFromProcess() {
62        // read inputs and flush out to the TerminalEmulator
63      }
64
65      public void finish() {
66        // clean up and exit all execprocesses etc.
67      }
68    }
```

6.1.4 Term

The Term object is an activity where we can swipe to change to another terminal, which we can define more terminals at *onCreateContextMenu()*. In the *onCreate()* method, we set the TermService as our intent and we also have a collection of TermViewFlipper so that we initiate a context view for our view flipper. To actually create more terminals, we have to base on our mTermSessions ArrayList where we add the TermSession element and the method for doing that is the *onCreateContextMenu()*. When we want to destroy the activity, we have to make sure to clean all created views on the

onDestroy() method.

```
1  public class Term extends Activity {
2      /**
3       * The ViewFlipper which holds the collection of EmulatorView
            widgets.*/
4      private TermViewFlipper mViewFlipper;
5
6      /**
7       * The name of the ViewFlipper in the resources.
8       */
9      private static final int VIEW_FLIPPER = R.id.view_flipper;
10     private ArrayList<TermSession> mTermSessions;
11     private Intent TSIntent;
12   . . .
13
14     @Override
15     public void onCreate(Bundle icicle) {
16         super.onCreate(icicle);
17   . . .
18         TSIntent = new Intent(this, TermService.class);
19   . . .
20         setContentView(R.layout.view_flipper);
21         mViewFlipper = (TermViewFlipper) findViewById(VIEW_FLIPPER
               );
22   . . .
23     }
24
25     @Override
26     public void onDestroy() {
27         super.onDestroy();
28         mViewFlipper.removeAllViews();
29     }
30   . . .
31
32     @Override
33     public void onCreateContextMenu(ContextMenu menu, View v,
           ContextMenuInfo menuInfo) {
34       super.onCreateContextMenu(menu, v, menuInfo);
35         //Show alist of windows
36         menu.setHeaderTitle("Terminals");
37         menu.add(0, 0, 0, "Terminal 1");
38        //add more terminals
39     }
40   . . .
41 }
```

6.1.5 EmulatorView

The EmulatorView gives us a possible way to show the transcript and a TerminalEmulator. Here, the most important thing is what data we receive from the TerminalEmulator which is the FileOutputStream. To receive data from any remote processes.

```
1  public class EmulatorView extends View implements GestureDetector.
       OnGestureListener {
2
3      private TermSettings mSettings;
4      private TermViewFlipper mViewFlipper;
5      private TermSession mTermSession;
6      . . .
7      /**
8       * Used to render text */
9      private TextRenderer mTextRenderer;
10
11     . . .
12     /**
13      * Our terminal emulator. We use this to get the current
            cursor position.*/
14     private TerminalEmulator mEmulator;
15     . . .
16     /**
17      * Used to receive data from the remote process. */
18     private FileOutputStream mTermOut;
19
20    . . .
21
22     /**
23      * Our message handler class. Implements a periodic callback.
            */
24     private final Handler mHandler = new Handler();
25
26     . . .
27     public EmulatorView(Context context, TermSession session,
           TermViewFlipper viewFlipper, DisplayMetrics metrics) {
28   super(context);
29         . . .
30     }
31     . . .
32             private void sendText(CharSequence text) {
33                 int n = text.length();
34                 try {
35       //map characters and send it to the remote process
36                     }
37                     mTermOut.flush();
38                 } catch (IOException e) {
39                 }
40             }
41
42       . . .
43
44     /**
45      * Call this to initialize the view.
46      *
47      * @param session The terminal session this view will be
            displaying */
48     private void initialize(TermSession session, TermViewFlipper
           viewFlipper) {
49   //initialize the value given and use session to initialize other
         global variables.
50     }
51    . . .
```

```
52 }
```

6.1.6 TerminalEmulator

The TerminalEmulator acts as a terminal in Linux. Which means that the object itself renders the text to the screen. The most important here is how we set the cursor for row and columns, how the characters are stored on the screen and how we send data to a process. With the help of FileOutputStream, we can then have the possibility to send data to the shell process.

```
1  public class TerminalEmulator {
2
3      /**
4       * The cursor row. Numbered 0..mRows-1. */
5      private int mCursorRow;
6
7      /**
8       * The cursor column. Numbered 0..mColumns-1. */
9      private int mCursorCol;
10
11     /**
12      * The number of character rows in the terminal screen. */
13     private int mRows;
14
15     /**
16      * The number of character columns in the terminal screen. */
17     private int mColumns;
18
19     /**
20      * Used to send data to the remote process. Needed to
            implement the various
21      * "report" escape sequences. */
22     private FileOutputStream mTermOut;
23
24     /**
25      * Stores the characters that appear on the screen of the
            emulated terminal. */
26     private Screen mScreen;
27     . . .
28
29     /**
30      * Construct a terminal emulator that uses the supplied screen
31      *
32      * @param screen the screen to render characters into.
33      * @param columns the number of columns to emulate
34      * @param rows the number of rows to emulate
35      * @param termOut the output file descriptor that talks to the
            pseudo-tty.
36      */
37     public TerminalEmulator(Screen screen, int columns, int rows,
38             FileOutputStream termOut) {
39         mScreen = screen;
40         mRows = rows;
41         mColumns = columns;
```

```
42          mTabStop = new boolean[mColumns];
43          mTermOut = termOut;
44    . . .
45      }
46
47      public void updateSize(int columns, int rows) {
48    //update the screen size of the terminal
49      }
50
51      public final int getCursorRow() {
52          return mCursorRow;
53      }
54
55      public final int getCursorCol() {
56          return mCursorCol;
57      }
58       . . .
59
60      /**
61       * Send data to the shell process
62       * @param data
63       */
64      private void write(byte[] data) {
65          try {
66              mTermOut.write(data);
67              mTermOut.flush();
68          } catch (IOException e) {
69              // We do not really care if the receiver is not
                    listening.
70              // We just make a best effort to answer the query.
71          }
72      }
73  }
```

6.2 Interacting C++ code through JNI

The Exec object does native calls where it has to communicate with the TermExec which is a C++ file. To do that, each of them has to go through the JNI.

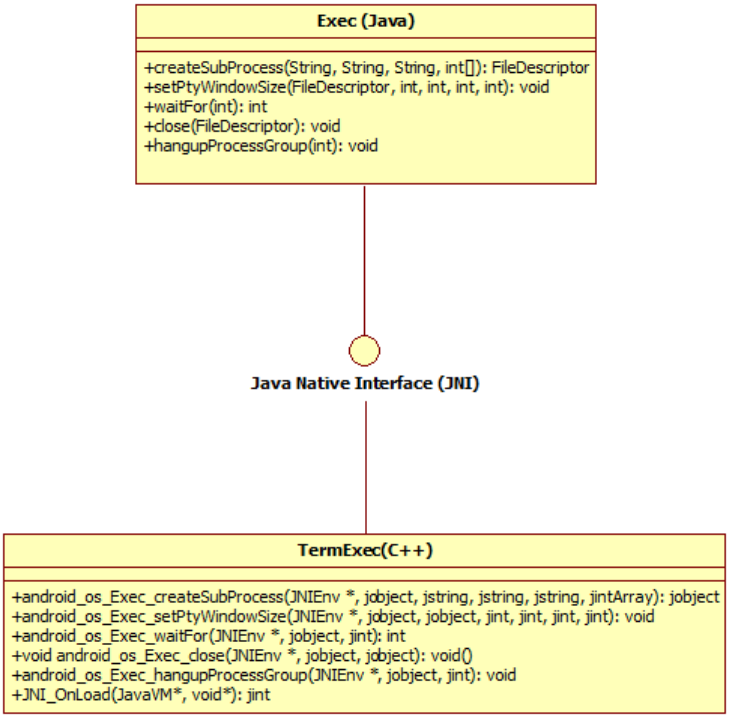

Figure 6.2: class diagram for native calls

The UML diagram on figure shows an example 6.2 of how both of them can communicate with each other. We see the methods have similar names but in the C++ file, we have to specify the path of where Exec.java lies so that both of them understand what method it is talking about. If we wrote a wrong path before the name of the method, our app would crash.

6.3 Implementation in Android

To make Exec Java class understand that it does native calls, each method must have a *native* modifier written after *public* and *static* modifier before the method name. We must also load the library within a static clause.
The code below shows how Exec looks like:

```
1 public class Exec
2 {
3     static {
4         System.loadLibrary("androidterm");
```

```
5       }
6
7       /**
8        * Create a subprocess. Differs from java.lang.ProcessBuilder
             in
9        * that a pty is used to communicate with the subprocess.
10       *
11       * @param cmd The command to execute
12       * @param arg0 The first argument to the command, may be null
13       * @param arg1 the second argument to the command, may be null
14       * @param processId A one-element array to which the process
             ID of the
15       * started process will be written.
16       * @return the file descriptor of the started process.
17       *
18       */
19      public static native FileDescriptor createSubprocess(
20          String cmd, String arg0, String arg1, int[] processId);
21
22      /**
23       * Set the widow size for a given pty. Allows programs
24       * connected to the pty learn how large their screen is.
25       */
26      public static native void setPtyWindowSize(FileDescriptor fd,
27         int row, int col, int xpixel, int ypixel);
28
29      /**
30       * Causes the calling thread to wait for the process
             associated with the
31       * receiver to finish executing.
32       *
33       * @return The exit value of the Process being waited on
34       *
35       */
36      public static native int waitFor(int processId);
37
38      /**
39       * Close a given file descriptor.
40       */
41      public static native void close(FileDescriptor fd);
42
43      /**
44       * Send SIGHUP to a process group.
45       */
46      public static native void hangupProcessGroup(int processId);
47  }
```

The most important for Exec is to load the library needed which is the *androidterm* so that the object knows that it is going to do native calls. As mentioned before, we need to define *native* before writing the whole method name.
All these methods are used to create, close or other functions for a child process.
The code below shows how TermExec looks like:

```
1  static jclass class_fileDescriptor;
2  static jfieldID field_fileDescriptor_descriptor;
3  static jmethodID method_fileDescriptor_init;
4  . . .
5
6  static jobject android_os_Exec_createSubProcess(JNIEnv *env,
       jobject clazz,
7      jstring cmd, jstring arg0, jstring arg1, jintArray
           processIdArray)
8  {
9    //create a subprocess and return a jobject. A typical creation
         is using fork()
10 }
11
12
13 static void android_os_Exec_setPtyWindowSize(JNIEnv *env, jobject
       clazz,
14     jobject fileDescriptor, jint row, jint col, jint xpixel, jint
           ypixel)
15 {
16   //set the pty windowsize
17 }
18
19 static int android_os_Exec_waitFor(JNIEnv *env, jobject clazz,jint
       procId) {
20   //return an answer if it was successful to make the process wait
         .
21 }
22
23 static void android_os_Exec_close(JNIEnv *env, jobject clazz,
       jobject fileDescriptor)
24 {
25   //retrieve the filedescriptor and close it
26 }
27
28 static void android_os_Exec_hangupProcessGroup(JNIEnv *env,
       jobject clazz,
29     jint procId) {
30     //kill the process itself
31 }
32 . . .
33
34 /*
35  * This is called by the VM when the shared library is first
       loaded. */
36  . . .
37
38 jint JNI_OnLoad(JavaVM* vm, void* reserved) {
39   . . .
40     jint result = -1;
41     JNIEnv* env = NULL;
42   . . .
43     result = JNI_VERSION_1_4;
44 }
```

In this file, we create a process, one example of creating process is using fork(). Now there are many different primitives that are presented like

jclass, *jobject jfieldID*, *jmethodID*, *JNIEnv* and *JavaVM*[20]. The most important method that is needed is the *JNIOnLoad* where this method is called by the virtual machine when the Exec java class has loaded the library (in the static clause).

6.4 Summary

This chapter, we focus on our app, the Emerald-Lite where it is a modified version of *Terminal IDE*[31]. The app is more of an emulator to virtualize a terminal as they have on Linux distros so we have now made it possible to have Emerald at the edge with the Emerald-Lite app. We avoid rooting smartphones by using Emerald-Lite.

There was also a detailed explanation of the code and how the relations are between them. The most interesting is how the Java code could interact with C/C++ code through JNI. This is needed for executing the Emerald compiler and run any Emerald programs while the output of the executions are displayed for the user on the application level.

The next chapter describes the result of different use cases where we distribute objects through Planetlab and on the Amazon server(s).

Chapter 7

Performance & Evaluation

In this chapter, we discuss what we want to measure and evaluate. As well as confirm the possibilities of having Emerald integrated into our smartphones and let it represent our thin client distributing data to near, far or edge clouds. For our evaluation, we use two different use cases implemented in Emerald:

- **Basic performance of Emerald on smartphones:** We use Emerald to do a *Round Trip Time (RTT)*, *call-by-visit* and remote procedure call.

- **Seamless Near-Far cloud evaluation:** We demonstrate how moving objects to nearby clouds can save battery life and have better computation than the thin client itself.

The basic performance has two cases, *Kilroy* and *Break-Even Point*.
The *Kilroy* which just travels around all the nodes that are available and returns home again. The second program finds out the *Break-Even Point*. *Break-Even* is the point where it is equally efficient to do a *call-by-move/visit* (thus making the parameter object local) versus having the invokee's node do call-backs to the invoker.
The reason why we use *Break-Even Point* is to see if it is more efficient to move an object from a local machine to a remote machine compared to doing a remote procedure call.
In the *Break-Even Point* benchmark, we display the time it took to do a remote procedure call and a *call-by-visit*. We are testing the same operation 5 times and using the average of those results for objects ranging from sizes 100-2,000 bytes. We do 20 remote procedure calls to see at which point *call-by-visit* and remote procedure calls would be equally efficient. We could have also used 1 remote procedure call, but *call-by-visit* would always take more time to process than doing a remote procedure call. The reason is because we have to move the object, as well as constructing the network package to the remote machine, and move it back to the original machine. While with 1 remote procedure call, we avoid the cost of packing the whole object and simply call the remote machine (requiring a small network package).

In the experiment done by Jul[17], *call-by-visit* turned out to be more efficient for objects above 1,800 bytes (with a single call-back invocation). For sizes above 10,000 bytes, *call-by-visit* seems worthwhile if there is at least one call-back per 1,600 bytes of data.
With the two example programs described, we used them to test it on four different types of distribution testbeds which are: *phone-to-phone*, *local machines*, *phone-to-Planetlab* and *phone-to-Amazon*.
When we are going to run break-even, we do not consider to run on every node, we only run the program on 2 machines which is from the *phone-to-phone*, *phone-to-local machine*, *phone-to-Planetlab* and *phone-to-Amazon*.
The *P2P model* represents our Near-Far cloud model as explained in chapter 1 where the main purpose is to see how good the computation is having either a near cloud doing computation work or the far cloud to it.
We also want to prove, that even though the far cloud is a more powerful machine than the near cloud. The near cloud is a better solution for reducing latency between the thin client to a further away server.

7.1 The Testbeds - Distributing from thin client to any possible clouds

For testing the app, we chose four different testbeds where each of them should provide stable result as each of them has been tested physically rather than simulating. Our main purpose of each testbed, is to allow our thin client to distribute any data to Near or Far clouds and find out if it was possible to do it. For evaluation and measurements, we use *Samsung Galaxy SIII*, *Samsung Galaxy Mini S 2* and *Samsung Galaxy Fame* For distributing objects to all types of clouds.

- **Phone-to-Phone:** We distribute objects to another smartphone. The motivation is to see how good the computations are for just sending data to the other smartphone.
- **Local machines:** We try out to distribute data from the smartphone to several local machines at the University of Oslo (UiO). The local machines in general represent a near cloud where we can create a near-near cloud relation.
- **Planetlab:** is a set of machines that are used for network testbed which makes it highly recommended for our benchmark to find out how well it went to distribute objects, not only to machines locally, but to several other places in Europe, Asia and America[10]. The Planetlab hosts in general represent a far cloud where we can create a near-far cloud relation.
- **Amazon EC2:** is a cloud server known as an *Infrastructure as a Service (IaaS)*[5] which gives us users virtual computers which are either used for creating webpages or just using them for

testbeds for distributed environments[28].The OS Amazon offers either Windows, Linux or FreeBSD instances depending on what we want, but for our measurements, we use Linux. We chose a free cloud server because we want to try to distribute objects where the location is in the US, Ireland, Asia Pacific and South America. The EC2 servers in general represent as a far cloud where we can create a far-far cloud relation.

There might have been several other testbeds that we could have chosen to cover. However, four types of testing should be more than enough to provide strong results if we could make our everyday smartphones to be coupled in a fully distributed way.
Each of the testbed including the smartphone has a different CPU execution time on any program. We need to find out what the benefit is for distributing objects or do local calls to several different types of machines or devices where each of them is either near or far cloud.
A simple code example written in Emerald for estimating the execution time for any machine can be done like this:

```
1  . . .
2    export operation findExecutionTime
3      const localhome <- locate self
4      var finishedTime : Time
5      var startTime: Time <- localhome.getTimeOfDay
6      var counter: Integer <- 0
7
8      for i: Integer <- 0 while i < 10000000 by i <- i + 1
9        counter <- counter + i
10     end for
11
12     finishedTime <- localhome.getTimeOfDay - startTime
13     localhome$stdout.putString["CPU Execution Time: " ||
           finishedTime.asString|| "\n"]
14
15   end findExecutionTime
16 . . .
```

Table 7.1 shows each of the machines or devices execution time based on running the code above. The time might vary based on how powerful the device/machine is.

Table 7.1: List of CPU execution benchmark of memory and CPU combination of the machine

Machines/Devices	**CPU Execution time**
Samsung Galaxy S III	8.11 s
Samsung Galaxy Nexus	9.5 s
Samsung Galaxy Mini S 2	17.12 s
Samsung Galaxy Fame	12.15 s
Localmachines at the UiO	2.16 s
Planetlab hosts	2 - 22 s
Amazon EC2 cloud servers	2.50 - 3 s

We also want to find out how fast the local and remote invocation is when we are going to attempt a *phone-to-phone*, *phone-to-local machine*, *phone-to-Planetlab* and *phone-to-Amazon*. As well as *local machine-to-local machine*, *Plantelab-to-Planetlab* and *Amazon-to-Amazon* where we want to measure how fast or efficient the mobility was compared to distributing from phone to several types of machines and from these machines to other machines.
Table 7.2 shows the attempt we made for each of the given testbeds following the call-back problem and avoidance which was described in chapter 3. Here we find the result of doing local invocation and remote invocation of distributing object. The devices and the machines that was used for table 7.2 are *Samsung Galaxy SIII*, *Samsung Galaxy Mini S 2*, local machines at the UiO, Planetlab host from Poland and Italy and Amazon server from Oregon and Virginia.

Table 7.2: List of each testbed, where we tried local and remote invocation

Testbeds	**Move + Local invocation**	**Remote invocation**
phone (Galaxy SIII)-to-phone (Mini S 2)	12 ms	19 ms
phone (Galaxy SIII)-to-local machine (UiO)	8 ms	14 ms
phone (Galaxy SIII)-to-Planetlab (Poland)	26 ms	38 ms
phone (Galaxy SIII)-to-Amazon (Virginia)	24 ms	37 ms
local machine (UiO)-to-local machine (UiO)	0.44 ms	0.98 ms
Planetlab (Poland)-to-Planetlab (Italy)	47 ms	71 ms
Amazon (Virginia)-to-Amazon (Oregon)	15 ms	24 ms

7.2 Discussion

Using near-near relation takes half milliseconds when moving the object and doing a local invocation. Planetlab and Amazon are slower due to the distance between the phone.. *phone-to-phone* takes longer time to do either a local or a remote invocation. Which means, to get to the edge of the cloud is slower. Phones are not so powerful than the local machines, Planetlab or

any Amazon servers, and uses wireless network for communication which makes it more slower to communicate with each other. It has much more complex network packing routes than any machines that was tested in table 7.2.
A simple code for testing local and remote invocation can be done like this:

```
1  const Main <- object Main
2    %ignore variables and private type and class
3    process
4      X <- A.create
5      all <- home.getActiveNodes
6      o1 <- ObjectCreation.create["Doing a remote call"]
7      o2 <- ObjectCreation.create["Doing a local call"]
8
9      %Do a remote call
10     move X to all[1]$theNode
11     startTime <- home.getTimeOfDay
12     X.doPrint[o1]
13     endTime <- (locate self).getTimeOfDay
14     stdout.putstring["Remote call took "||(endTime-startTime).
           asString||" seconds\n"]
15
16     %do a local call
17     startTime <- home.getTimeOfDay
18     move o2 to X
19     X.doPrint[o2]
20     endTime <- (locate self).getTimeOfDay
21     stdout.putstring["Local call took "||(endTime-startTime).
           asString||" seconds\n"]
22   end process
23 end Main
24 . . .
```

7.3 Basic Remote Invocation Performance

This section describes the performance of a simple remote procedure call on different machines and devices. We do not need to do from Amazon to Planetlab, from Planetlab to Amazon and from Amazon to Amazon tests. The *RTT* should be the same as doing a Planetlab to Planetlab test.
Table 7.3 shows each test where we do 1 single remote procedure call, and list up results of how long it took to do a simple remote call. We use *Samsung Galaxy SIII*, *Samsung Mini S 2*, local machines at UiO, Planetlab and Amazon cloud servers for testing and measuring the results. Near clouds are local machines, Planetlab from Poland and Italy and Amazon from Ireland. Far clouds are Planetlab from Japan and China and Amazon from Sao Paulo.

Table 7.3: List of each testbed, where we tried local and remote invocation

Testbeds	**Remote Procedure Call in seconds (s)**
phone (Galaxy SIII)-to-phone (Mini S 2)	0.144885 s
phone (Galaxy SIII)-to-local machine (UiO)	0.001979 s
phone (Galaxy SIII)-to-Planetlab (Poland)	0.038614 s
phone (Galaxy SIII)-to-Planetlab (Japan)	0.376195 s
phone (Galaxy SIII)-to-Amazon (Ireland)	0.044653 s
phone (Galaxy SIII)-to-Amazon (Sao Paulo)	0.292522 s
local machine (UiO)-to-local machine (UiO)	0:000361 s
local machine (UiO)-to-Planetlab (Poland)	0:035802 s
local machine (UiO)-to-Planetlab (Japan)	0:409746 s
local machine (UiO)-to-Amazon (Ireland)	0:042013 s
local machine (UiO)-to-Amazon (Sao Paulo)	0:270638 s
Planetlab (Poland)-to-Planetlab (Italy)	0:059384 s
Planetlab (Poland)-to-Planetlab (China)	0:420033 s
Planetlab (Japan)-to-Planetlab (China)	0.397667 s

7.3.1 Discussion

Phones are not a powerful computational device and use battery power to compute. That makes them less useful than distributing data to local machines, Planetlab and Amazon. The further away the services are, the more latency we get. Based on table 7.3, we see that doing near-far, far-far and phone-far tests is more expensive than doing a near-near and phone-near tests.
There might be a variation of the result on table 7.3. The reason is the bandwidth, packet loss, CPU and memory of the devices and machines. Bandwidth is the most essential as if there is not good bandwidth link between the location of the thin client and the near cloud, the latency might increase.

7.3.2 Great Circle Distance

The dominant factor for remote procedure call is latency. We will estimate the minimum *RTT* and calculate the *Great Circle Distance*.
Table 7.4 shows the geographically coordinated location of each device or machine. We display them in *longitude* and *latitude*. We can get the coordination by using the ip address for location.

Table 7.4: List of Longitude and Latitude for each machine or device

Device/Machine	Longitude (X)	Latitude (Y)
smartphones	10.74609	59.91273
local machines	10.74609	59.91273
Planetlab Poland	17.95	50.66667
Planetlab Japan	139.6917	35.689506
Planetlab China	116.39723	39.9075
Planetlab Italy	11.12108	46.06787
Amazon Ireland	-6.26719	53.34399
Amazon Sao Paulo	-46.63611	-23.5475

Table 7.5 shows the distance and the minimum estimated *RTT*. To estimate the round trip time, we divide the distance multiplied by 2 with the speed of light. Speed of light is 300,000 km/s, and it is not going in a great circle as the cable is not stretched out to the locations directly.

Table 7.5: List of Great Circle Distance and the estimated minimum round trip time

Location to Destination	Distance in km	Round Trip Latency
Oslo-to-Poland	1124.95 km	0.0075 s
Oslo-to-Japan	8426.56 km	0.0561 s
Oslo-to-Ireland	1265.45 km	0.0084 s
Oslo-to-Sao Paulo	10605.21 km	0.0707 s
Oslo-to-Italy	1540.91 km	0.0103 s
Poland-to-Italy	718.81 km	0.0048 s
Poland-to-China	7231.70 km	0.0482 s
Japan-to-China	2097.83 km	0.0139 s

7.4 Evaluation Criteria

For each testbed mentioned, we evaluate if it was a wise choice to use Emerald on each of them. What we are going to evaulate is:

Works We want to see if it was possible to use Emerald to distribute objects to a specified device or host. Which means we want to evaluate if we could make our everyday phones coupled.

Consistent The criteria here describes how stable the network is when it comes to distributing data around the network, there might be a slightl chance that a node crashes and that is why this criteria must be evaluated.

Efficient Here is where we evaluate how fast the response we get from distributing an object, doing local calls on the other machine and

getting the result back or just distribute an object and how long it took to return home again.

Useful We find out if using Emerald on smartphones was useful when distributing data to different devices or host which gives a different opinion based on the result we get.

Limitation and issues Even though the distribution works fine, there might be issues where each node might not recognize the node that is executing the program or the communication takes long time due to the distance from each host or server which makes it less useful to use Emerald.

Each criteria is specified in a table with the testbed. The most interesting criteria is the *Efficiency*, *Useful* and *Limitation and issues* where we really discover the usage of having Emerald on smartphones. But most of the criterias are relevant to fully display the result whether it provided sufficient result or not.
We also display the results and time measurements from using Kilroy and Break Even programs where we want to find out how long it took to distribute an object around the Emerald nodes and the difference between doing a remote procedure call or *call-by-visit*.

7.5 Phone-to-Phone

The goal for this section is to see how efficient it is to compute, store or migrate data to another phone.
Figure 7.1 shows a typical example of a a phone distributing object(s) to another phone.

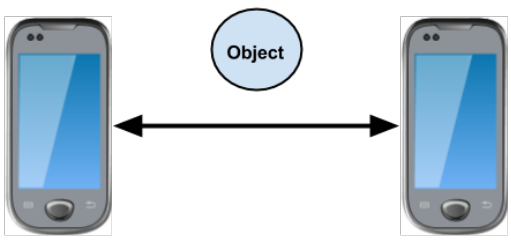

Figure 7.1: Two smartphones sending objects to each other

7.5.1 Results

Criteria	Phone-to-Phone	Comments
Works	Yes, sort of	You can only distribute objects to only one phone and the port must be opened
Consistent	No	You can only run a program at a time.
Efficient	Depends	You get the objects efficient and fast. The computation might not be so good .
Useful	Yes	You can distribute to a phone like a *Peer-to-Peer*.
Limitaion and issues	Yes	We are limited to distribute objects to one phone.

Table 7.6: Result if it was a good solution to distribute objects to another smartphone based on the criterias

7.5.2 Discussion

Distributing objects to another smartphone is really fast, but the computation might take longer time because of the CPU on the phone. The result depends on how powerful each smartphone is, like doing a break even would have a better result if the *Samsung Galaxy SIII* was the one that did the computation rather than *Samsung Galaxy Mini 2*.

7.5.3 Distribution measurements

The results below was tested using *Samsung Galaxy SIII*, *Samsung Galaxy Fame* and *Samsung Galaxy Mini 2* to distribute in objects in both ways. Kilroy was tested with all three phones while break-even point was tested with Galaxy SIII and Galaxy Mini 2.

Kilroy	
Communication	**RTT**
SIII - to - Mini 2	21 ms
SIII - to - Fame	17 ms
Mini 2 - to - SIII	12 ms
Mini 2 - to - Fame	15 ms
Fame - to - SIII	15 ms
Fame - to - Mini 2	20 ms

Table 7.7: Running the Kilroy program on phones and check how long it took to get back

There might be a variation of the result on table 7.7 and table 7.8. The reason is the bandwidth, packet loss, CPU and memory of the phone. Bandwidth is the most essential as if there is not good bandwidth link between the location of the thin client and the near cloud, the latency might increase. We could have also used 1 remote procedure call, but *call-by-visit* would always take more time to process than doing a remote procedure call. The reason is because we have to move the object, as well as constructing the network package to the remote machine, and move it back to the original machine. While with 1 remote procedure call, we avoid the cost of packing the whole object and simply call the remote machine (requiring a small network package).

Break-Even Point in seconds (s), Phone-to-Phone		
	Remote Procedure Calls	**Call-by-Visit**
Amount of Calls	**20**	**20**
100 Bytes	0.088 s	0.019 s
500 Bytes	0.086 s	0.014 s
1000 Bytes	0.085 s	0.013 s
2000 Bytes	0.453 s	0.083 s

Table 7.8: Doing Break Even Point on Phone-to-Phone

7.6 Localmachines

We do not want to make the thin client to only communicate with themselves, we want them to distribute objects to local machines that also act as nodes. Having Emerald ported on the smartphone and distribute objects across a spectrum of different devices, servers or machines is a postitive step to solve heterogeneity as long as each of them acts as an Emerald node so they understand the object that is getting transferred.

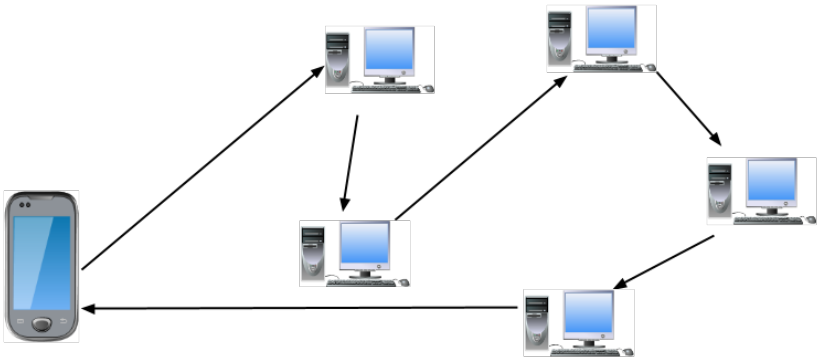

Figure 7.2: Smartphone distributing objects to a local machine where this machine passes it onwards to other machines

7.6.1 Results

Table 7.9: Result if it was a good solution to distribute objects from phone to local machines based on the criterias

Criteria	Localmachines	Comments
Works	Yes	All the machines works fine. No problems occurs.
Consistent	Yes	Each machine does not crash so easily.
Efficient	Yes	Distributing objects to machines locally goes extremely fast.
Useful	Yes	Object distribution to several machines efficiently is useful.
Limitaion and issues	Depends	Nodes might crash due to hardware faults or unstable communication.

7.6.2 Discussion

To let several phones distribute to many local machines works. The local machines recognizes the collection of nodes and continues distributing the object to the other machines.
Distributing objects to the local machines is fast and efficient, we get a quick response and we can have as many machines as we can to act as an Emerald node. However, the problem lies on the network, there might be an occasion where you are in a wireless network area where a firewall has been setup to prevent any phones to receive data at all. Which means, the local machines might have problems not finding an initiated Emerald

node on the phones if the machines are from a network with a tight security (closed ports, firewall etc.).

7.6.3 Distribution Measurements

The results below were tested using *Samsung Galaxy SIII* to distribute to several machines at the UiO.

Table 7.10: Running the Kilroy program on local machine hosts and check how long it took to get back

Kilroy	
Communication	**RTT**
Phone-to-LocalMachines	**6 active nodes:** 0.016 s
LocalMachines-to-LocalMachines	**6 active nodes:** 0.006 s

We see that there is a variation on the time in table 7.10 and table 7.11 where the reason is the bandwidth, packet loss, location of the local machines and the thin client or caching of data. We could have also used 1 remote procedure call, but *call-by-visit* would always take more time to process than doing a remote procedure call. The reason is because we have to move the object, as well as constructing the network package to the remote machine, and move it back to the original machine. While with 1 remote procedure call, we avoid the cost of packing the whole object and simply call the remote machine (requiring a small network package).

Table 7.11: Doing Break Even Point on different local machines

Break-Even Point in seconds (s), ifi.uio.no machines		
	Remote Procedure Calls	**Call-by-Visit**
Amount of Calls	**20**	**20**
<local machine>.ifi.uio.no		
100 Bytes	0.0529 s	0.010 s
500 Bytes	0.048 s	0.007 s
1000 Bytes	0.048 s	0.008 s
2000 Bytes	0.0377 s	0.012 s

7.7 Planetlab

We use these Planetlab nodes for our benchmark:

1. pandora.we.po.opole.pl (Poland)
2. planetlab2.science.unitn.it (Italy)
3. pl1.pku.edu.cn (China)

4. planetlab-01.kusa.ac.jp (Japan)

5. planetlab1.cs.cornell.edu (US)

Each node initiate and listens so they can understand and receive objects when the phone is distributing data. The nodes also distributes to one another and we see how fast they send and receive objects. We first run Kilroy from the phone to all the nodes and then use break-even to run on one of the hosts at a time. The Emerald nodes are formed from the closest country to the furthest so we can find out how much time it takes to do *RTT* for transferring an object in a single link for each closest country to the furthest. The arrows in figure 7.3 represents the *RTT* for transferring an object forward and back on each country from the phone or to each clouds.

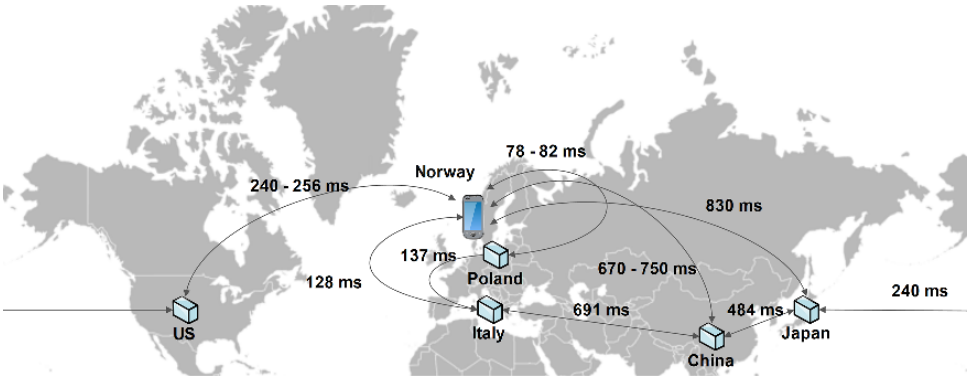

Figure 7.3: Smartphone distributing object to several Planetlab hosts and these host also distribute objects on other hosts that is closest[32]

7.7.1 Results

Table 7.12: Result if it was a good solution to distribute objects from phone to Planetlab host(s) based on the criterias

Criteria	Planetlab	Comments
Works	Yes	Every node recognizes each other and we can distribute to how many we want.
Consistent	Yes	Each Emerald node does not easily crash on Planetlab.
Efficient	Depends	It depends on the location of the host and how many nodes we are communicating with.
Useful	Yes	We can distribute and do local calls on objects to several places fast.
Limitaion and issues	Yes	Depending on the network bandwidth and latency.

7.7.2 Discussion

Distributing to several Planetlab hosts works where each and every host can act as an Emerald node and communicate with each other. We should take notice that a host can be unavailable at a given time due to update or maintenance. For a phone to distribute to many Planetlab hosts is efficient and fast but it also depends on the number of nodes we communicate with and how far the location is on each host.
It is very useful for a phone to push its workload to any hosts and let it process the work and get the result back from a specific location, but the limitations are the network bandwidth and the latency. However, the Planetlab hosts can not communicate with any phones at all if the ports in the wireless network you are in are closed.
In general, distributing objects from a phone to several Planetlab hosts is useful if you want anyone in the world to receive the object or do a local call.

7.7.3 Distribution Measurements

The results below was tested using *Samsung Galaxy SIII* to distribute to a Planetlab host, and this host distributes to the nearest country to measure the *RTT*. The network link between the countries are individually linked from the closest to the furthest country. This is to make sure that Kilroy visits each node from the nearest countries first.

Table 7.13: Doing Kilroy from phone to Planetlab hosts where Kilroy travels to each individual link and return back to the phone

Kilroy	
Communication	**RTT**
phone-to-Planetlabs	*Global visiting (6 active nodes):* 0.65 - 1.46 s

Table 7.14: Doing Kilroy from phone to each Planetlab host and check how long it took to get back.

Kilroy	
Planetlab host	**RTT**
pandora.we.po.opole.pl	0.078 - 0.082 s
planetlab2.science.unitn.it	0.13 - 0.16 s
pl1.pku.edu.cn	0.67 - 0.75 s
planetlab-01.kusa.ac.jp	0.83 s
planetlab1.cs.cornell.edu	0.24 - 0.26 s

We see that there is a variation on the time in table 7.13, table 7.14 and table 7.15 where the reason is the bandwidth, packet loss, location of the servers and the thin client or caching of data. We could have also used 1 remote procedure call, but *call-by-visit* would always take more time to process than doing a remote procedure call. The reason is because we have to move the object, as well as constructing the network package to the remote machine, and move it back to the original machine. While with 1 remote procedure call, we avoid the cost of packing the whole object and simply call the remote machine (requiring a small network package)

Table 7.15: Doing Break Even Point on each Planetlab hosts

Break-Even Point in seconds (s), Planetlab hosts		
	Remote Procedure Calls	**Call-by-visit**
Amount of Calls	**20**	**20**
pandora.we.po.opole.pl		
100 Bytes	0.865 s	0.137 s
500 Bytes	0.874 s	0.1438 s
1000 Bytes	0.881 s	0.1322 s
2000 Bytes	0.854 s	0.147 s
planetlab1.science.unitn.it		
100 Bytes	1.353 s	0.173 s
500 Bytes	1.188 s	0.178 s
1000 Bytes	1.159 s	0.263 s
2000 Bytes	1.211 s	0.176 s
pl1.pku.edu.cn		
100 Bytes	12.937 s	2.024 s
500 Bytes	16.586 s	3.127 s
1000 Bytes	14.558 s	2.522 s
2000 Bytes	13.239 s	1.882 s
planetlab-01.kusa.ac.jp		
100 Bytes	8.341 s	1.189 s
500 Bytes	8.752 s	1.336 s
1000 Bytes	8.565 s	1.156 s
2000 Bytes	8.460 s	1.252 s
planetlab1.cs.cornell.edu		
100 Bytes	2.659 s	0.4322 s
500 Bytes	2.667 s	0.438 s
1000 Bytes	2.652 s	0.434 s
2000 Bytes	2.655 s	0.415 s

7.8 Amazon EC2 server

When we have created instances for the cloud server, we need to set security privileges(open ports, open to receive any packets, etc.) where the server can receive data from other sources. If we do not do that, the server blocks any attempt for our phones to communicate with it.
When all setup is done, we can type in *ssh* to login to the server remotely, install the Emerald compiler (the original source, not the cross compiled) and initiate a node.
For this testbed, we use instances from different places in the world and each instance acts as a node:

- Virginia
- Oregon
- California
- Ireland
- Tokyo
- Sao Paulo

We first run Kilroy from the phone to all the nodes and then use break-even to run on one of the servers at a time. Figure 7.4 illustrates our phone distributing objects to several cloud servers at a time. The arrows in figure 7.4 represents the *RTT* for transferring an object forward and back on each country from the phone or to each clouds.

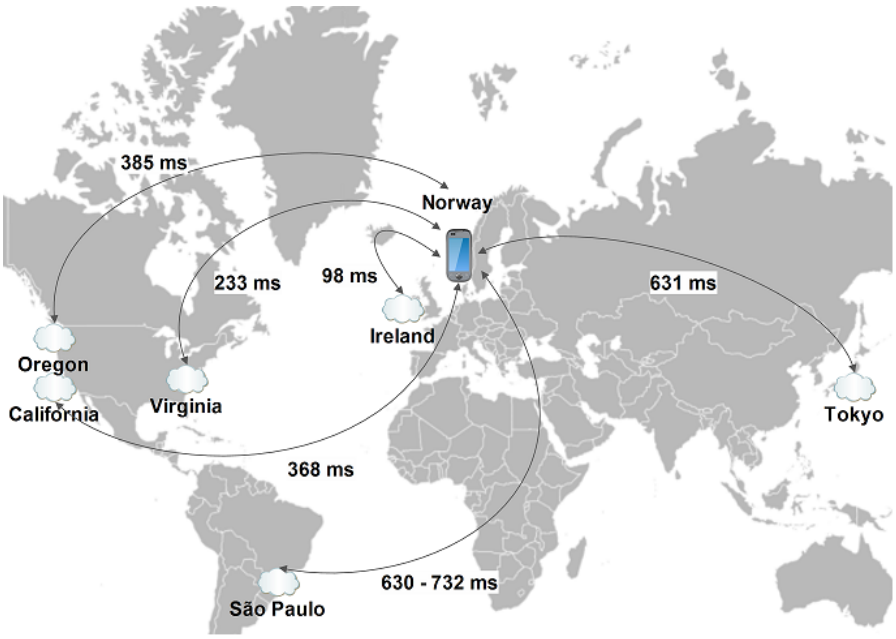

Figure 7.4: Smartphone distributes object to Virginia EC2 and it passes onwards to the rest of the EC2 cloud servers[32]

7.8.1 Results

Table 7.16: Result if it was a good solution to distribute objects from phone to Amaxon EC2 cloud server based on the criterias

Criteria	**Amazon EC2 cloud server**	**Comments**
Works	Depends	The limit of distributing an object to servers is 1.
Consistent	Yes	The servers are up and you terminate it manually.
Efficient	Depends	Phone distributing to only one EC2 server is efficient.
Useful	Depends	If you want to distribute to only one cloud server then yes.
Limitaion and issues	Yes	Firewall and port issues with the smarphones and location of hosts, servers has node recognizition problems.

7.8.2 Discussion

Distributing objects to Amazon EC2 cloud servers is not sufficient. The only usefuleness is when you only communicate with 1 server using only one phone. When it comes to efficiency, it depends on the thin clients location and where in the world the far cloud is. We are limited to test on only one cloud server so we can not distribute the object around the world as we did with Planetlab hosts.
The most hopeless case here is to get each of the servers to listen to the first Emerald node that initiated. It takes a lot of time until everyone begins to initialize and generate a port due to the location of each server for example, an Emerald node in Tokyo takes some time to contact another Emerald node in Ireland.
In general, this is not a good solution for distributing objects to many cloud servers as for some reason the nodes do not know about the phone at all or any other cloud servers than the ones that it is listening to and itself.

7.8.3 Distribution Measurements

The results below was tested using *Samsung Galaxy SIII* to distribute objects to several Amazon EC2 servers from USA, Ireland, Brazil and Japan.

Table 7.17: Doing Kilroy on each server and check how long it took to get back

Kilroy	
EC2 Server	**RTT**
Tokyo	0.63 s
Virginia	0.23 s
Oregon	0.38 s
California	0.37 s
Ireland	0.098 s
Sao Paulo	0.63 - 0.73 s

There might have been some variation on the time in table 7.17 table 7.18 where the reason is the bandwidth, packet loss, location of the EC2 servers and the thin client or caching of data. We could have also used 1 remote procedure call, but *call-by-visit* would always take more time to process than doing a remote procedure call. The reason is because we have to move the object, as well as constructing the network package to the remote machine, and move it back to the original machine. While with 1 remote procedure call, we avoid the cost of packing the whole object and simply call the remote machine (requiring a small network package).

Break-Even Point in seconds (s), Amazon EC2 Cloud Servers		
	Remote Procedure Calls	**Call-by-visit**
Amount of Calls	**20**	**20**
Tokyo		
100 Bytes	6.840 s	1.114 s
500 Bytes	6.746 s	1.088 s
1000 Bytes	6.828 s	1.010 s
2000 Bytes	6.831 s	1.088 s
Virginia		
100 Bytes	2.400 s	0.398 s
500 Bytes	2.390 s	0.363 s
1000 Bytes	2.396 s	0.363 s
2000 Bytes	2.386 s	0.366 s
Oregon		
100 Bytes	4.34 s	0.799 s
500 Bytes	4.33 s	0.665 s
1000 Bytes	4.333 s	0.658 s
2000 Bytes	4.319 s	0.663 s
California		
100 Bytes	3.894 s	0.654 s
500 Bytes	3.865 s	0.599 s
1000 Bytes	3.958 s	0.586 s
2000 Bytes	3.862 s	0.608 s
Ireland		
100 Bytes	0.970 s	0.155 s
500 Bytes	0.996 s	0.147 s
1000 Bytes	0.941 s	0.152 s
2000 Bytes	0.957 s	0.154 s
Sao Paulo		
100 Bytes	6.714 s	1.056 s
500 Bytes	6.609 s	1.105 s
1000 Bytes	6.739 s	1.003 s
2000 Bytes	6.713 s	1.08 s

Table 7.18: Doing Break Even Point on each Amazon EC2 servers

7.9 Seamless Near-Far Cloud Evaluation

In this section, we describe the seamless near-far cloud model where we use Emerald to do seamless computation and storage. We chose to describe the model at the end because we want to test and measure out different testbeds to see what we can use as for our model and the limitations each machine or device has when using Emerald to distribute data.
For this case model, we do as figure 2.3 where our thin client is our smartphones (the thin clients are our edge clouds), the near clouds are the local machines at the UiO and the far clouds are the Planetlab hosts. We build such a model to prove that mobility of either storing or computing data can be a benefit when it comes to performance.
For simplicity, we setup the connection from our thin client to 3 Near clouds and 3 Far clouds (the nearest to the furthest away based on the distance) We can illustrate a more detailed example from figure 7.5 below where local machines are the Near clouds while the Planetlab hosts are the Far clouds:

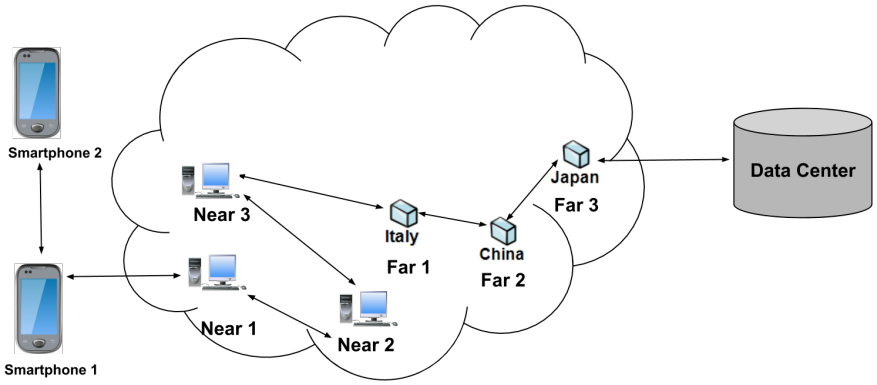

Figure 7.5: Near-Far cloud model, each machines inside the clouds are either Near or Far, depending on the distance

With Emerald-Lite, we can have several terminals where one can represent the communication between several Near-Far Clouds while the other terminal can represent the two-way communication to another smartphone that represents our Edge cloud. The *P2P* model application has two mobility operations that we are interested to measure:

- *Compute*: Each peer has the possibility to compute any data that it receives from the thin client (The peer is either the Near or Far cloud, but could also be the thin client itself)
- *Store*: The thin client has the possibility to distribute data to either a Near or Far cloud for storage. For our test, we only create an object that has a content and a data size.

Each Peer object represents either a thin client, near or a far cloud where each of them have the possibility to compute or store data. For simplicity,

the thin client is the only one that can distribute data(we represent our data as 1000 bytes and the result distributing the data object is shown at table 7.19 where each machine types are based on figure 7.5) to the clouds or to any other smartphones in our application.

Table 7.19: Measuring the computation and storage performance in either Near of Far cloud

Machine type	Computation	Storage
Smartphone 2	10 s	0.23 s
Near cloud 1	5.35 s	0.017 s
Near cloud 2	5.4 s	0.017 s
Near cloud 3	6 s	0.018 s
Far cloud 1 (Italy)	127 s	0.2 s
Far cloud 2 (China)	1466 s	1.46 s
Far cloud 3 (Japan)	850 s	1.22 s

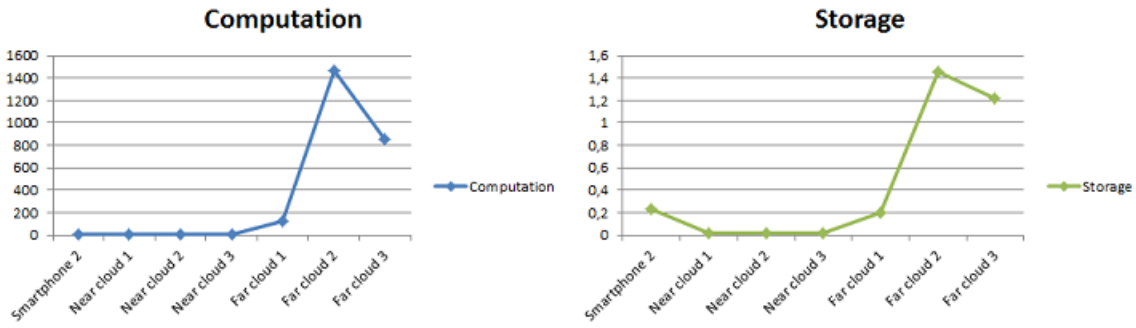

Figure 7.6: The Cost of doing Computation and Storage in a Seamless Near-Far cloud model

7.10 Discussion

Figure 7.6 shows a line graph with indicator where we have from the smartphone to near and far clouds as our *X- axis* where the *Y-axis* are the time in seconds. What we see is the time of computation and storage where the optimal point is to have a near cloud close by so we do not lose time on waiting to finish on either of the mobility operations.

7.11 The Data object

The code below shows the object that we are going to use for distributing to the different cloud types:

```
1 const Data <- class Data[content:String, d:Integer]
2   var bytes: Array.of[Integer] <- Array.of[Integer].create[d]
3   . . .
4 end Data
```

7.12 The Peer object

The code below shows how the Peer object which distributes to each of the clouds where each Peer can either compute or store data.

```
1  %a peer can store or compute any information it gets
2  const Peer <- class Peer[id:Integer, name: String]
3    %create an array of Data
4    var storageData: Array.of[Data] <- Array.of[Data].empty
5    %when computing, remember to move the data
6    export operation compute[d:Data]
7      var randNumber: Integer
8      self.srandom
9      var dataArray: Array.of[Integer] <- d.getDataBytes
10
11     for i: Integer <- 0 while i <= dataArray.upperbound by i <- i
           +1
12       randNumber <- self.random.abs # 2147483647
13       dataArray[i] <- randNumber
14     end for
15
16   end compute
17
18   export operation store[d:Data]
19     storageData.addUpper[d]
20   end store
21   . . .
22
23 end Peer
```

7.13 Summary

We have shown it is possible to have Emerald integrated into our smartphones and shown two use cases, basic performance and seamless Near-Far cloud evaluation. Basic performance has two use cases we focus on, *Kilroy* and *Break-Even Point*. To demonstrate and evaluate the seamless Near-Far cloud, we made a simple *P2P* application to illustrate a thin client distributing computation or store data to either Near or Far cloud. We have proven that distributing data to a Near cloud is more efficient than distributing to a Far cloud even though the cloud that is further has stronger computation than a Near cloud, the latency and the bandwidth is also a motivation for having a cloud in the middle.
Figure 7.6 shows that it pays to move calculation away from smartphones than doing the computation on the phone.
To summarize, using the phone as our thin client to distribute objects to any machines or smartphones that is either a Near or Far cloud is efficient. It does not cause a lot of work to the client itself, which is not what the client should do, because of battery power to do the computation.

Part IV

Conclusion

Chapter 8

Conclusion

In conclusion, we have now implemented a way to use Emerald to go on to the edge of the cloud and can now run Emerald on smartphones that has Android as their OS. Although most smartphones have different CPU architectures, we need to cross compile to the correct CPU to be able to run Emerald on the phone. To avoid to root the phone, we had to build an app that would be a layer on top of the Emerald compiler where this app emulates a terminal as the same as Linux has.

8.1 Contributions

Here is what I evaluate my contributions for the project

- Porting Emerald on Android Smartphones
- Created an app that would use Emerald
- Evaluate the Performance on Smartphones
- Limitations in the Emerald Implementation

8.1.1 Porting Emerald on Android Smartphones

Using standalone toolchain helped us port the Emerald compiler without any big problems. We have to choose the right compiler for the CPU the smartphone has, in order to make it work on the smartphone. For example an ARM executable file do not work on MIPS CPU phones. Since we needed to cross-compile, we have to consider that each architecture might translate a signed into an unsigned which leads to small minor issues and to avoid such problems, we need to specify flag variables. There are two ways to execute binary files, either you have to root the phone or create an app where the binaries must be stored in the apps internal memory and the app itself acts as a wrapper for the binaries.

8.1.2 Created an app that would use Emerald

We modified the existing app called *Terminal IDE* where our app is more suited for compiling and executing any Emerald programs. The app is like a terminal where from the application level it sends information to the lower level to execute any binary files. Creating an app solves the problem of executing binaries directly. We have implemented an app that acts as a top layer above the binary files.

8.1.3 Evaluate the Performance on Smartphones

We have proven that having a nearby cloud between the thin client and the data center is efficient. Even though the far cloud might be a much stronger and powerful machine than a near cloud, we have to consider the latency between the thin client and the clouds as well as the bandwidth. That is why we wanted to port the Emerald programming language into the smartphones to create a Near-Far cloud model program that would distribute to either one of the different clouds and measure if it was a good idea to have a cloud in the middle. The smartphones do not need to do much work than to distribute the data to nearby clouds and let them do the computation. This saves a lot of battery power for the phone as well as less latency to commmunicate with a server.

8.1.4 Limitations in the Emerald Implementation

Emerald is still a prototype. It is not an enterprise programming language as *Java* where the main challenge is when we have a bad bandwidth and a program that does a lot of processing. The Emerald nodes get a memory exception. Another issue is that Emerald have connection issues when it comes to communicating to Emerald nodes which was proven in chapter 6 when we tested using Emerald on phones and on the Amazon EC2 cloud servers. There might be a case where some Emerald nodes does not recognize the other Emerald nodes than itself and the other node it got in contact with. This problem do not have to do with firewall or closed port as each pair of phone or cloud server can get in contact with each other, but can not form a network with many of each devices. There are limited functions and built-in objects that we have to consider making most of the functions on our own rather than depending on many utilities that exists such as C#, C/C++ or Java provides.

Appendix A

Future works

The *Emerald* programming language has many advantages than any other languages when it comes to distributing objects efficiently. However, the Emerald runtime system needs a lot of improvements. The implementation of the language should increase the heap size of the virtual machine because there is a case when there are too many nodes that are initialized and listens, and you begin distributing objects when you have a bad bandwidth, you get a memory exception. The Emerald binary executables should also support 64 bits machines because we might get problems with understanding 32 bits binaries in a 64 bits system.

The app Emerald-Lite is still a prototype, which means this is not as user-friendly because most users should have some linux knowledge to use such app. What can be improved is to make a similar one that *C4Droid* has made. Another thing that should be improved is to create a more "code-friendly" soft keyboard because the app uses the soft keyboard that the phone already has.

Appendix B

Code

The source code and the apk file is available at:

http://code.google.com/p/master-thesis-emerald/source/browse/

Note that most of the source code is a modified version of ***Terminal IDE***[31] where this version supports Emerald compiler. You can change, modify and redistribute as much you want because my code follows the *GNU GPL 2.0*
Also a wiki page is available for explaining how we cross-compile the Emerald source code, install the Emerald compiler on any linux machines and how to run and distribute Emerald objects on our smartphones: ***https://code.google.com/p/master-thesis-emerald/w/list***

If you find any difficulties or bugs at all, please contact me: **kkho@ifi.uio.no**.

The code presented in this appendix shows the full version of the relevant codes presented in this thesis. I have chosen to put them here as all of the codes did not have any more space when describing these two codes. The Android implementation code that was presented in chapter 6 is not shown here as you can find them on the link of where the source code is.

B.1 Kilroy

The code below is an example of distributing an object to all available Emerald nodes. In this section, I will show the rest of the codes in this section.

```
1 const Kilroy <- object Kilroy
2   process
3     const home <- locate self
4     var there :     Node
5     var startTime, diff : Time
6     var all : NodeList
7
8     home$stdout.PutString["Starting on " || home$name || "\n"]
9     all <- home.getActiveNodes
```

```
10      home$stdout.PutString[(all.upperbound + 1).asString || " nodes
            active.\n"]
11      startTime <- home.getTimeOfDay
12
13      for i : Integer <- 1 while i <= all.upperbound by i <- i + 1
14        there <- all[i]$theNode
15        move Kilroy to there
16        there$stdout.PutString["Kilroy was here\n"]
17      end for
18      move Kilroy to home
19      diff <- home.getTimeOfDay - startTime
20      home$stdout.PutString["Back home.  Total time = " || diff.
            asString || "\n"]
21    end process
22  end Kilroy
```

B.2 CPU Execution Time

The code below shows how long it takes to compute a function. For simplicity, we only count up in the for-loop and measure the time before the loop and after to find out the finishing time of the function. I will show the rest of the codes in this section.

```
1  const CPUTime <- object CPUTime
2    export operation findExecutionTime
3      const localhome <- locate self
4      var finishedTime : Time
5      var startTime: Time <- localhome.getTimeOfDay
6      var counter: Integer <- 0
7
8      for i: Integer <- 0 while i < 10000000 by i <- i + 1
9        counter <- counter + i
10     end for
11
12     finishedTime <- localhome.getTimeOfDay - startTime
13     localhome$stdout.putString["CPU Execution Time: " ||
           finishedTime.asString|| "\n"]
14
15   end findExecutionTime
16
17   process
18     CPUTime.findExecutionTime
19
20   end process
21 end CPUTime
```

B.3 Local vs. Remote Invocation

The code below is an example of checking what is the difference between the local and remote invocation. This code shows an example of how the call-back problem occurs and how to avoid it (described more in chapter 3). I will show the rest of the codes in this section.

```
const Main <- object Main
  var X : I
  const home <- locate self
    var there,tmp :     Node
    var startTime, endTime : Time
    var all : NodeList
  var o1,o2 : ObjectCreation

  const I <- typeobject I
    operation doPrint[s:ObjectCreation]
  end I

  const A <- class A
    export operation doPrint[s:ObjectCreation]
      (locate self)$stdout.putstring[s.getPrint || "\n"]
    end doPrint
  end A

  const ObjectCreation <- class ObjectCreation[s:String]

    export operation getPrint[] -> [c:String]
      c <- s
    end getPrint

  end ObjectCreation

  process
    X <- A.create
    all <- home.getActiveNodes
    o1 <- ObjectCreation.create["Doing a remote call"]
    o2 <- ObjectCreation.create["Doing a local call"]

    %Do a remote call
    move X to all[1]$theNode
    startTime <- home.getTimeOfDay
    X.doPrint[o1]
    endTime <- (locate self).getTimeOfDay
    stdout.putstring["Remote call took "||(endTime-startTime).
        asString||" seconds\n"]

    %Do a local call
    startTime <- home.getTimeOfDay
    move o2 to X
    X.doPrint[o2]
    endTime <- (locate self).getTimeOfDay
    stdout.putstring["Local call took "||(endTime-startTime).
        asString||" seconds\n"]
  end process
end Main
```

B.4 P2P model - a Near-Far Cloud Model approach

B.4.1 The Data object

The code below shows how we have chosen our data object is represented. We use this object to distribute to either a Near or a Far cloud.

```
1  %The data that will be used to distribute around nodes
2  const Data <- class Data[content:String, d:Integer]
3    var bytes: Array.of[Integer] <- Array.of[Integer].create[d]
4    export operation getContent[] ->[s:String]
5    s <- content
6    end getContent
7
8    export operation getDataBytes[] -> [a:Array.of[Integer]]
9      a <- bytes
10   end getDataBytes
11
12 end Data
```

B.4.2 The Peer object

The code below shows how each Peer is represented in a Near-Far cloud environment. Each Peer has the possibility to either store or compute a data object. We should however represent each Data object as a type for securing "loose coupling".

```
1  %a peer can store or compute any information it gets
2  const Peer <- class Peer[id:Integer, name: String]
3    %create an array of Data
4    var storageData: Array.of[Data] <- Array.of[Data].empty
5    %when computing, remember to move the data
6    export operation compute[d:Data]
7      var randNumber: Integer
8      (locate self)$stdout.putstring["I will do the computation\n"]
9      %for now, the data object have integer, just fill values for
           the bytes of array
10     self.srandom
11
12     var dataArray: Array.of[Integer] <- d.getDataBytes
13
14     for i: Integer <- 0 while i <= dataArray.upperbound by i <- i
           +1
15       randNumber <- self.random.abs # 2147483647
16       dataArray[i] <- randNumber
17     end for
18
19   end compute
20
21   export operation store[d:Data]
22     (locate self)$stdout.putstring["I will store the data\n"]
23     storageData.addUpper[d]
```

```
24    end store
25
26    export operation getName[] ->[s:String]
27      s <- name
28    end getName
29
30    export operation getId[] ->[i:Integer]
31      i <- id
32    end getId
33
34    export operation printInformation[]
35      (locate Peer)$stdout.putstring["The node id: " || (id).
            asstring || " and the host name: " || name ||"\n"]
36    end printInformation
37
38    export function getStorage[] -> [a:Array.of[Data]]
39      a <- storageData
40    end getStorage
41
42      export operation random -> [n : Integer]
43         primitive "NCCALL" "RAND" "RANDOM" [n] <- []
44      end random
45
46    export operation srandom
47      const here <- locate self
48      const now <- here$timeofday
49      const usec <- now$microseconds
50      primitive "NCCALL" "RAND" "SRANDOM" [] <- [usec]
51    end srandom
52
53  end Peer
```

B.4.3 Main Object

The Main object is the thin client itself, however a thin client is also a Peer. So the first Peer object is also a thin client so we do not necessarily need to move the first Peer as we can just let it reside on the smartphone. The code below is a good example of a Main object that has the possibility to distribute data objects to either a Near or a Far cloud. Our application is limited, because the Emerald language is not complete and lacks several operations for example converting a *Character* to an *Integer*.

```
1  const Main <- object Main
2      const home <- locate self
3      var there :     Node
4      var startTime, diff : Time
5      var all : NodeList
6      var Peers : Array.of[Peer]
7
8    initially
9      Peers <- Array.of[Peer].empty
10     all <- home.getActiveNodes
11   end initially
12
```

```
13  process
14    %assume we have 1 smartphone node, 1 ifi node and
15    %1 planetlab node
16    if all.upperbound+1 >= 2 then
17      %here is reserved for only 2 smartphones!
18      Main.initNodes[all]
19
20      %initialize shell
21      (locate self)$stdout.putstring["thinclient@shell: "]
22      loop
23          Main.Shell[]
24      end loop
25    else
26      home$stdout.PutString["Initialize 2-4 Emerald nodes!\n"]
27    end if
28
29  end process
30
31  %initialize a lot of nodes,
32  export operation initNodes[a:NodeList]
33    var p: Peer
34    var pId: Integer
35    for i: Integer <- 0 while i <= a.upperbound by i <- i + 1
36      pId <- a[i]$theNode$lnn / 65536
37      p <- Peer.create[pId,a[i]$theNode$name]
38      Peers.addUpper[p]
39      move Peers.GetElement[i] to all[i]$theNode
40    end for
41  end initNodes
42
43
44  %check if you want to compute,store data on the peer, info or
        peer
45  export operation commandList[s:String]
46    if s = "compute" then
47      self.doCompute
48    end if
49
50    if s = "store" then
51      self.doStoreData
52    end if
53
54    if s = "info" then
55      self.printAvailableNodes
56    end if
57
58    if s = "help" then
59      self.printHelp
60    end if
61
62
63  end commandList
64
65  operation Shell[]
66    (locate self)$stdout.flush
67    (locate self)$stdout.putstring["thinclient@shell: "]
68
69    var s: String <- (locate self)$stdin.getString
70    %create substring, slice the newline
```

```
71      s <- s.getslice[0,s.upperbound]
72      self.commandList[s]
73
74    end Shell
75
76
77    operation doCompute
78      (locate self)$stdout.putstring["Which Node? (1 is the
            thinclient): "]
79      (locate self)$stdout.flush
80      var i: Character <- (locate self)$stdin.getChar
81
82      if !i.isdigit then
83        (locate self)$stdout.putstring["wrong data input, try again\
              n"]
84        return
85      end if
86
87      %create data
88      var inter: Integer <- self.intFromChar[i]
89
90      if inter > Peers.upperbound then
91        (locate self)$stdout.putstring["The value is higher than
              what Peers exists\n"]
92        return
93      end if
94
95      var d: Data <- Data.create["Contains data",1000]
96
97
98      move d to Peers.getElement[inter]
99
100     startTime <- (locate self).getTimeOfDay
101     Peers.getElement[inter].compute[d]
102     move d to Peers.getElement[0]
103     diff <- (locate self).getTimeOfDay - startTime
104
105     (locate self)$stdout.PutString["Computation took: " || diff.
            asString || "\n"]
106     Peers.getElement[0].getStorage.addupper[d]
107
108   end doCompute
109
110   export operation doStoreData
111     (locate self)$stdout.putstring["Which Node? (1 is the
            thinclient): "]
112     (locate self)$stdout.flush
113     var i: Character <- (locate self)$stdin.getChar
114
115     if !i.isdigit then
116       (locate self)$stdout.putstring["wrong data input, try again\
              n"]
117       return
118     end if
119
120     if Peers.getElement[0].getStorage.empty then
121       (locate self)$stdout.putstring["Compute data first, then
              call storage\n"]
122       return
```

```
123     end if
124
125     %create data
126     var inter: Integer <- self.intFromChar[i]
127
128     if inter > Peers.upperbound then
129       (locate self)$stdout.putstring["The value is higher than
              what Peers exists\n"]
130       return
131     end if
132
133     (locate self)$stdout.putstring[inter.asstring ||"\n"]
134
135     var d: Data <- Peers.getElement[0].getStorage.removeUpper
136     startTime <- (locate self).getTimeOfDay
137     move d to Peers.getElement[inter]
138     Peers.getElement[inter].store[d]
139     diff <- (locate self).getTimeOfDay - startTime
140     (locate self)$stdout.PutString["Storing took: " || diff.
          asString || "\n"]
141
142   end doStoreData
143
144   operation printHelp
145     (locate self)$stdout.putstring["---------- Commands
          ----------\n"]
146     (locate self)$stdout.putstring["compute - Ask any Near or Far
          cloud to do the computing or do it itself \n"]
147     (locate self)$stdout.putstring["store - Ask any Near or Far
          cloud to store data or let it store itself\n"]
148     (locate self)$stdout.putstring["info - get a list of id and
          names of the thinclient, Near and Far clouds\n"]
149     (locate self)$stdout.putstring["help - prints out commands
          available for the P2P Model\n"]
150
151   end printHelp
152
153   operation printAvailableNodes
154     (locate self)$stdout.putstring["Remember, The first Peer is
          the thinclient! \n"]
155     (locate self)$stdout.putstring["
          ----------\n"]
156     for i: Integer <- 0 while i <= Peers.upperbound by i <- i + 1
157       (locate self)$stdout.putstring["Peer:"||(i+1).asstring||" id
            : " ||Peers.getElement[i].getId.asstring ||" name: " ||
            Peers.getElement[i].getName || "\n"]
158     end for
159     (locate self)$stdout.putstring["
          ----------\n"]
160   end printAvailableNodes
161
162
163   function intFromChar[c:Character] ->[i:Integer]
164     %brutally function, luckily 7 nodes only
165     if c = '1' then
166       i <- 1
167     end if
168     if c = '1' then
169       i <- 0
```

```
170       end if
171       if c = '2' then
172         i <- 1
173       end if
174       if c = '3' then
175         i <- 2
176       end if
177       if c = '4' then
178         i <- 3
179       end if
180
181       if c = '5' then
182         i <- 4
183       end if
184       if c = '6' then
185         i <- 5
186       end if
187       if c = '7' then
188         i <- 6
189       end if
190
191     end intFromChar
192   end Main
```

Bibliography

[1] Tanenbaum S Andrew and Steen van Maarten. "Distributed systems-principles and paradigms." In: (2007).

[2] Developers Android. *Activity*. URL: http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle.

[3] Developers Android. *Application Fundamentals*. URL: http://developer.android.com/guide/components/fundamentals.html.

[4] Rajendra Asuri. *ACTIVITY STACK IN ANDROID*. URL: http://wwwsimpleandroid.blogspot.no/2013/05/activity-stack-in-android.html.

[5] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. "Cloud computing: A study of infrastructure as a service (IAAS)." In: *International Journal of engineering and information Technology* 2.1 (2010), pp. 60–63.

[6] George Coulouris et al. *Distributed Systems: Concepts and Design (5th Edition)*. 5th ed. Addison Wesley, May 7, 2011. ISBN: 0132143011. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20%5C&path=ASIN/0132143011.

[7] Thomas J Davidson and Michael T Kelley. *Method and system for implementing remote procedure calls in a distributed computer system*. US Patent 5,307,490. Apr. 1994.

[8] Android Developers. *Google Cloud Messaging for Android*. URL: http://developer.android.com/google/gcm/index.html.

[9] EDC4IT. *What do you know about Android Runtime*. URL: http://www.edc4it.com/2014/01/02/what-do-you-know-about-android-runtime/.

[10] PlanetLab - Europe. *PlanetLabEurope*. URL: http://www.planet-lab.eu/.

[11] Melissae Fellet. "Touch and go: fondle the digital world." In: *New Scientist* 214.2868 (2012), pp. 40–43.

[12] Developers Google. *Android Cloud to Device Messaging Framework*. URL: https://developers.google.com/android/c2dm/.

[13] Jerry Hildenbrand. *Editorial: Sometimes, root isn't the answer*. URL: http://www.androidcentral.com/sometimes-root-isn%C3%A2%C2%80%C2%99t-answer.

[14] Norman C Hutchinson. *EMERALD: An object-based language for distributed programming*. Tech. rep. Washington Univ., Seattle (USA), 1987.

[15] Norman C Hutchinson. "The Emerald Programming Language1." PhD thesis. Department of Computer Science, University of Copenhagen, 1997.

[16] Wallace Jackson. *Android apps for absolute beginners*. Apress, 2012.

[17] Eric Jul. "Object Mobility in a Distributed Object-Oriented System." In: (1989).

[18] Eric Jul and Norm Hutchinson. *The Emerald Programming Language*. URL: http://www.emeraldprogramminglanguage.org/.

[19] Rachel King. *Who wins, who loses in Oracle v. Google*. May 2012. URL: http://news.cnet.com/8301-1001_3-57440688-92/who-wins-who-loses-in-oracle-v-google/.

[20] Feipeng Liu. *Android Native Development Kit Cookbook*. Packt Publishing Ltd, 2013.

[21] Lisa Mahapatra. *Android Vs. iOS: What's The Most Popular Mobile Operating System In Your Country?* Nov. 2013. URL: http://www.ibtimes.com/android-vs-ios-whats-most-popular-mobile-operating-system-your-country-1464892.

[22] Reto Meier. *Professional Android 4 application development*. John Wiley & Sons, 2012.

[23] NewCircle. *Android Bootcamp Training Course*. URL: https://thenewcircle.com/training/android/android_bootcamp.html.

[24] Crystal Nichols. *Thin Clients and The Cloud*. URL: http://www.unitiv.com/it-solutions-blog/bid/59161/Thin-Clients-and-The-Cloud.

[25] Tomas Phelps. *To Root or Not to Root*. URL: http://google.about.com/od/socialtoolsfromgoogle/a/root-android-decision.htm.

[26] PlanetLab. *PlanetLab*. URL: https://www.planet-lab.org/.

[27] Margaret Rouse. *What is seamless interface?* URL: http://whatis.techtarget.com/definition/seamless-interface.

[28] Amazon Web Services. *Amazon Elastic Compute Cloud (Amazon EC2)*. URL: http://aws.amazon.com/ec2/.

[29] Andrew Sloss, Dominic Symes, and Chris Wright. *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann, 2004.

[30] Intel Software. *Android Application Development and Optimization on the Intel Atom Platform*. URL: software.intel.com/en-us/articles/android-application-development-and-optimization-on-the-intel-atom-platform.

[31] SpartacusRex99. *Spartacus Rex*. URL: http://www.spartacusrex.com/terminalide.htm.

[32] SpencerCS. *BlankMap-World-WWII*. Aug. 30, 2011. URL: https://en.m.wikipedia.org/wiki/File:BlankMap-World-WWII.PNG.

[33] TutorialsPoint. *Android Application Component*. URL: http://www.tutorialspoint.com/android/android_application_components.htm.

[34] Mike Williams. *Why are mobile phone batteries still so crap?* URL: http://www.techradar.com/news/phone-and-communications/mobile-phones/why-are-mobile-phone-batteries-still-so-crap--1162779.

[35] Xamarin. *CPU Architecture*. URL: http://docs.xamarin.com/guides/android/advanced_topics/cpu_architecture/.

[36] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly, 2013.