# Software/OS Updates of Midair Drones

**Anders Vik Lysne**

Submission date: June 2019
Responsible professor: Peter Herrmann, ITEM
Supervisor: Tor Rune Skoglund, FourC

Norwegian University of Science and Technology
Department of Telematics

**Title:** Software/OS Updates of Midair Drones
**Student:** Anders Vik Lysne

**Problem description:**

The process of updating software usually requires the program that is being updated to restart in order to apply the update. This will interrupt the service it provides, which can be critical in certain cases. FourC develops an platform aiming to solve this issue, by deploying software in separate containers. This way, old software can provide service while the new software is being deployed. When the new container is ready, it takes over providing service and the old container is removed. This thesis attempts to demonstrate the platform developed by FourC using a drone. The thesis divided into the following tasks:

- Updating application software on the drone while being midair
- Updating the Operating System on the drone while being midair

**Responsible professor:** Peter Herrmann, ITEM
**Supervisor:** Tor Rune Skoglund, FourC

## Abstract

Embedded systems with IoT-devices are becoming increasingly popular, having great potential in many areas. Such systems are often vastly complex, as well as being deployed without the consideration of long-term maintenance and upgrades. Hence, maintenance of such systems can be difficult, intimidating and time consuming. Being able to perform maintenance by accessing and updating embedded systems remotely, preferably without interrupting its major services, would be beneficial to both consumers and system maintainers. This Master's thesis will attempt to demonstrate such functionality, by updating both application software and the underlying operating system of a drone while being airborne. Different drone setups for this task are evaluated, and the chosen one is described in depth. A solution proposal to realize application software updates is presented, as well as implementation process and testing results. The project has been advertised by the company FourC, and is based on three Bachelor's thesis conducted earlier.

# Preface

This thesis concludes my Master of Science education in Communication Technology at the Norwegian University of Science and Technology in Trondheim, Norway, and was carried out in in the spring of 2019 during my 10th semester. The project was advertised by the company FourC, and was one of the proposed topics from the Department of Telematics.

I would like to thank my external supervisor and CEO of FourC, Tor Rune Skoglund, for providing guidance and equipment for this project. I would like to thank my responsible professor, Peter Herrmann at the Department of Information Security and Communication Technology, for guidance related to content and writing style. Finally, I would like to thank Pål Sturla Sæther at the Department of Information Security and Communication Technology for assisting with equipment.

# Contents

**List of Figures** viii

**List of Tables** ix

**List of Acronyms** xi

**1 Introduction** 3

1.1 Motivation . . . 3
1.1.1 Topic area . . . 3
1.1.2 FourC . . . 4
1.2 Background . . . 4
1.2.1 First attempt . . . 4
1.2.2 Second attempt . . . 5
1.2.3 Third attempt . . . 6
1.3 Objectives . . . 6
1.4 Thesis Layout . . . 7

**2 Theory** 9

2.1 Basic Computer Terms . . . 9
2.1.1 Hardware . . . 9
2.1.2 Software . . . 9
2.1.3 Firmware . . . 9
2.1.4 Operating System . . . 10
2.1.5 Instruction Set Architecture and x86 . . . 10
2.1.6 Updates . . . 10
2.2 Drone . . . 11
2.2.1 Internal Measuring Unit . . . 11
2.2.2 Global Navigation Satellite System . . . 11
2.2.3 Rangefinder . . . 13
2.2.4 Optical Flow . . . 13
2.2.5 Ground Control Station . . . 14
2.3 The FourC Platform . . . 14

2.3.1 Internet of Things (IoT) . . . . . . . . . . . . . . . . . . . . . . 14
2.3.2 Cloud Services . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3.3 Containers . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.3.4 The Platform . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**3 Equipment and Planned Setup 17**
3.1 Early Choices . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.1.1 General Design Choices . . . . . . . . . . . . . . . . . . . . . 17
3.1.2 Choice of Drone . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Hardware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.1 Intel Aero Compute Board . . . . . . . . . . . . . . . . . . . . 20
3.2.2 Flight Controller . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.3 Intel Aero Vision Kit . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.4 Rangefinder . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.5 Power supply . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.3 Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3.1 FC Firmware . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3.2 MAVLink . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3.3 Ground Control Station . . . . . . . . . . . . . . . . . . . . . . 22

**4 Implementation 23**
4.1 Initial Steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.1.1 Disk Cloning . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.1.2 Flashing and Calibration . . . . . . . . . . . . . . . . . . . . . 23
4.1.3 Improving Center of Gravity . . . . . . . . . . . . . . . . . . . 24
4.2 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.2.1 Position Hold . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.2.2 Operating System and Containers . . . . . . . . . . . . . . . . 25
4.2.3 Time Shortage and Scope Reduction . . . . . . . . . . . . . . . 27
4.3 Implementation Method . . . . . . . . . . . . . . . . . . . . . . . 27
4.4 Proposed Solution to Midair Software Updates . . . . . . . . . . . 28
4.4.1 Connecting the Components . . . . . . . . . . . . . . . . . . . 28
4.4.2 Container . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.4.3 OS Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.5 Choices of Software . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.5.1 Flight Controller API . . . . . . . . . . . . . . . . . . . . . . . 30
4.5.2 Communication Between Drone and GCS . . . . . . . . . . . . 30
4.5.3 Software Environments . . . . . . . . . . . . . . . . . . . . . . 30
4.6 Some Approaches for OS Updates . . . . . . . . . . . . . . . . . . 33

**5 Implementation Process and Results 35**
5.1 First Flight . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

5.2 Objective 1: Software Updates . . . . . . . . . . . . . . . . . . . . . 35
5.2.1 Drone Control over HTTP . . . . . . . . . . . . . . . . . . . . 36
5.2.2 File Transfer . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.2.3 Container Management . . . . . . . . . . . . . . . . . . . . . 36
5.2.4 Test of Proposed Solution . . . . . . . . . . . . . . . . . . . . 37
5.3 Objective 2: Operating System Updates . . . . . . . . . . . . . . . . 39

**6 Conclusion** **41**
6.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.2 Solution proposal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.3 Possible Improvements . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.4 Future Research . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**References** **43**

**Appendices**

**A FlightControllerHandler** **45**
A.1 Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
A.2 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**B Container HTTP Server** **53**
B.1 Docker Image . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
B.2 Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
B.3 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**C Container Handler** **57**
C.1 Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
C.2 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**D Proxy Server** **61**
D.1 Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
D.1.1 Forward Class . . . . . . . . . . . . . . . . . . . . . . . . . . 61
D.1.2 ProxyServer Class . . . . . . . . . . . . . . . . . . . . . . . . 61
D.1.3 Container Host Functions . . . . . . . . . . . . . . . . . . . . 62
D.2 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

**E Ground Station HTTP Client** **67**
E.1 Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
E.2 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

**F Simulated Cloud Service** **71**
F.1 Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
F.2 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

G Application Setup 73

# List of Figures

2.1 Computer system stack. . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Different multirotor types. Propeller colors indicate direction of rotation. 12
2.3 Altitude/Throttle, Yaw, Pitch and Roll movement. Taken from [droa] . 13
2.4 FourC Platform Architecture. . . . . . . . . . . . . . . . . . . . . . . . . 16

3.1 System architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

4.1 Screenshot of calibrating accelerometer. Note: Screenshot is taken while using the PX4 FC firmware, but the calibration process with ardupilot is similar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 A pillar was blocking the battery from sliding further into the battery room. The red area shows where this pillar was located. . . . . . . . . . 26
4.3 The small metal lock taped to the front of the drone improves center of gravity. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.4 Network Architecture of the overall system . . . . . . . . . . . . . . . . 29
4.5 The OS layer receives an update from the cloud service. The FC container is informed to maintain position . . . . . . . . . . . . . . . . . . . . . 31
4.6 The OS layer creates a new FC container that contains the updated software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.7 Proxy server on the OS layer changes forwarding of ground station requests to the new FC container . . . . . . . . . . . . . . . . . . . . . . . . 32
4.8 The OS layer stops and removes the old FC container . . . . . . . . . . 32

# List of Tables

2.1 Different Rangefinders. . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.1 Different drone setups . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Intel Aero Vision Kit Camera Modules . . . . . . . . . . . . . . . . . . . . 21

5.1 Time delays experienced in solution proposal. . . . . . . . . . . . . . . . . 39

A.1 Explanation of some of the most important functions . . . . . . . . . . 45

C.1 Explanation of some of the most important functions . . . . . . . . . . 57

D.1 Explanation of the ProxyServer class functions . . . . . . . . . . . . . . 62
D.2 Explanation of the container host's functions, used for updates and container management . . . . . . . . . . . . . . . . . . . . . . . . . . 62

E.1 Explanation of the ground station HTTP client functions . . . . . . . . 67

# List of Acronyms

**AMQP** Advanced Message Queuing Protocol.

**AP** Access Point.

**API** Application Interface.

**BIOS** Basic Input-Output System.

**DSU** Dynamic Software Updating.

**FC** Flight Controller.

**FPGA** Field-Programmable Gate Array.

**FTP** File Transfer Protocol.

**GCS** Ground Control Station.

**GNSS** Global Navigation Satellite System.

**GPS** Global Positioning System.

**GUI** General User Interface.

**IMU** Internal Measuring Unit.

**Intel Aero RTF** Intel Aero Ready To Fly.

**IoT** Internet of Things.

**ISA** Instruction Set Architecture.

**LiDAR** Light Detection and Ranging.

**LiPo** Lithium Polymer.

**NTNU** Norwegian University of Science and Technology.

**OS** Operating System.

**RADAR** Radio Detection and Ranging.

**RAM** Random Access Memory.

**SITL** Software in the Loop.

**SONAR** Sound Navigation Ranging.

**TCP** Transmission Control Protocol.

**UAV** Unmanned Aerial Vehicle.

Chapter 1

# Introduction

This chapter presents the thesis' motivation and background, as well as the main objectives that will be addressed. A description of the thesis layout and the following chapters are presented.

## 1.1 Motivation

### 1.1.1 Topic area

Information systems are becoming an increasingly important part of society. Taking public bus transit as an example, many aspects rely on the availability of information systems. Time-charts are displayed on digital screens, accompanied with real-time data of when the next bus departure will occur. For this to be accurate, GPS data from every single bus has to be constantly available. Ticketing is being moved to the mobile platform, setting high availability demands on back-end payment systems. Such availability demands are poorly matched with software updates and system maintenance. Another problem connected to software updates and maintenance is the complexity of embedded systems. Updating one component might cause an error in a second component, possibly taking the overall system into a failed state and leaving it unable to deliver its service. Having vastly complex dependencies between components in a system, anticipating how a small software update will influence the overall service before run-time can be challenging. Hence, a maintainer can be frightened by having to execute software updates and interfere with an already working system. However, maintenance of information systems is a necessity, and the process of rolling out software updates is a part of a system's life cycle. Breaking a working system due to a software update is always possible, and measures have to be taken in order to overcome this issue. This project will address issues related to software updates in complex embedded systems.

### 1.1.2 FourC

This project has been advertised by the company FourC [Fou]. They are developing an Internet of Things (IoT) platform that connects single x86 IoT-devices to a central management system in the cloud. A lack of focus on long-term costs related to the operation of distributed solutions can put huge limitations in delivering extended functionality on top of the same hardware. As the situation is today, performing debugging, troubleshooting or even rolling out small security patches can be time- and cost intensive. Also, conducting software updates in embedded systems could potentially cause complications, making the system unable to deliver its service. Hence, FourC wants to enable remote software updates of IoT-devices, with the ability to rollback the last working software if a failure is detected.

This project is conducted solely to demonstrate the platform developed by FourC, as a marketing tool. The goal is to update some software on an airborne drone while using FourC's platform. As drones are not self-stable, they require input from sensors in order to maintain an airborne state. The challenge this project presents is to make a drone able to process sensor input and perform adjustments in the air, while also executing an update. One example of a successful demonstration could be to upload software that enables the drone to do some acrobatic maneuver, and make the drone perform this maneuver after the update is completed. All of this without the drone touching the ground.

## 1.2 Background

Three similar projects advertised by FourC have been carried out prior to this thesis. Two of them attempted to solve the exact same challenges as this thesis, but neither were able to present a working solution. A third group looked into different types of sensors that could be used to keep a drone in a stable hovering state. This thesis is based upon knowledge gathered by these projects.

### 1.2.1 First attempt

The first attempt to enable mid-air software and OS updates of drones through FourC was conducted as a half-year long bachelor project at Norwegian University of Science and Technology in 2016 [JST16]. The drone chosen in this project was Parrot's AR.Drone 2.0 Elite Edition [parb]. The x86 unit used was Intel Edison with corresponding Intel Edison Breakout Board [intc]. They stated that their x86 component did not offer direct support of Gentoo, even though this was one of the criteria from FourC. It was decided to ignore this, having found a guide others had used to install Gentoo on Intel Edison [genb]. With this setup they came close to realizing mid-air software updates, but they encountered one major problem. At the time, FourC's IoT-platform was not compatible with Intel Edison without performing

several modification of the software. This was not achieved due to lack of time at the end of the project, ultimately leaving them unable to install the FourC's OS on their x86 component. As a result, they were not able to test updates mid-air using FourC's IoT-platform. They also attempted to install Gentoo following the guide they had found. Despite others having succeeded doing this earlier, they were not able to install Gentoo themselves. However, they did perform a successful test of rebooting Intel Edison mid-air over Internet, while maintaining control of the drone afterwards. They argue that their developed system would work on any x86-unit that has a wifi-unit and supports Linux, and with a bit more effort it should be possible to achieve the goal of mid-air software updates.

One of the biggest possible improvements they stated was to automate the system, connecting modules by itself. This would save time and give a better user experience. Having basic configuration files with IP-addresses and port numbers would also ease development, not having to modify and recompile the source code every time these had to change. Serial communication between x86 and the drone as oppose to wifi was mentioned to reduce controller reaction time. Other suggested improvements included GUI for the manual controller, mobile application for demo purposes and security measures such as using WPA2 to make outsiders unable to connect to the drones local network.

### 1.2.2 Second attempt

The second attempt was conducted as another half-year long bachelor thesis at Norwegian University of Science and Technology [BA17]. Instead of using a commercial drone, it was decided to 3D-print and build the drone from scratch. A LattePanda board consisting of an x86 unit and an 8-bit microcontroller was used as the compute board [Lat]. The idea was to have the microcontroller work as an external FC, handling the drone's hovering state while updates were executed on x86. When updates were executed however, the microcontroller was not able to keep the drone stable. It was argued that vibrations in the frame could be creating unwanted noise in the sensors, as well as the sensors could be of poor quality. Installing FourC's OS was not completely successful either, having encountered issues in enabling LattePanda's wifi and bluetooth interfaces on top of the OS. Even though FourC was able to fix these issues, the fixed version was not installed or tested due to lack of time. Other problems included fragile and easy breakable 3D-printed components, as well as ordering wrong propellers due to inaccurate measurements. The latter was fixed by 3D-printing complex propeller adapters. As a combined result, executing updates on the drone while being airborne was not successful.

The decision of 3D-printing the drone from scratch increased the scope of the project drastically. All physical frame parts had to fit each other and second-hand

parts, which made up a long lasting design phase. Even though purchasing an existing drone probably would increase the total cost of the project, time spent on designing parts would be omitted. This had a major impact in [BA17], being unable to install FourC's OS due to lack of time. In addition, potential noise problems related to the 3D-printed frame made this approach seem even less rational in retrospect. Combining the x86 unit and a microcontroller on the same board was a great idea for separating the flight-controlling process and the updating process. However, as seen in the next section, the LattePanda's integrated microcontroller was not powerful enough to keep the drone in a stable hovering state by itself.

### 1.2.3 Third attempt

As a result of the unstable hovering state problem in [BA17], the third attempt investigated different approaches to maintain such a state on the same 3D-printed drone [HK18]. Their first modification was to simply dampen the vibration of the sensors and perform better balancing of the propellers. Even though this drastically reduced noise in the sensors' outputs and increased the drones stability, the drone was still unable to maintain a sufficiently stable hovering state. Out of all the evaluated sensors, optical flow combined with a SONAR sensor and the already existing gyroscope and accelerometer was chosen for further testing. Integrating optical flow and SONAR should increase the drones ability to maintain a fixed height and position, but realizing this turned out to be difficult. MultiWii, the software used to handle the sensor data, did not support height stabilization [Mul]. Furthermore, the microcontroller was not powerful enough to run optical flow algorithms while processing the gyroscope and accelerometer input. They were not able to perform a test of their setup, but argued that such a combination of sensors would be sufficient to maintain a stable hovering state of the 3d-printed drone.

Being unable to utilize sensors beyond the already existing accelerometer and gyroscope due to lack of computing power, it was suggested to use some other solution than the LattePanda. The integrated microcontroller on this board was too weak to handle the processing requirements, and a more powerful microcontroller should be used for flight control. Having an FC able to handle input from optical flow, accelerometer and gyroscope simultaneously should be sufficient to keep the drone in a stable hovering state. Hence, it should be possible to execute updates on a x86 chip while such an FC maintains position.

## 1.3 Objectives

The overall goal of this thesis is the same as in the projects mentioned above; executing software and OS updates on midair drones. Prior experiences had made it easier to make good decisions when it came to equipment and setup, and the people at

FourC had gained more competence regarding their platform. Combining knowledge gathered by prior projects with the increased competence and improvements of FourC's platform, this thesis was expected to provide stronger results than before. Two objectives are extracted from the overall goal, and was to be addressed in the following order:

1. Execute software updates at the application layer while being airborne
2. Execute OS updates while being airborne

In order to complete this project and deliver a solution that can be used to demonstrate FourC's platform in a suitable fashion, a drone able to run their platform is required. This platform currently runs a 32-bit Linux distribution based on Gentoo Linux [gena], designed to run on top of Intel's x86 Instruction Set Architecture (ISA) (Secttion 2.1.5). Having a circuit board that supports the x86 architecture is therefore a minimal requirement for this project. Besides this, sensors sufficient to keep the drone in a hovering state were also required. These two requirements would be sufficient to achieve objective 1, since the platform of FourC delivers a container-solution that can run multiple, isolated processes simultaneously. This will enable Flight Controller software to run in a dedicated container while updates are being downloaded and executed. In order to achieve objective 2 and update the underlying OS, a reboot would be necessary. If the reboot is quick enough, simply keeping the propellers rotating without sensor input might be sufficient. If the reboot takes some time however, having an external FC able to process sensor data and perform adjustments accordingly would be required.

As the project went by, problems related to FourC's platform made it challenging to realize both objectives. The first working implementation of FourC'sOS was received towards the end of the project, and a solution to objective 1 using other systems was close to being finished. It was decided to focus on completing objective 1 without using FourC's OS, and to completely omit objective 2 due to time running out. This is also mentioned in section 4.2.3

## 1.4 Thesis Layout

As this project is highly practical, performing heavy literature surveys and research are considered marginally applicable. Methods that incorporates collecting, validating and categorizing research does not apply to this thesis, and a complete methodology chapter is therefore omitted. The research that formed the background of this thesis is presented in section 1.2. The development methodology that was used for objective 1 was agile software development, and the complete implementation method is further described in section 4.3.

Theory, terminology and concepts related to drones and computing are presented in chapter 2. Chapter 3 presents the hardware and software that has been used, along with justification of the equipment setup. Chapter 4 presents the implementation of the proposed solution to objective 1. As the development process was agile, the results are tightly connected to the implementation process. Thus, the results are presented together with the implementation process in chapter 5. Chapter 6 gives a brief summary of the thesis, an evaluation of the solution proposal as well as presenting some future research areas.

# Chapter 2 Theory

This chapter presents some basic theory related to computers and drones. The major features of FourC's platform is also presented.

## 2.1 Basic Computer Terms

This section describes some basic computing terms, which also apply to Unmanned Aerial Vehicles (Section 2.2). Figure 2.1 shows how these terms are connected.

### 2.1.1 Hardware

Hardware is an encompassing term that refers to all the physical parts that make up a computer. The internal hardware devices that are essential to a computers functions, such as the CPU, RAM and hard drive, are called components. External hardware devices that are not essential to a computer's functions, such as a mouse and keyboard, are called peripherals.

### 2.1.2 Software

Software is a set of instructions or a set of programs that instructs a computer to perform specific tasks. Software exist on multiple layers in the computer stack. A single hardware component's firmware, a computer's operating system and applications that directly interact with the end user are all types of software. Scripts, applications, programs and instruction sets are all terms often used to describe software.

### 2.1.3 Firmware

Firmware is a type of computer software that provides the low-level control of a single hardware component in a device. Firmware usually refers to semi-permanent software and data that is part of a hardware component, and is specifically designed to optimize the single component's performance.

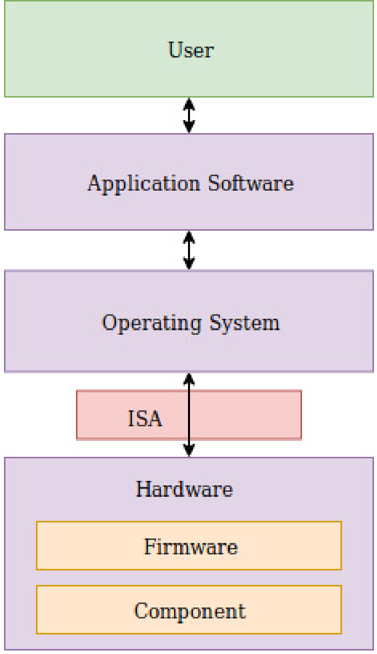



**Figure 2.1:** Computer system stack.

### 2.1.4 Operating System

An Operating System (OS) is the software that manages all the hardware and resources of a computer, providing services for computer programs. The OS controls processes, manages a computer's memory and file system and allocates resources to applications. Some of the most common Operating Systems are Linux and Windows for general-purpose computers, as well as iOS and Android for smart phones.

### 2.1.5 Instruction Set Architecture and x86

An Instruction Set Architecture (ISA) works as the interface between a computer's software and hardware. x86 is a a family of ISAs based on the Intel 8086 microprocessor and its 8088 variant. The Operating System used in this project runs on the 32-bit version of the x86 architecture.

### 2.1.6 Updates

Software updates are small enhancements of existing software that can provide new features, improve security or fix known issues of an application. The frequency and scope of updates vary, having some applications receive small updates multiple times a week while others receive comprehensive updates maybe once a year. Updates

can target all software layers, from a component's firmware to an application's General User Interface (GUI). As a computer has to load fixed code when starting an application, restarting the application after an update usually has to take place in order to apply its new enhancements. Hence, applying an OS update requires a reboot of the computer, while applying an update to a web browser requires restarting the web browser.

## 2.2 Drone

Also known as an Unmanned Aerial Vehicle (UAV), a drone is a flying robot that can be remotely controlled from a Ground Control Station (GCS) or fly autonomously using software-controlled flight plans in their embedded systems. Flight plans usually work in conjunction with internal sensors and GPS in order to accomplish desired missions. Multiple drone types exist, such as multirotor drones, fixed wing drones and single rotor drones. Multirotor drones are the most common type, and can be further classified based on its number of rotors. Figure 2.2 shows some common types of multirotor drones.

A drone is controlled by delivering power to each of its rotors. The terms yaw, roll, pitch and throttle is used whenever talking about UAVs, and their meanings are presented in figure 2.3. As the flying environment is dynamic, with constant changes in areas such as in pressure and wind, controlling a drone with no other input than a manual controller would be extremely difficult. Sensors are used in order to reduce the difficulties introduced by the ever changing environment, and automatically adjusts yaw, roll, pitch and throttle to simplify drone control. The following sections describe some of the most common sensors used.

### 2.2.1 Internal Measuring Unit

The Internal Measuring Unit (IMU) is the most essential component of a drones flight control system. The IMU consist of one or more accelerometers to calculate the drone's non-gravitational acceleration, and one or more gyroscopes to calculate its rotational rate. By combining output from both accelerometer and gyroscope, the IMU is able to maintain stability of the drone while being airborne. Some IMUs also have an integrated mangetometer used as a compass, which can determine orientation and the drone's heading direction.

### 2.2.2 Global Navigation Satellite System

GNSS is used to determine an object's position, by calculating the distance between the object and a set of satellites. A drone can use such data to maintain a constant position, or perform autonomous flights from one position to another. This system

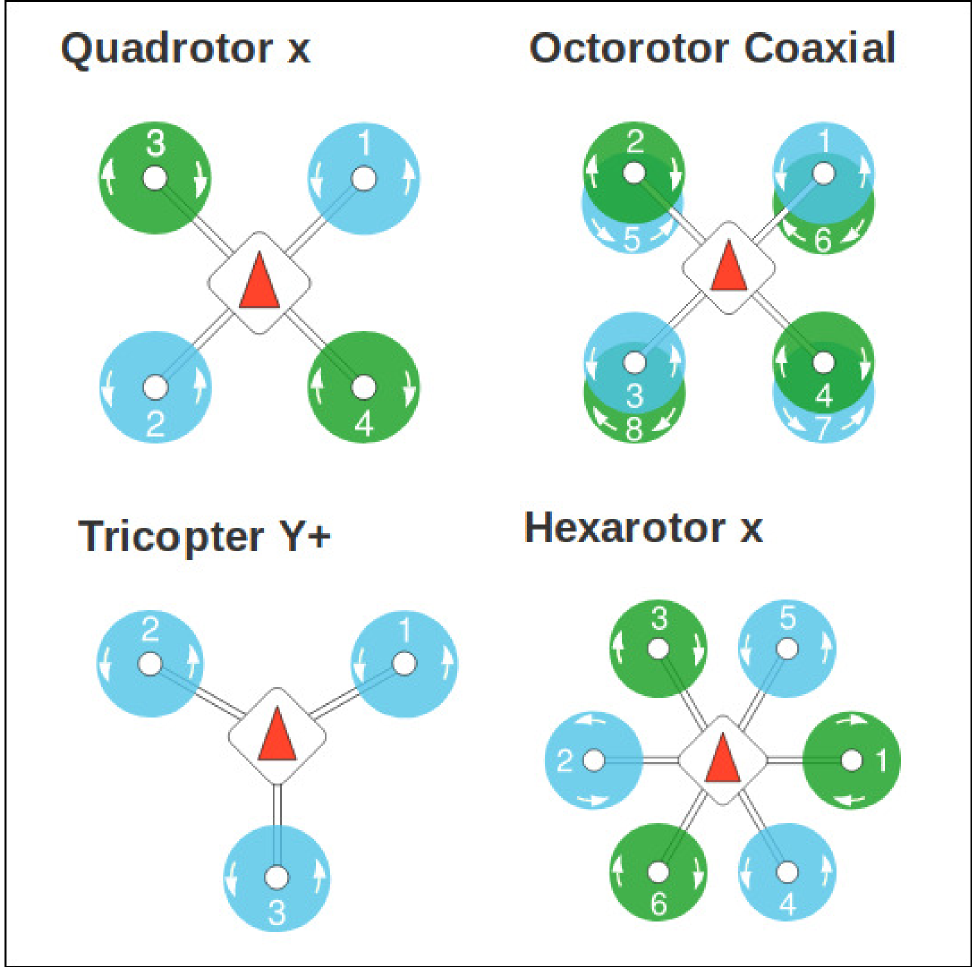



**Figure 2.2:** Different multirotor types. Propeller colors indicate direction of rotation.

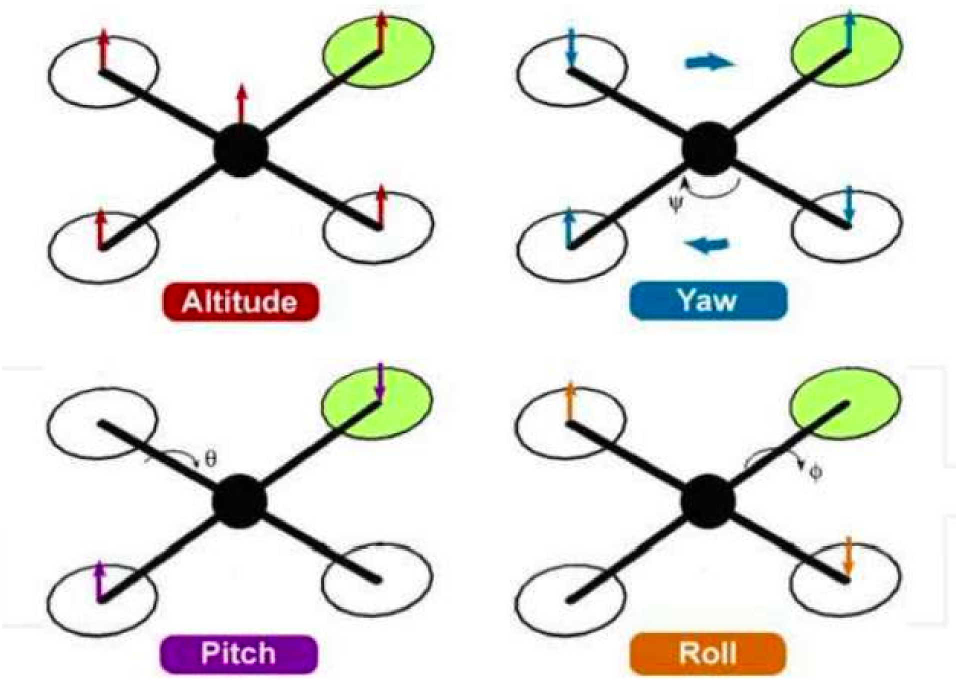



**Figure 2.3:** Altitude/Throttle, Yaw, Pitch and Roll movement. Taken from [droa]

is normally restricted to outside use, as it requires more or less a clear line of sight between the object and GNSS satellites. Another restriction is that some geographical areas are not covered by such satellites, and GNSS signals can thus be unavailable.

### 2.2.3 Rangefinder

A rangefinder is a sensor that is able to measure the distance between an observing object and a surrounding target. There exist different technologies that enables such measurements, and some of the most common ones are described in table 2.1.

### 2.2.4 Optical Flow

Optical flow or optic flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer and a scene. An optical flow sensor is a vision sensor capable of measuring optical flow or visual motion [opt], resulting in output that can be used for position holding. In contrast to GNSS positioning, optical flow are not dependant upon input from

| Technology | Description |
|---|---|
| RADAR | Uses radio waves to measure distance. Requires a source (transmitter) and target (receiver) to work. |
| SONAR | Uses sound propagation to measure distance. Used mainly underwater, as sound does not propagate well in air. |
| LiDAR | Illuminates a target with pulsed laser light and measures the reflected pulses with a sensor. Used to measure distance and make 3D representations of the target. |

**Table 2.1:** Different Rangefinders.

remote systems. Hence, optical flow can be used for position hold in environments where GNSS signals are weak or non-existent, such as indoors.

### 2.2.5 Ground Control Station

A Ground Control Station (GCS) is a land-based control centre that provides tools for human control of UAVs. Smaller UAVs are usually operated by a single portable radio transmitter. Extending this transmitter with a display, data and aerial video telemetry, creates what is effectively a GCS.

## 2.3 The FourC Platform

As stated in section 1.1.2, the main motivation behind this thesis was to demonstrate the platform developed by FourC. This section describes concepts related to their platform and how the platform works.

### 2.3.1 Internet of Things (IoT)

IoT refers to the concept of having simple devices being able to communicate and exchange data over the Internet. Such devices can be equipped with different types of hardware, such as sensors and actuators, and can be integrated in complex embedded systems, working together to perform a greater common task. IoT devices are highly specialized to perform specific tasks, and aspects that does not directly contribute to their functionality are often disregarded.

### 2.3.2 Cloud Services

When talking about the Cloud in the context of computers and the Internet, one refers to the concept of storing data somewhere else than locally on a single computer.

Usually, this is at a centralized location that implements multiple security measures, such as data redundancy and encryption. The data can be accessed from any remote end station, given that the end station is connected to the Internet and is authorized to access the data. Some examples of Cloud Services are Google Drive, iCloud, Netflix and Amazon Web Services.

### 2.3.3 Containers

In terms of computing and Operating Systems, a container is a set of one or more processes that is isolated from the rest of the system. A container packages up code and all its dependencies into a closed environment, running it as a stand-alone application. Containers provides benefits in multiple areas, and can be used for several purposes. Development can be made more efficient, as a container can easily replicate the development environment, testing environment and production environment of an application. They can also be used as a sandboxing tool for security reasons. Containers have many similarities to virtual machines, but are not as comprehensive as they do not simulate the underlying hardware. They sit on top of a host OS, sharing its resources such as libraries, binaries and kernel.

### 2.3.4 The Platform

FourC develops a platform that targets IoT devices and embedded systems. The platform is comprised by a 32-bit Linux OS based on the Gentoo distribution, and a Cloud Service that communicates with the OS over the Internet. The goal is to ease maintenance of embedded systems, by connecting IoT devices to a centralized management system in the Cloud. The main feature of their OS is the container solution, making it able to communicate with the cloud service while still running intended processes. This is done by running these processes in separate containers. The OS runs on the x86 ISA and communicates with the central management system over AMQP. Figure 2.4 displays the architecture of FourC's platform with a single IoT device and the backend Cloud Service.

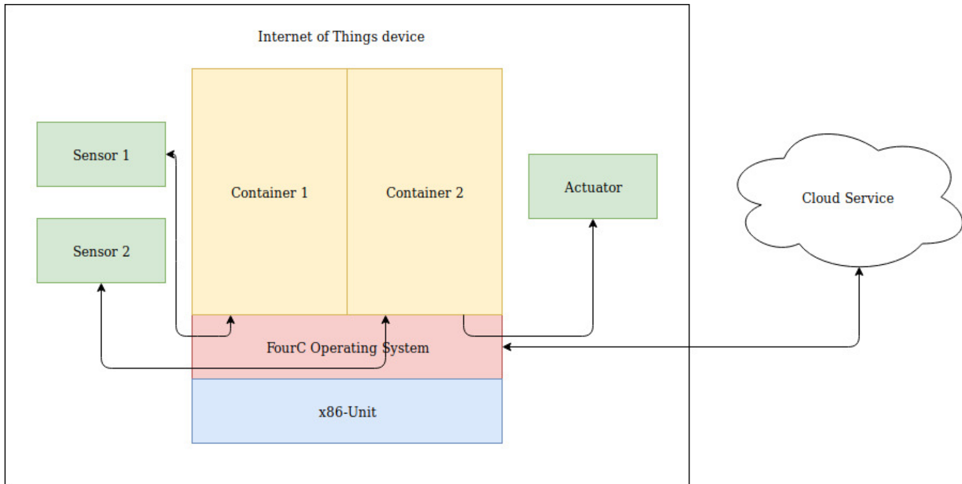



**Figure 2.4:** FourC Platform Architecture.

# Chapter 3
# Equipment and Planned Setup

This chapter describes choices in setup and equipment. Three different setups are evaluated, and the chosen one is described in greater detail.

## 3.1 Early Choices

### 3.1.1 General Design Choices

[BA17] decided to design a drone from scratch and use a 3D-printer to create the necessary parts. Even though this approach had advantages like quick replacements of broken parts and customizing the frame to fit a given compute board, a significant amount of disadvantages also existed. First, as already mentioned in section 1.2.2, having to design the parts from scratch were extremely time consuming. This project has a comprehensive scope without including the aspects of designing physical drone parts. Taking the project's duration into account, using an existing drone on the market and solely focus on the objectives stated in section 1.3 was considered a better approach. By using a commercial drone, issues related to sensors and general flight performance would probably be avoided. Being able to quickly start developing and testing seemed important, seeing projects conducted earlier had run into shortage of time.

In order to achieve objective 2 without knowing how long a reboot will last, having an external FC that can process sensor data while x86 undergoes a reboot was chosen. An overview of the planned system architecture is shown in figure 3.1. The 8-bit microcontroller integrated on the LattePanda that was used in [BA17] and [HK18] was too weak to process data from more sensors than gyroscope and accelerometer simultaneously. Hence, a more powerful microcontroller was chosen for this project.

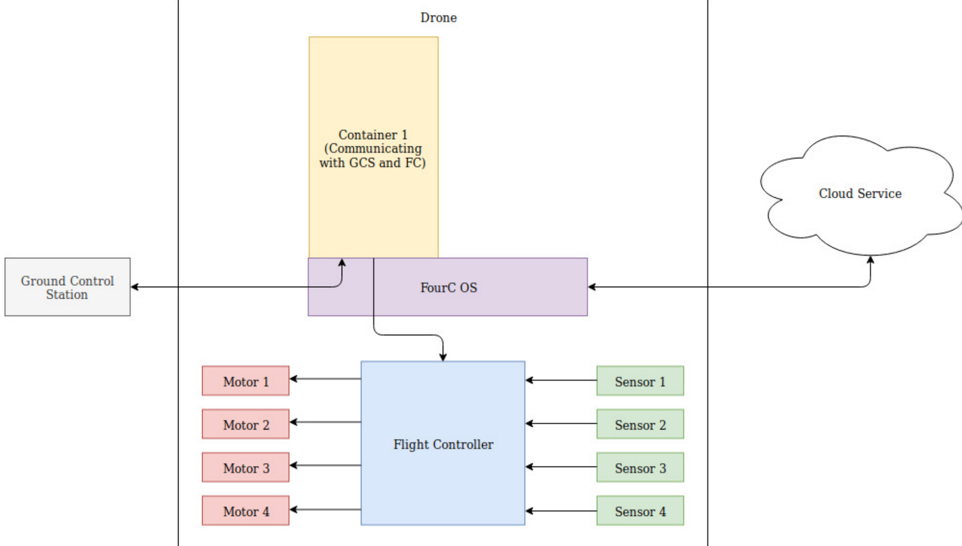



**Figure 3.1:** System architecture

### 3.1.2 Choice of Drone

Following the decision of using a drone on the market as a starting point, different approaches could be taken. The first one was to take the same approach as [JST16], having an external x86 unit "piggy-backed" on a drone already equipped with a dedicated FC, and attempt to connect them together. A second approach was to use a drone already equipped with an x86-unit, attaching a microcontroller and transferring FC functionality to this component. The third and possibly best approach was to use a drone that was already equipped with a dedicated FC as well as an x86 companion computer. Table 3.1 shows some setups that was evaluated for this project.

| Drone | Intel Aero RTF [intb] | Qualcomm snapdragon Flight [sna] | Parrot bebop [para] |
|---|---|---|---|
| Compute board | Intel Aero | Qualcomm Flight | N/A |
| x86-unit | Yes | No | No |
| External FC | Yes | Yes | Yes |
| Relevant interfaces | Wifi, Bluetooth, USB 3.0 | Wifi, Bluetooth, USB 3.0 | Wifi |
| Autopilot software | PX4, ardupilot | PX4, ardupilot | PX4 |
| Price(NOK) | 10 000 | N/A | 4 000 |
| Requires additional components | Battery | Frame, propellers, motors, x86-unit, battery | x86-unit, battery |
| Comments | Designed to get applications airborne quickly. No additional hardware besides battery required | Good for development, but does not support x86 architecture. Needs much additional hardware | Results in more or less the same setup as in [JST16], coming close to realize objective 1 and 2. Has to piggyback an x86-unit |

**Table 3.1:** Different drone setups

Having sufficient funding from FourC to purchase the Intel Aero RTF, this ended up being the drone of choice. This quadcopter was designed to let developers start developing and testing right away, without having to spend time on connecting hardware and components from different vendors. Using an external FC accompanied by an x86-unit had been successful in past projects, and this architecture would be preserved with Intel Aero RTF. The included FC also came with more power than the microcontroller on LattePanda. Combined with a carbon fiber frame rather than a 3D-printed plastic frame, possible problems in maintaining a stable hovering state should not be caused by the physical components. All components are designed to be interoperable, hence running into problems with hardware not working appropriately against each other seemed unlikely. On the software side, Intel Aero RTF is based on the Dronecode project [drob], making it compatible with multiple open source frameworks[PX4][ard][droc] and the standard drone communication protocol [mav], all which would be highly useful to achieve objectives 1 and 2. On top of this, Intel has a huge community able to assist if unforeseen problems should occur, as well as plenty of documentation regarding their drone platform. Having considered different setups with the issues presented in [JST16][BA17][HK18] in mind, the Intel Aero

RTF seemed like the obvious choice.

## 3.2 Hardware

### 3.2.1 Intel Aero Compute Board

The compute board on Intel Aero Ready To Fly is the Intel Aero compute board, a purpose-built UAV developer kit geared towards developers, researchers and drone enthusiasts. The Intel Aero has 4GB of RAM, industry standard interfaces and reconfigurable I/O to facilitate connecting to a broad variety of drone hardware subsystems. The Intel Aero compute board runs an OS based on the Yocto project by default, which is preconfigured and highly customized for the compute board. In addition to the Yocto build, full native Ubuntu* with Intel drivers is also verified by Intel as user installation. A complete overview of the Intel Aero's specs can be found in [Inte]

### 3.2.2 Flight Controller

The Intel Aero RTF comes with a dedicated FC, consisting of a STM32 microcontroller, IMU, magnetometer and altitude sensors. The FC connects with the x86-unit over HSUART serial communication and communicates using the MAVLink protocol. It can run both Ardupilot [ard] and PX4 [PX4] as the autopilot firmware, and the user can choose either one.

### 3.2.3 Intel Aero Vision Kit

The Intel Aero Vision Kit contains 3 different camera modules, and is included in the Intel Aero RTF. This Kit is mainly aimed towards depth sensing applications, such as 3D scanning and surface recognition. Objects can be measured and incorporated into real-world spaces to visualize possibilities, and users can create artistic and creative depth-based filters. Combined with a drone's mobility, this technology opens up numerous possibilities. The kit provides support for optical flow, which was planned to be used for position hold. Table 3.2 describes the different camera modules comprised in the kit.

### 3.2.4 Rangefinder

In order to enable optical flow on Intel Aero RTF, an additional rangefinder has to be installed. A rangefinder is used to determine the distance between an object and its surroundings, which can be based on different types of measurements. In this project, Benewake's TFMini LiDAR[Ben] was supposed to be used. This is a low cost, small volume and low power consuming LiDAR with range between 0,3 and 12

| Camera Module | Specification | Orientation and Intent |
| --- | --- | --- |
| Intel® RealSense™ R200 camera [Intd] | Depth Sensing and Vision Camera | Oriented forwards. Intended for depth-sensing purposes such as collision avoidance and 3D surface reconstruction |
| OmniVision* OV8858 [ov8] | 8 MP RGB Camera | Oriented forwards. Intended for regular video capturing |
| OmniVision* OV7251 [ov7] | VGA Camera | Oriented downwards. Intended for optical flow algorithms |

**Table 3.2:** Intel Aero Vision Kit Camera Modules

meters. The objectives requires the drone to maintain a stable hovering inside at low altitudes, and the TFMini combined with optical flow would suit this purpose.

### 3.2.5 Power supply

Even though the Intel Aero RTF is "ready to fly", it does not come with any power supply. The power interface on the Intel Aero RTF is XT60, and an external battery with XT60 connector had to be purchased. Intel recommends a LiPo battery with 4 cells that delivers 4000-4500 mAh for their drone. Additionally, it was considered useful to have a regular socket power supply for development purposes. The power supply specs used in this project are described below.

**Battery:**

- Type: LiPo
- Cells: 4
- Nominal voltage: 3.7V Per cell (14.8V total)
- Capacity: 4500mAh

**Wall Power Adapter:**

- Input: 100-240V AC
- Output: 12V DC, 5A

LiPo batteries requires a balance charger able to charge and balance the voltage of the battery's individual cells. The charger used in this project was a fake version of SkyRC's imax B6[Sky].

## 3.3 Software

### 3.3.1 FC Firmware

As stated in section 3.2.2, the Intel Aero FC supports two types of firmwares. The first one is PX4, which is the one Intel suggests and the one installed on their FC by default. This is an open source FC software for drones and other unmanned vehicles. PX4 is part of the dronecode project, a platform that contains everything needed for a complete drone solution. This platform includes FC hardware, ground control station, developer APIs as well as the autopilot software PX4. Using such a platform ensures that everything needed for a complete drone solution is delivered in a product that is a well integrated, well tested, easily modifiable and consistently licensed. Intel is part of the dronecode project, and their Intel Aero RTF is based on this platform.

The second type of firmware is ardupilot. This autopilot software has been in development for longer than PX4, and is claimed to be *"the most advanced, full-featured and reliable open source autopilot software available"* and *"the most tested and proven autopilot software"* on their website [ard]. Out of the two, Ardupilot is regarded as the more mature FC software in the UAV community. Both will be tested in this project.

### 3.3.2 MAVLink

MAVLink is a protocol for communicating with small unmanned vehicles. It is mostly used for communication between a Ground Control Station and the vehicle, but can also be used locally for inter-communication between the vehicle's subsystems. Intel Aero RTF uses MAVLink for all internal communication as well as with the GCS. The protocol can be interfaced through python, using either pymavlink [pym] or dronekit [droc]. Dronekit uses pymavlink as an underlying library, and this project will implement and test both dronekit and pymavlink.

### 3.3.3 Ground Control Station

The GCS that is used as an example in Intel's documentation regarding their UAV platform is QGroundControl [qgr]. This is an open-source GCS that provides full flight control and mission planning for any MAVLink enabled drone. It runs on multiple platforms such as Windows, OS X, Linux, iOS and Andorid. This is also the GCS software that was used in this thesis.

# Chapter 4
# Implementation

This chapter presents the implementation, as well as decisions and limitations that were made. A solution proposal to objective 1 and the implementation method are presented. Finally, some approaches to objective 2 are presented.

## 4.1 Initial Steps

### 4.1.1 Disk Cloning

As stated in section 3.2.1, the Intel Aero Ready to Fly Drone came shipped with Yocto Linux preinstalled. As the main motivation for this project was to demonstrate the platform of FourC, installing their Gentoo-based linux distribution would be required. The biggest issue in the previous projects had been installing FourC's OS on the compute boards. Each of the three prior attempts ran into some trouble with this task, from not being able to install the OS at all, to having some components of the board not being recognized due to unsupported or missing drivers. As there were uncertainties that Intel had some specialized drivers for the Intel Aero's proposed OSs, a complete clone of the Intel Aero disk as when the drone was received was executed.

### 4.1.2 Flashing and Calibration

Before a flight of the Intel Aero could take place, multiple parts of the drone had to be flashed. This is to ensure the latest firmware and software is installed, which hopefully will give users the best possible first flight experience. Instructions on how to do this was found on the Intel Aero github page[inta]. The components that required flashing were:

- Linux Operating System
- BIOS

– FPGA

– Flight Controller

As stated in section 3.3.1, two firmwares can be used as the FCs autopilot firmware. First is the PX4 autopilot, which was the one that is installed on Intel Aero RTF by default. The second choice is ardupilot, an open source autopilot that have been in development for longer than PX4. Even though Intel Aero uses PX4 by default, Ardupilot was chosen when flashing the flight controller the first time. This was because ardupilot have had more exposure and runs on more hardware than PX4, hence giving the impression of being the more mature FC autopilot software of the two.

After flashing of the different components was completed, the drone and the radio transmitter had to be calibrated. This was done using GCS software on a separate device, in this case QgroundControl on a computer running ubuntu. A complete calibration process needs to be performed any time the FC firmware changes. The compass should also be re-calibrated whenever the physical environment changes, due to possible differences in the magnetic field. Figure 4.1 displays a screenshot of accelerometer calibration using GCS and the PX4 firmware.

### 4.1.3 Improving Center of Gravity

The battery that was used for the Intel Aero RTF was just barely small enough to fit inside the battery room. With the battery attached the drone got quite tail-heavy, which influenced the flight performance. In order to improve the center of gravity, a pillar was removed so that the battery could slide further toward the center of the drone. Additionally, a small metal lock was taped to the front part of the drone. The weight of the lock was just about right to get a close to perfect center of gravity. Even though the overall weight of the drone increased, the battery and motors were more than powerful enough to carry the extra weight. Figure 4.2 and 4.3 shows the drone with improved battery placement and the taped metal lock respectively.

## 4.2 Limitations

### 4.2.1 Position Hold

Making the drone able to maintain a fixed position autonomously while being midair, sensors beyond accelerometer and gyroscope has to be used. The accelerometer and gyroscope provides data regarding the drone's physical planing angle, but does not provide data that can be used for positioning. GPS and optical flow are positioning techniques that were considered in this project. As the original plan was to enable

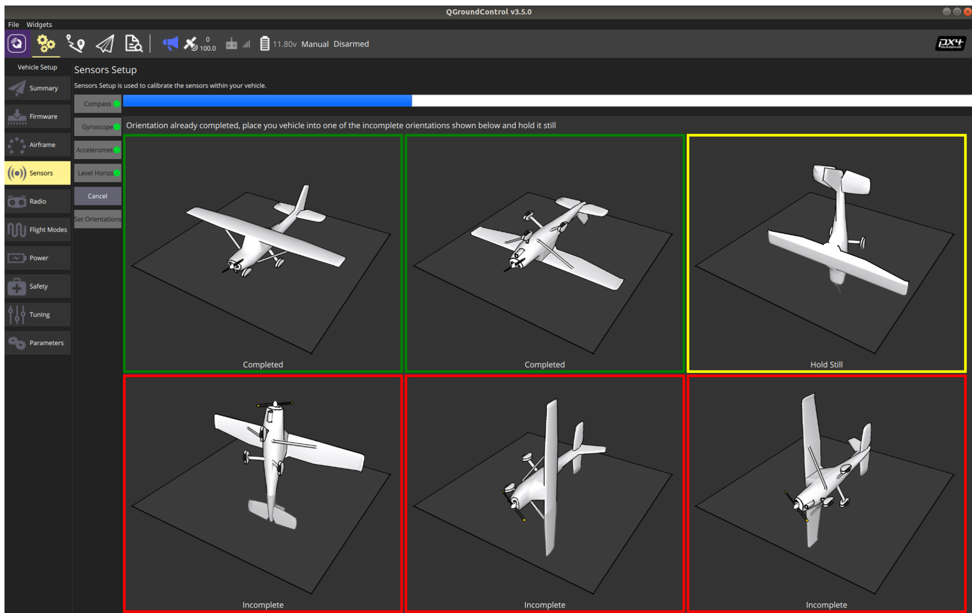



**Figure 4.1:** Screenshot of calibrating accelerometer. Note: Screenshot is taken while using the PX4 FC firmware, but the calibration process with ardupilot is similar.

position hold while being inside, GPS would be insufficient caused by the high probability of weak or non-existing GNSS signals. A TfMini rangefinder was ordered to enable the use of optical flow, but the rangefinder was never received. As a result, GPS remained the only option for position holding. The objectives were still solvable, but now with the limitation of having available GNSS signals.

### 4.2.2 Operating System and Containers

The potential challenge of installing FourC's OS on the Intel Aero ended up being a real issue. Getting their OS to work took way more time than expected, and as the project was coming to an end an alternative solution was discussed. Instead of using the container solution of FourC, the drone would run docker containers[doc] on top of its default OS instead. Intel Aero had been designed with the possibility of running docker containers in mind, and this solution would keep the original concept of using containers to maintain service availability while executing updates. FourC are also planning to integrate docker containers in their platform in the future. Caused by the delayed acquisition of a working OS from FourC, a solution using the docker alternative was pursued.

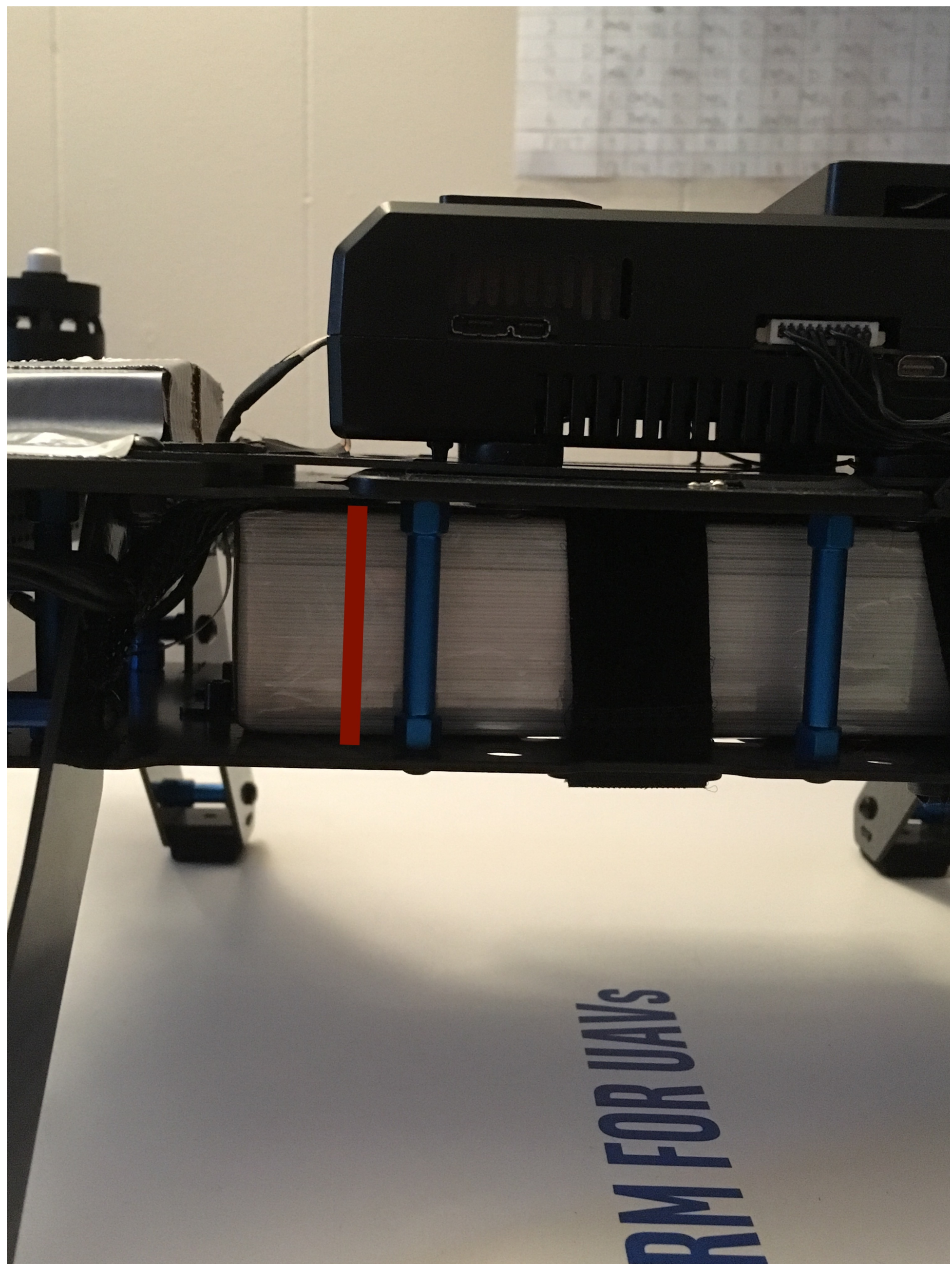



**Figure 4.2:** A pillar was blocking the battery from sliding further into the battery room. The red area shows where this pillar was located.

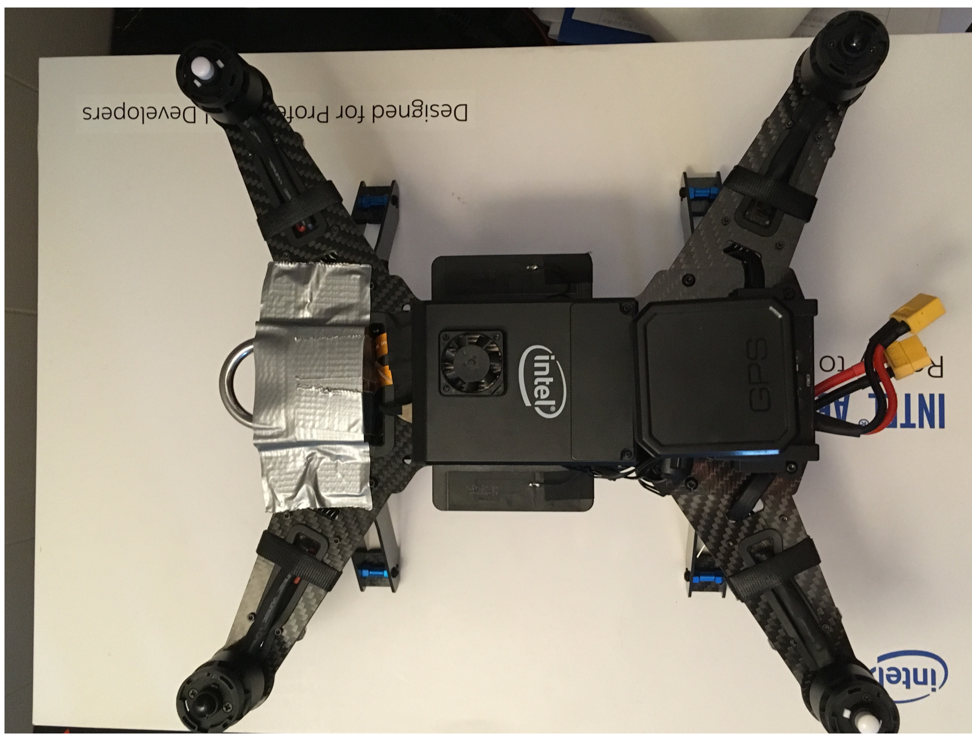



**Figure 4.3:** The small metal lock taped to the front of the drone improves center of gravity.

### 4.2.3 Time Shortage and Scope Reduction

Much of the equipment that was to be used in this project arrived much later than expected. This lead to reduced time for implementing and testing possible solutions. Also, the process of implementing a solution towards objective 1 went on for longer than expected at the beginning of the project. Because of these reasons, it was decided to focus on realizing midair software updates and omit objective 2. Some approaches that can be taken to solve the challenge of midair OS updates are presented in section 4.6

## 4.3 Implementation Method

Development of the solution presented in section 4.4 was achieved using an agile software development process. Sub problems related to objective 1 were isolated and solved separately before they were stitched together. As there could be misunderstandings related to how FourC's platform was operating, being able to redefine

system requirements and proposed solutions during development was considered a necessity. Also, having the ability to test portions of the software continuously would quickly reveal false or sub-optimal solutions.

Having no prior experience controlling a quadcopter or any other types of UAVs, it was beneficial to use a big and lucid testing area. As a some of the testing was performed during the winter season while the ground were mostly covered by snow, a suitable landing ground had to be found. The solution was a football field with artificial turf. The football field was big, flat and was cleared of snow regularly, making the ground sufficiently dry for a drone to land. Additionally, artificial turf is a bit soft which would reduce the consequences of a hard crash. All flight tests were performed on this football field around noon during weekdays, as the field was available and free of people at these hours.

Everything related to drone behaviour was tested using the dronekit_sitl simulator before physical tests at the testing area were performed. Some physical testing was also performed on a desk without attaching propellers, prior to taking the drone to the testing area. This was to make sure the communication aspect between the drone and GCS was working correctly. To ensure that the drone had actually been updated, attempts to execute the updated functionality were performed both prior and after the update had been transferred and deployed.

## 4.4 Proposed Solution to Midair Software Updates

### 4.4.1 Connecting the Components

Even though FourC's platform has not been used, a similar setup has been implemented. A separate laptop was used to simulate the backend cloud service of FourC, and the same laptop also worked as the drone's ground station. The drone hosts a closed network with a wifi Access Point (AP), and the laptop connects to the closed network through the drone's AP. Figure 4.4 displays the network architecture of the implemented system. The main difference in this setup is the removal of the Internet between the cloud service and the drone, using a direct link instead. As the transport protocol would be TCP in both scenarios, which ensures that packets are delivered to the application layer in the same order, this architecture was considered to be a sufficient alternative for transferring updates.

### 4.4.2 Container

The software that controls the drone behaviour is run within a container, hereby referred to as an *FC container*. The FC container consists of two components. The first component is the FC handler, which communicates with the drone's physical Flight Controller. The second component is an HTTP server, which receives HTTP

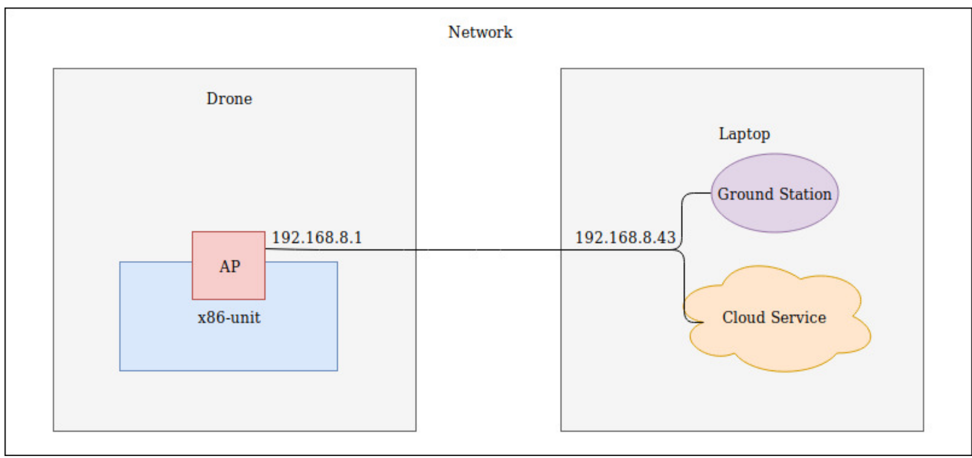



**Figure 4.4:** Network Architecture of the overall system

requests from the ground station. Any HTTP client that is connected to the drones AP can send requests to the drone, which are forwarded to the HTTP server inside the current FC container instance. The HTTP server makes further calls to the drones Flight Controller through the FC handler. The client can send HTTP POST request to change the behaviour of the drone, such as elevating to given altitudes or changing flight modes. The full documentation on container environment and initialization, as well as the code running within the container can be found in appendix B. Documentation and code regarding the client side can be found in appendix E.

### 4.4.3 OS Layer

The OS of FourC works as a container host. It manages all containers and their states, as well as communicating with the backend cloud service. The OS layer in the solution proposal mimics this behaviour. The OS hosts a proxy server, which forwards all requests from the ground station to the current FC container instance. Requests from the cloud service are handled directly on the OS layer, and do not interfere with the FC container. Whenever an update regarding the FC container is received, the OS informs the current FC container to keep the drone in a fixed position. It then creates a new FC container instance that contains the updated software. Once the new FC container has been initialized and is ready to start operating, forwarding of GCS requests are changed to this container. When all the steps have been completed and the new FC container has taken over as the current functioning instance, the old FC container is stopped and removed. Figures 4.5, 4.6, 4.7 and 4.8 shows the different steps taken, from receiving an update from the cloud

service to having them deployed in a new FC container. Documentation and code for the proxy server and container management can be found in appendix D and C respectively.

## 4.5 Choices of Software

### 4.5.1 Flight Controller API

After completing the first flight (Section 5.1) and making sure the FC and other components were working as expected, the next step was to execute drone commands through code without using the radio transceiver. The first approach to do this was to use the pymavlink Python library, which makes it possible to send MAVLink messages to the FC [pym]. A simple script that enabled arming and disarming of the drone was written, but extending the functionality through pymavlink was challenging. pymavlink was a low-level MAVLink API with poor documentation, which made it hard to understand function parameters and their possible values. A second approach was taken, using the dronekit API instead [droc]. Dronekit was an easier to understand, higher-level API that also provided better documentation than pymavlink. The second approach made it easier to write code towards the drone's FC, and dronekit became the API of choice.

### 4.5.2 Communication Between Drone and GCS

To enable communication between the drone and GCS, it was decided to implement a simple HTTP server on the drone. Since python was already used for communication withy the FC, the HTTP server was written in python as well. Flask is a simple python library for web development, and was chosen for implementation of the HTTP server [fla]. The HTTP client code was written using the requests API, which is a python framework that generates HTTP requests [req].

Since the HTTP server would run inside a container, subject to being removed at some point, it was decided to create a proxy server running on the OS layer. The proxy server would forward HTTP requests to the current FC container instance, enabling the client to send requests to a fixed IP address of the proxy server. This way, container management would take place at the OS layer, concealed from both the HTTP server and HTTP client. Using flask when writing the proxy was attempted, but for unknown reasons flask would not compile on the Intel Aero RTF's OS layer. The proxy was re-written using the built-in python sockets instead [pyt].

### 4.5.3 Software Environments

The drone itself had three different software environments to consider. The first one was the OS environment, that would work as the container host. The second one was

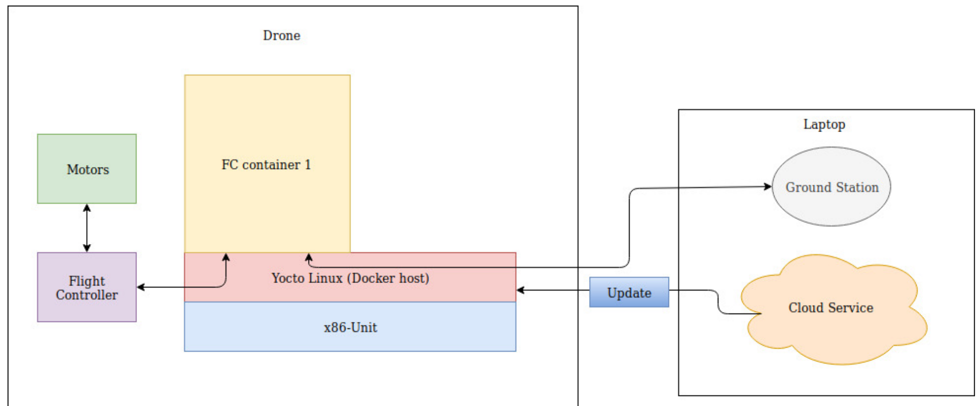



**Figure 4.5:** The OS layer receives an update from the cloud service. The FC container is informed to maintain position

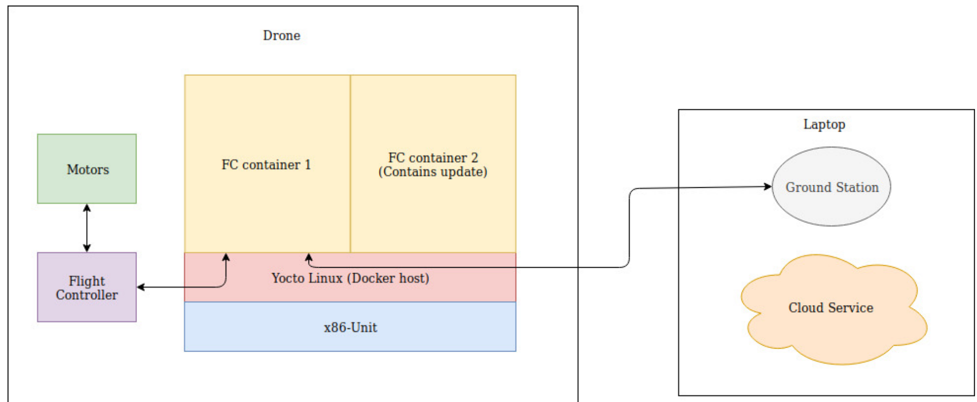



**Figure 4.6:** The OS layer creates a new FC container that contains the updated software

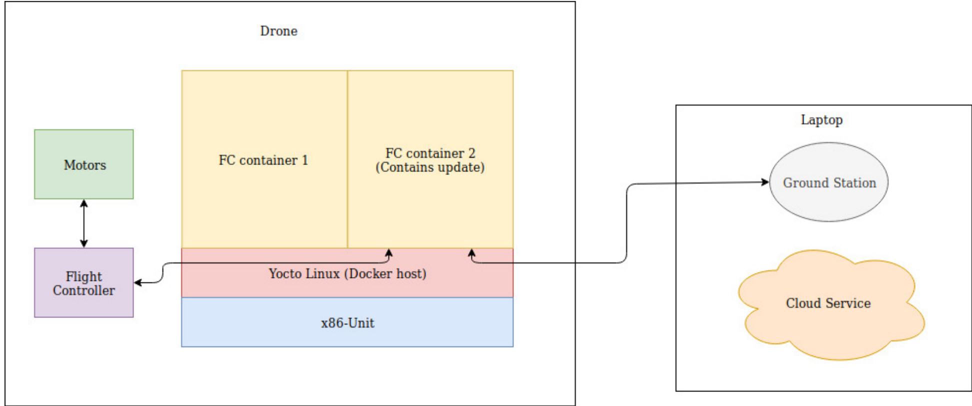



**Figure 4.7:** Proxy server on the OS layer changes forwarding of ground station requests to the new FC container

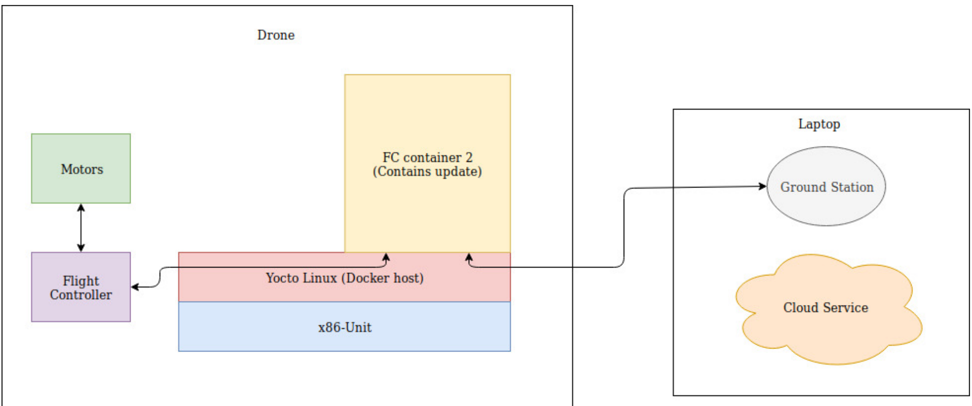



**Figure 4.8:** The OS layer stops and removes the old FC container

the container environment, running an HTTP server and FC code. The third one was the physical Flight Controller. Each software environment is briefly described below

#### OS Environment

The OS works as the container host, and runs a Yocto Linux distribution. This is the default OS of Intel Aero RTF. On top of the OS, docker and python 2.7 is used to manage containers and communicate with the GCS.

#### Container Environment

The container is based on a Ubuntu 18.02 docker base image, with python 2.7 and third-party python modules on top. More information on the container image, third-party python libraries as well as the code can be found in appendix B. Link to the actual container image that was used can be found in [and]. Version 3 of the image was used.

#### Flight Controller Firmware

As described in section 4.5.1, the dronekit API was chosen to communicate with the FC. The people behind dronekit has also developed a Software in the Loop (SITL) simulator, dronekit_sitl [drod]. The simulator simulates an Ardupilot version 3.3.0 Flight Controller, making it easy to test code without performing physical flights. To best utilize this feature, Ardupilot was chosen as the FC firmware. The Ardupilot version used on the drone was 3.5.7 however, but it was considered to be almost identical to version 3.3.0.

## 4.6 Some Approaches for OS Updates

Operating System updates usually requires a complete reboot. Doing this while also maintaining a stable hovering state is a challenge, and two approaches to overcome this are presented. The first approach is simply to use redundancy in the hardware. By having two units capable of controlling the drone behaviour, one can be in control while the other one performs the reboot and vice versa. Using redundancy was the original plan for this project, having an external Flight Controller that could maintain basic control of the drone while the x86-unit performed a reboot. It is uncertain if this would be possible on this specific drone however, since the FC was integrated with the compute board and seemed to reboot whenever the Intel Aero rebooted. Also, the positioning data was computed by the compute board itself and sent to the flight controller as input, leaving such data unavailable while rebooting the compute board. If this approach is taken, it could be wiser to use completely independent

components, giving more control of program execution and communication between the devices.

The second approach that could be taken is called *Dynamic Software Updating (DSU)*, which is a technique of updating software while the program is running. A running program can be thought of as a tuple $(s, P)$ where $s$ is the current program state and $P$ is the current program code. In DSU, a transformer function $F$ can be used to transform $s$ into the the corresponding state $s^*$ of the updated program code $P^*$, by computing $(F(s), P^*) = (s^*, P^*)$. This would remove the requirement of rebooting and reloading modules, but introduces other challenges such as validation of the new program state $s^*$. DSU is still a field of research, and current OSs and programs are usually not designed with DSU in mind. For a case like objective 2 however, Dynamic Software Updating could be an appropriate solution.

Chapter 5

# Implementation Process and Results

This chapter presents experiences from the first flight attempts, as well as the results received throughout the iterative implementation process of the solution proposal (Section 4.4). The final test of the complete solution proposal is also presented.

## 5.1 First Flight

The first flight was attempted a few days after receiving the drone, after the flashing and calibration steps described in section 4.1.2 were performed. The first thing noticed was that the drone was a bit tail-heavy, lifting the two foremost legs first when attempting to take off. The Flight Controller quickly corrected this after lifting the drone completely into the air, but control of the pitch axes was not even. The second thing noticed was that the drone was spinning in any one of the yaw directions, constantly increasing in speed. Attempts to compensate the unwanted yaw with the radio controller did not seem to have any effect. Multiple attempts and crashes were made, resulting in two broken propellers and breaking the GPS pole. After some investigation it was found that the FC was falsely calibrated, expecting the propellers to rotate in the opposite direction of what they were actually doing. By re-calibrating the FC with the propellers correct rotational directions, the drone stopped rotating uncontrollably and started to respond on input from the radio transmitter. After fixing this issue, flights with both the PX4 and Ardupilot firmware were performed. Both were successful.

## 5.2 Objective 1: Software Updates

A solution proposal to objective 1 was implemented and tested (Section 4.4. As the development process of this solution was agile, a continuous cycle of implementing, testing and evaluating testing results was performed. This section describes the iterative implementation process together with the results.

### 5.2.1 Drone Control over HTTP

The first sub-problem to be addressed was the communication aspect between the drone and the GCS. An HTTP server that interfaced the physical FC of the drone was implemented, along with an HTTP client running on a separate laptop connected to the drone's wifi AP. Combining dronekit and flask to create an HTTP interface between the physical FC and an HTTP client was successful, and it was possible to change the drone's behaviour by sending HTTP requests from a laptop to the drone. Arming the drone, taking off and changing the drone's altitude using this interface worked well.

The next step was to put the HTTP server into a container. This was performed by mapping one of the drone's ports to the container running the HTTP server, using built-in docker commands. This way, an HTTP client was able to access the HTTP server running inside a container. This implementation changed however, when issues related to container management and port mapping was encountered (Section 5.2.3). Instead, a proxy server forwarding all requests from the GCS to the IP address of the current functioning FC container was implemented.

### 5.2.2 File Transfer

Two approaches of file transfer were implemented and tested. The first approach ran this functionality in a separate container, hosting an FTP client. A separate laptop hosted an FTP server, making the drone able to fetch updates from a simulated cloud service in a dedicated container. The GCS would tell the drone to fetch an update from the simulated cloud service, and the drone would create the FTP client container. This approach worked well, but did not match the approach taken by FourC's platform.

The second approach was to move the file transfer functionality onto the OS layer, making the proxy server take care of updates instead of the dedicated FTP container. This approach was more similar to the way FourC's platform was operating, and also simplified container management. Thus, moving file transfer functionality to the OS layer was considered the better approach, and is the one implemented in the solution proposal.

### 5.2.3 Container Management

Two approaches of container management were implemented and tested. In the first approach, container management was handled by the FC container itself. This approach had the advantage of not having to inspect the incoming requests at two different points in the system. However, the disadvantages weighed more than the advantages. Creating a new FC container through the current instance was possible,

but removing the container from within itself was a problem. There was no way to inform the new FC container that the old instance was removed after it was actually removed. A workaround was implemented by making the new FC container remove the old instance instead, but this spread container management across several entities. This solution was a bit chaotic, and would not scale well. Additionally, this approach required the HTTP client to be notified whenever the FC container was renewed, since each container was assigned to one of the drone's ports when initialized. As HTTP is a synchronous protocol, additional client functionality had to be implemented in order to overcome this issue. Even though this implementation was tested and actually worked in practice, it was not considered a proper solution.

The second approach was to implement a proxy server at the OS layer that also handled the management of containers. When initialized, each container was assigned a unique IP address from a private address space. By using a container's IP address rather than a fixed port mapping, it was possible to exclude the client from the container management. Also, by pulling container management out of the FC container and placing it on the OS layer, the containers themselves became less complex. This approach also placed container management on the same layer as FourC's platform does, making the overall implementation function more like the implementation from FourC.

### 5.2.4 Test of Proposed Solution

A complete test of the solution proposal presented in section 4.4 was first performed using the dronekit_sitl simulator. This test was run on a single development computer, which simulated the complete drone, the GCS and the Cloud Service. First, a request to arm the drone simulator and rise to 5 meters was executed. The simulator would rise to 5 meters in about 4-6 seconds, printing its current altitude every second while rising. The time variance experienced is probably due to variance in simulated factors such as wind and pressure. After the altitude was reached, updated functionality was requested. As the drone had not learned to handle such requests yet, it responded with a HTTP 404 response. Now, the update was transferred. The drone simulator printed an update statement to the console, informing that it had received new software and was about to deploy it. The proxy server started a new container containing the new software, continuously printing its progress. The new FC container took about 10-15 seconds to initialize. The old FC container got removed, and the drone had changed the current functioning FC container. The updated functionality was once again requested, and this time the drone responded with an HTTP 200 response.

The FC simulator was only able to connect to and receive MAVLink messages from a single entity. To make sure that multiple containers could connect to and send

MAVLink messages to the physical FC, a test with three container instances was performed. The test was performed inside without propellers, and the FC container instances are referred to as con1, con2 and con3. All three instances connected to the physical FC without any issues, and the drone was armed through con1. con2 attempted to change the flight mode, and con3 fetched arming state and the current flight mode. The state and flight mode presented in con3 reflected the changes made through con1 and con2. The drone was then disarmed through con3, and it was demonstrated that the drone could be controlled through multiple containers simultaneously.

After a successful simulation had been performed, as well as making sure multiple containers could connect to the physical FC simultaneously, the drone was taken to the testing area described in section 4.3. The area was free of people, the wind was low and there was no precipitation at the time of testing. As the simulation had run on a single development computer, IP addresses was changed so they matched the setup with Intel Aero RTF and the separate laptop (Figure 4.4). The steps performed in the simulated environment was performed in the exact same manner, and the results were identical. The drone would rise to 5 meters when requested, respond with a HTTP 404 to the updated function request, receive the update, hold position while initializing a new FC container with the updated software and respond with an HTTP 200 to the updated function request. After a successful first test, a second test learning the drone to land was performed. Running the exact same procedure, the drone learned to land while being midair. Appendix G describes how to setup and run the tests described.

The time taken from the drone received an update until the drone had deployed it was about 14 seconds in the simulator and about 8 seconds in the physical scenario. In the simulated scenario, the new FC container had to create a new FC simulator before proceeding. This took about 6 seconds, which is approximately the time difference between FC container initialization in the simulated and physical scenario. Another difference was the processing of HTTP requests. Independent of the testing scenario, the time taken from a request was sent until a change was recorded on the drone was dependant on the type of change that was requested. For instance, a request to arm the drone took longer than a request to change the drone's altitude while already being airborne. The reason for this was probably because the drone performs a set of pre-flight checks before arming, such as checking the GPS signal, battery voltage etc. This is a safety measure, and they are not performed again after the drone has taken off. Additionally, the drone took approximately 1 second longer to respond to HTTP requests in the physical scenario. This could be caused by the increased distance between the drone and the GCS, possibly reducing the performance of the wireless link and increasing the number of TCP retransmissions. Another source could be delays in the internal communication on the drone itself.

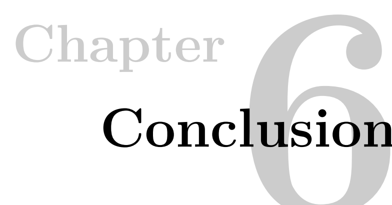
| Event | Simulation scenario | Physical scenario |
|---|---|---|
| Process arming request | 1.5 second | 2.5 second |
| Process altitude change while being airborne | 1 second | 2 seconds |
| Rise to 5m from the ground | 4-6 seconds | 4-6 seconds |
| Transfer update | ~0.1 seconds | ~0.3 seconds |
| Create new container | 13 seconds | 7 seconds |
| Remove old container | 1 second | 1 second |

**Table 5.1:** Time delays experienced in solution proposal.

The first reason seems more plausible, as serial communication is considered to provide stronger performance and be more reliable than wireless communication. Besides the minor differences in time, the physical and the simulated scenario gave the same results. Table 5.1 presents delays experienced in both the simulated and the physical testing scenario.

## 5.3 Objective 2: Operating System Updates

Objective 2 related to OS updates were omitted due to time constraints. No solution was implemented or tested for this objective, hence there are no results related to this objective.



# Chapter 6 Conclusion

## 6.1 Summary

This thesis has demonstrated how containers can be used to maintain service availability while software updates are being deployed. This functionality has been demonstrated by updating a drone while being midair. Attempting to update the underlying OS while being midair was not attempted due to time constraints, but two approaches directed towards this issue has been presented.

## 6.2 Solution proposal

The solution proposal described in section 4.4 realizes software updates of midair drones. By running FC software within a dedicated FC container, the task of downloading and executing updates ca be performed without interfering with flight control. Executing updated software in a new dedicated FC container enables continuous availability of service, by providing service in the old container until the new container is ready to take over as the functioning instance. This thesis has proven that a container solution can be used to maintain the availability of a service while the host receives updates and maintenance.

## 6.3 Possible Improvements

The proposed solution was tested by updating two python files. One of them was code for communicating with the physical FC controller and the second one was code for the HTTP server. Updating only two files were sufficient to test the solution proposal, but future versions could increase the size of updates. For instance, being able to update the complete FC container image could be useful. Also, the proposal does not perform any validation of the updated software before it is executed. An improvement would be to make sure the updated software will actually work before being executed or given any functional responsibility. One approach that can provide

some level of safety is to give identical input to both containers and compare their outputs. In a drone scenario, input can be basic safety commands such as attempting to land or changing the flight mode to position hold. If the output would be identical, both containers would handle the input the same way and some level of updating safety would be achieved.

## 6.4 Future Research

As there were not enough time to realize objective 2, approaches to solving such an issue should be researched. This thesis presents two approaches to realize this objective, but far more research can be directed towards different approaches, as well as towards how useful DSU would be in such a scenario. As mentioned in section 6.3, solutions to challenges related to validation of new software should be pursued. Testing prior to deployment is possible for application software, but testing OS software without actually deploying it can be a challenge.

Authentication of the entity that issues the update is also an interesting field of research. Even though there exist a vast amount of authentication protocols, validating their security and improving their efficiency should be continuously addressed. Research towards new authentication methods can also be conducted.

# References


[and] FC container docker image. https://hub.docker.com/r/andersvl/aero_server. Accessed: May 27. 2019.

[ard] Ardupilot. http://ardupilot.org/. Accessed: November 7. 2018.

[BA17] Hans Brenna and Ole-Johannes Aasbø. Drone med ota os update. Bachelor's thesis, Norwegian University of Science and Technology, May 2017.

[Ben] Benawke (Beijing) Co., Ltd. *Product Manual of TFmini.* Available online: https://cdn-shop.adafruit.com/product-files/3978/3978_manual_SJ-PM-TFmini-T-01_A03ProductManual_EN.pdf, Accessed: March 25, 2019.

[doc] Docker. https://www.docker.com/. Accessed: November 7. 2018.

[droa] Drone movement axes. Accessed: June 13. 2019.

[drob] Dronecode project. https://www.dronecode.org/. Accessed: November 7. 2018.

[droc] Dronekit. http://dronekit.io/. Accessed: November 7. 2018.

[drod] Dronekit software in the loop. https://github.com/dronekit/dronekit-sitl. Accessed: May 25. 2019.

[fla] Flask documentation. http://flask.pocoo.org/. Accessed: May 25. 2019.

[Fou] Fourc. https://www.fourc.eu/. Accessed: November 7. 2018.

[gena] Gentoo linux. https://www.gentoo.org/. Accessed: November 7. 2018.

[genb] Gentoo on edison. http://gentoo.ed-solutions.de/dokuwiki/start:edison. Accessed: May 23. 2018.

[HK18] Magnus Storhaug Hammer and Sindre Hammerø Kroknes. Dronekontroll prosjekt. Bachelor's thesis, Norwegian University of Science and Technology, Jan 2018.

[inta] Intel aero github page. https://github.com/intel-aero/meta-intel-aero/wiki. Accessed: April. 3, 2019.

[intb] Intel aero ready-to-fly drone. https://software.intel.com/en-us/aero. Accessed: November 7. 2018.

[intc] Intel edison. https://www.arduino.cc/en/ArduinoCertified/IntelEdison#toc4. Accessed: May 31. 2019.

[Intd] Intel. *Intel® RealSense™ Camera R200*. Available online: https://software.intel.com/sites/default/files/managed/d7/a9/realsense-camera-r200-product-datasheet.pdf, Accessed: March 25, 2019.

[Inte] Intel Corporation. *Intel®Aero Compute Board*.

[JST16] A. Christopher Janjua, John I. Sivertsvik, and Mia Tangvik. Drone med ota in-the-air sw update. Bachelor's thesis, Norwegian University of Science and Technology, may 2016.

[Lat] Lattepanda. https://www.lattepanda.com/. Accessed: November 7. 2018.

[mav] Mavlink. https://mavlink.io/en/. Accessed: November 7. 2018.

[Mul] Multiwii. http://www.multiwii.com/. Accessed: November 7. 2018.

[opt] Optical flow. https://en.wikipedia.org/wiki/Optical_flow. Accessed: March 8. 2019.

[ov7] Omnivision ov7251. https://www.ovt.com/sensors/OV7251. Accessed: March 25. 2019.

[ov8] Omnivision ov8858. https://www.ovt.com/sensors/OV8858. Accessed: March 25. 2019.

[para] Parrot bebop 2. https://www.parrot.com/us/drones/parrot-bebop-2. Accessed: November 7. 2018.

[parb] Pattor ar.drone 2.0 elite edition.

[PX4] Px4. http://px4.io/. Accessed: May 25. 2019.

[pym] Pymavlink. https://github.com/ArduPilot/pymavlink. Accessed: April 2. 2019.

[pyt] Python sockets. https://docs.python.org/2.7/library/socket.html. Accesses: May 29. 2019.

[qgr] Qgroundcontrol. http://qgroundcontrol.com/. Accessed: April 2. 2019.

[req] Requests: Http for humans™. https://2.python-requests.org/en/master/. Accessed: May 25. 2019.

[rtf] Intel® aero platform for uavs installation files. Accessed: May 28. 2019.

[Sky] SkyRC Technology Co., Ltd. *Instruction Manual of imax B6*. Available online: http://www.ittgroup.ee/files/IMAX_B6_manual.pdf, Accessed: March 25, 2019.

[sna] Snapdragon qualcomm flight kit. https://www.intrinsyc.com/vertical-development-platforms/qualcomm-snapdragon-flight/. Accessed: November 7. 2018.

# Appendix A
# FlightControllerHandler

This document contains documentation regarding the FlightControllerHandler class. The complete code can be found in A.2

## A.1 Description

The FlightControllerHandler is used to communicate with the physical FC. The class is dependant on the dronekit framework, which is a python API that communicates with vehicles over MAVLink. The framework contains an interface for communication with Ardupilot FCs. The complete dronekit documentation can be found in [droc].

The FlightControllerHandler constructor takes a dronekit vehicle-object as input. Has a set of functions that returns current state information or changes the behaviour of the drone. Some of the functions in this class are not used in this project. Table A.1 explains the most important functions that was used.

| Function | Arguments | Return value |
|---|---|---|
| **get_vehicle** | None | Dronekit vehicle object |
| **get_state** | None | Dict containing state information |
| **get_altitude** | None | The drone's current altitude |
| **get_vehicle_mode** | None | The drone's current flight mode |
| **set_vehicle_mode** | String mode | String explaining if the flight mode changed |
| **arm_disarm_vehicle**<br>Arms or disarms motors | Bool value | String explaining arming status |
| **go_to_alt**<br>Drone moves to given altitude | Int alt | None |

**Table A.1:** Explanation of some of the most important functions

## A.2 Code

**NOTE:** This code changes during flight in order to demonstrate midair software updates. This is the updated version of the code. Comments in the code describes functions and fields that got updated.

```
1  from __future__ import print_function
2  from dronekit import VehicleMode, LocationGlobalRelative
3  import time
4  import math
5
6  # Defines flightmodes that can be used through the client. Will change
       when runnin update
7  ALLOWED_FC_MODES = [
8      'STABILIZE',
9      'GUIDED',
10     'LOITER',
11     'FLIP',
12     'RTL',
13 ]
14
15 # Used for some movement functions. They are not tested yet.
16 EARTH_RADIUS_EQUATOR = float(6378137.0)
17 EARTH_RADIUS_POLE = float(6356752.3)
18
19
20 class FlightControllerHandler:
21
22     state = {
23         'ready_update': 'N'
24     }
25
26     def __init__(self, vehicle):
27         print("Creating new FlightControllerHandler")
28         self.set_vehicle(vehicle)
29
30         print('\n {} \n'.format(self.vehicle.system_status.state))
31
32     def get_vehicle(self):
33         return self.vehicle
34
35     def set_vehicle(self, vehicle):
36         self.vehicle = vehicle
37
38     def get_std_info(self):
39         return {
40             'verion': str(self.vehicle.version),
41             'location.global_frame': str(self.vehicle.location.
                   global_frame),
42             'location.global_relative_frame': str(self.vehicle.location
                   .global_relative_frame),
```

```
43                'location.local_frame': str(self.vehicle.location.
                      local_frame),
44                'attitude': str(self.vehicle.attitude),
45                'self.vehicle.velocity': str(self.vehicle.velocity),
46                'gps_0': str(self.vehicle.gps_0),
47                'gimbal': str(self.vehicle.gimbal),
48                'battery': str(self.vehicle.battery),
49                'ekf_ok': str(self.vehicle.ekf_ok),
50                'last_heartbeat': str(self.vehicle.last_heartbeat),
51                'rangefinder': str(self.vehicle.rangefinder),
52                'rangefinder.distance': str(self.vehicle.rangefinder.
                      distance),
53                'rangefinder.voltage': str(self.vehicle.rangefinder.voltage
                      ),
54                'heading': str(self.vehicle.heading),
55                'system_status.state': str(self.vehicle.system_status.state
                      ),
56                'groundspeed': str(self.vehicle.groundspeed),  # settable
57                'airspeed': str(self.vehicle.airspeed),  # settable
58                'mode.name': str(self.vehicle.mode.name),  # settable
59                'is_armable': str(self.vehicle.is_armable),
60                'armed': str(self.vehicle.armed),  # settable
61            }
62
63        def get_system_status(self):
64            return self.vehicle.system_status.state
65
66        def get_state(self):
67            self.state['system_status'] = str(self.get_system_status())
68            self.state['alt'] = str(self.get_altitude())
69            self.state['lon'] = str(self.get_longitude())
70            self.state['lat'] = str(self.get_lattitude())
71            self.state['location'] = str(self.get_location())
72            return self.state
73
74        def get_location(self):
75            return self.vehicle.location.global_relative_frame
76
77        def get_longitude(self):
78            return self.vehicle.location.global_relative_frame.lon
79
80        def get_lattitude(self):
81            return self.vehicle.location.global_relative_frame.lat
82
83        def get_altitude(self):
84            return self.vehicle.location.global_relative_frame.alt
85
86        def get_vehicle_mode(self):
87            return self.vehicle.mode.name
88
89        def get_self_did_update(self):
90            return self.state['did_update']
```

```
91
92      def get_ready_update(self):
93          return self.state['ready_update']
94
95      def get_update_finished(self):
96          return self.state['update_finished']
97
98      def set_ready_update(self, val):
99          self.state['ready_update'] = val
100
101     def set_update_finished(self, val):
102         self.state['update_finished'] = val
103
104     def set_self_did_update(self, val):
105         self.state['did_update'] = val
106
107
108     def set_vehicle_mode(self, mode):
109         if mode not in ALLOWED_FC_MODES:
110             return 'Vehicle mode {} is not supported'.format(mode)
111
112         print("Mode to be set: " + mode)
113         if self.vehicle.mode.name == mode:
114             print("Vehicle mode is already set to {0}".format(mode))
115             return "Vehicle mode is already set to {0}".format(mode)
116         print("\n Changing vehicle mode from {0} to {1}".format(self.
                vehicle.mode.name, mode))
117         self.vehicle.mode = VehicleMode(str(mode))
118         timeout = 0
119         while self.vehicle.mode.name != mode and timeout <= 5:
120             print(" Waiting for changes to take effect")
121             timeout += 1
122             time.sleep(1)
123         if timeout >= 5:
124             print("Exeeded timeout limit. Vehicle mode is {0}".format(
                    self.vehicle.mode.name))
125             return "Exeeded timeout limit. Vehicle mode is {0}".format(
                    self.vehicle.mode.name)
126         print(" Success! Vehicle mode changed to {0}".format(mode))
127         return " Success! Vehicle mode changed to {0}".format(mode)
128
129     def arm_disarm_vehicle(self, value):
130         if not self.vehicle.armed and value:
131             if not self.vehicle.is_armable:
132                 print("Waiting for vehicle to become armable")
133             print("Arming motors..")
134             self.vehicle.armed = True
135             while not self.vehicle.armed:
136                 print('Waiting for arming to take effect...')
137                 time.sleep(1)
138             self.set_ready_update('Y')
139             return 'Vehicle is armed'
```

```
140
141             elif value:
142                 print('Vehicle is already armed!')
143                 return 'Vehicle is already armed!'
144             else:
145                 print('Disarming motors...')
146                 self.vehicle.armed = False
147                 while self.vehicle.armed:
148                     print('Waiting for disarming to take effect...')
149                     time.sleep(1)
150                 self.set_ready_update('N')
151                 return 'Vehicle is disarmed'
152
153         def set_guided_mode(self):
154             if self.vehicle.mode.name != "GUIDED":  # CHANGE FOR PRODUCTION
155                 print("Changing vehicle mode to GUIDED")
156                 self.vehicle.mode = VehicleMode("GUIDED")
157                 while self.vehicle.mode.name != "GUIDED":
158                     print("Waiting for changes to take effect..")
159                     time.sleep(1)
160                 print("Vehicle mode is {0}".format(self.vehicle.mode.name))
161             else:
162                 print('Vehicle mode is GUIDED!')
163
164         def go_to_alt(self, alt, hold=False):
165
166             # Check if vehicle is armed
167             self.arm_disarm_vehicle(True)
168
169             # Check if vehicle mode is GUIDED. Has to GUIDED be in order to
                    change drone position and altitude.
170             self.set_vehicle_mode('GUIDED')
171
172             print("Going to altitude: {0}".format(alt))
173             if self.get_system_status() == 'STANDBY':
174                 self.vehicle.simple_takeoff(alt)
175             else:
176                 lat = self.get_lattitude()
177                 lon = self.get_longitude()
178                 pos = LocationGlobalRelative(lat, lon, alt)
179                 self.vehicle.simple_goto(pos)
180             if self.should_alt_increase(alt):
181                 self.do_increase_alt(alt)
182             else:
183                 self.do_decrease_alt(alt)
184             if hold:
185                 self.set_ready_update('Y')
186
187
188         def do_increase_alt(self, alt):
189             while True:
```

```
190                 print("Current alltitude: {0}".format(self.vehicle.location
                        .global_relative_frame.alt))
191                 if self.vehicle.location.global_relative_frame.alt >= alt *
                         0.95:
192                     print("Altitude reached!")
193                     break
194                 time.sleep(1)
195             return
196
197         def do_decrease_alt(self, alt):
198             while True:
199                 print("Current alltitude: {0}".format(self.vehicle.location
                        .global_relative_frame.alt))
200                 if self.vehicle.location.global_relative_frame.alt <= alt *
                         1.05:
201                     print("Altitude reached!")
202                     break
203                 time.sleep(1)
204             return
205
206         def should_alt_increase(self, alt):
207             return alt > self.get_altitude()
208
209     #
        ------------------------------------------------------------------------------------------

210
211     # These are functions contained in the updated FlightController.py
212
213         def get_updated_func(self):
214             string = 'This is an updated function!\n' \
215                      'Checking if status before update has remained:\n' \
216                      'Vehicle mode: {0}\n' \
217                      'System status: {1}\n' \
218                      'Altitude: {2}\n' \
219                      'Try to change something through the client!!'.format(
                             self.get_vehicle_mode(), self.get_system_status(),
                              self.get_altitude())
220             print(string)
221             return string
222
223
224     # These functions attempts to move the drone in each orientation. Has
            not been tested yet.
225
226         """
227         def move_east(self, distance):
228             if self.get_altitude() < 5:
229                 print('Aero should be at least 5m off the ground!')
230                 return False
231             dLat = distance/EARTH_RADIUS_EQUATOR
232
```

```
233
234             self.vehicle.simple_goto(self.get_lattitude()+dLat, self.
                    get_longitude(), self.get_altitude())
235
236
237         def move_west(self, distance):
238             if self.get_altitude() < 5:
239                 print('Aero should be at least 5m off the ground!')
240                 return False
241             dLat = distance/EARTH_RADIUS_EQUATOR
242             self.vehicle.simple_goto(self.get_lattitude()-dLat, self.
                    get_longitude(), self.get_altitude())
243
244
245         def move_north(self, distance):
246             if self.get_altitude() < 5:
247                 print('Aero should be at least 5m off the ground!')
248                 return False
249             dLon = distance/EARTH_RADIUS_POLE * 180/math.pi
250             self.vehicle.simple_goto(self.get_lattitude(), self.
                    get_longitude()+dLon, self.get_altitude())
251
252         def move_south(self, distance):
253             if self.get_altitude() < 5:
254                 print('Aero should be at least 5m off the ground!')
255                 return False
256             dLon = distance/EARTH_RADIUS_POLE
257             self.vehicle.simple_goto(self.get_lattitude(), self.
                    get_longitude()-dLon, self.get_altitude())
258         """
259
260     #
        ________________________________________________________________
```

# Appendix B
# Container HTTP Server

This document contains documentation regarding the FC container and the HTTP server running inside the FC container. The complete code for the HTTP server can be found in B.3.

## B.1 Docker Image

The container is based on Ubuntu 18.02, running python 2.7.15 with extra python modules Flask, dronekit and dronekit_sitl. The complete docker image can be found at [and].

```
docker pull andersvl/aero_server:v3
```

## B.2 Description

The HTTP server running within the FC container is written using Flask. Flask is a micro web framework written in Python that offer simple ways to handle HTTP requests. The complete Flask documentation can be found online[fla]. The HTTP server delegates communication with the physical Flight Controller (FC) to a FlightControllerHandler object (appendix A). To simplify development, the dronekit vehicle object given to the FlightControllerHandler can be simulated using dronekit_sitl[drod]. Dronekit_sitl is a Software in the Loop simulation of an Ardupilot FC.

## B.3 Code

```
1  from flask import Flask, request
2  from werkzeug.serving import WSGIRequestHandler
3  import json, threading
4  import dronekit, dronekit_sitl
5  import FlightControllerHandler
6
7
8  PORT = 5000
9  DEV_MODE = False
10
11
12 if DEV_MODE:
13         # Runs a simulated Ardupilot v3.3 Flight Controller
14         print("\n Running in DEV_MODE!!\n")
15         sitl = dronekit_sitl.start_default()
16         connection_string = sitl.connection_string()
17 else:
18         # String that connects to the docker host. The port is mapped
               to the Flight Controller
19         connection_string = "tcp:172.17.0.1:5760"
20
21 print('Attempting to connect to flight controller at: {}'.format(
       connection_string))
22 try:
23         Handler = FlightControllerHandler.FlightControllerHandler(
24                 dronekit.connect(connection_string, wait_ready=True)
25         )
26 except dronekit.APIException:
27         print("Not able to connect to vehicle due to being dronekitsitl
               .")
28
29
30 app = Flask(__name__)
31
32 # Threads that are used to handle some requests. Might be superflous
33 t1 = None
34 t2 = None
35
36
37 @app.route("/is_ready", methods=["GET"])
38 def is_ready():
39         return 'Y'
40
41
42 @app.route("/mode", methods=["POST"])
43 def set_vehicle_mode():
44         if request.method != "POST":
45                 print("vehicle mode change has failed due to request.
                       method not being POST")
46                 return
47         else:
```

```
48                  data = json.loads(request.data)
49                  mode = data["mode"]
50                  return Handler.set_vehicle_mode(mode)
51
52
53  @app.route("/arm", methods=["POST"])
54  def arm_disarm():
55          data = json.loads(request.data)
56          value = data["arm"]
57          return Handler.arm_disarm_vehicle(value)
58
59
60  @app.route("/go_to_alt", methods=["POST"])
61  def go_to_alt():
62          if request.method != "POST":
63                  print("Vehicle mode change has failed due to request.
                        method not being POST")
64                  return
65          data = json.loads(request.data)
66          alt = data["alt"]
67          response = "Going to altitude: {}".format(alt)
68          t2 = threading.Thread(target=Handler.go_to_alt, name="fc_thread
                ", args=(alt, True,))
69          t2.start()
70          return response
71
72
73  @app.route('/get_alt', methods=["GET"])
74  def get_alt():
75          alt = Handler.get_altitude()
76          return str(alt)
77
78
79  @app.route('/get_state', methods=['GET'])
80  def get_state():
81          state = Handler.get_state()
82          return json.dumps(state)
83
84
85
86  @app.route('/update', methods=['GET', 'POST'])
87  def update():
88          global t1
89          if request.method == 'GET':
90                  if not t1 or t1.isAlive():
91                          # Aero is still preparing, i.e going to
                                altitude 5 and holding.
92                          return 'N'
93                  state = Handler.get_state()
94                  return state['ready_update']
95
96          else: #Assumes request.method is POST
```

```
97                      print('AERO will receive an update! Going to alt 5m and
                            holding position...')
98                      t1 = threading.Thread(target=Handler.go_to_alt, args
                            =(5, True,))
99                      t1.start()
100                     return 'Aero is preparing to receive update!'
101
102   #
      ----------------------------------------------------------------------------------------
103
104   # These are functions contained in the updated container_server.py
105
106   @app.route('/new_function', methods=['GET'])
107   def get_new_func():
108           return Handler.get_updated_func()
109
110   #
      ----------------------------------------------------------------------------------------
111
112   if __name__ == '__main__':
113           WSGIRequestHandler.protocol_version = 'HTTP/1.1'
114           # Probsbly not needed
115
116           app.run(host='0.0.0.0', port=PORT, use_reloader=False)
```

# Appendix C
# Container Handler

This document contains documentation regarding the management of docker containers. The complete code can be found in C.2

## C.1 Description

The FC container has a set of dependencies in order to work properly. The image that the FC container uses includes dependencies such as the flask and dronekit library. Additionally, the container needs access to some of the host's file paths. The container host has to know the container's ip address, server state etc. This container management are handled by the container_handler module. Table C.1 explains some of its most important functions.

| Function | Arguments | Description |
|---|---|---|
| **create_server** | None | Creates an instance of an FC container. Returns its container ID |
| **start_server** | String *con_id* | Starts the container with id *con_id* |
| **start_flask_server** | String *con_id* | Starts the HTTP server within the FC container with id *con_id* |
| **server_is_ready** | String *con_id* | Checks whether the FC container with id *con_id* is ready to receive HTTP requests from the GCS |
| **get_container_ip** | String *con_id* | Gets the ip address of container with id *con_id* |
| **con_stopped** | String *con_id* | Checks whether the container with id *con_id* has stopped |
| **remove_con** | String *con_id* | Stops and removes the container with id *con_id* |

**Table C.1:** Explanation of some of the most important functions

## C.2 Code

```
from subprocess import call, Popen, PIPE
import requests

# Path to the aero repository
PATH_TO_AERO_DIR = '/home/root/aero'


CONTAINER_SERVER_PORT = '5000'


def remove_con(con_id):
 call(['docker', 'stop', con_id])
 call(['docker', 'rm', con_id])


def create_server():
 print('Creating server container...')
 p = Popen(['docker', 'create', '-it', '-v',
      '{}/fc_code/server:/aero_server'.format(PATH_TO_AERO_DIR),
      'andersvl/aero_server:v3'], stdout=PIPE)
 p.wait()
 return p.communicate()[0][:12]


def start_server_container(con_id):
 call(['docker', 'start', con_id])
 return con_id


def attach_server_container(con_id):
 start_server_container(con_id)
 call(['docker', 'attach', con_id])
 return con_id


def start_flask_server(con_id):
 call(['docker', 'exec', '-d', con_id, 'python', '/aero_server/
     container_server.py'])


def server_is_ready(server_ip):
 try:
  print('Trying to connect at: http://{0}:{1}/is_ready'.format(
      server_ip, CONTAINER_SERVER_PORT))
  r = requests.get('http://{0}:{1}/is_ready'.format(server_ip,
      CONTAINER_SERVER_PORT))
  print('r.text in server_is_ready(): '.format(r.text))
  if r.text == 'Y':
   return True
  else:
   print("no connection error, but not ready?")
```

```
49    except requests.exceptions.ConnectionError:
50     return False
51
52
53   def start_server_complete():
54    id = create_server()
55    start_server_container(id)
56    start_flask_server(id)
57    return id
58
59
60   def get_current_servers():
61    servers = []
62    cons = get_con_ids()
63    for i in range(len(cons)):
64     image = get_con_image(cons[i])
65     if image.startswith('andersvl/aero_server:'):
66      servers.append(cons[i])
67    return servers
68
69
70   def get_con_image(con_id):
71    con = Popen(['docker', 'ps', '-af', 'id={}'.format(con_id)], stdout=
         PIPE).communicate()[0].split('\n')[1]
72    print(con)
73    con_image = con.split()[1]
74    return con_image
75
76   def get_forwarding_port(con_id):
77    p = Popen(['docker', 'port', con_id], stdout=PIPE)
78    p.wait()
79    return p.communicate()[0].split(':')[1][:4]
80
81   # Clear all containers
82   def clear_cons():
83    print('Removing all containers...\n')
84    cons = get_con_ids()
85    for con in cons:
86     stop_container(con)
87     call(['docker', 'rm', con])
88    print('\nDone!')
89
90
91   def stop_container(con_id):
92    return call(['docker', 'stop', con_id])
93
94
95   def get_host_port_(con_id):
96    return Popen(['docker', 'port', con_id], stdout=PIPE).communicate()
          [0][:4]
97
98
```

```
 99 def get_container_ip(con_id):
100  p = Popen(['docker', 'exec', con_id, 'hostname', '-i'], stdout=PIPE)
101  p.wait()
102  return p.communicate()[0].split('\n')[0]
103
104
105 def get_con_ids():
106  cons = Popen(['docker', 'ps', '-aq'], stdout=PIPE).communicate()[0].
       split('\n')
107  return cons[slice(len(cons) - 1)]
108
109
110 def get_self_con_id():
111  return Popen(['cat', '/etc/hostname'], stdout=PIPE).communicate()
       [0][:12]
112
113
114 def get_stopped_containers():
115  return Popen(['docker', 'ps', '-aq', '--filter', 'status=exited'],
       stdout=PIPE).communicate()[0].split('\n')
116
117
118 def con_stopped(con_id):
119  return con_id not in get_stopped_containers()
```

# Appendix D
# Proxy Server

This document contains documentation regarding the proxy server running on the OS layer. The complete code can be found in D.2.

## D.1 Documentation

### D.1.1 Forward Class

The Forward class opens up a new socket between the proxy server and the target FC container.

### D.1.2 ProxyServer Class

The ProxyServer class hosts the actual proxy server. When a client connects to the proxy server, a Forward object (ref. D.1.1 is initialized, and the proxy saves a socket mapping between the client socket and forward socket. There can be multiple mappings. In this way, the proxy can forward data between clients and the FC container. Table D.1 shows this class' functions.

| Function | Arguments | Description |
|---|---|---|
| **main__loop** | None | Server starts listening for incoming connections |
| **on__accept** | None | Creates a new Forward instance and adds a mapping entry between the client and the FC container |
| **on__close** | None | Closes the connection between a client and the FC container. Socket mapping is removed |
| **on__recv** | None | Investigates request before it is eventually forwarded. Updates are recognized and handled outside the FC container |

**Table D.1:** Explanation of the ProxyServer class functions

### D.1.3 Container Host Functions

Table D.2 shows functions that are used when the container host receives an update. These functions do not belong to either of the classes described in D.1.1 and D.1.2

| Function | Arguments | Description |
|---|---|---|
| **aero__inform__update** | None | Informs the current FC container instance that an update will be executed |
| **aero__is__ready__update** | None | Checks whether the drone is ready to execute update, i.e hovering at fixed position |
| **start__new__fc__container** | None | Creates a new instance of the FC container and starts its HTTP server. Returns the container's id and ip address |
| **remove__old__fc__server** | String *con__id* | Used to remove the old FC container. Removes container with id *con__id* |
| **run__update** | None | Receives updates from the cloud service. Returns *True* if files were successfully received, *False* if not |

**Table D.2:** Explanation of the container host's functions, used for updates and container management

## D.2 Code

```
1  import socket
2  import select
3  import time
4  import sys
5  import requests
6  import container_handler as con
7
8
9  buffer_size = 1024
10 delay = 0.0001
11
12 HOST_ADDRESS = '192.168.8.1'
13 PORT = 8080
14
15
16 # Path to the directory that contains the files to update
17 PATH = '/home/root/aero/fc_code/server/'
18
19
20 # Values of the current docker instance that communicates with the
       physical FLight Controller
21 FC_CON_ADDRESS = Non
22 FC_CON_URL = None
23 FC_CON_ID = None
24
25 # IP address of the cloud service. In this case a seperate laptop is
       used.
26 CLOUD_ADDRESS = '192.168.8.43'
27
28 # Used for proxy functionality
29 forward_to = None
30
31 class Forward:
32     def __init__(self):
33         self.forward = socket.socket(socket.AF_INET, socket.SOCK_STREAM
               )
34
35     def start(self, host, port):
36         try:
37             self.forward.connect((host, port))
38             return self.forward
39         except Exception, e:
40             print e
41             return False
42
43 class ProxyServer:
44     input_list = []
45     channel = {}
46
47     def __init__(self, host, port):
48         self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
49            self.server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
                  1)
50            self.server.bind((host, port))
51            self.server.listen(200)
52
53        def main_loop(self):
54            self.input_list.append(self.server)
55            while 1:
56                time.sleep(delay)
57                ss = select.select
58                inputready, outputready, exceptready = ss(self.input_list,
                      [], [])
59                for self.s in inputready:
60                    if self.s == self.server:
61                        self.on_accept()
62                        break
63                    self.data = self.s.recv(buffer_size)
64                    if len(self.data) == 0:
65                        self.on_close()
66                        break
67                    else:
68                        self.on_recv()
69
70        def on_accept(self):
71            forward = Forward().start(forward_to[0], forward_to[1])
72            clientsock, clientaddr = self.server.accept()
73            if forward:
74                print clientaddr, "has connected"
75                self.input_list.append(clientsock)
76                self.input_list.append(forward)
77                self.channel[clientsock] = forward
78                self.channel[forward] = clientsock
79            else:
80                print "Can't establish connection with remote server.",
81                print "Closing connection with client side", clientaddr
82                clientsock.close()
83
84        def on_close(self):
85            print self.s.getpeername(), "has disconnected"
86            #remove objects from input_list
87            self.input_list.remove(self.s)
88            self.input_list.remove(self.channel[self.s])
89            out = self.channel[self.s]
90            # close the connection with client
91            self.channel[out].close()  # equivalent to do self.s.close()
92            # close the connection with remote server
93            self.channel[self.s].close()
94            # delete both objects from channel dict
95            del self.channel[out]
96            del self.channel[self.s]
97
98        def on_recv(self):
```

```
99            data = self.data
100           ip, port = self.s.getpeername()
101           # here we can parse and/or modify the data before send forward
102           if data.startswith('UPDATE') and ip == CLOUD_ADDRESS:
103               print('Received requsest to update! Informing Flight
                      Controller...')
104               global FC_CON_ID, FC_CON_ADDRESS, FC_CON_URL, forward_to
105               aero_inform_update()
106               if self.run_update():
107                   server_id, server_addr = start_new_fc_server()
108                   remove_old_fc_server(FC_CON_ID)
109                   FC_CON_ID = server_id
110                   FC_CON_ADDRESS = server_addr
111                   FC_CON_URL = 'http://{}:5000/'.format(FC_CON_ADDRESS)
112                   forward_to = (FC_CON_ADDRESS, 5000)
113           else:
114               self.channel[self.s].send(data)
115
116
117       # Receives files from the cloud service
118       def run_update(self):
119           try:
120               while True:
121                   size = self.s.recv(16) # Note that filename lengths are
                          limited to 255 bytes.
122                   if not size:
123                       break
124                   size = int(size, 2)
125                   filename = self.s.recv(size)
126                   filesize = self.s.recv(32)
127                   filesize = int(filesize, 2)
128                   file_to_write = open(filename, 'wb')
129                   chunksize = 4096
130                   while filesize > 0:
131                       if filesize < chunksize:
132                           chunksize = filesize
133                       data = self.s.recv(chunksize)
134                       file_to_write.write(data)
135                       filesize -= len(data)
136
137                   file_to_write.close()
138                   print 'File received successfully'
139               return True
140           except IOError:
141               print("IOError occured! failetranser failed")
142               return False
143
144
145   # Checks if Aero is ready to execute update
146   def aero_is_ready_update():
147       return requests.get(FC_CON_URL + 'update').text == 'Y'
148
```

```
149
150 # Informs the current FC container that an update is going to be
        executed
151 def aero_inform_update():
152     r = requests.post(FC_CON_URL + 'update')
153     while not aero_is_ready_update():
154         print('Waiting for Aero to get ready for update...')
155         time.sleep(2)
156     print('Aero is ready to receive update!')
157     return r.text
158
159 # Starts a new flight controller server in a new container
160 def start_new_fc_server():
161     print('Creating new Flight Controller container...')
162     new_server_id = con.start_server_complete()
163     new_server_address = con.get_container_ip(new_server_id)
164     while not con.server_is_ready(new_server_address):
165         print('Waiting for new server container to become ready...')
166         time.sleep(2)
167     print('New Flight Controller container whit ID {} is ready!'.format
            (new_server_id))
168     return new_server_id, new_server_address
169
170 # Removes the old flight controller
171 def remove_old_fc_server(con_id):
172     print("Removing old Flight Controller container with ID {}...".
            format(con_id))
173     con.remove_con(con_id)
174     print("Old Flight Controller container removed!")
175
176
177
178 if __name__ == '__main__':
179     # On startup, a fresh Flight Controller container is created and
            initialized
180     FC_CON_ID = con.start_server_complete()
181     FC_CON_ADDRESS = con.get_container_ip(FC_CON_ID)
182     FC_CON_URL = 'http://{}:5000/'.format(FC_CON_ADDRESS)
183
184     forward_to = (FC_CON_ADDRESS, 5000)
185
186     while not con.server_is_ready(FC_CON_ADDRESS):
187         time.sleep(2)
188
189     server = ProxyServer(HOST_ADDRESS, 8080)
190     try:
191         server.main_loop()
192     except KeyboardInterrupt:
193         print "Ctrl C - Stopping server"
194 sys.exit(1)
```

# Appendix E
# Ground Station HTTP Client

This document contains documentation regarding the ground station HTTP client. The complete code can be found in E.2

## E.1 Description

The ground station works as an HTTP client, sending HTTP requests to the HTTP server hosted by the FC container. The client side uses the python requests framework, an HTTP API written in python that provides simple methods to generate HTTP requests. The full documentation on requests can be found in [req]. Table E.1 describes some of the functions of the ground station HTTP client.

| Function | Arguments | Description |
|---|---|---|
| **request_arm_disarm** | Boolean *value* | Requests the FC container to arm or disarm the vehicle based on *value* |
| **request_set_vehicle_mode** | String *mode* | Requests the FC container to change the current flight mode to *mode* |
| **request_go_to_alt** | Int *alt* | Requests the FC container to take the drone to altitude *alt* |
| **request_get_state** | None | Requests the FC container to return current vehicle state information |
| **request_updated_function** | None | Requests the FC container to return data from an updated function. |

**Table E.1:** Explanation of the ground station HTTP client functions

## E.2 Code

```
1  import requests, json, time, sys
2
3
4  # IP address of the Aero drone
5  SERVER = '192.168.8.1'
6
7  # Port that maps to the proxy server on Aero
8  PORT = "8080"
9
10 # URL that is used to access the proxy server on Aero
11 URL = "http://{0}:{1}/".format(SERVER, PORT)
12
13 state = {}
14
15
16 # Change FlightMode
17 def request_set_vehicle_mode(mode):
18     data = {
19         "mode": mode,
20     }
21     r = requests.post(URL + "mode",
22                       data=json.dumps(data)
23                       )
24     print(r.text)
25
26
27 def request_get_current_alt():
28     r = requests.get(URL + "get_alt")
29     print('Current alt from server: {}'.format(r.text))
30     state['alt'] = float(r.text)
31     return state['alt']
32
33
34 def request_arm_disarm(value):
35     data = {
36         "arm": value
37     }
38     r = requests.post(URL + "arm",
39                       data=json.dumps(data)
40                       )
41     print(r.text)
42
43
44 def request_go_to_alt(alt):
45     request_get_state()
46     data = {
47         "alt": alt
48     }
49     r = requests.post(URL + "go_to_alt",
50                       data=json.dumps(data)
51                       )
```

```
52      print(r.text)
53      while state['alt'] < alt * 0.95:
54          state['alt'] = request_get_current_alt()
55          print("Current altitude: {}".format(state['alt']))
56          time.sleep(2)
57      return True
58
59
60  def request_update_inform():
61      r = requests.post(URL + 'update')
62      state['ready_update'] = 'N'
63      while state['ready_update'] != 'Y':
64          print('Waiting for Aero to get ready for update...')
65          state['ready_update'] = request_update_is_ready()
66          time.sleep(2)
67      print('Aero is ready to receive update!')
68      return r.text
69
70  def request_update_is_ready():
71      r = requests.get(URL + 'update')
72      return r.text
73
74  def request_do_update(*args):
75      data = {
76          'filenames': args
77      }
78      r = requests.post(URL + 'do_update',
79                        data=json.dumps(data)
80                        )
81      return r.text
82
83
84  def request_updated_function():
85      r = requests.get(URL + 'new_function')
86      try:
87          return json.loads(r.text)
88      except:
89          return r.text
90
91
92  def request_get_state():
93      global state
94      r = requests.get(URL + 'get_state')
95      state = json.loads(r.text)
96      return state
```

# Appendix F
# Simulated Cloud Service

This document contains documentation regarding the simulated cloud service. The complete code can be found in F.2

## F.1 Description

The simulated cloud service is nothing more than a short script that attempts to transfer files to the Drone.

## F.2 Code

```
1  import socket
2  import os
3
4  buff_size = 1024
5  CONN = socket.socket()
6  PORT = 8080
7
8  # Path to the local files that will be transferred.
9  PATH = '<PATH_TO_AERO_REPO>/ftp_server/files/'
10
11 # IP address of Aero drone
12 HOST = '192.168.8.1'
13
14 CONN.connect((HOST, PORT))
15
16 # Name of the files that will be transferred
17 FILENAMES = ['container_server.py', 'FlightControllerHandler.py']
18
19 CONN.send('UPDATE'.ljust(buff_size))
20
21 for filename in FILENAMES:
22     filename_size = bin(len(filename))[2:].zfill(16) # Encode filename
           size as 16 bit binary
23     CONN.send(filename_size)
24     CONN.send(filename)
```

```
25
26         filesize = bin(os.path.getsize(PATH + filename))[2:].zfill(32) #
               Encode filename size as 32 bit binary
27         CONN.send(filesize)
28         f = open('files/{}'.format(filename), 'rb')
29         CONN.sendall(f.read())
30
31
32         f.close()
33         print('Done sending file: {}'.format(filename))
34
35 CONN.close()
```

Appendix G

# Application Setup

This document describes how to setup and run the proposed solution presented in 4.4 on the Intel Aero Ready To Fly, step-by-step. Before attempting to run this application, make sure you are comfortable with using the transceiver to control the drone and changing flight modes, as well as QGroundControl [qgr] to monitor the drone state. If something goes wrong, you should be able to take control of the Intel Aero RTF using the transceiver. It is also suggested to perform all flying in an open, lucidly area that is clear of people.

When going through the steps, having a monitor and keyboard connected to the drone would be helpful, as the drone requires Internet access to perform some of the steps (2, 3, 4, 5). You can also ssh into the drone over usb, but this connection was experienced to be a bit unreliable.

## Step 1: Initial setup

First, make sure the drone is flashed with the latest versions of Operating System, BIOS, FPGA and FC. The OS .iso file can be downloaded from [rtf]. The .iso file contains the .rpm for the BIOS, .jam for the FPGA and the .px4 for the FC. Detailed steps of the flashing process can be found on the Intel Aero github wiki [inta].

## Step 2: Connect to the Internet

The Intel Aero RTF is by default using its wireless interface as a hotspot, creating its own wireless network. In order to connect to the Internet, this hotspot has to be taken down. Run the following commands to take down the hotspot, list available APs and connecting to the desired one, substituting <SSID> and <PASSWORD> with the AP's ssid and password:

```
1    $ nmcli con down hotspot
2    $ nmcli dev wifi
```

```
3    $ nmcli dev wifi connect <SSID> password <PASSWORD>
```

### Step 3: Make sure docker is working

The default Intel Aero RTF OS should contain docker by default. You can check this by running:

```
1    $ docker --version
2
3    Docker version 13.x.x, build xxxxxxx
```

The version should be 13 or higher. Then, run the following command to verify that docker works properly:

```
1    $ docker run hello-world
2
3    Unable to find image 'hello-world:latest' locally
4    latest: Pulling from library/hello-world
5    ca4f61b1923c: Pull complete
6    Digest: sha256:
         ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7
7    Status: Downloaded newer image for hello-world:latest
8
9    Hello from Docker!
10   This message shows that your installation appears to be working
         correctly.
11   ...
```

This commands pulls a test image from the remote dockerhub and runs it on your computer.

### Step 4: Pull the FC container image

Run the following command to download and extract the FC container image. This could take a few seconds:

```
1    $ docker pull andersvl/aero_server:v3
2
3    v3: Pulling from andersvl/aero_server
4    f476d66f5408: Already exists
5    8882c27f669e: Already exists
6    d9af21273955: Already exists
7    f5029279ec12: Already exists
8    3fa3a6069c9e: Already exists
9    c67ece1384e1: Already exists
10   3da8e716c67a: Already exists
11   d7fc24564a2b: Already exists
12   77be71115a27: Already exists
```

```
13      c2718f1e1e97: Already exists
14      da906a80bc4d: Pull complete
15      c08108ddad2a: Pull complete
16      7e6903301d2d: Pull complete
17      21aae9eaba70: Pull complete
18      Digest: sha256:7267
            c21f46e702b6fd63207fcb9538e9d9a84178fcbd8c8fa0ceb5a6f76ae793
19      Status: Downloaded newer image for andersvl/aero_server:v3
```

Now you should have the FC container image locally.

### Step 5: Pyhton and additional modules

Python 2.7, together with pip should already be installed on Intel Aero RTF. To make sure they are, run:

```
1   $ python --version
2   $ pip --version
```

Now, install the requests module [req]

```
1   $ pip install requests
```

Make sure that python 2.7 and requests are also installed on the separate computer that is used as the ground station.

### Step 6: Cloning the github repository

Git should be installed on the Intel Aero RTF by default. Run the following command to clone the aero repository containing code for the different components. This should also be ran on the laptop that is to be used as teh ground station.

```
1   $ cd
2   $ git clone https://github.com/andersvl/aero
```

Some variables might have to change in order for the code to work. These are local path variables and ip addresses, and are presented in step 8.The next steps require the Intel Aero RTF to host the default hotspot.

### Step 7: Enable hotspot and ssh into Intel Aero RTF

To enable the default hotspot, run

```
1   $ nmcli con up hotspot
```

Now, on a separate computer, connect to the Intel Aero RTF hotspot. The ssid should be "AERO-<MAC>" where <MAC> is the mac-address of the Intel Aero

RTF, and the password should be "1234567890". SSH into the Intel Aero RTF (Assuming running a linux terminal):

```
1    $ ssh root@192.168.8.1
```

192.168.8.1 is the ip address of Intel Aero RTF when hosting its own AP. No password should be required, but one can be set up after connected.

### Step 8: Changing file path and ip address variables

Some of the files contains file paths that has to change in order to work. These files should reflect the local file paths (i.e paths on Intel Aero RTF), and these files are:

**aero/fc_code/server/container_handler.py, line 8**
PATH_TO_AERO_DIR
Should hold the complete file path to the aero/ repository.

**aero/fc_code/server/proxy_server.py, line 18**
PATH
Should hold the complete file path to aero/fc_code/server/

**aero/fc_code/server/proxy_server.py, line 27**
CLOUD_ADDRESS
Should hold the ip address of the of the computer hosting the cloud service. In the project, this was the ip address of the laptop connected to the hotspot.

**aero/fc_code/server/container_server.py, line 9**
DEV_MODE
Should be True when wanting to simulate the FC (develop mode), False to use the physical FC

**aero/ftp_server/files/container_server.py, line 9**
DEV_MODE
Should be True when wanting to simulate the FC (develop mode), False to use the physical FC

**aero/ftp_server/filetransferTCPserver.py, line 10**
PATH
Should hold the complete file path to aero/ftp_server/files

**aero/ftp_server/filetransferTCPserver.py, line 13**
HOST

Should hold the ip address of Intel Aero RTF (which is 192.168.8.1 by default)

**aero/fc_code/client/simpleclient.py, line 5**
SERVER
Should hold the ip address of Intel Aero RTF (which is 192.168.8.1 by default)

### Step 9: Preparing the updating files

The files that will be transferred to Intel Aero RTF resides in the aero/ftp_server/files directory. The files that will be overwritten resides in the aero/fc_code/server directory, and the filenames are "FlightControllerHandler.py" and "container_server.py". Some of the functions and variables of these files should exist only in aero/ftp_server/files directory of the file transferring entity, and not in the aerp/fc_code/server directory of the Intel Aero RTF. These functions and variables are:

**FlightControllerHandler.py:**
Everything below line 203
line 6, ALLOWED_FLIGHT_MODES should only contain "STABILIZE", "GUIDED", and "LOITER". When updated, the list will contain flight modes "FLIP" and "RTL" as well.

**container_server.py:**
line 106, get_new_function()

When updating, the files will be transferred from the aero/ftp_server/files directory on the file transferring entity, into the aerp/fc_code/server directory of the Intel Aero RTF.

### Step 10: Running the whole thing

**It is suggested to first test the application running the FC simulator, making sure everything works as expected.**

After all the software is installed on the Intel Aero RTF, make sure its default hotspot is up and ssh into the drone. On the ground station, open QGroundControl and make sure the drone has GPS signal and that the flight mode is set to "Stabilize".

```
1    $ ssh root@192.168.8.1
```

Then, cd to the /server directory and start the proxy server.

```
1    root@aero$ cd aero/fc_code/server
```

```
2    root@aero$ python proxy_server.py
3    Creating server container...
4    004fd2995fef
5    Trying to connect at: http://172.17.0.2:5000/is_ready
6    Trying to connect at: http://172.17.0.2:5000/is_ready
7    Trying to connect at: http://172.17.0.2:5000/is_ready
8    Trying to connect at: http://172.17.0.2:5000/is_ready
9    Trying to connect at: http://172.17.0.2:5000/is_ready
10   Trying to connect at: http://172.17.0.2:5000/is_ready
11   Trying to connect at: http://172.17.0.2:5000/is_ready
12   Trying to connect at: http://172.17.0.2:5000/is_ready
13   Trying to connect at: http://172.17.0.2:5000/is_ready
14   Trying to connect at: http://172.17.0.2:5000/is_ready
15   r.text in server_is_ready():
```

When starting the proxy server, a new FC container is created and started. The FC container connects to the FC (simulated or physical). "r.text in server_is_ready():" means that the FC container is ready.

Now, attempt to take the drone to altitude 5 meters. On the laptop working as the ground station, start python and import the simpleclient.py module:

```
1    $ cd aero/fc_code/client
2    $ python
3    Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
4    [GCC 7.3.0] on linux2
5    Type "help", "copyright", "credits" or "license" for more
         information.
6    >>> import simpleclient as cli
7    >>> cli.request_updated_function()
8    u'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">\n<title
         >404 Not Found</title>\n<h1>Not Found</h1>\n<p>The requested
         URL was not found on the server. If you entered the URL
         manually please check your spelling and try again.</p>\n'
9    >>>
```

The snippet above attempts to get a function that does not yet exist. Now we will update the Intel Aero RTF to contain that function while being midair. The following code makes the drone take off to 3 meters and holding its position.

```
1    >>> cli.request_go_to_alt(3)
2    Going to altitude: 3
3    True
```

Now, in a new terminal window on the ground station, cd into the ftp_server directory:

```
1    $ cd aero/ftp_server
2    $ python filetransferTCPserver.py
```

```
3    Done sending file: container_server.py
4    Done sending file: FlightControllerHandler.py
```

If Intel Aero RTF received the files, it will rise to and altitude of 5 meters and apply the updates. This includes to start a new FC container with the new files, changing routing so that requests are received by the new container and closing the old FC container. If successful, the output of the proxy server should look something like this:

```
1    Received requsest to update! Informing Flight Controller...
2    Waiting for Aero to get ready for update...
3    Waiting for Aero to get ready for update...
4    Waiting for Aero to get ready for update...
5    Waiting for Aero to get ready for update...
6    Waiting for Aero to get ready for update...
7    Waiting for Aero to get ready for update...
8    Aero is ready to receive update!
9    File received successfully
10   File received successfully
11   Creating new Flight Controller container...
12   Creating server container...
13   fd7fa4f34617
14   Trying to connect at: http://172.17.0.3:5000/is_ready
15   Waiting for new server container to become ready...
16   Trying to connect at: http://172.17.0.3:5000/is_ready
17   Waiting for new server container to become ready...
18   Trying to connect at: http://172.17.0.3:5000/is_ready
19   Waiting for new server container to become ready...
20   Trying to connect at: http://172.17.0.3:5000/is_ready
21   Waiting for new server container to become ready...
22   Trying to connect at: http://172.17.0.3:5000/is_ready
23   Waiting for new server container to become ready...
24   Trying to connect at: http://172.17.0.3:5000/is_ready
25   Waiting for new server container to become ready...
26   Trying to connect at: http://172.17.0.3:5000/is_ready
27   Waiting for new server container to become ready...
28   Trying to connect at: http://172.17.0.3:5000/is_ready
29   Waiting for new server container to become ready...
30   Trying to connect at: http://172.17.0.3:5000/is_ready
31   Waiting for new server container to become ready...
32   Trying to connect at: http://172.17.0.3:5000/is_ready
33   r.text in server_is_ready():
34   New Flight Controller container whit ID fd7fa4f34617 is ready!
35   Removing old Flight Controller container with ID 4928ed22c8e1...
36   4928ed22c8e1
37   4928ed22c8e1
38   Old Flight Controller container removed!
```

We should now be able to request the new function from the ground station:

```
1    >>> print(cli.request_updated_function())
```

```
2      This is an updated function!
3      Checking if status before update has remained:
4      Vehicle mode: GUIDED
5      System status: ACTIVE
6      Altitude: 4.99
7      Try to change something through the client!!
8      >>>
```

We can also attempt to land the drone:

```
1      >>> cli.request_set_vehicle_mode("RTL")
2        Success! Vehicle mode changed to RTL
3      >>>
```

The drone should now go to a certain altitude, and then start slowly sinking to the ground. Hurray, we have now update the Intel Aero RTF while being midair!