Erweiterung eines Quadrokopters um einen kameragestützen Paketgreifer

12. März 2020

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bei dieser Projektarbeit geht es um die Entwicklung einer Paketgreiffunktion für einen bereits funktionierenden Quadrokopter. Die zu entwickelnde Hardware lässt sich dabei in vier Kategorien einteilen: Ein Greifarm, der in der Lage ist, Pakete mit einer Größe von ca. 10 cm zu greifen und zu halten; eine Kamera, die das zu greifende Paket erkennt; diverse Sensorik und Aktorik für den Greifarm und anderweitige Kontrollfunktionen; ein System-on-a-Chip (SoC) für die Bildanalyse, Regelung und Steuerung des Greifarms und des Quadrokopters, wobei dieser selbst bereits mit einem weiteren SoC ausgestattet ist, mit welchem für die Steuerung nur kommuniziert werden muss. Es gibt eine Vielzahl verschiedener SoCs, wobei sich hier aufgrund weiter Verbreitung für den Raspberry Pi (RPi) entschieden wurde, welcher, salopp gesagt, als das Gehirn der neu entwickelten Funktionalität dient. Im Folgenden soll es nun um die grundlegende Softwarearchitektur und -infrastruktur gehen, welche die Aufgaben des Raspberry Pis auf diesem implementiert.

Kapitel 2

Anforderungen

2.1 Softwareanforderung

Ein weiterer Teil des Projektes ist die Software. Die Aufgabe dieser erstreckt sich von der Bildverarbeitung über den Objekterkennungsprozess bis hin zu der Regelung und Steuerung der Drohne. Dabei muss sie verschiedensten Kriterien genügen.

Da sie Signale von einem Externen Bauteilen wie dem PX4 oder dem Abstandssensor über das Interface ROS(siehe Kapitel Interface) kann es zu Datenverlust kommen. Bedingt durch die unterschiedlichen Taktrate der Prozessoren. Deshalb muss auch bei gelegentlich fehlenden Datenpaketen der Rest verarbeitet und die Drohne auf Grundlage dieser geregelt werden.

Der PX4-Controller schreibt eine Datenrate von 20 Hz vor. Also muss in dieser Datenrate die Sollposition der Drohne geschickt werden. Fallen diese aus, so geht die Drohne in den Offboard-Modus und steigt auf 10 m Höhe, um dann automatisch zur Startposition zurückzukehren. Dies wäre bei einer Raumhöhe von 2,5 m fatal. Leider kann man diese „Schutzfunktion“ nicht abschalten da sie fest im Betriebssystem des PX4-Controllers verankert ist. Demzufolge darf das Signal der Position nie ausbleiben.

Um dies zu erreichen, darf sich die Regeleinheit der Drohne nie aufhängen. Alle Schleifen müssen deshalb ein zeitliches Abbruchsignal haben, um eine Auslastung des Arbeitsspeichers zu verhindern. Aufgrund des Gewichtes muss ein leichter Controller gewählt werden(siehe Hardware Komponenten). Da wir aufgrund der Software Inkompatibilität des Pi-4 mit dem PX4-Controller einen Pi-3 gewählt haben steht ein maximaler Arbeitsspeicher von 1 Gb zur Verfügung. Dabei hat er 2 Kerne und kann somit effektiv 2 Aufgaben gleichzeitig ausführen. Die Software muss also sehr ressourcensparend geschrieben sein.

Aufgrund der Struktur von ROS empfiehlt es sich, einen Event basierten Programmablauf zu schreiben. Dabei löst ein Event, also eine Aktion die stattfindet, einen Programmablauf aus. In diesem Fall sind die Events eingehende Datenpakete für die entsprechende Node (Siehe Programmstruktur). Diese werden verarbeitet. Dadurch kann sichergestellt werden, dass jedes Datenpaket verarbeitet wird, unabhängig in welchem Abstand es ankommt. Durch diese Warte-Bedien-Struktur wird eine maximale Effizienz

erreicht, da das Programm keine Schritte doppelt auf nicht aktualisierte Werte anwendet und so Prozessorzeit spart.

2.2 Hardwareanforderungen

Kapitel 3

Greifermechanik

3.1 Einleitung

In Zeiten wo manchmal jede Sekunde zählt und der Verkehrsfluss auf den Straßen nicht immer garantiert ist, ist der schnelle Luftweg mit einer Drohne oftmals die bessere Option. Ein Automatisierter Transport über den Luftverkehr könnte die Lebensqualität in vielen Bereichen verbessern und dabei Kosten und Umweltschadstoffe, durch die ersparte Fahrt, sparen. Dabei geht der konstruktive Aspekt, nicht nur aber vor allem, im Luftverkehr mit Leichtbau und gleichzeitiger Stabilität einher. Ziel der konstruktiven Arbeit am und mit dem Quadrokopter ist ein modular montierbarer Greifarm, welcher mit Kameras und Sensoren sein Ziel selbst finden kann. Dabei ist die Mechanik des Greifarms der letztendliche Kern der Arbeit und beginnt an der Verbindungsstelle mit dem Motor und endet mit der letztendlichen Kontaktstelle zur Last, welche transportiert werden soll. Dabei sollte der Fokus der konstruktiven Ausarbeitung eine stabile, funktionssichere, leichte und dauerfeste Konstruktion sein, welche selbst bei nicht idealen Bedingungen ihre Aufgaben erfüllt und dabei insbesondere das Wohlergehen von Passanten nicht gefährdet. Beim Aspekt der Stabilität und Sicherheit ist insbesondere die Greifsicherung im Falle eines Defekts und die Lage des Schwerpunkts zu beachten. Gleichzeitig darf ein maximales Gewicht von 2kg nicht überschritten werden, da sonst ein spezieller Drohnenführerschein als Nachweis zum Bedienen des Quadrokopters nötig ist. Für die Konstruktion steht als Grundmaterial für den Rahmen Plexiglas zur Verfügung, welches mit Laserschneidemaschinen in Formen geschnitten werden kann. Weitere benötigte Bauteile wie zum Beispiel ein Motor oder Lagerungen gelten als Zukaufteile. Die tiefere Forschungsfrage hinter dem Projekt ist die Visualisierung des Greifarms im späteren Verlauf, welche Möglichkeiten einem bei der Konstruktion beim Leichtbau und der Stabilität mit dem vorausgesetztem Material gegeben sind und wie gut das Konzept umsetzbar ist. Für die Auswahl eines Konzepts wird zwischen mehreren Prinzipskizzen eine kategorisch ausgewählt und vollständig dimensioniert. Dann wird nochmals die Realisierung geprüft mit einem CAD Modell bzw. Simulation.

Kapitel 4

Sesorik, Aktorik und andere Hardware

Kapitel 5

Softwareimplementierung

5.1 Python als Programmiersprache

Wie es häufig in der Entwicklung der Fall ist, stehen auch hier für die Implementierung der Software auf dem RPi verschiedene Plattformen, Frameworks und Programmiersprachen zu Verfügung. Bei der Wahl der Programmiersprache ist zu beachten das sie möglichst einfach und bereits relativ alt ist. Der Vorteil von alten Programmiersprachen ist, dass es eine umfangreiche Beispielsammlung gibt. Dadurch kann man sich leicht an bereits vorhandenem Code orientieren. Anforderungs-bedingt muss die Programmiersprache möglichst ressourcenschonend sein. Dadurch fallen alle grafischen Programmiersprachen wie Matlap, Simulink, Labview etc. heraus. Diese haben aufgrund ihres unstrukturierten Programmablaufs, eine eher schlechte Laufzeit.
In dem Studium wurde bei uns Java gelehrt. Der Vorteil von Java ist, das es plattformübergreifend funktioniert. Leider verwendet Java dafür eine Virtuelle Java Maschine. Also ein Betriebssystem, dass noch mal über dem Betriebssystem liegt. Dies verringert die Laufzeit von Java enorm. Ein weiteres Problem liegt darin, dass Schnittstellen immer plattformabhängig sind(USB, D-Sub, Ethernet etc.). Dadurch werden sie durch Java nur schlecht unterstützt. Aus diesen Gründen konnte Java ebenfalls nicht gewählt werden.

Eine weitere Alternative ist eine C-Programmiersprache(C++, C etc.). Diese haben den großen Vorteil, dass sie echtzeitfähig sind. Sie können also unter Umständen garantieren, dass das Ergebnis des Algorithmus nach einer bestimmten Zeit feststeht. Dies ist für eine Drohnensteuerung sehr geeignet, da man garantieren sollte, dass sie Drohne in einem bestimmten Zeitintervall überwacht wird. Außerdem sind C-Sprachen sehr Hardware nah geschrieben, wodurch die Laufzeit sehr gut ist. Allerdings nutzen wir die Hought-Transformation der Bibliothek Open-cv. Diese Bibliothek ist allerdings nicht echtzeitfähig, wodurch der große Vorteil von C verloren geht. Des Weiteren ist es sehr schwer/mühselig Open-cv in C zu implementieren.

Außerdem ist Python eine Alternative. Python ist leider nicht echtzeitfähig. Jedoch ist es sehr benutzerfreundlich, hinsichtlich des Importierens von Bibliotheken. Es ist nicht so Hardware nah wie C, allerdings immer noch deutlich näher als Java oder Labview. Außerdem gibt es zahlreiche Beispiele für Python und das Ansteuern von Schnittstellen geht problemlos. Aus all diesen Gründen ist die Wahl auf Python gefallen.

5.2 Vergleich Monolithische Single Application und Service Oriented Architecture

5.2.1 Monolithische Single Application

Bei der Softwarearchitektur wurde sich zunächst an einer normalen objektorientierten Struktur bedient: Es gibt eine main-Datei, welche vergleichsweise klein ist und nur die grobe Struktur bzw. den Gesamtablauf darstellt. Diese ruft dann weitere Softwaremodule auf, die die Unteraufgaben, wie beispielsweise das Empfangen und Verarbeiten von Bilddaten, implementiert. Ein Problem, das hier recht schnell auftritt ist, dass theoretisch zwei Prozesse gleichzeitig ablaufen müssen. Wenn ein Motormodul beispielsweise gerade darauf wartet, dass die Sollposition erreicht wurde, müsste in einem Programmierstrang, in dem ein Befehl nach dem anderen ausgeführt wird, das Kameramodul mit der Verarbeitung des Bildes auf das Motormodul warten, obwohl das Motormodul gerade gar nichts tut, außer auf den Motor in der echten Welt zu warten, bis er die gewünschte Position hat. Da das Kameramodul aber an das Motormodul eventuell kommunizieren muss, dass das Paket gar nicht mehr an der richtigen Stelle ist, weil der Quadrokopter aufgrund äußerer Umstände nicht ganz stillsteht, müssen beide Module gleichzeitig arbeiten und miteinander kommunizieren können.

In der traditionellen Softwareentwicklung gibt es dafür sogenannte Threads. Diese sind separate Ausführungsstränge, die quasi gleichzeitig ausgeführt werden und sich dann beispielsweise ein Datenobjekt im Speicher teilen und über dieses miteinander kommunizieren können. Das entspricht dann einer Dreischicht-Architektur. Wichtig bei dieser Art

Abbildung 5.1: Dreischichtarchitektur

der Architektur ist, dass jede Schicht nur auf die nächst untere zugreifen darf, um einen reibungslosen Ablauf garantieren zu können. In diesem Fall greifen nun die Threads T_1, T_2 ... T_n nicht direkt auf den Speicher zu, sondern müssen den Weg über die Datenzugriffsschicht gehen. Diese stellt unter anderem mittels Semaphoren sicher, dass immer nur ein Thread auf die entsprechende Ressource zugreifen kann. Damit wird verhindert, dass beispielsweise T_1 schreiben und T_2 gleichzeitig lesen möchte. Dies würde zu einem Speicherkonflikt führen und das Programm wird abstürzen. (Riedel, 2019)

5.2.2 Service Oriented Architecture

Bis zu einem gewissen Grad war es möglich die geforderten Funktionalitäten mit Multithreading zu implementieren. Allerdings wurde es im Laufe der Entwicklung immer komplizierter, da Bilddaten beispielsweise von zwei Ressourcen abhängen, die nicht immer verfügbar sind: zum einen der Speicherplatz im Datenobjekt, das andere Threads lesen und damit belegen wollen und zum anderen von der Kamera selbst, die natürlich auch nicht willkürlich Bilddaten liefert. Das zu synchronisieren ist zwar theoretisch möglich, praktisch aber mit einer Menge schwer wartbarem Code verbunden, der das Problem unnötig kompliziert löst.

5.2.3 Robot Operating System

Deshalb basiert die endgültige Software auf ROS, das genau diese Probleme adressiert. ROS steht dabei für Robot Operating System und ist ein Framework, das auf dem Publisher-Subscriber-Modell basiert. Die Idee ist dabei, dass es verschiedene Themen (Topics) gibt, in die verschiedene dezentrale Skripte, die gleichzeitig laufen, Informationen zu einzelnen Themen veröffentlichen oder abonnieren können. So gibt es beispielsweise ein Kameraskript, das das Bild und die erkannte Position des Pakets veröffentlicht und ein Motorskript abonniert die Paketposition und kann daraus dann weitere Schlüsse ziehen. Ein anderes Skript könnte dann auf einem anderen Rechner (z.B. Laptop) im gleichen (WLAN-) Netzwerk das Kamerabild abbonnieren und mit sehr wenig Programmieraufwand anzeigen. Das elegante dabei ist, dass sämtliche vorher beschriebene Multithreading-Probleme dabei vom sogenannten ROS-Core, der zentrale Schaltstelle des ROS, übernommen werden und damit die einzelnen (selbstgeschriebenen) Skripte sehr entschlackt werden. Das macht es deutlich einfacher diese zu debuggen. ROS ist zudem sehr gut dokumentiert und für viele Probleme gibt es bereits vorgefertigte Skripte, die Dank des einfachen P/S-Modells auch einfach anzubinden sind.

5.2.4 ROS als Beispiel einer Service Oriented Architecture (SOA) / Microservices

Die Architektur hinter ROS ist dabei keine neue Erfindungen, sondern basiert im Gegensatz zum Ansatz der Schichtenarchitektur auf dem Prinzip der serviceorientierten Architektur (SOA). Bei der SOA spielt der (Enterprise) Service Bus eine zentrale Rolle.

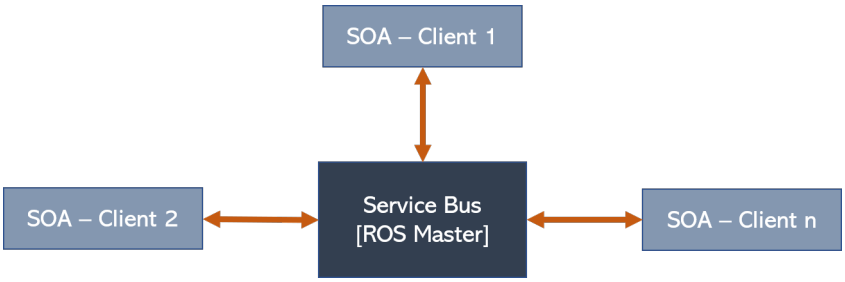

Abbildung 5.2: Service Oriented Architecture

Er ist das Bindeglied zwischen allen anderen Knotenpunkten im Netzwerk und stellt sicher, dass die Kommunikation funktioniert. Bei ROS ist das der ROS-Master, der über den Befehl `roscore` gestartet werden kann. Über ihn kommunizieren die Knoten miteinander, wobei jeder Knoten eine logisch abgeschlossene Aufgabe übernimmt und von außen über eine API angesprochen werden kann. Im Fall des Quadrokopters existiert nun beispielsweise ein Knoten, der das Kamerabild publiziert, einer, der das Bild verarbeitet und einer, der die Steuerung auf Basis der Bilddaten durchführt. Durch diese Modularität gibt es einige Vorteile. So ist es damit sehr leicht einen Knoten auszutauschen, zu erweitern oder neue Knoten hinzuzufügen. Zudem ist das System deutlich stabiler als eine monolithische Lösung, da bei einem fehlerhaften Knoten nur die Knoten ausfallen, die von diesem Ausfall logisch betroffen sind. Alle unabhängigen Knoten werden weiterhin funktionieren. Bei einer monolithischen Lösung führt ein Modulausfall zum Ausfall des Gesamtsystems.

Aber einer gewissen Granularität spricht man von Microservices. Diese folgen der Strategie Ërledige nur eine Aufgabe und erledige sie gut"Bei "normalenSSOAs sind die Knoten tendenziell groß und implementieren viele Funktionalitäten auf einmal. Microservices hingegen sind sehr klein und erledigen nur eine bestimmte Aufgabe. In der finalen Gesamtarchitektur kommen Knoten zum Einsatz, die sowohl den normalen SOAs als auch den Microservices zugewiesen werden können.

5.3 Finale Gesamtarchitektur

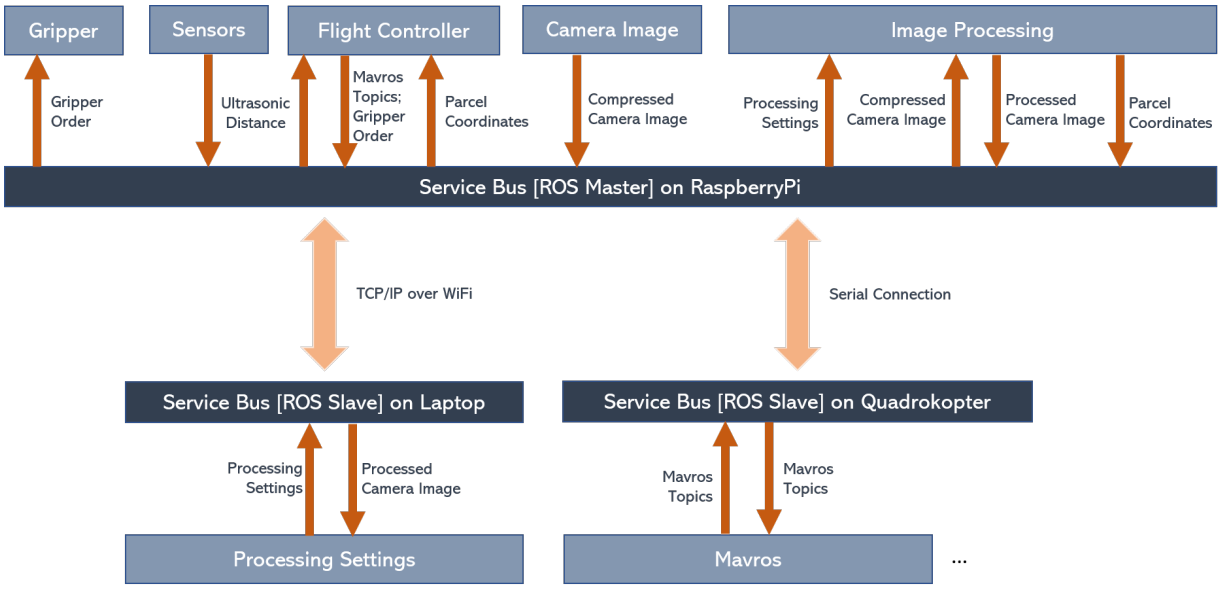

Abbildung 5.3: Service Oriented Architecture

In der finalen Gesamtarchitektur gibt es insgesamt drei ROS Instanzen: Die erste läuft auf dem Quadrokopter selber und dient als Schnittstelle zur Firmware des Quadrokopters. Der dazugehörige Knoten heißt Mavros, welcher mit MAVLink (Micro Air Vehicle Link) kommuniziert, das die gesamte tatsächliche Steuerung des Quadrokopters übernimmt. Dabei hat der ROS-Core jedoch keine Master-Funktion, sondern verbindet sich über eine serielle Schnittstelle mit dem ROS-Core auf dem RaspberryPi, auf dem der tatsächliche Master-ROS-Core läuft. Dieser ist in einem WLAN-Netzwerk mit einem Laptop, der für Debugging-Zwecke und zum Konfigurieren und Kontrollieren der Bildverarbeitung benötigt wird. Auch auf dem Laptop läuft ein ROS-Core, wobei auch dieser sich mit dem Master auf dem Raspberry Pi verbindet. Im folgenden werden nun die einzelnen Knoten genauer vorgestellt.

5.3.1 Camera Image

ist ein einfacher Publisher, der das Bild der Dronenkamera komprimiert und über den Service Bus den anderen Knoten im Netzwerk als serialisiertes numpy-Array zu Verfügung stellt. Aufgrund der später folgenden rechenintensiven Bildverarbeitung sendet dieser in der finalen Version das Bild jedoch nur mit 10 Hz. Dies is jedoch immer noch ausreichend, da sich die Geschwindigkeit der Drone nicht schnell ändern wird.

5.3.2 Image Processing

empfängt die Bilddaten von Camera Image und wir zudem über das Topic Processing-Settings vom Laptop aus konfiguriert. Dieser Knoten filtert dann das Bild um anschließend die Position und Drehung des Pakets zu ermitteln. Das gefilterte Bild und die Koordinaten als Typ `PoseStamped` des Pakets werden publiziert, wobei die Z-Koordinate des `PoseStamped` die Drehung angibt.

5.3.3 Sensors

ist ein Knoten, der Sensordaten wie den Ultraschallsensor zum Boden aufbereitet, validiert und dann publiziert. Die Ultraschalldaten werden dann beispielsweise unter der Topic `/sensor_dist` im Datenformat Float32 mit einer Frequenz von 20 Hz veröffentlicht.

5.3.4 Flight Controller

ist die zentrale Steuereinheit der neuen Funktionalität. Er empfängt alle aufbereiteten Sensordaten wie die Koordinaten des Pakets, Ultraschalldistanz, Sensordaten der Drone etc. und führt die Suche und Anflug des Pakets durch. Er gibt die Flugsteuerbefehle an Mavros über die serielle Schnittstelle weiter und gibt Greifbefehle weiter. Mehr dazu ist im Kapitel "Programmablaufßu finden.

5.3.5 Datenübertragungsformate

Es war sehr schwierig, die richtigen Formate zu wählen da Python im Allgemeinen eher unsauber mit den verschiedenen Datentypen umgeht. So gibt es z. B. keinen Unterschied zwischen einem 32-Bit oder einem 64-Bit Integrer. Falls die Zahl außerhalb des darstellbaren Bereiches des 32-Bit Interger liegt, so erweitert der Python-Interpretor es selbstständig zu einem 64 Bit Integer. ROS ist hingegen sehr Typenspezifisch, wodurch es zuerst einige Unklarheiten bezüglich der Typen gab und diese dementsprechend verändert werden mussten.

5.4 Softwareentwicklungsprozess

Für eine effektive und effiziente Entwicklung des Softwaresystems ist die Nutzung geeigneter unterstützender Software unabdingbar. Daher soll nun die verwendete Toolchain kurz vorgestellt werden.

5.4.1 ROS auf Ubuntu

5.4.1.1 Setup

Sowohl die Entwicklung als auch die Runtime läuft auf Ubuntu 18.04. Das gilt also sowohl für Entwicklungsrechner, als auch für den Raspberry Pi, wobei bei diesem Ubuntu Mate eingesetzt wird. Der Vorteil ist, dass ROS nativ darauf funktioniert und sehr viele Bibliotheken für ROS bereits vorkompiliert zur Verfügung stehen. Es ist dabei für die aktuelle ROS-Version "Melodic"dringend davon abzuraten, das System auf einer Raspian-Installation laufen zu lassen, da man einerseits alle Pakete selber kompilieren muss und darüber hinaus viele der Abhängigkeiten von MAVROS nicht zu Verfügung stehen. Wichtig ist auch, dass "Melodicäusschließlich von der Ubuntu-Version 18.04 unterstützt wird. Die Installation von ROS lässt sich auf Ubuntu nach Hinzufügen des Repositories über

`sudo apt install ros-melodic-ros-base` einfach durchführen. Hierbei ist es empfehlenswert für den Raspberry Pi die `ros-base` Version und für den Entwicklungsrechner die `desktop-full` Version zu nehmen, da diese bereits die wichtigsten grafischen Werkzeuge installiert hat. Nachdem man nun noch rosdep initialisiert, welches das Arbeiten mit Softwareabhängigkeiten deutlich vereinfacht, müssen nun die mitinstallierten ROS-Pakete im aktuell genutzten Terminal mit Hilfe des `source` Befehls geladen werden. Bei einer Standardinstallation geht das mit `source /opt/ros/melodic/setup.bash`. (Martinez & Fernández, 2013)

5.4.1.2 Workspaces

Alle eigenen Entwicklungen finden im sogenannten catkin-Workspace statt, worin sich die eigenen Pakete befinden. Dieser kann im home-Verzeichnis des Benutzers nach Erstellen des Verzeichnisses "catkin_ws"mit `catkin_make -DPYTHON_EXECUTABLE=/usr/bin/python3` erstellt werden. Dabei muss `catkin_make` jedes mal ausgeführt werden, wenn ein neues Paket erstellt wurde. Das kann mit dem Befehl `catkin_create_pkg my-package-name std_msgs rospy roscpp` erreicht werden. Dadurch wir ein gleichnamiger Ordner im ßrcÖrdner des Workspaces erstellt, in dem die Sourcedateien abgelegt werden können. Um die neuen Pakete nun auch in ROS zu Verfügung zu haben, müssen diese auch im Terminal geladen werden. Das geht analog zum Laden der vorinstallierten Pakete mit `source ~/catkin_ws/devel/setup.bash`.

5.4.1.3 .bashrc

Um nicht bei jedem neuen Terminal erst die Pakete manuell laden zu müssen, bietet Ubuntu die Möglichkeit dies automatisch zu tun. Dafür ist die Datei .bashrc zuständig, die sich im home-Verzeichnis eines jeden Benutzer befindet. In diese können die beiden Befehle einfach angehängt werden. Auch die Konfiguration für Gazebo (siehe Kapitel x.x.x) kann hier bereits erfolgen. Es sei aber zu erwähnen, dass dies nur so lange sinnvoll ist, wie ROS hauptsächlich auf dem System genutzt wird.

5.5 Entwicklungsumgebung und Versionsverwaltung

Eine (integrierte) Entwicklungsumgebung sollte den Entwickler in seiner Arbeit unterstützen und ihm sich wiederholende oder logisch einfache, aber zeitaufwändige Schritte abnehmen. Für diese Entwicklung fiel daher die Wahl auf "PyCharm"der Firma JetBrains. Dieses bietet neben dem obligatorischen Texteditor mit integriertem Auto-Complete und Compiler Vorschläge zu Codeverbesserung Refactoring etc. Es macht hier jedoch Sinn sich eine der ROS-Erweiterungen für PyCharm zu installieren, damit die ROS-eigenen Python Bibliotheken auch korrekt erkannt werden. PyCharm arbeitet zudem mit Virtual Environments (venv), also einer abgekapselten, meist projektspezifischen Python-Installation, die nur die zusätzlichen Bibliotheken enthält, die für das aktuelle Projekt benötigt werden. Diese sollte bei allen Entwicklern identisch sein, um Versionsinkompatibilitäten zu vermeiden. Ein weitere Funktion von PyCharm ist die integrierte Versionsverwaltung, kurz VCS (Version Control System). Diese bietet eine direkte Anbindung an Git, das Versionen des Sourcecodes verwaltet und Teilentwicklungen in den verschieden Entwicklungszwei-

gen (branch) der Entwickler effektiv zum Hauptzweig (master) zusammenbringt (merge). Konflikte, wenn beispielsweise eine Datei von zwei zu vereinenden Zweigen bearbeitet wurden, müssen jedoch oft von Hand gelöst werden. Deshalb macht es Sinn Funktionalitäten weitestgehend in einzelne Dateien zu unterteilen.

5.6 Implementierung der Kommunikation in ROS

Wie bereits erwähnt basiert ROS auf dem Publisher/Subscriber Prinzip. Ein Knoten kann in ROS sowohl Publisher als auch Subscriber sein und das auch für mehrere Topics. Die Topics sind dabei hierarchisch aufgebaut und gehen vom Groben ins Feine. So publiziert beispielsweise der "Camera Image"Knoten das komprimierte Bild in `/camera/image/compressed`. Anhand dieses Beispiels soll nun dargelegt werden, wie man prinzipiell bei der Erstellung eines Nodes vorgeht.

5.6.1 Kamera Publisher

Zunächst muss eine Datei mit dem Namen des Knotens und der Endung ".pyïm entsprechenden Paketordner im src-Ordner des Catkin-Workspaces erstellt werden. Diese kann dann in PyCharm geöffnet werden, wenn nicht schon das gesamte Paket als Projekt geöffnet wurde. Um klarzumachen, dass es sich hierbei um ein Python File handelt, muss die Datei mit `#!/usr/bin/env python3` starten, um die Pythonumgebung auszuwählen. Es folgt nun für gewöhnlich die Lizenzerklärung zur Datei. Es sollten nun `numpy as np, time, sys, cv2, roslib, rospy` importiert werden, sowie `CompressedImage` aus der ROS-Bibliothek `sensor_msgs.msg`. Es empfiehlt sich nun die gesamte Funktionalität mittels `def function_name` in eine Funktion zu packen. Dort muss dann zunächst mit `pub = rospy.Publisher(, DataType)` ein Objekt erstellt werden, wobei hier das Topic `/camera/image/compressed"`, sowie der Datentyp `CompressedImage` dem Konstruktor übergeben werden müssen. Mit `rospy.init_node('node_name', anonymous=True)` registriert sich der Knoten nun am ROS-Service-Bus. Nun wird die Computerbildverarbeitungsbibliothek OpenCV genutzt, um mit `cap = cv2.VideoCapture(0)` ein Objekt zu erstellen, dass auf den Datenstrom der ersten angeschlossenen Kamera zugreift. Der nun folgende Code soll so lange ausgeführt werden, wie ROS aktiv ist. Dies lässt sich mit `while not rospy.is_shutdown():` implementieren. Mit `ret, frame = cap.read()` wird nun der Datenstrom der Kamera ausgelesen und das aktuelle Bild in die Variable `frame` geschrieben. Jetzt muss mit `msg = CompressedImage()` ein Nachrichtenpaket des Typs `CompressedImage` erstellt werden, das mit `msg.header.stamp = rospy.Time.now()` um einen Zeitstempel erweitert wird und mit `msg.format = "jpeg"` das richtige Dateiformat erhält. Die Daten des Pakets werden mit der Property `msg.data` gesetzt. Diese müssen nun aus dem vorher gelesenen `frame` erstellt werden. Dafür wird zunächst mit `cv2.imencode('.jpg', frame)[1]` das `frame` mithilfe von OpenCV in ein JPEG umgewandelt und wird dann mit `np.array()` zu einem Bildarray transformiert wird und mit `tostring()` serialisiert wird. Nun ist Paket bereit mit `pub.publish(msg)` an den Service Bus gesendet zu werden. Die Funktion sollte nun mit `if __name__ == '__main__': function_name()` aufgerufen werden, wobei es Sinn macht den Funktionsaufruf mit einer Ausnahmeregelung für die `rospy.ROSInterruptException` zu erweitern.

5.6.2 Kamera Subscriber

Der dazugehörige Subscriber ist im Knoten İmage Processingïmplementiert und ist dem Publisher gegenüber recht ähnlich aufgebaut. Hier wird jedoch ein Subscriber-Objekt mit `rospy.Subscriber("topic", DataType, callback)` erstellt, wobei callback die Callback-Funktion ist, die aufgerufen wird, wenn eine neue Nachricht in der Topic ankommt. In den Argumenten dieser Funktion eine Variable `msg` definiert sein, in der die Nachricht bei Empfang gespeichert wird. Der Inhalt der Nachricht kann mit `msg.data` aufgerufen werden und mit `np.fromstring(data, np.uint8)` zuerst zu einem `numpy`-Array deserialisiert und dann mit `cv2.imdecode(np_arr, cv2.IMREAD_COLOR)` zu einem für OpenCV verarbeitbarem Format dekodiert werden. Der darauf folgende Code zur Bildverarbeitung ist in Kapitel x.x.x. beschrieben. (*Writing a Simple Publisher and Subscriber (Python)*, 2019)

5.7 Starten und Automation eines ROS-Systems

Damit die Knoten auch lauffähig sind, müssen die Dateien zunächst mit `chmod +x` für den Kernel ausführbar gekennzeichnet werden. Nun können, nachdem `roscore` ausgeführt wurde, in einem jeweils neuen Terminaltab mit `rosrun paketname knotenname.py` die verschiedenen Knoten ausgeführt werden. Mit `rostopic echo /topic/name` können die Nachrichten der verschiedenen Topics überwacht werden. Um das alle nicht jedes Mal von Hand machen zu müssen, gibt es bei ROS sogenannte Launch-Files, die, wenn ausgeführt, nicht nur einen ROS-Core ausführen, sondern auch alle notwendigen Knoten startet, Umgebungsvariablen setzt etc. Das Launch File basiert auf XML und enthält innerhalb des `<launch></launch>` Tags alle Komponenten. Ein Knoten ist dabei recht selbsterklärend aufgebaut: `<node name="CameraImagePub"pkg="parcelcopter"type="CameraImagePub.py/>`. Innerhalb des Tags können außerdem Startparameter mit `arg` definiert werden.Mit dem `env`-Tag können zudem Umgebungsvariablen gesetzt werden.

Man bemerke hier die Modularität und Stabilität von ROS. Denn die Knoten geben keinen Fehler aus, obwohl ein anderer Knoten, der eigentlich für die Ausführung von Nöten ist, noch gar nicht gestartet ist. Sobald dieser bereit ist, fangen die davon abhängigen Knoten automatisch an zu arbeiten.

5.8 Mavros und Simulation

Die Simulation der Drohne erfolgt in Gazebo. Dafür wird ein PX4-Controller simuliert. Gazebo greift die Signale von dem ROS Interface ab und verarbeitet sie. Dafür wird eine FPL-Drohne simuliert. Ihre Daten sendet dann Gazebo wieder an den PX4-Controller. Weiterhin sendet es einen Video-Stream mit den simulierten Bildern. Dieser wird dann von der Node ImagePub erkannt und in ROS gestreamt. Eigentlich kommt er schon über ROS aber auf diesem Wege ist es realitätsgetreuer. (Die Alternative wäre einfach eine andere Quelle für den ImagePub anzugeben.) Auf diesem Wege kann die Drohne nahezu eins zu eins simuliert werden.

Auf diesem Wege wurden bereits viele Fehler wie z. B. Vorzeichenfehler bei den Berechnungen gefunden. Jedoch ist es etwas kompliziert neue Drohnen zu implementieren, da dies erst die Umgebung programmiert werden muss. Unten sieht man einen Kurzfilm, welcher eine Simulation zeigt.
Auch bei der Simulation ist ROS von Vorteil, da man im Testmodus einfach einen Sensor simulieren kann ohne die eigentlichen funktionsverarbeitenden Programme umschreiben zu müssen. Diese Programme laufen dann auf einem externen PC, der sich im gleichen Netzwerk wie die der Pi befindet, und senden regelmäßig "Fake" Nachrichten (auch als Fake News) bekannt. Außerdem kann man so über einen externen PC einfach den Datenfluss kontrollieren, da man quasi die Rohdaten abrufen kann, um die Funktion der einzelnen Programme zu überprüfen. Das ist natürlich auch ein gewisses Sicherheitsrisiko, weshalb die eingehende Kommunikation in den Service Bus beispielsweise durch ein sicheres WiFi-Netzwerk restriktiert werden muss.

Kapitel 6

Bilderkennung

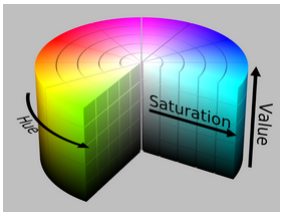

Abbildung 6.1: HSV-Raum

Bei dem Bildverarbeitungsprozess muss das Paket vor verschiedenen Hintergründen, erkannt werden. Dabei darf die Beleuchtung keine Rolle spielen. Die charakteristischen Eigenschaften sind neben Farbe somit ebenfalls Form und Größe.

Zuerst wurde RGB(Red, Green, Blue) Farbraum gefiltert. Jedoch ergaben Test's das der RGB-Farbraum sehr belichtungsempfindlich ist. Deswegen wird das Bild in den HSV(Hue, Saturation, Value) Farbraum konvertiert. Dieser ist deutlich belichtungsstabiler.

Als Farbe wäre Rot sehr geeignet, da es in unsere Umgebung sehr selten vorkommt. Allerdings gibt es keine Roten-Pakete. Außerdem ist Rot für eine Kamera sehr schwer zu erkennen, da es eine der Grundfarben ist und damit in fast jeder Farbe vorkommt. Deswegen wurde als Farbe weiß gewählt. Durch den Filter werden nun weiße Flächen weiß und der Rest wird schwarz. Da das Experiment auf einem dunklen Boden stattfindet, bietet sich dies an.

Weiterhin muss auf dem Paket mittig, ein weiteres schwarzes Viereck sein. Dies ist nötig um eine Orientierung bei naher Distanz zu gewährleisten. Befindet sich die Drohne nahe über dem Paket, so füllt es das komplette Kamerasichtfeld aus. Um sich nun orientieren zu können ist ein weiteres Viereck nötig. Dies muss mittig sein, da sich die Drohne im Verlauf des Anflugs parallel zu dem Rechteck ausrichtet. Damit bei allen Rotationen der Versatz immer gleich ist, muss es mittig sein. Dadurch kann man ihn in der Kameraregelung berücksichtigen.

Für die Form wird der Hought-Transformation verwendet. Bei dieser wird das Bild zuerst abgeleitet. Also werden Pixel neben den sich ein andersfarbiges befindet weiß und der Rest(Weiß neben Weiß, Schwarz neben Schwarz) wird schwarz. Nun sind die

Kanten deutlich zu sehen. Anschließend wird durch jeden weißen Punkt eine Linie mit verschieden Anstiegen zwischen - unendlich und + unendlich gelegt. Schneidet man dabei mit der Linie einen anderen weißen Punkt, so wird sich diese Linie gemerkt. Linien, die durch mehrere weiße Punkte gehen werden, so als Linien detektiert. Der Algorithmus wird aus der OpenCV Bibliothek implementiert. Er hat die Komplexität: n! (n Fakultät).

Anschließend wird das Bild nach Konturen durchsucht. Dabei sind nur die Konturen mit 4 Ecken von Bedeutung. Diese Konturen werden nun nach minimaler und maximaler Größe gefiltert. Die Intervallbreite variiert abhängig von der Flughöhe. Außerdem spielt die Auflösung der Kamera ein Rolle, diese ist 640*480p (vgl. unten)

Von den gefilterten Rechtecken wird nun der Mittelpunkt berechnet. Ist mit einem Umkreis von 10 Pixeln bereits ein weiteres Rechteck gefunden worden, so wird das Rechteck verworfen. Dadurch ist sicher gestellt, dass Rechtecke nicht doppelt erkannt werden. Erfüllt ein Rechteck all diese Bedingungen so wird noch die Rotation der oberen Kante berechnet. Diese wird dann zusammen mit den Mittelpunktkoordinaten an den Controller über ROS weitergeleitet. Es ist das Problem aufgetreten, dass das Viereck um 45° gedreht war. In diesem Fall gibt es 2 oberste kanten und der Algorithmus erkennt schlimmstenfalls abwechselnd beide. Um dies zu vermeiden, wurde eine 2 Regel eingeführt. Es wir immer die oberste Kante genommen. Gibt es 2, so wird die Linke davon genommen. Da es nicht 2 Oberste und Linkeste geben kann, ist der Algorithmus so eindeutig definiert und terminiert.

Um die Rechenzeit zu verringern, wird das Bild auf 640x480 Pixel herunter skaliert, da dies für einfache Vierecke völlig ausreicht. Außerdem wird der Hough Algorithmus bewusst erst nach dem Filtern ausgeführt, da seine Komplexität fakultativ mit der Anzahl der erkannten Objekte ansteigt. Auf diesem Wege ist eine Auswertung mit mehr als 5 Frames pro Sekunde möglich, was unseren Anforderungen genügt.

Der Bildverarbeitungsprozess läuft in der Image Progressing Node. Er wird maximal mit 10 Hz ausgeführt. Das stellt sicher das der Pi nicht überlastet und noch genügend Rechenkapazität für die anderen Codes/Prozesse hat. Da die Drohne nicht zu schnell fliegt, reicht dieses Schrittweite aus um ein stabiles Systemverhalten des PT1-Glied(Regler mit proportionaler Vergrößerung erster Ordnung). Es wurde experimentell in der Gazebosimulation herausgefunden das bei dem PX-4 Controller sogar eine Geschwindigkeit von 3 HZ ausreichend würde, da die Änderungsraten klein sind 0.1. Diese 0.1 steht dafür das, wenn der Regler eine Abweichung von 1 m erkennt, er der Drohne als neue Zielkoordinaten nur Koordinaten gibt, die 0,1 m entfernt sind. Dadurch fliegt sie langsamer und kippt nicht so stark. Durch das kippen wird die Kamera bewegt und ändert somit ihren Winkel auf den Boden. Dadurch wiederum wird das Paket falsch erkannt. (siehe Kapitel Fazit/Ausblick)

Kapitel 7

Gesamtintegration

7.1 Sensordatenverarbeitung

Prinzipiell gibt es 3. Sensoren. Zum einen der Ultraschallsensor, um die Höhe genau und jederzeit bestimmen zu können. Außerdem gibt es einen Kamerasensor, um die Position des Paketes zu bestimmen. Das dritte Sensorsystem ist da, um die Position der Drohne zu bestimmen. Dieses wird vom Institut für Technische und Numerische Mechanik gestellt. Es ist für das fliegen der Drohne existenziell wichtig.

Leider ist ein Fliegen der Drohne ohne dies nicht möglich, weil die Drohne mit globalen Koordinaten arbeitet. Das heißt, dass System bestimmt die Position der Drohne im Raum, anschließend berechnet der Flight Controller die nächste Sollposition und schickt sie an den PX4-Controller. Dieser regelt dann seine Lage dem entsprechend. Der Vorteil an diesem Vorgehen ist, dass die Drohne leicht auf eine externe Positionsbestimmung(z. B. GPS) umgestellt werden kann. Mit dem Ultraschallsensor ist eine genaue Höhenbestimmung immer noch möglich. Der Nachteil ist, dass man die Drohe ohne das System nicht fliegen kann. Die Umstellung auf GPS oder Galileo wäre z. B. ein denkbares Folgeprojekt. Mehr dazu ist unter Auswertung und Fazit zu finden.

Ein weiteres wichtiges Merkmal ist die Einheit der Sensordaten. Zu dem der Ultraschallsensor liefert dabei sein Ergebnis in mm. Näheres dazu findet man unter dem Kapitel “Sensorik, Aktorik und andere Hardware”.

Die Kamera bestimmt die Bildposition in Pixel. Abhängig vom Öffnungswinkel der Kamera ist die Position des Paketes daraus zu berechnen.

Die Formel für die x/y-Koordinate ist:
$x = \frac{x^*}{a^*} * tan(\frac{a}{2}) * h$
$y = \frac{y^*}{b^*} * tan(\frac{b}{2}) * h$
wobei:
a/b* = Bildpunkte in x/y Richtung
x/y* = erkannte Punkte
a/b = Öffungswinkel

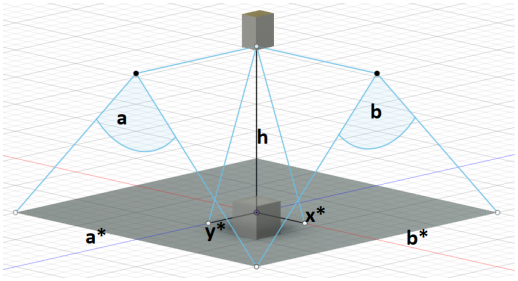

Abbildung 7.1: Kamerawinkel

7.2 Programmablauf

Prinzipiell ist jede Node gleich aufgebaut. Zuerst gibt es eine Initialisierung und Verbindung mit der Masternode. Anschließend deklariert die Node welche Streams die entsprechende Node benötigt und welche nicht. Nun wird gewartet, bis eine neue Nachricht auftaucht und die diese wird dann verarbeitet. Geprüft wird in diesem Fall mit einer Rate von 20 Hz.

Der Flight-Controller verhält sich jedoch leicht anders. Die PX-4 muss erst in den Offboard-Modus versetzt werden. Dieser bedeutet, dass die Drohne extern gesteuert werden kann. Außerdem muss der Arming-Modus aktiviert werden. Damit sie Befehle annimmt. Leider hat das einmalige Senden nicht gereicht. Sie hat nicht zuverlässig beide Befehle erkannt. Damit dies gewährleistet ist, sendet der Pi nun aller 5 Sekunden den Offboard Befehl, solange bis sich der Status der Drohne geändert hat, sodass sie sich nun im Offboardmodus befindet. Anschließend sendet der Controller das Arming Signal, solange bis die Drohne sich im Arming Modus befindet.

Wichtig ist das gleichzeitig mit 20 Hz, eine Sollposition gesendet werden muss, damit die Drohne im nicht wieder aus dem Offboardmodus geht. Dies hat am Anfang große Probleme bereitet, da man so schnell die Kontrolle verliert.

Anschließend wird zuerst eine Sollposition angeflogen. Aus dieser Position muss die Drohne, das Paket sehen. Anschließend richtet sich die Drohne mithilfe der Kamera mittig über dem Paket aus. Dabei vergleicht der Kontroller die aktuelle Position mit der Paketposition. Die Paketposition wird dabei mithilfe der Formel(siehe Sensorerfassung) berechnet. Anschießend wird es mit dem Faktor 10 dividiert, um sie langsam ausrichten zu lassen. Ist das Paket mittig und die Drohne nicht zu schnell(dies wird mithilfe der alten Position und der neuen Position bestimmt) so sinkt die Drohne. Hat sie eine Höhe von 60 cm erreicht, so dreht sie sich und richtet sich entlang des Paketes aus. Dabei verfährt sie nach dem gleichen Verfahren wie bei der Positionsausrichtung und wartet immer, bis das Paket wieder mittig ist. Ist die Verdrehung kleiner als 3 Grad, sinkt sie weiter. Ist sie nur noch 20 cm über dem Boden, so hat sie das Paket gefunden und schließt den Greifer. Anschließend könnte sie wieder abheben und das Paket zur Zielposition bringen. Der Prozess ist in dem unteren Strukturgramm dargestellt.

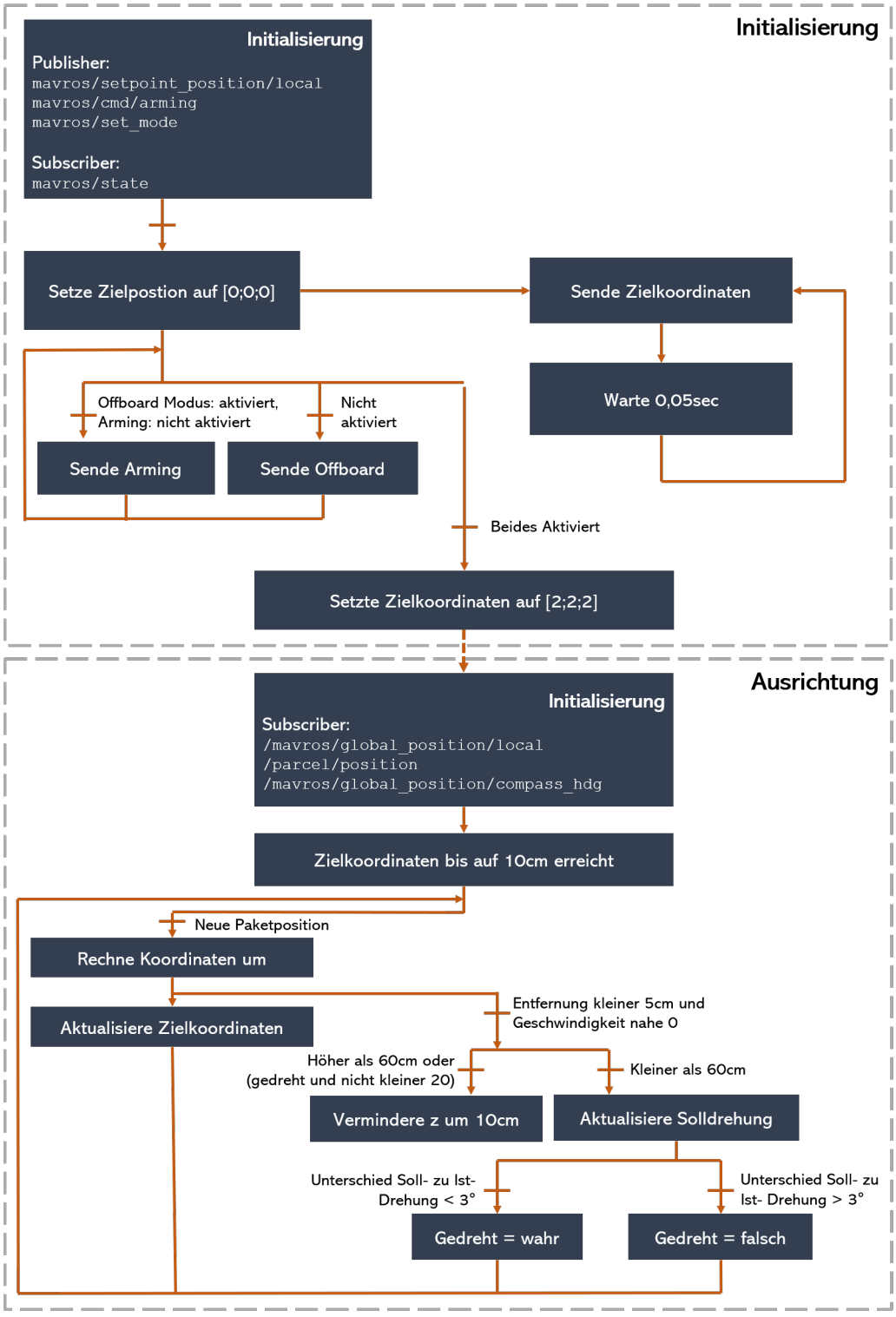

Abbildung 7.2: Funktionsablaufdiagramm

7.3 Fazit/Ausblick

Zusammengefasst ist zu erkennen, dass eine Drohnenregelung durchaus auch mit einfachen Mitteln machbar ist. Besonders überraschend war jedoch, dass die Bilderkennung nicht so gut funktioniert wie gedacht. Es ist sehr schwierig einzelne äußere Faktoren zu Eliminieren und immer zu dem „richtigen“ Ergebnis zu kommen. Jedoch funktioniert das Ansteuern des PX-4 Controllers erstaunlich problemlos. Es hat uns große Schwierigkeiten bereitet die Umgebung zu installieren da diverse Packages manuell nachinstalliert werden müssen. Gerade den Catkin-WS zu erstellen und in diesem die richtigen Pakete zu installieren hat sehr viel Zeit in Anspruch genommen. In den nächsten Wochen wollen wir die Software dann vollständig integrieren und testen. Wenn diese Test's erfolgreich verlaufen so ist die Projektarbeit damit abgeschlossen.

Trotzdem bleiben einige Themen leider offen. Vielleicht für nachfolgende Gruppen. Die größten softwareseitigen Probleme sind zum einen die Bilderkennung, dort wären mögliche Lösungsansätze das Auslagern des Prozesses der Bilderkennung. Dadurch könnte massiv Laufzeit gewonnen werden, weil dieser Anspruchsvolle teil auf einem Externen leistungsfähigeren Rechner gemacht werden könnte. Weiterhin könnte die Bilderkennung dahin verbessert werden das zusätzliche starke Lampen an der Unterseite befestigt werden, um das Paket bei verschiedenen Belichtungen erkennen zu können.
Ein weiterer Punkt ist das Herausrechnen des Drohnenwinkels bei der Bestimmung der Position. Der Vorteil dabei wäre, dass man deutlich schneller fliegen könnte, da die Drohne sich mehr neigen könnte.

Insgesamt könnte noch einiges an Laufzeit gewonnen werden, indem man Prozesse beschleunigt, jedoch sind dafür ausführliche Test notwendig. Am Anfang des Projektes war es angedacht ein GPS-Modul zu integrieren, um die Drohne auch draußen fliegen zu können. Leider ging dies Sicherheitsbedingt nicht. Trotzdem wäre es sinnvoll, da der Einsatz der Paketdrohne erst draußen richtig sinnvoll ist.

Trotz aller dieser Punkte die noch Fehlen, ist der Parcelcopter ein gutes Stück weiterentwickelt worden. Sowohl Hardware technisch mit einem neuen Greifer als auch Softwareseitigen mit einer Bilderkennung und Drohnensteuerung.

Literaturverzeichnis

Martinez, A. & Fernández, E. (2013). *Learning ros for robotics programming.* Packt Publishing Ltd.

Riedel, O. (2019). *Vorlesungsmanuskript it-architekturen in der produktion.* Universität Stuttgart.

Writing a simple publisher and subscriber (python). (2019, Jul). Zugriff auf `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)`

Erklärung

Hiermit versichere ich, dass

- ich die vorliegende Arbeit selbständig verfasst habe,
- ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe,
- ich die eingereichte Arbeit weder vollständig noch in Teilen bereits veröffentlicht habe,
- das elektronische Exemplar mit den anderen Exemplaren übereinstimmt.

Datum	Unterschrift