UiO : **Department of Informatics**
University of Oslo

# Porting the Distributed Object Language Emerald to iOS

"iOS: the final frontier. These are the students of Emerald. Its two-year mission: to explore object mobility, to seek out new platforms and new users, to boldly go where no man has gone before."

Audun Øygard

Master's Thesis Autumn 2016

# Abstract

This thesis discusses the research and implementation of porting a distributed object-oriented programming language known as Emerald, to iOS devices.

Our goal is to make a working implementation of the Emerald programming language on an iOS device and make this device communicate seamlessly with Emerald on other platforms.

We have successfully ported Emerald to an iOS device and further experimented with various use cases and performed an evaluation.
The evaluation includes benchmarking of the implementation and an evaluation of iOS as a development platform.

Our work concludes that an implementation is not possible without jailbreaking the device, and further, that with a jailbroken device we get an adequately performant implementation. We have also concluded that developing such "outlying" applications on iOS is unnecessarily difficult.

# Acknowledgements

I would like to thank my supervisor Eric Bartley Jul, who offered me this thesis and have been providing valuable feedback and invaluable beer along the way.

I would also like to thank my fellow students on the 10th floor for the many coffee breaks.

# Contents

# List of Figures

# List of Tables

# Part I

# Introduction

# Chapter 1

# Introduction

The iOS platform with its many security features and limitations could be described as the Fort Knox of smartphones.
In this thesis, we try to mitigate, work around and otherwise break these security features to be able compile and run Emerald programs on iOS.
Previous implementations of Emerald exists on many platforms including Android and with this thesis, we hope to bring a truly great distributed programming language used by dozens of people around the world every year to the iOS platform.

## 1.1 Motivation

Emerald implements the idea of fine grained mobility [10], this means that any object of any size can be freely moved around in the distributed system. Any entity can easily be moved by the programmer e.g. from a mobile device to a data center and vice versa.
From a mobile point of view this can have many advantages, like offloading a heavy computations to a cloud service and moving a process closer to a network node to lower the latency.
Because Emerald runs on most modern operating systems including the mobile operating system Android it will be beneficial to expand the reach to other mobile operating systems for a more ubiquitous experience.
As of the fourth quarter of 2014 Android covers 76.6% of the marked share for mobile operating systems while iOS covers 19.7%. [1] This means that with an implementation of Emerald for iOS Emerald can be

used on over 96% of the mobile phones in the world.

## 1.2 Goal

Our goal is to expand Emerald's reach further by making an working implementation for the iOS operating system.

## 1.3 Approach

We take an existing programming language and try to port it to an iOS device. Finally, we evaluate the results and evaluate the suitability of the iOS platform for such a task.

## 1.4 Work done

We have jailbroken an iOS device and ported the Emerald runtime to run in this environment. Further, we built an native iOS application as a front-end to the runtime, and measured how well the implementation performs on the device.

## 1.5 Results

We have successfully ported Emerald to an jailbroken iOS device and showed the following results:

- **Basic performance:** We have shown that the basic performance of the Emerald implementation is reliable, but somewhat slower compared to our other test beds.

- **Distribution performance:** Using distribution of objects we have shown that computational offloading can be very beneficial over doing all the computation on the device.

## 1.6 Contributions

We have contributed on the following:

- Ported the Emerald runtime to iOS
- Created a native iOS application using the port
- Evaluated the performance of the port for iOS
- Evaluated iOS as a development platform.

### 1.6.1 Ported the Emerald runtime to iOS

We ported the Emerald runtime to run on the iOS architecture by cross-compiling the source code using the iOS SDK provided clang compiler. However, the iOS environment is so restrictive that to allow execution of this binary, we had to jailbreak the device to be allowed to run it.

### 1.6.2 Created a native iOS application using the port

We created an native iOS application for convenient use of Emerald on the device. The application acts as a front-end for the Emerald runtime, letting us compile and run Emerald programs directly from the device.

### 1.6.3 Evaluated the performance of the port for iOS

We have evaluated the performance of distributing objects with Emerald for iOS. We have also shown the performance benefits computational offloading and moving objects for latency reduction gives us.

### 1.6.4 Briefly evaluated iOS as a development platform

We briefly evaluated iOS as a development platform concluding that when developing "normal" applications the platform is top notch, but it gives us some problems when developing niche applications like Emerald.

## 1.7 Conclusion

We successfully ported Emerald to iOS by cross-compiling the source code for the different iOS architectures. Because of the limitations that is set by the platform we needed to jailbreak a device to run the cross-compiled binary. Further we built an native iOS application acting as a front-end for the ported Emerald version, making it easy to use on the device.

## 1.8 Outline

**Chapter 1: Introduction**
Gives a brief introduction to the thesis and the work performed.

**Chapter 2: Background**
Provides background information on Distributed systems, Object oriented programming, distrusted objects, Planetlab and smartphones.

**Chapter 3: The Emerald programming language**
Gives a brief overview of the Emerald programming language.

**Chapter 4: iOS - the Apple smartphone OS**
Provides a overview and a short history if iOS.

**Chapter 5: Porting Emerald**
Covers the steps of porting Emerald to the iOS architectures.

**Chapter 6: iOS application**
Explains the implementation of the Emerald iOS application.

**Chapter 7: Distribution with Cydia**
Briefly covers the steps of distributing the port and the iOS application with Cydia.

**Chapter 8: Performance and Evaluation**
Covers the results of the performance and evaluation of the port and iOS as a development platform.

# Part II

# Background

# Chapter 2

# Background

This chapter introduces basic concepts within distributed systems, gives an introduction and a brief history to object oriented programming, distributed objects, Planetlab and smartphones in general.

## 2.1 Distributed systems

A distributed system is defined as one which *hardware or software components located at networked computers communicate and coordinate their actions only by passing messages*. [2]

Ghosh [3] also argues that a system can not truly be called a distributed system unless the interaction of the components works together to meet a common goal and further suggests that a distributed system fulfills the following criteria:

- The system consists of more than one independent process.
- Processes communicate with message passing.
- Processes have disjoint address spaces, in principle meaning no shared memory.

One of the motivations behind such systems is to be able to share the resources each individual machine have among the other nodes in the network. In this context, a resource could be anything from sharing CPU-time for load balancing, sharing local data or more concrete resources such as a peripheral device (printer, CD-ROM, etc.). Another important

motivation for distributed systems is fault tolerance. With a distributed system you can avoid having a single point of failure and instead have several nodes performing the same work avoiding a complete system collapse or a single computation error propagating through the system. [3]
There exists a wide variety of distributed systems but one of the main ways of separating them is the *coupling level* of the processors in the system. [14] This ranges from tightly coupled systems where the processors work in synchrony communicating fast and reliably, to systems where the processors are more independent and communication is less frequent.

Examples of distributed systems: [3]

- The Internet — The World Wide Web is the largest distrusted system we have with web servers, file servers, dns servers etc. spread over the whole planet and some end nodes even in interplanetary space.

- Social networks — Facebook has several data centers, load balancers and servers around the globe and is essentially a distrusted system.

- Banking systems — Banking systems can be seen as distributed systems with ATMs, branch offices and backend servers and services spread through different cities and countries.

- Peer-to-peer systems — Peer-top-peer systems such as Gnutella, Bittorrent etc. is an distributed system because the resources (processing power, network bandwidth, disk storage etc.) are distributed amongst its nodes ("peers") without any central coordination.

- Folding@home — A distributed computing project for disease research that simulates protein folding. Everyone can install the software and it will use the idle processing power to add to the processing power of the system.

- Sensor networks — Distributed sensor networks such as battlefield surveillance and home automation are also examples of distributed systems.

## 2.2 Object-oriented programming

### 2.2.1 History

The concept of objects in a programming language was first introduced in the programming language Simula 67 in 1967 by Kristen Nygaard and Ole Johan Dahl at the Norwegian Computing Center in Oslo. [4]
The motivation behind this was to provide a set of tools for expressing unifying properties among related processes in a system and Simula was the first language to introduce concepts we know from modern programming languages such as objects, classes, subclasses and inheritance.
Object-oriented programming had a rise in use and popularity in the 1990s and onwards when languages such as C++ and Java became widely available.
In figure 2.1, we see an example of a Java object representing a car.

### 2.2.2 Concepts

In object-oriented programming languages an object consists of *a set of operations on some hidden data*. [12] These operations serves as an interface that we interact with, which in turn interacts with the data encapsulated by the object.
The objects serves as a way to encapsulate functionality and data that fits logically together, and object-oriented design consists of identifying these structures and the relations between them.
Four basic concepts for object-oriented languages: [12]

- Dynamic lookup — When using an object, the code that is executed is determined by the way the object is implemented.
- Abstraction — The implementation details are hidden inside the object and we interact with the object through its public operations, that in turn manipulates the objects internal data.
- Subtyping — If some object A has all the functionality of an object B we can use object A where an object B is expected.
- Inheritance — Inheritance reuses the definition of one object to create another type of object.

```java
public class Car {

  int currentSpeed;

  public void accelerate() {
  }

  public void deaccelerate() {
  }

  public void getCurrentSpeed() {
  }
}
```

Figure 2.1: An example of an Java class representing a car

## 2.3 Distributed objects

Because object-oriented programming enforces the use of encapsulation this results in programs that are partitioned into logical parts. This again means that the distribution of these objects to other computers or processes is a natural extension of this modular design.

Distributed objects can use the server-client pattern where the objects are maintained by its server and the clients uses RMI (Remote Method Invocations) to access the objects on the server. [2]
Because the objects are only accessed through its methods, the state of the objects may only change through these remote invocations.

This gives rise to several advantages: [2]

- Security — Unauthorized methods cannot change the state of the object.
- Implementation — The implementation of the methods can differ from system to system as long as it conforms to the objects definition.
- Concurrency — The use of synchronization primitives in the methods can protect from conflicting access to the object instance variables.

Another convenient pattern introduced with distributed objects is object replication. Object replication is where an object is distributed amongst several hosts to achieve higher fault tolerance and possibly higher performance.

The distributed object model have two fundamental concepts at its core: [2]

- Remote object references — A remote object reference is an global identifier that can be used through the distributed system to refer to a object This is different from the local reference to the object.
- Remote interfaces — A remote interface is a interface describing which of the objects methods that can be invoked remotely. In figure 2.2 all the remote objects methods are also described in its remote interface.

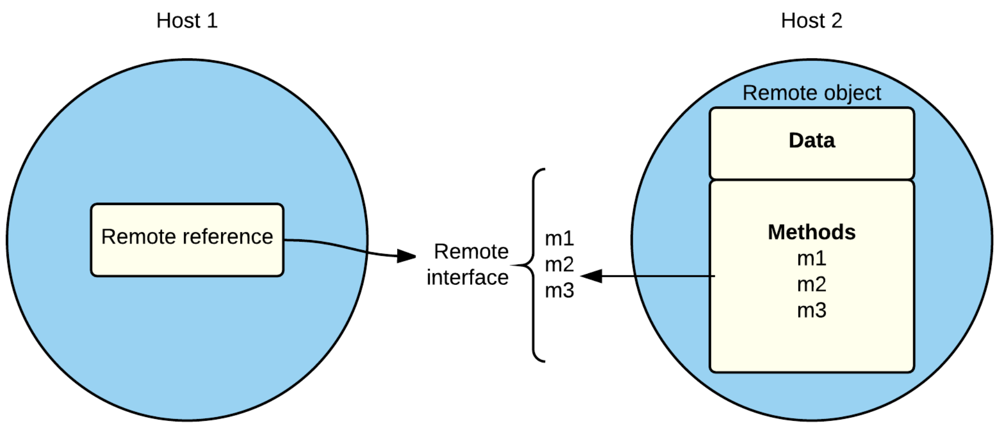


Figure 2.2: A remote object and its remote interface

## 2.4 Planetlab

Planetlab is a global research network used as a network testbed for distributed system research and other academic research requiring a planetary-scale network.
The project started in 2002 with one hundred machines funded by Intel Research and has since grown into a much larger network.
The idea behind Planetlab is to have a network testbed that operates under real-world condition and at a planetary-scale. Planetlab currently consists of over 1000 nodes placed all around the world and each project is given a *slice* of these nodes to perform on. Since the start of Planetlab more than 1000 researchers have used PlanetLab to develop new technologies for distributed storage, network mapping, peer-to-peer systems, distributed hash tables, and query processing.
To have access to Planetlab you have to be a member of an institution that is a member of the PlanetLab Consortium.

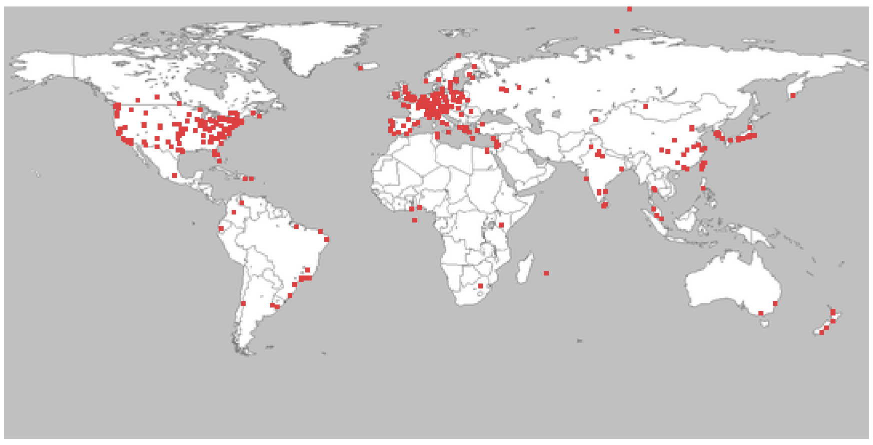


Figure 2.3: A map showing the distribution of Planetlab nodes.

## 2.5 Smartphones

After the first modern smartphones emerged in the early 2000s, mostly adopted by business users with phones like the BlackBerry, the adoption of smartphones has exploded around the world.
The smart phone usage in Norway in the first quarter in 2015 is in a gallup conducted by TNS Gallup estimated to be 84%. [11]
The forerunners to the modern smartphone was the PDAs of the late 1990s, which started with the IBM Simon developed in 1992 and marketed to consumers in 1994. [15] IMB Simon had a monochrome touchscreen, built in applications for e-mail, calendar, clock and featured predictive typing as seen in modern smartphones.
The IBM Simon sold about 50,000 units and the next-generation was abandoned by IBM.
After the IBM Simon the next step in the history of smartphones was the PDAs running operating systems such as BlackBerry OS, Windows CE and Palm OS.
These did not have the mobile phone aspects that smartphones or the IBM Simon had.
The Ericson R380, released in 2000 was the first mobile phone marketed as a smartphone, and was the first device to use device to use Symbian OS. In 2007, Apple Inc. released the first generation iPhone most notable for its large touchscreen and the use of ones finger as the main means of interaction in contrast to the stylus that previous touchscreens used.
The year after, in 2008, the first smartphone with the Android OS was released by HTC with the HTC Dream.
By the 4th quarter of 2010 Android overtook iOS as the best-selling smartphone OS. [16].

## 2.6 Summary

In this chapter, we describe basic concepts of distributed systems and define it as a system where hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.

We explain how object oriented programming was first introduced with the Norwegian programming language Simula and how in object oriented programming we encapsulate data and functionality that fits together in logical units called objects.

We explain distributed objects and how it has two fundamental concepts: remote object references and remote interfaces.

We also introduce Planetlab, a planet wide research network and gave a quick summary of the history of smartphones.

In the next chapter, we describe the Emerald programming language in more detail.

# Chapter 3

# Emerald

In this chapter, we introduce the Emerald programming language.
We explain the ideas and the history of the language, we and explain Emerald objects, the type system and the distribution primitives.

## 3.1 The Emerald programming language

Emerald is a distributed object programming language developed in the 1980's by Eric B. Jul, Norman C. Hutchinson, Andrew P. Black and Henry M. Levy at the Department of Computer Science at the University of Washington.
At the time, previous distributed programming languages used a twofold computational model, one for migrating entities and one for locally executing entities. Emerald tried to address this with using an uniform object model for both distributed objects and local objects, while simultaneously having comparable performance to procedural languages at the time.

Emerald was designed around these three ideas: [10]

- A uniform object model for both local and distributed computation
- On-the-fly fine-grained mobility
- Language support for mobility to achieve an efficient implementation

By uniform object model, we mean that all data (small objects, big objects, objects with processes etc.) is represented as actual objects, and is accessed using a single mechanism (object invocation). This is different from other distributed programming languages that use a twofold computational model where the programmer uses two different mechanisms for local and remote objects.

On-the-fly fine-grained mobility refers to the way that objects of any size can be moved (fine-grained) at any time (on-the-fly) even while the object has processes executing within it. As opposed to coarse-grained mobility where an entire process and its address space is moved together.

Emerald achieved large gains in efficiency over previous systems at the time. These gains stems from the following design decisions: [10]

- Integrating the object concept (including mobility) into the programming language.
- Close cooperation between the compiler and the runtime system.
- A single address space for all implemented objects on a node.

Benefits gained from Emerald's mobility and mobility in general: [10]

- Communication performance — Objects that interact intensively can be moved closer together (even on the same node).
- Load sharing — By moving objects around we can utilize unused resources.
- Availability — Data can be replicated and moved to different nodes to provide higher data availability.
- User mobility — A users processes can be moved around seamlessly with the user.
- Reconfiguration — Programs can be moved to other nodes during down-time to provide uninterrupted service.
- Special capabilities — Programs can be moved to nodes that have some special capabilities that other nodes may not possess.

## 3.2 Objects in Emerald

An object in Emerald consists of: [10]

- A network wide-network identity
- The local data of the object (Primitive objects or pointers to other objects).
- The objects operations. Exported operations can be invoked outside of the object while non-exported operations can only be invoked inside the object. Similar to public/private in Java.
- The optional **initially** operation that is executed when the object is created.
- The optional **process** operation that is executed after the object is initialized.

Figure 3.1 is an example of an Emerald object with a process and a local operation.

```
const obj <- object Example
    const hello <- "Hello,␣World!\n"

    operation printHello [] -> []
       stdout.putString[hello]
    end printHello

    process
       self.printHello
    end process
end Example
```

Figure 3.1: An example of an object in Emerald

## 3.3 Types in Emerald

The type system in Emerald is based on abstract types.

"An abstract type defines a collection of operation signatures, that is, operation names and the types of their arguments and results. All identifiers in Emerald are typed:

the programmer must declare the abstract type of the objects that an identifier may name. An abstract type is represented by an Emerald object that specifies such a list of signatures." [5]

The abstract type of the object being assigned must conform to the abstract type of the identifier. Conformity is the basis of type checking in Emerald. Informally, a type S conforms to a type T (written S > T ) if: [5]

- S provides at least the operations of T ( S may have more operations).
- For each operation in T , the corresponding operation in S has the same number of arguments and results.
- The abstract types of the results of S's operations conform to the abstract types of the results of S's operations.
- The abstract types of the arguments of T's operations conform to the abstract types

## 3.4 Distribution in Emerald

Emerald implements several language primitives that enable easy distribution of objects.

- **locate *X*** — Returns the current location of the object X.
- **move *X* to *Y*** — Moves the object X to the node where Y resides.
- **fix *X* at *Y*** — Moves the object X to the node where Y resides and prevents it from being moved.
- **unfix *X*** — Enables a fixed object X to move again.
- **refix *X* at *Y*** — Unfixes and fixes the object X at the node where Y resides.

Emerald provides an efficient yet simple way of distributing objects as shown in figure 3.2 where the *Kilroy* object is moved around to every node in the cluster and prints *Kilroy was here!* at every node.

```
const Kilroy <- object Kilroy
  process
    const origin <- locate self
    const up <- origin.getActiveNodes
    for e in up
        const there <- e.getTheNode
        move self to there
        (locate self)\$stdout.putString["Kilroy␣was␣here!\n"]
    end for
    move self to origin
  end process
end Kilroy
```

Figure 3.2: Moving an object through all the active nodes in a system

## 3.5 Summary

In this chapter, we introduce the Emerald programming language, a language created in the 1980's with effortless distribution in mind. We explaine what an Emerald object consists and that each distributable object has an global network id. We explaine how the type system in Emerald is based on abstract types, where a type is an interface describing what methods a object must implement to conform to the type. We also briefly explain the distribution primitives of the Emerald language.

In the next chapter, we give the details on iOS - the Apple smartphone OS.

# Chapter 4

# iOS - the Apple smartphone OS

In this chapter we take a closer look at iOS, the operating system running on Apple smartphones. We explain the history, how development works, and explain some of the core frameworks. We also expand on Apples publishing process.

## 4.1 The iOS operating system

iOS is an mobile operating system developed by Apple Inc. and was first revealed in 2007 (then named iPhone OS 1.x) for the first generation iPhone.
It has later been expanded to other Apple devices such as the iPad, iPod touch and Apple TV, and was renamed to iOS on June 7, 2010 with the introduction of the iPad.
As of the fourth quarter of 2014 iOS have 19.7% of the marked share of the mobile marked, only second to Android. [1]
On January 27, 2015 Apple announced that they had sold one billion iOS devices. [9]
Major versions of iOS is released each year and the latest major update was released on April 8, 2015.

iOS is a closed source operating system based on Darwin OS (developed by Apple) which in turn is based on Unix and NeXTSTEP. Darwin forms the core components of both OSX and iOS and is mostly POSIX compatible, but has never been fully certified. iOS also shares some frameworks with OSX such as *Core Foundation* and *Foundation* but iOS uses *Cocoa Touch* rather than the OSX *Cocoa* user interface toolkit

making it incompatible with OSX applications.
Even though iOS is based on Darwin OS it does not provide shell access or access to the file system.

## 4.2 The history of iOS

Following is a list of the major versions of iOS and its most prominent features. [17]

| **Version** | **Announced** | **Major features** |
|---|---|---|
| iPhone OS 1.x | June 29, 2007 | First release,<br>Mobile safari,<br>Multitouch gestures. |
| iPhone OS 2.x | July 11, 2008 | Introduced the App Store,<br>making third-party applications available. |
| iPhone OS 3.x | June 17, 2009 | Copy/Paste functionality,<br>MMS,<br>iPad introduced with version 3.2.<br>3.2 First version to be called iOS |
| iOS 4 | June 21, 2010 | Multitasking,<br>Home screen folders,<br>Retina Display support. |
| iOS 5 | June 6, 2011 | iMessage,<br>Siri,<br>No PC needed for activation,<br>iCloud |
| iOS 6 | June 11, 2012 | FaceTime over cellular network,<br>Facebook integration,<br>Apple Maps. |
| iOS 7 | June 10, 2013 | Major visual overhaul,<br>Control Center,<br>AirDrop. |
| iOS 8 | June 2, 2014 | Continuity (desktop integration),<br>third-party widget support,<br>Extensibility (making third-party keyboards available) |
| iOS 9 | June 8, 2015 | 3D Touch,<br>Proactivity (Context aware information),<br>Improved keyboard |

Figure 4.1: Evolution of the iOS homescreen from iPhone OS 1 to iOS 8.

## 4.3 iOS development

To be able to develop for iOS there is some prerequisites.

- Applications needs to be developed on a computer running OSX.
- Applications needs to be developed with Xcode.
- To publish the application and to the App Store, an Apple developer account is needed.

All applications that is to be developed for iOS and published on the App Store needs to be developed an signed on a Apple Mac using Xcode. There is ways to circumvent these restrictions but none are officially supported by Apple, and most include some sort of hack to make it work.

### 4.3.1 Supported programming languages

Traditionally, the programming language used for iOS and OSX development have been objective-C and has been used for Apple development since the 1980's. At the Worldwide Developers Conference in 2014 Apple announced the Swift programming language which is set to overtake the use of Objective-C in iOS and OSX application development. [18]

### 4.3.2 Objective-C

Objective-C is a object-oriented programming language originally developed in the 1980's and was used as the main programming lan-

guage for the NeXTSTEP operating system that Darwin OS is based on. Objective-C is a "strict superset" of C and inherits all of its types, syntax and flow control statements from C, meaning we are can freely include C code in a objective-C program.
The object syntax of Objective-C is based of the object syntax found in Smalltalk.
A Objective-C program is in no way bound to iOS or OSX and a program that uses none of their core frameworks or APIs can be compiled to run on any system using GCC or Clang.

#### Interfaces

Objective-C uses header files in much the same way traditional C uses them, with header files being denoted with the `.h` extension.
In the header files is where the interfaces are declared, an interface on objective-C is analog to a class declaration in similar languages. An interface for a car object might look like in figure 4.2, with `accelerate`, `deaccelerate` and `getCurrentSpeed` being declared as instance methods by the `-` sign.

```
@interface car
@property speed
- (void)accelerate;
- (void)deaccelerate;
- (int)getCurrentSpeed;
@end
```

Figure 4.2: An example of a interface in objective-C

#### Implementation

The implementation file contains the implementation of a previous declared interface (see figure 4.3) and uses the `.m` extention.

### 4.3.3 Swift

Swift is a multi-paradigm programming language created by Apple especially for iOS and OSX development. Swift development started in

```
@implementation car
@property speed
- (void)accelerate {
  // code
}
- (void)deaccelerate {
  // code
}
- (int)getCurrentSpeed {
  // code
}
@end
```

Figure 4.3: An example of a interface in objective-C

2010 and was announced to the public at the Worldwide Developers Conference in 2014. [18]
Swift is created as a replacement for Objective-C in iOS and OSX development and has a simpler and cleaner syntax than its predecessor as seen in figure 4.4.
As opposed to Objective-C Swift does not provide access to pointers, variables are always initialized and arrays and integers are checked for overflow. These features make simple mistakes easier to avoid in Swift than in Objective-C.
Swift uses the Objective-C runtime allowing Objective-C and Swift to run together in a single program.

```
// Objective-C
NSString *str = @"hello,";
str = [str stringByAppendingString:@"␣world"];

// Swift
var str = "hello,"
str += "␣world"
```

Figure 4.4: Initalizing, assigning and appending a string in objective-C and Swift

### 4.3.4 Automatic Reference Counting

ARC or *Automatic Reference Counting* is a memory management scheme used by both Objective-C and Swift.
Apple introduced ARC in 2011 and before this the programmer was

required to use the `release` and `retain` keywords to manually handle marking objects for deallocation or preventing them from being deallocated.
With Automatic Reference Counting the compiler analyzes the code and automatically inserts these where they are needed.
This results in less work for the programmer and minimizes the chances for memory leaks.
Automatic Reference Counting does not handle strong reference cycles. The programmer is in charge of using the `weak` keyword to mark a reference as weak, meaning that the reference is not protected from garbage collection and the cycle is broken.

### 4.3.5 Xcode

Xcode is an integrated development environment (IDE) developed by Apple first released in 2003, it is free to use for OSX users and the latest version is 6.3.
It does not only consist of the IDE but also contains developer documentation and the Apple Interface Builder which is a tool for building graphical interfaces. The latest version uses Apple Clang as a compiler frontend with Apple LLVM as the backend.
Xcode can build, run debug and install iOS applications both through the iOS simulator and by transferring the application to a physical device.

#### Storyboard

The Xcode Storyboard is an user interface designer used in Xcode for designing graphical iOS applications.
With the Storyboard the developer lays out the *path* through the application using *scenes*, *segues* and *controls*.
A scene represents a content area in an application, this can be the whole view a user seen or the view can consist of several scenes.
A segue represents the transition from one scene to another scene.
Controls are used to trigger the segues between the scenes.
The Storyboard makes designing user interfaces fast an relatively easy, and makes an solid separation between the application logic and the user interface. In figure 4.5 we see an example of a storyboard.

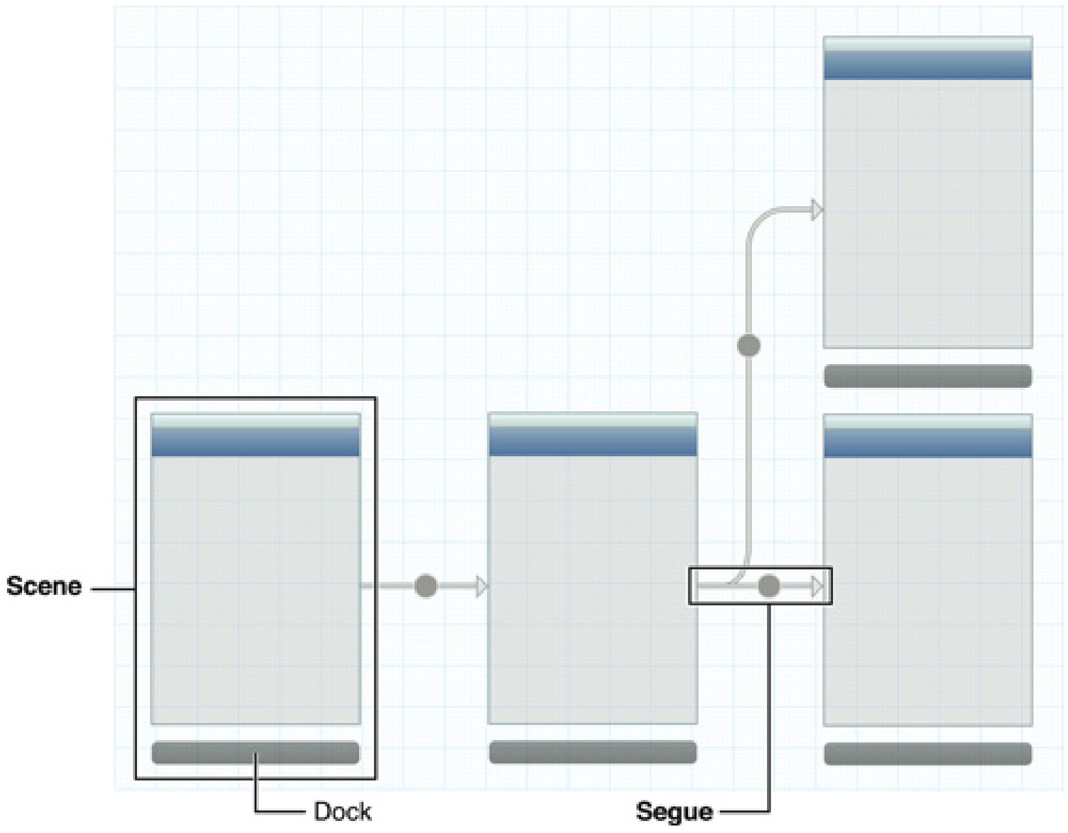


Figure 4.5: A storyboard showing 4 views and several segues.

## 4.4 Core frameworks and APIs

*Cocoa touch* is described as the application development environment for iOS.
Cocoa touch includes the Objective-C runtime and two core frameworks: Foundation and the UIKit Framework.

### 4.4.1 Foundation

The Foundation Framework defines a base layer of Objective-C classes. It provides a set of useful classes and introduces several paradigms that include functionality that Objective-C does not include.
The Foundation framework was designed with these goals in mind: [7]

- Provide a small set of basic utility classes.

- Make software development easier by introducing consistent conventions for things such as deallocation.
- Support Unicode strings, object persistence, and object distribution.
- Provide a level of OS independence, to enhance portability.

The Foundation framework provides the base object class `NSObject`, classed for representing basic data types such as strings (`NSString`) and various collection classes for storing objects such as `NSArray`, `NSDictionary` and `NSSet` and many other useful classes such as `NSDate`, `NSTimeZone` and `NSCalendar` for storing and using times and dates.

### 4.4.2 UIKit Framwork

The UIKit framework provides the infrastructure needed for constructing iOS applications.

UIKit provides the components for creating views, managing event handlers, user input and running the main loop of the application.

UIKit also provides several other features: [8]

- Handling touch and motion events.
- iCloud integration
- Cut, copy and paste support
- Accessibility support for disabled users
- Support for the Apple Push Notification service

### 4.4.3 iOS SDK

The iOS SDK is the iOS Software Development Kit released by Apple. The SDK provides the core frameworks like Cocoa Touch, Core Services, TCP/IP and power management for iOS.
The SDK is used along with the Xcode toolchain for developing and compiling for iOS and also contains the iOS Simulator.
Each new iOS version is released along with a new version of the SDK.

## 4.5 The Apple publishing process

In the Apple ecosystem the applications operate in a very restricted environment.
As opposed to Android all applications on an iOS unit must be installed through the Apple App Store. This means that no untrusted application can be installed by downloading it from the Internet or in any other way.
While with the Android Marked (The Android application distribution channel) anyone can distribute anything at any time (with the exception of confirmed malware), the App Store uses an extensive license agreement which dictates what is allowed and what is not. [6]
This means that an application might be rejected for reasons such as:

- Bad design
- Only applies to a niche audience
- Not enough functionality

The license agreement is an 80 page legal document covering everything from design requirements to how and what APIs are allowed to use. While these restrictions make the platform more resilient against malware it makes distribution of more outlying applications such as the Emerald application much harder.

## 4.6 Summary

In this chapter, we explain the history of iOS from its inception in 2007 to iOS 9.0 in June, 2015. We explain that to develop and deploy Applications for iOS you need a OSX installation, Xcode and a Apple developer account. We also describe the core frameworks used when developing iOS applications: the foundation framework and UIKit framework. Finally we explain the publishing process involved in distributing applications on the App Store and the advantages and disadvantages this include.

# Part III

# Porting Emerald

# Chapter 5

# Porting Emerald

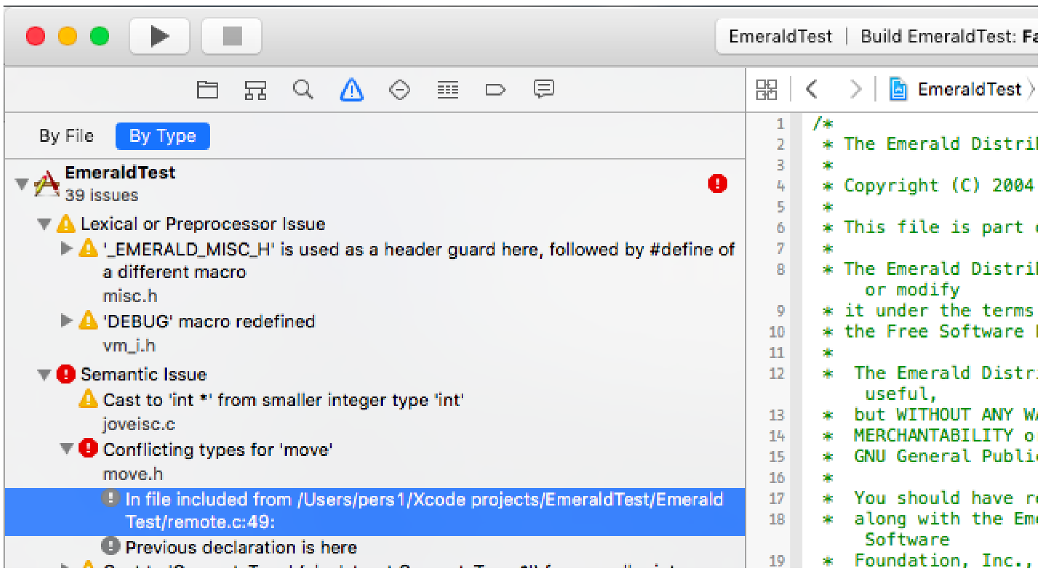
In this chapter, we explain the need to port Emerald to iOS as opposed to just executing a precompiled binary of Emerald.
We test different strategies of making Emerald run on the device without modifying it, and we explain some of the limitations the iOS platform gives us as developers.
Further we discuss some of the compromises we have to make to run Emerald on an iOS device such as sideloading and jailbreaking, and explain some of the advantages and disadvantages of these. We also jailbreak an iPhone and use the iPhone SDK for cross-compiling the Emerald source code for running on the arm architecture.

## 5.1 Running Emerald on an unmodified device

Emerald is an application best run from an console which means it would be very helpful to have a terminal available to test the application from when we are in the proccess of porting it.
iOS does not provide a terminal for users or developers, like the adb for Android (Android Debug Bridge, a debug console). This makes testing the binary significantly harder, and means we have to run the binary from an iOS application, controlling the execution of Emerald from the iOS application.

Using the Foundation Framework, we can normally run another program as an subprocess of the current application by using the NSTask class. In the iOS SDK as opposed to OSX, the header file for NSTask is not included in the distribution, meaning we cannot use this part of the API. The functionality of NSTask is still included, but the header is removed from the iOS SDK. One possibility would be to simply copy the header from an OSX environment and bundle it with the application along with the Emerald binaries, but because it is removed this means its a part of what Apple calls the private API, and usage of the private API is prevented by the license agreement for the App Store.

The first attempt to mitigate this was to simply dump the source code for Emerald into Xcode along with the code for the iOS application and try to let Xcode compile Emerald alongside the application. Emerald has a fairly complex build cycle so this turned out to be very hard to do, and after some more research it became clear that the only way to make an application use code added like this if the code is a statically linked library. Xcode compiles the library before the application and then links the library to the application, ruling out the ability to do this with a normal executable file.



EmeraldTest
39 issues
Lexical or Preprocessor Issue
'_EMERALD_MISC_H' is used as a header guard here, followed by #define of a different macro
misc.h
'DEBUG' macro redefined
vm_i.h
Semantic Issue
Cast to 'int *' from smaller integer type 'int'
joveisc.c
Conflicting types for 'move'
move.h
In file included from /Users/pers1/Xcode projects/EmeraldTest/Emerald Test/remote.c:49:
Previous declaration is here

```
/*
 * The Emerald Distri
 *
 * Copyright (C) 2004
 *
 * This file is part
 *
 * The Emerald Distri
     or modify
 * it under the terms
 * the Free Software
 *
 *  The Emerald Distr
     useful,
 *  but WITHOUT ANY W
 *  MERCHANTABILITY o
 *  GNU General Publi
 *
 *  You should have r
 *  along with the Em
     Software
 *  Foundation, Inc.,
```

Figure 5.1: A failed attempt at making Xcode compile Emerald automatically

Even if we could bypass the NSTask problem, there is several terms in the license agreement that disagree with some of Emerald's core functionality and prevents any such application from passing the App Store review and landing in the App Store.

Mainly paragraph 3.3.2 [6]:

*" ...an Application may not download or install executable code. Interpreted code may only be used in an Application if all scripts, code and interpreters are packaged in the Application and not downloaded... "*

This defeats the whole purpose of an Emerald interpreter because some of its core functionality is distributing and sharing executable code.
It is clear at this point that because of the strict license agreement we will not be able to distribute the application in the official App Store or any other way without some workaround.

### 5.1.1 Bypassing the App Store

A possible solution for making Emerald easier to compile and run on an iOS device would be to somehow bypass the App Store and thus bypassing the limitations set by the license agreement.
One way to achieve this could be to distribute the iOS application with the bundled Emerald binaries directly from Xcode in the same way that it is done when debugging and testing an application. This is called sideloading.
Sideloading allows users to install applications that breaks the license agreement but that would still run on an unmodified device. This would remove the problems concerning the license agreement and allow us to use the private API.

Sideloading the application would also give rise to some problems concerning the testing and distribution of the application:

- We would still lack a proper console to test if the compilation of Emerald is successful before we build the iOS application.

- Any special permissions the application might need to run could not be given beyond the normal permissions the application sandbox gives it.

- To download and use the application a user would need:
  - The source code of the application.
  - Access to a computer running OSX.
  - A Xcode installation and basic understating of how it works

As of June 2015 you no longer need to pay the license fee for the developer program to test applications on an device, an apple account is all that is needed. That means that to sideload an application all you need is access to an Xcode environment.
Before we can test sideloading we need to have an Emerald binary that is compiled for the iOS architecture.

### 5.1.2 Cross-compiling Emerald

We cannot simply transfer the precompiled binaries from a computer to the device and run it, because the iPhone uses a different architecture than most computers.
We need to cross-compile the source into an arm binary. Depending on the device this means compiling for armv7, armv7s or arm64.
Cross-compilation is using a cross-compilation enabled compiler running on one architecture, and compiling a executable for running on a different architecture with a different instruction set.

When compiling for the iPhone it makes sense to use the compiler and the headers included in the iPhone SDK (Software Development Kit) which supports cross-compiling for all arm versions.
Because Emerald uses Automake, a GNU tool for automatically generating makefiles, we can create a script that tries to set the compiler and the right environment variables thus compiling Emerald correctly.
An reduced version of a script that can be used to compile Emerald for armv7:

```
arch=armv7s
host=arm-apple-darwin11
prefix="/Users/pers1/em_output_new/"

```

```
export DEVROOT="/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/
        Developer"
export SDKROOT="$DEVROOT/SDKs/iPhoneOS8.3.sdk"

export CC="/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/
        usr/bin/clang"
export CFLAGS="-arch␣${arch}␣-std=c99␣-isysroot␣$SDKROOT␣-isystem␣$SDKROOT/usr/include"

./configure \
    --prefix="$prefix" \
    --host="${host}" \
    --enable-static \
    --disable-shared \
```

Before configuring and creating the makefiles we specify what architecture we want to compile for instad of using the architecture we are running the compiler on. We use the *clang* compiler included in the iPhone SDK, and we also specify sysroot to point to the SDK so the compiler looks for libraries and headers from where the SDK is located.

Running this script is successful but running *make* to actually compile the binaries fails. The reason for the failure is that clang builds code in GNU C11 mode meaning it uses standard C99 semantics. The Emerald source code uses *inline* with the semantics from GNU C89 (C89 with extensions), which differs from the ones in C99. While in GNU C89 inline is a hint to the compiler that this function should be inlined, in C99 it means that this is the inline *version* of the function, and for this to be correct C99 code there needs to be a corresponding non-inlined function somwhere in the code. This problem is easily corrected with just removing the inline keyword from the applicable functions, and letting clang optimize as desired.
Make now runs without errors and we get a complete armv7 binary we can run on any armv7 device.

### 5.1.3 Sideloading

We now have a binary we can try to execute on the device.
And because we don't have to adhere to the license agreement we can now make use of NSTask. To use NSTask we copy the header file `NSTask.h` from the OSX distribution of Foundation and add it to the iOS project. NSTask objects can now be created and used like normal.

With the compile script we compile one binary for each iOS architecture and one for the iOS Emulator (armv7, armv7s, arm64 and i386) and add them to the project. Xcode supports running custom scripts during the build phase of an application so we use a bash script to copy the correct binary into the app bundle depending on which architecture Xcode is currently building for. Armv7 for the iPhone 4, arm64 for the iPhone 6 etc. The app bundle is all the compiled code and other application resources bundled as one file, and is what is installed on the device.

For NSTask to be able to run the binary we need to change the permissions of the file to executable.
An application only has read access to the data in the bundle so we need to copy the files to a folder where we have write access to change the file permissions. We use NSFileManager to copy the file from the bundle to the `Library` folder where we have write access. Here we use the `setAttributes` method from NSFileManager to set the file as executable.

The last thing we need to do is to set the `EMERALD_ROOT` environment variable to the root Emerald folder. Here we use the `setenv` function from the C standard library.

Now we should be able to create a new NSTask object, set its launch path to `emx` in the emerald root folder and execute it.
Unfortunately, this did not work as NSTask would always throw an exception with the message "launch path not accessible" even though we make sure the file is present and executable with the
`isExecutableFileAtPath` method from NSFileManager.
After some research it turns out that execution of binaries, forking and similar functions is prohibited by the sandboxed environment the iOS applications run in.

**App Sandbox**

The App Sandbox is an access control mechanism used by both OSX and iOS.
Sandboxing is a way to control what data and resources the application have access to. This is a way to prevent malicious code exploiting insecure applications by making sure the application only can access a set of predetermined resources.

In OSX, you can disable the sandbox or use entitlements to give the application access to the resources it needs, but all iOS applications always runs in the sandbox and we have very limited options when it comes to allowing access to resources.

As there is no way of executing the Emerald binary on an unmodified device we need to look at the last option available: jailbreaking.

### 5.1.4 Jailbreaking

Jailbreaking is a general term describing privilege escalation in closed devices such as an iPhone or other restricted hardware. iOS jailbreaking is a process where you install a modified set of kernel patches that allows you to install and run unsigned code. Jailbreaking also gives you root access to the device and access to the file system.
The main reasons for jailbreaking is to be able to install applications or tweaks to the operating system in ways Apple does not normally allow. It is also an easy way for developers to test ideas and play with the hardware in ways not previously possible.

iOS Jailbreaking tools generally comes in the form of an Windows, OSX or Linux application we run on a computer connected to the device, which installs the kernel patches for us.

Jailbreaking a device comes with many advantages when porting Emerald:

- Because we can install unsigned code and get access to the file system we can run a standard command line shell for testing.
- We can bypass the application sandboxing, meaning we can give the application the permissions it needs.
- Usage of private APIs like NSTask is possible.
- We can use alternative distribution channels bypassing the App Store, like Cydia which is a "App Store" for jailbroken devices.

While these advantages would help us greatly porting Emerald to iOS, jailbreaking also have two serious drawbacks:

- Jailbreaking becomes an prerequisite for running the application.
- While it is unlikely, there is no guarantee that jailbreaking is possible for all future iOS devices.

Making Emerald run on an unmodified iPhone was more complicated than previously thought. Both the Apple license agreement and the sandboxed architecture contains limitations when it comes to making an application like this. The biggest problem is the part of the license agreement that prohibits downloading or transferring executable code, and the sandbox preventing execution of binaries. Jailbreaking seems to be the best and probably only way to solve the problems with the restrictive environment.
The only prerequisite would be an jailbroken device as opposed to having access to OSX, Xcode and an apple developer account.

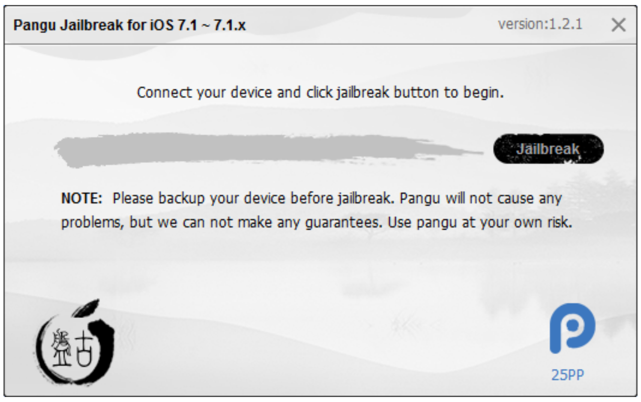
We also get access to using Cydia to distribute the application in much the same way as the App Store. Cydia is the *de facto* distribution channel for applications for jailbroken devices. Because Cydia is built using Debian APT we can set up dependency trees where all users would download the same iOS application, but phones with different architectures would download the correct Emerald binary for the corresponding architecture (armv7, armv7s, arm64). We also have the option of downloading a console for testing the Emerald binary without the need for an iOS application acting as a layer.

## 5.2 Porting Emerald to a jailbroken device

When Jailbreaking the iPhone it is important to determine what iOS version the iPhone is running. This is to make sure that we use the correct jailbreaking software for the iOS version. Because our test phone is an iPhone 4 running iOS 7.1 we choose to use the PanGu jailbreak tool. [13]
The process is simple:

- Install the PanGu jailbreak tool
- Install iTunes (a requirement for the Windows version of PanGu)
- Connect iPhone

- Let the jailbreak tool run to completion

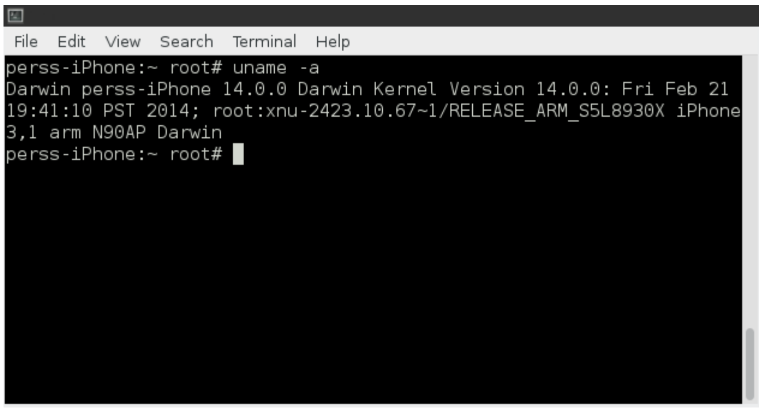


Connect your device and click jailbreak button to begin.

NOTE: Please backup your device before jailbreak. Pangu will not cause any problems, but we can not make any guarantees. Use pangu at your own risk.

Figure 5.2: The PanGu jailbreak tool

When the jailbreak is complete we need to reboot the iPhone and see that Cydia is installed as an application. Using Cydia we can install openSHH which gives us access to a console on the device by remote connecting to it through SSH. The jailbreaking was successful, in figure 5.3 we see a terminal connection to the jailbroken device over SSH. Be-

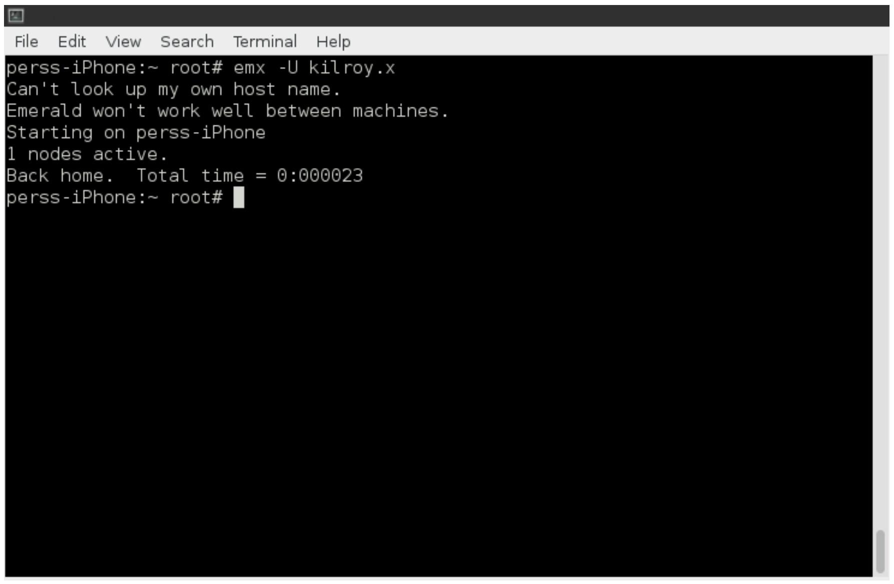


```
perss-iPhone:~ root# uname -a
Darwin perss-iPhone 14.0.0 Darwin Kernel Version 14.0.0: Fri Feb 21
19:41:10 PST 2014; root:xnu-2423.10.67~1/RELEASE_ARM_S5L8930X iPhone
3,1 arm N90AP Darwin
perss-iPhone:~ root#
```

Figure 5.3: SSH connection to the jailbroken iPhone

cause we now have access to the file system and a console on the device we can install Emerald by creating a `/bin` folder in the root directory and copying the Emerald binaries here.
To run an Emerald program we simply run `emx` from the console. Running `ec` to compile a program does not work because the tr command line tool is missing. To use the compiler we can manually run emc , or

even better, install tr from Cydia. Compiling Emerald for arm was successful as seen in figure 5.4. It even works when distributing objects between machines!



```
perss-iPhone:~ root# emx -U kilroy.x
Can't look up my own host name.
Emerald won't work well between machines.
Starting on perss-iPhone
1 nodes active.
Back home.  Total time = 0:000023
perss-iPhone:~ root# 
```

Figure 5.4: Emerald running successfully on iOS. The error messages is caused by network restrictions, not the application.

We now know that porting Emerald to iOS is possible even though we had to make some compromises making it run.
With jailbreaking we loose the ability to install Emerald at any time through the App Store, but this is a compromise that have to be done to make it possible to run at all.
The iPhone SDK provides us with a toolchain that enables us to compile any preexisting source code for running on an arm architecture and in the end enabling us to run it on an iPhone.

## 5.3 Summary

In this chapter we explain the need to cross compile Emerald to make it run on a iOS Device, and cross compiled it using the iOS SDK. We try to sideload an application to bypass the App Store but are prevented from running Emerald due to the app sandbox. We jailbreak an iOS Device to mitigate the restrictions set by the license agreement and the sandbox, and successfully run Emerald in this environment.
Currently the only way to run Emerald is by remote connecting to the device through SSH as there is no command line shell running on the actual device. This is why we need to create a native iOS application that acts as a layer on top of Emerald, with an interface we can use on the actual device making it easier to run.
Porting the Emerald runtime and compiler using the iPhone SDK is a success, but some simple changes to the source code had to be made.

# Chapter 6

# iOS application

In this chapter, we describe the implementation of the iOS application. The application gives the user an interface running on the device avoiding the need for an terminal.
We go through the different views the application is built from and explain the code and how we stitch them together. The application acts as a layer by executing the Emerald binary and presenting the output in the user interface. The application has a *terminal like* text view acting as the Emerald console. We use the Xcode storyboard for implementation of the user interface and implement the different view controllers using Objective-C. The application is called iOS Emerald.

## 6.1 Overview

The user interface of the application consists of 5 views:

- Main view
- Compiler view
- Connection preferences view
- Program choose view
- Emerald console view

The design of the views are mainly done in the storyboard interface designer but all the implementation specific code is put in subclasses of the view types. The design itself is relatively uninteresting and the process differs very little between projects, so we focus more on the implementation specific code in this chapter.

In figure 6.1 we show an overview of the program flow with the arrows representing the segues between the views.
In the following sections, we explain each views function and layout in more detail.

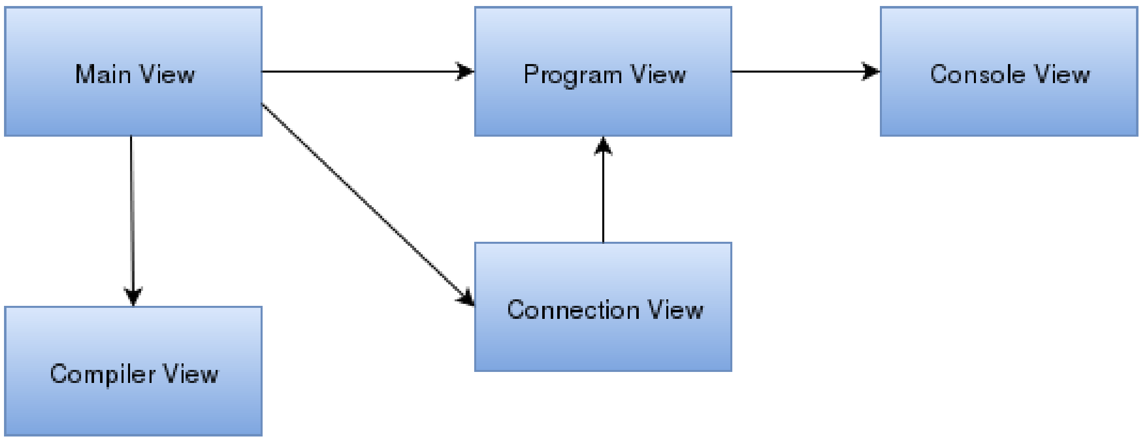


Figure 6.1: An overview of the program flow.

## 6.2 Main menu

The main view is the first view we see when we open the application, it gives us 3 choices as seen in figure 6.2.

- Host session — Starts an Emerald session without connecting to any other nodes. Shows the program choose view.
- Connect to session — Start an Emerald session with user provided connection information. Shows the connection info view.
- Compiler — Shows the compiler view.

The code for the main view is contained in `MainViewController.m` and is relatively uninteresting as the segues between the views is handled by the storyboard.

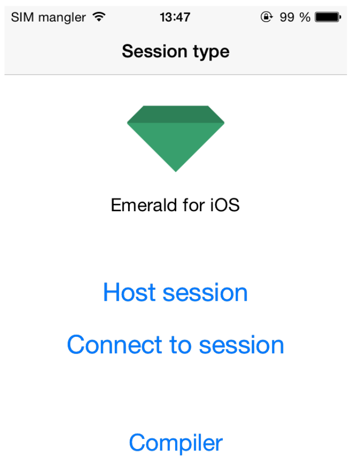


Figure 6.2: The main view.

## 6.3 Compiler view

The compiler view lets us compile Emerald programs directly on the device. Each application in the iOS environment have access to its own documents folder. This is where the application can store files which the users needs access to, and the user can transfer files to this folder for the application to use. The compiler view reads all the files in this folder and presents us with a list of the ones ending with the *.m* extension.
Because the compiler view is a `UITableView` the interesting code is contained in the table cells, the actual view only finds the Emerald files and populates the table with one cell for each file.

In the cell we have a button that calls the `compileFile` function. `compileFile` creates a `NSTask` and runs the Emerald compiler.
We bind the `NSTaskDidTerminateNotification` to the `compileDone` function which shows the outcome of the compilation as a label with the text *Success* or *Failure*. By having a `NSTask` instance for each cell, we can compile several files concurrently. If the files are compiled successfully they are saved to the documents folder for use by the Emerald interpreter.

```
#import "CompileCellTableViewCell.h"
#import "NSTask.h"

@interface  CompileCellTableViewCell ()
@property NSTask *task;
- (void)compileDone;
@end
@implementation CompileCellTableViewCell



- (BOOL)compileFile {
    // alloc task
    self.task = [[NSTask alloc] init];

    // set path
    self.task.launchPath = @"/bin/ec";

    // set arguments
    [self.task setArguments:[NSArray arrayWithObjects:self.filePath, nil
            ]];

    // Add callback for completion
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self
           selector:@selector(compileDone)
               name:NSTaskDidTerminateNotification
             object:self.task];

    // Hide button
    self.compileButton.hidden = YES;

    // Show progress icon
    self.activity.hidden = NO;
    [self.activity startAnimating];

    // Launch compiler
    [self.task launch];
    return YES;
}

- (void)compileDone {
    int status = [self.task terminationStatus];

    self.activity.hidden = YES;
    self.compileButton.hidden = NO;
    // Success
    if(status == 0) {
        [self.compileButton setTitleColor:[UIColor greenColor] forState:
                UIControlStateNormal];
        self.compileButton.titleLabel.text = @"Success";
    } else { // Failure
        [self.compileButton setTitleColor:[UIColor redColor] forState:
                UIControlStateNormal];
        self.compileButton.titleLabel.text = @"Failure";
    }
}

@end
```

## 6.4 Connection view

The connection view is where we can give Emerald a hostname or IP-address and a port to connect to already running node. The connection view is a simple view containing two instances of `UITextField` embed-

ded in a `UIScrollView`.
When we perform the segue from this view we create an array with the connection details the user provided and pass it along to the program chooser view which in turn passes it along to the console view.

```objc
#import "ConnectionInfoViewController.h"
#import "ChooseProgramViewController.h"

@interface ConnectionInfoViewController ()
@property (weak, nonatomic) IBOutlet UITextField *hostname;
@property (weak, nonatomic) IBOutlet UITextField *port;
@end

@implementation ConnectionInfoViewController

- (void)viewDidLoad {
    ....
}

- (void)didReceiveMemoryWarning {
    ....
}


#pragma mark - Navigation
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    ChooseProgramViewController *programView = [segue
            destinationViewController];

    if(self.hostname.text.length != 0 && self.port.text.length != 0) {
        programView.connectionInfo = [[NSArray alloc] initWithObjects:
                self.hostname.text, self.port.text, nil];
    } else {
        programView.connectionInfo = nil;
    }
}

@end
```

## 6.5 Program view

The program view is presented when connecting to a session or hosting one. This is where we choose what program to start the Emerald

interpreter with. The view consists of a `UIPickerView` and a button for continuing. The `UIPickerView` is populated with the filenames of the files ending with the *.x* extension in the documents folder. This is performed in the `forComponent:` function for the picker view.
When a user scrolls to the program he wishes to run, `didSelectRow:` is called and we set the chosen program here. Finally when the continue button is pressed we pass the program info along to the console view along with the connection info we may have got from the connection view.

```
#import "ChooseProgramViewController.h"
#import "Utilities.h"
#import "ConsoleViewController.h"

@interface ChooseProgramViewController () <UIPickerViewDataSource,
       UIPickerViewDelegate>
@property (weak, nonatomic) IBOutlet UIPickerView *picker;
@property NSMutableArray *emeraldFiles;
@property NSString *chosenProgram;
@end

@implementation ChooseProgramViewController

- (void)viewDidLoad {
    ...
    self.emeraldFiles = [NSMutableArray arrayWithArray:[Utilities
           getDocumentsFileListByFileExtention:@".x"]];
    [self.emeraldFiles insertObject:[NSArray arrayWithObjects:@"", @"<
           none>", nil] atIndex:0];
}

- (NSString *)pickerView:(UIPickerView *)pickerView titleForRow:(
       NSInteger)row forComponent:(NSInteger)component {
    return self.emeraldFiles[row][1];
}

....

// returns the # of rows in each component..
- (NSInteger)pickerView:(UIPickerView *)pickerView
       numberOfRowsInComponent:(NSInteger)component {
    return self.emeraldFiles.count;
}

```

```
// set chosen program
- (void)pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger)
       row inComponent:(NSInteger)component {
    self.chosenProgram = self.emeraldFiles[row][0];
}

.....

#pragma mark - Navigation
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    ConsoleViewController *consoleView = [segue
           destinationViewController];
    consoleView.connectionInfo = self.connectionInfo;
    consoleView.program = self.chosenProgram;
}

@end
```

## 6.6 The Emerald console

The console is where we run the Emerald interpreter showing the output of the program and taking input from the user. This view is by far the most complex view in the application as it deals with subclassing components, task execution, input/output piping and preventing some default iOS behavior. We have broken it down into three parts and designated one section explaining each part.

### 6.6.1 The console

The console is a subclassed `UITextView` with a black background and monospaced font for resembling a console.
Writing to the console directly is disabled meaning we have to explicitly enable the keyboard from the surrounding view controller. We have extended the the text view with three methods: write, append and clear. Write is a private function setting the actual text of the view. Append is a public function that appends text to the console. Clear removes all text in the view. We have overloaded `scrollRectToVisible` disabling animations so the automatic scrolling of the view looks smooth.
The behavior of the console is controlled from the parent view controller.

```
#import "Console.h"

@implementation Console
- (void)write:(NSString *)text {
    // To avoid resetting text formatting and losing text color
    if (![self isSelectable]) {
        self.selectable = YES;
    }

    // Set text
    self.text = text;

    // To avoid resetting text formatting and losing text color
    if (![self isSelectable]) {
        self.selectable = NO;
    }
}

// Append to the console
- (void)append:(NSString *)text {
    NSString *newText = [self.text stringByAppendingString:text];
    [self write:newText];
}

// Clear the console
- (void)clear {
    [self write:@""];
}

// For automatic scrolling
- (void)scrollRectToVisible:(CGRect)rect animated:(BOOL)animated {
    [super scrollRectToVisible: rect animated: NO];
}

@end
```

### 6.6.2 Emerald runner

The Emerald runner is the class that takes care of the initialization, running and stopping of the Emerald interpreter. It contains three public methods: `initialize`, `run` and `stop`.
`initialize` checks the given connection/program data and builds command line arguments for Emerald with it. It creates a input pipe for

writing and a output pipe for reading and assigns them to the Emerald task.
A callback notification is registered on the output pipe so the parenting view controller is given a notification when there is data on the pipe. We set the launch path to the Emerald binary and also set the appropriate command line arguments.
Run simply launches the task and `stop` sends a SIGTERM signal to the task and waits for it to exit.

```
#import "EmeraldRunner.h"
#import "NSTask.h"

@import Foundation;

@interface EmeraldRunner ()
@property NSString *hostPort;
@property NSPipe *inputPipe;
@property NSPipe *outputPipe;
@property NSTask *task;
@property NSString *program;
@end

@implementation EmeraldRunner
- (void)initalize:(NSString *)host portNumber:(NSString *)port
       callbackObject:(ConsoleViewController *)callback program:(
       NSString*)program {

    // Create host:port string
    if(host.length != 0 || port.length != 0) {
        self.hostPort = [NSString stringWithFormat:@"%@:%@", host, port
                ];
    } else {
        self.hostPort = @"";
    }

    // Set program
    if(program == nil) {
        self.program = @"";
    } else {
        self.program = program;
    }

    // Initalize pipes
    self.inputPipe = [NSPipe pipe];
    self.outputPipe = [NSPipe pipe];
    self.output = [self.outputPipe fileHandleForReading];
    self.input = [self.inputPipe fileHandleForWriting];

    // Add callback to console view, for when there is output on pipe
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:callback selector:@selector(notifiedForOutput:) name
           :NSFileHandleReadCompletionNotification object:self.output];

    // Create task and assign pipes
    self.task = [[NSTask alloc] init];
    [self.task setStandardInput:self.inputPipe];
    [self.task setStandardOutput:self.outputPipe];

    // Set binary to launch (emx)
    self.task.launchPath = @"/bin/emx";

    // Set arguments
    if(![self.program  isEqual: @""]) {
        [self.task setArguments:[NSArray arrayWithObjects:@"-U",
                                 [NSString stringWithFormat:@"-R%@",
                                        self.hostPort],
                                 [NSString stringWithFormat:@"%@", self.
                                        program],
                                 nil]];
    } else {
        [self.task setArguments:[NSArray arrayWithObjects:@"-U",
                                 [NSString stringWithFormat:@"-R%@",
                                        self.hostPort],
                                 nil]];
    }

    // Notify callback when there is data in pipe
    [self.output readInBackgroundAndNotify];
}

- (void) run {
    // Launch emerald
    [self.task launch];
}

- (void) stop {
    // Kill the current session
    kill([self.task processIdentifier], SIGTERM);
    [self.task waitUntilExit];
}
@end
```

### 6.6.3 Console view controller

In console view controller is where the magic happens, it connects the console view and the Emerald runner and takes care of data flowing between the two.
In the `viewDidLoad` method we initialize the console and the Emerald runner, and we pass the data we have been given from the previous views to the Emerald runner.
We start the Emerald runner and append the devices current IP-address to the console.

The `notifiedForOutput` method is the callback method for the output pipe of the Emerald runner. We are notified when there is data on the output pipe by this method being called. Here we fetch the data from the pipe and append it to the console, we scroll the console down to the most recent line and tell the output pipe to keep listening.

The `toggleButton` method is the action for the toggle keyboard button in the user interface and simply calls the `toggleKeyboard` method. `toggleKeyboard` sets the console as editable and makes it the responder for keyboard input making the keyboard appear.

`shouldChangeTextInRange` is a method that is called when the text in an `UITextView` changes by user input. In this case that means when the user inputs some text to the console. It is called for each character the user writes so we must append this character to `self.currentCommand` and when the user presses return (`\n`) we write this command to the input pipe of the Emerald runner.

Finally `prepareForSegue` is called when we navigate away from the view. Here we kill the runner task, preventing it from running in the background. We could have implemented functionality for running several Emerald processes in the background, but chose not to for simplicity.

```
#import <UIKit/UIKit.h>
```

```

@interface ConsoleViewController : UIViewController
@property NSArray *connectionInfo;
@property NSString *program;
-(void) notifiedForOutput: (NSNotification *)notified;
@end
```

```
#import "ConsoleViewController.h"
#import "Console.h"
#import "EmeraldRunner.h"
#import "Utilities.h"

@interface ConsoleViewController () <UITextViewDelegate>
- (void) toggleKeyboard;
@property (weak, nonatomic) IBOutlet Console *console;
@property EmeraldRunner *runner;
@property BOOL keyboardVisible;
@property NSString *currentCommand;
@end
@implementation ConsoleViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // init
    self.currentCommand = [[NSString alloc] init];

    // Set self as delegate for console view
    [self.console setDelegate:self];
    self.keyboardVisible = NO;

    // Hide back button. We have a custom handler
    self.navigationItem.hidesBackButton = YES;

    // Allocate runner
    self.runner = [[EmeraldRunner alloc] init];

    // Clear console
    [self.console clear];

    // Initalize runner
    // Check if we have connection info
    if (self.connectionInfo != nil) {
        NSLog(@"%@:%@", self.connectionInfo[0], self.connectionInfo[1]);
        [self.runner initalize:self.connectionInfo[0] portNumber:self.
               connectionInfo[1] callbackObject:self program:self.
               program];
    } else {
        [self.runner initalize:@"" portNumber:@"" callbackObject:self
               program:self.program];
    }
    [self.runner run];
    // Append ip to console
    [self.console append:[NSString stringWithFormat:@"Running␣with␣IP:␣%
           @\n", Utilities.getIPAddress]];
}

- (void)didReceiveMemoryWarning {
    ....
}

-(void) notifiedForOutput: (NSNotification *)notified
{
    // Get data
    NSData * data = [[notified userInfo] valueForKey:
           NSFileHandleNotificationDataItem];

    if ([data length]){
        // Create string from data
        NSString * outputString = [[NSString alloc] initWithData:data
               encoding:NSUTF8StringEncoding];

        // Write to console
        [self.console append:outputString];

        // Keep listening
        [self.runner.output readInBackgroundAndNotify];
    }
    [self.console scrollRangeToVisible:NSMakeRange(self.console.text.
           length-1, 1)];
    // Hack for making scrolling smooth
    self.console.scrollEnabled = NO;
    self.console.scrollEnabled = YES;
}

// Listener for keyboard button
- (IBAction)toggleButton:(id)sender {
    [self toggleKeyboard];
}

// Detect changes in TextView
- (BOOL)textView:(UITextView *)textView shouldChangeTextInRange:(NSRange
        )range replacementText:(NSString *)text {

    // Delete
    if(range.length == 1) {
        if (self.currentCommand.length > 0) {
            self.currentCommand = [self.currentCommand substringToIndex:
                self.currentCommand.length-1];
        }
    } else if ([text  isEqual: @"\n"]) { // New line
        // Send command to Emerald
        self.currentCommand = [self.currentCommand
                stringByAppendingString:text];
        [self.runner.input writeData:[self.currentCommand
                dataUsingEncoding:NSUTF8StringEncoding]];
        // Clear command
        self.currentCommand = @"";
    } else { // Append new character to command string
        self.currentCommand = [self.currentCommand
                stringByAppendingString:text];
    }

    return YES;
}

// Enable or disable keyboard
-(void) toggleKeyboard {
    if (self.keyboardVisible) {
        self.console.editable = NO;
        [self.console setUserInteractionEnabled:YES];
        [self.console resignFirstResponder];
    } else {
        self.console.editable = YES;
        [self.console setUserInteractionEnabled:NO];
        [self.console setSpellCheckingType:UITextSpellCheckingTypeNo];
        self.console.autocorrectionType = UITextAutocorrectionTypeNo;
        [self.console becomeFirstResponder];
    }
    self.keyboardVisible = !self.keyboardVisible;
}

#pragma mark - Navigation

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    // Kill the emerald task
    [self.runner stop];
}

@end
```

## 6.7 Summary

In this chapter, we describe the implementation of the Emerald iOS application. We describe how we design the user interface and explain in detail how we implemented each of the five views the application uses. We explain how we subclassed and modified a `UITextView` making it look like a console, and how we use pipes for redirecting the input and output of the Emerald process.

# Chapter 7

# Distribution with Cydia

We now have a working Emerald binary and an iOS application for easy interaction. The last step is distributing both the binary and the iOS application.
In this chapter we explain how we package the applications we want to distribute and how we create an APT repository.

Because we cannot use the App Store and have a jailbroken device, using Cydia is very convenient.
Cydia is an application for jailbroken devices that allows users to install custom software packages. Cydia is built upon APT (Advanced Packaging Tool), and is basically a graphical interface for this.
This means that all we need to do to distribute Emerald is setting up a APT repository.

## 7.1 Creating the packages

To distribute the files we need to package them as `.deb` files.
This means running the `dpkg-deb` tool on a folder that has the follwing structure:

- The files and folders as they should appear in the file system
- A DEBIAN folder with a file called `control` containing the package information

The following is the folder structure for one of the Emerald binary packages:

```
EmeraldArm7
bin
    -ec
    -emc
    -emx
    -emxx
    -gencctab
DEBIAN
    -control
include
    -array.h
    -extract.h
    ...
lib
    -bcdef
    -Builtins
    -ccdef
    -Compiler
    -jsdef
```

The `control` file contains information about the package with the `architecture` key being the most important:

```
Package: link.pers.emeraldarmv7
Name: Emerald
Version: 1.0
Architecture: iphoneos-arm
Description: Emerald binaries for armv7
Homepage:
Depiction:
Maintainer: audunjoy@uio.no
Author: Audun Oygard
Sponsor:
Section: Development
```

We do this with all the binaries for the different architectures and the iOS application resulting in 4 deb files to distribute.

## 7.2 Hosting the repository

The last step of the process is hosting the packages on a web server.
We make an `emerald` end point on the web server and put the packages we created there.
By running the `dpkg-scanpackages` tool a package index is created in

the same folder and is used by clients when they search for packages. The last thing we do is compressing the package index with bzip2 and the repository is ready for use.

To install the packages from Cydia we simply add the web server end-point as a source in Cydia and as seen in figure 7.1 it lists the packages we can install.

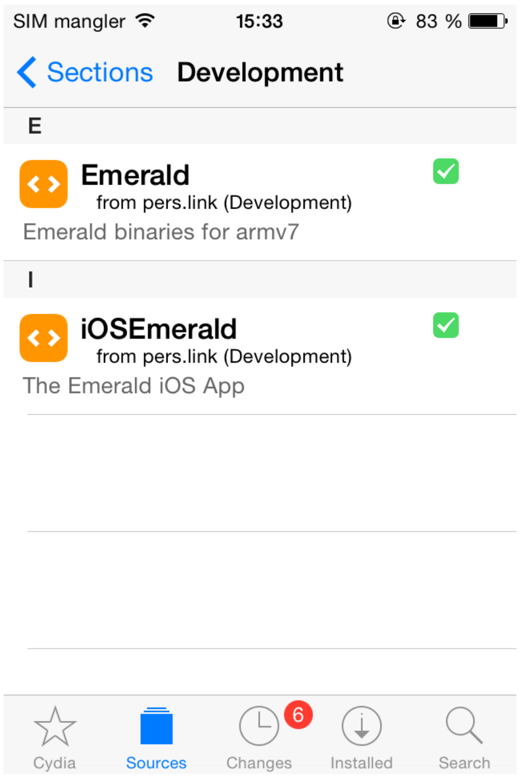


Figure 7.1: Cydia showing the Emerald packages for installation

## 7.3 Summary

In this chapter, we explain how we make an APT repository for distributing the Emerald binaries and application with Cydia. We explain how we package the applications using `dpkg-deb` and how we use `dpkg-scanpackages` to create a package index on a web server.

The next chapter describes the results of testing iOS Emerald with distribution of objects through Planetlab and remote servers.

# Chapter 8

# Performance and Evaluation

In this chapter we discuss what and how we want to test and evaluate the Emerald implementation. We test the basic performance of the implementation and also test the distribution capabilities of the implementation. We try to find limitations of the implementation and compare it to other Emerald implementations.
Finally we briefly evaluate iOS as a development platform.

## 8.1 The tests

We run two sets of tests. One set of tests to evaluate the basic performance of the Emerald implementation on iOS, and one set of test where we evaluate the distribution performance. The basic performance tests is meant to identify the basic capabilities of the implementation without any distribution while the distribution tests evaluate the basic distribution capabilities.

### 8.1.1 Basic performance

#### The testbeds

In the basic performance tests we use 3 devices to run the tests. A iPhone 4, a Samsung Galaxy Fame S6810 and a Macbook Air with specifications seen in table 8.1. For testing on Android we had to compile Emerald for armv7 with the Android NDK in much the same way we did for iOS in chapter 5. While there is a previous Emerald implementation for Android it turns out that this implementation uses a different Emer-

Table 8.1: Performance specifications for basic performance tests

| | **Processor** | **RAM** |
|---|---|---|
| iPhone 4 | 1.0 GHz Cortex-A8 | 512 MB |
| Samsung Galaxy | 1.0 GHz Cortex-A9 | 512 MB |
| Macbook Air 2014 | 1,4 GHz Intel Core i5 | 8 GB |

ald version (0.99) than the iOS version of Emerald (1.06), and that the two where not always compatible.

**Test cases**

The basic performance tests we have 3 test cases:

**Stress tests:**
The stress test have two test cases:

- Maximum supported threads
- Thread spawn time

In the first test we incrementally spawn threads until Emerald crashes or runs out of memory.
The second test measures the time the platform uses to spawn 1000 threads.

In Emerald threads is called processes and each thread is bound to a Emerald object and is denoted by the `process` keyword. The threads start running as soon as the object it belongs to is initialized. This means that in the first test we continually create objects with a process until Emerald crashes, and in the second we measure the time it takes to spawn a thousand such objects.

**Calculation test:**
In the calculation test we use an Emerald program that calculates the first 50000 prime numbers and time this. This gives us a basic idea of the performance of the platform.

### 8.1.2 Evaluation criteria

We evaluate each test case by the following criteria:

- **Functionality and Stability:** We evaluate whether the functionality and stability is as expected from the iOS implementation. Do the programs run as expected, can we distribute normally etc.

- **Consistency:** We evaluate the results in terms of the consistency of the iOS samples.

- **Efficiency:** We evaluate the efficiency of the test compared to the other devices.

In all the test cases we run each test 20 times and present the average of the samples.

### 8.1.3 Distribution tests

#### The testbeds

For the distribution test we try to determine the basic distribution performance of the implementation using 4 different testbeds with specifications seen in table 8.2.

- **Phone-to-phone:** We test the distribution performance between two smartphones on a local network.

- **iPhone to local machine:** We test the distribution performance between one iPhone and a computer on a local network.

- **iPhone to remote machine:** We run the same distribution tests but with a remote computer.

- **iPhone to various Planetlab machines:** The same distribution tests on a set of Planetlab computers described in table **??**.

Table 8.2: Performance specifications

| **Device** | **Processor** | **RAM** | **Location** |
|---|---|---|---|
| iPhone 4 | 1.0 GHz Cortex-A | 512 MB | Local network |
| Samsung Galaxy | 1.0 GHz Cortex-A9 | 512 MB | Local network |
| Local computer | 1,4 GHz Intel Core i5 | 8 GB | Local network |
| Remote Computer | 2,4 GHz Intel Core 2 Duo | 1 GB | Falkenstein, Germany |

**Test cases**

The distribution tests have two test cases:

**Round-Trip-Time:**
We determine the average round-trip-time by moving an object to another node then back to the original node again. This gives us data on how much time is needed to move objects around on a given testbed.

**Break-even-point for computational offloading:**
Computational offloading is one of the most advantageous things we can do when using an distributed programming language on a mobile device, both to save time and save power on the mobile unit.
In this test we try to find the break-even-point for computational offloading. We incrementally calculate prime numbers both locally on the device and distribute the operation to a remote host and try to find the point where offloading the operation to the remote device is faster than performing the calculation on the device itself.

### 8.1.4 Evaluation criteria

We evaluate each test case by the following criteria:

- **Functionality and Stability:** We evaluate whether the functionality and stability is as expected from the implementation.

- **Consistency:** We evaluate the results in terms of the consistency of the samples.

- **Efficiency:** We evaluate the efficiency of the test compared to the other test beds.

In all the test cases we run each test 20 times and present the average of the samples.

## 8.2 Basic performance

In this section we test the basic performance of the implementation to identify some of the limitations Emerald has on iOS. These tests should help us to determine the usability of Emerald on iOS.

### 8.2.1 Thread spawning

In this test we try to spawn the maximum amount of threads the different test beds support.
Having a number on the maximum number of concurrent processes is useful when designing Emerald programs.

#### Results

| **Device** | **Number** | **Time** |
|---|---|---|
| Macbook Air | 65241 | 1200ms |
| iPhone 4 | 7894 | 6.94s |
| Samsung Galaxy Fame | 65241 | 20.5s |

#### Discussion

Both the Android and the OSX implementations have a maximum number of 65241 processes before the run time crashes. This is probably due to an integer overflow in the virtual machine.
The iOS implementation crashes after 7894 processes. This could be due to different sizes of the primitives between the Android NDK and the iOS SDK causing the stack to be smaller on iOS. It is worth mentioning that increasing the stack size of the virtual machine with the `-s` command line switch solves the issue and we can spawn more than 7894 processes.

**Evaluation**

**Functionality and Stability:** The test program runs as expected. No problems when starting processes.
**Consistency:** No Variation in the samples.
**Efficiency:** The iOS implementation performs worse than the other two implementations with 7894 maximum processes to 65241 on Android and OSX.

### 8.2.2 Thread spawn performance

In this test we determine the performance of spawning 1000 processes on the different test beds. This gives us an indication of the overhead of spawning processes with the iOS implementation.

**Results**

| **Device** | **Time** |
|---|---|
| Macbook Air | 4ms |
| Samsung Galaxy Fame | 20ms |
| iPhone 4 | 92ms |

**Discussion**

The iOS implementation performs worse than the Android and OSX implementations with 92ms per 1000 processes. While it is expected that both the iOS and the Android implementations perform worse than the OSX implementation it is noteworthy that iOS is almost five times slower than the Android implementation. This is partly due to some samples being much higher than the others, but overall the iOS implementation is slower.

**Evaluation**

- **Functionality and Stability Stable:** The test program runs as expected. No problem when spawning processes.
- **Consistency:** There is some inconsistencies in the samples that makes the iOS average somewhat higher.

- **Efficiency:** The iOS implementation performs worse than the other two with an average of 92ms for every 1000 processes.

### 8.2.3 Heavy calculation test

In this test we simply evaluate the computational performance of the implementation and compare it to the two other devices. This is done by an Emerald program that calculates the first 50,000 prime numbers and we present the average time of this operation.

#### Results

| **Device** | **Time ( avg.)** |
|---|---|
| Macbook Air | 3.72ms |
| Samsung Galaxy Fame | 42.89ms |
| iPhone 4 | 55.96ms |

#### Discussion

The iOS implementation performs worse than the two other test beds. As expected it is much slower than the OSX implementation, but also somewhat slower than the Android implementation. Again this is a result of the variation in the iOS samples. Some of the samples where more than 200% slower than the median. This is due to App Store updates or similar operations out of our control happening on the device while Emerald is running.

#### Evaluation

- **Functionality and Stability:** The test program runs as expected.
- **Consistency:** Generally consistent samples, but at times the phone slows down causing deviating samples.
- **Efficiency:** As expected against OSX and slightly slower than the Android implementation due to the deviating samples.

## 8.3 Distribution tests

In this section we test and evaluate the distribution performance of the Emerald implementation. This helps us determine if and when distribution of certain tasks can be beneficial.

### 8.3.1 Phone to phone

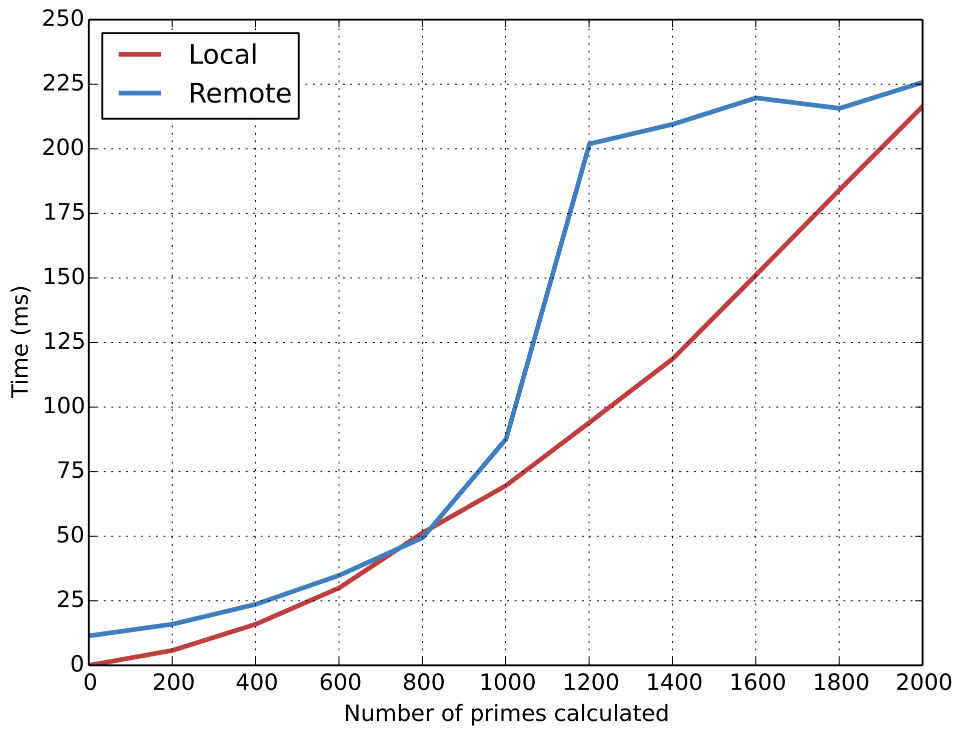
In this test we check the round-trip-time of distributing a object to the Android phone and back to the iPhone, and evaluate if and when computational offloading is useful. The round-trip-time says something about the general performance of distribution, and the computational offloading graph will tell us at what time computational offloading is viable.

**Results**

**Round-trip-time**

| **Testbed** | **Time** |
|---|---|
| Phone-Phone | 20.9ms |

**Computational offloading**



### Discussion

The round-trip-time is as expected compared to the other testbeds, slightly slower than distribution to a local more powerful device, and faster than distribution to a remote server.
In the computational offloading graph we see that this is not ever viable as the computation is always slower on the Android phone.

### Evaluation

- **Functionality and Stability:** Both test runs as expected.
- **Consistency:** Consistent samples.
- **Efficiency:** As expected for RTT and computational offloading not viable:.

### 8.3.2 Phone to local machines

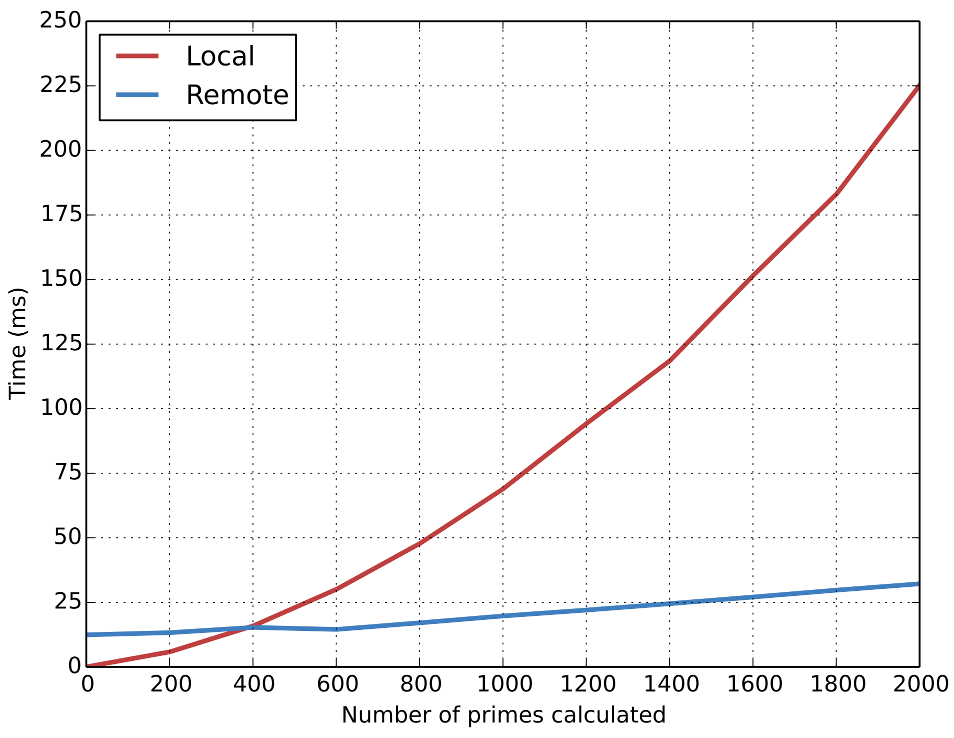
In this section we perform the same two tests as previously but distribute from the iPhone to a local machine.

#### Results

#### Round-trip-time

| **Testbed** | **Time** |
|---|---|
| Phone-Local Machine | 14,6ms |

#### Computational offloading



**Discussion**

The round-trip-time is faster when distributing to a local machine compared to the Android device on the local network. This is not surprising as the local machine have more computational power than the Android device.
The computational offloading graph shows us that if we have tasks where the execution time exceeds about 13ms it would be beneficial to distribute the task to the local machine and to the computation there.

**Evaluation**

- **Functionality and Stability:** The test program runs as expected.
- **Consistency:** Consistent samples.
- **Efficiency:** As expected for RTT and computational offloading viable for execution times exceeding 13ms.

### 8.3.3 Phone to remote server

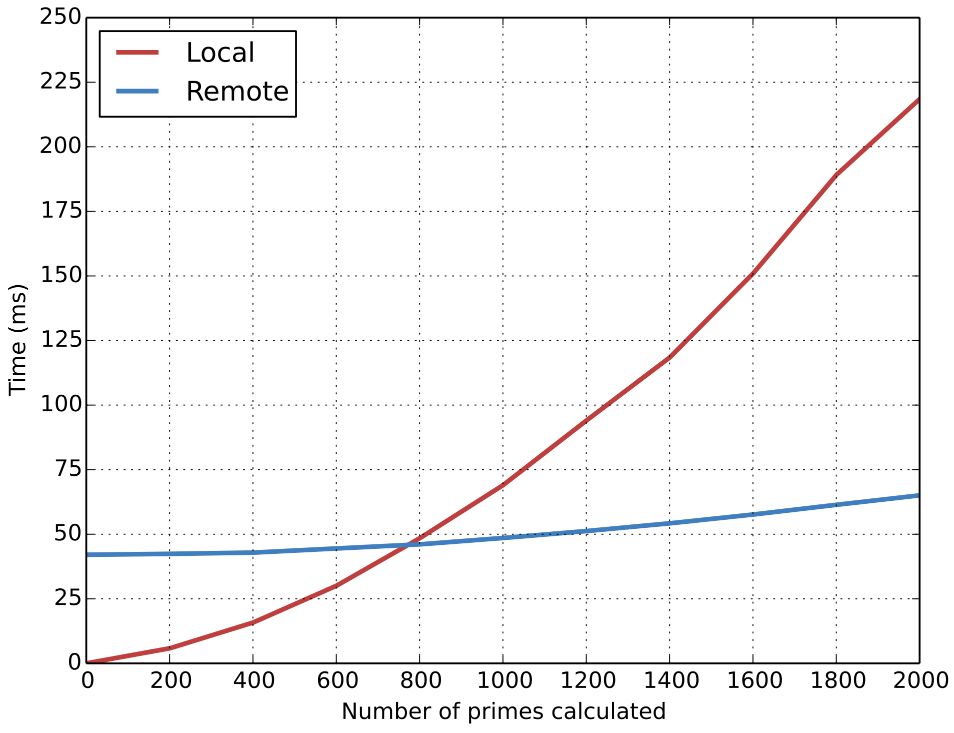
In this section we perform the same two tests as previously but distribute from the iPhone to a remote server.

**Results**

**Round-trip-time**

| **Testbed** | **Time** |
|---|---|
| Phone-Remote Server | 44ms |

**Computational offloading**



### Discussion

The round-trip-time is slower than both the Android phone and the computer on the local network, as expected because the server lives in Germany. The computational offloading graph shows that offloading is viable if the execution time of a task exceeds about 45ms.

### Evaluation

- **Functionality and Stability:** The test program runs as expected.
- **Consistency:** Generally consistent samples, but some variation in the round-trip-time samples.
- **Efficiency:** As expected for RTT and computational offloading viable for execution times exceeding 45ms.

Table 8.3: Spesification of Planetlab nodes

| Host | CPU | RAM | Location |
|---|---|---|---|
| planetlab1.pop-mg.rnp.br | 3.16GHz | 4GB | Brazil |
| planetlab3.cesnet.cz | 2.66GHz | 4GB | Czech Republic |

### 8.3.4 Phone to Planetlab

In this section we do the same two distribution test against two Planetlab nodes. This will show us how the implementation works with distributing to devices over longer distances than the previous tests. The spesifications and locations of the two nodes can be seen in table 8.3.

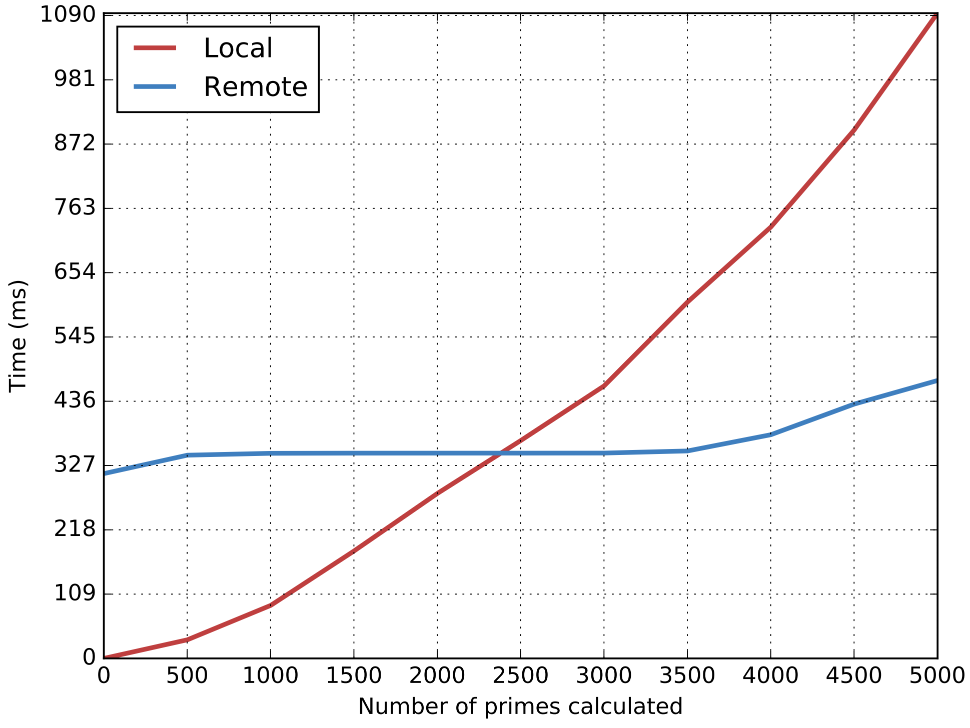
**Results**

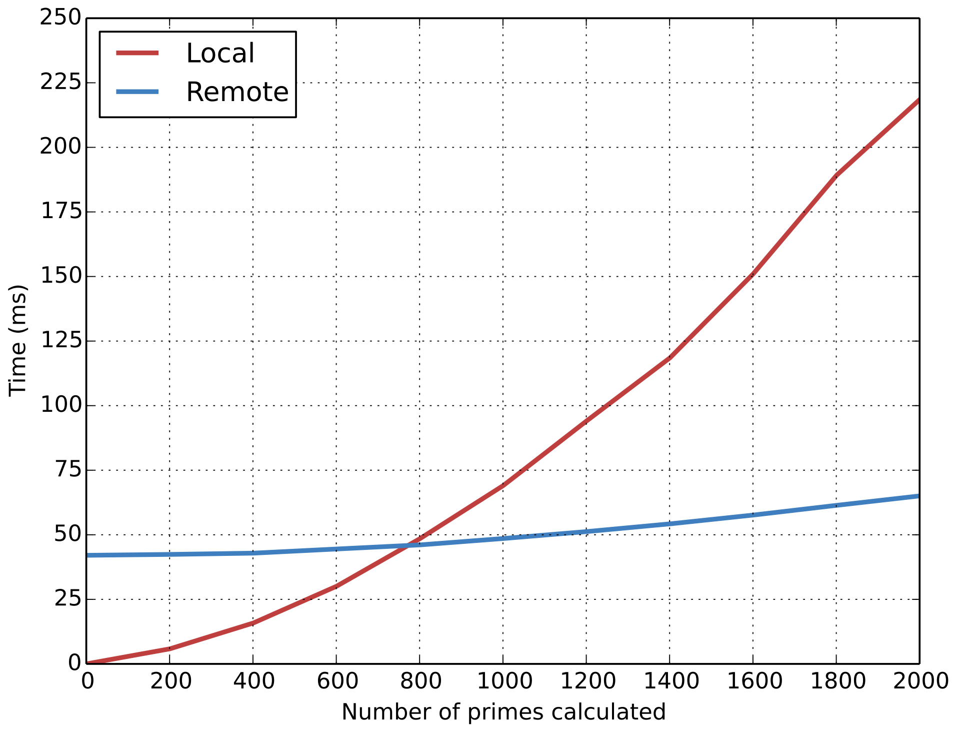
**Round-trip-time**

| Testbed | Time |
|---|---|
| Phone-planetlab1.pop-mg.rnp.br | 328ms |

| Testbed | Time |
|---|---|
| Phone-planetlab3.cesnet.cz | 46ms |

**Computational offloading**


Local
Remote
Time (ms)
Number of primes calculated
0
109
218
327
436
545
654
763
872
981
1090
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000



### Discussion

The server in Brazil have a much higher round-trip-time than the server in the Czech Republic which also shows in the computational offloading graphs where the breakpoints is at 45ms (CZ) and 330ms (BR).

### Evaluation

- **Functionality and Stability:** The test program runs as expected.
- **Consistency:** The CZ server gave consistent samples, while the BR server gave some inconsistent samples.
- **Efficiency:** The CZ server had a lower break point for computational offloading with about 45ms than the BR server with about 330ms.

## 8.4 Evaluation iOS as a development platform

When it comes to developing "standard" applications iOS is a very good platform for developers.

- There is a uniform way of developing applications with Xcode, Objective-C/Swift.
- The same interface builder for all iOS devices.
- Predictable hardware to develop for.
- A safe and somewhat predictable way to distribute the application.

iOS comes with many security features and conventions that when developing standard applications is very welcome, but gives us some problems when porting programs such as Emerald:

- The restrictive license agreement prevents applications that have a to small audience and exists outside of the norm of being distributed.
- The application sandboxing gives more security to the platform but prevents a lot of operations like executing binaries.
- Even if the previous restrictions can be mitigated the only way to distribute applications is through the official App Store.

In conclusion, for normal development iOS and its ecosystem is very good for developers but when it comes to doing something out of the box like porting Emerald the restrictive environment prevents us from doing the things we need with the platform forcing us to jailbreak the device.

## 8.5 Summary

We have shown that integrating Emerald into an iOS device is possible and shown test cases in two categories. **Basic performance** has three test cases, *Maximum supported threads*, *Thread spawn Performance* and *Computational performance*.

The **distribution tests** have two test cases, *Round-trip-time* and *Computational offloading*. We have proved that distribution with objects with the iOS implementation of Emerald is relatively efficient and that in some cases computational offloading can give great benefits when it comes to execution time. To summarize, the Emerald implementation for iOS works, and the performance is on par or slightly worse than the other smart phone we ran tests on.

# Part IV

# Conclusion

# Chapter 9

# Conclusion

In conclusion, we have now successfully ported the Emerald run-time to iOS. We have cross compiled the Emerald source code for the different architectures supported by the different iOS devices, created an iOS native application as a front-end to the runtime and distributed these through Cydia. Because of the many security restrictions on the iOS platform a prerequisite for running the application is a jailbroken device.

## 9.1 Contributions

We have made the following contributions with this thesis:

- Ported Emerald to iOS
- Evaluated the performance for Emerald on iOS

### 9.1.1 Porting Emerald to iOS

Using the compiler distributed with the iOS SDK we cross-compiled Emerald for the ARM architecture.
Because different iOS devices uses different versions of the ARM architecture we had to compile one binary for each of the available architectures. Some small changes to the source code had to be made in order for the Clang compiler to accept, and compile the sources. To execute the binaries we can either use Secure Shell to login to the device or run it from a application created to run the binary.

#### Emerald for iOS native App

We created an native iOS application to be able to run Emerald conveniently without having to use Secure Shell.
This application have a user interface that is more suited for using on a touchscreen than a console and makes compiling and executing Emerald programs easy.

#### Distributed Emerald to Cydia

Because Apple does not allow such an application to be distributed through the App Store we had to distribute the binaries and the application through Cydia, an App Store for jailbroken devices.

### 9.1.2 Performance evaluation

We have evaluated the performance of the Emerald implementation and shown that computational offloading can be very beneficial. Even if the remote node have more computing power we have to consider the latency and the bandwidth between the nodes before offloading computational tasks. In the cases where computational offloading is viable the smart phone will save both time and battery power.

#### Limitations of the implementation

The stack size of the Emerald implementation is smaller om iOS versus other devices, but can be increased with the `-s` command line switch.
Emerald is unstable when distributing to nodes running other Emerald versions. Generally stable as long as we make sure that all the nodes runs the same version (1.06). Because the application is not tied in to iOS in any way we cannot distribute anything defined outside of an Emerald program or use iOS functions from Emerald in any way.

## 9.2 Future work

The Emerald for iOS application is more of a proof-of-concept than a fully fledged iOS application and there is some thing that should be fixed to make it more usable.
There should be an easier way of transferring Emerald programs to the

device when the application is installed through Cydia. If the application is installed through Xcode we can use the file transfer in iTunes, but this option is not available for Cydia applications. The current solution is copying the files through secure shell. The terminal in the application could also be easier to use.

When it comes to Emerald it should be ported to support 64-bit architectures. Further than that the holy grail for Emerald on iOS would be an re-implementation of Emerald as a library that could be statically compiled. This would mean that Emerald could be embedded in any device, and for iOS it means that we could get it through the App Store without breaking the license agreement.

# Appendices

# Appendix A

# Code

The source code of the iOS application is available at:
**`https://github.com/apers/iOSEmerald`**

The end-point for the Cydia repository is:
**`http://pers.link/emerald`**

The rest of the code presented in this chapter is the test programs from the evaluation chapter.

### A.0.1 Maximum supported threads

```
const Thread <- class  Thread[n: Integer]
    var count: Integer
    initially
        (locate self)$stdout.putString["Starting␣thread:␣" || n.asString
                 || "\n"]
    end initially
    process
        loop
            count <- count + 1
        end loop
    end process
end Thread

const Main <- object Main
    var thread_list: Array.of[Thread] <- Array.of[Thread].empty
    var n: Integer
process
    n <- 0
    loop
        thread_list.addUpper[Thread.create[n]]
        n <- n + 1
    end loop
end process
end Main
```

### A.0.2 Thread spawn performance

```
const Thread <- class  Thread[n: Integer]
    var count: Integer
    initially
        %(locate self)$stdout.putString["Starting␣thread:␣" || n.
               asString || "\n"]
    end initially
    process
        loop
            count <- count + 1
        end loop
    end process
end Thread

const Main <- object Main
    var thread_list: Array.of[Thread] <- Array.of[Thread].empty
    var n: Integer
    var startTime: Time
    var finishTime: Time
process
    n <- 0
    startTime <- (locate self)$timeOfDay
    for j : Integer <- 0 while j < 1000 by j <- j + 1
        thread_list.addUpper[Thread.create[n]]
    end for
    finishTime <- (locate self)$timeOfDay
    stdout.putString["Time:␣" || (finishTime-startTime).asString || "\n"
           ]
end process
end Main
```

### A.0.3 Calculation test

```
const driver <- object driver
  export op findPrimes
```

```
    attached var finishTime: Time
    attached var startTime: Time <- (locate self)$timeOfDay
    attached const limit <- 50000
    attached const primes <- Vector.of[Integer].create[limit]
    attached var howmany : Integer <- 0
    attached var j : Integer
    for i : Integer <- 2 while i < limit by i <- i + 1
      j <- 0

      loop
            exit when j >= howmany or i # primes[j] = 0
            j <- j + 1
      end loop

      if j >= howmany then
            primes[howmany] <- i
            howmany <- howmany + 1

        if howmany#1000 = 0 then
            (locate self)$stdout.putString[i.asString || "\n"]
        end if
      end if
    end for
    finishTime <- (locate self)$timeOfDay - startTime
    stdout.putString["Execution␣time:␣" || finishTime.asString || "\n"]
  end findPrimes
end driver

const Main <- object Main
initially
    const up <- (locate self)$activeNodes

    % Only this node available
    if up.upperbound = 0 then
        driver.findPrimes
    else
        move driver to up[1]$theNode
        driver.findPrimes
    end if
end initially
end Main
```

### A.0.4 Round-trip-time

```
const Kilroy <- object Kilroy
```

```
  process
    const home <- locate self
    var there :     Node
    var startTime, diff : Time
    var total: Time
    var all : NodeList
    var theElem :NodeListElement
    var stuff : Real

    home$stdout.PutString["Starting␣on␣" || home$name || "\n"]
    all <- home.getActiveNodes
    home$stdout.PutString[(all.upperbound + 1).asString || "␣nodes␣
           active.\n"]

    total <- Time.create[0,0]

    for j : Integer <- 0 while j < 20 by j <- j + 1
        startTime <- home.getTimeOfDay

        for i : Integer <- 1 while i <= all.upperbound by i <- i + 1
          there <- all[i]$theNode
          move Kilroy to there
          there$stdout.PutString["Kilroy␣was␣here\n"]
        end for

        move Kilroy to home
        diff <- home.getTimeOfDay - startTime
        total <- total + diff;

        home$stdout.PutString["Back␣home.␣␣Total␣time␣=␣" || diff.
               asString || "\n"]
    end for
    home$stdout.PutString["Finished.␣␣Avg.=␣" || (total/20).asString ||
           "\n"]
  end process
end Kilroy
```

### A.0.5 Computational offloading

```
const driver <- object driver
  export op findPrimes[limit: Integer]
    attached const primes <- Vector.of[Integer].create[limit]
    attached var howmany : Integer <- 0
    attached var j : Integer
    for i : Integer <- 2 while i < limit by i <- i + 1
```

```
      j <- 0

      loop
            exit when j >= howmany or i # primes[j] = 0
            j <- j + 1
      end loop

      if j >= howmany then
            primes[howmany] <- i
            howmany <- howmany + 1
      end if
    end for
  end findPrimes
end driver

const Main <- object Main
process
    const here <- (locate self)
    var finishTime: Time
    var startTime: Time
    const up <- here$activeNodes
    const file <- OutStream.toUnix["primes-data.txt", "w"]

    file.putString["Local:\n"]

    for i : Integer <- 1 while i <= 10001 by i <- i + 200
        startTime <- here$timeOfDay
        driver.findPrimes[i]
        finishTime <- here$timeOfDay
        file.putString[i.asString || ";" || (finishTime-startTime).
               asString || "\n"]
        here$stdout.putString[i.asString || ";" || (finishTime-startTime
               ).asString || "\n"]
    end for

    fix driver at up[1]$theNode
    file.putString["Remote:\n"]

    for i : Integer <- 1 while i <= 10001 by i <- i + 200
        startTime <- here$timeOfDay
        driver.findPrimes[i]
        finishTime <- here$timeOfDay
        file.putString[i.asString || ";" || (finishTime-startTime).
               asString || "\n"]
        here$stdout.putString[i.asString || ";" || (finishTime-startTime
               ).asString || "\n"]
    end for
    here$stdout.putString["Done\n"]
    file.flush
    file.close

end process
end Main
```

# Bibliography

[1] International Data Corporation. Smartphone os market share, q4 2014. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Online; accessed: 05-May-2015.

[2] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[3] S. Ghosh. *Distributed Systems: An Algorithmic Approach, Second Edition*. Chapman & Hall/CRC Computer and Information Science Series. Taylor & Francis, 2014.

[4] Jan Rune Holmevik. Compiling simula: A historical study of technological genesis. *IEEE Ann. Hist. Comput.*, 16(4):25–37, December 1994.

[5] N.C. Hutchinson. *Emerald: an object-based language for distributed programming*. PhD thesis, Univ. of Washington,Seattle, WA, Jan 1987.

[6] Apple Inc. Apple developer program license agreement. `https://adcdownload.apple.com/Documentation/License_Agreements__Apple_Developer_Program/Apple_Developer_Program_Agreement_20150909.pdf`. Online; accessed: 06-Jan-2016.

[7] Apple Inc. The foundation framework. `https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/ObjC_classic/`. Online; accessed: 10-Nov-2015.

[8] Apple Inc. Uikit framework reference. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/. Online; accessed: 11-Nov-2015.

[9] Computer World Jonny Evans. Apple has sold one billion ios devices, company claims. http://www.computerworld.com/article/2876320/apple-has-sold-one-billion-ios-devices-company-claims.html. Online; accessed: 19-May-2015.

[10] E. B. Jul. *Object Mobility in a Distributed Object-oriented System*. PhD thesis, Univ. of Washington,Seattle, WA, Seattle, WA, USA, 1989. UMI Order No: GAX90-00257.

[11] Medienorge. Andel som har smarttelefon. http://medienorge.uib.no/statistikk/medium/ikt/379. Online; accessed: 04-Nov-2015.

[12] J.C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[13] Pangu. Pangu jailbreak tool. http://en.7.pangu.io/. Online; accessed: 12-Jan-2016.

[14] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[15] Ira Sager. Before iphone and android came simon, the first smartphone. www.bloomberg.com/bw/articles/2012-06-29/before-iphone-and-android-came-simon-the-first-smartphone. Online; accessed: 04-Nov-2015.

[16] Kevin C. Tofel. Android sales overtake iphone in the u.s. https://gigaom.com/2010/08/02/android-sales-overtake-iphone-in-the-u-s/. Online; accessed: 04-Nov-2015.

[17] The Verge. ios: A visual history. http://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad. Online; accessed: 19-May-2015.

[18] Owen Williams. Apple announces swift, a new programming language for ios and os x. `http://thenextweb.com/apple/2014/06/02/apple-announces-swift-new-programming-language-ios/`. Online; accessed: 05-Nov-2015.