Master's Degree in Informatics Engineering
Dissertation
Final Report

Analysis and detection of anomalies in mobile devices

António Carlos Lagarto Cabral Bastos de Lima
aclima@student.dei.uc.pt

Supervisor:
Prof. Dr. Tiago Cruz

Co-Supervisor:
Prof. Dr. Paulo Simões

Date: September 1, 2017

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Master's Degree in Informatics Engineering
Dissertation
Final Report

Analysis and detection of anomalies in mobile devices

António Carlos Lagarto Cabral Bastos de Lima
aclima@student.dei.uc.pt

Supervisor:
Prof. Dr. Tiago Cruz

Co-Supervisor:
Prof. Dr. Paulo Simões

Date: September 1, 2017

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Acknowledgements

I strongly believe that both nature and nurture playing an equal part in shaping an individual, and that in the end, it is what you do with the gift of life that determines who you are. However, in order to achieve great things motivation alone might just not cut it, and that's where surrounding yourself with people that want to watch you succeed and better yourself comes in. It makes the trip easier and more enjoyable, and there is a plethora of people that I want to acknowledge for coming this far.

First of all, I'd like to thank professor Tiago Cruz for giving me the support, motivation and resources to work on this project. The idea itself started over one of our then semi-regular morning coffee conversations and from there it developed into a full-fledged concept quickly. But this acknowledgement doesn't start there, it dates a few years back when I first had the pleasure of having him as my teacher in one of the introductory courses. It didn't take long for all of us, his students, to notice that he was different from most other teachers as he actively promoted self-work, complementary learning and didn't condemn our then limited work in a binary fashion. He encouraged and inspired us to be better programmers, sure, but also better people. He didn't look down on us from high above on his golden pedestal; no, he joined us on the same level and helped with the nitty-gritty code, bugs and logic flaws. Don't get me wrong, there were other fine teachers during these past few years, but in my most honest opinion, none compare to professor Tiago Cruz when it comes to nurturing one's desire to better themselves while still having fun.

The next person I'd like to thank is someone who has been with me through good and bad, thick and thin, for almost 5 years. I'm talking of course of my girlfriend Inês Petronilho, who also inspires me on a daily basis to be a better person, to always strive for higher heights, and who unconditionally supports me. I love you.

Following up, my best mate Pedro Janeiro. We've coded, laughed, drinked, cooked, eaten, pulled all-nighters, and we've even slept together! There were times when I think I spent more time with you on a daily basis than with Inês, and I wouldn't have had it any other way.

Speaking of noteworthy friendships, ever since our first semesters at DEI, I've had the pleasure of having some of the most entertaining moments and the most pointlessly philosophical debates with a very special group of friends. We motivated and picked each other up, commuted long distances, helped each other out with the courses' projects, had our lunches together, and had the best LAN parties Coimbra has probably ever seen. A big

thank you to the guys, it is wonderful that after all these years we still regularly reunite over some tasty sushi and catch up on our lives.

For the past year I've spent a lot of time over at the Nest Collective here in Coimbra, and they have become like a second family to me, particularly the people at Deemaze. I've learnt a lot from all of the people over there and made friendships aplenty. You guys are party animals, and every day and night spent with you lot is one to remember fondly, every time.

Of course my family and relatives have also played a very big part in shaping the person I am today, they are the nurture aspect of this story, and I wouldn't have gotten here today if it weren't for them.

Lastly, and coming back to the initial portion of this section, I'd like to thank everyone that I have encountered up to this point, part of what I am and will be is due to you.

Abstract

Organizations are often faced with the need to manage large numbers of mobile device assets, including tight control over aspects such as usage profiles, customization, applications and security. Moreover, the rise of the Bring Your Own Device (BYOD) paradigm has further contributed to hamper these requirements, making it difficult to strike a balance between corporate regulations and freedom of usage.

In this scope, security is one of the main requirements both for individual and corporate usage. Device and information protection on mobile ecosystems is quite different from securing other assets such as laptops or desktops, due to specific characteristics and restrictions. For instance, the resource consumption overhead of security mechanisms, which is less relevant for desktop/laptop environments, is critical for mobile devices which frequently have less computing power and must keep power consumption as low as possible.

Security mechanisms for mobile devices combine preventive tools (e.g. Trusted Execution Environments and sandboxed applications), monitoring solutions and reactive and mitigation techniques. In this thesis we start by overviewing these security solutions, presenting a survey on the technologies, frameworks and use cases for mobile device security monitoring and management, with an emphasis on the associated open challenges and benefits, from both the end-user and the corporate points-of-view.

Having analyzed the technological state of the art, we showcase our attempt at analyzing and detecting anomalies in mobile devices on an enterprise scenario, the contemplated and solved implementation ordeals, and the employed development details to achieve it. The described system is comprised of: an Android application, intended to be installed on the target devices; lightweight Message Brokers; a Central Aggregator, serving as the core of the system, processing and managing the collected data from the mobile assets; a Monitoring Dashboard, enabling the system to be altered at runtime by supervising humans.

Lastly, we evaluate the project, exhibiting the preliminary results obtained through the developed system, examining the implications that the results warrant, assessing the current state of the project's proposed tasks and requirements, and proposing the course of action for future work.

Keywords

Anomaly, detection, malware, management, mitigation, mobile devices, monitoring, prevention, security

Acronyms and Abbreviations

AM - Application Master
API - Application Programming Interface
BYOD - Bring Your Own Device
CPU - Central Processing Unit
DAG - Directed Acyclic Graph
GPS - Global Positioning System
HDFS - Hadoop Distributed File System
HTTP - Hypertext Transfer Protocol
IoT - Internet of Things
JSON - JavaScript Object Notation
M2M - Machine-to-Machine
MDM - Mobile Device Management
MQTT - Message Queue Telemetry Transport
NM - Node Manager
OODA - Observe, Orient, Decide, and Act
OS - Operating System
RAM - Random Access Memory
REST - Representational State Transfer
RM - Resource Manager
SSL - Secure Sockets Layer
TEA - Tiny/Trivial Encryption Algorithm
TLS - Transport Layer Security
UML - Unified Modeling Language
VPN - Virtual Private Network
YARN - Yet Another Resource Negotiator

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Although they are called "mobile", many mobile devices themselves cannot move but rather do not hinder the movement of the carrier or host, as opposed to what is are now commonly known as stationary devices. Because of this distinction, mobile devices can be characterized per several other traits: dimensions and weight, connectivity to other devices, and when the mobility occurs.

Most mobile devices have an embedded display screen (or can be connected to one) and receive input from either physical buttons and keyboards or from on-screen virtual buttons and keyboards (if the display is touch-sensitive), some devices also support audio input, namely voice recognition. Sensors such as accelerometers, compasses, magnetometers, or gyroscopes, allowing for detection of orientation and motion, data capture devices such as barcode, RFID, fingerprint and smart card readers can also be embedded or attached to the mobile device. On top of all the already mentioned extras, the most prominently used features are related to connectivity and usually fall under Wi-Fi, Bluetooth, NFC and GPS capabilities.

The first mobile devices did not have much to offer, devices like the IBM Simon (released in 1994, regarded as the first smartphone) were very "dumb" by today's standards, having very limited electronics and processing power. For the sake of context, the Table 1.1 compares mobile devices (almost) equidistant in time: an iPhone 6, an iPhone 1st Generation (commonly referred to as the first actual smartphone) and the IBM Simon.

Table 1.1: Comparison of iconic mobile device (smartphone) specifications

	IBM Simon	**iPhone (1^{st} Gen.)**	**iPhone 6**
Release Date	August 16, 1994	June 29, 2007	September 19, 2014
Dimensions	200 x 64 x 38 mm	115 x 61 x 11.6 mm	138.1 x 67 x 6.9 mm
Weight	510 g	135 g	129 g
CPU	16-bit, 16 MHz, x86-compatible	32-bit, 412 to 620 MHz, ARM	64-bit, 1.4 GHz dual-core, ARM
GPU	None	PowerVR MBX Lite 3D GPU	PowerVR Series 6 GX6450 (quad-core)
Memory	1 MB	128 MB DRAM	1 GB LPDDR3 RAM
Storage	1 MB	4, 8, or 16 GB	16, 64, 128 GB
Battery	7.5V NiCad	3.7 V 1400 mA·h Lithium-ion	3.82 V 6.91 W·h (1,810 mA·h) Lithium-ion
Data Inputs	Microphone, Touchscreen with stylus	Multi-touch touchscreen, 3-axis accelerometer, Proximity sensor, Ambient light sensor, Microphone	Multi-touch touchscreen, Triple microphone, 3-axis gyroscope, 3-axis accelerometer, Digital compass, iBeacon, Proximity sensor, Ambient light sensor, Fingerprint reader, Barometer
Display	4.5 in × 1.4 in, 160px x 293px monochrome backlit LCD	3.5 in diagonal, 320x480 px resolution at 163 ppi, 2:3 aspect ratio, 18-bit (262, 144-color) LCD	4.7 in diagonal, 1334x750, LED-backlit IPS LCD, 326 ppi (128 px/cm) pixel density 16:9 aspect ratio
Camera	None	Rear: 2 MP	Rear: 8 MP, 1080p HD video recording, Slow-motion video, Panorama (up to 43 MP) Front: 1.2 MP (1280×960 px max.), 720p video recording
Connectivity	2400-bps Hayes-compatible modem, 33-pin connector, 9600-bps Group 3 send-and-receive fax, I/O connection port, PCMCIA type 2	Quad-band GSM/GPRS/EDGE, Wi-Fi (802.11 b/g), Bluetooth 2.0	UMTS/HSPA+/DC-HSDPA, CDMA EV-DO Rev. A and Rev. B, GSM/EDGE, Wi-Fi, Bluetooth 4.2, NFC, GPS & GLONASS

As one can see from Table 1.1, mobile devices such as smartphones have come a long way, having their computing power and capabilities increase exponentially. Because of the plethora of possibilities these more modern mobile devices offer, it is no wonder that they have spread to almost every job that is associated with mobility and have even begun to substitute previously dominant (and often unportable) forms of computing devices such as desktop computers.

The surge of more capable mobile devices has also created the need for what are now known as Bring Your Own Device (BYOD) environments. There are several types of BYOD schemes available for enterprise usage, with the key distinctive factor being Mobile Device Management (MDM) policies, detailing which devices are admissible and what type of control is held over them (accessible data, applications and functionalities). Figure 1.1 illustrates the trade-off between control and number of devices in the environment

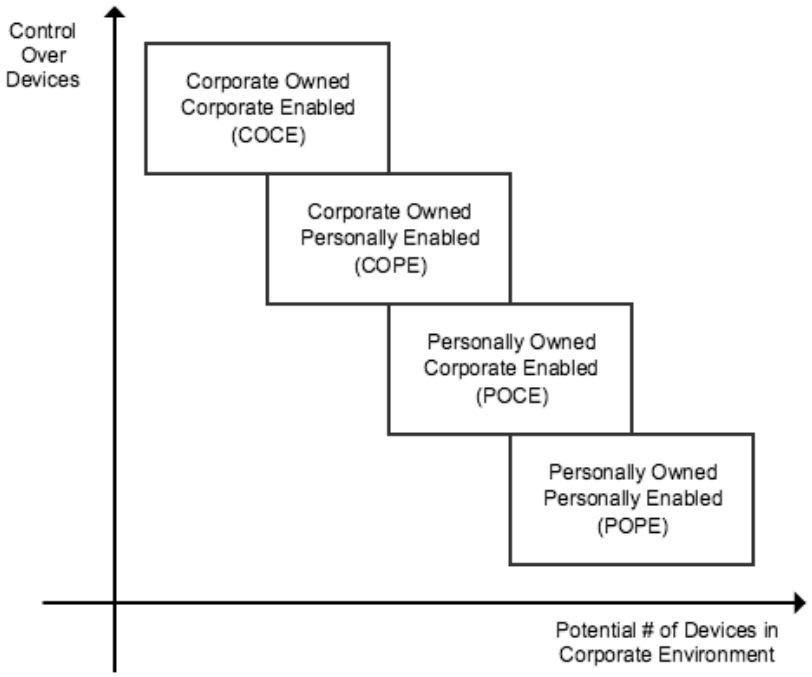


Figure 1.1: Enterprise mobility schemes

Recently, large scale companies have begun to move from the left most scheme (COCE) in Figure 1.1 to more BYOD friendly alternatives, namely COPE and POCE.

Although under some circumstances one can be exchanged for the other, several defining features separate stationary from mobile devices which in turn lead to cut corners when it comes to device security. The most noteworthy distinction is the limited battery life inherent to truly mobile devices as it is also the root of most of other differences. Since the battery is finite (unless recharged) the devices computing power can only be pushed thus far, to the point where its timed usage does not consume too much energy. Mobile devices' com-

pactness also poses concerns when it comes to cooling or increasing storage space. Another, often treated separately, concern is the risk of misplacement or theft of mobile devices which grants a malicious user the freedom for a completely different suite of attacks or paths to the data stored in the device. As it should be obvious, mobile devices also have their beneficial qualities, namely their inherent eased mobility or transportation from one location to another and, their simplified interfaces and minimal physical appendages for ease of use. With such accentuated differences, it is only inevitable that security assurance in mobile devices faces a different set of challenges from conventional computers' security risks and defense strategies, working with more limited resources.

During the course of this project, a paper (Lima 2016) was published for the ICCWS (International Conference on Cyber Warfare) conference, surveying the state of the art on security for mobile device assets, analyzing and comparing various security methods and techniques from supported operating systems to the threats or attack vectors involved. Much of the material in that paper was used for Chapter 2.

This project aims to develop an alternative method of analyzing and detecting anomalies in mobile devices, not unlike what has already been done for stationary and laptop computers, but which still seems illusive or not well established in mobile device contexts.

This document focuses on surveying mobile device management solutions for security problems, ranging from prevention, to monitoring and mitigation of threats at various points in the system's and its applications' lifecycles, in hopes of identifying the best and worst practices for enterprise environments. Additionally, it presents a system with several components designed for the analysis and detection of anomalies on mobile devices. It is organized in the following manner. **Chapter 2** delves into the aspects of Prevention and Monitoring in mobile devices, overviewing the most common used methods of each strategy, their benefits and shortcomings, and comparing them amongst themselves. It also examines mitigation techniques employed in mobile devices. **Chapter 3** focuses on the objectives and scope of this project, detailing its requirements. **Chapter 4** provides both an overview and a more specific analysis of the work and work methodology planned for this project.

Starting at **Chapter 5**, a more technical analysis of the system is provided. This chapter is dedicated to the system's architecture, its components, their connections, relationships and interactions. In **Chapter 6** we describe implementation problems that arose during the course of the project, how we confronted them as engineering tasks, and explain the solutions we propose for them. Following up, and in order to leave no essential details out of this document, **Chapter 7** acts as a user guide explaining any noteworthy considerations that were made during the development phases, how the system's components can be managed individually, and how they should be deployed.

Once the system has been presented from one end to the other, **Chapter 8** overviews the preliminary data results that have been collected, assessing the efficiency and promise of the proposed solutions. **Chapter 9** assesses the current completeness of the project, in light of what was initially proposed, and lists what still is missing and should be done in

the future. Finally, **Chapter 10** is dedicated to final remarks and conclusions related to the project.

Chapter 2

State of the Art

This chapter details the list of current and historically relevant techniques used for analyzing and detecting anomalies in mobile devices, breaking them into three main categories: prevention, monitoring, and mitigation. For each of these categories there is a plethora of subcategories and techniques which we explain superficially and provide examples for. Lastly, we analyze already existing competitor applications for some (if not all) of the same goals as our proposed application, which we detail later on, and why they aren't suitable for the problem at hand.

2.1 Prevention and Monitoring

Currently there is a very apparent emphasis on mobile device data security through preventive methods when compared with monitoring or detection approaches. The most often cited cause for this existing discrepancy is the limited battery life of mobile device assets, without a doubt, but it is also coupled with the more, and often forgotten, less intrusive nature of prevention methods. The following sections detail the most common prevention and monitoring methods found in mobile devices along with their respective advantages and disadvantages. The takeaway message from this section is that albeit both defensive strategies can be effective on their own, they are not mutually exclusive and in fact may benefit vastly (in terms of effectiveness) from being combined.

2.1.1 Prevention

Several prevention methods have been adopted in mobile devices to minimize the amount of potentially harmful malware applications that get into the device or that might exploit known attack vectors as a precautionary measure. An attack vector is a method or exploited security feature used to compromise the security of a system. Among the most commonly used prevention methods are:

- **Sandboxing** – creating a separate virtual space for the untrusted application or code to run with limited to no interaction with other apps and under tight restrictions in the underlying host system, thus diminishing the possible attack vector. Most modern

mobile device Operating Systems (OSs) such as Android, iOS and Windows Phone run their applications in sandboxed environments.

- **Personalized Approval** – detailed analysis of each application's conformity to security policies before being approved for distribution. The most common example of this technique is that of Apple's App Review system wherein each submitted application goes through a series of meticulous review steps before being available for download on the App Store. Microsoft has a similar system in place the Windows Phone App Store. By contrast, Google does not have a similar policy for apps delivered to the Google Play Store, relying on its users' capacity to identify and report malicious applications or on Trusted Party Management (which is covered later).

- **Code and Assets Assessment** – there are several well-known types of malware which can be identified by distinct signatures. Most anti-virus protection mechanisms rely on databases of such cases to identify potential threats to the system. Another way of assessing an app is to check which system calls, frameworks and methods it uses and for what reasons. Both iOS and Windows Phone enforce this behavior through their respective official application stores, for which each submitted application undergoes scrutinous inspection before being available for download.

- **Compartmentalization** – the basic premise is that of completely separating the user's personal space from the work space within the device, aiming to avoid unnecessary compromise and encapsulate threats from the personal space. The work space is usually subject to much stricter security and behavior restrictions and might even not allow the installation or running of new scripts apart from the default assets. Since Android API level 21, devices can have user accounts that are managed by the main account, which can have a more limited set of permissions.

- **Trusted Party Management** – this method relies on the verification of trusted certificates, keys and signatures to manage which entities can interact with the system or which applications can be installed. For instance, only a single trusted computer might be able to write to the mobile devices storage (aside from the device itself) or only apps approved and properly signed by a company beforehand might be installed on the device. Android, iOS and Windows Phone all have the capability to install and manage custom trusted enterprise applications in the devices, be it through direct installation or certified third party application stores.

- **Permission Systems** – one way of circumventing the ordeal of assessing an app's functionalities or used tools is to leave it to the user to either accept or reject a list of granted permissions. Android and iPhone apps, for instance, must explicitly detail which system features or hardware capabilities they (might) need access to. For Android devices, this is a mandatory step performed before the app is installed unto the device whilst for iPhone devices each individual permission will be asked during runtime when needed for the first time. In both scenarios, this information can be checked before installing the app and it is up to the user to gauge the risk granting access to sensitive features such as access to the camera or SD storage.

- **Authentication Schemes** – comprised of heavy encryption standards, with the possibility of depending on the existence of an external super user, the idea is to lock system management behind complex password/PIN mechanisms before any (or even every) write or read action is performed on the system. Most modern mobile Operating Systems such as Android, iOS and Windows Phone come with the option to encrypt the phone regarding a locking mechanism, usually a PIN code or a fingerprint, without which the contents of the phone cannot be accessed.

- **Limited Time Access** – ephemeral fixed length sessions are scheduled for the devices usage, and all services are interrupted or aborted when the allocated time slot elapses. Not available by default, this method is usually achieved at an application level by using timed access tokens

These prevention strategies are not free of advantages and disadvantages. Most of them try to delegate the risk assessment to a more powerful or infinite power-sourced entity (human or server) reducing the energy footprint on the target device. They benefit from vast amounts of previous knowledge, be it through databases of well know signatures or attack vectors, or even through sheer human suspicion of years of trial and error. However, as can be seen, most of these methods are fallible for the exact same reasons. Previously unknown attack vectors or methods could simply pass by these security precautions because they are not suspicious when compared with the knowledge base. Leaving it up to end user to figure out if the list of permissions is unacceptable or a potential hazard relies too much on common-sense and is subjective to everyone's aptitudes and skills. Perhaps the most common subliminal pitfall of such methods is the intrusive steps required for the installation of content.

Most of these preventive methods have been directly ported or adapted from already well-established and tested stationary computers.

Table 2.1 analyses at which moment of an application's lifecycle the already mentioned prevention strategies act, and for which mobile Operating Systems they can be implemented or are already available.

As we can see from Table 2.1, many prevention strategies occur at runtime. This is the opposite of what is commonly done on stationary devices such as desktop computers, mainly due to the fact that it can be harder to find suspicious applications solely based on their appearance rather than on their behavior without the constant monitoring provided by anti-viruses or similar tools, thus, mobile devices rely on runtime approaches.

Table 2.1: Comparison of key moment of action for prevention methods

	Pre-Installation	**Installation**	**Runtime**	**OS Availability**
Sandboxing			•	Android, iOS, Windows Phone
Personalized Approval	•			iOS, Windows Phone
Code and Assets Assessment	•			iOS, Windows Phone
Compartmentalization			•	Android
Trusted Party Management		•		Android, iOS, Windows Phone
Permission Systems		•		Android, iOS, Windows Phone
Authentication Schemes			•	Android, iOS, Windows Phone
Limited Time Access			•	Android, iOS, Windows Phone

2.1.2 Monitoring

Monitoring faces a different set of challenges from the prevention mechanisms already described as it is intended to facilitate detection and (if possible) gather additional information to ease halting malicious activity within the system at runtime. The most common approaches are:

- **Behavior Analysis** – focused on what is and isn't expected of each type of application, the app or code is scrutinized at runtime for any anomalous or suspicious activity with the main purpose of detecting unwanted accesses or transfers of delicate or out of scope information in the device. This process can be coupled with a training phase or expected default behavior thresholds for known applications, or with Log Analysis, the next monitoring method on this list. (Bose 2008, Burguera 2011)

- **Log Analysis** – the host system logs systematic information regarding memory and battery usage, storage accesses, data transfers and services used (Bluetooth, Wi-Fi, Mobile Data, etc.) in hopes of portraying the apps true nature. The logs are then periodically sent to a trusted supervising server to delegate the heavier, more resource intensive, analysis burden out of the mobile device, albeit some minor analysis can still be performed on site. Log integrity measures should be considered on both endpoints for this method to work properly. (Becher 2008, Mazumdar 2011, Halilovic 2012)

- **System Call Hooking (SCH)** – the act of altering the underlying host system for more fine-grained logging of the system's function calls and frameworks used (along with other features) with the goal of easing the monitoring task (Willems 2007). This can help detect unnecessary use of mobile device features such as the camera or microphone.

- **Runtime Permission Checks** – every hardware and software feature external to the app requires that explicit permission for said action be granted at runtime at the host system's level. One common example would be requesting permission to use the device's camera every time to stop a malicious app from stealthily collecting unwanted pictures of the user or his surroundings. Android and iOS have runtime permission verifications, allowing for the user to toggle or opt out permissions after installing the application.

The main advantages of monitoring security strategies are a better and more refined assessment of the sensitive data and what actions or applications could be compromising it (hopefully) in time for intervention. They also enable a stricter control of what are acceptable behaviors through a more detailed logs and variable limitations. Given the more limited computing power and battery life, some of these methods can easily become an overkill for the device or rely too heavily on consistent and systematic connection with secure networks and servers, which might not be a possibility all together depending on the use case scenario. Another disadvantage is the risk margin for potential interventions in "real time" which, depending on network status, system resources, cryptographic complexity and connectivity issues might result in an unacceptable latency.

It is worth mentioning that some monitoring methods cannot be implemented out of the box, the perfect example being System Call Hooking, which is very hard to perform or even impossible on most mobile operating systems without rooting or running a custom modified version of the original OS. Android is the most commonly adopted OS for these kind of barred monitoring techniques because it is based on the Android Open Source Project, meaning that it can be altered at core level to accommodate the desired changes more easily, and it can run on a variety of devices and hardware, unlike iOS and Windows Phone.

Table 2.2: Comparison of prevented threats by monitoring method

	Privilege Escalation	**Worms, Trojans, Viruses**	**Resource or Data Misuse**	**Spyware**	**Theft, Device Cloning**	**OS Availability**
Behavior Analysis		•	•	•	•	Android, iOS, Windows Phone
Log Analysis		•	•	•	•	Android, iOS, Windows Phone
System Call Hooking	•	•	•	•		Android
Runtime Permission Checks			•			Android, iOS, Windows Phone

Table 2.2 examined which threats each monitoring method can help prevent, and it can clearly be seen that apart from Runtime Permission Checks, they all seems equally effective against most threats. However, since SCH is usually not readily available on most mobile devices, and Behavior Analysis requires a priori training, Log Analysis seems to be the most effective technique that can be employed out-of-the-box.

Although prevention and monitoring can improve security immensely, there will always be new malware or attack vectors against which they are ineffective. The best examples of this kind of malware are zero day, undisclosed and undetectable exploits, ranging from target specific applications such as Windows' Stuxnet to global pandemics such as Android's Stagefright. For these threats, mitigation is a suitable countermeasure.

2.2 Mitigation

Since traditional Desktop devices are mostly immovable, they are far less susceptible to physical attacks, and can be monitored much more easily, they can even be more reliant on safe connections since they are susceptible to few changes. Mobile devices, however, are the complete opposite and can commute through various different device ecosystems in a matter of minutes, hence, additional care must be taken when it comes to their security.

Some of the preventive and monitoring techniques already described in Section 2 can be considered attack vector or vulnerability mitigation techniques on their own, but there are other means to try and repair a compromised device, or to at least minimize the damage it can do:

- **Security Policies** – written documents that detail which devices can connect to an infrastructure, what types of data they should contain, what constitutes as sensitive data, a list of supported operating systems and applications, and terms of acceptable use and penalties for misuse.
- **Patch Support** – the ability to submit updates or patches for an application or system without having to physically access it can be sufficient to fix newly discovered vulnerabilities or potential attack vectors.
- **Long Term Support (LTS)** – after distributing the system/application there is a time frame for which the developer agrees to provide support for potential threats.
- **Remote Control** – if a device becomes compromised it can be wiped, reset, shut-down or locked remotely, removing the possibility for an escalated compromise or continued/recurrent access to sensitive data.
- **Location Tracking** – if a mobile device is stolen and ends up in a conspicuous location it can be tracked down or simply located after the theft is reported.
- **Full/Coordinated Disclosure** – if a researcher/developer finds a vulnerability in an application/system he should contact the developer about his findings and, if given permission (in case of coordinated disclosure), make them available as soon as possible, or simply make them public as soon as he is certain of them (in case of full disclosure). This mitigation strategy is controversial as disclosing vulnerabilities can in fact potentiate the discovery of related but stealthier vulnerabilities.
- **Recall for Analysis** – the potentially targeted devices are periodically recalled for a temporary system and application assessment. This method can be invasive to personal data if the device is also used for personal purposes or outside of the enterprise environment.
- **User Feedback** – a means for the user of the device to report any suspicious or anomalous behavior, no matter how ridiculous it might seem. This must be enforced with a sense of trust between the user and coordinators/feedback-receivers.

- **Documentation** – documents dedicated to scrutinize successful and unsuccessful aspects of a project, application or system, along with identified threats, how they were handled and how they could have been avoided. Usually performed at the end or at pre-defined iterative stages of the lifecycle, a Post Mortem document is a common example of this strategy.

Table 2.3 analyses the moment at which each of the addressed mitigation strategies can help handle a potential threat: before, during or after it is detected or does damage.

Table 2.3: Moment of mitigation comparison

	Before	During	After
Security Policies	•		
Patch Support	•		•
Long Term Support		•	•
Remote Control		•	•
Location Tracking			•
Full/Coordinated Disclosure			•
Recall for Analysis	•	•	•
User Feedback		•	
Documentation			•

As was expected from the definition of the word "mitigation" itself, there is great emphasis mitigating a threat after it has done damage more than the preventive and monitoring aspects that we have already discussed, which have more preemptive and persistent natures.

Good mitigation strategies are just as important in terms of informatics as they are when accounting for human resources. The more detailed a mitigation strategy and security countermeasures the easier it can become to overcome threats. Figure 2.1 exemplifies how all the identified preventive, monitoring and mitigation methods can be combined into a sturdier security lifecycle for mobile asset management, denouncing the three main phases at which malware can be identified and, possibly, prevented.

Figure 2.1 is reminiscent of a simpler Observe-Orient-Decide-Act (OODA) loop (Boyd, 1995), a common strategic approach to military and commercial operations and learning processes. It focuses on agility when dealing with a threat in order to mitigate its advances and damage at any given point in time, learning from any form of input and feedback obtained. Figure 2.2 portrays a simplified version of the Boyd's OODA loop, representing all of these aspects.

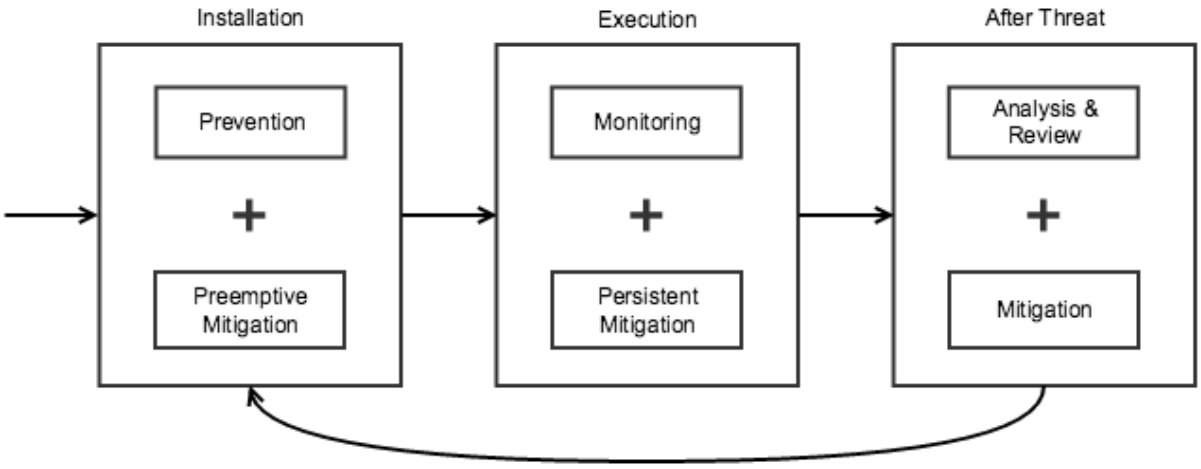


Figure 2.1: Security lifecycle for mobile asset management

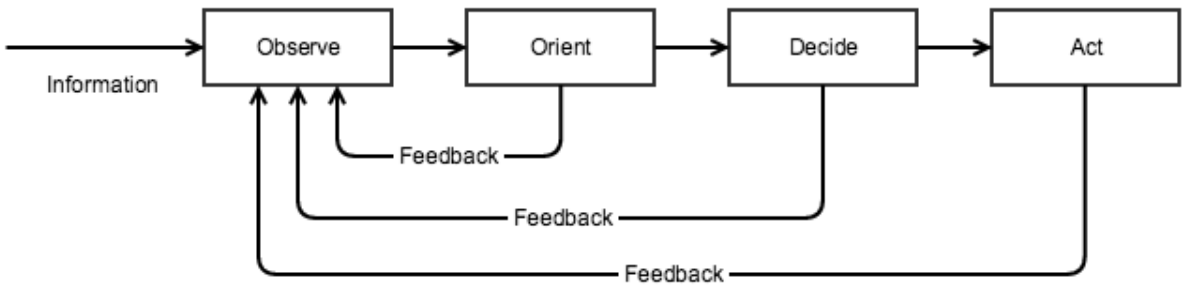


Figure 2.2: Simplified OODA loop

2.3 Competitors and Motivation

With the growing availability of smartphones and reliable networks all across the world, an increasing number of users are starting to perform their once usual web content browsing and simple tasks such as e-mail checking (Mobile Marketshare 2017a) on their mobile devices' applications rather then on laptops or desktop computers (Mobile Marketshare 2017c). Some reports (Mobile Marketshare 2017b) even go as far as stating that mobile web usage has already overtaken their desktop counterpart.

Most mobile device analytics tools available are focused on the aspects of a particular application's performance, user retention, user experience, and centralized crash log management; they are not intended for monitoring the devices' contents, user actions within and outside a particular app, two way communication to remotely control what should be reported, among other features that are indeed what interests us. In short, the ability to log user/device specific/contextual data is not their main focus, even if they allow for it in some limited form, nor is it to retro-actively allow for a roll in it that sense.

Some of the most widely used tools for such analyses as the one's already described are Firebase's Google Analytics, Yahoo's Flurry Analytics, Microsoft's HockeyApp, and Fabric's tools such as Crashlytics and Answers. In Table 2.4 we compare these solutions regarding their capabilities.

As we can see in Table 2.4, none of those tools provide us with the "simple" yet crucial information that a device's location (**Device Location Reporting**) or the list of other installed applications (**Other Apps Details Reporting**) can provide. Another thing that we can immediately realize is that these services are used for spectating and analyzing, not interacting and reacting to the applications and devices that are being monitored (**Actions Over Reports** and **Remote App Customization**). For these services, **anomaly detection and analysis** focuses heavily on monitoring crash logs, and monitoring user behaviour within a particular app rather then monitoring the user or his behaviour with the system and his environment.

Of course that certain anti-virus companies already do the type of monitoring we are aiming for on mobile devices, much like they have done for regular computers throughout the years, however, their tools and the data they collect are kept secret for security purposes. The main takeaway from this section's analysis should be that there truly are no products that provide all the monitoring control we aim for with targets as limited as mobile devices. Chapter 3 details the objectives and requirements of the system we intend to achieve.

Table 2.4: Comparison of existing mobile device monitoring services

	Google Analytics	**Flurry Analytics**	**Fabric**	**HockeyApp**
Supported Platforms	Android, iOS, Others	Android, iOS	Android, iOS	Android, iOS, Others
Dashboard & Graphs	•	•	•	•
Report Customization	•	•	•	•
Actions Over Reports			Very limited	
App Details Reporting	•	•	•	•
Other Apps Details Reporting				
Remote App Customization				
Device Specifications Reporting	•	•	•	•
Device Location Reporting	•			
User Actions Reporting	•	•	•	•

Chapter 3

Objectives and Requirements

3.1 Scope and high-level approach

The purpose of this project is to assess and develop an alternative anomaly analysis and detection system for mobile devices. Anomalies can come under several guises, with the most common kind for concern being malware applications. The main objective for this project is to develop a solution capable of gathering meaningful information that can help determine if and when a device has become compromised, a scenario that is both real and desirable for users and corporations alike.

From a high-level perspective, the proposed solution will consist of three separate entities that communicate with each other: data collection terminals, a central data aggregator and processor, a controller dashboard interface. The terminals, owned by human assets, keep track of various user and system actions, producing data that will be sent to the central aggregator. The central aggregator receives the data from several mobile terminal devices and performs some machine learning and statistical processing on the data, displaying its findings on the dashboard. The dashboard can be operated by an user, altering the inner workings of the central aggregator. All of this can be briefly visualized in 3.1.

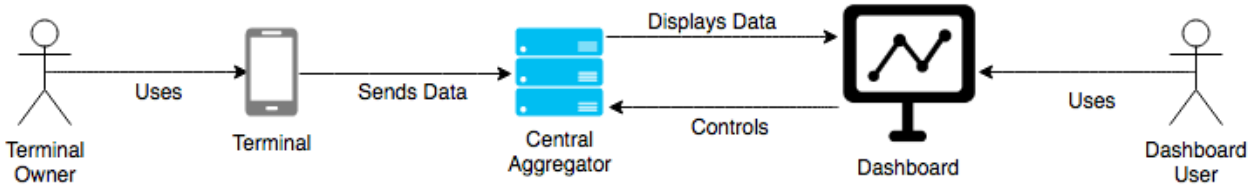


Figure 3.1: High-level approach for the proposed solution

This platform we're proposing will gather information from an ecosystem of devices that can reach large proportions, and as we've seen from Table 2.4 in the previous Chapter, there are already many companies investing money and resources into similar platforms, trying to tap into this kind of problem but one a more restrictive scope. A platform such as the one we're describing could then be sold to large corporations wishing to more easily monitor and

manage their employees' mobile devices, or it could be used for research purposes regarding human and device demographics.

3.2 Platform Requirements

Having no specific client in mind and being a purely academic project, it is no easy task to provide thorough and specific requirements for this project. As such, this section is an example of an engineering thought experiment on creating sensible specifications for a set of tools that can be adapted to distinct large-scale enterprise scenarios, without coming off as too rigorous or focused on a single scenario. That said, some technologies had already been pre-determined beforehand for the project (e.g. the use of Android for the device agents), but fundamented reasons are given further on for them.

It should be noted that due to these consideration, many of the technological choices and reasoning was performed on an *ad hoc* basis, and it becomes particularly clear in Chapters 6 and 7 which are dedicated to the Implementation, Development & Deployment nuances of the project as it progressed.

The following sections provide deeper specifications of what should be done, how it should be done, for what purpose, and how each specification should be assessed. Thus, we present the architectural, behavioral, functional and non-functional requirements for this project, given its aforementioned conditions.

However, since we'll be presenting very distinct requirements for the project and, as with any project, not all are equally as important or feasible. Thus, in order to not compromise the projects development, we have decided to prioritize requirements according to the **MoSCoW** method, assigning each requirement to one of four categories: "*Must Have*", "*Should Have*", "*Could Have*" or "*Won't Have*".

3.2.1 Architectural

The following constraints in Table 3.1 denote decisions and imposed limitations for the architecture as a whole.

Table 3.1: MoSCoW prioritization of architectural requirements

#	Requirement	Priority
1	**Device Agent Operating System** - the Device Agent Application runs on Android devices.	**Must Have**
2	**Loose coupling between Device Agent and Central Aggregator** - changes to the data collected with the terminals is transparent to the central aggregator.	**Must Have**
3	**Central Aggregator** - Uses Machine Learning tools instead of hardcoded classification.	**Must Have**
4	**Dashboard-Aggregator control** - The actions in the Dashboard intended as modifications alter the behaviour of the Central Aggregator.	Should Have [1]
5	**Dashboard Browser Support** - The dashboard runs on Mozilla's Firefox browser.	Should Have [2]
6	**Dashboard Browser Support** - The dashboard runs on browsers other than Mozilla's Firefox.	Could Have
7	**User Anonymity** - Secure connections are used by the system	**Must Have**
8	**User Anonymity** - Secure connections are used by the system and uphold anonymity for the mobile terminal users.	Should Have

É um bocadinho mau ter o Must Have nas partes do dashboard e depois não ter conseguido concretizar. A minha sugestão: reduzir a prioridade e colocar uma chamada de atenção para um texto descritivo que explique, dadas as limitações de tempo, foi dada ênfase aos componentes de coleta e processamento de dados, que constituem o core da solução. Do mesmo modo deverá ficar claro que o Dashboard fará parte dos next steps.

[1] Altered due to time constraints, previously graded as *Must Have*.
[2] See footnote 1.

3.2.2 Behavioral

The following specifications in Table 3.2 detail what has to be done by identifying the necessary behavior of the system and its components:

Table 3.2: MoSCoW prioritization of behavioural requirements

#	Requirement	Priority
9	The **Device Agent Application** collects data from the mobile terminal it is installed in.	**Must Have**
10	The **Device Agent Application** has a low impact on the terminals battery, CPU and bandwidth.	Should Have
11	The **Device Agent Application** periodically transfers data to the central aggregator.	**Must Have**
12	The **Device Agent Application** securely transfers data to the central aggregator.	**Must Have**
13	The **Device Agent Application** copes with connectivity issues with the central aggregator.	Should Have
14	The **Central Aggregator** employs Machine Learning on the collected data for each user.	Should Have
15	The **Central Aggregator** employs Data Analysis on the collected data for each user.	Could Have
16	The **Central Aggregator** employs Machine Learning on the collected data between users.	**Must Have**
17	The **Central Aggregator** employs Data Analysis on the collected data between users.	Could Have
18	The **Dashboard** displays metrics, findings and collected data.	Should Have [3]
19	The **Dashboard** displays user activity.	Could Have

[3] See footnote 1.

3.2.3 Functional

These requirements in Table 3.3 are what has to be done by identifying the necessary task, action or activity that must be accomplished for that purpose:

Table 3.3: MoSCoW prioritization of functional requirements

#	Requirement	Priority
20	The **Device Agent Application** handles the temporary storage of the data in the mobile terminal.	**Must Have**
21	The **Device Agent Application** handles the deletion of unnecessary data stored in the mobile terminal.	Should Have
22	The **Device Agent Application** handles the transfer of data stored in the mobile terminal to the Central Aggregator.	**Must Have**
23	The **Device Agent Application** only deletes stored data from the mobile terminal that has already been transferred to the Central Aggregator.	**Must Have**
24	The **Device Agent Application** requests updated data collection configurations.	**Must Have**
25	The **Device Agent Application** updates its data collection configurations.	**Must Have**
26	The **Central Aggregator** stores the collected data for each user.	**Must Have**

3.2.4 Non-functional

This section specifies in Table 3.4 criteria that can be used to judge the operation of the system, rather than specific behaviors it should uphold:

Table 3.4: MoSCoW prioritization of non-functional requirements

#	Requirement	Priority
27	The data collected with the **Device Agent Application** does not occupy more than 100MB of the mobile terminal's storage space.	Should Have
28	The **Central Aggregator** scales for dozens of users.	**Must Have**
29	The **Central Aggregator** scales for hundreds of users.	Could Have
30	The **Central Aggregator** scales for thousands of users.	Won't Have
31	The **Central Aggregator** stores up to 3 months of consecutive data for each user.	Should Have
32	The **Central Aggregator** stores all the data ever received for each user.	Won't Have
33	The **Central Aggregator** stores up to 3 months of consecutive data for each user.	Should Have

Chapter 4

Planning and Methodology

The following Chapter delineates the project's expected time frame, split in two to accommodate the first and second semesters, and how the work is arranged throughout it. For each semester a detailed plan of its tasks is presented with the visual help of Gantt charts, and each comprising task is given a brief explanation of what it entails. Lastly, the adopted working methodology for the tasks is described.

4.1 First Semester Planning

4.1.1 Gantt Charts

The Gantt charts illustrated in Figures 4.1 and 4.2 detail the tasks, sub-tasks and milestones carried out during the first semester of this project. Figure 4.1 is the condensed visualization of the semester as a whole, emphasizing on the four main topics expanded upon during this period: the writing of the ICCWS Conference paper, the development of the Android Device Agent Application prototype, the regular project management meetings, the writing of this report, and the co-authoring a book chapter proposal, which was accepted (book chapter is due for September).

Figure 4.2 contrasts with Figure 4.1 by going into greater detail regarding the sub-tasks (written chapters or sections, research and software features) and milestones (meetings and

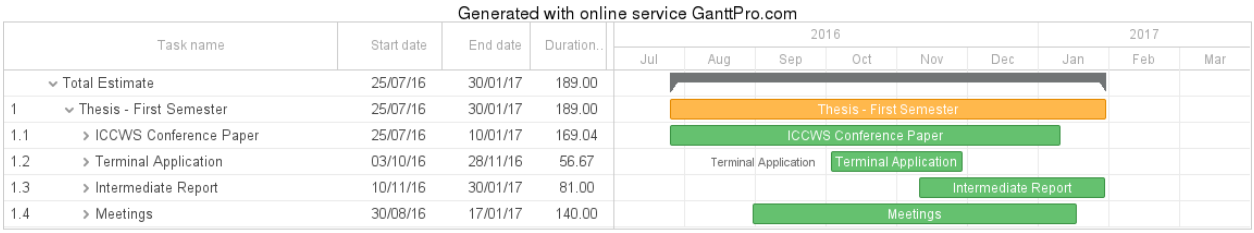


Figure 4.1: First Semester - Collapsed overview Gantt chart

submissions) of each of the already mentioned main work topics thus far.

As one may notice, work on this thesis started before the academic school year so as to accommodate the submission deadlines for the ICCWS conference. Nonetheless, we encountered still encountered a minor setback as the amount of time dedicated to the ICCWS Conference paper, even though it served as the backbone for this document's state of the art research, was greater than expected, but thankfully it paid off by providing a fundamented state of the art and opening up several other opportunities to expand on the subject.

Regarding the first semester of the project, once the planning phase was surmounted, all of the proposed and established goals were met at the specified time frames, in fact, the development of the Device Agent Android Application was moved ahead of schedule due to this, freeing up a large portion of the already heavily time constrained second semester.

4.1.2 Meetings and Deliverables

In order to make the most of the available time and perform as many of this project's objectives whilst minimizing any setbacks, weekly meetings were scheduled with the intent of planning the following week. A short **meeting report** was drafted at the end of each meeting, detailing what was supposed to be researched or implemented up until the following meeting. Moreover, during the course of each week, an additional **weekly activity report** was written and it detailed all the solutions, questions, problems and references encountered. Finally, at the end of each month a **monthly activity report** was put together, overviewing the expected goals for the current month, if they were attained or why not, along with a brief expectation of the following month's objectives.

4.1.3 Publications

During the course of the first semester the opportunity arose to submit a paper for the ICCWS conference to be held on march of 2017. The paper was entitled **Security Monitoring for Mobile Device Assets: A Survey** and its main purpose was to analyze the state of the art in mobile anomaly prevention, monitoring and mitigation, with special focus on how these techniques have changed over time, with not only stationary computers in mind but mobile devices as well. Much of that paper was re-used in the writing of this document's own state of the art. The paper was accepted for the conference and Professor Paulo Simões was chosen as the designated speaker.

4.1.4 Device Agent Application

Initially, the plan was to dedicate the majority of the semester to writing documents and performing research on the topic at hands, which might have meant that more technical aspects would have been left out of this report, as such it was decided early on that the development of the Android Device Agent Application would also be carried out in parallel. This way, the following semester would have a more stable foundation to be built upon.

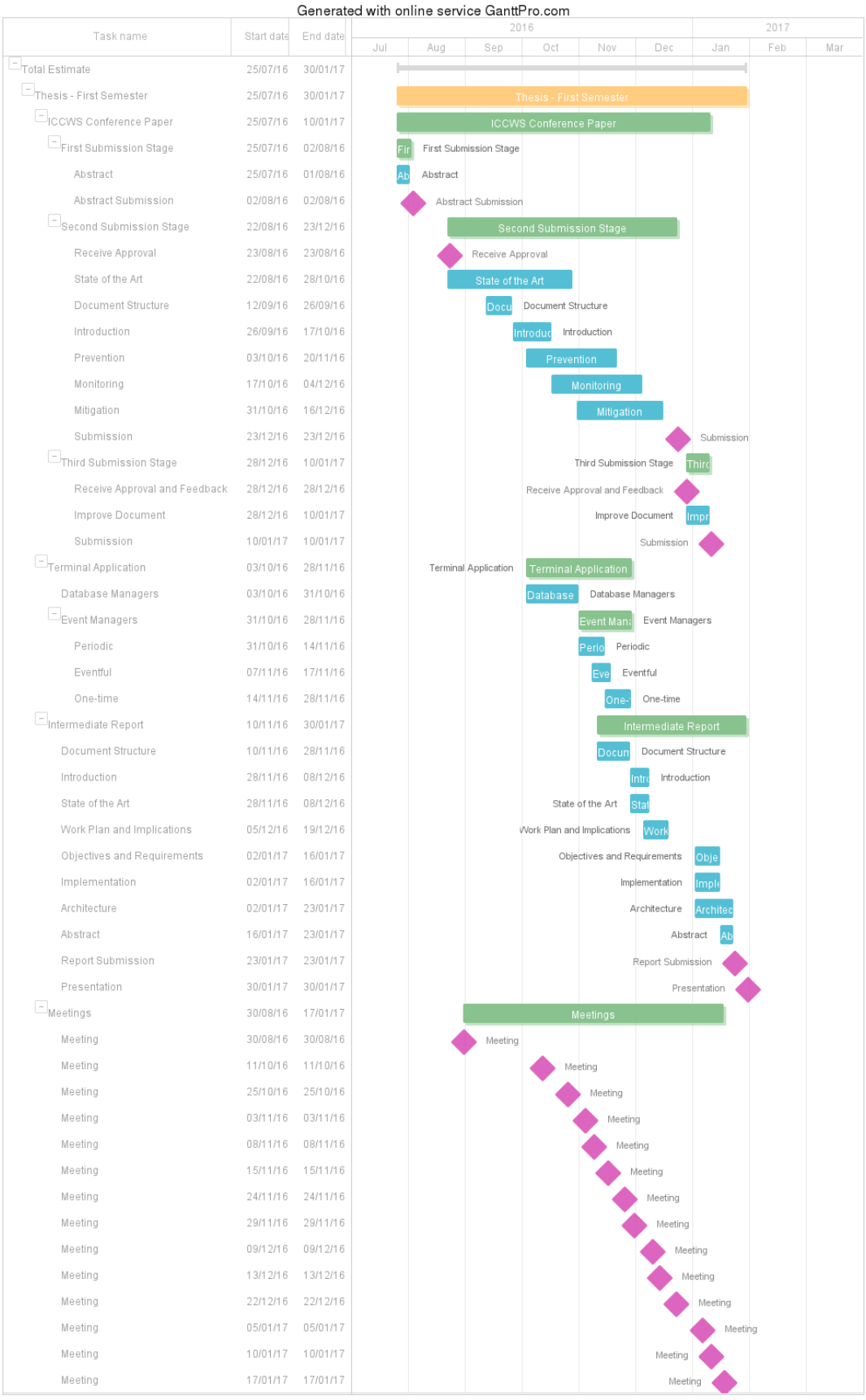


Figure 4.2: First Semester - Expanded overview Gantt chart

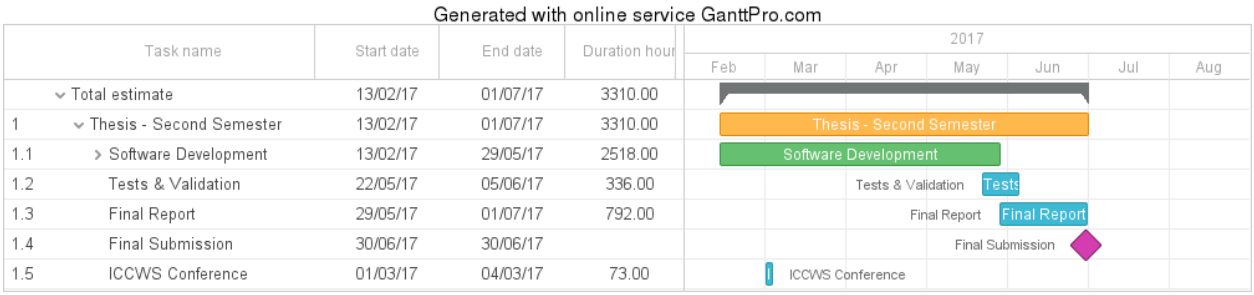


Figure 4.3: Second Semester Predicted Effort - Collapsed overview Gantt chart

4.2 Second Semester Planning

This section is to be interpreted as a compilation of work that is planned and will be carried out during the second half of this project.

4.2.1 Gantt Charts

The Gantt charts illustrated in 4.3 and 4.4 detail the tasks, sub-tasks and milestones carried that are intended to be carried out during the second semester of this project.

The following subsections summarize the tasks and milestone in 4.3 and 4.4.

4.2.2 Sampling

The sampling rate will depend on the impact that each created service has on the device. The use of a profiler should help us analyse this impact, whilst performing several simulations of the services running, we should be able to ascertain an average scan time and logging time, providing us with enough information to determine appropriate sampling times.

4.2.3 Dataset Generation

We will also need to amass a dataset in order to train the Central Aggregator's classifier. The dataset needs to contain examples of both compromised and uncompromised behaviour, as such, we will install the Device Agent Application on several non-compromised emulator and real devices, make a sped up use of them for a prolonged period of time and, subsequently, infect them with several of the most common and most widely spread known malware. This way we will be able to gather data before and after the devices become infected.

4.2.4 Synchronisation

Synchronization should be attempted each hour, every 6 hours, or daily depending on the targeted devices and security desired. Log each synchronization attempt and, in case of

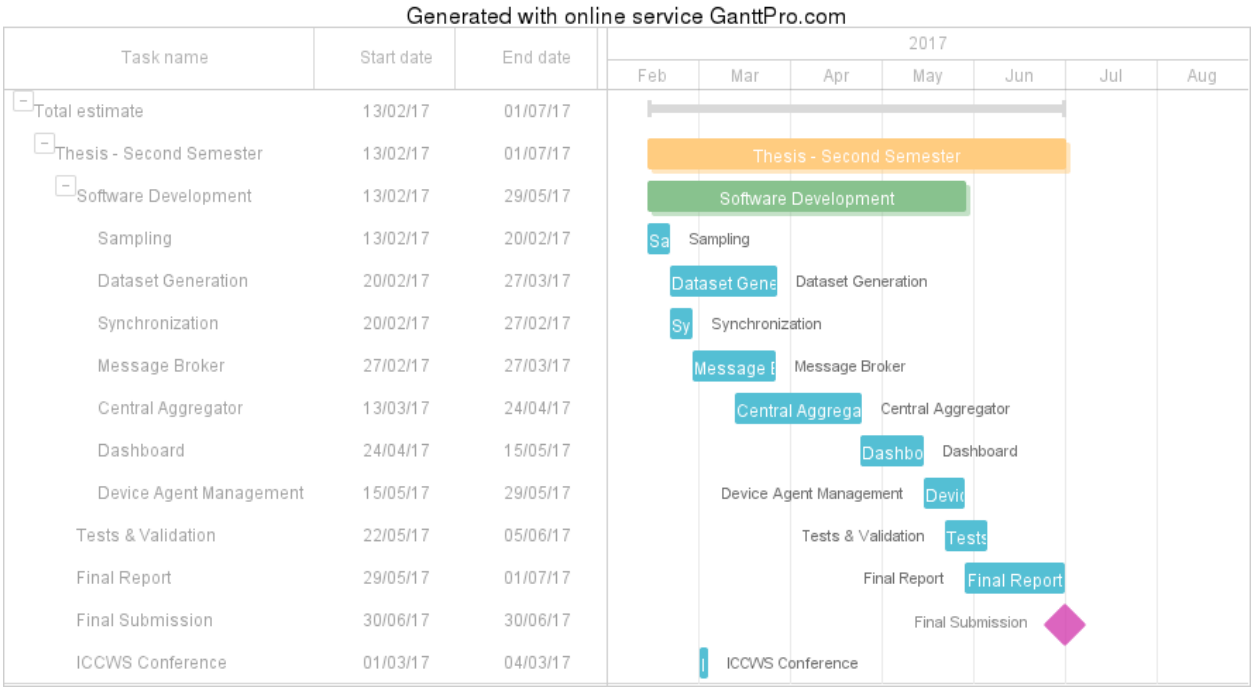


Figure 4.4: Second Semester Planned Effort - Expanded overview Gantt chart

failure, pickup from the last successful synchronization.

4.2.5 Message Broker

In order to transfer the collected data from the Device Agents to the Central Aggregator there must exist a secure connection, and we've opted for a message queue solution.

4.2.6 Central Aggregator

The Central Aggregator is the core section of the system and this project. It is meant to process the gathered information and infer from it if a mobile terminal has become compromised. In order to do this we will make use of the Machine Learning classification tools offered by Apache Spark.

4.2.7 Dashboard

In order to allow for easy control of the system without having to re-compile and re-factor the Central Aggregator, a browser dashboard will serve as the medium between the human analysts and the the core section of the system, with actions carried out by the user having direct effect on the inner mechanics of the system. This section of the system will also serve to display reports on the gathered information and its inferred results, and manage the terminals' user base.

4.2.8 Device Agent Management

The Dashboard should also allow us to control the data that each Device Agent is collecting and how frequently, how it is aggregating its data and how frequently before publishing it to the Central Aggregator. It is a feature with the intent of diminishing versioning issues between Device Agents and re-deploys, going hand-in-hand with the previous subsection, describing how the Dashboard could control the Central Aggregator.

4.2.9 Testing and Verification

We have dedicated a period of time near the end of the semester in which we will be revising, testing, validating and improving the system, this is due to the work methodology we have chosen, which will be explained further on.

4.2.10 Publication

Since our paper from the first semester was accepted for the ICCWS Conference, it was presented by Prof. Paulo Simões in Dayton Ohio from the 2^{nd} to the 3^{rd} of March 2017.

A second paper was also drafted during the second semester, with it's purpose being to analyze our proposed architecture along with the impact that the Device Agent Android application had on the host device's resources, namely the CPU and RAM, but after missing a couple of call-for-papers submission deadlines we ended up not publishing it. The obtained results are however reused in Chapter 8, and it is our intent to submit this paper for the ECCWS 2018: European Conference on Cyber-Warfare and Security.

Thanks to the first paper we published, we were contacted and invited to write a book chapter proposal for CRC's press book entitled *Mobile Apps Engineering: Architecture, Design, Development and Testing regarding mobile development best practices*. Our book chapter proposal was accepted by the editor, and the book chapter itself will be delivered for revision sometime in September of this year. Our chapter focuses on security for mobile devices and applications, and delves into several themes such as:

- Security risks and open challenges
- Consumer vs. corporate environments, niche application fields
- Mobile Device Management
- Security approaches: prevention, monitoring & detection, reaction
- Mobile device and application management
- Security hardening and prevention techniques

4.3 Second Semester Execution

Figure 4.5 is the updated version of Figure 4.4, detailing the offsets that ultimately occurred in the second semester's predicted plan.

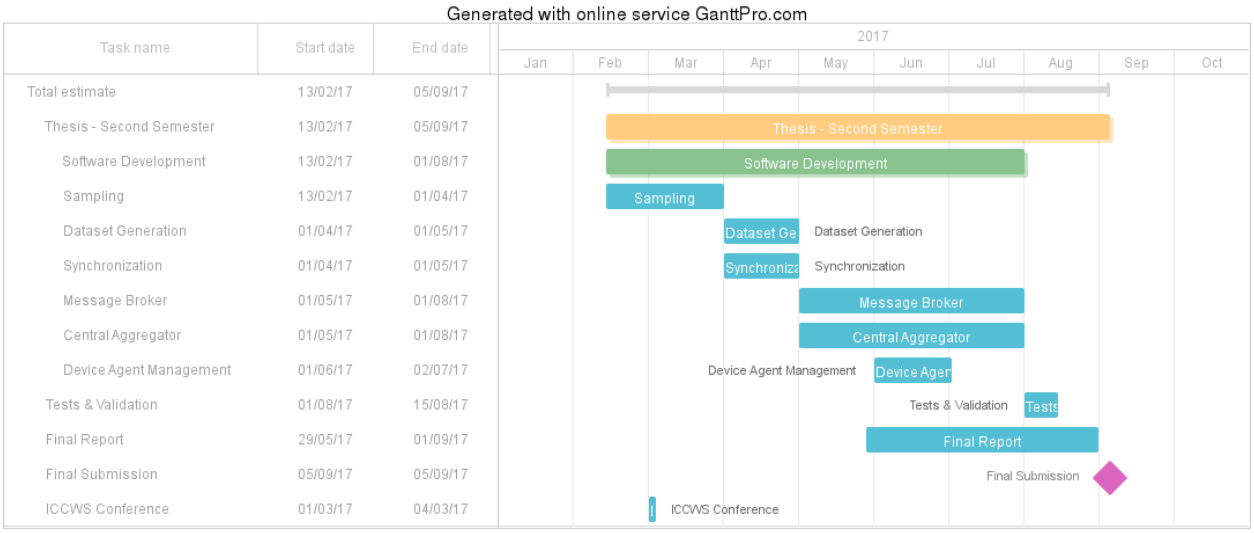


Figure 4.5: Second Semester Executed Effort - Expanded overview Gantt chart

As can be seen Figure 4.5, this project's deadline had to be postponed to September in order to prioritize the development of the Central Aggregator, and in turn compromising the Dashboard component of the platform.

Moving on to the second semester, we started by trying out and setting up several systems from scratch, namely Hortonworks' and MapR's sandboxes, and failing to customize them to the project's needs, even going so far as starting to completely build up the system from the separate components and failing. After all of these experiments we finally tried out **Oryx**, and even though it wasn't as straight forward to configure as the documentation suggested, we managed to eventually get it setup and working. We should have anticipated that a system as complex as a fully fledged Lambda Architecture with Big Data scaling properties would take more time and attempts to configure, ultimately mitigating or avoiding the largest setback for this project.

4.4 Work Methodology

Several work methodologies can be applied to software projects, each with its own strengths and weaknesses and distinct focuses. Some are more focused on the requirements management whilst others ensure that more time is spent on implementation or testing.

Historically, the Waterfall Model has been used for all types of projects, not necessarily related to software development, and because of it's versatility, some defend that it is not suitable for current software projects. It establishes a very straightforward approach to management where one step of the project cannot begin before the previous one is signed off as finished. It is also very relevant to notice that once a part of the model is closed it is not supposed to be revisited.

Figure 4.6: Waterfall Model

Still within an historical aspect, the V-Model can be seen as an extension of the Waterfall Model in the sense that, after the implementation phase each phase can send the project increasingly backwards, denoting more serious and deep rooted problems in the project that need to be revised or adapted.

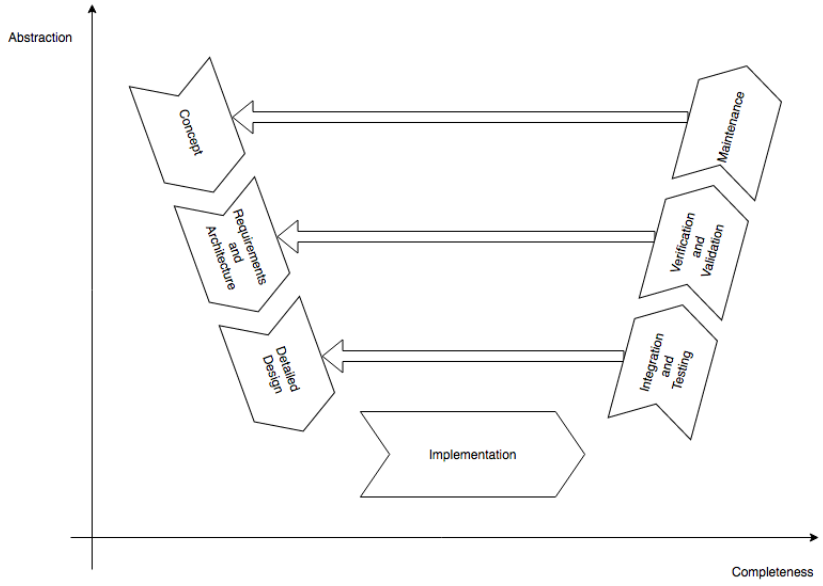


Figure 4.7: V-Model

The work methodology for this project follows a similar reasoning to what propelled the Waterfall Model to become the V-Model. It can be best illustrated with 4.8, where we can

see that there is much more emphasis on feedback loops between the Design, Implementation and Testing phases. The reasoning for this is that, due to the nature of the project's requirements and future maintenance not being very subjective to changes, the iterative improvements will be expressed either on design, code or data flaws.

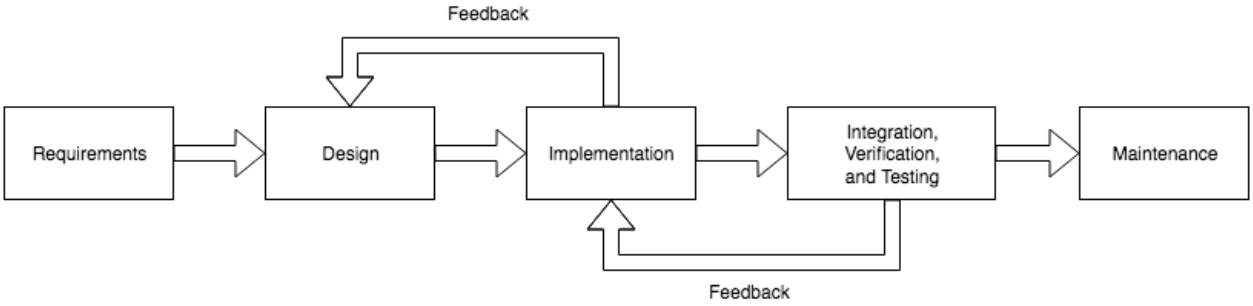


Figure 4.8: Custom Model

Chapter 5

Architecture

This chapter is solely dedicated to present the detailed architecture of the project, from interactions between the several components, to the components themselves and all of their intricacies.

5.1 Overview

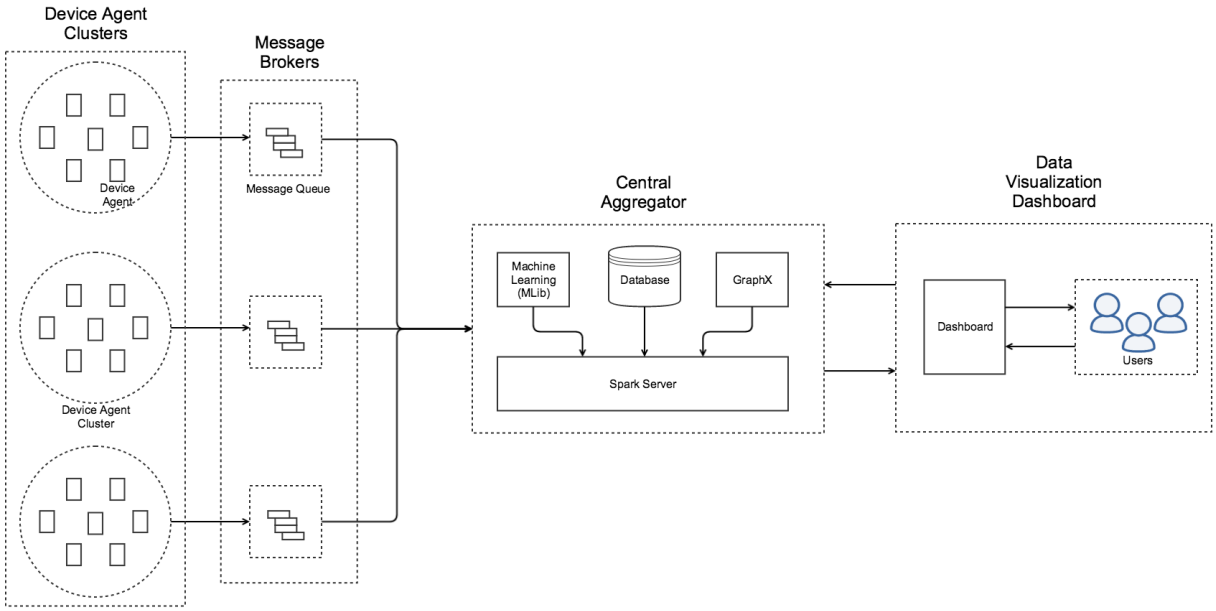


Figure 5.1: Overview of all the architectures

Perhaps the most appropriate diagram to start off this section is that of an overview of the implemented components and how they are related. We can start analysing Figure 5.1 by following the flow of information from left to right:

- We define a cluster of Device Agents, Android mobile devices with our application running. These mobile devices don't need to coexist in the same physical ecosystem.

- Each device agent collects data throughout its stand-by and usage time.
- Each cluster is managed by a message broker, which receives the data collected by each cluster's device agent and relays it to the Central Aggregator.
- The Central Aggregator joins all the collected data in its own Database.
- The Central Aggregator performs some Machine Learning on the data stored in the Database.
- The Central Aggregator creates visual representations of the collected data and the findings from its Machine Learning component.
- The Central Aggregator sends data to be visualized in the Monitoring Dashboard.
- The Monitoring Dashboard can alter the Central Aggregator execution through user actions.

This overview is already sufficient to address some of the **Architectural Requirements** in Table 3.1, namely requirements #**2** and #**4**, which state, respectfully, that the Central Aggregator and Device Agents must be loosely coupled, and that the Dashboard can influence the Central Aggregator's behaviour. Requirements #**1** and #**3** are also contemplated, but we delve into more detail about them in Chapter 6.

5.2 Device Agent

The first section to be analyzed is the Android Device Agent application, which can be divided into three separate components: Collectors, Aggregators, and Data Publisher. Figure 5.2 illustrates this separation.

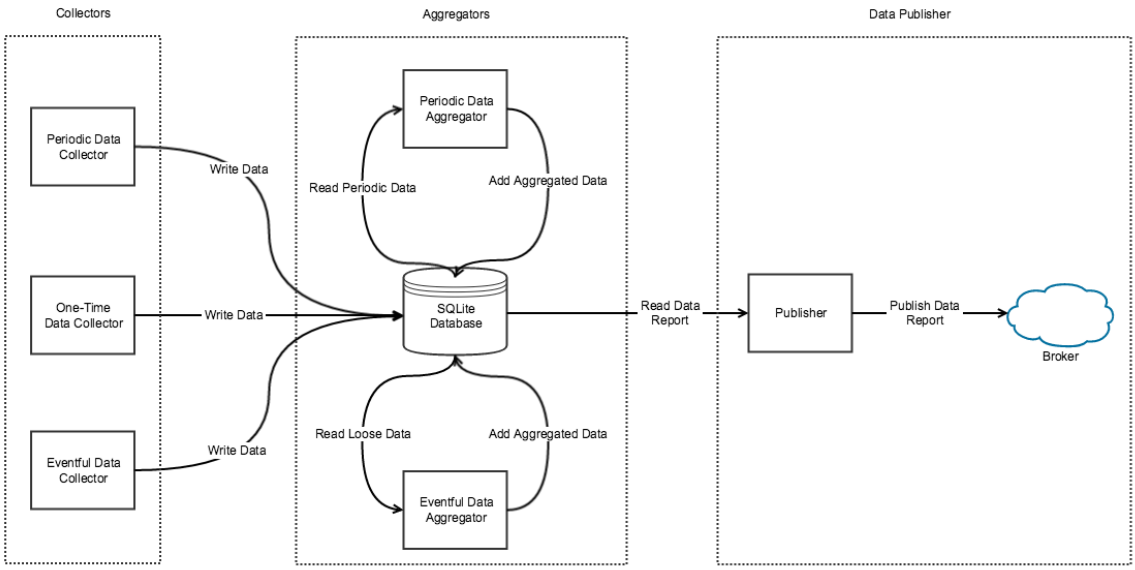


Figure 5.2: Architecture of a device agent's components

As we can see in Figure 5.2, there are three distinct types of Collectors, dictating the separations between the Aggregators further on. The Collectors are in charge of, as the name implies, collecting data generated by the device, and depending on the nature of said data, the method to obtain it varies and so does the way it is handled, processed and stored, which is summarized in Figure 5.3. There are two types of Aggregators, one for the collected Periodic Data and another for the Eventful Data, and their purpose is to join their respective measurements and produce a smaller report of them, summarized in Figure 5.4. The Data Publishing component reads the reports created by the Aggregators and publishes it to a Message Broker.

Each Collector in Figure 5.3 is composed of an IntentService and a BroadcastReceiver. The BroadcastReceiver is responsible for listening for events that will trigger a corresponding action in the IntentService, responsible for handling the events themselves. Figure 5.4 is very similar in structure to Figure 5.3 since the same type of event-driven scheduling mechanism is used for both the Collectors and the Aggregators.

Since we use a database to store data before it is offloaded from the device, it should come as no surprise that we have some utility classes that represent the database, its tables

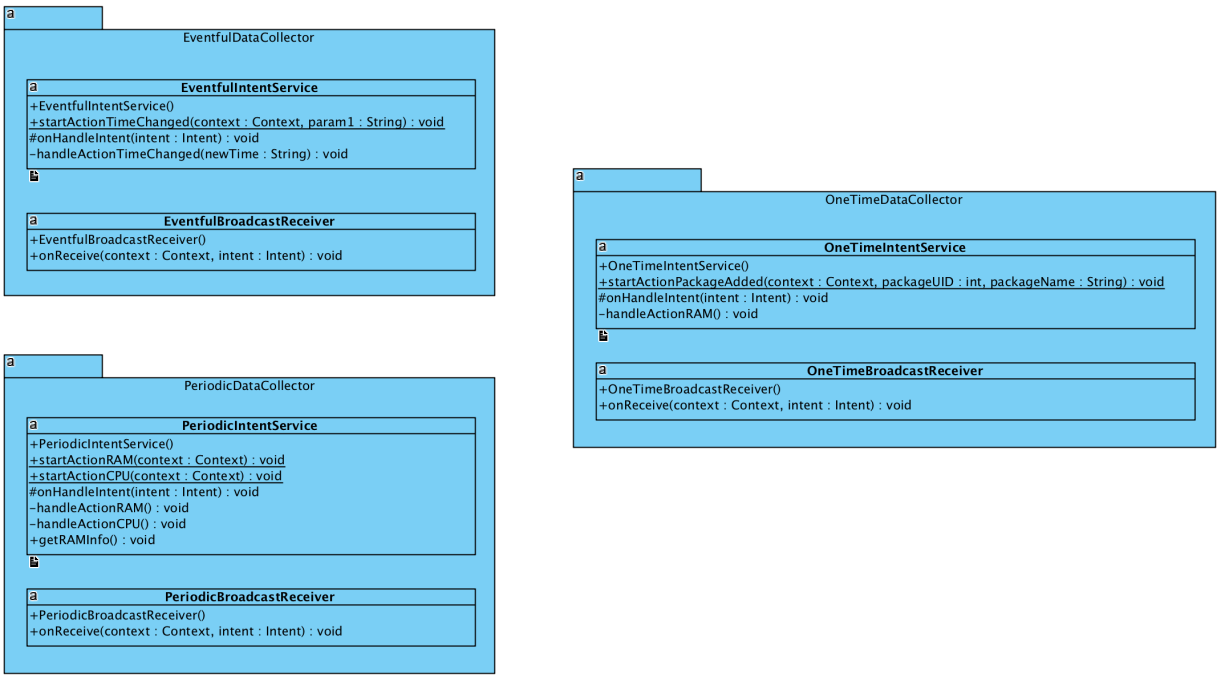


Figure 5.3: UML Diagram of Collector classes used in the device agent application

and their entries, using them to abstract the actual management of the database. We detail these classes in Figures 5.5, 5.6 and 5.7.

We can see in Figure 5.7 that all entries are subclasses of the Measurement object, and they diversify based on their core purpose. We opted to keep the simple leaf classes in order to preserve code readability and to minimize code refactoring should any differences arise for the leaf sibling classes (EventfulMeasurement, PeriodicMeasurement and OneTimeMeasurement, or EventfulAggregatedMeasurement and PeriodicAggregatedMeasurement).

Finally, Figure 5.8 details the tables that exist within the SQLite database. The attentive reader might notice that this entity relationship diagram has no relationships between its entities, but this is no mistake, the database is only used to store data in an easy and structured fashion, and the complexity of the device agent component does not warrant more complex relations.

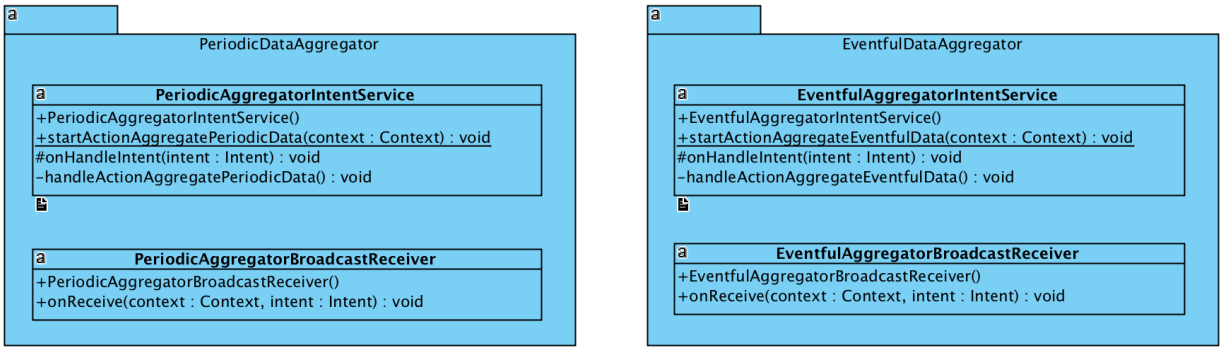


Figure 5.4: UML Diagram of the Aggregator classes used in the device agent application

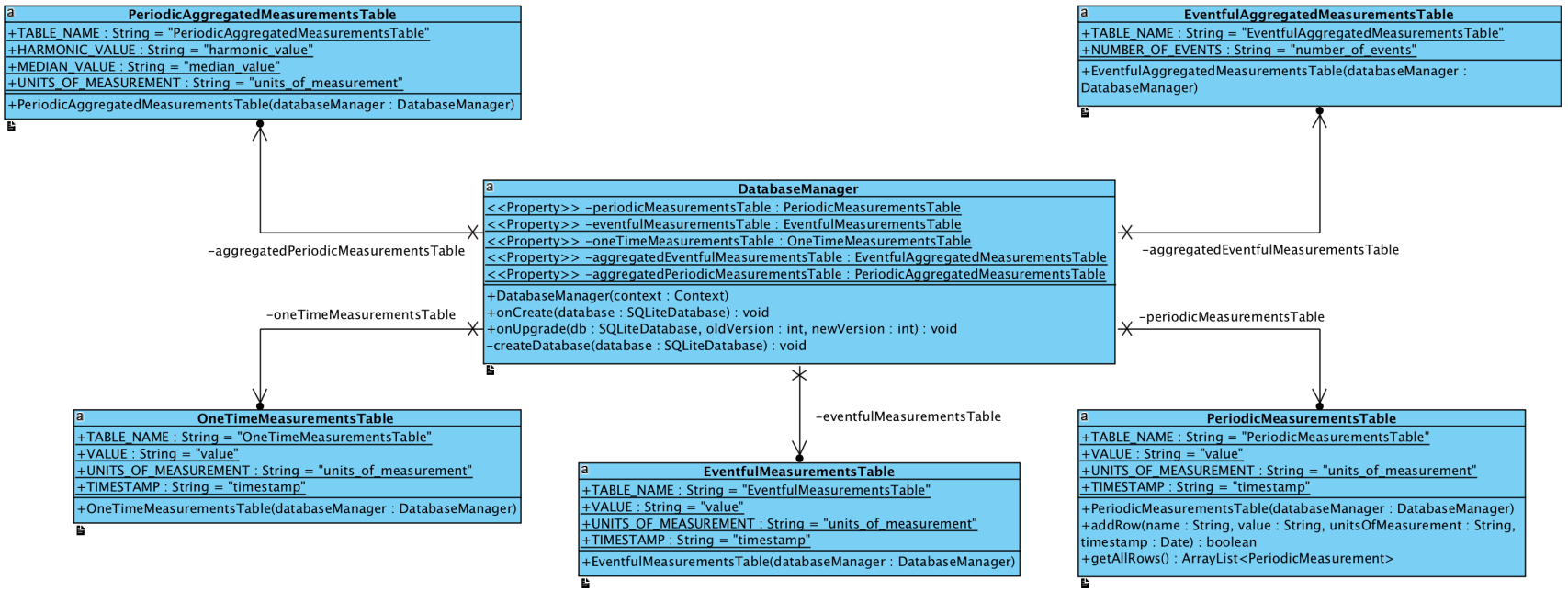


Figure 5.5: UML Diagram of Database Manager and it's relationship with other classes used in the device agent application

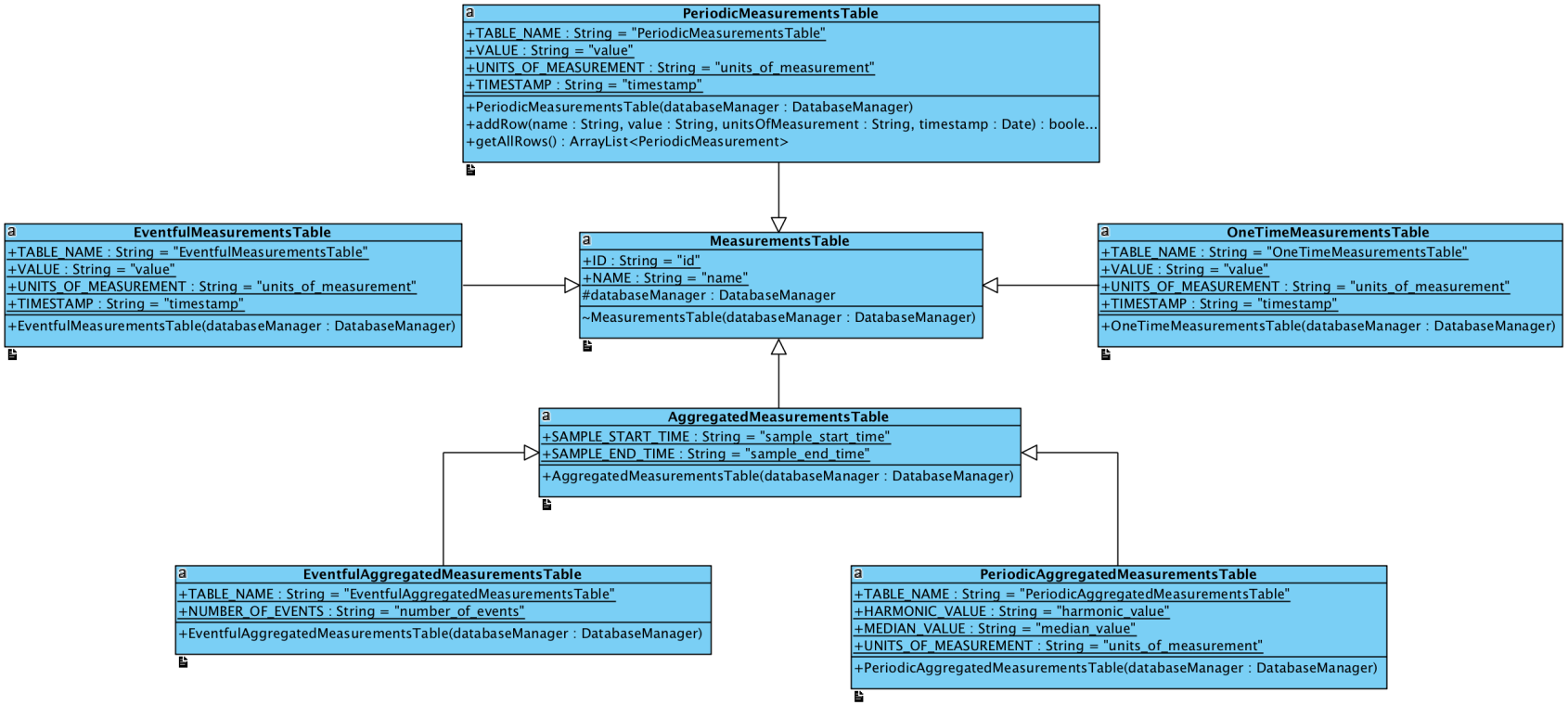


Figure 5.6: UML Diagram of Database Table classes used in the device agent application

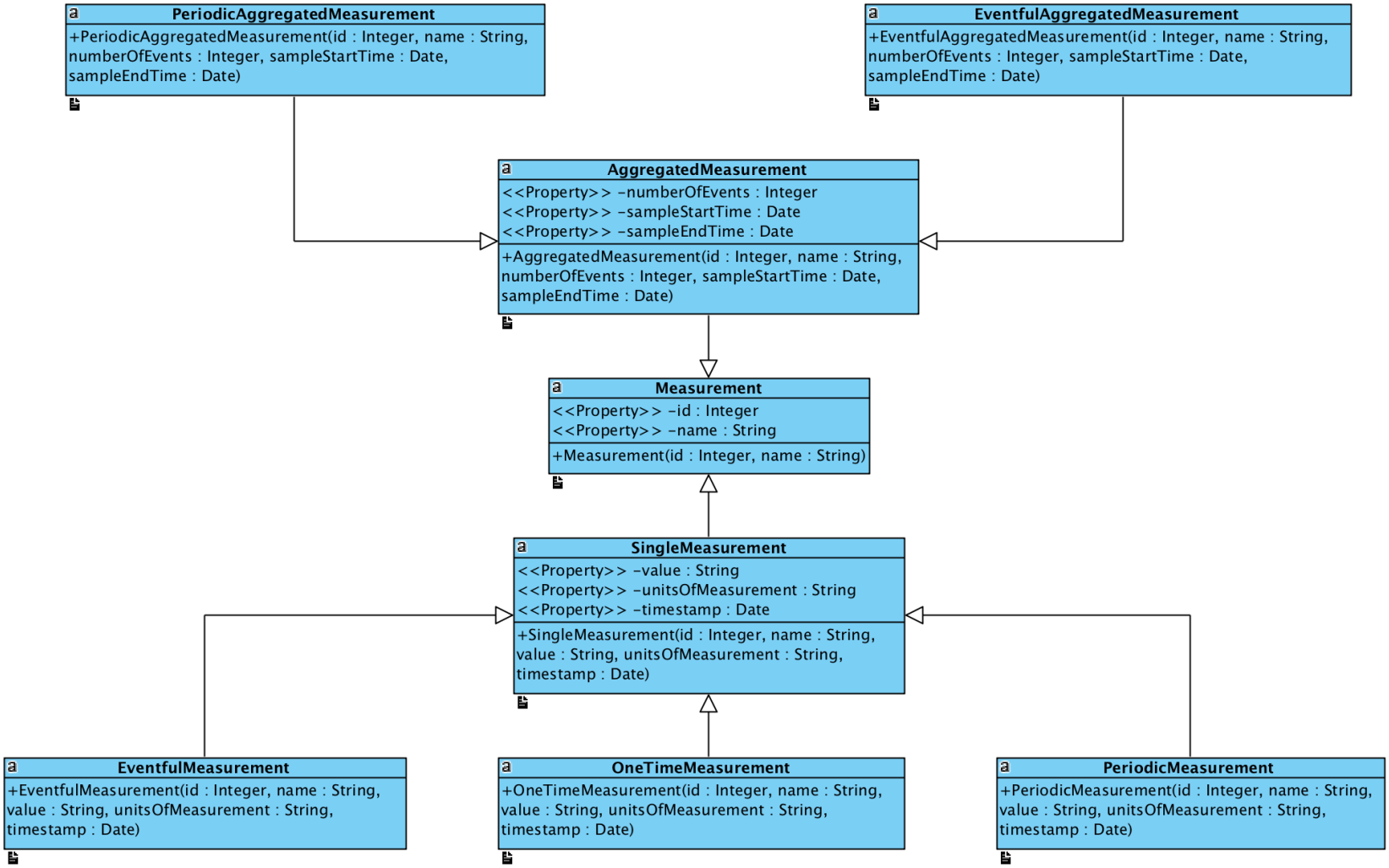


Figure 5.7: UML Diagram of Database Entry classes used in the device agent application

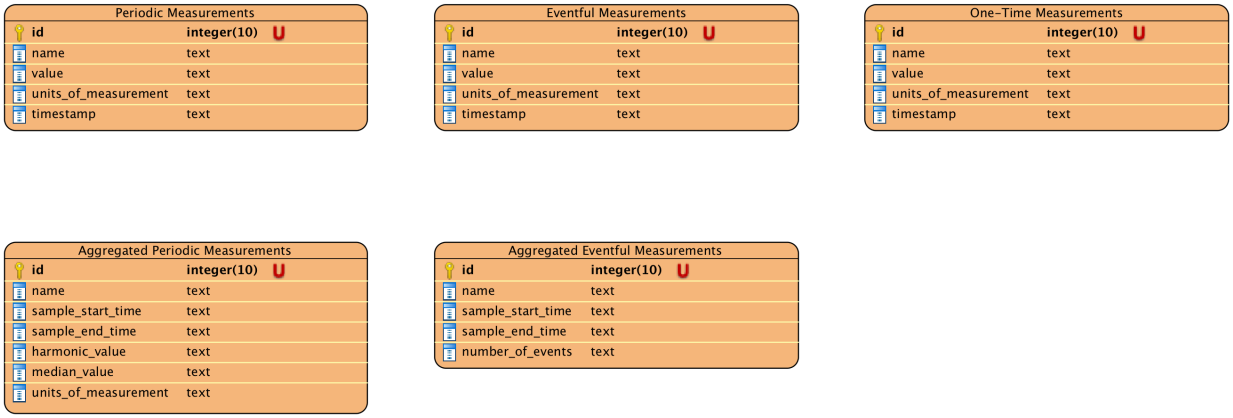


Figure 5.8: Entity relationship diagram for the device agent application's SQLite database

5.3 Device Agent Management

An interesting concept that has gained increasing interest is that of **Mobile Device Management** (MDM), already discussed in the State of the Art, especially for large networks or clusters of devices. It aims to allow for asynchronous customization of the devices without having to physically or locally interact with them. In Figure 5.9 we explain our intention as far as device management goes, namely:

- A device agent checks for changes to its configurations with the Device Management Server.
- The Device Management Server can request the configurations for a specific Device Agent.
- The Device Management Server can send the new configurations to a specific Device Agent.
- The Dashboard can alter the configurations of a specific Device Agent via the Device Management Server.
- The Dashboard can alter the configurations of all Device Agents via the Device Management Server.

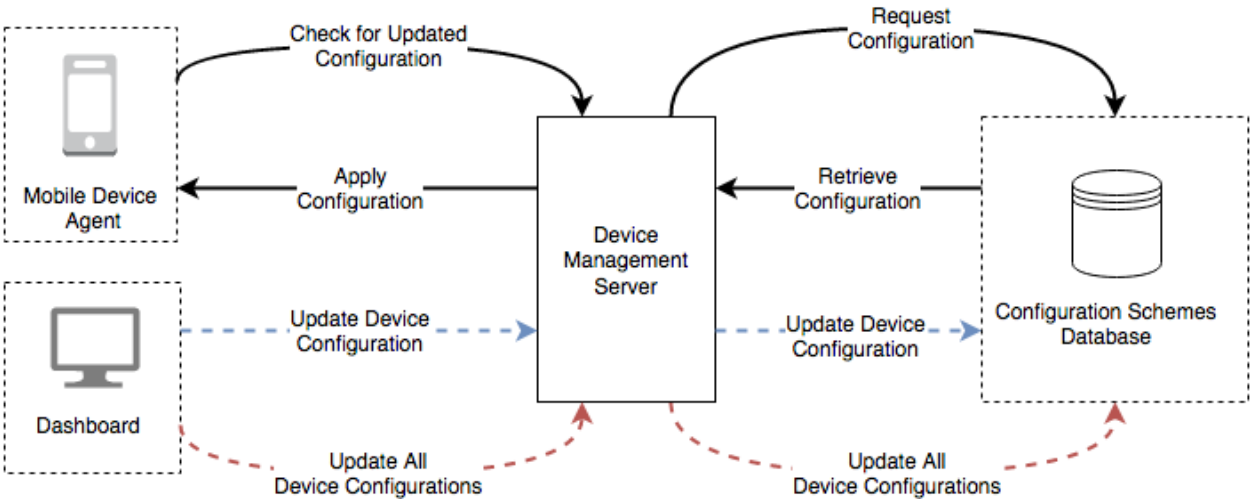


Figure 5.9: Diagram for device agent management

Chapter 6

Implementation Details

The following sections detail every technology used for each part of this project, why they were chosen over their possible counterparts/competitors, and how they influenced the course of the project. These sections do not exclusively contain components that have already been developed.

6.1 Device Agent

The agents are Android devices, each running an application that manages all of its data collection, aggregation and communication tasks. The device agents' main goal is to collect data, which can be pre-processed, and send it to a more competent processing unit.

In order to achieve this, we need to establish some goals for our application:

- It must be small in size, in order to fit on reduced space devices.
- It must be device-agnostic, meaning that it can run on wide range of devices without special considerations for the device's hardware.
- It must collect data autonomously and periodically.
- It must collect data without hogging too may system resources.
- It must perform some pre-processing on the collected data without hogging too may system resources.

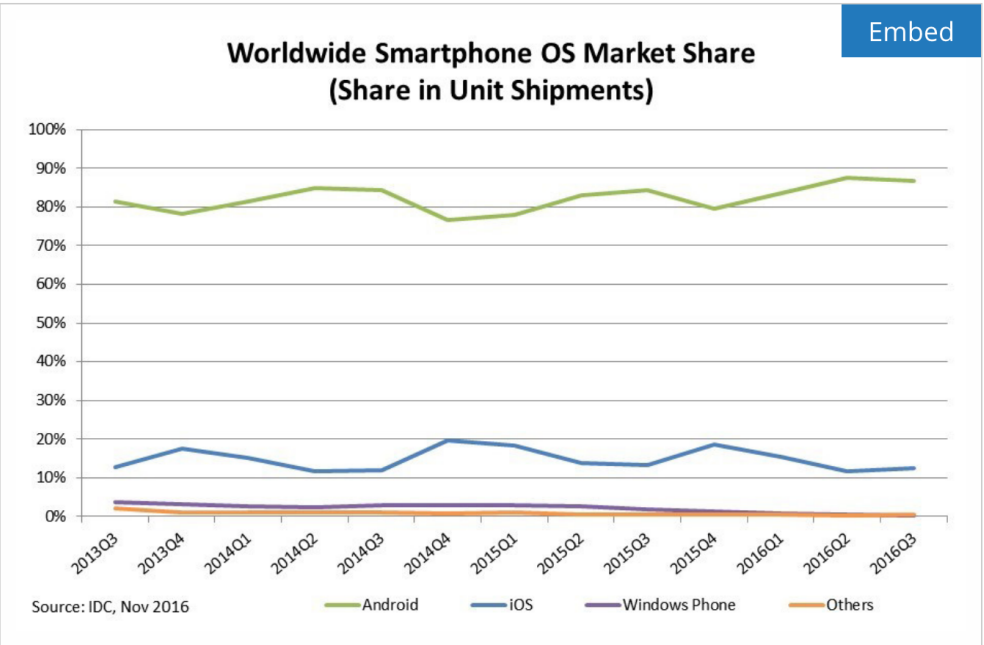
Having all of these considerations in mind, we can now make a more conscious choice of the **Operating System** as well as the tools and behaviour of the **Data Collection**, **Event Management**, and **Data Aggregation**.

6.1.1 Operating System

There currently are three distinct prominent operating systems for mobile devices: Google's Android, Apple's iOS and Microsoft's Windows Phone. While there are other mobile platforms, they are not as versatile, popular and widespread as these three. Even among these

top three operating systems there is one that dominates the others, be it in market shares or in the amount of literature written for it: Android.

Figure 6.1 illustrates the sheer gap between Android's market share and the remaining competitors estimated user base. It should be noted that this figure does not account for the distinction between enterprise and regular users.



Period	Android	iOS	Windows Phone	Others
2015Q4	79.6%	18.7%	1.2%	0.5%
2016Q1	83.5%	15.4%	0.8%	0.4%
2016Q2	87.6%	11.7%	0.4%	0.3%
2016Q3	86.8%	12.5%	0.3%	0.4%

Source: IDC, Nov 2016

Figure 6.1: Mobile device operating systems market shares
Source: (IDC 2016)

Moreover, we can see from Table 6.1 that there is a proportional expected difference in the amount of literature, content and references to each of the operating systems. Searching for the following keywords yields the following amount of results:

Not only is Android the most popular on both of the previous accounts, it is also the only operating system which provides out-of-the-box the means to easily collect the most

Table 6.1: Number of search results for keywords and different search engines

	Keywords		
	Google Android	**Apple iOS**	**Microsoft Windows Phone**
Google	**553,000,000**	345,000,000	94,000,000
Bing	**50,500,000**	27,300,000	49,100,000
Yahoo	8,490,000	**13,700,000**	2,600,000

amount of data from the device itself, as well as other applications installed in it.

All of these factors were considered, hailing Google's Android platform as the most suitable development tool for this project, meanwhile addressing one of the **Architectural Requirements** in Table 3.1, namely #**1**, stating that the Device Agent Application runs on Android devices.

6.1.2 Data Collection

As far as data acquisition goes, we can split the types of data collected between the **Application Level** and the **Device Level**.

Application Level:

- Install/Update/Remove
- Permission List
- RAM
- CPU

Device Level:

- GPS **(optional)**
- Base station connection/disconnection
- CPU usage
- RAM usage
- Battery
- Charging State
- Connected Network (WiFi/3G)
- Open Ports
- Data Traffic **(optional)**

Although all of these different data sources will have varying sampling rates, they will have to be fine tuned through experimentation. It should also be noted that optional parameters are the most likely battery drainers, and will have to be analysed more in depth than others. It is also worth mentioning that Data Traffic analysis can only be achieved for rooted devices in conjunction with tcpdump, which as of writing is still not a certainty as far as this project's objectives are concerned.

Another way of dividing the different types of data collected is their frequency and triggering conditions, namely if it is a repeating scheduled event, triggered by an action that can occur several times, or even triggered by an action that can only occur once.

Periodic:

- RAM
- CPU
- GPS
- CPU usage
- RAM usage
- Battery
- Open Ports
- Data Traffic

Eventful:

- Base station connection/disconnection
- Charging State
- Network (WiFi/3G) connection/disconnection
- Time and Timezone changes
- Power On/Off

One Time:

- application install/update/remove
- application permission list

Most of the collected data's purpose is to identify any potential anomalous behaviour such as unexplained power or processing spikes, excessive data transfer, and so on, but also to be able to facilitate the construction of an event timeline for each device agent.

With that said, each and every piece of gathered data is timestamped, which is also the reason for the collection of *Time* and *Timezone* changes: an attempt to sort out any potential ambiguous time mismatches and corner cases. Of course that, in order for this to happen we assume local time coherence.

6.1.3 Event Management

In Android there are several ways of performing tasks in the background of an application and even the device, the main ones are:

- ***Service*** - used for short, single threaded tasks that need not interact with an activity.
- ***IntentService*** - used for tasks that require more resources (threads, time, etc.) than regular *Services*, they also need not interact with an activity.
- ***Handler*** - objects that allow us to send and process *Message* and *Runnable* objects associated with a thread's *MessageQueue*, effectively allowing for background management of UI components.
- ***AsyncTask*** - abstractions of *Handler* objects, tightly coupled with an *Activity*'s UI thread and the possibility of updating it before, during or after the task finishes running.

Most, if not all, of these methods are abstractions for Java *Thread* management, providing easily overridden methods for thread management without all of the security and integrity issues that they are usually associated with:

- Race Conditions
- Deadlocks
- Livelocks

The main differences between all of these threading abstractions are the appropriate task length, if they can/should manage UI components, and how independent they are from an *Activity*. We will be using *IntentServices* because they are the most versatile and decoupled option.

Selecting the most appropriate medium to manage events is only part of the problem, the other part is setting up the system to listen for and schedule the events. An *IntentService* does not schedule itself, for that we need to setup several other helper classes, namely:

1. Register a ***BroadcastReceiver*** for that type of events.

2. Listen for events.

3. Handle the event occurrence with the appropriate ***IntentService*** and execute it.

Events can be system generated or customly generated by applications, in case of the latter some additional precautions have to be accounted for:

- If the event is generated by our application:

    1. Create a ***PendingIntent*** (the event) that will be triggered in the future and picked up by the appropriate *BroadcastReceiver*.
    2. Schedule an ***AlarmManager*** to execute the *PendingIntent* after an interval of time has elapsed.

- If the event is system generated then the system handles the process of dispatching an ***Intent*** for it to any *BroadcastReceiver* listening for events of its kind.

This flow of information can also be visualized in Figure 6.2.

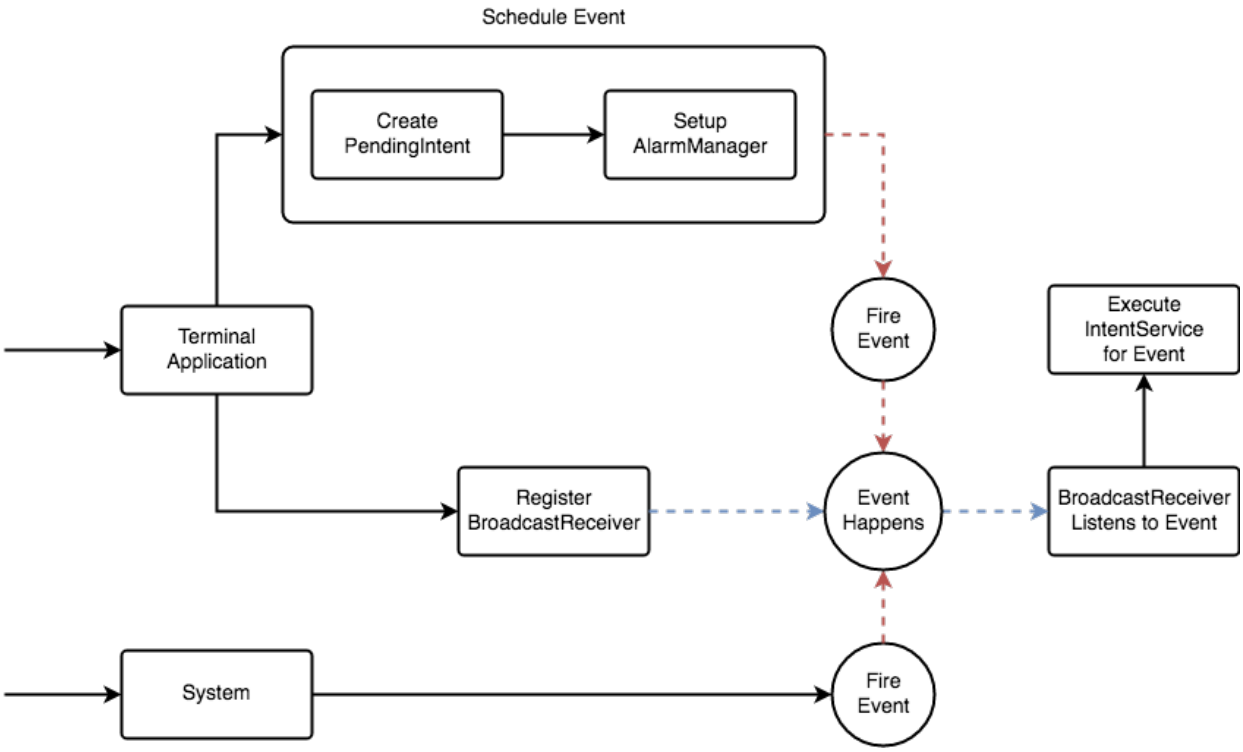


Figure 6.2: Application event management - creating custom events, listening and handling events

It should be noted that an *AlarmManager* will re-schedule automatically unless an execution counter is added to it. We also opted for the *Inexact time measurement* mode of *AlarmManager* to minimize the resources associated with the application while it is running

in the background.

This section addresses two of the **Behavioural Requirements** in Table 3.2, namely #**10** and #**11**, stating that the Device Agent Application has low impact of the device's battery, CPU and bandwidth, and that the Device Agent Application periodically transfers data tot he Central Aggregator.

6.1.4 Data Management

There are two main ways of storing information outside of an application's **Shared Preferences** in Android, **SQLite** and **logging**. Although logging is a straightforward and fast means of storing the collected data in files, it is not enough in our case as we wish to do some preliminary work on the collected data, which raises the need for added complexity in reading, modifying and writing again to the log files.

Additionally, we want to be able to restrict the amount of collected data by each device to a maximum of 100 MB, which both Shared Preferences and standard logging don't easily allow for, and to be able to delete no longer necessary data after it is succesfully sent to the Central Aggregator. Because of these restrictions, we opted for a local **SQLite Database** to manage the collected data.

6.1.5 Data Aggregation

The purpose of the thes aggregation stage is to diminish the amount of data that needs to be transmitted from the device agents to the central aggregator, mainly by condensing the collected data into timeframe windows of value changes.

There are several measures of central tendency that can be calculated from the collected data, and each can be used for different approaches:

- **Median** - provides the "most likely to be the correct" value in a series of observations.
- **Geometric Mean** - often used when comparing different items, where each item has multiple properties that have different numeric ranges, by finding a single "figure of merit" for them
- **Arithmetic Mean** - when the context is clear, is the sum of a collection of numbers divided by the number of numbers in the collection.
- **Harmonic Mean** - the harmonic mean of a list of numbers tends strongly toward the least elements of the list, it tends (compared to the arithmetic mean) to mitigate the impact of large outliers and aggravate the impact of small ones.
- **Generalised mean** - also known as a power mean, it serves a non-linear moving average which is shifted towards small signal values or big signal values according to its parameter p.

Having the project's data in mind, and not wanting to waste time and space on unused or irrelevant calculations, we decided that it would be best to only use the median and harmonic mean, mainly for their natural inclination of providing accurate summaries of data containing outliers.

6.2 Message Brokers

Since base stations are black boxes with which we cannot meddle easily, the need for a communication intermediary to manage the information gathered by the device agents became a necessity. But it could not just use any means of communication, particularly on the device's side, it needed to be lightweight and fast, but also connect easily with the other parts of the system. After some research, the best candidates seemed to be plain HTTPS communications and MQTT.

6.2.1 MQTT and HTTPS

MQTT (Message Queue Telemetry Transport) is minimalistic a publish-subscribe service based on message queues, designed for unreliable connections and very simple devices, which in turn makes it perfect for the IoT (Internet of Things) and M2M (Machine-to-Machine) scenarios as it attempts "to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery." Despite all of these desirable characteristics, MQTT does not provide any security to the messages it manages, as this is often the most costly step of the communication. It should be noted that MQTT is not incompatible with the actual inclusion of a security scheme around the messages (such as SSL/TLS [Secure Sockets Layer/Transport Layer Security]), it just doesn't provide one out-of-the-box.

Figure 6.3: MQTT Organization logo

HTTPS (Hyper Text Transfer Protocol over SSL/TLS) is the most common way of consuming and sending data over secure connections between device agents and a server. Its security comes from a generated public key certificate, which is verified by a trusted certificate authority and used to establish each connection. Communications between the server and the client are usually established in the following manner:

1. The client asks server for public key, public key certificate, certification authority and other metadata.
2. The server sends a X.509 certificate chain and a digital signature signed with its own private key.
3. The client validates all the sent data.
4. The client generates a session key, encrypts it with the server's public key, and sends it to the server.

5. The server decrypts the message with its private key, obtaining the session key.

6. The session key is used by both sides to encrypt the following message exchanges.

As we can see, an asymmetric encryption scheme is added on top of a symmetrical encryption scheme in order to secure the messages between two ends and avoid MITM (man-in-the-middle) attacks. This form of attack is achieved by placing a malicious entity between the two communicating entities on the edges, reading and relaying the messages between both by impersonating them, and having the potential to alter the message's payload.

A few considerations were had in order to choose between MQTT and HTTPS:

- HTTPS is widely used with REST (Representational State Transfer) in order to organize and strengthen interactions between independent systems (Fielding 2000)
- HTTPS/REST services might be excessive since it is usually used as a two way communication service. The device agents will not consume information, only push it to the server, and the aggregator will do the opposite by not sending data to the device agents, only receiving it.
- IBM uses MQTT for Big Data analysis related projects with millions of devices, in fact they invented it for that same purpose.
- MQTT is less resource intensive than an equivalent HTTPS service.
- The max payload size possible in MQTT is 256Mb, and 65Kb for the topic message.

Although all of them have their advantages and disadvantages, we ended up settling for MQTT mainly for its out-of-the-box lightweight virtue. However, MQTT does not provide encryption due to this very nature, as such, we plan on encasing the messages with the help of an encryption scheme with less overhead than SSL/TLS, so as to avoid eavesdropping or man-in-the-middle compromises but without increasing the overall battery drain on the device agent.

Figure 6.4: Mosquitto logo

Thankfully, **mosquitto**, an open source message broker that implements the MQTT protocol, keeps true to the philosophy of MQTT all the while providing SSL support for

encrypted network connections and authentication. By using this framework we can combine the best of SSL and MQTT just as we had intended, all out-of-the-box and without the additional overhead of implementing it ourselves, which could lead to unintentionally compromising either MQTT's lightweightness or SSL's privacy in the process.

6.2.2 Hadoop and Kafka

Since we are going to use Hadoop to manage a large portion of the Central Aggregator, and its ecosystem also has a messaging tool to answer messaging needs, one would think that the whole debate over whether to use MQTT or HTTPS would be pointless but, as we explain in the following paragraphs, that is not the case.

Figure 6.5: Kafka logo

Apache's Hadoop provides its own alternative out-of-the-box. Apache Kafka is Hadoop's publish-subscribe messaging framework and is pretty similar to MQTT as it aims to provide fast, scalable and redundant messaging, along with streaming and storage capabilities. An established Kafka broker will intermediate between producers and consumers, or, respectively, publisher and subscriber clients. Producers can publish messages to topics on a broker, which will then be forwarded to the consumers that have subscribed to them, very much akin to MQTT.

Just like MQTT, Kafka was also made to satisfy overwhelming messaging needs, but in this case it was for a particular company: Linkedin. Linkedin boasts that several million messages circulate and need to be managed by their systems at any given time, always ensuring that their causality and integrity is maintained across systems. This is the reason they designed and open-sourced Kafka under the an Apache Foundation license, and as such Kafka is deeply rooted in and supported by Apache's own Hadoop, but because of this, it is not as lightweight as MQTT, since there is no available mobile-friendly lightweight version of this ecosystem for Android, making near impossible and suboptimal to establish an Hadoop cluster on a mobile device due to its limited resources.

Ultimately, the reason we need both MQTT and Kafka is that we need to make the most out of each one's strengths. MQTT is a lightweight framework which we can embed in the Android Application design for inconspicuous data collection, whilst we need to use Kafka for its absurd throughput, which can only be achieved thanks to its tight-coupling with the behemoth that is Hadoop.

6.2.3 MQTT and Kafka bridge

Thus far we've established that we are going to require the use of MQTT on the mobile device's android application, and Kafka on the Hadoop central aggregator, but now we have another problem: translating and forwarding the messages from MQTT to Kafka. Fortunately, messages in MQTT and Kafka are comprised of plain text and not serialized proprietary objects, so no translation is required between the two technologies. However, forwarding is the same messages between the two is not as trivial.

Although MQTT and Kafka both queue their messages according to topics, MQTT goes a step further and allows for hierarchically nested subtopics, intended to progressively specify the purpose and context of the message, without losing information as to how it is related to other topics and messages. Topic names follow a couple of considerations:

- They are case sensitive
- They must be comprised of UTF-8 strings
- They must consist of at least one character to be valid
- The "/" character is reserved for delimiting topic sublevels
- The "+" character is reserved for specifying a single level wildcard
- The "#" character is reserved for specifying a multi level wildcard, and can only be used as the last subtopic specifier
- A client can only publish to an individual topic, wildcards are not allowed
- A client can subscribe to and individual topic or multiple topics by using wildcards

Let us consider the following examples of topics to describe a house:

```
1 myhome/groundfloor/livingroom/temperature
2 myhome/groundfloor/livingroom/humidity
3 myhome/groundfloor/kitchen/temperature
4 myhome/1st floor/bedroom 1/temperature
5 myhome/1st floor/bedroom 1/humidity
6 myhome/1st floor/bedroom 2/temperature
7 myhome/1st floor/bedroom 2/humidity
```

Listing 6.1: MQTT: example topics

If a client were to subscribe to `myhome/groundfloor/#` he would only receive updates from the following topics:

```
1 myhome/groundfloor/livingroom/temperature
2 myhome/groundfloor/livingroom/humidity
3 myhome/groundfloor/kitchen/temperature
```

Listing 6.2: MQTT: example of topics filtered by subscription

However, if the client were to subscribe to `myhome/1st floor/+/temperature` he would receive updates from the following topics:

```
myhome/1st floor/bedroom 1/temperature
myhome/1st floor/bedroom 2/temperature
```

Listing 6.3: MQTT: example of topics filtered by "+" wildcard

Because of these features, we were immediately inclined to use a similar topic hierarchy for our mobile device MQTT brokers. We would aggregate everything under a common name, specify the type of measurement that the message contained, and which device had made it, resulting in the following topics:

```
COLLECTED_DATA/PeriodicMeasurement/<device_id>
COLLECTED_DATA/OneTimeMeasurement/<device_id>
COLLECTED_DATA/PeriodicAggregatedMeasurement/<device_id>
COLLECTED_DATA/EventfulAggregatedMeasurement/<device_id>
```

Listing 6.4: MQTT: example of desired topic hierarchy

This way, if a client were to subscribe to `COLLECTED_DATA/PeriodicMeasurement/#` he would receive updates from all devices for periodic measurements.
If he subscribed to `COLLECTED_DATA/+/<device_id>/` he would receive updates from all types of measurements for a particular device.
And if he subscribed to `COLLECTED_DATA/PeriodicMeasurement/<device_id>/` he would receive updates of all periodic measurements made by a particular device.

Kafka also has some limitations regarding the naming of topics, namely:

- The maximum length is 255 characters (or 249 as of Kafka 0.10)
- Only alphanumeric characters, dots ("."), underscores ("_"), and minuses ("-") are allowed

The main constraint we face is that, for our solution which uses **Oryx** (described later on in its own section), we need to setup two predefined Kafka topics for receiving input and communicating model updates, *"OryxInput"* and *"OryxUpdate"* respectively. As such, in order to simplify the bridging between the MQTT and Kafka brokers, MQTT will only use the same two topics as Kafka.

However, this solution itself raises one more question: which device and under which context, generated each message. To address this problem we need to re-utilize the same logic we would have used for the MQTT broker's subtopics and incorporate this information into the message itself. In Figure 6.6 we can see the class that both the Device Agents and the Cluster use to communicate with each other.

The following listing is a simplified JSON represenation of a DeviceMessage object sent from an Android Device Agent to the MQTT broker, which itself redirects it to the Cluster through the MQTT-Kafka bridge.

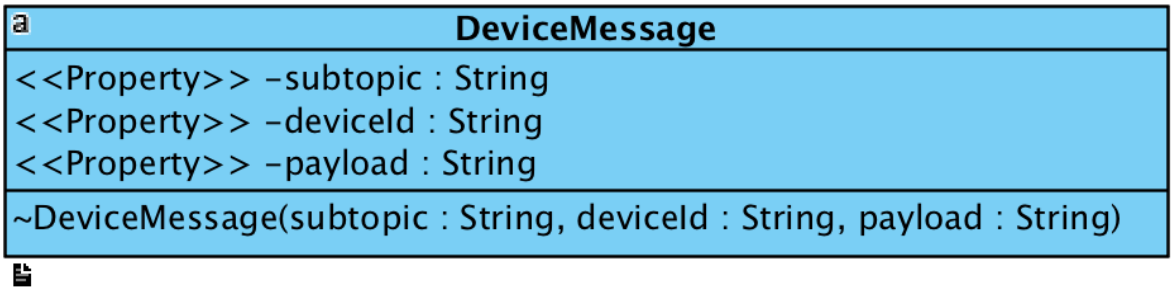


Figure 6.6: UML Diagram of Device Message class used to convey data between device agent applications and the cluster

```
{
   "deviceId":" - 000000000000000 - 89014103211118510720 - 8
   c240a5bf3a293e5",
   "payload":"[...]",
   "subtopic":"EventfulMeasurement"
}
```

Listing 6.5: JSON representation of a simplified DeviceMessage object

The **payload** is a character-escaped string that contains a list of measurements for the specified **subtopic**, serialized into a JSON string. The following text listing is an example of what the payload might contain once it is converted back into a pure JSON string.

```
[
   {
      "timestamp":"Jul 22, 2017 11:38:47 PM",
      "unitsOfMeasurement":"CID - LAC - PSC - NetworkType",
      "value":"0 - 0 - 0 - GSM - ",
      "id":1,
      "name":"BaseStationChange - Current"
   },
   {
      "timestamp":"Jul 22, 2017 11:38:47 PM",
      "unitsOfMeasurement":"CID - LAC - PSC - Latitude - Longitude -
   NetworkType - RSSI dBm",
      "value":"0 - 0 - 2147483647 - GSM - -113",
      "id":2,
      "name":"BaseStationChange - Neighbours"
   },
   {
      "timestamp":"Jul 22, 2017 11:38:47 PM",
      "unitsOfMeasurement":"GSM SignalStrength",
      "value":"12",
      "id":3,
      "name":"BaseStationChange - Current"
   }
]
```

Listing 6.6: JSON representation of a DeviceMessage's payload content

6.2.4 Encryption and Security

In the previous section we've already mentioned SSL/TLS and how it adds a security layer to HTTP. However, it isn't the only way to achieve secure encrypted messages, one other alternative to SSL/TLS is **(Full) Homomorphic encryption**, which is a form of encryption that allows for operations to be carried out on the cyphertext just as they would be on the plaintext, without loosing any meaning in the process. Although extremely useful, it is still very slow: "In late-2014, a re-implementation of homomorphic evaluation of the AES-encryption circuit using HElib, reported evaluation time of just over four minutes on 120 inputs, bringing the amortised per-input time to about 2 seconds" (Halevi 2014). Hence it is not suitable for our fast generation/consumption idea.

As far as security is concerned, there are three different approaches we can have with regards to this project:

- **Network Level** - Using a physically secure network or VPN as foundation for any communication between clients and broker is one way to provide a secure and trustworthy connection. This would be suitable for gateway applications, where the gateway is connected to devices on the one hand and with the broker over VPN on the other side.
- **Transport Level** - When the goal is to provide confidentiality in most cases TLS/SSL is being used for transport encryption. It provides a secure and proven way to make sure nobody can read along and even authenticate both sides, when using client certification authentication.
- **Application Level** - On the transport level it can be ensured that the communication is encrypted and the identity is authenticated. The MQTT protocol provides a client identifier and username/password credentials, which can also be used to authenticate devices on the application level. These properties are provided by the protocol itself. When it comes to authorisation or what each device is allowed to do, it lays in the hand of the broker implementation, how to handle it. Another possibility is to use payload encryption on the application level in order to make the transmitted information secure even without having a full fledged transport encryption.

To this end, adding a hash signature to each message could certify its integrity, as the aggregator would only have to check the signatures, which in conjunction with an SSL/TLS connection would suffice to avoid tampering and eavesdropping.

However, the most common implementations of SSL, namely OpenSSL, are too resource intensive for clients that aren't standard computers, such as mobile devices or microcontrollers. In fact, this is an already widely regarded issue, and as such, faster and lighter alternatives such as wolfSSL, bearSSL or matrixSSL were developed with these scenarios in mind. As of writing, matrixSSL seems the most enticing candidate as it was specifically made for IoT scenarios and is open source as well.

There is a common misconception surrounding the overhead that SSL/TLS brings to the transport layer, when in reality great effort has been put into optimizing it in recent years,

further blurring the line between encrypted and unencrypted communication. Moreover, with additional care upon setup for each case, SSL/TLS's handshakes can become the only apparent overhead if appropriate session and caching are customized.

Last, but certainly not least, there exist other alternatives to SSL-esque communication such as pure encryption of the message. A somewhat know example are TEA (Tiny/Trivial Encryption Algorithm) like mechanisms. While TEA is nowadays considered insecure, it's descendant XXTEA has been considered cryptographically-safe until the present day. The TEA family of algorithms is intended for devices with very little code storage space and that perform regular or constant data streaming, which could very well be a way to define this project: a system of data streaming devices.

This section addresses one of the **Behavioural Requirements** in Table 3.2, namely #**12**, which states that the Device Agent Application securely transfers data to the Central Aggregator. It also addresses some of the **Architectural Requirements** in Table 3.1, namely #**7** and #**8**, stating that secure connections are used by the system and that user anonymity is upheld.

6.2.5 Device Agent configuration management within MQTT

We've already analyzed how a Device Agent can send its collected data to the central aggregator, but as we've already mentioned in this project's objectives, the central aggregator is also supposed to be able to alter the behaviour of the device agents to some extent.

In order to propagate configuration updates from the Central Aggregator to the Device Agents we serialize **ConfigurationMessage** objects (Figure 6.7) and send them to the MQTT broker, where each Device Agent can find their respective updated configurations under the `Configurations/<device_ID>` topic.

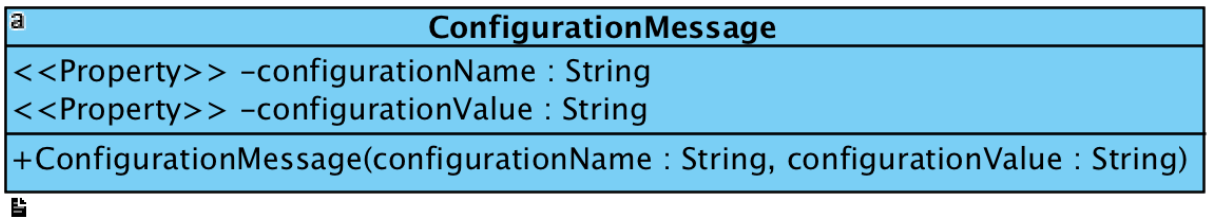


Figure 6.7: UML of the ConfigurationMessage class

If we wanted to change the frequency by which a device agent should aggregate it's periodic data, we could just send the following JSON-encoded ConfigurationMessage to its update subtopic. In this example we are stating that instead of attempting to aggregate the periodic data every 30 minutes (the default value), it should update and instead try every 45 minutes ($45 * 60 * 1000 = 2700000ms$).

```
{
  "configurationName":"pt.uc.student.aclima.device_agent.Aggregators.
   PeriodicAggregatorIntentService.action.AGGREGATE_PERIODIC_DATA",
  "configurationValue":"2700000",
}
```

Listing 6.7: JSON representation of a ConfigurationMessage's content

Tables 6.2, 6.3 and 6.4 list the existing configurations for Device Agents.

Table 6.2: Device Agent Configurations: Periodic Collection Actions

Configuration Name (ACTION_)	**Default Value (ms)**	**Description**
RAM	60 000	Inspect device's RAM usage every minute
CPU	60 000	Inspect device's CPU usage every minute
GPS	60 000	Inspect device's GPS location every minute
CPU_USAGE	10 000	Inspect application's CPU usage every 10 seconds
RAM_USAGE	10 000	Inspect application's RAM usage every 10 seconds
BATTERY	300 000	Inspect device's battery level every 5 minutes
OPEN_PORTS	60 000	Inspect which ports are open for communications every minute
OPEN_DATA_TRAFFIC	300 000	Inspect amount of bytes received and sent by device every 5 minutes

[1]Redacted for security purposes.
[2]See footnote 1.
[3]See footnote 1.

Table 6.3: Device Agent Configurations: Other Actions

Configuration Name (ACTION_)	**Default Value (ms)**	**Description**
AGGREGATE_PERIODIC_DATA	1 800 000	Aggregate the collected periodic data every 30 minutes
AGGREGATE_EVENTFUL_DATA	1 800 000	Aggregate the collected eventful data every 30 minutes
PUBLISH_DATA	3 600 000	Attempt to send the collected data to the MQTT server every 60 minutes
UPDATE_CONFIGURATIONS	3 600 000	Check for updated configurations sent from the MQTT server every 60 minutes

Table 6.4: Device Agent Configurations: Messaging

Configuration Name (MESSAGING_)	**Default Value**	**Description**
MQTT_TIMEOUT	10 (s)	The maximum time interval the client will wait for the network connection to the MQTT server to be established
MQTT_KEEP_ALIVE	10 (s)	The maximum time interval between messages sent or received for MQTT connections
SERVER_PROTOCOL	tcp	Protocol used to communicate with the MQTT server
SERVER_URI	[1]	MQTT server's URI or IP address
SERVER_PORT	[2]	Port used for communications with the MQTT server
SERVER_BASE_PUBLISH_TOPIC	"OryxInput"	MQTT topic used to publish device's collected data
SERVER_BASE_UPDATE_TOPIC	"Configurations"	MQTT topic used device subscribes to and listens for configuration updates
SERVER_PASSWORD	[3]	MQTT server password

In order to update these configurations and their respective alarms we follow the flowchart in Figure 6.8, which is tightly coupled with the flowchart already described for event management in Figure 6.2. At startup, the device agent fetches its alarm configurations and schedules an alarm for each of them. One of these alarms is the **Update Alarm**, which is responsible for triggering an attempt to connect to the MQTT server and check for configuration updates. If there are any, the device agents updates his configurations, and if an updated configurations is related to an alarm, we must first cancel it and schedule a new alarm with the updated configuration's value.

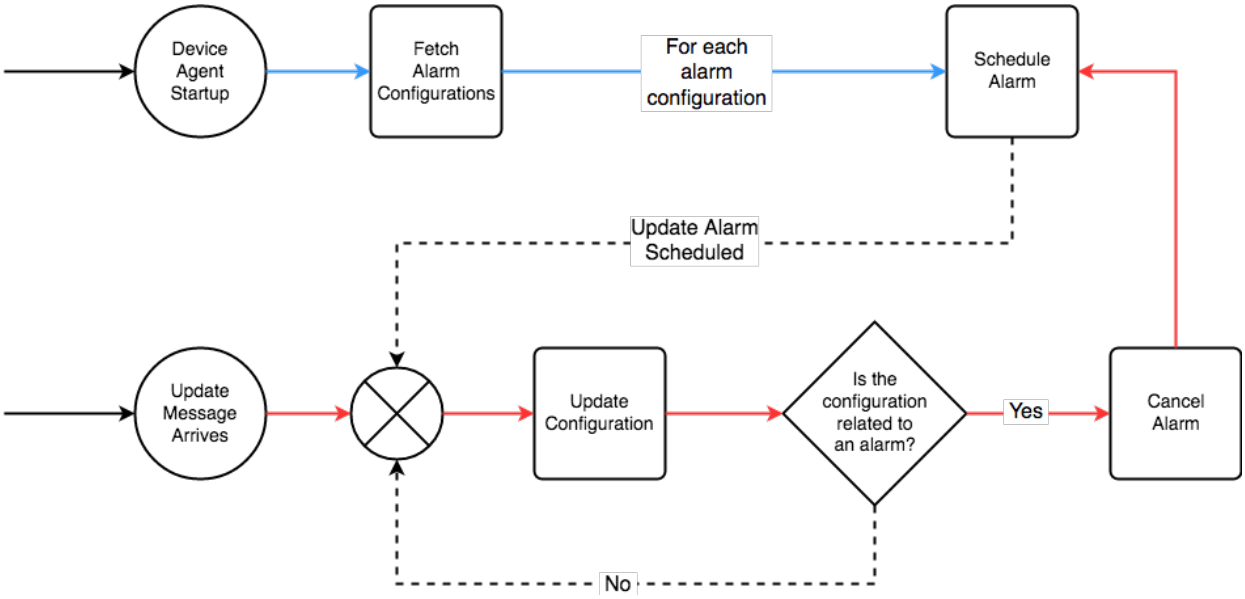


Figure 6.8: Update Management Flowchart

6.3 Aggregator

6.3.1 Hadoop and Spark

Our aggregator needs to be able to manage data originating form multiple sources, perform operations on new and on previously collected data. What we are looking for is a **Big Data** framework, and the most prominent contenders are Apache's Hadoop and Apache's Spark.

Figure 6.9: Hadoop logo

Hadoop is almost synonymous with Big Data, as it was one of the first and most successful frameworks to tackle its challenges. It is better know for its processing tool **MapReduce** and its file manager **HDFS** (Hadoop Distributed File System). It operates in a very simple fashion by processing data in batches, splitting it into smaller processing jobs spread across a cluster, and later combines the results.

(a) HDFS logo

(b) MapReduce logo

Figure 6.10: Hadoop components: MapReduce and HDFS

Spark is different from Hadoop in several ways, with the most obvious ones being that it works **in-memory**, potentially speeding up processing times, and that it doesn't have a file manager out-of-the-box. Spark also circumvents the imposed linear data flow offered by MapReduce, allowing for a more **customizable pipeline**.

Before one makes any rash decisions, it should be noted that Hadoop and Spark are not incompatible (in fact, they are both part of the Apache Foundation), meaning that many hybrid architectures can be created and tailored for each case. It is exactly because of this that we have decided to employ a mix of Hadoop's tools, namely HDFS, and Spark, effectively having the best of both frameworks.

Figure 6.11: Spark logo

This section addresses one of the **Architectural Requirements** in Table 3.1, namely #**3**, stating that the Central Aggregator runs Apache Spark.

6.3.2 Hadoop, Zookeeper, and YARN

Figure 6.12: Zookeeper logo

Zookeeper is a crucial part of any large scale Hadoop based system, as its job and purpose, although often unnoticed, is to act as a centralized service for maintaining configuration information, providing naming, distributed synchronization, leader election, message queues, notifications, and group services between components. More specifically, it serves as a standard linchpin to connect truly distributed systems with little or nothing in common by reliably coordinating their processes, while also taking care of storing, managing and propagating updates and data.

As is stated by the CAP theorem, "You can have at most two of these properties for any shared data system" referring to the near impossibility of achieving all three of **Consistency**, **Availability** and **Partition Tolerance**.

1. Consistency: Data is not discrepant across all parts of the system.

2. Availability: Each request is guaranteed to generate a response.

3. Partition Tolerance: When faced with broken or compromised communications between some nodes the remaining nodes works as guaranteed.

Zookeeper is no exception to this theorem, and as such it addresses this problem by sacrificing Availability in order to ensure Consistency and Partition Tolerance.

Figure 6.13: YARN logo

As is explained in Figure 6.14, **YARN** attempts to separate resource management and job scheduling/monitoring into two separate systems by having a global **Resource Manager** (RM) and **Application Master** (AM) for each application. An application is either a single job or a Directed Acyclic Graph (DAG) of jobs. while the Resource Manager and the Node Manager form the data-computation framework, the **Resource Manager** (RM) arbitrates resources among all the applications in the system. The **Node Manager** (NM) is responsible for the containers on each machine and monitoring their resource usage (CPU, memory, disk, network, etc.), which are reported to the Resource Manager/Scheduler. Each application's Application Master is tasked with negotiating resources from the Resource Manager and working with the Node Managers to execute and monitor the scheduled tasks.

As one can see, Zookeeper and YARN aim to shoulder the burden of managing resources across a very wide system, which is intended to be composed of several independent applications, each with it's own resource needs and purposes; moreover, client's task and job requests must be balanced between applications and within each application's containers. By using this standardized and efficient resource manager, one can focus on the really interesting and determining portions of large scale projects, rather than on guaranteeing that it will scale seamlessly and efficiently.

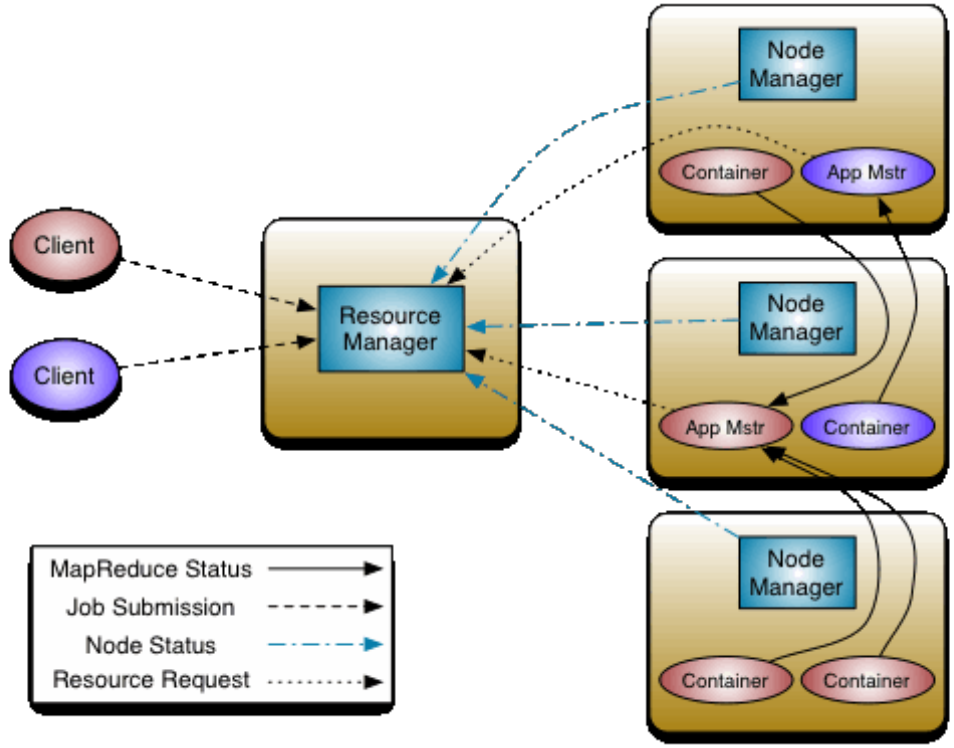


Figure 6.14: YARN Architecture
Source: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn_architecture.gif

6.3.3 Sequential and Batch processing

There are essentially two methods of processing Big Data, using Stream processing or Batch processing.

Stream processing is used to analyze small chunks of transferred data, usually by using sliding windows across data streams, and is especially useful for small, real-time computations. The analyzed data must contain timestamps for easy ordering and adjustment of the sliding window, which might vary from minutes to days.

Batch processing is used to process large blocks of data in one go, with little to no regard for the amount of time it takes, as some batch jobs might take hours or days to compute. This processing methodology is mainly employed when most or all of the data available for several entities is utilized for computation task.

Upon this analysis, we came to the conclusion that for this project it would benefit us more to not restrict the Central Aggregator to either batch or stream processing, but rather have both of these features. We will be using **Stream processing** mainly to have real-time updated information in the Dashboard, maintaining a viable human-in-the-loop architecture that provides enough feedback for a human supervisor to alter the Central Aggregator. Moreover, we will be using **Batch processing** for the Machine Learning layer of the Central Aggregator, analyzing the slew of collected data.

This decision to make due with the benefits of both processing methods is not novel, in fact it is called a **Lambda Architecture** (Bijnens 2017). A Lambda Architecture is an architecture that has two parallel layers of processing data, one for faster, real-time operations, and another for larger tasks, both connected to a data source layer and a query management layer. It tries to balance the latency and throughput of Big Data analysis with high demand for updated results. Figure 6.15 is an illustration of this concept.

Due to its focus on low-latency reads and updates, various Lambda Architecture example use cases have been proposed throughout the years, mainly for data analysis of IoT scenarios (Nierbeck 2016) or messages from vast social networks such as Twitter (Bertran 2013). This architecture has an almost endless number of applications because of the sheer number of existing tools and frameworks, their unique benefits and how they can be combined, all of this during a time where more data is generated than it can analyzed or consumed.

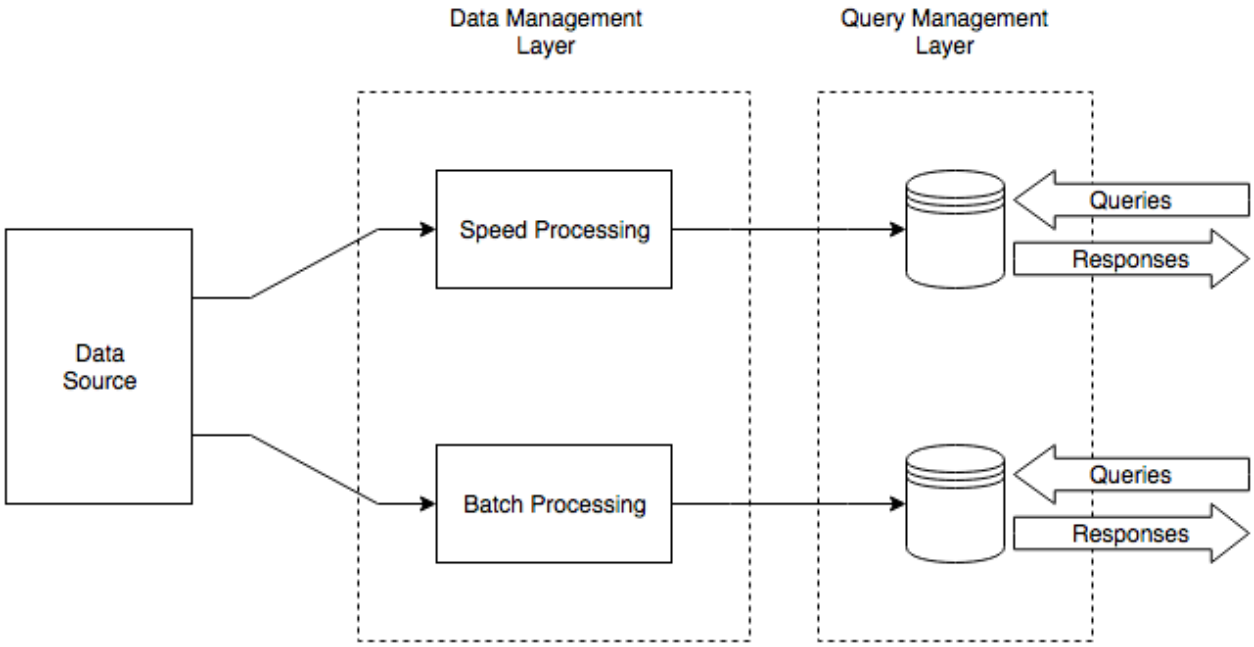


Figure 6.15: Example of a Lambda Architecture

6.4 Machine Learning and Data Visualization

6.4.1 Machine Learning

Figure 6.16: MLib logo

This project's value resides in its ambition to identify potential anomalies using Machine Learning. It is no coincidence that we have chosen to use Apache Spark in this project, as it has **MLlib**, a set of machine learning tools, including:

- **Dimensionality Reduction** - singular value decomposition (SVD), and principal component analysis (PCA), feature extraction and transformation functions
- **Classification** - logistic regression, naive Bayes, support vector machines (SVMs), decision trees, random forests, and gradient-boosted trees
- **Regression** - generalized linear regression, survival regression
- **Recommendation** - collaborative filtering, alternating least squares (ALS)
- **Clustering** - K-means, Gaussian mixtures (GMMs), K Nearest Neighbours
- **Topic Modeling** - latent Dirichlet allocation (LDA)
- **Statistics** - summary statistics, correlations, stratified sampling, hypothesis testing, random data generation
- **Miscellaneous** - Neural networks, frequent item sets, association rules, sequential pattern mining, stochastic gradient descent, limited-memory BFGS, etc.

As one can see, MLlib provides a very heterogeneous and large set of tools from which to pick and experiment with, leaving plenty of room for deciding and fine-tuning which is the most effective and relevant subset of tools for this projects data and expected results.

Thus far we are inclined to allow for the usage of the following tools, in the following order, and for the following purposes:

1. **Pre-processing** - Before any form of heavy processing or machine learning is used, we might want to sanitize or alter the used data. Extra caution has to be taken in this step because, since this is a **binary classification problem (anomalous VS normal)** the uniqueness of certain measurements might be crucial for the classification step.

- Outlier Removal
- Normalization
- Missing Value Interpolation

2. **Feature Selection** - Before any form of classification is done we want to be able to select which features are relevant

   - Summary statistics
   - Correlation Matrix
   - Average Values

3. **Feature Reduction** - Not all dimensions of the dataset have to be relevant for the classification step and can just hinder the performance of the classifier, as such, we will employ Principal component analysis (PCA) to reduce the number of considered features, using the Kaiser Criteria and Scree Test to determine the number of considered factors.

4. **Classification** - Because malware and other anomalies are constantly changing, we would have to regard this as an **Unsupervised Classification** problem and use suitable **Clustering** methods such as **K-means**. However, after some data has been collected we can label it with unsupervised classification and run the now labeled data through **Supervised Classification** methods such as:

   - SVM
   - Decision Tree
   - K Nearest Neighbours
   - Neural Network

   For this last part, we will employ **Triple Modular Redundancy** so that the final classification is the rounded median value of three distinct classification techniques, minimizing misclassification errors.

While training and/or developing a model classifier to fit and/or describe the data can be a heavy ordeal, taking anywhere from minutes, to hours, to days, depending entirely on the complexity of the chosen learning algorithm and the volume of data. It should be noted that the obtained trained and validated model can then be used and re-used to test and classify newer data in (near) real time.

These tools allow us to address some of the **Behavioural Requirements** in Table 3.2, namely #**14**, #**15**, #**16**, and #**17**, which state that machine learning and data analysis is employed on the collected data of each user and between users.

6.4.2 Oryx

Up until now we've already mentioned the need for tools or frameworks that are exceptional at certain tasks, namely: data transport, filesystem management, machine learning, and the ability to extrenuous and light tasks alike. Fortunately, this is no novel idea, and opensource projects such as Oryx exist (http://oryx.io/).

Oryx is a framework for building lambda architecture clusters written in Java, and uses Apache Spark, Hadoop, Tomcat, Kafka, Zookeeper and many other frameworks. It is designed to develop real-time large scale machine learning applications by providing out-of-the-box mechanisms to deploy a data transport mechanism as well as batch, speed and serving layers. Moreover, the variety of machine learning methods is provided by Apache Spark MLib's package.

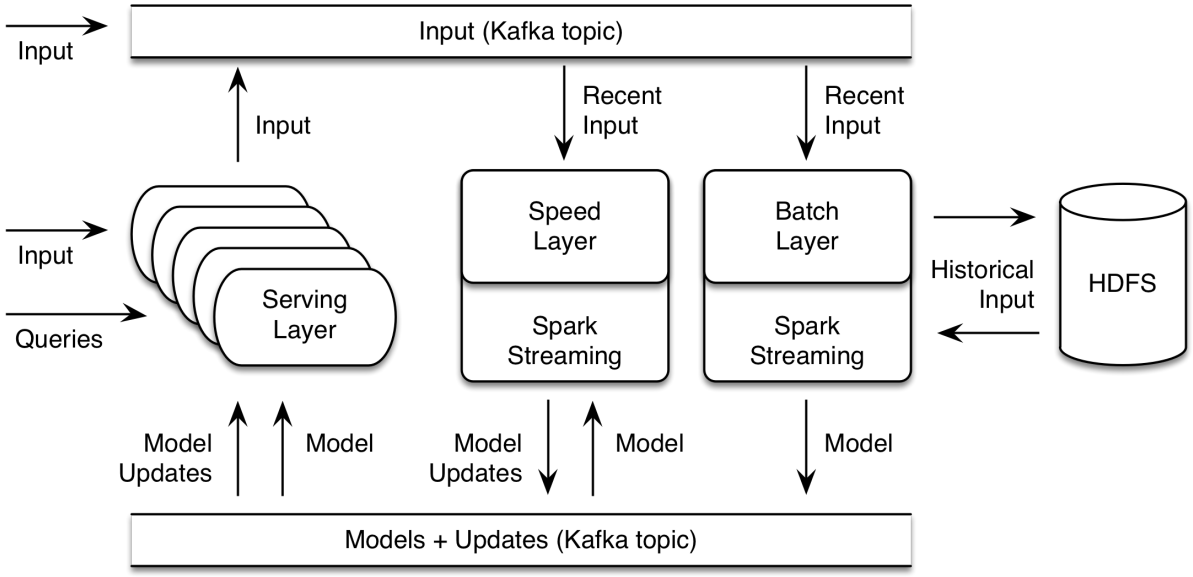


Figure 6.17: Oryx Architecture
Image source: http://oryx.io/img/Architecture.png

Since Oryx is a very complex framework, the least that can be done is explain its overall behaviour, as can be seen in Figure 6.17.

The data transport mechanism is comprised of Apache Kafka topics, used to communicate data to the speed and batch layers. These topics are also used to publish both models and model updates, for consumption by the speed and serving layers.

The batch layer is implemented as a Spark Streaming process on a Hadoop cluster, which reads data from the input Kafka topic. The current window of data is saved to HDFS, and is then combined with all previously stored data on HDFS. This data as a whole is then used to build a new result, which is written to HDFS and published to the update topic.

The speed layer is similar to the batch layer, but it periodically loads a new model from the update topic and continually produces model updates, instead of joining them with previous records before producing them, which results in much simpler models.

The serving layer listens for models and updates performed on them on the update topic, and can query them for changes or particular features.

6.4.3 Data Visualization

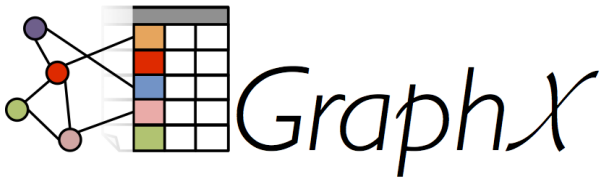


Figure 6.18: GraphX logo

Luckily enough, Spark also provides us with another very useful tool named GraphX, capable of generating immutable graphs for the existing data, which we intend to use after each batch of Machine Learning processes is executed. This will enable the Dashboard Users to analyze the system's current state and the collected data visually.

There are three fundamental aspects to the system we are proposing:

1. Massive data collection of device measurements
2. Intensive upload of collected data
3. Identification of anomalies through the collected data

Since the collected data is common to all of these items, it becomes obvious that we should be able to determine the relationship that each collected datum has with the device agents. As we've explained for Figure 6.6, and Listings 6.5 and 6.6, each collected datum is a Device Message Object which contains some measurements from a particular device agent, so the two are connected, and this is also true for all unique measurements. But we can go a step further, as each measurement falls into one of several types/categories of measurements. So we can connect Devices to their uniquely collected measurements, which in turn are connected to their more general type of measurement. Figure 6.19 is the visual representation of said connections in an **undirected weighed graph**.

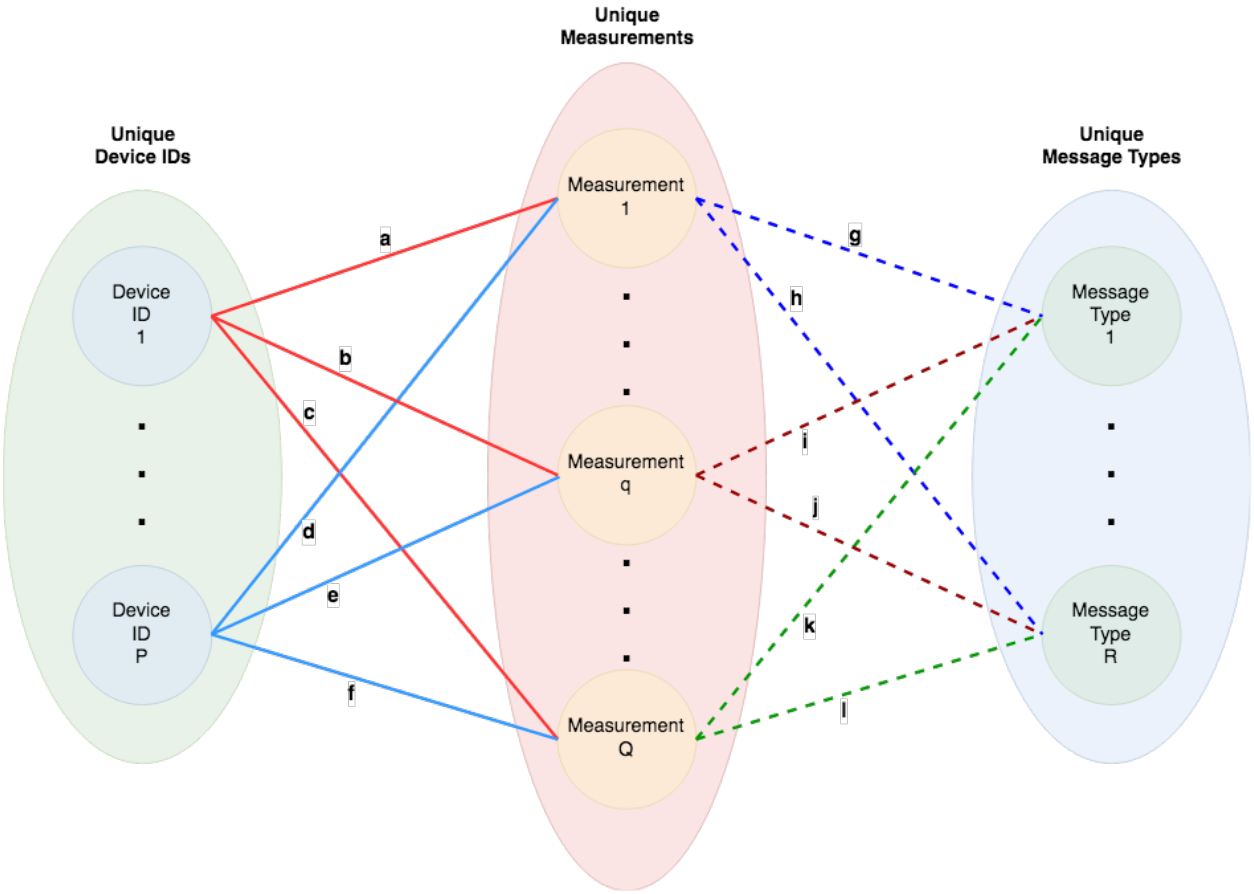


Figure 6.19: Device Message Graph Concept

Let's breakdown Figure 6.19:

- There are three node groups, each comprised of a variable number of nodes
- Each node on the **left node group** represents a Device Agent through it's unique Device ID
- Each node on the **middle node group** represents a unique measurement
- Each node on the **right node group** represents a type of message
- A Device Agent from the **left group** is connected to each of the unique measurements it produced in the **middle group**
- A unique measurement in the **middle group** is connected to each type it belongs to in the **right group**
- Each edge between the **left and middle node groups** has weight equal the amount of times the Device ID node has made that measurement
- Each edge between the **middle and right node groups** has weight equal the amount of times the that measurement has been collected in the system

With the rules for creating a Device Message Graph established, we can now proceed and interpret a hypothetical scenario such as the one in Figure 6.20.

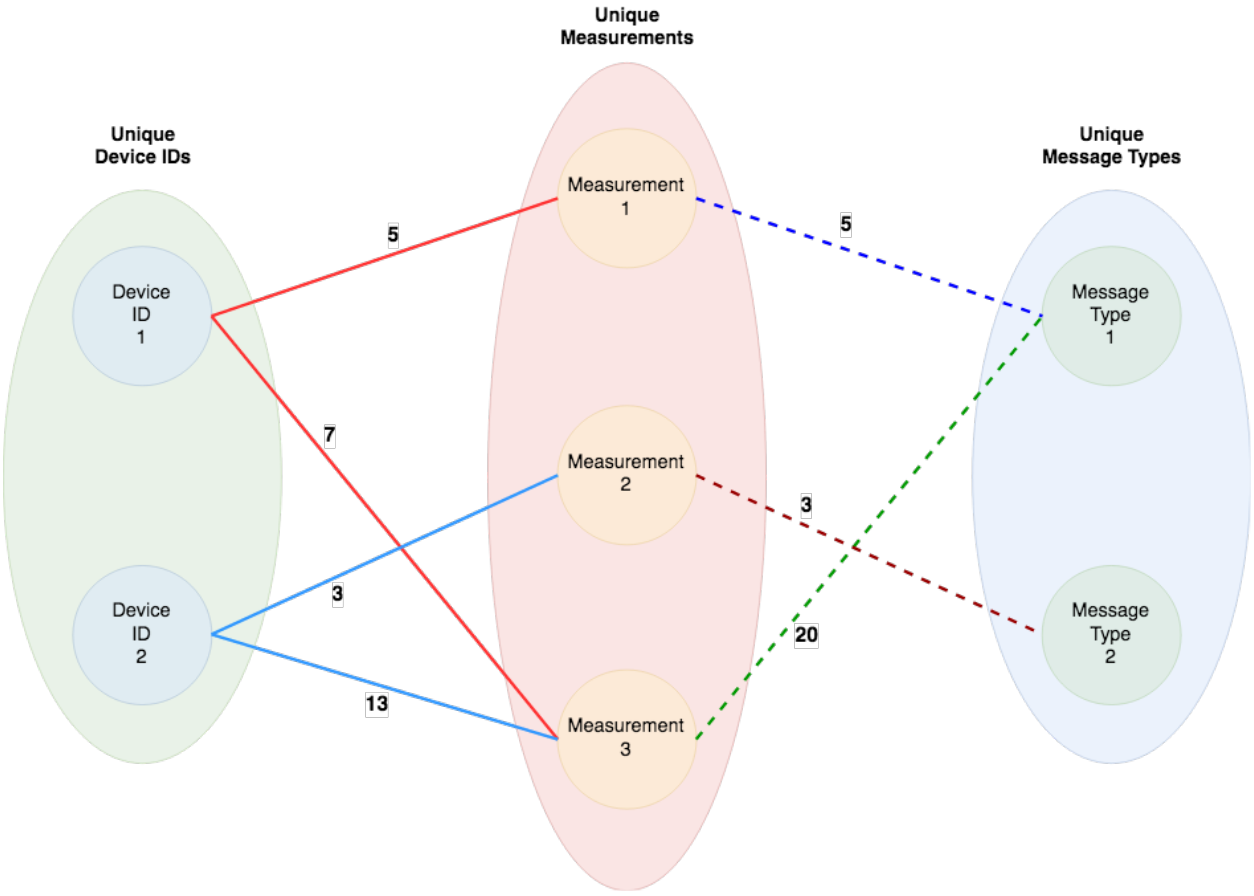


Figure 6.20: Device Message Graph Example

We can see in Figure 6.20 that in this scenario there are two distinct devices. **Device 1** has collected 5 measurements of **Measurement 1** and 7 of **Measurement 3** over time, whilst **Device 2** has collected 3 measurements of **Measurement 2** and 13 of **Measurement 3**. We can also verify that **Measurements 1 and 3** are classified as belonging to the **Message Type 1** type, of which a collective 5 and 20 measurements have been collected respectfully, and 3 measurements of **Measurement 2** have been collected in total, belonging to the **Message Type 2** type.

The main benefit of this graph representation is that it lets us quickly identify many aspects of the data by simply inspecting the connections between the node groups, namely:

- The amount of measurements
- The least common measurement
- The most common measurement

- The amount of measurements by type
- The least common type of measurement
- The most common type of measurement
- The amount of measurements collected by each device agent
- The device agent with the most amount of measurements collected
- The device agent with the least amount of measurements collected

This specific representation of such connections has the benefit of improving the readability and, thus, the capability for an end-user to detect any anomalies with more ease.

6.5 Dashboard

Although nowadays machine learning algorithms can perform feats that would take humans far too long or that are just impossible, the opposite can also be observed. The purpose of the dashboard is to apply the concepts held by human-in-the-loop simulation systems and Humanistic Intelligence to the management and analysis of the system results, by enabling human supervisors to interact with the system in real time, with hopes of improving the feedback loop. This can be summarized in Figure 6.21.

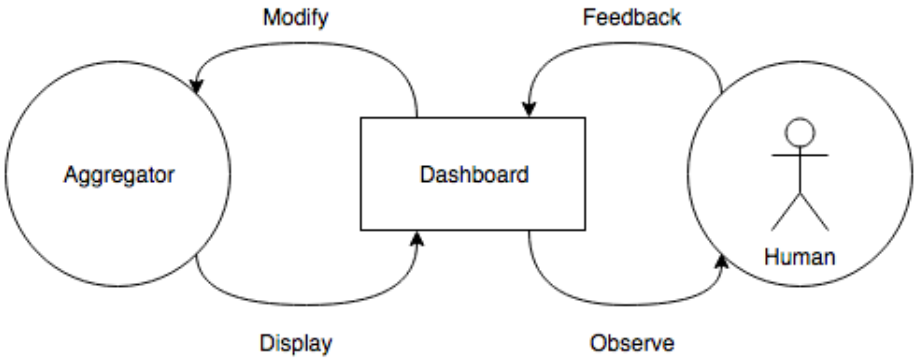


Figure 6.21: Interaction between Human, Dashboard and Aggregator

Machine Learning is powerful, but sometimes direct data analysis can be just as effective and less costly, both in terms of resources and time, depending mainly on the complexity of the conclusions or indicators one is searching for. With this in mind, we intend to monitor several aspects of the collected data for each user and for all users:

- Average entries
- Most common entries
- Least common entries

These metrics might help the user of the Dashboard to identify compromised devices that the Central Aggregator's Machine Learning component has missed, or simply help them adjust it. This approach is named **Exploratory Data Analysis** (EDA) and its purposes are suggesting of hypotheses about the causes of observations, assessing assumptions for statistical inference, supporting the selection of appropriate statistical tools and techniques, and providing a basis for further data collection. EDA attempts to retrieve information from data beyond the formal modeling or hypothesis testing task, usually with the help of visual aids such as plots and graphs, easing the identification of outliers, trends and patterns in data that might have otherwise gone unnoticed.

This section addresses some of the **Behavioural Requirements** in Table 3.2, namely #**18** and #**19**, which state that the Dashboard displays user activity, the collected data, and the findings of the Central Aggregator.

Chapter 7

Development and Deployment

7.1 Technologies, Tools and Frameworks

This section is dedicated to describing the list of the significant technologies, tools, and frameworks that we used to make this project along with their respective official sources:

1. Android - https://www.android.com/index.html
2. CentOS - https://www.centos.org/
3. Git version control - https://git-scm.com/
4. Github version control - https://github.com/
5. GraphX, Apache - https://spark.apache.org/graphx/
6. Hadoop, Apache - https://hadoop.apache.org/
7. Java - https://www.java.com
8. Kafka, Apache - https://kafka.apache.org/
9. Maven, Apache - https://maven.apache.org/
10. Mosquitto project - https://mosquitto.org/
11. MQTT organization - http://mqtt.org/
12. Oryx project - https://oryx.io/
13. Paho project - https://eclipse.org/paho/
14. Scala - https://www.scala-lang.org/
15. Spark MLib, Apache - https://spark.apache.org/mllib/
16. Spark, Apache - https://spark.apache.org/

17. YARN, Apache - https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

18. Zookeeper, Apache - https://zookeeper.apache.org/

A great portion of this project could not be possible without the structure that version control and project management systems provide, especially when several multi-version tools need to be combined and maintained. For Version Control we used **git** and **GitHub** whilst we used **Maven** to manage project dependencies.

As for the Android Application, its codebase is written in **Java**, as this is the officially supported **Android** language, even though there are alternatives such as Kotlin. In order to get the information produced by the device agents to the central aggregator, we use **MQTT** through a framework called **Paho**.

As for the Central Aggregator, it is a cluster comprised of several tools. Firstly, it runs on a Linux distribution of **CentOS**. In order to run the other tools it contains, it also has **Java** and **Scala** installed. The cluster is a fork of **Oryx**, which is a collection of several frameworks such as **Spark**, **Hadoop**, and **Zookeeper**, which having a crucial role. Regarding messaging, the cluster hosts two brokers, one for **Kafka** and another for **MQTT** through a framework called **Mosquitto**.

7.2 Device Agent

The codebase for the Device Agent's Android application can be found at https://github.com/aclima93/Terminal.git. There are two ways of deploying the application to an Android device, (all of which are detailed in the official documentation https://developer.android.com/training/basics/firstapp/running-app.html) and they can be applied to both real and emulated devices:

1.
   - Connecting the Android device to the computer.
   - Using Android Studio, build the application.
   - Using Android Studio, run the built application on the connected device by setting it as the intended target.
   - Wait for the aplpication to be installed.

2.
   - Connecting the Android device to the computer.
   - Using Android Studio, build the application.
   - Copy the generated Android Application Package (APK) file located at `Terminal/app/build/outputs/apk/app-debug.apk` to the target device.
   - Install the application by double tapping it on the target device.

Any Android application is supposed to be well summed up by its manifest, and ours is no exception. This particular, single activity, application is quite greedy as it requires the user to grant permission for several tasks (some of which are actually discouraged by Google's guidelines for regular applications):

- Internet access
- Automatic launch after device boot-up
- Storage read and write permissions
- Location services
- Detailed information regarding the device's hardware
- Detailed information regarding other running applications

```
<!-- Application's Tasks Information -->
<uses-permission android:name="android.permission.GET_TASKS"/>

<!-- Internet permission for network communication -->
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

<!-- Boot Loading Completion -->
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"
    />

<!-- Storage access -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>

<!-- Device location services -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"
    />

<!-- Cellular network information -->
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

Listing 7.1: Android Manifest: Permissions

After the list of permissions, Since this application was design to gather data, it is not surprising that it is comprised of a single **Activity** whose sole purpose is to launch the provided services and initialize the local database; the scheduled event services are the core functionality of this application.

```
<!-- Activities -->
<activity android:name="pt.uc.student.aclima.device_agent.
    DeviceAgentActivity">
     <intent-filter>
         <action android:name="android.intent.action.MAIN"/>
```

```
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>

<receiver android:enabled="true" android:name="pt.uc.student.aclima.
   device_agent.BootUpReceiver"
          android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
        <action android:name="android.intent.action.QUICKBOOT_POWERON" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</receiver>
```

Listing 7.2: Android Manifest: Activity and BootUpReceiver

The list of services provided by the application is comprised of the periodical, eventful, and one-time collector services; the periodical and eventful aggregation services; and finally the MQTT publishing service. The **Event Data Collector** is a very complete example of just how versatile the intent-filtering process can be, with this one listing for system time changes, connectivity changes, charging and battery status, and all of these intent-actions are broadcasted automatically by Android.

```
<!-- Eventful Data Collector -->
<service
    android:name="pt.uc.student.aclima.device_agent.Collectors.
   EventfulDataCollector.EventfulIntentService"
    android:exported="false">
</service>
<receiver
    android:name="pt.uc.student.aclima.device_agent.Collectors.
   EventfulDataCollector.EventfulBroadcastReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <!-- Time Changes -->
        <action android:name="android.intent.action.TIME_SET"/>
        <action android:name="android.intent.action.DATE_CHANGED"/>
        <action android:name="android.intent.action.TIMEZONE_CHANGED"/>

        <!-- Battery - Charging -->
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED
   "/>
        <action android:name="android.intent.action.
   ACTION_POWER_DISCONNECTED"/>

        <!-- Battery - Power -->
        <action android:name="android.intent.action.BATTERY_LOW"/>
        <action android:name="android.intent.action.BATTERY_OKAY"/>

        <!-- Connectivity -->
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />

```

```
        <!-- Power On/Off -->
        <action android:name="android.intent.action.ACTION_SHUTDOWN" />

    </intent-filter>
</receiver>
```

Listing 7.3: Android Manifest: Eventful Data Collector Service and Receiver

```
<!-- MQTT Service -->
<service android:name="org.eclipse.paho.android.service.MqttService">
</service>
```

Listing 7.4: Android Manifest: MQTT Service

7.3 Central Aggregator

Most of the tools we've already mentioned in Section 7.1 have very high systems requirements, namely Hadoop and Spark, which boast minimums of 8GB of RAM and enough disk space to cope with the expected data. As a proof-of-concept, we decided to host the Central Aggregator on a virtualized single node machine with the following specifications:

- **Operating System**: CentOS Linux release 7.2.1511 (Core)
- **CPU**: Intel(R) Xeon(R) CPU X5660 @ 2.80GHz
- **CPU Cache size**: 12MB
- # **CPU Cores**: 1 (single core)
- **RAM**: 16GB
- **Swap Memory**: 10GB
- **Disk Space**: 40GB

7.3.1 Oryx Cluster Setup

In order to setup the Oryx cluster properly we need to follow the official documentation. Since this cluster is a conglomerate of several other complex frameworks and tools, the choice of compatible versions between them is of paramount importance so as to guarantee that the cluster doesn't breakdown or behave unexpectedly.

For this cluster setup of Oryx 2.4.0 the following need to be configured:

- Java Runtime Environment (JRE) 8 or later
- Scala 2.11 or later
- A Hadoop cluster running the following components:

- Apache Hadoop 2.7.0 or later
- Apache Zookeeper 3.4.5 or later
- Apache Kafka 0.10 or later
- Apache Spark 2.1.0 or later

In order to install Java Runtime Environment 8 we can follow the instructions provided by (Oryx Cluster Setup 2017a, 2017b). The first step is downloading from Oracle a package containing Java .

```
$ wget --no-cookies --no-check-certificate --header "Cookie: gpw_e24=http
   %3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie" "
   http://download.oracle.com/otn-pub/java/jdk/8u141-b15/336
   fa29ff2bb4ef291e347e091f7f4a7/jdk-8u141-linux-x64.tar.gz"
$ tar xzf jdk-8u141-linux-x64.tar.gz
```

Listing 7.5: Java: Installation

Last but by no means least, we need to define the path variables associated with Java in the `~/.bashrc` file. If this step isn't performed, some of the tools in the following sections cannot determine where java has been installed.

```
$ export JAVA_HOME=/opt/jdk1.8.0_141
$ export JRE_HOME=/opt/jdk1.8.0_141/jre
$ export PATH=$PATH:/opt/jdk1.8.0_141/bin:/opt/jdk1.8.0_141/jre/bin
```

Listing 7.6: Java: ∼/.bashrc path variables

In order to apply these changes one can execute the following command: `source ~/.bashrc` .

Having installed JRE 8, we can proceed to install Scala 2.11. The reason we first needed to install Java is that Scala code is meant to be compiled to Java bytecode and ran on the Java Virtual Machine (JVM). Scala and Java can reference libraries from each other seamlessly, which is quite favorable to Java as Scala was created to make use of many functional programming features such as type inference and lazy evaluation, more so than Java. As such, Scala libraries make up for a lot of the heavy work that our cluster is meant to perform, namely regarding the Spark framework.

In order to install Scala we can just fetch a package from a trusted source and installing it on the server:

```
$ wget http://downloads.lightbend.com/scala/2.11.8/scala-2.11.8.rpm
$ sudo yum install -y scala-2.11.8.rpm
```

Listing 7.7: Scala: Installation

7.3.2 Deploying Hadoop

The first real portion of the Oryx cluster starts by deploying the base Hadoop cluster. We have chosen to use Hadoop 2.8 and to follow the tutorial provided by (Deploying Hadoop 2017), in order to setup a simple single node example cluster which can later be expanded/scaled for more nodes.

The first step to configuring Hadoop is to get a copy of it and installing it on the host machine.

```
$ wget http://www-eu.apache.org/dist/hadoop/common/hadoop-2.8.0/hadoop
   -2.8.0.tar.gz
$ tar xzf hadoop-2.8.0.tar.gz
$ mv hadoop-2.8.0 hadoop
```

Listing 7.8: Hadoop: Installation

As with Java, we also need to define some path variables for Hadoop to properly function across the machine.

```
export HADOOP_HOME=/home/hadoop/hadoop
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
```

Listing 7.9: Hadoop: ∼/.bashrc path variables

Now that we have a fresh installation, we can start configuring the Hadoop cluster. For that, we have to focus on three files located under `$HADOOP_HOME/etc/hadoop`:

- core-site.xml
- hdfs-site.xml
- mapred-site.xml
- yarn-site.xml

Since HDFS is literaly Hadoop's File System, these files define where each

Starting with `core-site.xml`, we define the URL that can be accessed to check HDFS's status.

```
<configuration>
<property>
  <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
</property>
</configuration>
```

Listing 7.10: Hadoop: Content of core-site.xml

In `hdfs-site.xml` we are stating that we want the filesystem to use replication, and specifying the location of the name and data nodes.

```
<configuration>
<property>
 <name>dfs.replication</name>
 <value>1</value>
</property>

```

```
<property>
  <name>dfs.name.dir</name>
    <value>file:///home/hadoop/hadoopdata/hdfs/namenode</value>
</property>

<property>
  <name>dfs.data.dir</name>
    <value>file:///home/hadoop/hadoopdata/hdfs/datanode</value>
</property>
</configuration>
```

Listing 7.11: Hadoop: Content of hdfs-site.xml

The file `mapred-site.xml` specifies MapReduce formulas and parameters, specifically, we are configuring MapReduce to use the frameworks provided by YARN.

```
<configuration>
 <property>
  <name>mapreduce.framework.name</name>
   <value>yarn</value>
 </property>
</configuration>
```

Listing 7.12: Hadoop: Content of mapred-site.xml

In `yarn-site.xml` we can find configuration information that overrides the default values for YARN parameters, namely we're interested in specifying the location of the resource manager (localhost), and we are also disabling checks for virtual memory availability.

```
<!-- disabling the virtual memory check, so that the application master
    does not get precariously terminated -->
<property>
 <name>yarn.nodemanager.vmem-check-enabled</name>
 <value>false</value>
</property>

<!-- Fixing ResourceManager connection issues -->
<property>
    <name>yarn.resourcemanager.address</name>
    <value>127.0.0.1:8032</value>
</property>
<property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>127.0.0.1:8030</value>
</property>
<property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>127.0.0.1:8031</value>
</property>
```

Listing 7.13: Hadoop: Content of yarn-site.xml

Now that we have configured the bare minimum, we can start the cluster.

```
# hadoop's working command directory
$ cd $HADOOP_HOME/sbin/

```

```
# start the hadoop cluster file system
$ ./start-dfs.sh

# start yarn, the resource manager
$ ./start-yarn.sh
```

Listing 7.14: Hadoop: start HDFS and Yarn servers

7.3.3 Deploying Zookeeper

Next we install Zookeeper 3.4.10 by getting its package from a trusted source, such as apache's mirrorring server, unpackaging it, and starting the Zookeeper server.

```
# download the package and unpackage it
$ cd /opt
$ wget http://apache-mirror.rbc.ru/pub/apache/zookeeper/zookeeper-3.4.10/
   zookeeper-3.4.10.tar.gz
$ gunzip -c *gz | tar xvf -

# zookeeper directory
$ cd /opt/zookeeper-3.4.10/

# start zookeeper server
$ bin/zkServer.sh start
```

Listing 7.15: Zookeeper: Installation and server start

7.3.4 Deploying Kafka

Kafka also requires Java to already be installed, and Zookeeper must also bee installed. We are going to use Kafka 10.2.1 and proceed by following similar steps to the ones mentioned in (Deploying Kafka 2017a, 2017b). First, as usual, we download the Kafka package and unpack it.

```
# download and unpack the package
$ wget http://www-us.apache.org/dist/kafka/0.10.2.1/kafka_2.12-0.10.2.1.
   tgz
$ tar xvzf kafka_2.12-0.10.2.1.tgz --strip 1

# kafka directory
$ cd /opt/kafka_2.12-0.10.2.1/
```

Listing 7.16: Kafka: Installation

While having the Zookeeper server running, we can start a Kafka server with one SSH session.

```
$ bin/kafka-server-start.sh config/server.properties
```

Listing 7.17: Kafka: Start a broker

If it is successful a similar output will appear on the console:

```
    INFO [Kafka Server 0], started (kafka.server.KafkaServer)
```

Listing 7.18: Kafka: Successfully started broker

On another SSH session we can create a kafka topic named `<TOPIC_NAME>`.

```
$ cd /opt/kafka_2.12-0.10.2.1
$ bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-
    factor 1 --partitions 1 --topic <TOPIC_NAME>
```

Listing 7.19: Kafka: Topic creation

We can immediately test our Kafka broker by creating another SSH session and producing messages for the kafka topic named `<TOPIC_NAME>`.

```
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic <
    TOPIC_NAME>
```

Listing 7.20: Kafka: Producer

And on yet another SSH session we can list the messages that have been delivered for the kafka topic named `<TOPIC_NAME>`.

```
$ cd /opt/kafka_2.12-0.10.2.1
$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic <
    TOPIC_NAME> --from-beginning
```

Listing 7.21: Kafka: Consumer

We can also inspect the Kafka server's list of existing topics.

```
$ bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Listing 7.22: Kafka: List existing topics

7.3.5 Deploying Spark

In order to install Spark 2.1.1 we must already have Java and Scala installed on the target machine, which by this point should be an issue. Next we can follow similar instructions to those detailed in (Deploying Spark 2017).

```
# download package and unpack it
$ wget http://d3kbcqa49mib13.cloudfront.net/spark-2.1.1.tgz
$ tar xf spark-2.1.1.tgz

# spark directory
$ mv spark-2.1.1 /opt/spark-2.1.1
$ cd /opt/spark-2.1.1/
```

Listing 7.23: Spark: Installation

Unlike the previous tools we've mentioned, we must now build Spark before we can use it. Luckily, Spark comes bundled with a self-assembly file which we can run with the help of **sbt**, Scala's de facto build tool, and Maven, a project management tool.

```
$ ./opt/spark-2.1.1/build/sbt assembly
$ cd /opt/spark-2.1.1/
$ build/mvn -Pyarn -Phadoop-2.8 -Dhadoop.version=2.8.0 -DskipTests clean
    package
```

Listing 7.24: Spark: Assembly

After Spark is done assembling itself, we can finally run its shell

```
$ ./bin/spark-shell
```

Listing 7.25: Spark: Launch a Spark shell

and if no problems were met, one should expect a similar output to the following:

```
$ ./bin/spark-shell
Spark context Web UI available at http://10.3.2.9:4040
Spark context available as 'sc' (master = local[*], app id = local
    -1501166979903).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0
    _121)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Listing 7.26: Spark: Successful launch output

If all of this is true, we have command line interface with which we can interact directly with Spark's components, namely access its Resilient Distributed Datasets (**RDD**) and perform computations with them.

7.3.6 Deploying Oryx Server Layers

We can finally get back to the Oryx server installation now that all the necessary tools and frameworks have been successfuly installed. The first real step is to edit `lambda_app.conf` with our Oryx Server configurations, of which we have particular interest in specifying the location of the Kafka brokers, the zookeeper servers, and the HDFS server.

This is also where we can deifne the codebase that Oryx should use, in our case it's the default "com.cloudera.oryx.app" We are going to use two Kafka brokers (**"OryxInput"** and **"OryxUpdate"**) to manage input data sent to the system, and updated data within the system.

We can also change the specifications of the batch, speed and serving layers, namely, which classes/codebase they should consider for their respective operations, where to store

data, and where to store the models produced by the system to describe the amassed data.

For this project, and following up on the previous deployment steps in this section, we are only going to specify one Kafka broker and one Zookeeper server on the same machine (localhost) as the Oryx cluster itself, but as one can see this can easily be scaled for more and independent machines.

```
# Reusable values for the configurations #
hostname = "oryx.aclima.dei.uc.pt"
kafka-brokers = ${hostname}":9092"
zk-servers = ${hostname}":2181"
hdfs-base = "hdfs:///user/lambda-app/Oryx"
base-package = "com.cloudera.oryx.app"

# Configuration of the app #
oryx {
  id = "LambdaApp"

  # Configuration for the Kafka input topic #
  input-topic {
    broker = ${kafka-brokers}
    lock = {
      master = ${zk-servers}
    }
  }

  # Configuration for the Kafka update topic #
  update-topic {
    broker = ${kafka-brokers}
    lock = {
      master = ${zk-servers}
    }
  }

  # Configuration for the Batch Layer #
  batch {

    # Streaming Configurations #
    streaming {
      generation-interval-sec = 300
      num-executors = 4
      executor-cores = 8
      executor-memory = "4g"
    }

    # Configuration of the class used for updating the k-means model #
    update-class = ${base-package}".batch.mllib.kmeans.KMeansUpdate"

    # Storage Configurations for the Data and Models #
    storage {
      data-dir =  ${hdfs-base}"/data/"
      model-dir = ${hdfs-base}"/model/"
    }
```

```
47
48      # UI Configurations #
49      ui {
50        port = 4040
51      }
52    }
53
54    # Configuration for the Speed Layer #
55    speed {
56
57      # Configuration of the class used for managing the k-means speed model
        #
58      model-manager-class = ${base-package}".speed.kmeans.
      KMeansSpeedModelManager"
59
60      # UI Configurations #
61      ui {
62        port = 4041
63      }
64    }
65
66    # Configuration for the Serving Layer #
67    serving {
68
69      # Configuration of the class used for managing the k-means serving
      model #
70      model-manager-class = ${base-package}".serving.kmeans.model.
      KMeansServingModelManager"
71
72      # Configuration of the packages used for managing the app's served
      resources #
73      application-resources = ${base-package}".serving,"${base-package}".
      serving.kmeans"
74
75      # API Configurations #
76      api {
77        port = 8080
78      }
79    }
80  }
```

Listing 7.27: Oryx: Content of lambda_app.conf

As we've done before, it helps if we edit the `~/.bashrc` file and add some path variables before we proceed.

```
1  # User specific aliases and functions
2
3  # kafka
4  export KAFKA_HOME=/opt/kafka_2.12-0.10.2.1
5
6  # spark
7  export SPARK_HOME=/opt/spark-2.1.1
8
9  # oryx
```

```
export ORYX_HOME=/opt/oryx

# zookeeper
export ZOOKEEPER_HOME=/opt/zookeeper-3.4.10

# hadoop
export HADOOP_HOME=/home/hadoop/hadoop
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native

# path env. variable
export PATH=$PATH:$KAFKA_HOME/bin:$SPARK_HOME/bin:$ORYX_HOME/bin:
   $ZOOKEEPER_HOME/bin:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
```

Listing 7.28: Oryx: ∼/.bashrc path variables

Now we are ready to run the Zookeeper Server and the Kafka Server, each on their own SSH instance.

```
$ sudo bin/zkServer.sh start
```

Listing 7.29: Oryx: Start Zookeeper server

```
$ sudo bin/kafka-server-start.sh config/server.properties
```

Listing 7.30: Oryx: Start Kafka server

Next, we can let Oryx automatically setup the Kafka input and update topics we specified earlier

```
$ bash oryx-run.sh kafka-setup
```

Listing 7.31: Oryx: Automatic setup of Kafka topics

which should result in an output similar to the following:

```
Input   ZK      <HOSTNAME>:2181
        Kafka   <HOSTNAME>:9092
        topic   OryxInput
Update  ZK      <HOSTNAME>t:2181
        Kafka   <HOSTNAME>:9092
        topic   OryxUpdate
```

We can even watch messages sent to these topics, and monitor the actions of the application by listening to the Kafka's message tail:

```
$ bash oryx-run.sh kafka-tail
```

Listing 7.32: Oryx: Kafka message tail

Finally, the core of this whole project's endeavour, launching the **Batch**, **Speed** and **Serving** layers of the application, each on their own SSH instance.

```
$ bash oryx-run.sh batch --app-jar lambda_app.jar
```

Listing 7.33: Oryx: Batch layer

```
$ bash oryx-run.sh speed --app-jar lambda_app.jar
```

Listing 7.34: Oryx: Speed layer

```
$ bash oryx-run.sh serving --app-jar lambda_app.jar
```

Listing 7.35: Oryx: Serving layer

7.3.7 Restarting Oryx Server Layers

Should something go wrong, or the need to roll-out an update arise it is our best interest to properly stop the running services, so in order to do that, we can run the following commands:

Restart the Zookeeper server.

```
$ cd /opt/zookeeper-3.4.10/
$ bin/zkServer.sh start
$ bin/zkServer.sh start
```

Listing 7.36: Oryx: Restart Zookeeper server

Restart Hadoop's HDFS and YARN systems.

```
$ cd $HADOOP_HOME/sbin/
$ ./stop-all.sh
$ ./start-all.sh
```

Listing 7.37: Oryx: Restart Hadoop's HDFS and YARN

Re-setup and restart Kafka brokers and topics.

```
$ cd /opt/oryx/
$ bash oryx-run.sh kafka-setup
```

Listing 7.38: Oryx: Re-setup and restart Kafka brokers and topics

Re-launch the Oryx cluster application layers.

```
$ bash oryx-run.sh batch --app-jar lambda_app.jar
```

Listing 7.39: Oryx: Restart Batch layer

```
$ bash oryx-run.sh speed --app-jar lambda_app.jar
```

Listing 7.40: Oryx: Restart Speed layer

```
$ bash oryx-run.sh serving --app-jar lambda_app.jar
```

Listing 7.41: Oryx: Restart Serving layer

For ease of use, we recommend using the included executable file `restart-all.sh`, which promptly restarts all subsyetms of the cluster.

```
$./restart-all.sh
```

Listing 7.42: Oryx: Restart the cluster

```
# restart zookeeper
cd /opt/zookeeper-3.4.10/
bin/zkServer.sh stop
bin/zkServer.sh start

# restart hadoop cluster (HDFS and YARN)
cd $HADOOP_HOME/sbin/
./stop-all.sh
./start-all.sh

# re-setup kafka brokers and Topics
cd /opt/oryx/
bash oryx-run.sh kafka-setup
```

Listing 7.43: Oryx: Content of restart-all.sh

7.4 Message Exchange

Now that the Device Agents and the cluster have been deployed, we need to deploy the means to connect one to the other. The device Agents use MQTT and the cluster uses Kafka to exchange data around, so what we are missing are an MQTT broker, and the bridge between the MQTT broker and the Kafka broker.

7.4.1 Deploying the MQTT Broker

We are going to follow some of the steps specified in (Deploying the MQTT Broker 2017a) in order to configure a Mosquitto MQTT broker. Mosquitto is an open source message broker that implements the MQTT protocol, is easy to deploy, and keeps true to the lightweight aspect of MQTT.

We start by installing Mosquitto.

```
$ sudo yum -y install epel-release
$ sudo yum -y install mosquitto
```

Listing 7.44: MQTT: Installing Mosquitto

The next stop is launching a MQTT broker, which can be achieve by simply calling the `mosquitto` command in the command-line.

```
$ mosquitto
```

Listing 7.45: MQTT: Launching a broker

However, this will start a broker with all of the default configurations as stated in the official documentation (Deploying the MQTT Broker 2017b) and the the default configuration file *mosquitto.conf* (Deploying the MQTT Broker 2017c). A broker can be started using other configuration settings as such

```
$ mosquitto custom_mosquitto.conf
```

Listing 7.46: MQTT: Launching a broker with a custom configuration

The ability to customize the broker's configuration settings is particularly useful to us since we allow Device Agents to use other ports than the default mosquitto ports, communication protocols and their intended purposes as evidenced by the official test server (Deploying the MQTT Broker 2017d):

- **1883**: MQTT, unencrypted
- **8883**: MQTT, encrypted
- **8884**: MQTT, encrypted, client certificate required
- **8080**: MQTT over WebSockets, unencrypted
- **8081**: MQTT over WebSockets, encrypted

The configuration file follows the structure depicted in Figure 7.1.

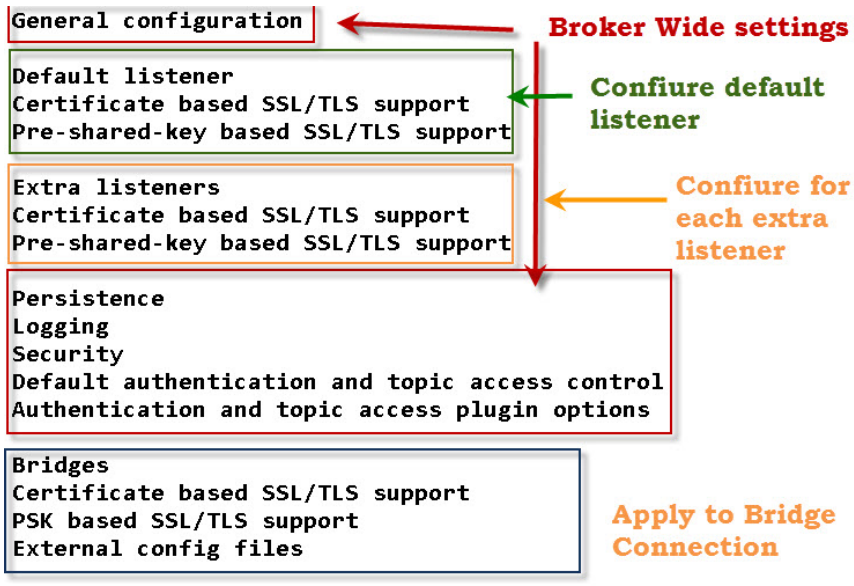


Figure 7.1: MQTT: Configuration file structure
Adapted from the original source: http://www.steves-internet-guide.com/wp-content/uploads/Mosquitto-conf-Structure.jpg

If the broker is successfully launched, we can now subscribe to a topic named `<TOPIC_NAME>`.

```
mosquitto_sub -h localhost -t <TOPIC\_NAME>
```

Listing 7.47: MQTT: Subscribe to a topic

Or we can publish the message `<CONTENT_STRING>` to a topic named `<TOPIC_NAME>`.

```
mosquitto_pub -h localhost -t <TOPIC_NAME> -m <CONTENT_STRING>
```

Listing 7.48: MQTT: Publish to a topic

7.4.2 Deploying the Bridge between MQTT and Kafka

Kafka is not easily available for Android, and is not lightweight enough, so we are sticking with MQTT. But that means we have to bridge the MQTT and Kafka brokers to get the best of both worlds. The source code for this bridge is available at https://github.com/aclima93/mqttKafkaBridge, and it is a fork of the original opensource project but updated enough to make it work at the time of writting, as the project was already several years old and its dependencies were outdated.

```
# clone the repository
$ git clone https://github.com/aclima93/mqttKafkaBridge
$ cd mqttKafkaBridge

# build the executable jar
$ mvn clean compile assembly:single
$ cd target
$ java -jar mqttKafkaBridge-0.1.0-jar-with-dependencies.jar
```

Listing 7.49: MQTT-Kafka bridge

7.5 Oryx Application

The codebase for the Oryx application is available at https://github.com/aclima93/oryx, which is a fork of the original open source project distributed by http://oryx.io/. The original project has a bundled example for a recommendation system, which we have refactored, adapted and expanded from in order to create our Lambda Architecture Application using Oryx's framework.

We can begin by downloading and compiling the codebase. Because Oryx is such a complex tool, a full installation requires several dozen tests and licensing verifications to be made, but this results in a building time that can take anywhere from a couple dozen minutes, to a couple of hours, depending on the available hardware and internet connection. As such, we recommend performing a clean installation without these cumbersome tests and checks.

```
# clone the repository
$ git clone https://github.com/aclima93/oryx
$ cd oryx/

# build the executable jars for the project, skip license checks, skip
    tests
$ mvn clean install -Drat.skip=true -DskipTests=true package
```

Listing 7.50: Oryx Lambda Application: Installation

A successful installation should produce an output similar to the following:

```
[INFO] ------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Oryx ............................. SUCCESS [  2.430 s]
```

```
[INFO] API ............................... SUCCESS [ 20.464 s]
[INFO] Common ............................ SUCCESS [  5.708 s]
[INFO] Kafka Utilities ................... SUCCESS [  3.606 s]
[INFO] Lambda ............................ SUCCESS [  4.323 s]
[INFO] Lambda Serving .................... SUCCESS [  3.899 s]
[INFO] ML ................................ SUCCESS [  3.829 s]
[INFO] Apps: API ......................... SUCCESS [  3.092 s]
[INFO] Apps: Common ...................... SUCCESS [  6.532 s]
[INFO] Apps: Spark MLlib ................. SUCCESS [  5.740 s]
[INFO] Apps: Oryx ........................ SUCCESS [  3.885 s]
[INFO] Apps: Oryx (Serving) .............. SUCCESS [ 33.934 s]
[INFO] Example ........................... SUCCESS [ 10.306 s]
[INFO] Lambda App ........................ SUCCESS [  6.543 s]
[INFO] Batch Layer ....................... SUCCESS [ 33.441 s]
[INFO] Speed Layer ....................... SUCCESS [ 23.876 s]
[INFO] Serving Layer ..................... SUCCESS [ 36.430 s]
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 03:28 min
[INFO] Finished at: 2017-07-23T12:28:53+01:00
[INFO] Final Memory: 148M/1515M
[INFO] ------------------------------------------------------------
```

Listing 7.51: Oryx Lambda Application: Successful build output

We have successfully installed and built the whole Oryx system, all that remains is to deploy it.

```
$ cd target
$ java -jar lambda-app-2.5.0-SNAPSHOT.jar
```

Listing 7.52: Oryx Lambda Application: Deployment

If any changes are made to the codebase of the Lambda Application and we want to re-deploy it we can merely rebuild this portion of the Oryx system and re-launch it, saving us several minutes. This can be achieved with the following commands:

```
$ cd oryx/app/lamda-app

# build the executable jars for the project, skip license checks, skip
   tests
$ mvn clean install -Drat.skip=true -DskipTests=true package
```

Listing 7.53: Oryx Lambda Application: Targeted rebuilding

If the building phase faces no issues, an output similar to the following will be produced:

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 11.409 s
[INFO] Finished at: 2017-07-23T12:30:38+01:00
[INFO] Final Memory: 56M/721M
[INFO] ------------------------------------------------------------
```

Listing 7.54: Oryx Lambda Application: Successful targeted re-building output

7.6 Machine Learning

Oryx has 3 out-of-the-box machine learning model management options from which any application can build from, each completely different and suitable for distict datasets and objectives. The three methods provided are:

- **ALS**: a collaborative filtering/recommendation system
- **KMeans**: a k-means clustering system
- **RDF**: a random decision forest classification system

We can't use the **ALS** system because it does not fit our problem's description, that is, we are querying the gathered data in order to assess if a given device should be classified as compromised, we are not asking the system to recommend us devices, much like a movie or book, based on our preferences. The **RDF** system is also inappropriate as it is a supervised learning method, meaning that we would need to firstly amass a large enough dataset or pre-classified samples, and only then train a model, which goes against this projects intent to address fast "Big Data"-esque growth. This leaves us with **K-Means**, which fits all of our criteria perfectly, being an unsupervised learning method, and having the number of clusters (K) that we wish to obtain already fixed at 2, compromised and uncompromised devices.

Although we had previously devised in Chapter 6 for there to be a **Triple Modular Redundancy** fault tolerant classification, it appears that Oryx's architecture was not designed with this in mind, as it keeps only one model in the Batch Layer, and this model is generated solely through the singular input and update Kafka topics, so it is limited to performing only one type of Batch & Speed computations on its model. However, this should not impede us from performing a **Triple Redundancy** classification, where we can simply re-execute the Speed Layer's job 2 additional times, and assume the most consensual binary classification.

We can summarize the intended Machine Learning behaviour with Figure 7.2, illustrating how the Triple Redundancy is integrated into the Speed Layer's flowchart and is absent from the Batch Layer.

So the machine learning portion of our system is going to use the K-Means clustering algorithm, and its underlying behaviour is very straightforward:

1. Create K initial random points for the data, these will act as the mean centroids for the first iteration
2. Create K clusters by assigning each data point to the nearest centroid
3. Calculate a new mean centroid for each of the K clusters
4. Repeat from step 2 until the calculated centroids/clusters converge

Since the algorithm is kickstarted from initially random centroids, it is entirely possible that only local optimal solutions are achieved through this heuristic algorithm. As the algorithm

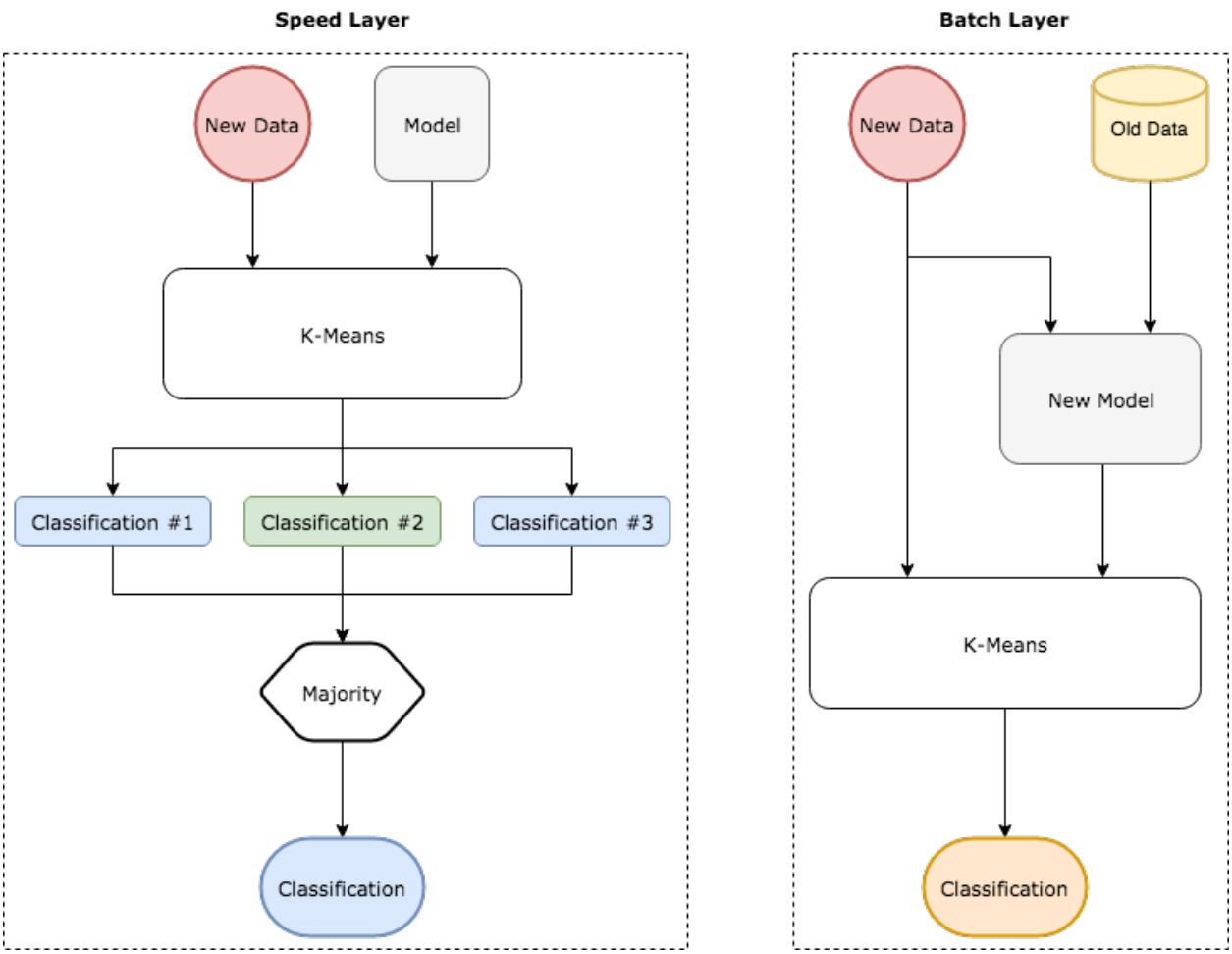


Figure 7.2: Speed and Batch layer's classification flowchart

is usually very fast, running in smooth polynomial time, it is common to run it multiple times with different starting conditions in order to determine the majority classification for each point, thus minimizing the chance that bad initial centroids will lead to bad clusters.

Usually, the distance between data points and the centroids is calculated using the Euclidean Distance or the Manhattan Distance, but other equally valid esoteric distance metrics such as the Mahalanobis Distance can be used, depending on the meaning or information that the distance should represent. Oryx uses the **Euclidean Distance** in its standard algorithm.

Figure 7.3 illustrates how the K-Means clustering algorithm separates samples into two clusters over time, using the Euclidian Distance between the data points and the centroids.

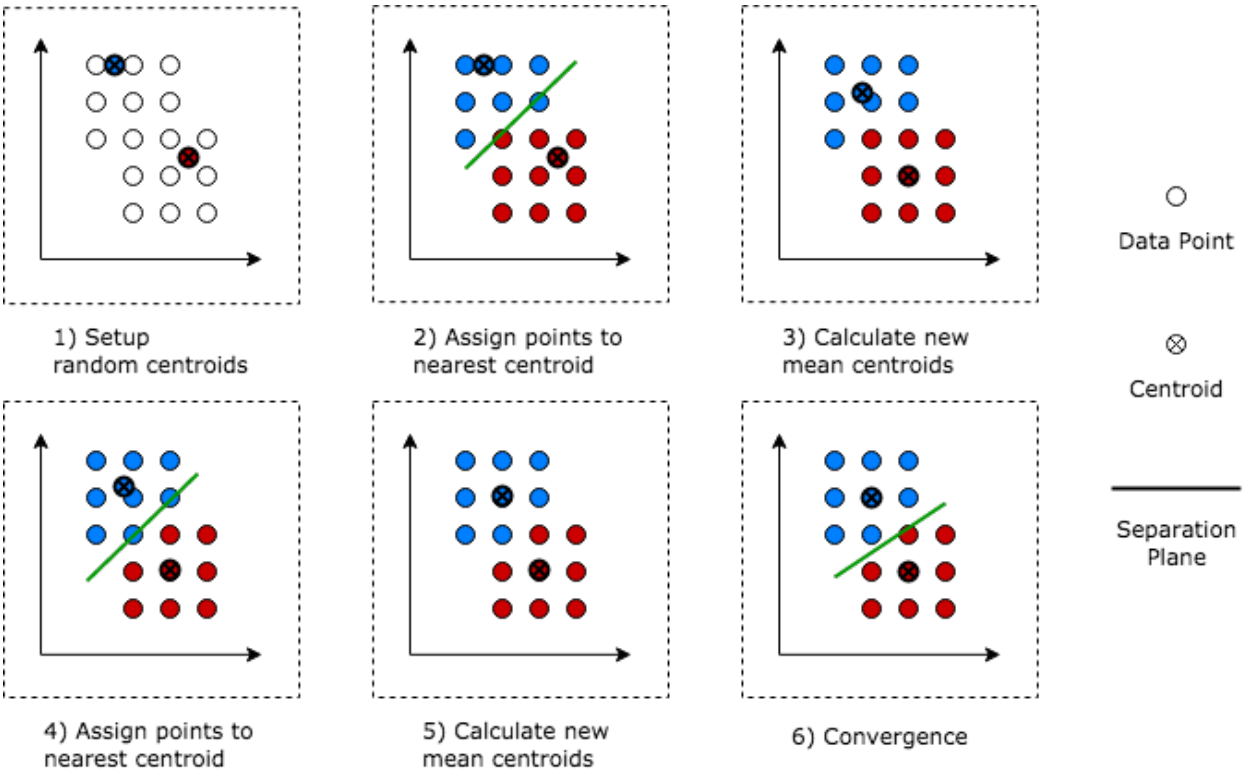


Figure 7.3: K-Means Example

Chapter 8

Preliminary Results Analysis

The following chapter details the testing and validation aspects envisioned for the proposed architecture. We have to keep in mind that the proposed architecture is intended to always keep running, be it the Central Aggregator or the Device Agents, thus some restrictions have to be placed on it, and there is no better way to gauge these restrictions than by testing each component.

8.1 Device Agent

Regarding the Device Agent, it is design and intended to constantly run in the background of the host mobile device, which by definition is expected to have limitations. Some of those limitations are the available CPU, RAM, battery life, Free Memory and Connectivity, thus we propose that the CPU and RAM overhead should be analysed in depth, and the rest could be analysed on a future iteration of this architecture, particularly because they are not as easily monitored.

Analysing the Device Agent's impact on the System's RAM and CPU, as can be seen in Table 8.1 and Figures 8.1, 8.2, 8.3, and 8.4, consisted of running our application on a standard, factory reset, Nexus 5 emulator for an approximate total of **6 hours**. During this period of time, we record the application's RAM usage every **10 seconds**, and the system's available RAM and CPU every **60 seconds**, culminating in an approximate total of **500 measurements** for each attribute during that period of time. Since these measurements were performed on a factory reset device, we had a greater control of the testing environment thus, aside from our application only a select few other default stock Android applications and utilities such as the clock, telephony, calendar, launcher were running simultaneously in the device, reducing the amount of "noise" in the system that could otherwise be present in a user's device.

Table 8.1: Device Agent test results analysis

	# **Measurements**	**Minimum**	**Average**	σ	**Maximum**
Application RAM (MB)	467	17756	18773	544	23521
System RAM (MB)	493	1365270	1383270	6361	1395916
Application CPU (%)	473	0	17.7	41.3	200
System CPU (%)	476	0	0.8	1.6	22.6(6)

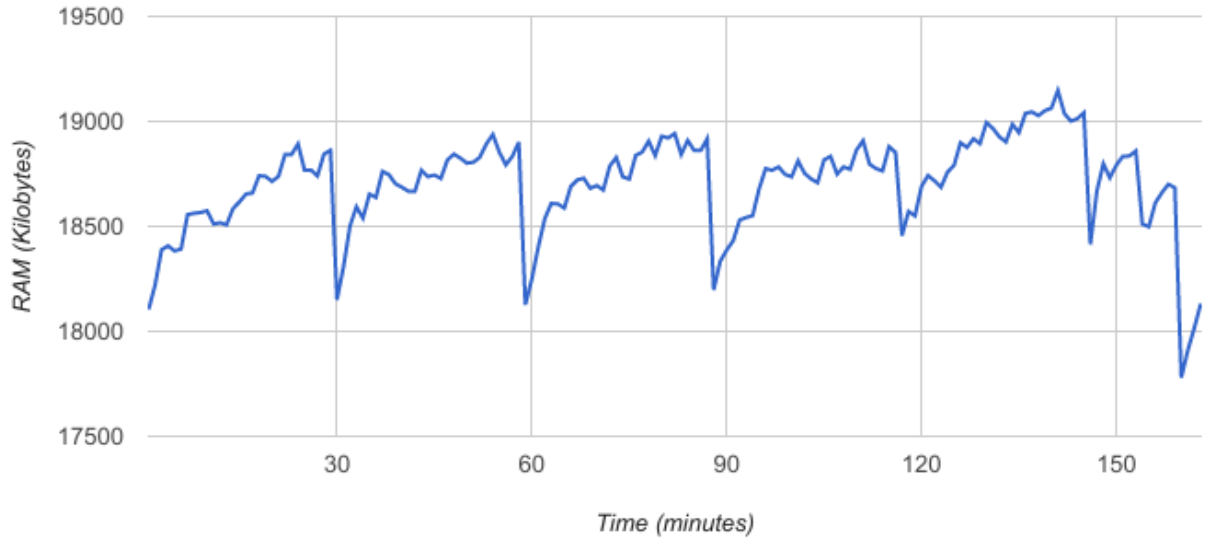


Figure 8.1: Device Agent RAM usage over time

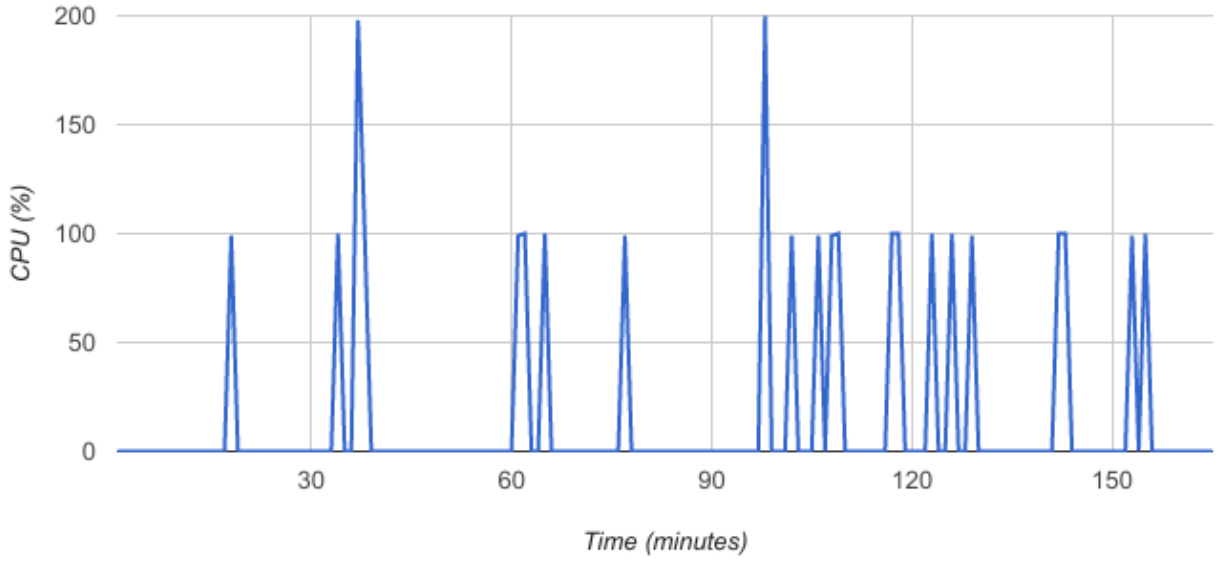


Figure 8.2: Device Agent CPU usage over time

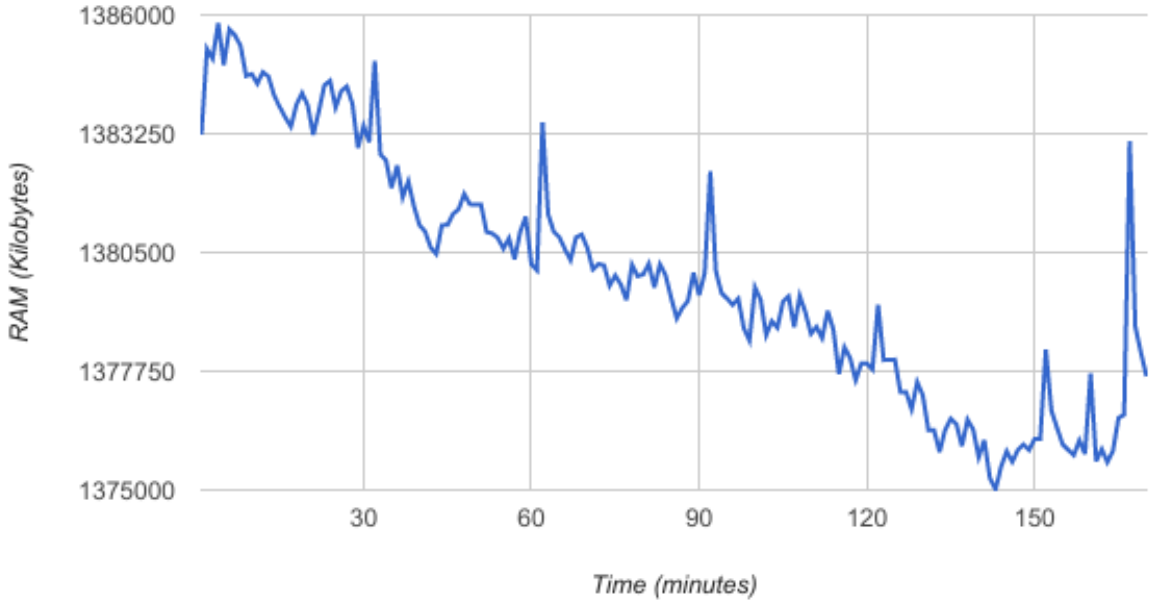


Figure 8.3: System RAM usage over time

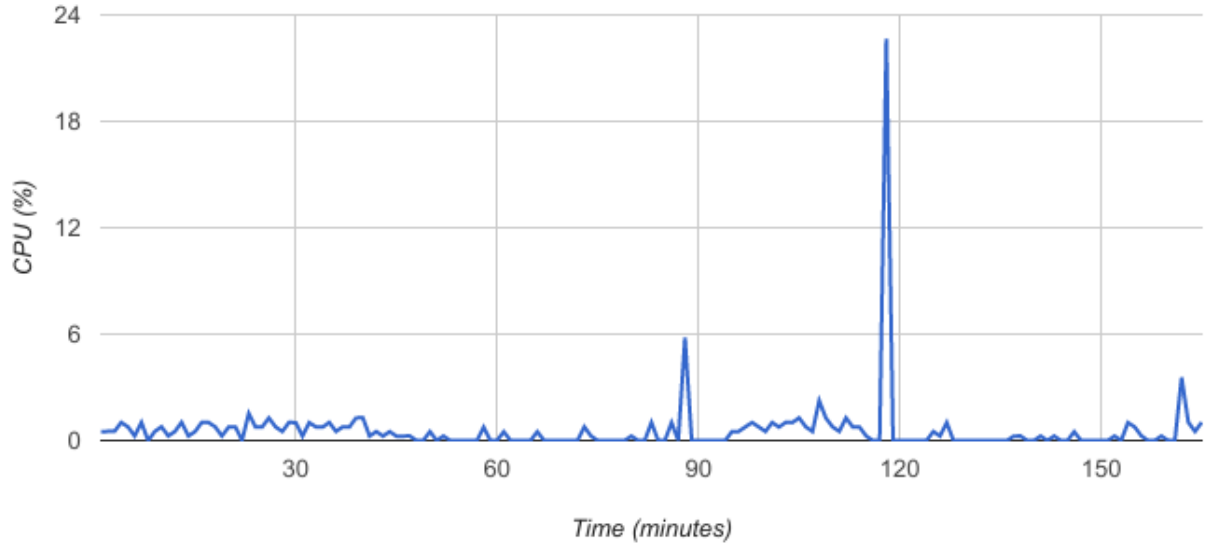


Figure 8.4: System CPU usage over time

Regarding Figure 8.2, its content might seem erroneous or dubious at first due to the over 100% y axis, but it is completely justified one we understand that the CPU % usage in this context also conveys the number of cores being allocated by the Android system to the application, where a 200% value indicates that 2 cores are being allocated, each at 100% usage capacity. Therefore, Figure 8.2 illustrates that only during a few brief moments does our application require system resources, and in doing fully makes the most out of them.

We also decided to monitor the device externally using ADB (Android Debug Bridge) and the ***top*** command, obtaining **5179 measurements** summarized in Table 8.2 and Figure 8.5. Table 8.2 analyzes the application's CPU percentile usage and the number of threads used, whilst Figure 8.5 puts into perspective the RAM usage of the device and the application.

Table 8.2: ADB "top" analysis

	Minimum	**Average**	σ	**Maximum**
Application CPU (%)	0	0.007	0.368	26
Application Threads (#)	14	17.87	0.823	22

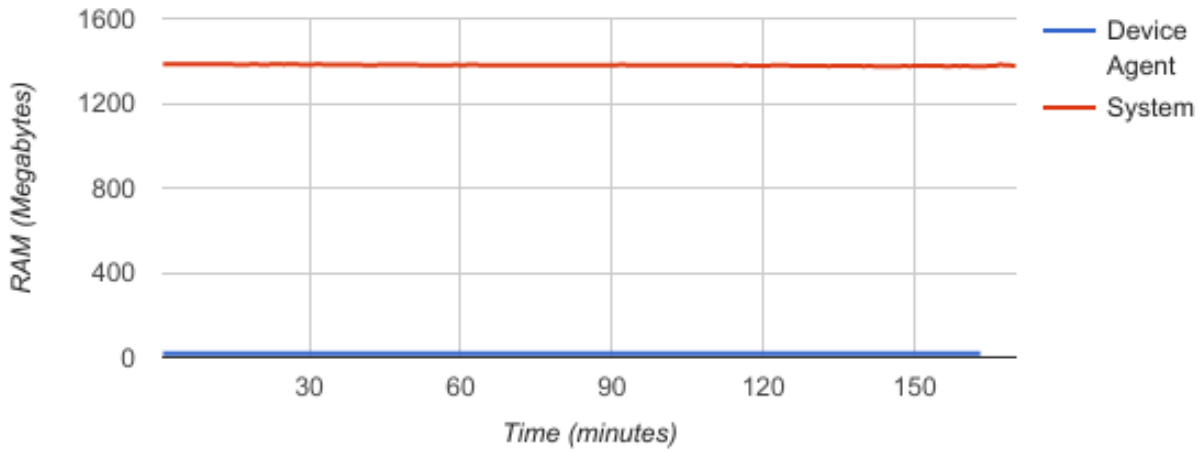


Figure 8.5: Avaialble RAM over time

As can be seen, our Android application has minimal impact on the device's resources, meaning that it can be used to monitor it without hindering it or interfering with other applications it is running. **Our application's RAM and CPU usage accounts for about 0-1% of the system's**, according to Figure 8.5, and Tables 8.1 and 8.2. It is noteworthy that although our application constantly has a substantial number of worker threads, the reduced RAM and CPU footprint can be attributed to the eventful and short lived nature of each thread, existing only to record the current particular status of the the system.

8.2 Network Traffic

As Cisco has stated in their white paper (Cisco 2017), the traffic generated by smartphone usage alone is well underway to surpass the once traditional portion of global network traffic. In this Big Data era, a reduced network footprint should be a concern so as to diminish the amount of data that has to be needlessly processed in a system. This has been one of this project's main concerns from the start and, as we've stated before, one of the driving forces for our technological choices (e.g. MQTT).

Regarding our system's network traffic, when connected to the Internet a device agent is supposed to send the collected data to a message broker, but this connection can be done either via WiFi or Mobile Networks such as GSM. Over time, data traffic plans for WiFi have become more lenient and faster, especially in stationary environments, but Mobile Networks are still very restrictive in all of these aspects. With all this in mind, we have to analyze and determine the average and maximum expected data transfers (in bytes) per connection, per device, and for the system as a whole.

There are several ways of tackling this problem, especially if the system were already in use by a substantial amount of active users, namely, amassing and analyzing network logs with software such as Wireshark, recording and analyzing every sent message within the system, etc. However, since we don't currently have a large number of active users, we've decided to predict the expected impact on network traffic by analyzing the size and occurrence frequency of the existing messages when sent from the device agents, giving us theoretical limits.

All of the intermediate calculations for this section are available at the following spreadsheet (Lima 2017a), which can be copied and edited to simulate other scenarios.

8.2.1 Individual Message Analysis

We have to take into consideration that the majority of the system is written in Java, and as such, it's primitive data types should to be accounted for. In the case of time configurations, all of them use Integers (*int*) for their respective value, and as such, they always occupy 32 bits or 4 bytes in memory. But, we also have to remember that what is sent through the network are not the 4 bytes representing these integers, but rather plain text string representations of them, so the number *60 000* while capable of being stored in 4 bytes in memory, will in fact occupy 5 bytes as "*60000*", where one byte is used to represent each character (which is only true because the messages are writing using only ASCII characters). So, we take from this, is that they will at most occupy 10 bytes ($2^31 - 1 =$ "2147483647"), and at minimum 1 byte ("0") while still being valid entries.

Tables 8.3 and 8.4 list the expected overhead for each of the **ConfigurationMessage** and **DeviceMessage** types within the system, namely their sizes and occurrence frequencies per day. Unlike with Table 8.3, the range of possible values for the messages in Table 8.4 is

substantially wider, thus we analyze only the size of the average message per message type, based on our test observations. It is also worth mentioning that Table 8.3 does not take into account the JSON overhead associated with the messages, only the attribute's string size overheads.

Table 8.3: Configuration Message Size Analysis

	Aprox. Size (bytes)		
Configuration Message Type	**Minimum**	**Average**	**Maximum**
Data Generation			
RAM	1	5	10
CPU	1	5	10
GPS	1	5	10
CPU_USAGE	1	5	10
RAM_USAGE	1	5	10
BATTERY	1	6	10
OPEN_PORTS	1	5	10
OPEN_DATA_TRAFFIC	1	6	10
AGGREGATE_PERIODIC_DATA	1	7	10
AGGREGATE_EVENTFUL_DATA	1	7	10
Message Management			
PUBLISH_DATA	1	7	10
UPDATE_CONFIGURATIONS	1	7	10
MQTT_TIMEOUT	1	2	10
MQTT_KEEP_ALIVE	1	2	10
SERVER_PROTOCOL	3	3	4 [1]
SERVER_URI	1	15 [2]	23 [3]
SERVER_PORT	1	4	10
SERVER_BASE_PUBLISH_TOPIC	1	9	65 536 [4]
SERVER_BASE_UPDATE_TOPIC	1	14	65 536 [5]
SERVER_PASSWORD	0	16	65 535 [6]

Regarding Table 8.4, it should be noted that **Eventful** and **One Time** messages do not have stable or predefined occurrence frequencies because they are dependant on certain triggering events, which can happen at any time, and that depend on the user's behaviour, so, in

[1] Possible values: "tcp", "mqtt", "ssl"
[2] IPv4 address
[3] IPv6 address
[4] Limit as per the official MQTT documentation for topic names (MQTT Documentation 2017a)
[5] See footnote 4
[6] Limit as per the official MQTT documentation for passwords (MQTT Documentation 2017b)
[7] A screen change can be either a "screen on" or "screen off" event, so every pair denotes one interaction.
[8] Source for this seemingly random value (Mail Online 2015)

Table 8.4: Device Message Daily Frequency Analysis

Message type (m)	$ANDM$ (#)	$ASDM$ **(bytes)**
Periodic		
RAM	1 440	37
CPU	1 440	21
GPS	1 440	35
CPU Usage	8 640	44
RAM Usage	8 640	72
Battery	288	13
Open Ports	1 440	155
Data Traffic	288	84
Eventful		
Time Change	0	65
Charging Change	1	70
Connection Change	1	63
Cell Location Change	1	100
Screen Change	50 [7] [8]	70
Power Change	0	70
One Time		
Package Change	0	150
Aggregated		
Aggregated Periodic	2 880	120
Aggregated Eventful	2 880	90
Total	29 429	1 259

order to not discard them completely from this exercise, for our calculations we assume the values that have been listed, using them as rough and optimistic estimations of what could be a regular device's behaviour throughout a day. This means that we are assuming that the user will at least interact with his phone 25 times, and experience 1 cell tower handover (Wikipedia 2017a). This user won't however install/modify any new/existing applications, nor will he experience a Timezone change, nor power off his phone. The values for the **Periodic** and **Aggregated** are obtained through the default Device Agent Configurations listed in Tables 6.2, 6.3, and 6.4.

Another assumption for these calculations is that no Device Agent Configuration changes are made, thus, no **Configuration Messages** are exchanged in the network.

8.2.2 Device-level network traffic overhead analysis

Using the values provided by Tables 8.3, and 8.4, we can now infer the expected network traffic generated by each device, which we will call the Average Device Network Traffic (**ADNT**). We calculate it by multiplying the size overhead of a Device Message by the sum of the expected impact of each type of message in the system, which itself is the product of the Average Number of Device Messages of type m per day (**ANDM(m)**) and the Average Size of Device Messages of type m (**ASDM(m)**). We also account for some of the JSON representation's size overhead (**DMJSO** and **CJSO**).

$$\text{ADNT} = \sum \text{DMJSO} + (\text{ANDM}(m) * (\text{ASDM}(m) + \text{CJSO}))$$

- ADNT - Average Device Network Traffic
- DMJSO - Device Message JSON Size Overhead
- ANDM(m) - Average Number of Device Messages of type m per day
- ASDM(m) - Average Size of Content in Device Messages of type m
- CJSO - Content's JSON string representation Size Overhead
- Message Type - $m \in \{\text{PM} \mid \text{EM} \mid \text{OM} \mid \text{AM}\}$
- Set of Periodic Messages - PM = { RAM, CPU, GPS, CPU Usage, RAM Usage, Battery, Open Ports, Data Traffic }
- Set of Eventful Messages - EM = { Time Change, Charging Change, Connection Change, Screen Change, Power Change }
- Set of One Time Messages - OM = { Package Change }
- Set of Aggregated Messages - AM = { Aggregated Periodic, Aggregated Eventful }

Accounting for an estimated **CJSO** of 73 bytes and a **DMJSO** of 124 bytes by using 6.5 and 6.6 as average message containers, we can calculate that the **ADNT** should be around 4 146 254 bytes, which means that each device should produce around **4.1 MB of data every day**, while using the default configurations and the subsequent values in Table 8.4.

8.2.3 Aggregated network traffic overhead analysis

Now that we have the **ADNT**, we can extrapolate the expected network traffic generated for any number of similar devices by simply multiplying it.

$$\text{ANT} = \text{ADNT} * \text{NSDN}$$

- ANT - Average Network Traffic

- ADNT - Average Device Network Traffic
- NSDN - Number of Similar Devices in Network

These formulas give us the expected **Average Network Traffic**, but the same logic could be applied to the minimum and maximum Network Traffic, giving us boundaries to assess the system's Network Traffic. Since our system was designed with BYOD and MDM environments in mind, we might as well extrapolate how this analysis scales up for and increasingly higher number of users, going from a personal/testing environment (1 to 100), to an enterprise environment (100 to 100 000), to an international environment (100 000 to 10 000 000) (Wikipedia 2017b)

Table 8.5: Expected Daily Network Traffic Analysis

NSDN (#)	**Aprox.** *ANT* **(bytes)**
1	4 MB
10	40 MB
100	400 MB
1 K	4 GB
10 K	40 GB
100 K	400 GB
1 M	4 TB
10 M	40 TB

The numbers provided by Table 8.5 are hard to understand unless they are put into perspective, so let's look at Google, the tech giant know for its Big Data managerial endeavours. In 2008, almost a decade ago, it was estimated that Google was handling 20 PetaBytes (PB) (Kennedy 2008) of data on a daily basis, and since Google provides services on a global scale, let us consider the global population at the time as its user base, which in 2008 would be around 6.7 billion (B) (Population Reference Bureau 2008). This means that in this scenario each person would produce around 2 985 074 bytes or an approximate **3 MB of data per day**. The amount of data that Google processes on a daily basis has value has most certainly increased dramatically with the advent and widespread usage of mobile devices in recent years. In fact, if we apply Moore's Law (computing power doubles every two years) to Google's processing power, we find that it should be currently handling 480 PB of data, and with a current global population of around 7.5 billion (B) (Wikipedia 2017c), each of Google's users should be responsible for around **64 MB daily**.

With these values as comparison guidelines, we can see that our system is fairly conservative, even when collecting data thousands of times per day per user, even by Google's speculated standards. Furthermore, even if this amount of data traffic is not sustainable, there is plenty of space for improvement, be it by **reducing the data collection frequencies or by reducing the overhead of each message**. The former should be trivial, thanks to the Device Agent Management feature, the latter however is more challenging. Firstly, instead of sending the messages in JSON **plaintext**, they can be **compressed** beforehand

and **decompressed** on the other end; this of courses raises the question of whether or not this additional computing overhead is justified over many messages and it would require testing. Secondly, a substantial part of the exchanged messages themselves is attributed to the serialized variables' names, so by simply having **shorter but well-documented and well-formed variable names**, a couple of tens of bytes could potentially be saved for each message. Lastly, the collected data content itself could cleaned up, even by just stripping excessive whitespace. Given that for the default configurations there are approximately 30 000 messages sent daily per user, these humble alterations are sure to add up.

Chapter 9

Project Analysis

This chapter is dedicated to reviewing this project's development up until this point, assessing the state of the requirements and features, whether they have been completed or not, and the overall state of completion of each of the system's components. The purpose of this analysis is that we hope to show that our initial predictions and estimations in Chapters 3 and 4 underestimated the sheer complexity of the final system and of the configuration and development effort needed to bring it to fruition, and that nonetheless, we came really close to achieving it, and laid out the work foundations for it to be possible, should the project ever be picked up in the future.

9.1 Progress Assessment

In this section we first analyze the proposed requirements, followed by an analysis of the features, tallying up the number of completed tasks and met expectations in the final subsection.

9.1.1 Requirements

Looking back at Chapter 3, namely at the requirements section and Tables 3.1, 3.2, 3.3, and 3.4, we had established the requirements for our system and its components. We revise these lists and assess how this project stands on fulfilling them thus far with Tables 9.1, 9.2, 9.3, and 9.4.

Table 9.1: Development Progress: Architectural Requirements

Requirement	Priority	Status
Device Agent Operating System - the Device Agent Application runs on Android devices.	**Must Have**	✓
Loose coupling between Device Agent and Central Aggregator - changes to the data collected with the terminals is transparent to the central aggregator.	**Must Have**	✓
Central Aggregator - Uses Machine Learning tools instead of hardcoded classification.	**Must Have**	✓
Dashboard-Aggregator control - The actions in the Dashboard intended as modifications alter the behaviour of the Central Aggregator.	Should Have	✓
Dashboard Browser Support - The dashboard runs on Mozilla's Firefox browser.	Should Have	✗
Dashboard Browser Support - The dashboard runs on browsers other than Mozilla's Firefox.	Could Have	✗
User Anonymity - Secure connections are used by the system	**Must Have**	✓
User Anonymity - Secure connections are used by the system and uphold anonymity for the mobile terminal users.	Should Have	✓
Total		**6/8**

Table 9.2: Development Progress: Behavioural Requirements

Requirement	Priority	Status
The **Device Agent Application** collects data from the mobile terminal it is installed in.	**Must Have**	✓
The **Device Agent Application** has a low impact on the terminals battery, CPU and bandwidth.	Should Have	✓
The **Device Agent Application** periodically transfers data to the central aggregator.	**Must Have**	✓
The **Device Agent Application** securely transfers data to the central aggregator.	**Must Have**	✓
The **Device Agent Application** copes with connectivity issues with the central aggregator.	Should Have	✓
The **Central Aggregator** employs Machine Learning on the collected data for each user.	Should Have	✗
The **Central Aggregator** employs Data Analysis on the collected data for each user.	Could Have	✓
The **Central Aggregator** employs Machine Learning on the collected data between users.	**Must Have**	✓
The **Central Aggregator** employs Data Analysis on the collected data between users.	Could Have	✓
The **Dashboard** displays metrics, findings and collected data.	Should Have	✗
The **Dashboard** displays user activity.	Could Have	✗
Total		**8/11**

Table 9.3: Development Progress: Functional Requirements

Requirement	Priority	Status
The **Device Agent Application** handles the temporary storage of the data in the mobile terminal.	**Must Have**	✓
The **Device Agent Application** handles the deletion of unnecessary data stored in the mobile terminal.	Should Have	✓
The **Device Agent Application** handles the transfer of data stored in the mobile terminal to the Central Aggregator.	**Must Have**	✓
The **Device Agent Application** only deletes stored data from the mobile terminal that has already been transferred to the Central Aggregator.	**Must Have**	✓
The **Device Agent Application** requests updated data collection configurations.	**Must Have**	✓
The **Device Agent Application** updates its data collection configurations.	**Must Have**	✓
The **Central Aggregator** stores the collected data for each user.	**Must Have**	✓
Total		**7/7**

Table 9.4: Development Progress: Non-functional Requirements

Requirement	Priority	Status
The data collected with the **Device Agent Application** does not occupy more than 100MB of the mobile terminal's storage space.	Should Have	✓
The **Central Aggregator** scales for dozens of users.	**Must Have**	✓
The **Central Aggregator** scales for hundreds of users.	Could Have	✓
The **Central Aggregator** scales for thousands of users.	Won't Have	✓
The **Central Aggregator** stores up to 3 months of consecutive data for each user.	Should Have	✗
The **Central Aggregator** stores all the data ever received for each user.	Won't Have	✓
The **Central Aggregator** stores up to 3 months of consecutive data for each user.	Should Have	✗
Total		**5/7**

9.1.2 Features

This project's requirements have indubitably shaped it throughout the development process, however they do not address the full extent of features that have been developed and their subtle nuances, so we've decided to also include Tables 9.5, 9.6, and 9.7, which try to include all planned, relevant, and unmentioned features, depicting their current development state. Table 9.5 analyzes the Device Agent Android Application, Table 9.6 inspects the Central Aggregator Cluster's, and Table 9.7 focuses on the Dashboard subsystem.

Table 9.5: Development Progress: Device Agent Features

Feature	Status
Scheduling	
Schedule periodic data collection	✓
Schedule eventful data collection	✓
Schedule one time data collection	✓
Schedule periodic data aggregation	✓
Schedule eventful data aggregation	✓
Schedule data upload	✓
Schedule configuration updates	✓
Data Collection	
Collect periodic data	✓
Collect eventful data	✓
Collect one time data	✓
Data Aggregation	
Aggregate periodic data	✓
Aggregate eventful data	✓
Data Exchange	
Upload collected data through MQTT	✓
Upload aggregated data through MQTT	✓
Retrieve updated configurations through MQTT	✓
Delete uploaded data	✓
Data Management	
Store data in SQLite database	✓
Read data from SQLite database	✓
Update data in SQLite database	✓
Delete data in SQLite database	✓
Limit amount of data in SQLite database	✓
Total	**21/21**

Table 9.6: Development Progress: Central Aggregator Features

Feature	Status
Mosquitto Server	
Receive data through mosquitto topic	✓
Upload updated configurations through mosquitto topic	✓
Kafka Server	
Receive data through Kafka topic	✓
Move input data through Kafka topic "OryxInput"	✓
Move update data through Kafka topic "OryxUpdate"	✓
Data Exchange	
Upload updated configurations through mosquitto server	✓
Receive data through mosquitto server	✓
Bridge received data between the MQTT and Kafka servers	✓
Receive data through Kafka server	✓
Data Management	
Store data in HDFS	✓
Read data from HDFS	✓
Update data in HDFS	✓
Coordinate and manage HDFS with Zookeeper	✓
Coordinate and manage HDFS with YARN	✓
Lambda Architecture	
Communicate with serving layer	✓
Serving layer communicates with batch layer	✓
Serving layer communicates with speed layer	✓
Machine Learning	
Classify data with K-Means Clustering	✓
Perform Triple Modular Redundancy	✗
Perform Triple Redundancy	✓
Total	**19/20**

Table 9.7: Development Progress: Dashboard Features

Feature	**Status**
Device Management	
Display device configurations	✗
Interact with device configurations	✗
Data Visualization	
Generate graph of relationships across all devices	✗
Generate graph of relationships across all data	✗
Generate graph of relationships across all devices and all data	✗
Display relationships across all devices	✗
Display relationships across all data	✗
Display relationships across all devices and all data	✗
Highlight anomalous devices	✗
Highlight anomalous data	✗
Total	**0/10**

9.1.3 Assessment

Looking at the totals in Tables 9.1, 9.2, 9.3, and 9.4 we are proud to conclude that the developed system satisfies the majority of the architectural, behavioural, functional, and non-functional requirements, respecting almost all of the **Must Have** and **Should Have** priority requirements.

Following up with Tables 9.5, 9.6, and 9.7 and their respective totals, both the Android Application and the Server Cluster correspond almost entirely to all of their feature expectations, whilst the Dashboard doesn't satisfy a single feature requirement.

We can condense the previous subsection's status totals from Tables 9.1, 9.2, 9.3, and 9.4, and from Tables 9.5, 9.6, and 9.7 into Table 9.8. All of this information is also calculated dynamically using the spreadsheet at (Lima 2017b).

Table 9.8: Development Progress: Totals

Requirements

Architectural	6/8
Behavioural	8/11
Functional	7/7
Non-functional	5/7

Features

Device Agent Application	21/21
Central Aggregator Cluster	19/20
Dashboard	0/10

Total	**66/84**

From Table 9.8 we can conclude that by looking exclusively at the total number of satisfied requirements and developed features, the project is **approximately 79% complete** with 66 out of 84 tasks. It is apparent that both the Device Agent Application and the Central Aggregator Cluster are **nearly complete**, whilst the Dashboard, although started, is essentially incomplete on all aspects. Having in mind that plenty of setbacks were encountered, that multiple initial features and requirements had to be **revised and altered** for the Central Aggregator, and that both the Central Aggregator and Device Agent were developed extensively, **this assessment isn't dissatisfying**.

9.2 Future work and next developments

The following section overviews the tasks that haven't been completed yet and details the future work necessary to bring them to fruition.

9.2.1 Central Aggregator

Regarding the Central Aggregator, some technical considerations must also be taken into account, namely the maximum delay per stream connection, the number of streaming connections available, the number of concurrent batch processing jobs, the minimum number of generated reports per day, database backup frequency, and expected uptime per year. All of these considerations will have to be analyzed once data has been collected, in order to fine tune the Central Aggregator and minimize unnecessary strain on the system.

9.2.2 Dashboard

As has been stated in the previous progress assessment section, the Dashboard is essentially incomplete given that currently it does not allow for a user to manage Device Agent Configurations, nor does it provide the means to analyze the collected or the classified data in the Central Aggregator through a UI. As such, all of the development effort needs to be put forward in order to finish this subsystem.

9.2.3 Dataset Generation

In order to validate our system we must gather data from various users on various devices, broadening the range of what is to be accepted as normal behaviour, and ultimately train the Central Aggregator. After that we will have to gather some recent, widespread, and well known malware examples, infect some of the previous devices, let the same users use them again, and ultimately check if our system can correctly detect anomalous behaviour.

9.2.4 Device and Data Analysis

Once the Dashboard is fully functional and the Dataset Generation step have been surmounted, we need to use the GraphX generated graphs and graphics and perform a more in depth analysis of the underlying data, namely the amount of messages, the average, least common and most common values regarding messages, and how each device is classified by the K-Means algorithm.

- Number, average, least common and most common
  - Messages sent
  - Messages sent by message type
  - Messages sent by each device, by message type
- K-Means Classification

- Number of devices in each class
- Which devices are in each class

We believe that these are the most fundamental yet crucial metrics that should be prioritized regarding the data analysis, and that they will ease the process of manual anomaly detection.

Chapter 10

Final Remarks and Conclusions

Since mobile devices are far more resource constrained than stationary computers or laptops and are not meant to be constantly or frequently connected to a charging outlet, any attempt at performing additional computations needs to be extremely conservative. The best alternative to performing any resource intensive task on such devices is to perform it on another device that isn't as limited. This also makes it more feasible to manage and analyze data from other devices.

The purpose of this project is deeply intertwined with energy and resource consumption, and how anomalies such as malware can become apparent due to the increase in consumption they represent. As such, one cannot ignore the fact that any potential progress originated by this project will probably not end the action-reaction cycle of malware development for mobile devices, resulting in more subtle anomalies. Ultimately, this whole problematic might become just another case study of a Jevons Paradox (Wikipedia 2017d), a scenario in which technological progress increases the efficiency by which a resource is used, however, the rate of its consumption increases due to rising demand.

Lastly, we would like to express our enthusiasm and satisfaction towards the obtained intermediate results, which show much promise and lead us to believe that this is in fact a effort-worthy approach to the problem. Unfortunately, almost a full year wasn't enough to bring this project to fruition, however close it might have gotten.

References

[1] Alfalqi, K., Alghamdi, R. and Waqdan, M. (2015) Android Platform Malware Analysis., 6(1), pp.140–146.

[2] Android. Available at: https://www.android.com/index.html (last accessed September 1, 2017).

[3] Answers, Fabric. Available at: https://fabric.io/kits/android/answers (last accessed September 1, 2017).

[4] Apache Spark (2017a) Machine Learning Library (MLlib). Available at: https://spark.apache.org/docs/1.0.1/mllib-guide.html (last accessed January 22, 2017).

[5] Apache Spark (2017b) Spark Overview. Available at: https://spark.apache.org/docs/1.0.1/index.html (last accessed January 22, 2017).

[6] Apple App Center (2016a) App Store Review Guidelines. Available at: https://developer.apple.com/app-store/review/guidelines/ (last accessed October 21, 2016).

[7] Apple Developers (2016b) About Info.plist Keys and Values. Available at: https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Introduction/Introduction.html (last accessed October 21, 2016).

[8] Armando, A. et al. (2012) Changing user attitudes to security in bring your own device (BYOD) & the cloud. Network Security, 2012(3), pp.5–8.

[9] Bandugula, N. (2015) The 5-Minute Guide to Understanding the Significance of Apache Spark. Available at: https://www.mapr.com/blog/5-minute-guide-understanding-significance-apache-spark (last accessed January 22, 2017).

[10] Becher, M. and Freiling, F.C. (2008) Towards Dynamic Malware Analysis to Increase Mobile Device Security. In Proc. of SICHERHEIT, P-128, pp.423–433. Available at: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84873743297&partnerID=40&md5=568aa830f6fbf59ca90683d4c5c34d23.

[11] Bertran, P. F. (2013) An example "lambda architecture" for real-time analysis of hashtags using Trident, Hadoop and Splout SQL. Available at: http://www.datasalt.com/2013/01/an-example-lambda-architecture-using-trident-hadoop-and-splout-sql/ (last accessed January 20, 2017).

[12] Bertran, P. F. (2014) Lambda Architecture: A state-of-the-art. Available at: http://www.datasalt.com/2014/01/lambda-architecture-a-state-of-the-art/ (last accessed January 20, 2017).

[13] Bijnens, M. (2017), Lambda Architecture. Available at: http://lambda-architecture.net/ (last accessed 1 Sep. 2017).

[14] Bijnens, N., Hausenblas, M. (2016) Lambda Architecture: A state-of-the-art. Available at: http://lambda-architecture.net/ (last accessed January 20, 2017).

[15] Bornstein, Dan. "Presentation of Dalvik VM Internals" (PDF). Google. https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf (last accessed on November 11, 2016).

[16] Bose, A. et al. (2008) Behavioral detection of malware on mobile handsets. Proceeding of the 6th international conference on Mobile systems, applications, and services - MobiSys '08, p.225.

[17] Boyd, J. R. (1995) The Essence of Winning and Losing. A Discourse on Winning and Losing.

[18] Boyd, J.R. (1976) Destruction and Creation. A Discourse on Winning and Losing, (September), pp.3–9.

[19] Bradley, J., Meng, X. and Lee, D. (2016) Why you should use Spark for machine learning. Available at: http://www.infoworld.com/article/3031690/analytics/why-you-should-use-spark-for-machine-learning.html (last accessed January 22, 2017).

[20] Burguera, I., Zurutuza, U. and Nadjm-Tehrani, S. (2011) Crowdroid: Behavior-Based Malware Detection System for Android. Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11, p.15.

[21] CentOS. Available at: https://www.centos.org/ (last accessed September 1, 2017).

[22] Cisco (June 207), The Zettabyte Era: Trends and Analysis. Available at: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html (last accessed 31 August, 2017)

[23] Crashlytics, Fabric. Available at: https://fabric.io/kits/android/crashlytics (last accessed September 1, 2017).

[24] Deploying Hadoop (2017). Available at: https://tecadmin.net/setup-hadoop-single-node-cluster-on-centos-redhat/ (last accessed September 1, 2017).

[25] Deploying Kafka (2017a). Available at: http://www.bogotobogo.com/Hadoop/BigData_hadoop_Zookeeper_Kafka.php (last accessed September 1, 2017).

[26] Deploying Kafka (2017b). Available at: https://www.vultr.com/docs/how-to-install-apache-kafka-on-centos-7 (last accessed September 1, 2017).

[27] Deploying Spark (2017). Available at: http://www.aodba.com/how-to-install-apache-spark-in-centos-standalone/ (last accessed September 1, 2017).

[28] Deploying the MQTT Broker (2017a). Available at: https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-the-mosquitto-mqtt-messaging-broker-on-centos-7 (last accessed September 1, 2017).

[29] Deploying the MQTT Broker (2017b). Available at: https://mosquitto.org/man/mosquitto-conf-5.html (last accessed September 1, 2017).

[30] Deploying the MQTT Broker (2017c). Available at: https://github.com/eclipse/mosquitto/blob/master/mosquitto.conf (last accessed September 1, 2017).

[31] Deploying the MQTT Broker (2017d). Available at: https://test.mosquitto.org/ (last accessed September 1, 2017).

[32] Distefano, A., et al. (2011) SecureMyDroid: Enforcing Security in the Mobile Devices Lifecycle.

[33] Eslahi, M. et al. (2015) BYOD: Current state and security challenges. ISCAIE 2014 - 2014 IEEE Symposium on Computer Applications and Industrial Electronics, (April 2014), pp.189–192.

[34] Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Dissertation. University of California, Irvine.

[35] FireEye (2015) Zero-Day Danger: A Survey of Zero-Day Attacks and What They Say About the Traditional Security Model., p.16.

[36] Flurry Analytics, Yahoo. Available at: https://developer.yahoo.com/analytics/ (last accessed September 1, 2017).

[37] Frei, D. (2012) Conducting a Risk Assessment for Mobile Devices.

[38] Git version control. Available at: https://git-scm.com/ (last accessed September 1, 2017).

[39] Github version control. Available at: https://github.com/ (last accessed September 1, 2017).

[40] Google Analytics, Firebase. Available at: https://firebase.google.com/products/analytics/ (last accessed September 1, 2017).

[41] Google Android Developers (2016a) App Manifest. Available at: https://developer.android.com/guide/topics/manifest/manifest-intro.html (last accessed October 21, 2016).

[42] Google Android Developers (2016b) Launch Checklist. Available at: https://developer.android.com/distribute/tools/launch-checklist.html (last accessed October 21, 2016).

[43] Google Android Developers (2016c) Requesting Permissions at Run Time. Available at: https://developer.android.com/training/permissions/requesting.html (last accessed October 21, 2016).

[44] GraphX, Apache. Available at: https://spark.apache.org/graphx/ (last accessed September 1, 2017).

[45] Hadoop, Apache. Available at: https://hadoop.apache.org/ (last accessed September 1, 2017).

[46] Halevi, S., Shoup, V. "HElib: An Implementation of homomorphic encryption". Available at: https://github.com/shaih/HElib (last accessed December 31, 2014).

[47] Halilovic M., A. Subasi (2012) Intrusion Detection on Smartphones.

[48] Harris, M.A. and Patten, K.P. (2015) Mobile Device Security Issues Within the U.S. Disadvantaged Business Enterprise Program. Journal of Information Technology Management, XXVI(1), pp.46–57.

[49] HockeyApp, Microsoft. Available at: https://hockeyapp.net/ (last accessed September 1, 2017).

[50] Hortonworks' Sandbox. Available at: https://hortonworks.com/products/sandbox/ (last accessed September 1, 2017).

[51] IDC (2016), "Smartphone OS Market Share" report for November 2016. Available at: https://www.idc.com/promo/smartphone-market-share/os (last accessed September 1, 2017).

[52] Jaatun, M.G., Jaatun, E.A.A. and Moser, R. (2014) Security considerations for tablet-based eHealth applications. CEUR Workshop Proceedings, 1251(Pahi 2014), pp.27–36.

[53] Jaramillo, D., B. Furht, and A. Agarwal (2014) Mobile Virtualization Technologies. In: Virtualization Techniques for Mobile Systems, 14th ed., Vol 73, pp. 5-20.

[54] Java Datatypes. Available at: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (last accessed 1 Sep. 2017).

[55] Java. Available at: https://www.java.com (last accessed September 1, 2017).

[56] Kafka, Apache. Available at: https://kafka.apache.org/ (last accessed September 1, 2017).

[57] Kennedy, M. (January 8, 2008), Google's MapReduce Stats. Available at: https://www.niallkennedy.com/blog/2008/01/google-mapreduce-stats.html (last accessed 1 Sep. 2017).

[58] Kocher, P. et al. (2004) Security as a new dimension in embedded system design. DAC '04: Proceedings of the 41st annual Design Automation Conference, pp.753–760.

[59] Kotlin. Available at: https://developer.android.com/kotlin/index.html (last accessed September 1, 2017).

[60] Lebek, B., Degirmenci, K. and Breitner, M.H. (2013) Investigating the Influence of Security, Privacy, and Legal Concerns on Employees' Intention to Use BYOD Mobile Devices. Amcis, (2008), pp.1–8.

[61] Lee, J., T. Kim, and J. Kim (2009) "Energy-efficient Run-time Detection of Malware-infected Executables and Dynamic Libraries on Mobile Devices", Software Technologies for Future Dependable Distributed Systems.

[62] Li, Q. and Clark, G. (2013) Mobile Security: A Look Ahead. IEEE Security & Privacy, 11(1), pp.78–81

[63] Lima, A. (2017a), Network Traffic Spreadsheet. Available at: https://goo.gl/BHn2oZ (last accessed September 1, 2017).

[64] Lima, A. (2017b), Project Assessment Spreadsheet. Available at: https://goo.gl/B48cGq (last accessed September 1, 2017).

[65] Lima, A., Sousa, B., Cruz, T., Simões, P. (2016) Security for Mobile Device Assets: a Survey

[66] Luo, J., and m. Kang (2011) "Application Lockbox for Mobile Device Security", Eighth International Conference on Information Technology: New Generations.

[67] Mail Online (October 29, 2015), We check our phones 85 TIMES a day - twice as often than we realise. Available at: http://www.dailymail.co.uk/sciencetech/article-3294994/How-check-phone-Average-user-picks-device-85-times-DAY-twice-realise.html (last accessed September 1, 2017).

[68] MapR's Hadoop Sandbox. Available at: https://mapr.com/products/mapr-sandbox-hadoop/ (last accessed September 1, 2017).

[69] Maven, Apache. Available at: https://maven.apache.org/ (last accessed September 1, 2017).

[70] Mazumdar, S. and Paturi, A. (2011) Tamper-resistant database logging on mobile devices. Internet Security (WorldCIS), 2011 World Congress on, pp.165–170.

[71] Microsoft Windows Dev Center (2016) The app certification process. Available at: https://msdn.microsoft.com/windows/uwp/publish/the-app-certification-process (last accessed October 21, 2016).

[72] Miettinen, M., and P. Halonen. (2006) "Host-Based Intrusion Detection for Advanced Mobile Devices", 20th International Conference on Advanced Information Networking and Applications (AINA'06), Vol 1.

[73] Mobile Marketshare (2017a), Email client marketshare. Available at: https://emailclientmarketshare.com/ (last accessed August 4, 2017).

[74] Mobile Marketshare (2017b), Mobile web usage. Available at: http://www.telegraph.co.uk/technology/2016/11/01/mobile-web-usage-overtakes-desktop-for-first-time/ (last accessed August 4, 2017).

[75] Mobile Marketshare (2017c), Mobile marketing analytics. Available at: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ (last accessed August 4, 2017).

[76] Mosquitto project. Available at: https://mosquitto.org/ (last accessed September 1, 2017).

[77] MQTT Documentation (2017a). Available at: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718109 (last accessed September 1, 2017).

[78] MQTT Documentation (2017b). Available at: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc385349247 (last accessed September 1, 2017).

[79] MQTT Essentials Part 5: MQTT Topics & Best Practices. Available at: http://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices (last accessed July 24, 2017).

[80] MQTT man page. Available at: https://mosquitto.org/man/mqtt-7.html (last accessed July 24, 2017).

[81] MQTT organization. Available at: http://mqtt.org/ (last accessed September 1, 2017).

[82] Mulligan, B. (2014) The Right Way To Ask Users For iOS Permissions, Techcrunch. Available at: https://techcrunch.com/2014/04/04/the-right-way-to-ask-users-for-ios-permissions/ (last accessed October 21, 2016).

[83] Nabi, R.M. and Mohammed, R.A. (2015) Smartphones Platform Security a Comparison Study., 5(11), pp.44–48.

[84] Nierbeck, A. (2016) IoT Analytics Platform. Available at: https://blog.codecentric.de/en/2016/07/iot-analytics-platform/ (last accessed January 20, 2017).

[85] Official Documentation, Oryx. Available at: http://oryx.io/docs/admin.html (last accessed September 1, 2017).

[86] Olalere, M., et al. (2015) "A Review of Bring Your Own Device on Security Issues", SAGE Open Apr 2015, 5 (2) 2158244015580372; DOI: 10.1177/2158244015580372.

[87] Oryx Cluster Setup (2017a). Available at: https://tecadmin.net/install-java-8-on-centos-rhel-and-fedora/ (last accessed September 1, 2017).

[88] Oryx Cluster Setup (2017b). Available at: https://www.digitalocean.com/community/tutorials/how-to-install-java-on-centos-and-fedora (last accessed September 1, 2017).

[89] Oryx project. Available at: https://oryx.io/ (last accessed September 1, 2017).

[90] O'Murchu, L., Falliere, N. (2011) "W32.Stuxnet dossier", Symantec White Paper, February 2011.

[91] Paho project. Available at: https://eclipse.org/paho/ (last accessed September 1, 2017).

[92] Population Reference Bureau (Aug. 19, 2008), 2008 World Population Data Sheet. Available at: http://www.prb.org/Publications/Datasheets/2008/2008wpds.aspx (last accessed 1 Sep. 2017).

[93] Rhee, K., Jeon, W. and Won, D. (2012) "Security requirements of a mobile device management system", International Journal of Security and its Applications, 6(2), pp.353–358.

[94] Scala. Available at: https://www.scala-lang.org/ (last accessed September 1, 2017).

[95] Scarfo, A. (2012) New security perspectives around BYOD. Proceedings - 2012 7th International Conference on Broadband, Wireless Computing, Communication and Applications, BWCCA 2012, pp.446–451.

[96] Schmidt, A.-D., et al. (2009) "Static Analysis of Executables for Collaborative Malware Detection on Android.", 2009 IEEE International Conference on Communications, doi:10.1109/icc.2009.5199486.

[97] Shabtai, A. U. Kanonov, and Y. Elovici (2010) "Intrusion Detection for Mobile Devices Using the Knowledge-Based, Temporal Abstraction Method", Journal of Systems and Software, Vol 83, No. 8, pp. 1524–1537.

[98] Skovoroda, A. and Gamayunov, D. (2015) Securing mobile devices: Malware mitigation methods. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 6(2), pp.78–97.

[99] Souppaya, M. and Scarfone, K. (2013) NIST Special Publication 800-124 Guidelines for Managing the Security of Mobile Devices in the Enterprise., p.30.

[100] Spark MLib, Apache. Available at: https://spark.apache.org/mllib/ (last accessed September 1, 2017).

[101] Spark, Apache. Available at: https://spark.apache.org/ (last accessed September 1, 2017).

[102] Team, U.-C.C.R. (2010) Technical Information Paper-TIP-10-105-01 Cyber Threats to Mobile Devices. US-CERT Technical Information Paper, pp.1–16.

[103] Understanding MQTT Topics and Topic Naming Design. Available at: http://www.steves-internet-guide.com/understanding-mqtt-topics/ (last accessed July 24, 2017).

[104] Venugopal, D., and G. Hu. (2008) "Efficient Signature Based Malware Detection on Mobile Devices", Mobile Information Systems, Vol 4, No. 1, pp. 33–49.

[105] Väisänen, T. et al. (2015) Defending mobile devices for high level officials and decision-makers, pp.1–107.

[106] Wassermann, G. (2015), "Vulnerability Note VU#924951 – Android Stagefright contains multiple vulnerabilities". CERT. Available at: https://www.kb.cert.org/vuls/id/924951 (last accessed November 2, 2016).

[107] Weiss, A. (2013) How to Deploy Enterprise Mobile Apps. Enterprise Apps Today. Available at: http://www.enterpriseappstoday.com/crm/how-to-deploy-enterprise-mobile-apps.html (last accessed October 21, 2016).

[108] Wikipedia, The Free Encyclopedia (2017a), Cell site. Available at: https://en.wikipedia.org/wiki/Cell_site (last accessed September 1, 2017).

[109] Wikipedia, The Free Encyclopedia (2017b), List of the largest information technology companies. Available at: https://en.wikipedia.org/wiki/List_of_the_largest_information_technology_companies (last accessed September 1, 2017).

[110] Wikipedia, The Free Encyclopedia (2017c), World population. Available at: https://en.wikipedia.org/wiki/World_population (last accessed September 1, 2017).

[111] Wikipedia, The Free Encyclopedia (2017d), Jevons paradox. Available at: https://en.wikipedia.org/wiki/Jevon%27s_paradox (last accessed September 1, 2017).

[112] Willems, C., T. Holz, and F. Freiling (2007) "Toward Automated Dynamic Malware Analysis Using CWSandbox", IEEE Security and Privacy, Vol 5, No. 2, pp. 32-39, March.

[113] Wu, F. and Chen, C., (2014) Sensitive Data Protection on Mobile Devices., 5(9), pp.38–41.

[114] YARN, Apache. Available at: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html (last accessed September 1, 2017).

[115] Zookeeper, Apache. Available at: https://zookeeper.apache.org/ (last accessed September 1, 2017).