

Mobile Devices In the Distributed IoT Platform

Calvin

ALEXANDER NAJAFI

MASTER'S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY



Mobile Devices In the Distributed IoT Platform

Calvin

Alexander Najafi
elt12ana@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Maria Kihl (LTH) and Ola Angelsmark (Ericsson)

Examiner: Christian Nyberg

June 1, 2017

© 2017
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Development towards a connected world where anything can talk to anything is ongoing. Calvin is a research project at Ericsson that simplifies the development of peer-to-peer applications for the Internet of Things. Calvin is created in order to let the developer of an application for the Internet of Things focus on the implementation instead of having to worry about underlying protocols, hardware access, security, application deployment, and application management. A Calvin application is defined by a dataflow graph where every node is a small reusable computational component called an Actor. The actors can move and be migrated to any connected unit during runtime which allows for complex applications to be created.

Mobile devices are computationally powerful devices with many sensors and have a lot of functionality through mobile device applications. The devices have traditionally only been used for configuration and control of Internet connected devices. This thesis successfully shows how mobile devices can be part of the distributed platform Calvin as any other device in the Internet of Things.

The thesis proposes a general solution of how Calvin can make advantage of a mobile device at any time it is connected to the Internet. The thesis shows how mobile devices and third party applications on the device can share their capabilities in the Calvin platform with other connected devices. The thesis analyses the effect of the long term running Calvin runtime on a mobile device and shows that it is possible and feasible to use Calvin in Android without draining the device's resources.

Keywords: Calvin, Mobile Devices, Android, Internet of Things, Distributed Platforms, Dataflow.

Popular Science Summary

The Internet might be the greatest innovation in modern time. It is used by billions of people every day to do all sorts of tasks. The number of Internet connected device is growing rapidly and Ericsson is expecting 26 billion devices to be connected in 2020 [1]. To meet these demands, development of new network types is ongoing. Ericsson is currently developing the next generation networks for mobile phones called 5G and more people get their houses connected to the Internet through fast fiber optic cables.

Project Motivation

The *Internet of Things (IoT)* is a term referring to historically non Internet connected devices that are connected today. IoT therefore includes a broad spectrum of device types such as temperature sensors, thermostats, light bulbs, cars, speaker systems and even refrigerators. It is a complex task to enable all of these devices to connect and communicate. The devices must be able to recognize each other and they must have a common method of how they should share data. For example, a temperature sensor must have a method of how temperature data should be sent to other devices such as a mobile phone. The mobile phone must furthermore be able to parse the data from the temperature sensor to be able to display the data to a user. The devices must also be able to decide if the communication should be done over for example Wifi, 3G, 4G or even Bluetooth.

At Ericsson research in Lund, a group of scientists are trying to solve the problem of connecting these devices through a project called *Calvin*. The project enables any device capable of running Calvin to communicate with other Calvin devices. Calvin also allows users to easily specify how the devices should be connected. It is a simple task to connect a light bulb with a Internet connected button to allow the lights to be controlled in Calvin.

This master's thesis focuses on bringing mobile devices into the IoT platform Calvin as any other Internet connected device. Mobile devices has historically only been used to control and read data from other devices. Mobile devices are for example common as remote controls in home automation systems where they can be used to control the house lighting or to fetch the current outdoor temperature. By letting mobile devices become a part of Calvin, more complex and powerful applications can be created.

Mobile Devices in Calvin

Mobile devices are in some aspects very different from other Internet connected devices. While they are very powerful in terms of processing power, they are very limited in some aspects such as battery and data usage. A mobile phone user normally expects the device to be able to run for at least a full day of normal usage while the data consumption often is limited by the mobile network carrier. All programs running on a mobile device must therefore be very careful in their use of these resources and must be adapted to fit in to the mobile device system.

The available mobile operating systems in the world is dominated by Android which has a market share of 86.6% [2]. This thesis therefore focuses on bringing the Android system into the Calvin platform. This is done by proposing adoptions and needed functionality in order to let Calvin take advantage of a mobile device and to let the mobile device share its capabilities with other Calvin connected devices. The main problem formulation for the thesis therefore comes naturally to be “*how can the distributed platform Calvin make the best use of a mobile device?*”

The thesis takes on and proposes a solution of many different aspects of mobile devices. One aspect is how to solve the problems with the mobility of mobile devices. Since mobile devices can move, they connect to many different networks all the time. For example, in a home automation system, the mobile is only connected to the home LAN while it has Wifi coverage. If the mobile device leaves the home it loses its connection to other devices since it connects to a mobile network instead. This thesis proposes a solution where the the mobile device can be anywhere in the world while still being able to communicate with Calvin connected devices.

Mobile devices has a large amount of sensors and other hardware capabilities. But a lot of the functionality of a mobile device is implemented in various applications that are downloaded. This thesis shows how both the hardware capabilities, but also how third party applications can share their functionality with other Calvin connected devices in a standardized and simple way.

Evaluation

This thesis shows how Calvin must be adapted in order to meet the high demands on mobile device applications in terms of resource usage. The RAM memory is a limited resource that has to be used with great care in a mobile device. The RAM memory usage of Calvin was examined through a set of experiments in and the results are used to show that Calvin behaves good on the mobile device. The RAM usage is dominated by Android itself and the Android Framework when running the application but the actual Calvin implementation.

The data usage of the proposed solution for connectivity was also examined. The solution enables flexibility in terms of mobility, but it comes with a price since much more data has to be sent. The solution was evaluated and it is shown how much more data the proposed solution consumes.

Overall the thesis shows that it is feasible to run Calvin on a mobile device, and with modifications and added functionally great applications can be created in an easy way.

Acknowledgements

I would like to express my gratitude to everyone involved in making this master's thesis go from an idea to a successful result.

I would like to thank the members of the Cloud EE and DC Operations team at Ericsson Research in Lund for supporting me with their expertise and knowledge within the area of research. I would especially like to thank Fredrik Svensson and my main coordinator Ola Angelsmark for always helping me and giving me comments on my work along the road. It has been a pleasure working with you and contributing to the Calvin project.

I would also like to thank my supervisor Maria Kihl from the department of Electrical and Information Technology at Lund University for her guidance during the work on this thesis.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Project Aims and Main Challenges	2
1.3	Previous Work	2
1.4	Resources	3
1.5	Overview of the Thesis	3
2	Background	5
2.1	Distributed Systems	5
2.2	Dataflow Oriented Programming	6
2.3	Actor Model	7
2.4	XMPP	10
2.5	Data Encoding	11
3	Mobile Operating Systems	13
3.1	Android Application Format	13
3.2	Android SDK and NDK	14
3.3	Services in Android	14
3.4	Android Messaging System	15
3.5	Security	16
3.6	Capabilities in Android	17
4	Calvin - The distributed IoT platform	19
4.1	Calvin Lifecycle	19
4.2	Actors	19
4.3	Applications	20
4.4	Runtime	21
4.5	Communication	22
4.6	Calvin Constrained	22
5	Mobile devices in Calvin	25
5.1	Architecture	25
5.2	Connectivity	26
5.3	Discovery	31

5.4	Choosing Transport Mechanism	31
5.5	Calvin and the Mobile Platform	32
5.6	Power Management	34
5.7	Serialization	34
5.8	Configuration and Key Handling	35
5.9	Capabilities	36
5.10	Hardware Abstraction Layer	36
5.11	Third Party Applications	40
5.12	Proof of Concept	41
6	Evaluation and Result _____	43
6.1	Evaluation Method	43
6.2	Setup	45
6.3	Result	46
6.4	Evaluation	51
7	Conclusions _____	55
7.1	Mobile Devices in Calvin	55
7.2	Future Work	55
	References _____	57
A	Example of the Calvin Constrained PROXY_CONFIG command _____	61
B	Example of an XMPP authentication handshake _____	63
C	Calvin Configuration for FCM _____	65

Abbreviations

API Application Programming Interface

APNS Apple Push Notification Service

ART Android Runtime

BT Bluetooth

CC Calvin Constrained

CPU Central Processing Unit

DFP Dataflow Programming

DHT Distributed Hash Table

DNS Domain Name System

FCM Firebase Cloud Messaging

FD File Descriptor

FIFO First In First Out

HAL Hardware Abstraction Layer

I/O Input/Output

IoT Internet of Things

JNI Java Native Interface

JSON JavaScript Object Notation

JVM Java Virtual Machine

LAN Local Area Network

LRU Last Recently Used

LTE Long-Term Evolution. Also known as 4G LTE

OS Operating System

P2P Peer to Peer

RAM Random Access Memory

REST Representational state transfer

SASL Simple Authentication and Security Layer

SDK Software Development Kit

SSDP Simple Service Discovery Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

WLAN Wireless Local Area Network

XMPP Extensible Messaging and Presence Protocol

1.1 Motivation

Devices connected to the Internet surround our every day life. Devices such as mobile phones, tablets, TV:s, cars, and even dishwashers can today connect to the Internet and cloud infrastructures. Ericsson is expecting 26 billion [1] devices to be connected in 2020 and that there is no stopping beyond that.

To meet the demands of the growing data flow on the Internet, Ericsson is currently developing the next generation mobile network 5G. Development is focused on enabling connectivity with higher bandwidth and lower latency while still trying to communicate using less energy in the signals. There is focus from the Internet giants, such as Facebook, to bring Internet connectivity to extremely remote locations using state of the art technology [3]. More and more devices around the globe gets connected, and there is more and more pressure from the market on the companies developing the techniques to enable this.

At Ericsson Research in Lund a research group is currently developing an IoT platform called Calvin to take on the challenge of connecting devices. Calvin [4] is a platform for the Internet of things (IoT) that lets any device communicate with other devices. Calvin standardizes a way for the connected devices to communicate and make use of each others capabilities in a scalable and flexible way. Calvin is also used as an application platform to make it easy to develop, deploy and maintain IoT applications.

A device that almost everyone carries around and uses frequently in their every day life is the mobile phone. Modern smart phones are capable and powerful small personal computers with access to a lot of sensors and capabilities. Furthermore they are almost always connected to the Internet trough a wireless connectivity method such as WIFI and LTE. The mobile phone has traditionally been used for configuring, controlling and receiving data from other devices, however this must not be the case. This MSc thesis focuses on letting mobile devices be part of distributed networks as any other device. These networks can then be used for example in distributed computing, home automation systems or car to car communication systems.

1.2 Project Aims and Main Challenges

This MSc thesis will focus on how to bring mobile devices into the distributed platform Calvin. The mobile device must be able to communicate and share its capabilities with other devices and the platform must be able to distribute the tasks of an application to the best fitted device. While the mobile device is a great device with great possibilities, it is very limited in some matters. This thesis will take on the problems concerned with the limited resources of a mobile phone and how to make the best use out of it. For mobile phones, aspects regarding power consumption, data consumptions, CPU usage, and RAM usage could be potential pitfalls for long term running applications. There are problems regarding mobile phones connectivity methods and how they can change rapidly during the runtime of an application that must be handled. To summarize, this MSc thesis will focus on solving the problems concerning:

- Bringing mobile devices into the distributed system Calvin.
- The effect on a mobile device's resources from long running applications.
- The effect on an application when ported to fit into a mobile operating system's system.
- How third party applications on the mobile device can take advantage of Calvin.
- How applications handle connectivity and can assure availability.

which in turn forms the main problem formulation *How can the distributed platform Calvin make the best use of a mobile device?*.

1.3 Previous Work

Calvin is an IoT platform created for research purposes, there exists two reference implementations written in the Python and C languages that will act as a foundation for this masters thesis. The main coordinator at Ericsson, Ola Angelsmark has been working with Calvin for a long time and has been one of the main contributors to the project. He, together with the rest of the Calvin team, holds great knowledge about the platform and the area of research. The members of the Calvin team writes blog posts on the Ericsson Research blog [5] and they have released a number of papers [1] [6] [7] that have been presented at conferences around the world.

There have been several MSc theses done within the area of Calvin. Tomas Nilsson described how Calvin can handle actors and how they can be migrated depending on security rules [8] in his thesis in 2016. Other MSc thesis works done within the area of dataflow programming, distributed systems and Calvin include [9], [10], [11], and [12].

1.4 Resources

The work of this MSc thesis was carried out in cooperation with Ericsson Research in the Cloud Execution Environment and DC Operations team. Ericsson provided Raspberry Pi computers, LEGO, network equipment and servos for my work.

The software development was done using the VIM text editor for C and Python development. For Android development, the Android SDK and NDK was used together with the Android Studio IDE. The Google GCC compiler and LD software for Android which is bundled with the Android SDK was used for the X86 and Arm 64 architectures to build C code.

1.5 Overview of the Thesis

This thesis report consists of eight chapters disposed in the following way:

Chapter 2 gives an understanding about the ideas and techniques used for this MSc thesis. The chapter explains the concepts of dataflow programming and distributed systems. An overview of the XMPP protocol used in chapter 5 is given together with a section about data encoding.

Chapter 3 gives theory about mobile operating systems with a focus on Android. This chapter gives an introduction to how messaging systems, connectivity and capabilities are handled in Android.

Chapter 4 gives an introduction to Calvin and how the platform operates using techniques and concepts described in chapter 2. The chapter also describes the role of the Calvin implementation for constrained devices.

Chapter 5 proposes a solution to the given problem description, how it was designed, and implemented. The chapter explains how the evaluation of the proposed solution was done and how the results were obtained.

Chapter 6 shows and discusses the result from the evaluation of the proposed solution.

Chapter 7 concludes the thesis and proposes areas of future work.

Appendix Shows examples discussed in the thesis.

This chapter explores areas of interest for this master thesis. It gives theory about how distributed systems work and explains the actor model which is the foundation of how Calvin is created. It finishes up by giving the background of the XMPP protocol and two data encoding techniques which are used in chapter 5.

2.1 Distributed Systems

Almost all programs and applications on our mobile devices uses some kind of information sharing and communicates with other entities. To make a phone call, one makes use of the cellular network to share our voice. When browsing the web, one downloads content from a remote server using the Internet. And when navigating through a city one uses the global positioning system. The term *distributed computing* refers to systems where the application is divided into components that are executed on different physical or virtual units. The units in a distributed system uses a transportation mechanism, such as LTE or WIFI, to pass messages between the application components which they act on. Distributed systems are crucial in our every day life and includes a large number of platforms and applications.

One group of distributed systems are systems where every component in the system communicates directly with the other components without having any centralized unit. These *peer to peer* (P2P) [13] systems aim to spread information and capabilities on a network among all connected entities. As opposed to classical server-client systems, the availability of a unit in a P2P system cannot be guaranteed. The system's availability and capabilities can change during runtime which both puts requirements on the platform but also allows for flexible and scalable applications.

In an ideal platform, the application's components should always be executed on the units where it is most feasible for the application. Distributed P2P systems play a big role in IoT. Normally non-connected devices get connected to each other and allow them to take advantage of each others capabilities. This has led to a great expansion in the areas of home automation, smart cities and Internet connected cars to name a few. For example, in a P2P system of connected cars, the cars could potentially communicate directly to each other to send messages about their geographical location. This could be used in software that can avoid collisions. However, some of the earliest distributed systems where not meant for

vehicle communications, but for file sharing. One of the first P2P systems for file sharing was proposed by the creators of *Napster* [13] which developed to be one of the most used file sharing software on the planet.

A common issue with general distributed systems is to implement a common area that all units in the network can reach to share data. Distributed systems can make use of a distributed map of key/value pairs that is accessible from all nodes in the network by implementing and using a *distributed hash table* (DHT) [13].

2.1.1 Distributed Hash Table

A DHT is a distributed storage entity used in distributed networks as a registry where data can be stored. There are multiple ways of implementing a DHT where one of the most common is called Kademlia [14]. From the DHT user's point of view, the DHT works like a normal hash table that can store key/value pairs. However the nodes of the hash table are stored across different entities in the distributed application, and finding a value for a given key is done in an efficient way.

Every node in the distributed platform needs to have a unique ID for identification. Kademlia uses a distance function to calculate the relative distance between the nodes' IDs. The distance between two nodes is defined as the exclusive or between the node's IDs. When a value of a certain key is to be found, Kademlia uses an algorithm to iteratively find nodes that are closer to the node that holds the searched value. The algorithm has a time complexity of $\mathcal{O}(\log n)$ [14] during the search of a value, which makes it a popular choice when implementing a DHT.

2.2 Dataflow Oriented Programming

Typical computer programs are made up of lines that are executed sequentially in the order they are written. This classic and linear model was first described by John Von Neumann [15] who divided the computer system into three parts, the input/output (I/O) units, the processing unit, and the memory unit. The input data in such a program typically comes from the input unit and is stored in the memory from which the CPU can read and perform computations.

The dataflow programming (DFP) model focuses on the data in a program and how it is sent between different units, called nodes. An application written using the DFP model can be represented as a directed graph with nodes rather than a set of code lines. Each node is a simple unit that can perform a task based on the input data and then forward its result to the next node in the graph. Since it is easy to represent a program written using the DFP model graphically, it is a natural choice of model when doing visual programming. It is also a good model that allows for efficient concurrent execution of programs without using traditional threads [16]. Since every node in the graph is independent they can all process data at the same time without interrupting each other, one can think of the graph as a multi core processor. By letting the input and output ports of the nodes being FIFO queues, nodes do not have to wait for other connected nodes before performing a task. Every node simply performs its task independently based on

the data in the input queue and then forwards the result to the connected node in the graph.

The nodes are divided into **sources**, **sinks** and **computational units**. A computational unit is a unit that has input and output ports and that executes a task based on the input data. Sources are special nodes that do not have any input ports, but only output ports. They are used to initiate the program and feed data into the graph. Examples of source nodes are keyboards or simple buttons. A sink is the opposite, it is the terminal node for data flowing in the graph and only has input ports. A typically sink receives a result and then presents it to the entity that executed the program, such as a monitor displaying a result to the user.

The model of dataflow programming allows for very complex graphs which does not necessarily have to be a straight line of nodes. A node's output port could be connected to multiple node's input ports, a property called *fan out*. Similarly, a node's input port can receive data from multiple node's output ports. This is called *fan in*.

An example of a dataflow oriented system that uses the fan in property is a printer system as shown in figure 2.1. In this simplified printer system, the printer is a sink node and the clients are sources. The graph specifies that the printer is connected to a printer server (usually built into network printers). There can be an arbitrary number of devices connected to the printer server which listens for data on a fan in port. The printer server is a computational node that has an interface for communicating over a network and knows how to schedule and queue printer jobs. As soon as any device sends data to the printer server it forwards it to the printer which starts printing. If a device sends data to a busy printer, the job is queued until the printer is free.

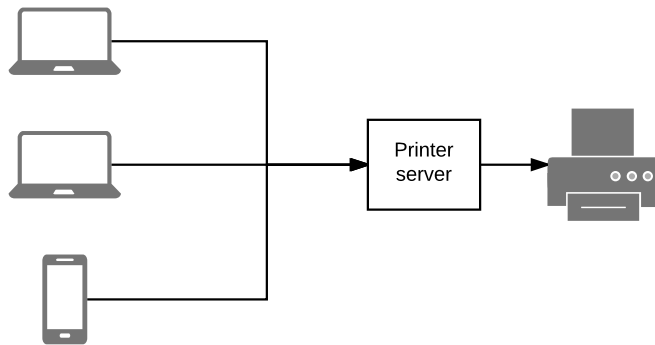


Figure 2.1: Simple dataflow graph of a system with a printer, a network connected printer server and clients that can print documents.

2.3 Actor Model

Every node in a dataflow program graph can be thought of as an entity called an actor. An actor is simple in its design and implements a small and reusable

actions such as a monitor displaying media content or an accelerometer reading the acceleration of a mobile phone. By combining actors and specifying the application graph of how they are connected, an application can be created. The implementation of an actor is platform specific, on one platform the actor might be implemented using the C language while it may be implemented in Python on another platform.

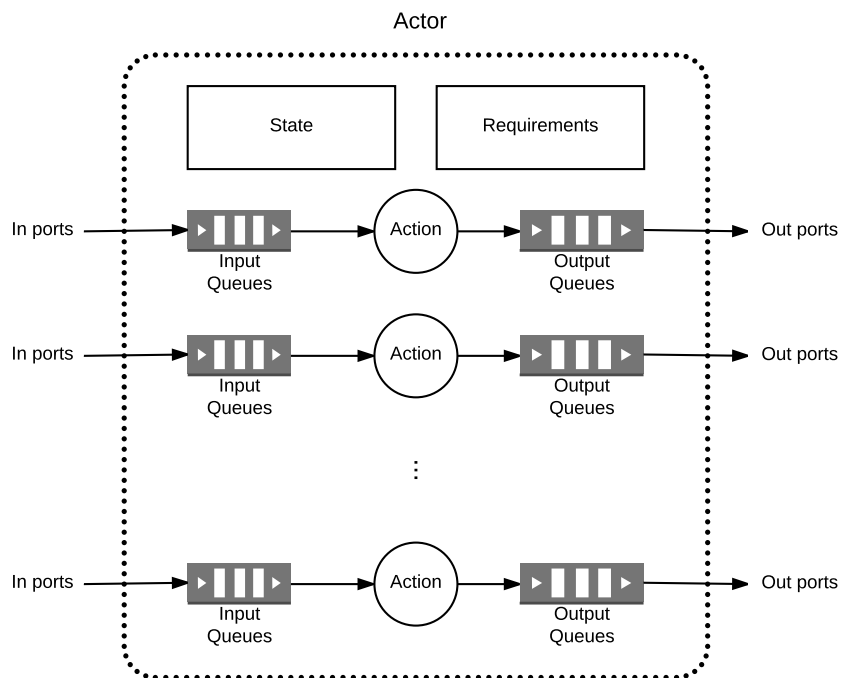


Figure 2.2: The architecture for an actor with multiple actions.

Every action of an actor is bound to a set of in ports and out ports as can be seen in figure 2.2. An important aspect of an actor is how it behaves in the application graph. An actor's action is only triggered either by an external event or when there is data on the actor's input port. When an action has finished its task with resulting output data, it sends its result to the output port which is forwarded in the application graph. An actor may however store data which is consistent between action triggers. The actor may for example store a counter that counts the number of action triggers or configuration parameters. All data that an actor stores form the *actor's state*. If an actor is to be migrated as described in section 2.3.4 to another runtime, the internal actor's state must also follow the actor to not lose any information in the migration process. Therefore all data that the actor stores in its state must be serializable to a JSON format.

In figure 2.1, three different actor types are used on four nodes. On the mobile phones and laptop an actor that produces a document is running, assume this is a word processor. The word processor is however implemented in two different ways

on the phone and the laptop. The UI, architecture and even the language may be different. The common factor is the out port, and the actors role in the graph. On the printer server platform, a printer server actor is running. And on the printer, an actor that can perform the actual print job is executing.

When an application is deployed using the application graph the dataflow graph is created. The dataflow graph is not static in its design and is subject to change during runtime. The application graph may allow for dynamic creation and deletion of actors during application runtime. This can be used for performance up and downscaling if a program needs to perform many heavy computations. If the physical computing power needed is more than available more actors that can handle the computations can be spawned and connected in the graph. By making use of the fan in and fan out properties of the dataflow graph, this can be done in a seamless and efficient way. The scalability can also be used for program graphs where the number of actors are unknown at the initiation of an application. For example, in the printer system in figure 2.1 the number of clients is unknown when the printer is first started and clients may come and go without affecting the systems availability or having to restart the application.

2.3.1 Implementing Actors

There are many benefits from building distributed applications using the dataflow and actor models. The actor model allows the developer of the platform to add a layer of abstraction between the application and platform development. An actor developer does not have to care about all aspects of the underlying platform, these aspects are discussed in the following sections.

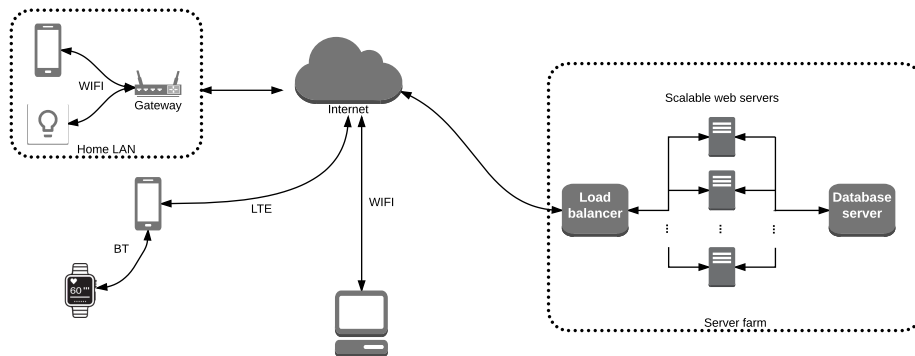


Figure 2.3: Example of a distributed platform for a cloud connected home automation system.

2.3.2 Executing Unit

The application and actor developer does not know on what unit the codes will end up being executed. Nor will he need to know the physical location of the unit or how it should be deployed there. The developer only has to implement the needed

actors and specify the application graph for a program to work. Deployment and execution is handled by the underlying platform. For example, the developer of an email system should not have to care if emails are sent from a personal computer, mobile phone or from a server.

2.3.3 Transport Mechanism

It is up to the distributed platform to handle and implement a way for the units of the network to communicate. The transport mechanism should be completely transparent for the application developer and should only provide a simple application programming interface (API). The transport mechanism can vary much within the network. A laptop often has a WIFI connection, but a phone might use LTE and a smart watch might use Bluetooth. In all cases, the devices must be able to communicate with each other on the same level and be able to understand the same actors.

2.3.4 Replication and Migration

If specified by the application graph, actors might replicate for different purposes. A common reason for replication is for upscaling of the performance of an application. A web application running in a server farm might replicate its web or database servers to handle more traffic at some times during the day. The developer of an application only has to specify how the graph should handle replication. Then the distributed platform handles the deployment, setup and rewriting of the application graph.

During runtime, an actor might have to change physical unit it is executing on. The reasons could be if a resource becomes unavailable, something fails, or there is a requirement from the application developer that triggers it. The procedure for moving an actor is called *actor migration*. The migration and all the involved steps are handled by the distributed platform and the developer of an application does not have to care about and does not even know when a migration occurs.

2.4 XMPP

The Extensible Messaging and Presence Protocol is an open and free lightweight XML streaming protocol used for instant messaging, multi-party voice and video calls, and for IoT communication [17]. It was primarily developed for the use in the Jabber instant messaging client, but it has grown to be used in many different types of applications that need streaming of XML. The protocol is for example used by Google in their Firebase service which is used in this thesis and explained in section 5.2.

XMPP allows for bidirectional messages being sent and has an asynchronous way of sending data packets. XMPP offers an open, flexible, and secure way of communicating and can provide a strong authentication mechanism through the Simple Authentication and Security Layer (SASL) [17].

The authentication process consists of a handshake between the server and client. The client that would like to connect to the server opens a connection

using the *user datagram protocol* (UDP), usually encrypted with a *transport layer security* (TLS), to the servers specified port. An example of the messages sent in the handshake is shown in appendix B. The handshake begins with the client sending a request for authentication, the server answers with information about the method for authentication and continues to authenticate the client.

2.5 Data Encoding

2.5.1 Messagepack

JavaScript Object Notation (JSON) is a lightweight and easy to understand data-interchange format [18]. JSON can format lists and collections of key/value pairs for integers, strings and boolean values. It is a very popular choice for formatting data to be transmitted and there exist many libraries for multiple languages for parsing the format.

Messagepack is a binary format built for object serialization like JSON [19]. MessagePack is opposed to the JSON format, a binary format that compared to JSON is compressed in its syntax. Syntactics in JSON such as brackets, colons and hyphens do not exist. These and combination of these syntactic symbols are binary coded. MessagePack also compresses the data by removing unnecessary padding around data values, short integers can for example be coded as a single byte. The JSON payload `{"compact": true, "schema": 0}` is a 27 bytes long string. In MessagePack this is only coded using 18 bytes [19]. The full specification of MessagePack is open [20] and free which makes it a popular format and allows for many parsers in different languages.

2.5.2 Base64 Encoding

Base64 encodes binary data to human readable characters that can be transmitted and bundled in non binary formats such as JSON for transmission. Base64 encodes the binary data by transforming it into a number system with the base 64 instead of the binary 2.

To encode binary data in Base64, the data is first formed into groups of 3 bytes (24 bits). These groups are then divided into four 6 bit groups which each represents a number in the base 64 since $2^6 = 64$. To decode Base64 encoded data the reverse is done. Base64 is standardized in RFC3548 [21] and uses the characters A-Z, a-z, 0-9 together with the two characters + and / to represent the values 0-63 [21]. A 65th character = is used for padding.

Mobile Operating Systems

There are many mobile operating systems on the market, Android, iOS, Windows Phone 7, Symbian, Sailfish OS, and Ubuntu Touch to name a few of them. The market is however dominated by Android which in 2016 was running on 86.8% [2] of all smartphones worldwide. In second place comes Apples iOS with 12.5% [2] of the market followed by Windows Phone 7 at 0.3% [2]. When developing applications and services for mobile devices it is crucial to implement systems in a way that large parts are platform independent for the biggest platforms. Both iOS and Android supports native development in C, while Android requires a layer of Java on top of it. By using the Java Native Interface (JNI), Java code can execute native code written in C. C, Objective C and Swift are the supported language by the iOS software development kit (SDK). Developing services and applications in C can therefore be favorable, especially when the service is a long running background service that does not need access many API methods on the platforms.

Since the Android operating system is dominating the market, this chapter will focus on giving the reader an introduction to how the platform works and how necessary concepts used in this thesis work.

3.1 Android Application Format

Every Android application that is installed on a mobile device is packaged as a zip file with the file ending `.apk`. The zip file contains compiled Java byte code and static binary files that are needed by the application to run. It also contains compiled native shared library files and compiled Java libraries that are used by applications.

In the root of the APK archive there must be a meta data file called the *Android manifest* file. The file specifies the name, version, app icon, requirements on the hardware, required permissions, all application components, and other application specific attributes. The file is used to tell the operating system how the application should be installed and executed. An application may for example specify UI components called *activities* and background services called *services* in the manifest file.

Every Android application that is to be installed on a commercial Android device must be signed with a public-key certificate. Any unsigned application will

be denied installation on the device if it is not signed correctly. The certificate is normally self generated by the application developer before uploading the application to Google's market place Google Play. During the development process of an application developers may sign the application using a developer certificate that comes with the SDK. This allows for easier development and installment of Android applications during development.

3.2 Android SDK and NDK

The Android SDK [22] consists of a large amount of tools to make it easier to develop applications for the Android operating system. The tools include cross-compilers, debuggers, editors, API documentation and a suit of tools for profiling applications.

The Android Native Development Kit (NDK) [23] allows developers to use native code languages such as C or C++ to develop applications and components of applications for Android [23]. The NDK consists of a limited set of APIs for accessing the platform's components such as sensors, the screen and event tracking mechanisms. The NDK provides a port of the compilers GCC and Clang for Android and the ARM, x86, and MIPS architectures. The NDK also provides tools to debug and profile binaries on the Android platform.

3.3 Services in Android

Android allows for application components called services to be created and started by applications. Services are long lived application components [24] that run in the background of an application and does not provide a user interface. An Android Service has its own life cycle and can run indefinitely after it has been started. A service can be instantiated and started in three ways.

- Another component of the application that the service belongs to starts the service.
- The service is started by another application on the phone.
- The service is started by an event from the Android operating system.

Examples of events in the third case are that the phone boots, the phone receives a text message or the connectivity of the phone changes from WIFI to LTE.

A Service's lifecycle depends on how it was started. If it was started by a component *binding* to the service or if it was started by a `startService` call. This section will only handle services that are started from a call to the system's `startService` function since the runtime only will be started manually and not from other bounding application components.

When starting a Service, the service goes through a set of stages as shown in figure 3.1. The call to `onCreate` allows the service to initiate itself and handle everything that needs to be done before the actual task starts. The `onStartCommand` is the trigger to the service to actually start performing its task. This stage can

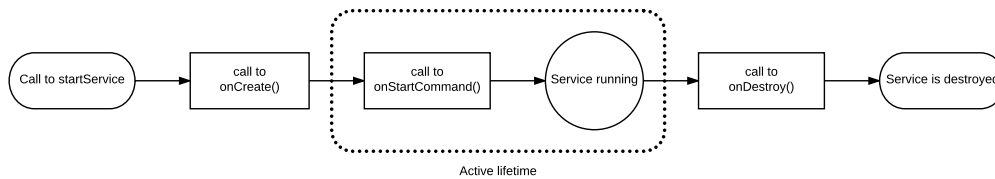


Figure 3.1: The lifecycle of an Android Service as described in [24].

live indefinitely, and will not stop until its task is performed or if it receives a message about stopping.

When an Android Service receives a call to be destroyed, it is up to the Service to stop executing its task, clean up its resources and exit. The reasons for a Service to stop is usually a call from the Service to itself about having finished its task. However other components of the app, and even other apps can also trigger the destroy command of a Service if they are allowed.

The Android operating system (OS) can also trigger a Service to be destroyed if the platform is running low on resources such as RAM [25]. Android can do this by keeping track of a prioritized list of all running processes based on their importance to the system. When the device is running low on RAM, it will destroy applications and processes with a low priority. Components that are bound to a graphical UI component and are visible to the user has a low risk of being destroyed while a long running Service in the background gets moved down in the list over time and risks being destroyed [25]. When the system has retained resources after having killed a service at a later point in time, it will start killed services again.

The operating system may perform actions to cache or even kill services that has ran over 30 minutes to save battery and RAM even if it is not critical for the system at that point in time. The operating system keeps track of a *last recently used* (LRU) list of active services which it can demote services that are running to long. This helps the system to avoid situations where long running services leak memory or interrupts the system in the background.[26]. When cached services in the LRU list needs to perform an important action, they are started again by the OS automatically.

3.4 Android Messaging System

Android implements a very competent messaging system for inter and intra application communications. Application components can register themselves in the OS to receive certain types of messages [27]. When messages are received they can be used to start application components that can perform an actions base on received messages.

Messages sent in the messaging system are called Intents and are instances of a data structure that can hold a set of key-value pairs for payload data. Intents are categorized in a namespace that depends on the sender of the intent which is used for identification purposes. It is the namespace that an application component uses to register for a certain type of messages. The namespace usually starts

with the sender applications package name in reverse order separated by a dot and concatenated with a string representing the intent's purpose. Intents that are meant to be distributed in the entire operating system to all applications that listen, are called broadcast intents. The OS uses broadcast intents to notify applications about events in the platform such as when the connectivity changes from WIFI to LTE. To receive connectivity changes, the application component has to register to listen for the `android.net.conn.CONNECTIVITY_CHANGE` [28] intent.

Applications and application components can also use the messaging system to send messages to the operating systems, mainly to request a startup of an application component. If an application component needs to start another component such as a service, an explicit intent is sent to the OS with information about the service. The OS receives the message, starts the Service, and handles its lifecycle. Application components may also ask the OS to be bound to already running Services. This lets application components, not necessarily from the same application, to communicate with an already running service.

3.5 Security

Android is designed to be an open platform that lets developers build advanced tools while still being easy to use for users. Android is built on top of the Linux kernel which means that it can take advantage of the robust Linux security system such as its user and permissions services [29]. Figure 3.2 shows the building blocks of the Android operating system where each of the building blocks in the figure assumes that the underlying block is secure and is implementing its own security measurements.

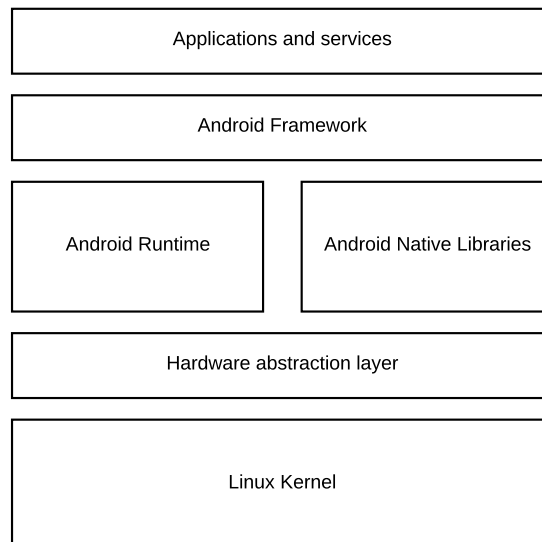


Figure 3.2: The Android operating system architecture [29].

The Linux kernel provides file permissions, isolation of processes, and RAM access control. All Android applications that use native code are compiled into position independent code, which means that the code can be executed from anywhere in the RAM. This allows the OS to use address space layout randomization for security.

3.5.1 Sandboxing

All code running on top of the Linux kernel is sandboxed which for applications means that they only have to provide security for themselves while relying on the underlying layers being secure. An application can for example use the file system in Android to write sensitive data in the application directory without having to worry about other applications reading that data. Secure access to sensors and other hardware components are done through the Android framework and the implemented API methods.

3.5.2 Application Permissions

If an application needs to access anything outside the sandboxed area, it must ask the operating system for permission. If the permission involves sensitive data such as access to the camera, the user of the device must explicitly give the application permissions [30]. For non-sensitive features such as Internet access, the platform can grant permissions upon application installation without asking the device user. Permissions are specified in an XML format in the Android Manifest file by using the `<uses-permission name=' [PERMISSION NAME] '>` tag. Examples of permissions relevant to this thesis are

- **android.permission.INTERNET** Request permission to access the internet via an unspecified interface.
- **android.permission.READ_EXTERNAL_STORAGE** Request permissions to read data from a persistent storage such as an SD card or flash drive
- **android.permission.WRITE_EXTERNAL_STORAGE** Request permissions to write data to a persistent storage such as an SD card or flash drive.
- **android.hardware.sensor.[SENSOR NAME]** Request access to a hardware sensor. The sensor name specifies what sensor to request access to. Examples of sensors are gyroscope, accelerometer, and proximity sensors.

3.6 Capabilities in Android

Mobile devices are normally equipped with a large amount of sensors and IO units. Android provides a hardware abstraction layer (HAL) through a framework for accessing these sensors. The framework divides the sensors into three categories, *motion sensors*, *environment sensors*, and *position sensors* [31] and lets

applications get access to raw sensor data together with calibrated and filtered data.

Via the specification of an application's permissions, an application developer can also specify requirements for an application to be installed on a device. A compass application could for example not be installed on an Android TV without a magnetometer. The framework also provides an interface for applications to poll the platform for information on available sensors. Based on this information, applications can adapt by disabling features that use unavailable sensors.

Calvin - The distributed IoT platform

As more and more devices get connected, the number of different platforms that need to be able to communicate with each other increases. Today cars, mobile phones, smart homes, smart cities, surveillance systems, and many more traditionally non connected devices are all connected to the Internet. There is a need for a scalable and dynamic platform that connects these devices to each other and lets them share their capabilities with the rest of the Internet. Calvin is a distributed IoT platform built to make it easier to develop, deploy and run applications for things that would like to talk to things [4]. It sets a standard for a communications protocol and defines how a distributed dataflow application runs. It does not matter if a Calvin connected device is a car, mobile phone, or a large data center. The Calvin platform handles all types of devices and lets the application take advantage of the platforms's capabilities.

4.1 Calvin Lifecycle

The life of a Calvin application is strictly defined by the Calvin architecture. All Calvin applications have to behave in a defined way and follow certain steps when being deployed, migrated, executed, and destroyed. The lifecycle of an application can be divided into four distinct steps - describe, connect, deploy, and manage [6]. This chapter describes these steps and explains what they do.

4.2 Actors

When developing a Calvin application one first has to define the smallest computational units, the *Actors*, of the distributed dataflow platform. An *actor store* is packaged together with the Calvin code base (see *actorstore* in [4] for available actors). This growing actor store contains the implementation of a number of actors that can make up an application. Examples of available Actors include actors for arithmetic computations, graph structure elements, sensor readers and I/O pin handlers.

Actors are very easy to implement in Calvin and are described using either the Python or C language. The developer has to *describe* the input ports, output ports and how the actor should be triggered to execute its task. An example is an actor that reads the temperature using a temperature sensor. The actor would typically

have two ports, one input port for triggering the reading of the temperature, and one output port for transmitting the result of the read. Since a temperature actor relies on the availability of a temperature sensor, the developer of the actor also has to specify that as a requirement for the actor. How requirements are specified and matched with capabilities is described in section 4.4.1.

4.3 Applications

4.3.1 Calvin Script

The *connect* stage of the Calvin lifecycle is when the dataflow graph for the application is described. The application graph is described using a small and simple to use script language called *CalvinScript*. There also exists a visual tool for generating CalvinScripts from a graph if one prefers visual programming.

In the first part of a CalvinScript all actors are specified. Then the connections and rules for the graph are described. A simple application graph is shown in figure 4.1 with its corresponding Calvin script in listing 4.1. There is one source node in the graph which is a timer that triggers at a given interval (of two seconds in the example). A camera receives the trigger from the timer and takes a picture. The bitmap is forwarder to the *sink* node that displays the image on a screen.

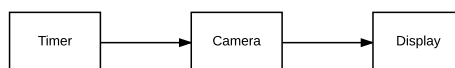


Figure 4.1: Simple Calvin application graph for a camera system.

```

1 src      :  std.Trigger(tick=2, data=1)
2 cam      :  media.Camera()
3 screen   :  media.ImageRenderer()
4 src.data > cam.trigger
5 cam.image > screen.image
  
```

Listing 4.1: Simple Calvin application for a camera system

As can be seen in listing 4.1, the actors are named in different namespaces in the actor store for organizational purposes. The simplicity of the Calvin script language makes application development very fast provided that the actors needed are already implemented. It lets the developer focus on the application itself while not having to worry about the underlying communication between the actors or platform specifics [32].

The CalvinScript language allows for grouping of actors into components. Components behave exactly like actors to the application and allows for easy reuse of larger graph sections [33].

4.3.2 Deployment

The actors of an application such as the one described in listing 4.1 do not necessarily all have to run on the same runtime. When deploying the Calvin application actors are deployed to runtimes that can handle that type of actor. If there is no runtime that can handle a specific actor, it becomes what is called a *shadow actor*. A shadow actor is an actor which is instantiated but not running and waiting to be migrated to a runtime that it can execute on. The **Camera** actor must for example run on a runtime that has access to a camera such as a laptop, and the **ImageRenderer** must run on a device that has a screen.

The deployment of an actor is not static and can change during runtime. The *migration* of an actor can either be triggered automatically through the control or automatically by the platform. In [8] Tomas Nilsson described how an actor can be migrated to and from a runtime based on factors such as the time of the day. The **Camera** actor could be moved around over the day to different security surveillance cameras based on their location.

An important aspect and consequence of the actor model is that an actor instance during runtime can be fully expressed by its internal state [6]. An actor can be fully serialized only by storing the data on the port queues together with the actor's internal variables. This allows for simple migration of actors during runtime without interrupting the running application.

4.4 Runtime

The entity that handles the execution of Calvin actors is called a Calvin runtime. Runtimes can exist on many different types of platforms and handle multiple number of actors at once. The runtimes can be controlled via the RESTful *control API*.

4.4.1 Requirement Matching

To decide on which runtimes the actors of an application should run, an application is deployed with certain requirements. The requirements of an application can be used to connect an actor to a certain runtime by name or geographical location. Calvin handles and matches actor and runtime attributes when deploying an application. If multiple runtimes fulfill an actors requirements the actor is instantiated at an arbitrary node decided by the runtime that handles the deployment.

Before an actor gets migrated to a runtime, the actor's system requirements must also be fulfilled. For a **Camera** actor to run on a certain runtime, that runtime must have access to and know how to use a physical camera. When implementing an actor, the developer must describe what resources from the platform that the actor needs to use. Calvin implements a HAL called **Calvinsys** to handle this. An actor may use Calvin API calls to request Calvinsys object references that can be used to access hardware components. The objects have the same structure on all platforms but the underlying implementation may vary.

Requirements are specified and organized in namespaces. If an actor has the requirement to use a camera, it must specify so by informing the Calvin runtime that

it needs to use the Calvinsys object named `calvinsys.media.camerahandler`.

4.5 Communication

Calvin defines a protocol in the application layer of the OSI model to be used for all communication between runtimes. The protocol is a text-format protocol and uses JSON for data formatting. The protocol does however allow runtimes to negotiate about data encoders to be used to encode the JSON data.

4.5.1 Runtime Addresses

The communication between nodes is completely transparent to Calvin applications, the application developer does not have to get involved in the actual mechanism that the runtimes uses to send data in between them. To identify a runtime and the method of communication, a certain address namespace is used internally in Calvin. All addresses are divided into an address part and a transport mechanism part as `[transport mechanism]://[address]:[address option]`. For example, for communication over sockets using the Internet Protocol (IP), the address could be `calvinip://192.168.0.5:5000`. The address is the IP address to the unit and the address option is used to specify the TCP port to connect to.

4.5.2 The Calvin Protocol

Calvin implements a pluggable infrastructure for the transportation and coding of messages between runtimes. The first thing that happens when two runtimes connect is that a `join request` message is passed from the entity initiating the connection. The message holds information about how the connection should be formed and how messages should be coded. The `join request` message is always uncoded and formatted using JSON.

Calvin currently implements data coders for ASCII coded JSON and the binary JSON like standard MsgPack [19]. The runtimes may however negotiate about using other data encoders if they are implemented. The protocol the runtimes uses is however well defined. Messages are sent as dictionary data structures that can hold key/value pairs of data and are called *tokens*.

4.6 Calvin Constrained

The vision of Calvin is to have a platform that allows any device to talk to any other device. There is a large set of common IoT devices with very constrained resources in terms of computing power, battery assets and radio connectivity [34]. A simple temperature sensor might not, and should not, need a full computer capable of running Python and Calvin base in order to work. Calvin Constrained (CC) is a Calvin implementation aiming at those constrained devices. By offloading some of the heavier tasks of a Calvin runtime, CC can run on very limited devices and execute actors and handle a HAL. CC is currently a project in fast development. The implementation can run on any platform that has a GCC compiler and is

capable of using sockets. CC has also been implemented and ported for Nordic Semiconductor nRF51 chips running an ARM CPU that uses IPV6 over Bluetooth for communications. With this thesis CC is also available for the Android operating system running on devices with an x86, ARM or MIPS architecture.

4.6.1 Architecture

Although Calvin Constrained is a full member of the Calvin platform, it relies on being connected to a Calvin base runtime as shown in figure 4.2. All communications to and from CC is proxied over the Calvin base host runtime, and tasks such as writing to the DHT can only be performed by the Calvin base runtime. There is therefore currently no way for a Calvin application to only run using instances of CC.

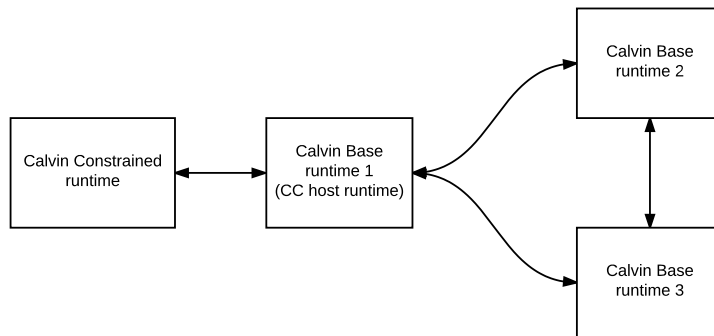


Figure 4.2: A setup of a Calvin system with a device running Calvin Constrained.

A CC runtime behaves just like any other Calvin runtime for an application or actor developer. It is possible to migrate actors to and from a CC runtime, and via Micropython [35], a CC runtime can even run actors written in Python. There is however, to the application developer, a hidden layer when a CC runtime connects to a Calvin base host runtime. Calvin constrained extends the Calvin protocol with the `PROXY_CONFIG`.

4.6.2 Setup

When a CC runtime has discovered and connected to a Calvin base runtime it issues the `PROXY_CONFIG` command. This is done in order to configure and set up the CC environment and register itself with the Calvin base host runtime. Configuration parameters are sent together with the `PROXY_CONFIG` command. The configuration parameters are stored in the DHT by the Calvin base runtime to make the CC runtime visible from other Calvin connected nodes. The capabilities of the CC runtime are specified using the normal namespacing structure as for any other Calvin device and is sent with the `PROXY_COMMAND`. An example of the `PROXY_CONFIG` command is shown in appendix A.

Mobile devices in Calvin

This chapter explains and discusses the proposed solution to the main problem formulation of how Calvin can make the best use of a mobile device. The chapter discusses different methods to solve problems concerning application structure, mobile device communication, battery consumption, and Calvin application capabilities access (Calvinsys). A demo application that shows how a mobile device as part of the Calvin platform is presented in the end of the chapter.

The solutions described in the chapter have been implemented as part of the Calvin Base [4] and Calvin Constrained [36] repositories and are available on GitHub for download. As part of the implementation, documentation on how to use the code and continue development has been created. The documentation is also available in the two Github repositories.

5.1 Architecture

Every application that is to be executed in Android is ran under the Android framework and must therefore be adapted to fit in the system. The proposed Calvin runtime application for Android has been divided into a native runtime part and a Java layer as seen in figure 5.1. The Java layer is responsible of maintaining the contact with the Android framework and make the Calvin Constrained implementation fit into the application framework. The native part of the application is responsible of running the actual Calvin runtime which originates in the Calvin constrained implementation.

The Java layer is furthermore divided into four components as can be seen in figure 5.1. The layer exposes an API to let third party applications on the mobile device communicate with the Calvin runtime. The API is presented and discussed in section 5.10. The Java layer uses the Android API methods to implement a transportation method for Calvin using FCM which is explained in section 5.2. The Java part of the Calvin Android application also controls and maintains the lifecycle of the native runtime code. It controls when the runtime is to be started, stopped, and how it is configured. There is a channel of communication between the Java part of the application and the native part. The Java layer implements a protocol to be able to communicate with the native part of the application which is described in section 5.5.1.

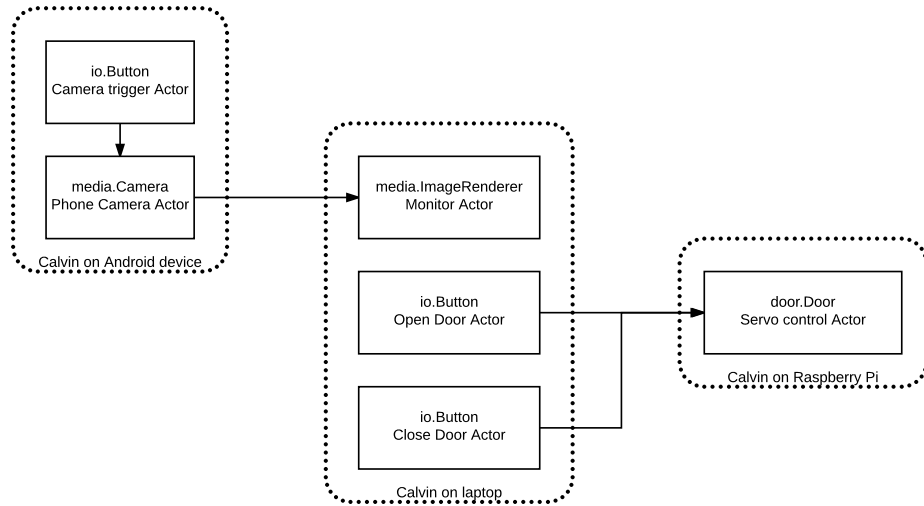


Figure 5.1: Calvin Constrained for mobile devices architecture

5.2 Connectivity

A mobile phone can communicate over the Internet using many different methods depending on the supported hardware. Most modern phones can send data over 3G, 4G, WIFI, USB, Bluetooth and the soon to be 5G. The big difference between a mobile device and a more static device such as a PC or Raspberry Pi is that mobile devices move around and change connectivity method very often. If a user of a mobile phone enters a building, the mobile phone may connect automatically to the WIFI if it has been connected before. And in the same way, when a user exits the building he may connect to the Internet using a cellular network. Even when connected to cellular networks, the mobile device's connectivity method may change between 3G, 4G, and 5G depending on cellular coverage. With a change of connectivity the IP address usually changes. Since the mobile device is changing network the number of accessible devices through a local IP address changes. The need of a universal connectivity method that works on any network where it can access the Internet is a must for mobile devices. The mobile device must be able to communicate even if the proxy runtime does not have a static IP address or a DNS hostname pointing at it. The communication channel must also be able to bypass any firewall, such as built in firewalls in home routers.

5.2.1 Transport Mechanism

Calvin constrained has support to communicate using either a socket implementation or IPv6 over Bluetooth on the Nordic Semiconductor NRF52 platform. In the case of Bluetooth, the communication is very limited. Bluetooth requires the devices to be within a distance of 100 meters [37], has relatively low bandwidth and is not ideal for mobile devices that move around a lot. The CC runtime may

also connect to the proxy runtime using a socket. The socket connection however requires the CC runtime to know the IP address or hostname to connect to, this is not the case for most common Calvin base runtimes. The Calvin base runtime is often running on an isolated LAN behind a firewall with no open ports and there is often no DNS record for that IP address.

The proposed and implemented solution to solve the connectivity issues are to use a third party server as shown in figure 5.2. The third party server is accessible from the whole Internet via a record in a domain name system (DNS) server and acts as a proxy server. It is the responsibility of the mobile device and the Calvin proxy runtime to act as clients and always keep an open connection to the proxy server. If a mobile device loses its connection to the proxy for example from a change in connectivity method, it is the mobile device's responsibility to open a new connection to the proxy server. By keeping the connections open at all time, data can be routed from mobile devices to the Calvin proxy runtime and vice versa.

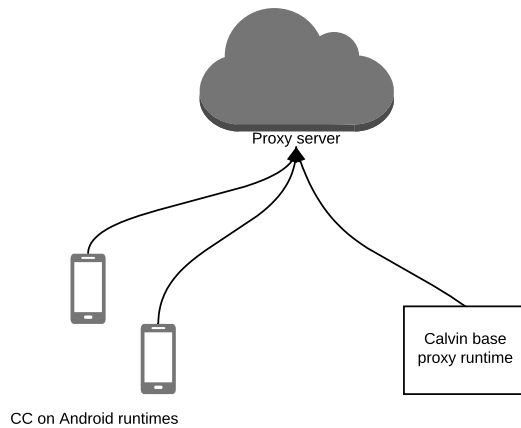


Figure 5.2: Communication between CC and Calvin proxy runtime using a proxy server.

The use of a proxy server adds a layer to the protocol used for all datapackets sent. The proxy server uses this layer to implement an address namespace for the Calvin base proxy runtimes and the CC runtimes running on the mobile phones. Together with the destination and source addresses, information about authentication is sent to keep the communications channels secure.

Both Google and Apple, the vendors of Android and iOS, distribute services, servers and APIs that are of help when implementing the proposed solution above. Google distributes a service called *Firebase Cloud Messaging* (FCM) [38] and is built into the core of the Android operating system. Apples version of a similar service is called *Apple Push Notification Service* [39] (APNS).

There are many advantages of using a widely spread and maintained service such as FCM and APNS. The services are built into the operating systems which

maintains the long lived connection to the proxy server in a battery and data efficient way. In the case of Android and FCM, the service only keeps one shared connection to the FCM service for all apps running on the phone. FCM may bundle data from different apps and send them at the same time to make the connection more efficient. In the same way, upstream messages from the server may be buffered in the proxy server if the mobile device is in hibernation mode or if it is not efficient for some other reason to send it right away. All these algorithms for battery and data efficiency are built into the FCM system and should therefore be used opposed to building a separate proxy server solution.

5.2.2 Firebase Cloud Messaging Layer

The connection between the mobile phone and the proxy server in FCM is for Android developers a big black box. The communication is handled by Android, and Google has created APIs to receive and send messages over the tunnel. However, the tunnel between the proxy server and the Calvin base proxy runtime is well defined. The communication is done over a UDP connection and encrypted using TLS. The application layer protocol used is XMPP. FCM uses XMPP for authentication as described in section 2.4. The communication for payload packages is then done using JSON wrapped in a XML structure as seen in listing 5.2.

It is important to have a defined way in a distributed platform to identify and be able to connect to the nodes. For IP connections the address schema `calvinip` is used inside Calvin to route traffic to the correct host. For connections established over FCM, a new schema `calvinfcm` was proposed and implemented. The schema has the form

```
calvinfcm://[sender id]:[device token]
```

For each setup of Calvin that needs to use FCM as a transport mechanism, an API key must be retrieved from Google. A part of the API key is the sender id. The sender id is a numeric string, unique for each setup of FCM and is connected to the Android applications package name. The device token is a unique token for each mobile device and works as an identifier. The token is retrieved by the Android device from the proxy server the first time the application is started. For servers (Calvin base proxy runtimes) listening for messages from any device in the given sender id namespace, a `*` is allowed as a wildcard address.

To start a Calvin base runtime proxy that listens for FCM messages, one must use the `--uri` flag. An example is that one can start a runtime that listens for FCM messages and initiates a socket server by using the `csruntime` tool with the flags

```
csruntime --host 127.0.0.1 --port 5000 --controlport 5001 --uri  
calvinfcm://773482069446:*
```

When starting a Calvin base proxy runtime with a URI in the `calvinfcm` namespace, the runtime starts by authenticating itself with the proxy server. An example of the authentication handshake with a Google FCM server as a proxy

server is shown in appendix A. When the Calvin proxy runtime is authenticated, it can receive and send data packets over the open link.

As shown in figure 5.3 there is a structure in the layers of protocols when communicating over FCM in Calvin. The application layer in the OSI model [40] is divided into three new layers, the protocol used by FCM, the protocol to handle FCM Calvin messages and the actual Calvin protocol. Comparing to the normal Calvin communication over sockets, the two extra layers for FCM and Calvin FCM have been added.

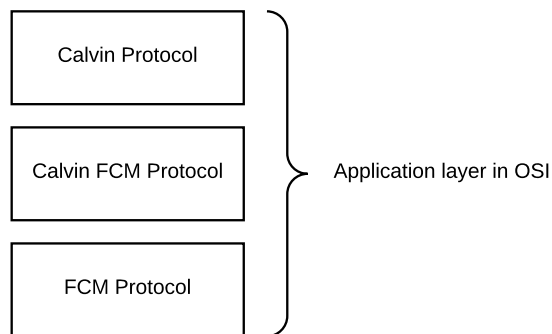


Figure 5.3: The three layers of the protocol used for FCM in Calvin.

The FCM layer is handled by the definition of the FCM service. All data sent over FCM is in a JSON structure wrapped in an XML tag. The Calvin FCM layer is used by Calvin to bundle information about the messages and is used for setting up and destroying Calvin links.

The Calvin FCM layer specifies two different types of messages. The message type is specified with the `msg_type` key in the `data` structure. The purposes of the three different message types are

- **set_connect** is used to initiate and destroy an FCM connection. The key is used together with the `connect` tag that can hold a boolean value of `true` if the connection is to be initiated or `false` if it is being destroyed. An example of a connect message is shown in listing 5.1. If this message is sent to initiate a connection from the mobile device, this call can be compared to a system call to `connect()` on a socket when using sockets for transportation. When the Calvin proxy runtime receives this message it sets up an FCM connection and returns an ACK message to the mobile device.
- **payload** is used with the `payload` key to send raw Calvin messages as can be seen in listing 5.2.

```

1 <message id="[message id]">
2 <gcm xmlns="google:mobile:data">
3 {
4   "data":{
5     "msg_type":"set_connect",
6     "connect":"1"
7   },
8   "time_to_live":0,
9   "from":"[unique device token]",
10  "message_id":"[unique message id]",
11  "category":"ericsson.com.calvin.calvin_constrained"
12 }
13 </gcm>
14 </message>

```

Listing 5.1: Example of a FCM connection message.

Listing 5.2 shows an example of an FCM message with Calvin payload data and the necessary key/value pairs needed by the FCM protocol. Payload data in FCM is wrapped in the JSON structure and identified with the `data` key as seen in the example in listing 5.2. The `data` tag is used by the Calvin FCM protocol to send messages about connection, disconnection and payload data messages. The `msg_type` tag is used to identify the type of Calvin FCM message being sent. The actual Calvin payload data is identified by the `payload` string. Tags in the first level of the structure are parts of the FCM protocol and are used by the proxy server. The `time_to_live` (TTL) tag is used to specify the time that the server should buffer the message if the device or Calvin proxy runtime is unavailable. A value of 0 informs the proxy server to deliver messages immediately or never at all. The `message_id` tag is a uniquely specified value for each sent message over the proxy server. This value is used to identify the messages and is sent with ACK messages from the Calvin proxy runtime to the FCM proxy server. The protocol also requires the `from` tag to be specified. This is for mobile devices, the device token and for Calvin proxy runtimes the sender id. The `category` tag links the message to a certain Android application.

```

1 <message id="[message id]">
2 <gcm xmlns="google:mobile:data">
3 {
4   "data":{
5     "payload":"[base64 encoded data]",
6     "msg_type":"payload"
7   },
8   "time_to_live":0,
9   "from":"[unique device token]",
10  "message_id":"[unique message id]",
11  "category":"ericsson.com.calvin.calvin_constrained"
12 }
13 </gcm>
14 </message>

```

Listing 5.2: Sample FCM packet with payload data.

Calvin constrained uses Message Pack instead of JSON to code payload data. Message Pack is a binary data structure and cannot be directly wrapped in JSON. The solution is to code all payload data to be sent over FCM using the Base64 algorithm described in section 2.5.2 since that base is represented by allowed characters in JSON. The Base64 encoded data can then be wrapped in the JSON message structure used by FCM. The encoding however increases the size of the actual payload data. This effect of the encoding is discussed and shown in chapter 6.

5.3 Discovery

When starting a CC runtime, a list of URIs to proxy runtimes has to be specified. One can trigger a search over the *simple service discovery protocol* (SSDP) by specifying the string `ssdp` in this list. When a runtime has been found using SSDP, the CC runtime can connect to the found runtime and use it as its Calvin proxy runtime.

When using the FCM transport mechanism however this is not needed. The mobile device can connect to the Calvin Base proxy runtime regardless of the network the mobile device is connected to as long as the mobile device knows the sender id of the Calvin base runtime. This allows for very useful applications to be written. For example, if the runtime is started when the device only has LTE connectivity, it can still find the Calvin proxy runtime to connect to and start sending and receiving data.

5.4 Choosing Transport Mechanism

When the runtime on the mobile device is started, a list of Calvin interfaces which it can use for communication is specified. The list is prioritized in the order of which Calvin should try to find and connect to a Calvin proxy runtime. To make the best use of this feature on Android, the list should start with a `calvinip` address followed by a `calvinfcm` address. If a connected interface loses its connection, the runtime tries to connect to the other interfaces in the list. For example, if a runtime is connected over Wifi using a socket and the mobile device loses its connection, the runtime will switch the connectivity method and start using FCM for communication with the proxy runtime.

The FCM transportation method is almost always connected to the proxy runtime via the FCM server. In some cases FCM may be slow, and may not be the best choice of transportation method. For example if the device connects to a network that can be used for direct communication over a socket to the proxy runtime, it may still use FCM as long as the mobile device has internet connectivity. The Calvin service on Android devices therefore uses the broadcast intent messaging system, provided by the Android OS to listen for connectivity changes. When the Calvin runtime is started, the service registers for the `android.net.conn.CONNECTIVITY_CHANGE` intent. The Android OS sends this to indicate a change of connectivity, and by filtering these messages, newly connected networks may be discovered. When a newly connected network is discovered by

the Calvin service, the runtime is indicated to try to connect to the highest prioritized interface. This way, the mobile device will always try to use the best interface possible at all times, but can always rely on falling back to FCM.

5.5 Calvin and the Mobile Platform

5.5.1 Platform communications

The native Calvin runtime part of the application must run in its own thread in an Android Service at all times, since it is a completely blocking task. The native CC runtime is completely asynchronous and only acts on data written to file descriptors. To make the Calvin runtime be able to communicate with the Java layer of the application a UNIX pipeline is set up at the initialization of the application. The pipeline allows the Java part to write data that can be read by the native part and vice versa by letting the native part listen for the pipe's file descriptor.

5.5.2 Pipe Protocol

The pipe protocol is a simple binary protocol with three parts as seen in figure 5.4. The first four bytes (32 bits) in each message sent over the PIPE are set as the total size of the data. The following two bytes (16 bits) are set as two human readable characters to represent a command as described below. The bytes after the command bytes are used for payload data. The size of the payload message is only limited by the largest number that is possible to write in the size part (minus two bytes needed for the command). Since the size part consists of 32 bits, the maximum number of bytes in the payload is limited to $2^{32} - 2$ bytes. The size of the payload data plus the size of the two byte long command must however match the size set in the first four bytes. The contents of the payload data is specified by the different commands but is expected to be Message Pack encoded.

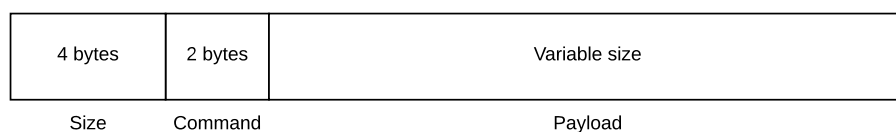


Figure 5.4: Figure explaining the sections of the data sent over the pipe.

The different commands are used to trigger different tasks and can be sent either way on the pipe. The implemented commands are as follows with the two command bytes within parentheses.

Runtime started (RR) is sent from the native end of the pipe to indicate that the native runtime is started, connected to the proxy runtime and ready to receive incoming tokens. There is no payload data sent with this command.

Runtime stop (RS) is sent from the Java end of the pipe and indicates that the runtime should terminate itself. There is not payload data sent with this command.

Runtime stop and serialize (RA) is sent from the Java end of the pipe and indicates that the native part should serialize itself and write the contents to the disk. When it has been serialized it should terminate itself. There is no payload data sent with this command.

Calvin data (CM) is sent from the Java end of the pipe to pass payload data directly to the CC runtime code. The payload data of this command is handled in exactly the same way as data received on a socket connection when using sockets for transportation would be. This command is used by the FCM transportation mechanism to pass Calvin messages between the native part and the Java part.

FCM Connect (FC) is sent from the native part of the application, more precisely from the FCM transportation implementation, to indicate that the Java part should send a FCM connect message to the proxy runtime. This command is part of the FCM transportation mechanism and is more explained in section 5.2.2. There is no payload data sent with this command.

FCM Connect reply (FR) is sent from the Java end of the application. The message indicates that the Java part of the application received a connect reply message over FCM from the proxy runtime. This command is part of the FCM transportation mechanism and is more explained in section 5.2.2. There is no payload data sent with this command.

Trigger reconnect (RC) is sent from the Java end of the application to indicate a change in connectivity. This is for example sent when the mobile device loses WIFI connectivity and is connected over LTE instead. This command is more explained in section 5.4. There is no payload data sent with this command.

Register Calvinsys (CS) is part of the Calvinsys API described in section 5.10. The CS command is sent from the Java end of the application to indicate that a third party application is registering itself to be a Calvinsys. The payload data of the command is MessagePack encoded data and contains information about the Calvinsys object to be created and the third party application.

Destroy Calvinsys (CD) is part of the Calvinsys API described in section 5.10. The command is sent from the native part of the application to tell an external Calvinsys application that the Calvinsys was destroyed by the application. The payload data is MessagePack encoded and contains information about the Calvinsys object to be destroyed.

Init Calvinsys (CI) is part of the Calvinsys API described in section 5.10. The command indicates that an actor would like to use a Calvinsys object. The payload data contains information about the Calvinsys object to be initiated.

Calvinsys payload (CP) is part of the Calvinsys API described in section 5.10. The command can be sent both from the Java and native end of the pipe to indicate data being sent to and from a Calvinsys implemented in a third party application. The payload of the command is MessagePack encoded and holds the

data to be sent between an actor and a Calvinsys object. One can think of this command as a mechanism to simply forward data sent when using the read and write functions of a Calvinsys as described in 5.10.

5.6 Power Management

Power management is an important topic that every mobile application must be aware of and make the best of. A mobile device is very limited when it comes to power and users expect the mobile device's battery to last for a long period of time and not to be drained quickly. Applications that drain too much battery from the device are likely to be uninstalled and are not appreciated by the users.

Techniques to reduce the power consumption of the Calvin runtime have been implemented in especially one major way. The Calvin runtime can go into a deep sleep mode where its Service is completely killed. The decision to kill the Calvin runtime can come from two different sources, the operating system or by runtime inactivity. As described in chapter 3.3 the operating system may destroy the service for different reasons, mainly to let other applications execute when the Calvin runtime is in the background and not bound to any UI component.

The trigger to destroy the Calvin service may also come from inactivity. When the runtime has not received or sent any token on the active transportation method for a specified time, the service is destroyed. The default time for destroying the service because of inactivity is ten minutes. This time may however be modified during compile time by defining the `SLEEP_AT_UNACTIVITY` variable in the C pre-compiler. When the service is destroyed it does not let the other nodes in the Calvin platform know about it being destroyed. From the other nodes, the Calvin runtime on the mobile device always seems to be running to keep the availability of the node high. This means that other nodes that would like to communicate with the sleeping runtime must have a method to wake it up.

A sleeping runtime on a mobile device can only be woken in two ways. Either by a manual trigger to wake the Service or by a trigger from a token sent over FCM. Since the Android operating system always keeps a link to the FCM proxy server, the Java layer of the Calvin runtime is always listening for tokens sent over FCM. If the runtime is sleeping when an FCM token is received, it simply wakes the runtime up and passes the FCM message to the runtime when it is ready.

5.7 Serialization

When a runtime goes to sleep it is expected to be able to wake up at any time and continue executing without interrupting the Calvin application. Since the Calvin runtime is completely destroyed when it goes into deep sleep it must have methods to be able to save its state to a persistent storage area.

It is easy to serialize the complete state of a runtime since it is fully defined by the running actors together with meta information about the runtime's configuration. An actor is further serializable since it is fully defined by its internal state and the port queues. This property is used when running any Calvin application. When an actor is to be migrated or deployed, it is serialized to JSON and sent

to the receiving runtime. The actor can then be instantiated using the serialized data. Since Calvin Constrained uses MessagePack to encode all data sent to and from the runtime, it is a natural choice to serialize the runtime into MessagePack encoded data.

When a runtime is destroyed, it serializes itself and stores the data on the mobile device's SD card or internal storage area. At a later point in time when it is supposed to wake up, this file can be read and used to restore the runtime to exactly the same state as it was when it was destroyed. The serialized runtime is stored in the isolated Android application storage area. This ensures that no other application on the phone can access the serialized runtime and its data.

5.8 Configuration and Key Handling

5.8.1 Proxy Runtime

The Calvin proxy runtime can be started using the standard `csruntime` tool with the `--uri` flag to specify that it should use FCM as a transportation method. The proxy runtime must however also know the FCM API key which is specified in a Calvin configuration file. An example of a configuration file to specify the FCM API key is available in appendix C.

5.8.2 Android

The Android runtime is started in a completely different way than the proxy runtime. The user who is starting CC on Android does not in general have access to a command line interface and can therefore not specify runtime flags. The configuration of the Android runtime has been divided into two parts, configuration via an UI and build configurations. The build configurations are set by defining pre compiler variables using the gcc compiler flag `-D`. The variables that have been added to extend the original CC build systems as part of this MSc thesis are as follows.

PLATFORM_ANDROID is specified to build CC for Android.

SLEEP_AT_INACTIVITY is specified to build the runtime to let it go into a sleep mode when no tokens are sent or received by a specified time.

PLATFORM_PIPE is specified to build the runtime so that it sets up a UNIX pipe that can be used for communications with the platform. This is used to activate the functionality so the native part of the application may communicate with the Java part.

TRANSPORT_FCM is specified to build the files needed to communicate using the FCM transport mechanism.

To control the actual runtime a simple UI was created which can be seen in figure 5.5. The UI allows the Android user to start and stop the runtime and to configure parameters. The name of the runtime may be set together with a list of URI:s that the CC runtime should use when connecting to the proxy runtime.

The application also allows the runtime to be started as soon as the application is launched through the autostart feature. The user may also specify that the CC runtime should be started in a clean way and not use any serialized state that potentially could exist on the disk.

5.9 Capabilities

A modern mobile device is a small and portable computer running an advanced operating system and equipped with copious sensors. Many devices have sensors and capabilities including a touch screen, dual speakers, an accelerometer, a gyroscope, a humidity sensors, a pressure sensor, and a proximity sensor. All of which can be used cleverly by mobile phone applications. However mobile devices have very different hardware, some models may introduce new sensors that are not available on older phones, and some models may simply be simpler models with less number of sensors. Calvin introduces a capability matching system to match capabilities with actors that can execute on a runtime based on its capabilities.

5.10 Hardware Abstraction Layer

The hardware abstraction layer (HAL) Calvinsys has not been subject to much research within Calvin. As described in section 4.4.1 Calvinsys is used by actors to access hardware that is available to the runtime through an API. A Calvinsys object is returned by the platform to an actor when it needs to use a hardware feature such as a thermometer sensor. It is up to the actor developer to know how the structure of the returned Calvinsys object looks like and what the available functions are. Different Calvinsys objects have a different set of functions that take different parameters. There is no version handling of the Calvinsys implementations, and making a change to a Calvinsys implementation may result in errors in many actors. Calvin is in need of a generalized way of accessing hardware that is future safe and allows for expansion.

The proposed solution for the new hardware abstraction layer in Calvin was inspired by how the Linux kernel handles device drivers. Linux device drivers are categorized into three types, *character (char) drivers*, *block drivers*, and *network interfaces* [41]. The proposed solution for the new Calvinsys is somewhat similar to char drivers in the sense that almost all communication with the driver, Calvinsys object in Calvin, is a character based stream of data. A convenient way of implementing a char driver is therefore to let the driver expose a *file descriptor* (FD) that applications on the device can use. Applications that would like to communicate with the driver can write and read from the FD to access functions in the driver.

To make the HAL in Calvin look and behave the same among different platforms and implementations of Calvin, a generalization and standard was set. The standard states that every Calvinsys implementation must expose four functions that are well defined and visualized in figure 5.6. On all platforms where the Calvinsys is implemented, these functions must behave the same and perform the same actions. The functions are meant to be exposed for every Actor that needs to

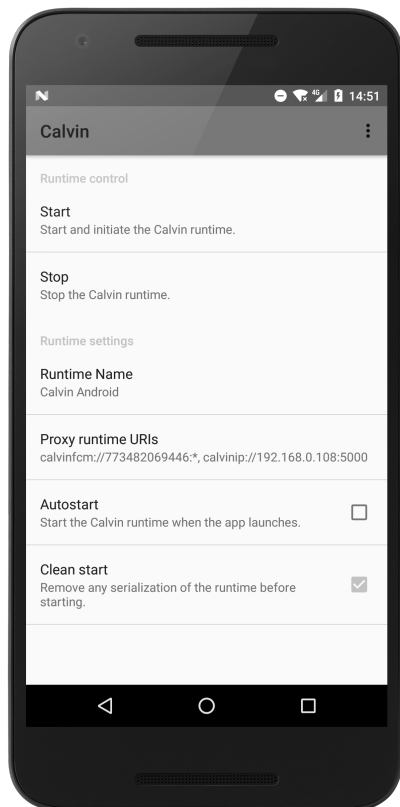


Figure 5.5: The Android UI for controlling and configuring the Calvin runtime.

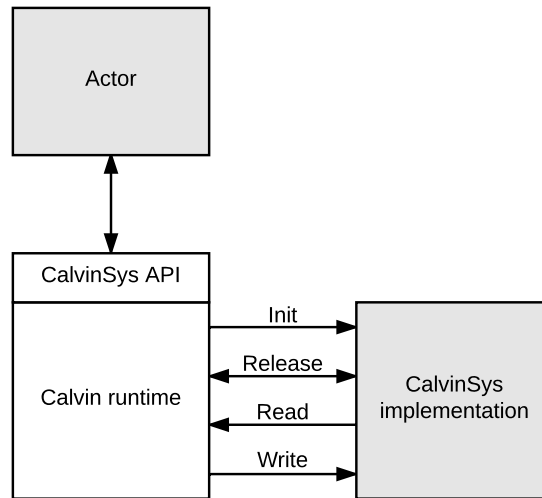


Figure 5.6: The architecture of the new CalvinSys implementation.

access a CalvinSys implementation. The Actors can do this by requesting a CalvinSys object by using an API exposed by the Calvin system. The CalvinSys API handles all instantiation and distribution of CalvinSys objects on the platform.

The CalvinSys implementation is plugin based depending on the platform type the Calvin runtime runs on, since it may vary a lot. For example may there be thermometer sensors of different models on different platforms, but they should all be accessed in the same way through the CalvinSys. The actual implementation of the CalvinSys should be a black box to the actor developer who only should see the four exposed functions, which are:

- Init
- Release
- Read
- Write

When an actor needs to use a hardware device it can request a reference to a CalvinSys object that exposes the functions above. This is done through the CalvinSys API which is defined in the Calvin runtime. When an actor actually wants to use the requested CalvinSys it must call the **init** function of the object. This lets the CalvinSys object get a chance to initialize variables and allocate memory for future use. If the CalvinSys object is for measuring the acceleration using an accelerometer, the init function can for example tell the hardware to initiate the sensor and start measuring.

When an actor has finished using the CalvinSys object calls the CalvinSys to clean up its resources and stop its task by calling the **release** function. The actor may also get a call from the CalvinSys object telling the actor that the hardware has become unavailable through a release call. The actor must be able to receive

these calls and handle the unavailability of the hardware. In cases where the actor requires the Calvinsys in question, it must be migrated to another runtime. If no other runtime exists that can handle the actor, it becomes a shadow actor. A call to the release function is only valid if the Calvinsys object has received a previous call to init. But a release call can be followed by calls to init again if the actor would like to use the hardware at a future point in time.

The two functions **read** and **write** are used by the actor to communicate with the Calvinsys when it is active. The actor may call the write function to send messages to the Calvinsys object. To make it easier for actor developers, messages are divided into a command part and a payload part. The command is issued to the actor as a statement to do something. For an LCD monitor, this could be a trigger to show an image. Every command sent can be bundled with payload data. This data must be a dictionary structure that can hold key/value pairs that can be serialized to a JSON format. There may therefore not be any binary data in the payload part of a message to a Calvinsys object. If the actor needs to send binary data it must first encode it using Base64 or similar. An example of payload data is when triggering a monitor to display an image. The image may be Base64 encoded and sent as a key/value pair to the Calvinsys object. The dictionary structure used to hold the data may be implemented in different ways depending on the language that the runtime is written in. The structure in the Python language would be a Python dictionary while the dictionary is encoded using MessagePack in Calvin Constrained.

The actor may read data from the Calvinsys object by a call to the read function. If there is data available from the Calvinsys object it will be sent in the same structure as when writing data to a Calvinsys object with a command and payload part.

The structure of the proposed new Calvinsys implementation comes with many benefits. It makes it easier for both actor and Calvinsys developers to create new components. The protocol to be used for the payload data and the command part must however be well defined and the same for all implementations. The proposed Calvinsys object is however easy to upgrade and change without affecting older systems. It is just the internal code of the Calvinsys implementation that has to be changed, and there can be no change of the exposed functions. One may also create commands to let actors poll the Calvinsys object for available functionality. This could be useful since similar sensors may work in different way. On one runtime a temperature sensor may be polled for data while it may write the temperature on the read port when it has data on another runtime. By letting these two methods of collecting temperature data be exposed in different way, the application developer has more control over the system.

For future implementations the proposed HAL structure could allow the deployment mechanism in Calvin to not only filter on requirements but also on the available set of commands for a Calvinsys implementation. This allows the runtimes to filter on a finer level than before. An actor may for example require that an accelerometer can deliver data with an accuracy of 64 bits by looking at the available set of commands.

5.10.1 Dynamic Calvinsys Loading

The structure of the generalized Calvinsys object allows for development of dynamically loaded Calvinsys objects. The Calvin runtime may expose an API and let other entities register themselves to expand the capabilities of a runtime. Since all data that is sent between the Calvinsys object and the actor is characters, this API could for example be exposed using the control API over a normal socket port or USB interface. This could allow for example a USB camera that is connected to a device, to register itself with the Calvin runtime and allow actors that require a camera to be migrated. The API can also be exposed over the internal messaging system used in Android to allow third party applications to register themselves with Calvin to increase the number of capabilities in the platform. This is further investigated in section 5.11.

When a CC runtime starts it issues the `PROXY_CONFIG` command to a proxy runtime to set itself up as described in section 4.6.2. This command holds information about the runtime's capabilities which is stored in the DHT. When a Calvinsys object is loaded dynamically during runtime, it must tell the platform about its new capabilities. By resending the `PROXY_CONFIG` command with a new set of capabilities specified, this can be overwritten in the DHT to allow actors to be migrated.

5.11 Third Party Applications

One of the great functionalities of a mobile Android device is implemented in different downloadable applications. Applications may implement features such as the ability to post on Facebook via the Facebook application or authenticate users via the Bank-ID application. The nature of an Android device lets mobile applications depend on other mobile applications. If one receives a URL to a web page in a text message, the messaging application may tell the browser on the phone to display the contents of the web page since the messaging application is unable to render web content. Similarly other applications on the phone need to be able to communicate with the Calvin runtime. As part of this MSc thesis an API has been exposed to let applications register themselves to implement a Calvinsys. For example could the Twitter Android application register itself to be let Calvin applications post on Twitter. When an Actor then would like to use the `web.twitter` Calvinsys, the implementation of the Calvinsys would be in the Twitter application. The Calvinsys would still behave and act like any other `web.twitter` Calvinsys for the Actor.

5.11.1 API

The Calvinsys API for third party applications is exposed over the bind Service functionality in Android and fits into the Calvin Android architecture as seen in figure 5.1. If an application on the phone would like to register its ability to implement a Calvinsys it first has to bind to the `ericsson.com.calvin.calvin_constrained.CalvinService` Service. The com-

munication between the third party application and the Calvin application is handled using the Android Messenger functionality.

When the third party application is bound to the Service it can start using the Calvinsys API. The proposed structure of the Calvinsys as described in section 5.10 is used as a base to implement the API. The API exposes six methods needed for the external Calvinsys implementations. It exposes the four functions needed by an Actor to communicate with a Calvinsys as described in 5.10. The API also implements one method that third party applications can use to register themselves as a Calvinsys implementation and one method to unregister themselves.

Every message sent over the API is divided into a `what` variable and a set of key/value data pairs. The `what` variable is used to tell the receiver what method should be called and the key/value pairs can be used for payload data.

When a third party application is bound to the Calvin Service it issues the `REGISTER_CALVINSYS` call over the retrieved Messenger. The call to register a Calvinsys is bundled with the name of the Calvinsys, as a key/value data pair, that the third party application implements. When the Calvin Service receives a `REGISTER_CALVINSYS` message it dynamically loads the Calvinsys object as described in section 5.10.1 which allows actors that use the loaded Calvinsys to be migrated to the mobile device.

If a Calvinsys that an Actor would like to use is implemented in a third party application, all calls to the four Calvinsys methods are forwarded to the third party application by the platform. It is no difference for Actor in how they use Calvinsys objects depending on how they are implemented. It is simply up to the third party applications to handle forwarded calls over the Messenger API from Actors.

5.12 Proof of Concept

To show the strengths and flexibility of the proposed and implemented solution for how Calvin should handle mobile devices, an example application was created. The example application, which is shown in figure 5.7 runs on three devices. It needs an Android device, a computer and a Raspberry Pi with a connected servo motor.

The example demos an access control system for a door. Commonly access systems include a surveillance camera next to a door, and a person monitoring the camera to open the door for authorized persons. The idea of the example is to replace the camera with the camera in people's mobile phones. When a user would like to get authorized to access a door, two actors are migrated to the person's phone. The person uses the camera in the mobile device to take picture of his face. The picture is sent to the person monitoring the door who can decide whether or not to let the person enter. When the person has entered the door, the actors can be migrated to the next person in line to enter for authentication.

The application graph for the setup is shown in figure 5.7. On the users mobile phone there is a camera actor and a button actor to trigger the camera to take a picture. The application clearly needs a UI on the mobile device to be able to display the trigger button together with a view showing the cam-

era image. The button and camera actors need the `calvinsys.io.button` and `calvinsys.media.camerahandler` CalvinSYS to run. These CalvinSYS are implemented in a third party application which also handles the UI. When the third party application is launched, it register itself to handle the two CalvinSYS implementations. This allows the button and camera actor to be migrated to the phone.

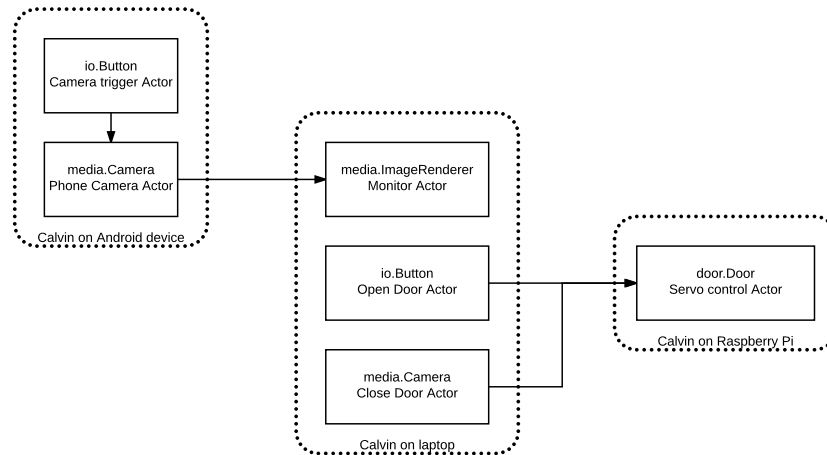


Figure 5.7: The application graph of the example application.

When the mobile phone user takes an image, the image is JPEG compressed and encoded using Base64 since no binary data is allowed in the communication between an actor and a CalvinSYS. The image is forwarded to the computer which runs a media renderer actor. The computer also runs two button actors that are connected to the Raspberry Pi to allow control of the door. The Raspberry Pi runtime runs the Door actor that knows how to control the door.

The demo application and instructions on how to run it are available in the Calvin repository [4] on Github.

Evaluation and Result

A mobile phone is a very powerful device with a lot of computing power and has access to many different sensors. The device is however very limited when it comes to RAM and battery usage. Mobile device users expect the device's battery not to drain quickly and the availability of running many applications at the same time. All applications on a mobile device must therefore be very careful in the use of RAM memory and must run in a power efficient way. This is especially important for long term running applications such as Calvin since small errors in the code may result in large memory leaks or unnecessary battery consumption.

The Calvin runtime application was analyzed and profiled in different aspects regarding the consumption of RAM. One of the most power consuming component of a mobile device is the radio. It is therefore crucial to keep the data transmissions small and only send data when necessary. The proposed FCM protocol was analyzed and compared to the socket implementation.

This chapter will explain the procedure of how these evaluations were done and present the results of how the application behaves.

6.1 Evaluation Method

6.1.1 RAM Heap Analysis

Android handles RAM memory for Android applications in a very complex way that is conceptually different from normal programs in Linux. Since every Android application is executed on top of the Android runtime (ART) Java virtual machine(JVM) it is up to the ART JVM to manage the heap.

In every Android system there is a background process called *Zygote* which is started when the device boots. When an Android application is to be started by the OS, this process is forked. The new process is used to load the application's code and start executing it. This allows every application on the system to have an area of shared memory with other applications and the OS. This area is used to store Android framework code and is used when memory needs to be shared across applications [42].

The heap of an Android application is divided into two different groups, private and shared heap memory. The two groups are then further divided into *clean* and *dirty* heap memory. The clean memory are pages in the RAM mem-

ory that is mapped to a disk while the dirty memory only exists in RAM. The dirty heap memory is of little interest in Android since Android does not swap and hence never maps RAM pages in the heap to the disk. There exists many profilers for investigating the RAM such as the `massif` tool in Valgrind. The tool can however not differentiate between the different heaps used by an Android application. Google provides profilers for different aspects of the RAM where the most capable is called `dumpsys meminfo`. The `dumpsys` tool allows for analysis of both private and shared memory using a simple to use command line interface. There also exists API methods in the Android framework for collecting data from `dumpsys` which were used for the analysis in this thesis. To get an idea of the available data in the tool an example of the output from the command line interface tool is provided in listing 6.1. The tool was executed using `dumpsys meminfo ericsson.com.calvin.calvin_constrained` in an Android shell as the `shell` user.

```

1 Applications Memory Usage (in Kilobytes):
2 Uptime: 162481918 Realtime: 422168600
3
4 *MEMINFO in pid 2368 [ericsson.com.calvin.calvin_constrained]*
5
6           Pss   Private   Private   SwapPss   Heap
7           Total   Dirty     Clean     Dirty     Free
8
9   Native Heap    4245      4200        0         0     1536
10  Dalvik Heap    2257      2188        0         0     3415
11  Dalvik Other    468       384         0         2
12     Stack       158       156         0         0
13     Ashmem        2         0         0         0
14     Gfx dev     1678     1144         0         0
15   Other dev        4         0         4         0
16   .so mmap     3342      152     2196     77
17   .apk mmap      473         0         96         0
18   .ttf mmap     107         0         36         0
19   .dex mmap    1372         4     1368         0
20   .oat mmap    5018         0     3228         0
21   .art mmap    1847      736         312        30
22  Other mmap     119         4         76         1
23  EGL mtrack   25920    25920         0         0
24   GL mtrack    4532     4532         0         0
25     Unknown     469      464         0         6
26     TOTAL    52127    39884     7316     116     4951

```

Listing 6.1: Example output from the `meminfo` tool when executed on the Calvin Constrained process.

In the output from the `meminfo` tool there are two columns for the private memory and one column for the PSS, the proportional set size (PSS) memory, which is the memory shared with other processes. For application analysis it is however only the **private dirty** memory which is important since this is the memory that is being used only by the application process and is returned to the operating system when the application terminates.

The private dirty memory is divided into the *native* memory and the *Dalvik* memory (the tool presents it as the Dalvik memory but the actual JVM is ART). The native memory are heap allocations made by native code such as from calls to `malloc` in the Calvin Constrained code. The Dalvik memory allocations are memory allocated on the heap in Java. Therefore this analysis focuses on the private dirty native heap and the private dirty Dalvik heap.

To analyze the heap data a small monitor program was created. The program samples the memory for the application on a given time interval. Crucial points in the code where the memory would like to be analyzed was also used for sampling the RAM. The data was stored on the phone and used for plotting the heap usage in different scenarios. The result is presented in chapter 6.3.

The Android API methods used to access the `dumpsys meminfo` tool allocated a small amount of memory on the Java heap to do the actual measurements. Even though this is a fairly small amount of data it affects the measurements of the Java heap usage. The tool does not allow the different objects on the heap be analyzed. The `dumpsys meminfo` tool can for example not distinguish between memory allocated for the Calvin runtime and the memory allocated for the UI.

The Android platform allows for Java heap dumps to be performed by using the heap dump tool. The tool dumps the heap to a binary format called `hprof`. The tool was used to analyze the Java heap at interesting moments in time and is presented in chapter 6.3.

6.1.2 Network Usage

The three sample applications described in section 6.2 were analyzed. The new transport method FCM was compared to the socket implementation. All tokens sent between the Calvin runtime on the Android device and the proxy runtime were logged. The tokens sizes were calculated and are presented for the three different applications in table 6.5.

6.2 Setup

All experiments used two runtimes, one running on a mobile device and one running on a laptop. The mobile device was a LG Nexus 5x with the specifications as described in table 6.1. For all experiments, all applications other than the Calvin runtime were terminated to minimize the effect of external sources.

Table 6.1: Table showing the mobile device's specification.

Model	LG Nexus 5x
Android	7.1.2
CPU	Qualcom MSM8992
RAM	2 Gb
Battery	2700 mAh

Three different Calvin applications were created in order to perform bench-

marks. The **connect** application has no application graph but is rather a test conducted to benchmark the transportation setup and the connect handshake between the CC runtime on the Android device and the Calvin base proxy runtime on the laptop.

The **identity** application consisted of three actors where the Identity actor was running on the mobile device. The application was deployed on the laptop runtime and the Identity actor was migrated to the mobile device manually. The Calvin script is shown in listing 6.2. The application was stopped after five triggers from the **CountTimer** actor.

```

1 /* Actors */
2 src : std.CountTimer(sleep=5)
3 echo : std.Identity()
4 snk : io.Print()
5
6 /* Connections */
7 src.integer > echo.token
8 echo.token > snk.token

```

Listing 6.2: Calvin script for the identity application

The **button** application consisted of two actors and is quite similar to the identity application. The application was deployed on the laptop runtime and the Button actor was migrated to the mobile device manually. The Calvin script implementing the button was implemented in a separate standalone application that was started and bound to the Calvin runtime before the button actor was migrated. All tests with the button application consisted of setting the system up and pressing the virtual button three times. The Calvin script for the button application is shown in listing 6.3.

```

1 /* Actors */
2 button : io.Button()
3 snk : io.Print()
4
5 /* Connections */
6 button.trigger > snk.token

```

Listing 6.3: Calvin script for the button application

6.3 Result

6.3.1 Heap Analysis

The native private dirty heap were sampled and plotted for the identity application in figure 6.3.

The allocations made on the Java heap were sample at the same events as specified in figure 6.3. The heap allocations were filtered and only the allocations made in the application Java package `ericsson.com.calvin.calvin_constrained` were considered. The result is show in table 6.2 together with the native private heap allocations at the same events in time.

Connect

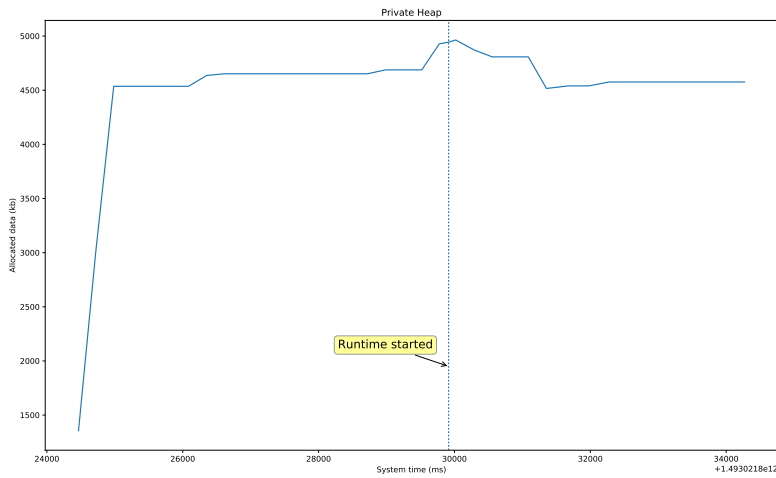


Figure 6.1: Native heap allocations over time when running the connect test.

Table 6.2: Table showing native allocations and Java allocations made within the Calvin Java package when running the connect application. The measured Native heap is the private dirty allocations and the CC Java heap are allocations made in the Calvin Java package.

Event	CC Java (bytes)	Native heap (bytes)
Runtime started	4149	4944k

Identity

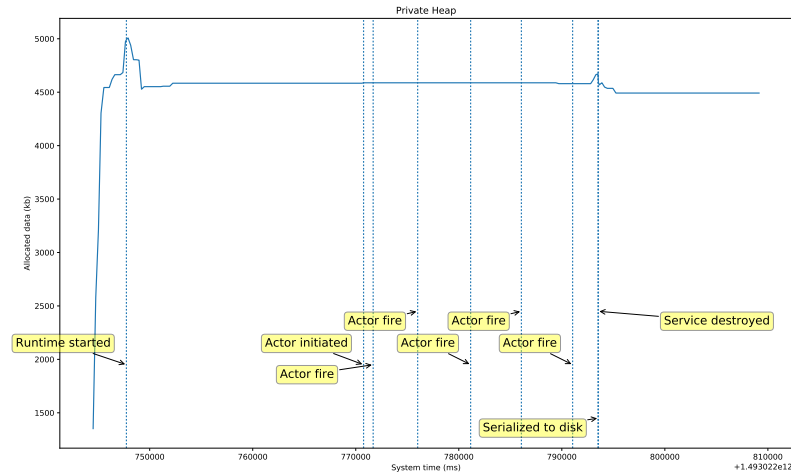


Figure 6.2: Native heap allocations over time when running the connect test.

Table 6.3: Table showing native allocations and Java allocations made within the Calvin Java package. The measured Native heap are the private dirty allocations and the CC Java heap are allocations made in the Calvin Java package.

Event	CC Java (bytes)	Native heap (bytes)
Runtime started	4713	4992k
Actor initiated	4149	4588k
Actor fire 1	4149	4588k
Actor fire 2	4149	4588k
Actor fire 3	4149	4588k
Actor fire 4	4149	4588k
Actor fire 5	4149	4588k
Service being destroyed	5583	4672k
Serialized to disk	6451	4584k
Average	4627	4642k

Button

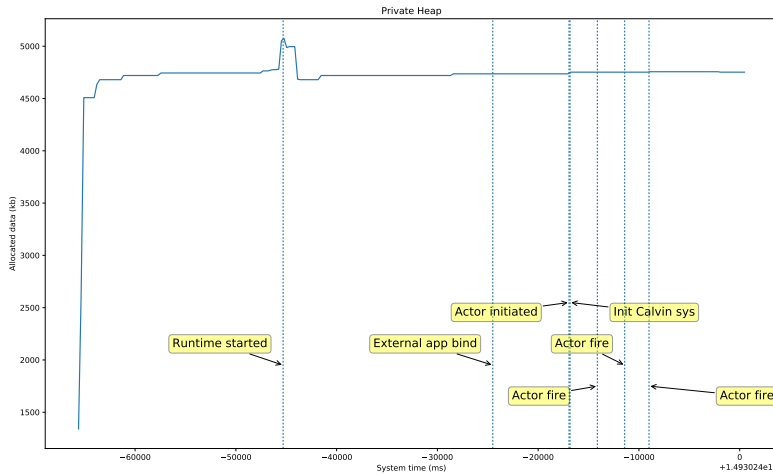


Figure 6.3: Native heap allocations over time when running the connect test.

Table 6.4: Table showing native allocations and Java allocations made within the Calvin Java package. The measured Native heap are the private dirty allocations and the CC Java heap are allocations made in the Calvin Java package.

Event	CC Java (bytes)	Native heap (bytes)
Runtime started	4149	5068k
External app bind	4089	4736k
Actor initiated	4547	4744k
Init Calvinsys	4013	4744k
Actor fire 1	3675	4752k
Actor fire 2	3675	4752k
Actor fire 3	4547	4756k
Average	3526	4793k

6.3.2 Network Usage

The network usage was captured on the proxy runtime end of the the network. All tokens sent when running the test applications were logged and their size were captured. All applications were executed using both the socket and the FCM implementation as transportation methods. The total number of bytes sent to the proxy runtime and from the proxy runtime are presented in table 6.5 together

with the size of the FCM SET_CONNECT handshake procedure. Note that the FCM handshake token is not needed when using the socket transportation method since it is the underlying socket that establishes the connection.

Figure 6.4 and 6.4 show the connect stage of the CC runtime with the proxy runtime when using socket and FCM for transportation. The figures show the data packets sent over time and their size.

Table 6.5: Table showing the average total memory usage of common applications over a time period of one hour. 174 tokens were sent in total when running the seven tests.

Test	Method	To CC (b)	From CC (b)	Total (b)
FCM Set Connect	FCM	382	320	702
Connect	Socket	635	716	1351
Connect	FCM	2300	2222	4522
Identity	Socket	9121	7642	16763
Identity	FCM	28092	24266	52358
Button	Socket	4739	4031	8770
Button	FCM	10978	9537	20515

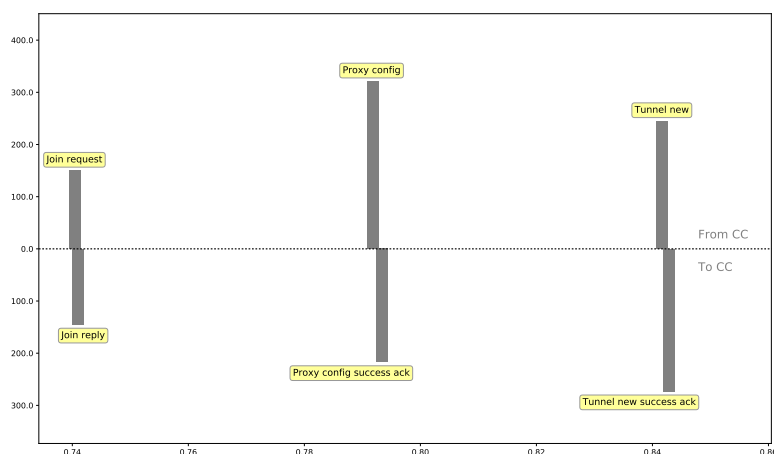


Figure 6.4: Bar chart showing the connect stage when the CC runtime connects to the proxy runtime over socket.

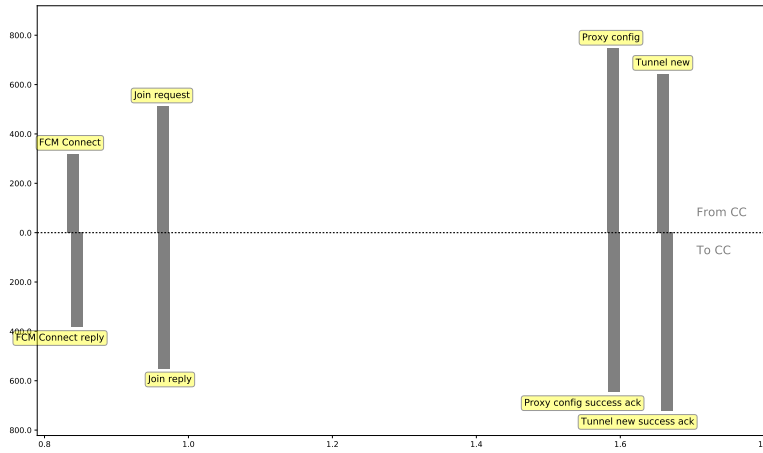


Figure 6.5: Bar chart showing the connect stage when the CC runtime connects to the proxy runtime over FCM.

6.4 Evaluation

6.4.1 Comments on Heap Analysis

It is hard to measure the complex heap structure of an Android application and even harder to interpret what the resulting data from the different tools mean. The focus of the tests for the heap analysis was to investigate the effect of the added layer of Java needed in order to run Calvin under an Android application. In table 6.2 6.3 6.4 the allocations made within the actual Calvin Java layer is compared to the total number of native allocations for the complete Android application.

As can be seen in all the tables the allocations made in the Java layer is a factor 1000 less than the total number of bytes allocated in total natively. This means that the allocations made for Calvin in Java do not affect the total heap usage much at all. If one were to optimize the heap usage, the Java layer is not where to start. The large native heap allocations mainly comes from Android framework code. That is code that is needed to actually run the Android application and maintains its lifecycle. Native allocations are also made for the UI components in the simple configuration UI.

In [43] the native heap allocations in Calvin Constrained running on a x86 system were analyzed and presented. The analysis shows that the native heap allocations made when migrating the Identity actor to a CC runtime is $8.9kb$. This compared to $4588kb$ seen in table 6.3 is very small. It is clear that the total allocations made for the Calvin application on Android is dominated by the allocations made for Android itself. The experiments presents the heap usage of very small applications with very simple actors. It should be noted that the

total heap usage of course is affected by the application running. If an actor for example generates graphical components using OpenGL, the heap usage would be much larger. The experiments however show that the bare minimum heap usage of the Calvin application is very small.

It is clear from the heap analysis that it is not the Calvin runtime implementation in the Android app that uses the most memory, but everything around it such as the simple UI. Android requires much more heap memory to initiate and run a bare Calvin Service and the heap usage of the actual Calvin runtime is negligible.

6.4.2 Comments on FCM Protocol Analysis

It is clear that the FCM protocol adds a large cost in terms of the size of the data sent compared to when using sockets for transportation when looking at the results in table 6.5. The overhead however comes with added functionality in the protocol such as the ability to wake up a device and use the mobile device anywhere as long as it is connected to the Internet.

There are two layers of overhead when using the FCM transportation compared to the socket implementation as can be seen in figure 5.3. The Calvin FCM protocol requires the payload data that sent over FCM to be Base64 encoded. This is needed since the CC runtime uses MessagePack instead of JSON to code all incoming and outgoing data packets. The overhead from only the Base64 encoding can easily be calculated. As described in section 2.5.2 Base64 encodes every three bits into four bits. There may also in the worst case be an additional three bits for padding. That means that the Base64 at least increases the size of the payload with a factor of $\frac{4}{3}$.

The Calvin FCM protocol needs to know what type of message is being sent over. Messages may either be Calvin payload messages or messages to set up the FCM connection which adds a static size of overhead to all messages sent. Furthermore the FCM protocol requires a JSON structure around every message sent which is wrapped in an XML tag. This is also data that needs to be added to every message sent over FCM. Since many messages sent are small compared to the size of the overhead, the ratio between the Calvin payload size and the FCM over head becomes large in many cases.

The FCM protocol has a different way of setting up the connection compared to when using sockets for communications. Figure 6.4 and 6.5 show the connect procedure when using sockets and FCM. As can be seen the socket implementation requires the extra two datapackets `SET_CONNECT` to be sent. As can be seen in table 6.5 the two packets have a total data size of 702 bytes which also is a static overhead for the FCM protocol.

Tests were conducted in order to show the average overhead size of all messages. In total 174 data packets were sent in the seven tests. The total number of bytes sent in both directions over the socket connection were $1351 + 16763 + 8770 = 26884b$ while the total size of the data sent over FCM was $4522 + 52358 + 20551 = 77431b$. If one deducts the effect of the FCM `SET_CONNECT` messages to only compare the sizes of the same data packets sent, the total number is $77431 - 702 * 3 = 75325b$. This means that the the total cost in incremented data size of using the FCM is a factor $\frac{75325}{26884} = 2.80$.

The cost of using FCM as a transportation method is quite large when it comes to data consumption. For very long term running applications this may be a problem since mobile devices often has a restriction on the data consumption per month from the carrier. The methods that change connectivity method as described in 5.4 is therefore very beneficial. FCM should only be used as a fall back solution and should be the last URI in the prioritized URI list. It should only be used when the mobile device is unable to connect using sockets, which it can if it is on the same LAN as the proxy runtime.

7.1 Mobile Devices in Calvin

Mobile devices come with a large set of sensors and capabilities and are very powerful small devices. Letting the mobile devices become a part of the distributed IoT platform Calvin enables for many different use cases. The proposed solution furthermore lets third party applications register themselves and lets Calvin take advantage of not only a mobile phone's sensors and hardware components, but also its software capabilities.

The limitations of Calvin having a constant link to the proxy runtime has been removed by taking advantage of Google Firebase Cloud Messaging. Calvin can be used anywhere in the world with FCM as long as the mobile device has Internet access which is needed in order to build usable IoT applications for mobile devices.

Long running applications on mobile devices must be implemented with a lot of care, and must not drain the battery or any other available resources. It has been shown that the RAM usage of the Calvin applications does not stick out among other common applications while the total heap usage is much higher than when running a stand alone implementation of Calvin Constrained. Mechanisms to let the application go into a sleep mode have been proposed to decrease battery and RAM. By using FCM the connected proxy runtime may wake the Calvin runtime on the mobile device to let it act on messages.

This thesis has had a very successful result and has shown that mobile devices can be part of the distributed platform Calvin and increase its usability. The thesis has shown how Calvin can make the best use of mobile devices and all of their capabilities in the world of IoT.

7.2 Future Work

With this thesis it is possible to build Calvin applications that make use of mobile devices. But there are areas that could be more investigated and desirable features to be implemented for mobile devices.

7.2.1 Dynamic Actors

To be able to run a Calvin application, all actors must have been implemented on the desired platforms. Actors in CC must be implemented and built when building the application. It is desirable to be able to dynamically load Actor implementations without the need of rebuilding the whole application.

This could be done either by letting third party applications implement Actors or by creating a service where Actors could be downloadable. This of course raises a lot of concerns regarding security.

7.2.2 Calvinsys for Calvin Base

The proposed Calvinsys implementation has been implemented for Calvin Constrained and Android but is yet not implemented for Calvin Base in Python. It is desirable that all Calvin implementations use the same structure of Calvinsys implementations. This would allow the Calvin Base Control API to be extended to allow dynamic loading of external Calvinsys implementations. For example this could allow a USB device register itself as a Calvinsys to allow actors to use the device.

7.2.3 Automated Testing

Calvin Constrained is currently tested frequently using an automated test suit. It would be desirable to also include tests for mobile device's functionality. This would require the tests to be run either on emulated Android devices or on physical hardware connected to the test server.

References

- [1] Ericsson, “Accelerating iot.” <https://www.ericsson.com/ourportfolio/telecom-operators/accelerating-iot?nav=marketcategory002>. Accessed: 2017-01-02.
- [2] “Smartphone os market share, 2016 q3.” <http://www.idc.com/promo/smartphone-market-share/os>. Accessed: 2017-03-24.
- [3] M. Zuckerberg, “The technology behind aquila.” <https://www.facebook.com/notes/mark-zuckerberg/the-technology-behind-aquila/10153916136506634/>, 2016. Accessed: 2017-03-23.
- [4] “Ericsson calvin base.” <https://github.com/EricssonResearch/calvin-base>.
- [5] Ericsson, “Ericsson research blog.” <https://www.ericsson.com/research-blog/>. Accessed: 2016-12-28.
- [6] O. Angelsmark, “A closer look at calvin.” <https://www.ericsson.com/ourportfolio/telecom-operators/accelerating-iot?nav=marketcategory002>, 2015. Accessed: 2017-01-03.
- [7] P. Persson and O. Angelsmark, “Calvin – merging cloud and iot,” *Procedia Computer Science*, vol. 52, pp. 210 – 217, 2015.
- [8] T. Nilsson, “Authorization aspects of the distributed dataflow-oriented iot framework calvin,” 2016. Student Paper.
- [9] N. Lindskog, “Consistent authentication in distributed networks,” 2016. Student Paper.
- [10] J. Broberg and P. Ståhl, “Dynamic fault tolerance and task scheduling in distributed systems,” 2016. Student Paper.
- [11] J. M. Roldan Gil, “Secure domain transition of calvin actors,” 2016. Student Paper.
- [12] J. Mejvik, “Cloud robotics for 5g,” 2016. Student Paper.
- [13] G. F. Coulouris, *Distributed Systems*. Pearson Education, 2011.
- [14] D. M. Petar Maymounkov, “Kademelia: A peer-to-peer information system based on the xor metric,” Accessed: 2017-04-04.

-
- [15] S. Tiago Boldt, “Dataflow programming concept, languages and applications,” *Faculty of Engineering, University of Porto*, 2012.
- [16] A. Carlsson, J. Eker, T. Olsson, and C. Von Platten, “Scalable parallelism using dataflow programming,” 2010.
- [17] “An overview of xmpp.” <https://xmpp.org/about/technology-overview.html>. Accessed: 2017-03-21.
- [18] “Introducing json.” <http://www.json.org/>. Accessed: 2017-03-16.
- [19] “Messagepack.” <http://msgpack.org>. Accessed: 2017-03-16.
- [20] “Messagepack specification.” <https://github.com/msgpack/msgpack/blob/master/spec.md>. Accessed: 2017-03-16.
- [21] “Base64 standard.” <https://www.rfc-editor.org/rfc/rfc3548.txt>. Accessed: 2017-03-16.
- [22] “The android sdk.” <https://developer.android.com/studio/index.html>. Accessed: 2017-03-21.
- [23] “The android ndk.” <https://developer.android.com/ndk/index.html>. Accessed: 2017-03-21.
- [24] “Android service.” <https://developer.android.com/guide/components/services.html>. Accessed: 2017-03-21.
- [25] “Processes and threads.” <https://developer.android.com/guide/components/processes-and-threads.html>. Accessed: 2017-03-21.
- [26] “Processes and application life cycle.” <https://developer.android.com/guide/topics/processes/process-lifecycle.html>. Accessed: 2017-03-22.
- [27] “Intents and intent filters.” <https://developer.android.com/guide/components/intent-filters.html>. Accessed: 2017-03-22.
- [28] “Connectivitymanager.” <https://developer.android.com/reference/android/net/ConnectivityManager.html>. Accessed: 2017-03-22.
- [29] “Security.” <https://source.android.com/security/>. Accessed: 2017-03-24.
- [30] “Requesting permissions.” <https://developer.android.com/guide/topics/permissions/requesting.html>. Accessed: 2017-03-24.
- [31] “Sensors overview.” https://developer.android.com/guide/topics/sensors/sensors_overview.html. Accessed: 2017-03-24.
- [32] P. Persson, “Open source release of iot app environment calvin.” <https://www.ericsson.com/research-blog/cloud/open-source-calvin/>, 2015. Accessed: 2017-03-10.
- [33] <https://github.com/EricssonResearch/calvin-base/wiki/CalvinScript>. Accessed: 2017-03-20.
- [34] <https://www.ericsson.com/research-blog/internet-of-things/want-calvin-available-many-devices-possible/>. Accessed: 2017-03-21.

-
- [35] <https://micropython.org/>. Accessed: 2017-03-21.
- [36] “Ericsson calvin constrained.” <https://github.com/EricssonResearch/calvin-constrained>.
- [37] “Things you should know about bluetooth range.” <http://blog.nordicsemi.com/getconnected/things-you-should-know-about-bluetooth-range>, 2016. Accessed: 2017-03-27.
- [38] “About firebase cloud messaging server.” <https://firebase.google.com/docs/cloud-messaging/server>. Accessed: 2017-03-22.
- [39] “Notifications.” <https://developer.apple.com/notifications/>. Accessed: 2017-03-27.
- [40] M. Kihl and J. Andersson, *Datakommunikation och nätverk*. Studentlitteratur, 2013.
- [41] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. 2005. Accessed via <https://lwn.net/Kernel/LDD3/> on 2017-04-11.
- [42] “Overview of android memory management.” <https://developer.android.com/topic/performance/memory-overview.html>. Accessed: 2017-04-14.
- [43] A. Mehta, F. Svensson, R. Baddour, H. Gustafsson, and E. Elmroth, “Calvin constrained - a framework for iot applications in heterogeneous environments,” 2017.

Example of the Calvin Constrained PROXY_CONFIG command

An example of the PROXY_CONFIG command and the data sent with it is shown below.

```
1 {
2   "attributes": {
3     "indexed_public": {
4       "node_name": {
5         "name": "Calvin Android"
6       }
7     }
8   },
9   "capabilities": [
10    "calvinsys.io.button",
11    "calvinsys.media.camerahandler"
12  ],
13  "cmd": "PROXY_CONFIG",
14  "from_rt_uuid": "c8ff08a3-b648-a23a-3afd-156e59222112",
15  "msg_uuid": "MSGID_83765d60-99cd-4453-c9a5-49b467aea25f",
16  "name": "Calvin Android",
17  "port_property_capability": "runtime.constrained.1",
18  "to_rt_uuid": "05225cc0-15a9-4f4e-9e9e-50b03d7a8d07"
19 }
```

Example of an XMPP authentication handshake

The following listings are an example of the authentication process in XMPP to a Googlew server [38].

```
1 <stream:stream to="gcm.googleapis.com"  
2   version="1.0" xmlns="jabber:client "  
3   xmlns:stream="http://etherx.jabber.org/streams">
```

Listing B.1: The first step of the handshake. The client tries to initiate a connection [38].

```
1 <stream:features>  
2   <mechanisms xmlns="urn:iETF:params:xml:ns:xmpp-sasl">  
3     <mechanism>X-OAUTH2</mechanism>  
4     <mechanism>X-GOOGLE-TOKEN</mechanism>  
5     <mechanism>PLAIN</mechanism>  
6   </mechanisms>  
7 </stream:features>
```

Listing B.2: The second step of the handshake. The server answers that the client needs to authenticate first and how this should be done [38].

```
1 <auth mechanism="PLAIN"  
2 xmlns="urn:iETF:params:xml:ns:xmpp-sasl">  
3   MTI2MjAwMzQ3OTMzQHByb2pY3RzLmdjbS5hb  
4   mFTeUIzcmNaTmtmbnFLZEZiOW1oekNCaVlwT1JEQTJKV1d0dw==</auth>
```

Listing B.3: The third step of the handshake. The client authenticates [38].

```
1 <success xmlns="urn:iETF:params:xml:ns:xmpp-sasl"/>
```

Listing B.4: The fourth step of the handshake. The server responds with the result of the authentication [38].

```
1 <stream:stream to="gcm.googleapis.com"
2   version="1.0" xmlns="jabber:client "
3   xmlns:stream="http://etherx.jabber.org/streams">
```

Listing B.5: The fifth step of the handshake. The server answers with the features of the server [38].

```
1 <iq type="result">
2   <bind xmlns="urn:ietf:params:xml:ns:xmpp-bind">
3     <jid>SENDER_ID@gcm.googleapis.com/RESOURCE</jid>
4   </bind>
5 </iq>
```

Listing B.6: The sixth and last step of the handshake. The server answers that the stream is set up [38].

Calvin Configuration for FCM

```
1 {  
2   "global": {  
3     "transports": ["calvinip", "calvinfcm"],  
4     "fcm_server_secret": "[SERVER KEY]"  
5   }  
6 }
```

Listing C.1: Example of how to configure the proxy runtime to use FCM.



LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2017-574

<http://www.eit.lth.se>