



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Distributed
Computing*



Seamless Drone Replacement

Semester Thesis

Louis Kögel

lkoegel@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:

Manuel Eichelberger, Simon Tanner
Prof. Dr. Roger Wattenhofer

June 24, 2018

Acknowledgements

I would like to gratefully thank my supervisors Manuel Eichelberger and Simon Tanner for supporting me throughout my semester theses. They motivated and helped me over the course of my work and the numerous challenges I encountered. Beyond that I want to thank Laurin Göller who worked closely with me and wrote his own similar thesis. The work we did together was fun and highly productive and we shared a lot of insights that enhanced my work. We wrote cooperatively the second chapter. In addition to that I want to thank Beat Küng who advised me as an official PX4 developer.

Abstract

One of the biggest restriction of quadcopters is their limited flight time. This limitation can be a drawback in some application, for example in industrial inspection, security surveillance or search and rescue. The aim of this thesis is to address this restriction by presenting a way of seamlessly replacing these drones mid-air. Since the ready-to-fly drones of today are very fast outdated, we decide to build our own quadcopters in order to be independent regarding hardware. After building the drone we explore the implementation of the replacement system and algorithm. We use ROS to simplify software design and reduce the complexity of the communication. The drone's position and the battery status are the informations we use to implement the replacement. Because of driver problems we can not test our implementation with our self built drones, however we prove that our implementation works by simulating the replacement with Gazebo.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Motivation	1
1.2 Related Work	2
2 Drone Assembly	3
2.1 Hardware	3
2.1.1 Flight Controller: Raspberry Pi + Emlid Navio2	3
2.1.2 Electronic Speed Controller (ESC)	4
2.1.3 Motors and Propellers	4
2.1.4 Battery	4
2.1.5 Radio and Telemetry	5
2.1.6 Frame	5
2.1.7 Setup	5
2.2 Software	8
2.2.1 PX4	8
2.3 PID Tuning	10
3 Drone Replacement	13
3.1 Implementation	14
3.1.1 ROS	14
3.1.2 Mavros	15
3.1.3 MAVLink	15
3.1.4 Structure of a self coded <i>UAV Node</i>	15
3.1.5 Replacing procedure	16

CONTENTS	iv
3.2 Simulation	16
3.2.1 Gazebo	16
3.3 Application	18
3.3.1 GPS status " <i>NOT OK</i> "	18
3.3.2 Barometer driver	18
4 Conclusion	19
4.1 Future work	19
Bibliography	20
A Appendix Chapter	A-1
A.1 Raspberry Pi 3 Model B Installation	A-1
A.2 Instruction to launch ROS (<i>Master Node</i>), the <i>MAVROS Nodes</i> and the <i>UAV Nodes</i>	A-2
A.3 Instruction to launch Gazebo Simulation	A-2

Introduction

1.1 Motivation

In many of the applications quadcopters have today the flight time is one of the biggest restrictions. Nowadays these drones can usually fly up to 20 minutes. This is often still not enough for certain missions. The quadcopter has to come back to swap battery in order to continue its mission. Seamlessly replacing quadcopter mid-air would accordingly mean an extension of the flight time. In addition, there is always at least one drone in the air guaranteeing a continuous operation.

Therefore this thesis focuses on creating a system to seamlessly replace quadcopters in the air. A new fully charged drone should replace one already flying drone with almost empty battery by flying next to it and then continuing its mission, while the empty drone is flying back.

This continuity of quadcopters in the air implies a huge benefit and will make them more useful in already existing use cases. An example is the industrial inspection of bridges with a high resolution camera on a drone. Flying for a longer period of time will simplify the whole process of the inspection. It will save time for the inspector who does not have to manually fly back and forth anymore, in order to change the battery of the drone to continue his actual work.

Not only in existing applications the drone replacement will be of enormous use but especially this replacement ability will enhance the use of quadcopters to a much broader range of applications. Having a continuous view from above can be useful for example for firefighters who need better view of an incident. They can get continuously important insights they simply can not get from the ground. It can also be useful for police security surveillance applications. In the case of collecting evidence it is crucial to have no interruption of the video due to a battery swap. An other example are search and rescue applications that often last longer than 20 minutes. Here this replacement feature will drastically enhance the benefit of using drones.

Since buying a ready-to-fly quadcopter with some computational power is

expensive, we try to find and build an easy self constructed platform. This gives us the flexibility to add additional hardware. We hope that this platform will become the foundation of following research projects.

1.2 Related Work

We start by reviewing existing implementations of similar approaches. We find that most approaches focus on one unmanned aerial vehicle (From now on UAV). There are three approaches.

The first approach relates to the landing of the quadcopter and to change the battery on the ground autonomously as explored in this paper [1]. This is a useful feature regarding autonomous quadrocopter systems. It enables flying autonomously without an operator who changes the batteries. Nevertheless this system is not capable of working on continuous missions which is one of our goals. However this system can be augmented with seamlessly replacing drones which will enable having drones flying forever replacing themselves all the time.

The next attempts focus on charging the UAV mid-air. The second approach is researched at *Imperial College London* and tries to charge the drone through a magnetic field via induction [2]. This is not yet a viable solution because in order to recharge the drone has to fly very close to the wireless charging pad. They use a drone without battery to have a lighter weight to lift and to demonstrate the technique. Only short flight times are possible with this system, taking account the fact that those devices carry only a small battery. In fact, this system implicates that the drone has to fly to a charging station and is therefore not continuously available for a mission.

The last approach we review charges a UAV through a laser beam from the ground [3]. This will in theory enable a continuous flight assuming there is no obstacle between the laser and the UAV. However this is still only possible with UAVs using wings to be more efficient during their flight. For quadcopters an off the shelf laser will not provide enough power transmission. Furthermore a lot of additional equipment is necessary for the laser system which is a huge disadvantage. Nevertheless for unmanned aerial planes this can be a viable solution.

Drone Assembly

2.1 Hardware

For our application, the drone needs to meet the following conditions:

- GPS controlled flight for autonomous missions.
- A telemetry link to a Ground Computer to enable communication and automated replacement.
- Long flight time. Which implies the drone to be as light as possible.

These requirements lead us to use the following components.

2.1.1 Flight Controller: Raspberry Pi + Emlid Navio2

As explained below in Section 2.2.1 we used PX4 as software to control our quadcopter. The standard flight controller using PX4 is the Pixhawk board. Additional to that we need a little computing power which would mean to add a small companion computer like the Raspberry Pi. This would mean having two boards resulting in a heavier and larger setup. We found another solution, which is also supported by PX4. It unites these two different boards into a single one. It is the Emlid Navio2 shield [4] for the Raspberry Pi [5]. This shield provides all sensors for creating a powerful flight controller with a Raspberry Pi running PX4 software: dual IMU (inertial measurement unit) for orientation and motion sensing, a barometer for precise altitude control, GNSS receiver for GPS positioning and extension ports (UART) for a telemetry radio. Furthermore, this combination provides compatibility to Robot Operating System (ROS). This hardware is the lightest and most compact stack for our application. The Navio2 mounted on a Raspberry Pi is shown in Figure 2.1.

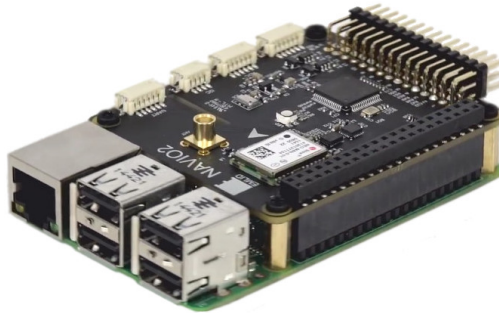


Figure 2.1: Picture of a Raspberry 3 Model B with the Emlid Navio2 shield [6]

2.1.2 Electronic Speed Controller (ESC)

The flight controller sends a pulse width modulated signal to the electronic speed controller (ESC) indicating how fast the motor should spin. The ESC controls and powers the motors of the drone to the desired rotation speed. We use the Hobbywing xRotor 40A 4in1 ESC [7]. These are up to date ESCs widely used in the drone industry, especially for applications where high performance and low weight is required. In contrast to most ESCs this is a single board holding all four ESCs making assembly and cable management a lot easier.

2.1.3 Motors and Propellers

The choice of the motor and propeller is also based on common components from today's drone industry. As the estimate drone's weight is less than 1.4kg, the motors of our choice are T-Motors F40 1600KV [8]. According to their manual, these motors can spin a 7 inch propeller [9] with 5S lithium polymer batteries with a maximum lift of 1400g per motor resulting in a thrust to weight ratio of around 4:1. According to several online platforms, a ratio of 2:1 is the lower boundary for quadcopters to still allow for maneuverability and enough power to arrest a descent. Therefore, 4:1 is a rather powerful drone setup. However, this drone shall be a platform for further projects where the duty might be to carry a little weight.

2.1.4 Battery

The power for the drone is provided by a 5S lithium polymer (LiPo) battery with 3800mAh [10]. LiPo batteries are predominantly used in all RC hobbies based on their very high power density. The voltage of our LiPo battery (5S means there are 5 cells of 3.7V in series resulting in a total of 18.5V) matches the combination of motor and propeller choice according to the motor's manual.

2.1.5 Radio and Telemetry

As a radio link to manually control the copter we use the Frsky Taranis [11] which is the most well-known RC transmitter for drones world wide together with a Frsky XSR receiver [12]. The telemetry connection is provided by a standard 433 Mhz module [13]. It enables MAVLink communication through an UART port of the Navio2.

2.1.6 Frame

In order to have a quadcopter frame which is strong enough to serve as a mounting platform for all different electronics and still light enough to keep the flight time of our drone on a satisfactory level we decided to design a carbon frame ourselves.

2.1.7 Setup

To get a better impression on how the components are connected we provide in Figures 2.2 and 2.3 the basic setup of our drone. In Figure 2.2 At the left side we see the motors with the ESCs below (we used not four individual but a *4in1* ESC solution. Right next to the ESCs is the radio control module. Above the Raspberry Pi and the Navio2 is the telemetry kit. Below is the GPS antenna. To the right is the battery connected to a power module, measuring the power output to the ESC and the battery's voltage, while also providing power to the Raspberry Pi and the Navio2. In Figure ?? is a picture of the fully built drone.

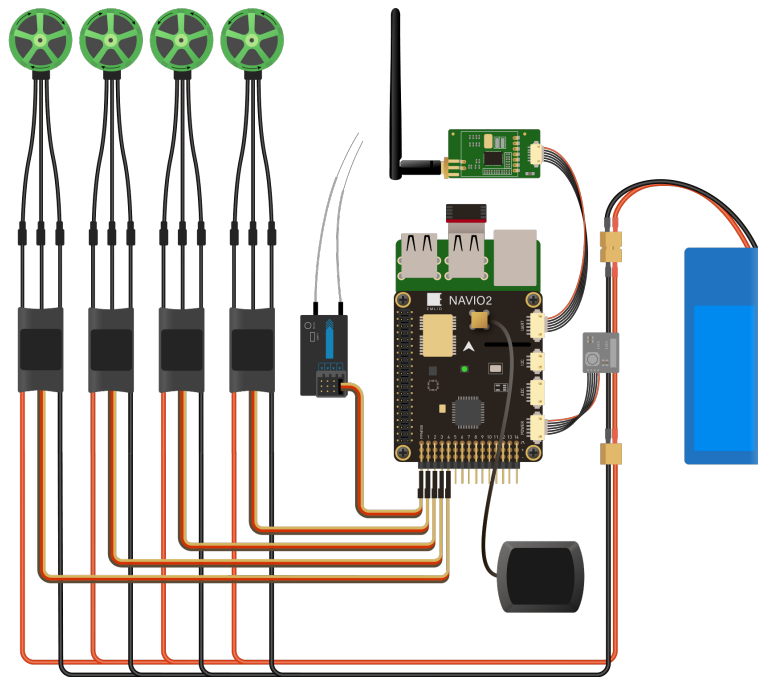


Figure 2.2: All components connected to the Raspberry Pi and Navio board. In this picture there are four ESC, whereas we used a *4in1* ESC solution. [14]

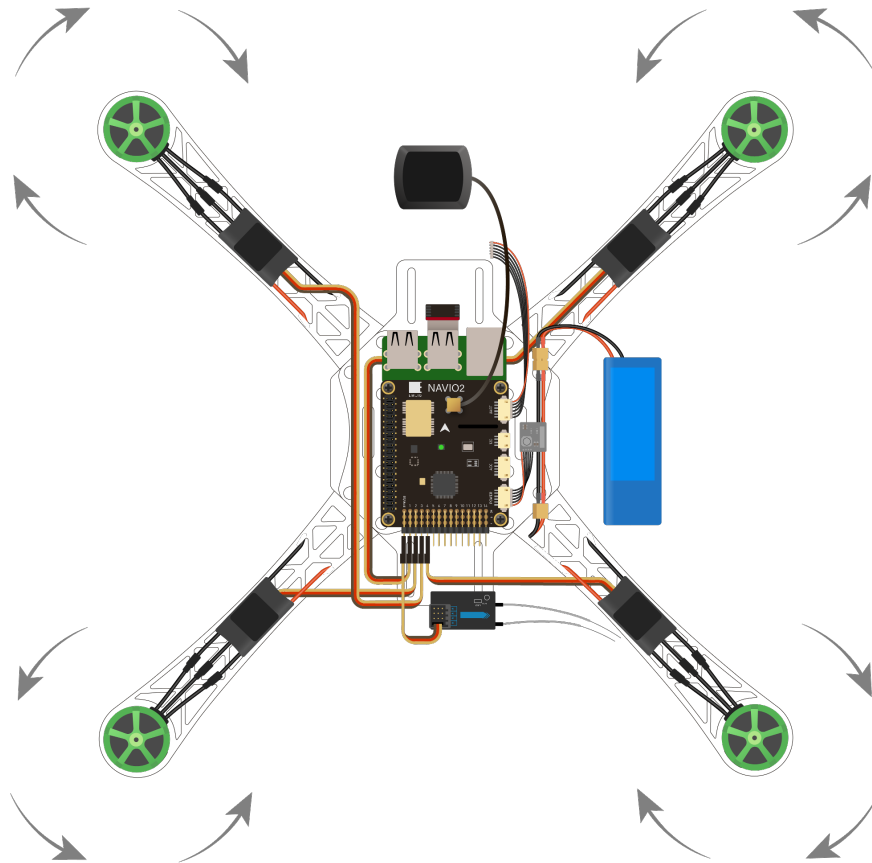


Figure 2.3: All components mount on a quadcopter frame. The arrows indicate the direction of rotation of the motors. In this picture the Telemetry module is left out [14]



Figure 2.4: Picture of the finished drone. In addition to the components described above there is a Pi camera installed which was only in use in the thesis of Laurin Göller

2.2 Software

So far we have elaborated the hardware components of our aircraft. Now we want to make these components work together in order to fly. There are open source autopilot software for plenty of applications and different UAV setups. During our semester thesis we primarily used PX4 as our autopilot software, because it brings along a large toolset for autonomous flight. In addition PX4 bring the ability to communicate with our laptop running Ground Control Station and later ROS Nodes.

2.2.1 PX4

PX4 is the autopilot we chose because the PX4 project was founded at ETH Zürich and still many ETH engineers are developers for the project. This was indispensable for complications we had because we could directly address our problems to the developers.

The high level software architecture of PX4 is shown in Figure 2.4. One of the core algorithms in the *Flight Controller* is the State Estimator which is an extended Kalman filter (EKF). It fuses the IMU sensor data with the GPS sensor data from the *Drivers* to estimate the orientation and a local and global position. PX4 then uses this position estimation to control the drone using PID controllers. In this overview we also see the PX4 features like *Connectivity* and *Storage* which we use to build and test our application. We use MAVLink as the communication protocol to communicate with our Ground Computer. From our Ground Computer we will later interpret data of our drones and send commands back to it. The Logging is of use while setting up the drone. For example we use the log data to tune our PID controllers.

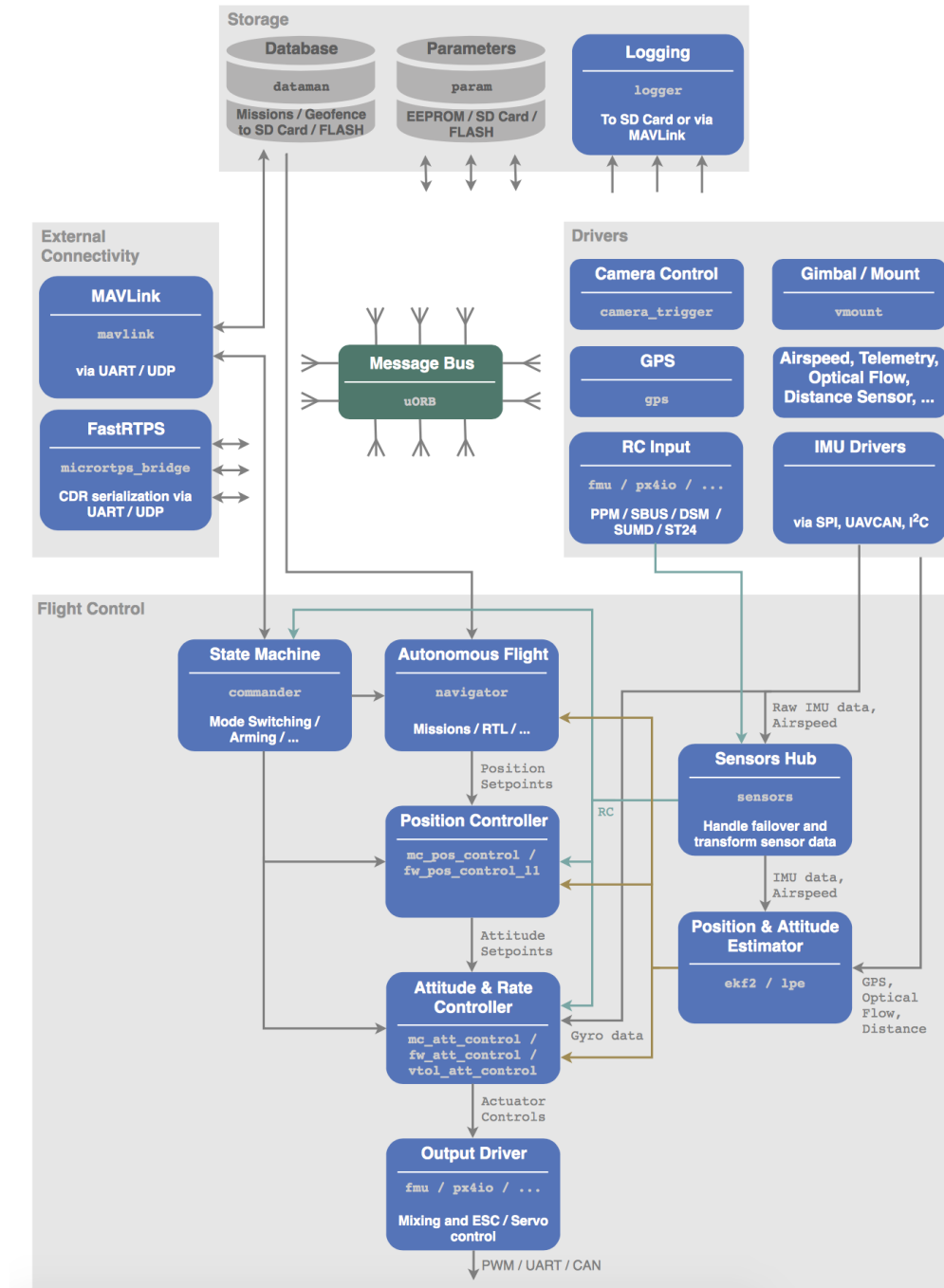


Figure 2.5: PX4 high level software architecture [15]. The *Drivers* collect data from all the different sensors. The *Flight Control* uses the data to maneuver the drone. Over the *External Connectivity* the drone can be connected to a Ground Computer. Data can be exchanged and commands can be sent to the drone. The *Storage* logs the flight.

2.3 PID Tuning

PX4 uses PID controllers, which are the most used feedback loop controller in quadcopter control systems. They continuously calculate an *error value* $e(t)$ as the difference between a desired *setpoint* $r(t)$ (given by the *RC*) and a measured *actual system state* $y(t)$ (which is an estimate from the sensor data through the EKF). Based on the error term it calculates a proportional, integral, and derivative correction term (denoted by P, I, and D respectively). Each term is multiplied by a corresponding coefficient to create a correction control output.

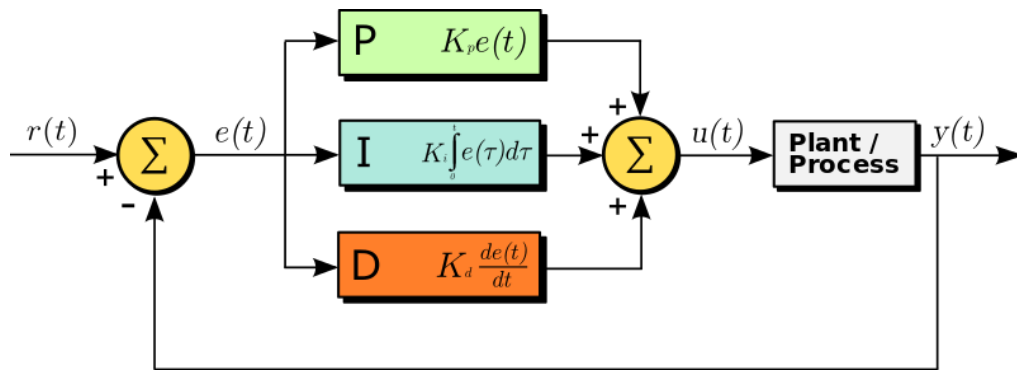


Figure 2.6: PID control loop [16].

PX4 has multiple layer of controller and depending on the flight mode of the drone, it controls more or less to hold the position [17]. In our first flight we use only the inner-most controllers (the *rate controller* by using the "*Manual Flight Mode*"). The controllers are three independent PID controllers responsible for controlling only the change of angle in roll, pitch and yaw orientation. The orientations are indicated in Figure 2.6

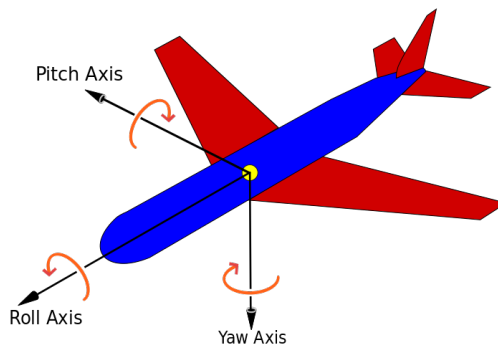


Figure 2.7: Aircraft principal axes [18].

Those controllers are not tuned appropriately as our first flight quickly proved. The drone oscillated and was nearly uncontrollable. To tune the PID controllers we use the steps of the PX4 *PID Tuning Guide* [17].

In the Figures 2.7 and 2.8 the effect of PID tuning is exemplified. As an example we took the *roll-rate* controller (also possible are the pitch- and yaw-rate). Figure 2.7 shows a log using the default parameters whereas Figure 2.8 is a log using tuned parameters. The green line is the *setpoint*, the control input. The red line however is the actual *roll-rate* estimated by the EKF. It should be as close as possible to the green line. As we can see in Figure 2.7, the red line oscillates around the green one with much higher amplitude than the actual control input. This explains the strong oscillation of the whole quadcopter. In contrast to that we see in Figure 2.8 (with tuned parameter) that the red line closely follows the green one. This results in a smooth flight and an easy to steer drone.

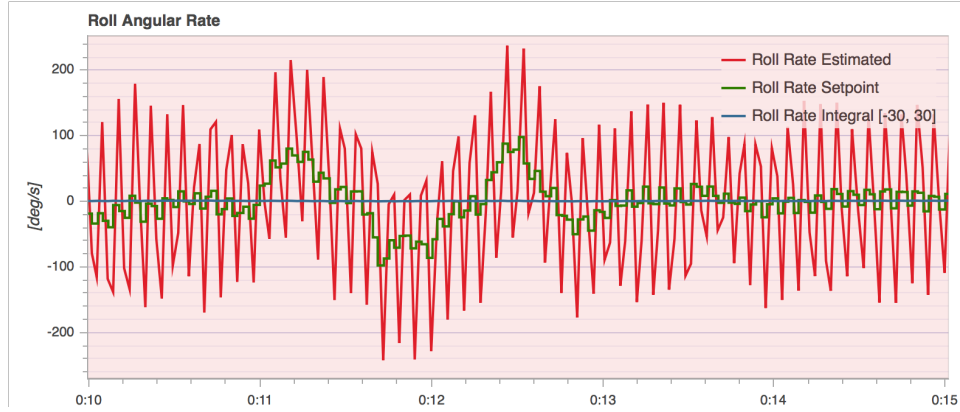


Figure 2.8: The *Roll Angular Rate* comparison from a test flight. This rate is controlled by a PID controller. The green line is the supposed rate and the red line is the actual rate of the quadcopter. Clearly we can infer that the PID controller is not tuned properly. The actual rate oscillates with a much higher amplitude which results in an unstable flight.

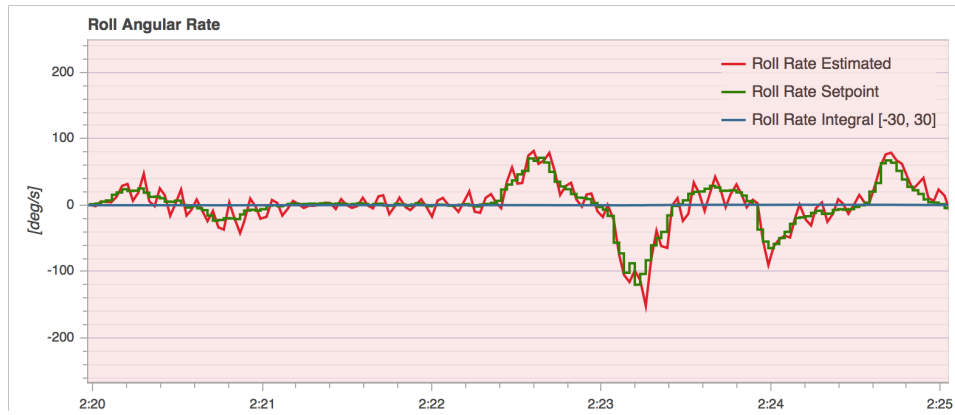


Figure 2.9: The *Roll Angular Rate* comparison as in Figure 2.7 but with tuned PID parameter. The green line is the supposed rate and the red line is the actual rate of the quadcopter. With tuned PID parameter we achieve to control the quadcopter much better to the rate it is supposed to have. This results in a smooth flight.

Drone Replacement

PX4 has a lot of useful features. First of all, with PX4 the drone can stabilize itself and hold a position without any RC (Radio Control) input of a pilot. On top of that, it can autonomously start, fly to predefined waypoints, and land. But whenever the battery of a drone is empty it has to stop its mission and return to the landing spot to swap the battery. Over all the features implemented in PX4 are numerous but only for single drones. This is what we want to augment. We need an infrastructure that enables communication between at least two drones. We use a laptop as Ground Computer running ROS which manages our communication and the control of our drones. The setup communicates state information like position and battery status using MAVLink messages. This structure is shown in Figure 3.1.

We will show in detail our implementation - which tools we use and commands we need to execute the replacement. In order to demonstrate and proof that our implementation is working we will show a simulation. After our concept and implementation is verified we present how we try to do the application with our self made drone. We will reveal the trouble we ran into and exhibit how we solved the problems.

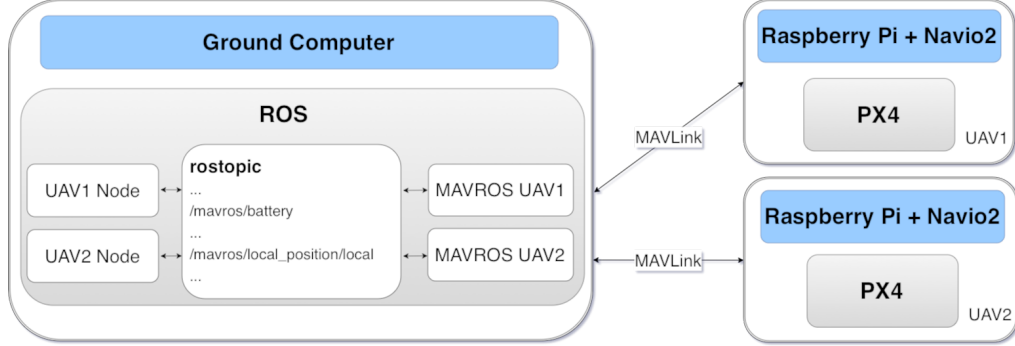


Figure 3.1: This Graph presents the connection between the different components of the replacement system. On the left hand we have the Ground Computer managing the replacement using ROS. Each *UAV Node* implements the logic and commands for one drone. Through each *MAVROS Node* the communication via MAVLink to the separate UAVs is enabled. On the right hand we have our two quadcopter running PX4 and communicating via MAVLink.

3.1 Implementation

The communication between our two drones is essential for the success of the task. It is of crucial importance that they know at any time where the other UAV is to replace it at the right spot but first and foremost to not collide and crash. On top of that, the battery status is of importance because the system has to know when the replacement has to be initialized. The drone is supposed to have enough remaining battery power to wait for the next drone and to be able to fly back to the landing point.

Since we used ROS in our implementation, we will illustrate how the framework works and how it helps us to let the two drones communicate and control them autonomously from a Ground Computer.

3.1.1 ROS

We want to create software that can run on any type of drone. Therefore we use *Robot Operating System* (ROS), which is not actually an operating system, but a robotics middleware. It is designed to reduce the complexity of hardware and communication of robots to make simple software design possible that is reusable across multiple platforms [19]. A ROS system works through a network of independent nodes communicating with each other through a publish/subscribe message model. Information is stored in the so called *rostopic* as indicated in Figure 3.1. After launching ROS we run two *MAVROS Node*, one for each UAV

on our Ground Computer. They manage all the information coming from PX4 in form of MAVLink messages and organize them in *rostopic*. To implement the actual flight logic we need two more self created *UAV Nodes*. We use for each UAV a separate node to have a simple extendable system. An instruction on how to launch ROS (*Master Node*), the *MAVROS Nodes* and the *UAV Nodes* can be found in the Appendix A.2. ROS comes with the *MAVROS Node* to manage the communication with PX4.

3.1.2 Mavros

We work with a ROS package called MAVROS. It enables us to work with the MAVLink messages for the direct communication with the drone's Flight Control Unit (FCU). It allows to publish to *rostopic* all incoming information passed by MAVLink from the drones FCU. This permits a self created node to get the information from the drones FCU. In addition, it enables the node to send a command through a publishing message. This command is first written in *rostopic* and then by the *MAVROS Node* converted in a MAVLink command and send to the FCU (in our case for example we publish a point where the drone has to go).

3.1.3 MAVLink

MAVLink means *Micro Air Vehicle Link* and is a protocol for communication. It is used by PX4 for the communication between the Flight Control Unit (FCU) and the Ground Computer. In Figure 3.1 we see where this protocol is in use.

3.1.4 Structure of a self coded *UAV Node*

In the self coded *UAV Node* we create the logic behind the replacement. We use mainly three messages. How we use them and the structure of these Nodes is shown below.

1. Including types of messages that should be received, for example:
 - (a) The charge percentage and voltage of the drone's battery:
`#include <geometry_msgs/BatteryState.h>`
 - (b) The position of a drone in cartesian coordinates:
`#include <geometry_msgs/Pose.h>`
 - (c) The target position of a drone:
`#include <geometry_msgs/PoseStamped.h>`

2. Defining callback functions to save data for received messages. Callback functions make it possible to retrieve the last send message containing for example the position of a UAV.
3. Initializing a subscriber or a publisher to a message included above. This is associated with listening and writing to *rostopic*.
4. Implementation of the logic with commands for drones.

3.1.5 Replacing procedure

We implement a logic that navigates the UAVs to do a simple replacement. This is the procedure controlled by the Ground Computer.

1. Start first UAV1 and fly mission.
2. Detect that the battery of UAV1 is getting low.
3. UAV1 stops the mission and holds position.
4. Start UAV2.
5. UAV2 listens to the position of the UAV1.
6. UAV2 flies to the position of the UAV1.
7. After a set amount of time UAV1 flies back and lands.
8. UAV2 continues task.
9. UAV2 flies home.

3.2 Simulation

Simulations are a great tool to test code and to see what the implementation does. Therefore, it was clear that we will implement a test procedure in which we will simulate two quadcopters seamlessly replace each other in an manner that would be easily replicable with our hardware. How we implement this simulation is shown in the following subsections.

3.2.1 Gazebo

As a powerful 3D simulation environment we use Gazebo (Gazebo Version 7.0) [20]. It is designed to test or prove autonomous robots by running it in the simulation environment (SITL, Software in the Loop), which is supported by

PX4. We need to work with not only one drone but at least two drones. So the environment had to be capable of working with multiple UAVs. In this case PX4 even provides a multi UAV launch file which works perfectly with Gazebo. In the Appendix A.3 there is an instruction to launch the simulation and in Figure 3.2 it is visualized how it looks like.

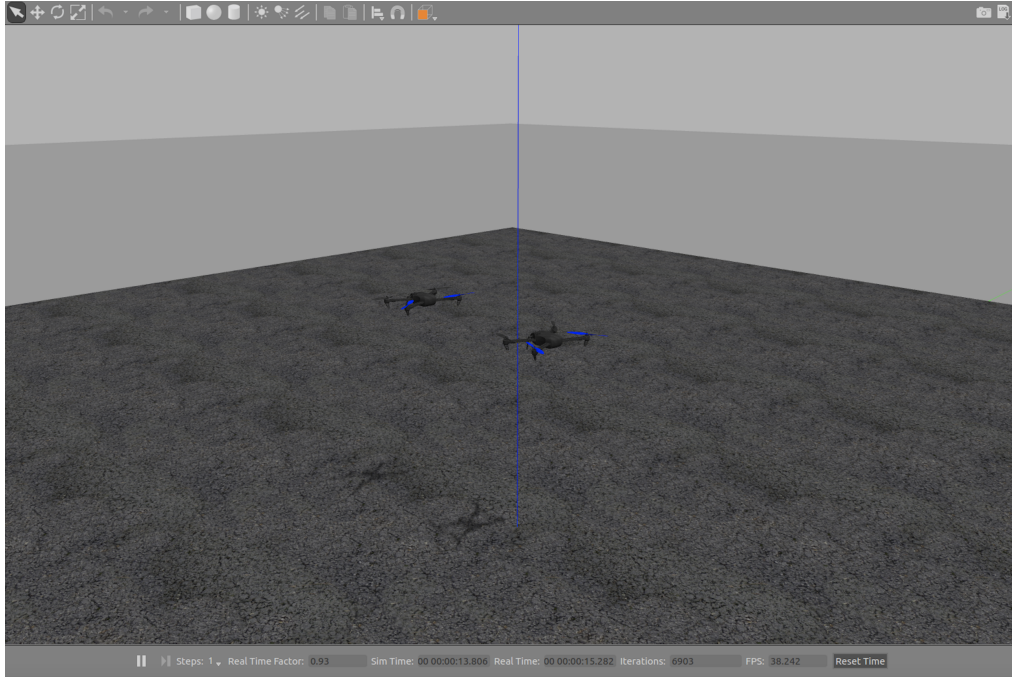


Figure 3.2: Drones flying next to each other to replace themselves.

3.3 Application

The next step is to make the replacement work with our self built drones. Our goal was to run it with PX4 because it is the autopilot developed at ETH Zürich. Our hardware was supposed to be supported since the PX4 Firmware release 1.4.1 [21]. However this was not the case. We ran into some problems and sadly we still have no autonomously flying PX4 code. Nevertheless, we want to show how we tackled the challenges and aimed to solve the occurring problems by applying a systematic approach.

3.3.1 GPS status *"NOT OK"*

After achieving a smooth manual flight by tuning the PID controllers we tried more autonomous flight modes. We tested the *Position Hold* flight mode because it is essential for our task. A problem appeared when switching in the flight modus. An error occurred indicating that the flight mode is rejected. We tried to isolate the problem and quickly found that we were not receiving any GPS position. We ensured that the antenna was receiving GPS satellite information and tested the GPS module with tools provided by Emlid [22] to ensure it was not broken. Since the GPS module was receiving satellites with Emlid code, we thought of PX4 having a driver issue. At this point we contacted the PX4 driver developer to help us. Together we found out that with the newer Debian version *Stretch* the GPS is not working but with the older *Jessie* it is. Up to fourteen satellites could be received and we had a GPS lock that works for *Position Hold* flight mode. However after having the GPS problem solved we still could not switch into the *Position Hold* flight mode.

3.3.2 Barometer driver

As we still were not able to switch into *Position Hold* flight mode, we had to find again the problem causing this fault. Again we tried to isolate the problem. With the Navio2 shield we are not using standard hardware for PX4. Therefore we were certain that again the problem has to come from the sensors or their drivers. We found by looking through the sensor status that something was wrong with the barometer. By checking with the *Emlid Tool* [23] we could ensure that the sensor was running and that it produces reasonable data. After trying to run a calibration for the barometer on PX4 and attaching foam over the barometer in case the air current of the propeller was influencing it, we derived that again it had to be a driver problem. Therefore we asked again for help from the PX4 driver developer. He found out that there was a bug in the barometer driver for the Navio2 shield. He fixed it and now the *Position Hold* flight mode is running. Now we could start with our autonomous flight tests but sadly this fix just came four days before our submission date [24].

Conclusion

Despite the fact that we could not assess the implementation with our own quadcopters, the simulation shows that seamlessly replacing drones work. We believe that replacing drones mid-air has the potential to become a key feature of autonomously flying quadcopter systems. Since buying a ready-to-fly quadcopter with some computational power is expensive, we tried to build an easy self constructed platform.

We faced unexpected challenges but with the effort we put in and the adjustments we made we were able to create a drone, which could represent a solid starting point for further research projects in this field.

4.1 Future work

- **Chain of replacement:** Why replacing only once and not more often? This expansion of our work can easily be implemented by adding more ROS nodes and adjusting a few lines of code. This feature would enhance the continuity aspect even further.
- **Combining charging on the ground with replacement in the air:** So far the flight time of the whole system is still bound to the amount of drones and their individual flight time. Autonomously changing the battery or recharging the drone on the ground would make a "life cycle" possible. This feature would unlock the ability of a system to practically having a drone flying and at the disposal at any time.

There are many more possibilities to expand this project besides the two features shown above. We are looking forward to see future projects and if autonomous quadcopter system of the future will use similar techniques.

Bibliography

- [1] Katsuya Fujii, Keita Higuchi and Jun Rekimoto: Endless flyer: A continuous flying drone with automatic battery replacement. <https://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=6726212&tag=1> Accessed: 2018-06-22.
- [2] Colin Smith: Flying drones could soon re-charge whilst airborne with new technology. <https://www.imperial.ac.uk/news/175318/flying-drones-could-soon-re-charge-whilst/> Accessed: 2018-06-22.
- [3] Jie Ouyang, Yueling Che, Jie Xu and Kaishun Wu: Throughput maximization for laser powered uav wireless communication systems. <https://arxiv.org/pdf/1803.00690.pdf> Accessed: 2018-06-22.
- [4] Emlid website: Emlid navio2 shield. <https://emlid.com/introducing-navio2/> Accessed: 2018-06-22.
- [5] Raspberry Pi website: Raspberry pi 3 model b. <https://www.raspberrypi.org> Accessed: 2018-06-22.
- [6] Linuxgizmos.com: Picture of the navio2 mounted on the raspberry pi. http://linuxgizmos.com/files/emlid_navio2_pi_angle1.jpg Accessed: 2018-06-22.
- [7] Hobbywing website: Hobbywing esc. <http://www.hobbywing.com/goods.php?id=588> Accessed: 2018-06-22.
- [8] Tiger Motors website: F40 1600kv. <http://store-en.tmotor.com/goods.php?id=718> Accessed: 2018-06-22.
- [9] FPVRacing.ch: 7 inch Propellers. <https://fpvracing.ch/de/propeller/2109-hqprop-dp-7x35x3-vis-durable-pc-propeller-schwarz.html> Accessed: 2018-06-22.
- [10] Conrad: Lipo battery shoppage. <https://www.conrad.ch/de/swaytronic-modellbau-akkupack-lipo-185-v-3800-mah-zellen-zahl-5-35-c-softcase.html> Accessed: 2018-06-22.
- [11] Frsky-rc website: Frsky taranis remote controller. <https://www.frsky-rc.com/product/taranis-x9d-plus-2/> Accessed: 2018-06-22.
- [12] Frsky-rc website: Frsky xsr receiver. <https://www.frsky-rc.com/product/xsr/> Accessed: 2018-06-22.

- [13] Telemetry Module website: Telemetry module. <https://synosystems.de/de/px4-open-hardware-kompatible/278-433-mhz-telemetrie-kit-mavlink-fur-pixhawk-px4-kompatible.html> Accessed: 2018-06-22.
- [14] Emlid website: Picture of the raspberry pi and navio2 connected with the other components. <https://docs.emlid.com/navio2/ardupilot/typical-setup-schemes/> Accessed: 2018-06-22.
- [15] PX4 Project website: Px4 architecture. <https://dev.px4.io/en/concept/architecture.html> Accessed: 2018-06-20.
- [16] Arturo Urquizo: Pid controller. <https://commons.wikimedia.org/wiki/File:PID.svg> Accessed: 2018-06-21.
- [17] PX4 Project website: Px4 controller tuning guide. https://docs.px4.io/en/advanced_config/pid_tuning_guide_multicopter.html Accessed: 2018-06-21.
- [18] Wikipedia: Picture of yaw, pitch and roll in an aircraft. https://en.wikipedia.org/wiki/Aircraft_principal_axes#/media/File:Yaw_Axis_Corrected.svg Accessed: 2018-06-23.
- [19] Elkady, A., Sobh, T.: Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*
- [20] Gazebo website: Gazebo. <http://gazebo.org> Accessed: 2018-06-20.
- [21] PX4 Project on Github: Px4 firmware release 1.4.1. <https://github.com/PX4/Firmware/releases/tag/v1.4.1> Accessed: 2018-06-19.
- [22] Emlid website: Emlid gps testing tool. <https://docs.emlid.com/navio2/common/dev/gps-ublox/> Accessed: 2018-06-23.
- [23] Emlid website: Usage of emlid tool. <https://docs.emlid.com/navio2/common/ardupilot/installation-and-running/> Accessed: 2018-06-22.
- [24] PX4 Project on Github: Barometer bug fix. <https://github.com/PX4/Firmware/pull/9721> Accessed: 2018-06-22.
- [25] Raspberry Pi website: Tutorial to expanding the filesystem using raspi-config. <https://www.raspberrypi.org/documentation/configuration/raspi-config.md> Accessed: 2018-06-21.
- [26] PX4 Project website: Setting up the raspberry pi. https://docs.px4.io/en/flight_controller/raspberry_pi_navio2.html Accessed: 2018-06-21.

- [27] PX4 Project website: Setting up the linux toolchain. https://dev.px4.io/en/setup/dev_env_linux.html Accessed: 2018-06-21.
- [28] PX4 Project website: Cross-compiling px4 firmware. https://dev.px4.io/en/setup/building_px4.html Accessed: 2018-06-21.

Appendix Chapter

A.1 Raspberry Pi 3 Model B Installation

This is a guide to installing Debian (Jessie) with the PX4 Firmware on the Raspberry Pi by cross-compiling from another Linux machine.

- Flash the OS Image *Jessie* to the SD card of the *Raspberry Pi* (Using Etcher for example)
- Make sure that connecting over ssh to the Raspberry Pi is possible.
\$ ssh pi@navio
- Make sure to expand the file system of the Raspberry PI with raspi-config [25].
- Edit */boot/config.txt* by commenting the line enabling the navio-rgb overlay [26].
- Install the development toolchain for Linux to cross-compile for the Raspberry Pi [27]
- Download the PX4 Firmware on the Linux machine.
\$ git clone https://github.com/PX4/Firmware.git
- Compile the PX4 Firmware for the Raspberry Pi on the Linux machine [28].
\$ cd Firmware
\$ make posix_rpi_cross
- Get the IPv4 address of the Raspberry Pi.
\$ ifconfig
- Set the IP of the Raspberry Pi on the Linux machine.
\$ export AUTOPILOT_HOST=192.168.X.X
- Upload the PX4 Firmware from the Linux machine [28].
\$ cd Firmware
\$ make posix_rpi_cross upload

- Run PX4 in terminal of the Raspberry Pi.
\$ *sudo ./px4 px4.config*
- PX4 can run on autostart by adding the following line to */etc/rc.local* just above *exit 0*.
\$ *cd /home/pi && ./px4 -d px4.config > px4.log*

More details can be found on the *dev.px4.io* homepage.

A.2 Instruction to launch ROS (*Master Node*), the *MAVROS Nodes* and the *UAV Nodes*

- Launch the *Master Node*.
\$ *roscore*
- Launch the autopilot (in Raspberry Pi terminal).
\$ *sudo ./px4 px4.config*
- Launch the *MAVROS Nodes* twice addressing each FCU separately.
\$ *roslaunch px4.launch*
- Launch the *UAV Nodes*
\$ *roslaunch <uav1_node_folder> <uav1_node_file>*

A.3 Instruction to launch Gazebo Simulation

- Launch the *Master Node*.
\$ *roscore*
- Launch the gazebo, two PX4 and two MAVROS nodes.
\$ *roslaunch px4_multi_uav_mavros_sitl.launch*
- Launch the *UAV Nodes*
\$ *roslaunch <uav1_node_folder> <uav1_node_file>*
\$ *roslaunch <uav2_node_folder> <uav2_node_file>*