Smart-Component-Framework
A ROS-based toolkit for modular hardware in flying robot ensembles

Masterarbeit

im Studiengang Ingenieurinformatik von

Martin Schörner

September 2019

Zusammenfassung

Diese Masterarbeit behandelt Konzepte zur Missionsplanung und modularer Hardware in Multicopterensembles. Hierbei wird der Entwurf eines Systems zur Planung und Ausführung von Missionen sowie zur Synchronisation von Ensembles erarbeitet. Zudem wird eine Vorgehensweise zur Integration modularer Sensorik in ein Ensemble von Multicoptern ausgearbeitet. Basierend auf diesen Konzepten erfolgt die Realisierung und Implementierung eines Prototypen, dessen Funktion im Rahmen eines Proof of Concepts gezeigt wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

Multicopter werden zunehmend sowohl im privaten und kommerziellen als auch im wissenschaftlichen Bereich eingesetzt. Mögliche Einsatzzwecke umfassen Inspektionen von Gebäuden (vgl. [50]), das Erstellen von Luftaufnahmen (vgl. [44]) oder das Ausliefern von Paketen (vgl. [1],[21]. Die amerikanische Raumfahrtbehörde NASA plant im Rahmen des New Frontiers Programms zur Erforschung des Sonnensystems durch Raumsonden (vgl. [73]) die Erkundung von Saturns Mond Titan mithilfe eines Multicopters (vgl. [58]). In Verbünden, sogenannten Multicopterensembles, können Copter den Menschen verschiedensten Bereichen unterstützen. Im Katastrophenschutz können solche Ensembles eingesetzt werden, um einen Überblick über die Lage bei großen Waldbränden (vgl. [40]) oder bei Erdbeben und Fluten (vgl. [13]) zu ermöglichen. Außerdem ist es Multicoptern möglich, in Umgebungen vorzudringen, die für Menschen zu gefährlich sind, wie z.B. in brennende Industriehallen (vgl. [66]). Ebenso ist es möglich, die Ausbreitung von schädlichen Stoffen in der Luft zu verfolgen (vgl. [98]) oder Menschen bei der Überwachung großer Gelände unterstützen (vgl. [12], [56]). In der Wissenschaft können Multicopterensembles für geografische Messungen eingesetzt werden (vgl. [100] [55]), oder als Teil eines heterogenen Roboterschwarms Suchaufgaben erfüllen (vgl. [36], [94], [87]). Auch das Aufspannen von Ad Hoc Netzwerken zur Kommunikation mit Geräten in der Luft und am Boden ist möglich (vgl. [96]). Ebenso können Multicopterensembles zu Unterhaltungszwecken eingesetzt werden und beispielsweise Lichtshows am Nachthimmel aufführen (vgl. [53]) oder Zuschauern von Sportveranstaltungen eine Vielzahl von Kamerawinkeln aus verschiedenen Positionen bieten (vgl. [13]).

Diese Projekte sind jedoch alle für nur einen Einsatzzweck ausgelegt und wurden speziell dafür implementiert. Um mehr Flexibilität in Multicopterensembles zu erreichen, werden austauschbare Peripheriegeräte benötigt, die ohne viel Implementierungsaufwand in die Mission eingebunden werden können. Das Forschungsgebiet der modularen Hardware schafft die Grundlage hierfür. Einheitliche Schnittstellen ermöglichen das Zusammenstecken von simplen Robotern in einem Baukasten (vgl. [14]). Die Entwicklung komplexerer Geräte (vgl. [90]) oder kompletter modularer Roboter sind ebenfalls möglich (vgl. [78]).

Die gleichzeitige Steuerung mehrerer Multicopter stellt eine große Herausforderung dar und es konnte sich unter anderem auf Grund der vielseitigen

Einsatzgebiete kein einheitliches System zur Steuerung solcher Ensembles durchsetzen. Aktuelle Missionsplanungssoftware für Multicopter ist meist auf einen konkreten Anwendungsfall zugeschnitten und bietet keine bis schlechte Unterstützung für das Fliegen im Ensemble. Soll zusätzliche Sensorik oder Aktuatorik in einem Flug verwendet werden, so muss meist ein eigenes System entwickelt werden, in welches die Hardware manuell integriert wird. Dabei nutzen die Peripheriegeräte verschiedenste Protokolle, was die Austauschbarkeit der Geräte erschwert. Hier leistet diese Arbeit einen Beitrag und stellt im Folgenden ein Konzept für ein Framework zur missionsbasierten Ausführung von Aufgaben im Ensemble vor. Dieses ergänzt bisherige Systeme um eine größere Flexibilität, welche sich durch einheitliche Schnittstellen und einen modularen Aufbau realisieren lässt. Dafür wird nach der Vorstellung verwandter Arbeiten und einer Anforderungsanalyse, auf die Grundlagen der Steuerung von Multicoperensembles und modularer Hardware eingegangen. Außerdem wird der Aufbau und die Implementierung des erarbeiteten Konzepts vorgestellt sowie ein Proof of Concept durchgeführt. Abschließend werden wichtige Erkenntnisse und eine Zusammenfassung der Ergebnisse in einem Fazit herausgearbeitet.

2 Verwandte Arbeiten

In den folgenden Abschnitten wird auf relevante verwandte Arbeiten zur behandelten Thematik eingegangen. Dabei werden zunächst Ansätze zur Planung und Ausführung von Missionen von Robotersystemen betrachtet. Hier werden zu Beginn Bodenkontrollstationen zur Überwachung und Steuerung von Multicoptern behandelt. Anschließend wird ein Überblick über die von der Roboter Middleware ROS bereitgestellten Möglichkeiten zur Definition von Abläufen in der Robotik gegeben. Zuletzt werden Arbeiten vorgestellt, die sich mit modularer und rekonfigurierbarer Hardware beschäftigen.

2.1 Missionssteuerung

Die Planung, Durchführung und Überwachung von Aufgaben im Roboterensemble sind kritische Bestandteile für die Durchführung einer erfolgreichen Mission. Nachfolgend wird zunächst ein Überblick über Arbeiten, die sich mit der Ausführung von Missionen für Multicopter und Multicopterensembles beschäftigen gegeben. Dabei werden sowohl klassische Bodenkontrollstationen für Multicopterensembles, als auch Projekte aus der Robotik und anderen Fachgebieten betrachtet. Anschließend wird ein Überblick über verschiedene Möglichkeiten zur Definition von Abläufen für Roboteranwendungen in ROS gegeben.

2.1.1 Bodenkontrollstationen für Multicopter

Die Missionsplanung von Multicoptern erfolgt in der Regel über eine Bodenkontrollsoftware, welche auf einem externen Laptop ausgeführt wird. Der Copter ist dabei entweder per Funk oder über ein USB-Kabel mit der Bodenkontrollstation verbunden. In einer grafischen Benutzeroberfläche können Missionen über das Setzen von Wegpunkten definiert und Einstellungen am Copter verändert werden (vgl. Abbildung 2.1). Außerdem ist die Überwachung des Copterstatus während des Flugs möglich. Diese Softwares werden oftmals von den Herstellern von Flugcontrollern für Copter entwickelt und sind deshalb meist nur für Copter mit Controllern eines bestimmten Herstellers geeignet. Beispiele für solche Bodenkontrollsoftwares sind der APM Mission Planner (vgl. [5]) und QGroundControl (vgl. [74]) für die Pixhawk

Flugcontroller (vgl. [45]), die DJI GCS Pro (vgl. [43]) für Copter des Herstellers DJI (vgl. [46]) oder die Bodenkontrollstation des Paparazzi Flugcontrollers (vgl. [11]). Eine Ausnahme bildet die Software UgCS (vgl. [92]), welche die Flugcontroller mehrerer Hersteller unterstützt.

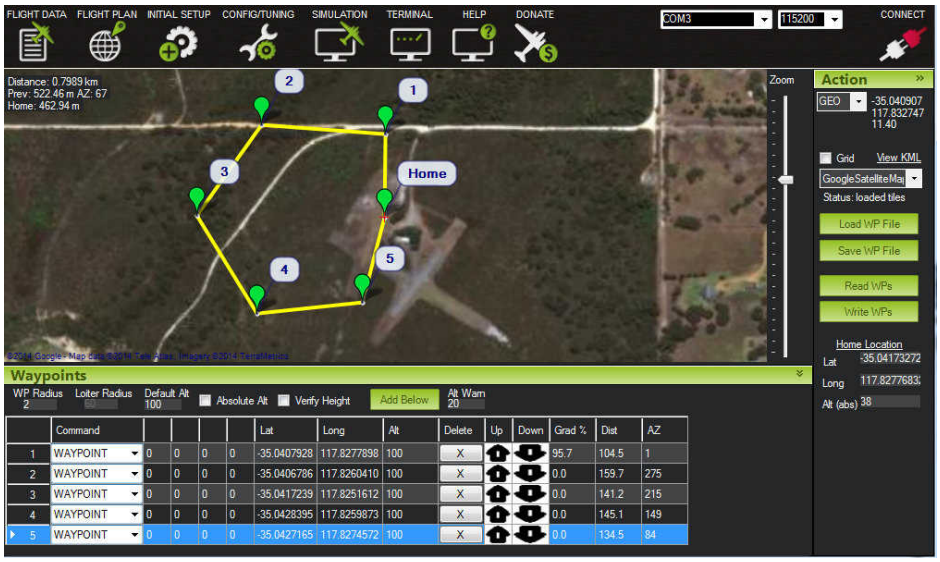

Abbildung 2.1: Missionsplanung des APM Mission Planner (Bildquelle: [6])

Die im vorhergehenden Abschnitt vorgestellten Tools sind alle für die Nutzung mit einem oder einigen wenigen Coptern ausgelegt. Das Definieren von Missionen aus der Sicht eines Ensembles ist nicht möglich. Die Arbeit von Dousse u. a. (vgl. [37]) erweitert die Software QGroundControl über ein Plugin um ein übersichtliches Benutzerinterface zur Steuerung von mehr als 15 Quadcoptern. Dadurch wird das Gruppieren von Coptern durch Selektion in der GUI ermöglicht, was das Ausführen von Befehlen auf mehreren Coptern gleichzeitig ermöglicht. Die Karlsruhe Generic Agile Ground Station (vgl. [56]) des Fraunhofer Instituts für Informations- und Datenverarbeitung ist eine Bodenkontrollstation für verschiedene Roboter- und Sensorplattformen. Sie ermöglicht den zwei Piloten das Überwachen von unzugänglichen und weitläufigen Arealen. Dabei übernimmt einer der Piloten die Steuerung des mobilen Roboters, während der andere die von dem Roboter übermittelten Sensordaten auswertet. Das Intel Drone 100 Projekt (vgl. [53]) nutzt mit RGB LEDs ausgestattete Quadcopter, um Lichtshows am Nachthimmel aufzuführen. Nach den ersten erfolgreichen Aufführungen mit 100 Coptern werden inzwischen Shows mit über 1000 Geräten geflogen. Um dieses Ziel zu erreichen, wurde ein Copter entwickelt, der durch sein geringes Gewicht und Schutzvorrichtungen um die Rotoren die Gefahr für das Publikum reduzieren soll (vgl. [95]). Außerdem wurde eine spezielle Software für das Design

und die Ausführung solcher Lichtshows entwickelt. Mit dieser lässt sich eine Choreografie für die Copter erstellen und der Status vor und während des Flugs überwachen. Das Swarmix Projekt (vgl. [87]) beschäftigt sich mit dem Einsatz heterogener Schwärme in Search and Rescue-Szenarien. In dem Fallbeispiel *Finding Linda* (vgl. [94]) wird ein Szenario mit einer vermissten Person, die gefunden werden muss, durchgespielt. Dabei sucht ein Schwarm aus Menschen, Quadcoptern, Starrflüglern und mit zusätzlicher Sensorik ausgestatteten Suchhunden nach der vermissten Person. Die Teilnehmer des Schwarms übernehmen bei der Suche verschiedene Aufgaben. Die mit Kameras ausgestatteten Starrflügler sind für das grobe Absuchen des Areals verantwortlich und identifizieren relevante Punkte für das Bodenpersonal. Diese Punkte werden anschließend von Menschen überprüft, die die Position der Punkte über eine Smartphone-App einsehen können. Alternativ können weit entfernte Punkte auch von Suchhunden überprüft werden. Diese sind darauf trainiert, einem Quadrocopter zu folgen, der den relevanten Punkt autonom anfliegt.

Die hier vorgestellten Systeme bieten neben dem Planen und Ausführen des Flugablaufs meist wenig zusätzliche Funktionalität. Programme, die neben der reinen Flugplanung noch weitere Funktionen mitbringen, sind meist für genau einen Use-Case ausgelegt und bieten keine Möglichkeit zur Erweiterung des Funktionsumfangs.

2.1.2 Robot Operating System (ROS)

ROS (vgl. [79]) ist eine Middleware für Roboteranwendungen. Das Projekt stellt eine nachrichtenbasierte Kommunikationsmöglichkeit bereit, über die die modular strukturierten Programmteile (Nodes) miteinander kommunizieren können. Neben der Kommunikationsinfrastruktur bietet ROS zusätzlich Tools für das Erstellen, Debuggen und Starten von Anwendungen sowie zur Visualisierung des Zustands des Systems. Durch den generischen Aufbau kann das System auch für das Erstellen und Ausführen von Missionen für Multicopterensembles genutzt werden. In ROS existieren neben der klassischen starren Programmierung von Abläufen in einem Programm noch weitere, flexiblere Methoden, die denselben Zweck erfüllen. Hierzu zählt das Starten mehrerer Nodes über Launch files, die Definition von Aufgaben (Actions) über das Modul Actionlib sowie das Erstellen von Abläufen über die Erweiterung smach, welche alle in den nachfolgenden Abschnitten behandelt werden.

ROS Launch-Files

Launch-Files bieten in ROS die Möglichkeit, mehrere Nodes parallel und parametrisiert zu starten. Dabei gibt es Unterschiede zwischen den Launch-Files in ROS1 und ROS2. Launch-Files in ROS1 sind in XML definiert und stellen eine Auflistung der zu startenden Nodes mit ihren Parametern dar (vgl. [32]). Zusätzlich können auch andere Launch-Files eingebunden werden. Dabei werden alle Nodes gleichzeitig gestartet und es kann keine zeitliche Reihenfolge festgelegt werden. Die Steuerung des Programmflusses oder eine Reaktion auf Ereignisse sind nicht möglich. In ROS2 sind Launch files über Python-Skripte gelöst (vgl. [83]). Die Nodes in ROS2 haben einen Lebenszyklus, der den aktuellen Zustand des Nodes beschreibt (vgl. [23]) Bei einer Änderung des Zustands wird ein Event emittiert, auf das im Launch-File reagiert werden kann. Das manuelle Emittieren eigener Events sowie die Reaktion auf diese sind ebenfalls möglich.

Actionlib

Der Actionlib-stack für ROS ermöglicht Nodes, die Ausführung konkreter Aufgaben (Actions) anzubieten oder die angebotenen Actions anderer Nodes zu nutzen (vgl. [82]). Der ActionServer bietet eine Aktion (z.B. das Schließen eines Greifers) an, die ein anderer Node (ActionClient) nutzen kann. Dabei können beim Aufruf der Aktion Parameter übergeben werden. Nach Ende der Action liefert der Server ein Ergebnis zurück. Der Unterschied zu den vom Konzept her ähnlichen ROS-Services (vgl. [80]) ist die Möglichkeit den Status der Action über ein kontinuierliches Feedback zu überwachen. Actions sind im Vergleich zu Services für längere Aufgaben gedacht, weswegen sie zusätzlich die Option zum Abbrechen der Action bieten.

smach

Das Framework smach (vgl. [27]) für ROS erlaubt die Erstellung und Koordination von Fähigkeiten von Robotern zu vollwertigen Roboterapplikationen. Das System basiert auf der Erstellung von hierarchischen State Machines aus vorgegebenen Fähigkeiten. Nach der Erstellung des Programms ermöglicht das Framework eine Einsicht in den Programmablauf zur Laufzeit. Durch die Nutzung bestehender Komponenten, die lediglich miteinander verschaltet werden, wird ein schnelles Erstellen von Prototypen ermöglicht.

2.1.3 Weitere Systeme zur Modellierung von Abläufen

Neben der Robotik wird auch in anderen Feldern wie der Spiele-Entwicklung oder anderen wissenschaftlichen Bereichen grafische Programmierung

zur Modellierung und Programmierung von Abläufen und Prozessen genutzt. Exemplarisch für die Vielzahl dieser Anwendungen werden nachfolgend Blueprints aus der Unreal 4 Engine und das Tool Simulink der Firma MathWorks vorgestellt.

Blueprints (vgl. [54]) sind eine grafische Scriptsprache in der Spiele-Engine Unreal 4. Sie sind als Alternative zur klassischen Programmierung durch das Schreiben von Code gedacht. Zum Erstellen eines Blueprints werden Knoten in diesen eingefügt, welche wiederum Schnittstellen enthalten. Diese Schnittstellen ermöglichen das Verbinden der Knoten untereinander und damit die Erstellung eines Skripts. Die Schnittstellen sind typisiert und können beispielsweise für das Triggern von Events oder das Weiterreichen von Variablenwerten genutzt werden. Zur besseren Strukturierung von komplexeren Blueprints ist die Gruppierung von mehreren Knoten zu einem neuen Knoten möglich.

Das Tool Simulink der Firma MathWorks (vgl. [47]) ermöglicht die Simulation und das modellbasierte Design verschiedenster Systeme. Als Einsatzgebiete werden unter anderem Signalverarbeitung, Regelungstechnik und Fahrassistenzsysteme angegeben. Das Tool bietet die Möglichkeit, ein System - ähnlich wie bei den Blueprints der Unreal Engine - grafisch durch Verschalten einzelner Bausteine zu modellieren. Eine Steigerung der Übersichtlichkeit ist auch hier über die Gruppierung von Elementen zu neuen Bausteinen möglich. In Simulink wird eine Vielzahl von Bausteinen für verschiedene Domänen vom Hersteller bereitgestellt. Ein weiteres Feature ist die Generierung von Code basierend auf dem Modell. Dieser kann direkt auf das Zielsystem, für das entwickelt wird, aufgespielt werden.

2.2 Modulare Hardware

Modulare Hardware ermöglicht durch die Definition von einheitlichen Schnittstellen die Rekonfiguration eines Systems zur Laufzeit. Die ersten Grundlagen für modulare Hardware wurden von IBM mit der Spezifikation von Plug and Play geschaffen (vgl. [49]). Über das zur Steigerung der Usability gedachte Konzept können an Computer angesteckte Peripheriegeräte genutzt werden, ohne vorher Treiber zu installieren oder den Rechner neu zu starten. Cubelets (vgl. [14]) sind kleine Hardwarebausteine, die einen konkreten Zweck erfüllen. Sie lassen sich in Action- und Sense-Cubelets einteilen, die jeweils ihre Umgebung manipulieren oder wahrnehmen können. Zusätzlich existieren Think-Cubelets, die Werte von Sense-Cubelets verändern können. Die Bausteine lassen sich über ein magnetisches Verbindungssystem zu einfachen Robotern verbinden. Über die Steckverbindung wird neben der mechanischen auch eine elektrische Verbindung zum Übertragen von Strom und Daten an

die benachbarten Module hergestellt. Die Art und Weise des Zusammensteckens der Bausteine beeinflusst dabei die Funktion des Roboters. In der Umsetzung ähnlich sind die M-Blocks (vgl. [59]). Diese ebenfalls würfelförmigen Bausteine lassen sich wie auch die Cubelets über eine magnetische Verbindung mit anderen Blöcken zusammenschließen. Die Besonderheit der M-Blocks ist deren Fähigkeit, sich selbst physikalisch zu rekonfigurieren. Ein abrupt abbremsender Motor im Inneren erzeugt ein Drehmoment, welches den M-Block in Rotation versetzt, wodurch dieser sich einen oder mehrere Blöcke weiter bewegt (vgl. [75]).

Das H-ROS Projekt (vgl. [78]) ist ein Versuch, modulare Hardware in der Robotik umzusetzen. Das Kernkonzept ist die Erweiterung existierender Hardware um ein H-ROS SoM (System on Module, vgl. [62]), um das Gerät damit mit einer einheitlichen Schnittstelle für die Nutzung in H-ROS zu versehen (vgl. [77]. Das SoM kümmert sich um die Ansteuerung des zu integrierenden Gerätes. Gerätespezifische Treiber, die für dessen Nutzung erforderlich sind, werden auf dem SoM installiert. Zusätzlich bietet das SoM einen Ethernet Port, mit dem es mit anderen Modulen kommunizieren kann. Grundlage für die Kommunikation bildet TSN (vgl. [91]) und ROS2 (vgl. [35]). Das Projekt scheint jedoch Probleme mit der Finanzierung zu haben (vgl. [77]) und die offizielle Webseite ist zur Zeit (Stand 6.9.2019) nicht erreichbar.

Das Unternehmen Tinkerforge (vgl. [90]) bietet eine Reihe von Modulen für das Erstellen von elektronischen Aufbauten an. Die Module werden dabei in Bricks und Bricklets unterteilt. Bricklets sind einfache elektrische Geräte wie simple Sensoren, Funkmodule oder Relays. Bricks sind komplexere Module die entweder mehr Rechenleistung (beispielsweise eine inertiale Messeinheit (IMU)) oder Strom (beispielsweise ein Schrittmotortreiber) benötigen. Angesteuert werden alle von einem über USB programmierbaren Masterbrick, auf dem ein frei programmierbarer Mikrocontroller die Verarbeitung der Daten und Ansteuerung der Aktoren übernimmt. Die Bricklets und Bricks können über Hardwareschnittstellen nahezu beliebig miteinander verbunden werden. Ein proprietäres Protokoll sorgt für die Kommunikation der Module mit dem Masterbrick.

Das Problem der hier vorgestellten Systeme ist die Proprietarität der eingesetzten Hardware. Mit den in den vom Hersteller angebotenen Cubelets lassen sich zwar einfache Roboter realisieren, mit der beschränkten Menge verfügbarer Bausteine sind den Einsatzmöglichkeiten jedoch Grenzen gesetzt. Bei dem System von Tinkerforge sind zwar mehr Module verfügbar und die Möglichkeit zur Kommunikation mit anderer Hardware ist über Schnittstellen wie CAN oder RS232 möglich, die Problematik der fehlenden Erweiterbarkeit bleibt aber nach wie vor durch das proprietäre Kommunikationsprotokoll bestehen. Die Komplexität des H-ROS SoM und der damit vermutlich ein-

hergehende hohe Preis machen das System unattraktiv für die Integration günstiger und simpler Sensoren. Zudem ist aktuell (Stand 6.9.2019) unklar, ob H-ROS und das SoM überhaupt als Produkt auf dem Markt erhältlich sein wird.

3 Anforderungsanalyse und Aufgabenstellung

Das folgende Kapitel behandelt die Durchführung einer Anforderungsanalyse, bei der zunächst eine im Rahmen des ScaleX 2015 Projekts vom Lehrstuhl für Softwaretechnik der Universität Augsburg durchgeführte Messkampagne vorgestellt wird und anschließend die dabei aufgetretenen Probleme aufgezeigt werden. Darauf basierend werden Anforderungen an das zu entwickelnde System festgehalten und damit die Aufgabenstellung spezifiziert.

3.1 Fallstudie ScaleX 2015

Im Rahmen einer Kooperation zwischen verschiedenen Universitäten wurde 2015 in Fendt (Bayern) ein Feldexperiment über mehrere Wochen durchgeführt. Das Ziel des Experiments war die Verifikation von klimatologischen Modellen mit möglichst vielen echten Sensorwerten in einem abgesteckten Areal. Das Ergebnis sowie die Kampagne selbst sind unter dem Namen ScaleX in einer abschließenden Publikation (vgl. [100]) festgehalten. Der Beitrag des Instituts für Software und Systems Engineering der Universität Augsburg umfasst die Durchführung von Messungen mit sogenannten virtuellen Messtürmen. Bei der Kampagne wurden die Messgrößen Temperatur und Luftfeuchtigkeit durch Quadcopter in verschiedenen Höhen gemessen. Dabei flogen die Copter in einer Formation, die einem virtuellen Messturm entsprach. Bei einem konventionellen Messturm (vgl. Abbildung 3.1) sind Sensoren auf verschiedenen Höhen des Turms fest angebracht (vgl. [3]). Um einen virtuellen Messturm zu bilden, fliegt ein Copter von seiner Startposition senkrecht nach oben und zeichnet dabei Höhe und Sensordaten auf. Als Quadrocopterplattformen kamen Forschungsdrohnen des Typs *Saphira* der Firma rOsewhite Multicopters zum Einsatz (vgl. [99]), welche mit einem Autoquad M4 Flugcontroller ausgestattet sind. Bei der ScaleX 2015 Kampagne flogen drei dieser Copter parallel einmal pro Stunde über einen Zeitraum von 24 Stunden. Dabei stiegen die Copter auf eine Höhe von 100 Metern und landeten anschließend wieder. Bei den Flügen befanden sich SHT-75 Sensoren zum Messen von Temperatur und Luftfeuchtigkeit (vgl. [20]) an den Coptern. Diese waren über Mikrocontroller-Boards vom Typ Arudino Nano (vgl. [9]) mit dem ebenfalls am Copter angebrachten Odroid-XU4 Ein-

platinenrechner (vgl. [63]) verbunden, auf dem ein Skript zum Aufzeichnen der Messdaten ausgeführt wurde. Jeder Copter erhielt eine Mission von seinem Bordcomputer und wurde zu Beginn der Messung manuell von einem Piloten mittels Funkfernbedienung gestartet. Anschließend wurde über ein Funkgerät ein Sprachsignal gegeben, worauf alle Piloten die Mission an der Fernbedienung starteten. Daraufhin flogen die Copter ihre Mission ab und wurden anschließend wieder manuell von den Piloten gelandet.

Abbildung 3.1: Messturm der Universität Dresden (Bildquelle: [3])

Probleme von ScaleX 2015

Der im vorherigen Abschnitt beschriebene Aufbau hat einige Mängel, die bei der Messkampagne zu ungültigen Messungen und unerwünschtem Verhalten führten oder die Ausführung der Mission unnötig verkomplizierten. So musste beispielsweise ein Pilot pro Copter eingesetzt werden, um das manuelle Starten und Landen des Copters zu übernehmen. Das führt zu einem hohen Personalbedarf beim Einsatz mehrerer Copter. Durch defekte Sensoren waren einige Messungen unbrauchbar, da der Defekt erst bei der Auswertung des Fluges bemerkt wurde. Außerdem war das Tauschen von Sensoren wegen der Befestigung der Sensoren über Kabelbinder umständlich. Nach dem Flug erfolgte ein manueller Transfer der Messwerte von den Coptern. Die anschließende Korrektur der Sensorwerte mit sensorspezifischen Korrekturkurven konnte nicht durchgeführt werden, da über den Zeitraum der Messungen Sensoren und ganze Copter aufgrund von Defekten ausgetauscht werden mussten. Dadurch war eine Zuordnung von Sensorwerten zu einem konkreten Sensor und dessen Korrekturkurve nicht mehr möglich. Ein weiterer Mangel war die ab dem Start der Mission nicht mehr vorhandene

Synchronisation der Copter untereinander. Jeder Copter flog seine Mission ab, ohne sich mit den anderen auszutauschen. Durch Umwelteinflüsse und variierende Akkuleistung trat der Fall auf, dass einzelne Copter die Mission früher oder später beendeten als andere. Ein ebenfalls nicht zu unterschätzendes Problem war die fehlende Möglichkeit zur Überwachung der Mission. Statt Telemetriedaten stand nur ein LiPo-Warner (vgl. Abbildung 3.2) zur Verfügung. Dieses Gerät wird direkt mit dem Akku des Copters verbunden und gibt bei Unterschreiten einer einstellbaren Zellspannung ein akustisches Signal von sich, um einen leeren Akku zu signalisieren. Trotz der hohen Lautstärke des Signals kann es jedoch abhängig von den Windbedingungen selbst bei verhältnismäßig geringen Entfernungen nicht mehr wahrnehmbar sein. Das Fehlen von Telemetriedaten führte zu einer Situation, in der ein Copter eine Mission nicht korrekt ausführte und auf eine Höhe von ca. 200 Metern stieg. Da am Boden keine Möglichkeit bestand, die Höhe des Copters anzuzeigen, blieb das Verhalten zunächst unbemerkt. Auch das Piepsen des LiPo-Warners war aufgrund der großen Höhe nicht vernehmbar. Letztendlich fiel die Spannung des Akkus weit genug ab, um die Notabschaltung des Flugcontrollers auszulösen, welche alle Rotoren des Copters deaktivierte und zu einem Absturz aus ca. 200 Metern Höhe führte.

Abbildung 3.2: *LiPo Warner* zur Erkennung eines niedrigen Akkustands (Bildquelle: [67])

3.2 Aufgabenstellung

Aufgabe der Arbeit ist die Erstellung eines Systems, welches es ermöglicht, Missionen für Roboterensembles flexibel planen und ausführen zu können. Ebenso soll ein Konzept für die Integration modularer Hardware in das System entwickelt werden. Eine Schnittstelle soll ein einfaches Austauschen von

Sensorik ermöglichen. Mit dem System soll es möglich sein, die erstellten Missionen mit minimalem Personalaufwand auszuführen. Im Rahmen der Arbeit soll eine Möglichkeit zur Steuerung von Multicoptern in dieses System integriert werden. Diese müssen eine sequenzielle Abfolge von Punkten im Raum abfliegen und dabei auf Fehlerfälle reagieren können. Ebenso soll eine Strategie zum flexiblen Loggen von Messwerten und Telemetriedaten und eine Synchronisationsmöglichkeit für die Teilnehmer des Ensembles in das System integriert werden. Zudem muss es möglich sein, Diagnosedaten des Flugs live anzuzeigen und damit dem Bodenpersonal das frühzeitige Erkennen fehlerhafter Messungen ermöglichen.

4 Grundlagen

In diesem Kapitel wird der grundlegende Aufbau eines Multicopters erklärt. Dabei wird insbesondere auf die Steuerungs- und Missionsplanungssoftwares verschiedener Hersteller eingegangen. Anschließend werden Systeme zur Steuerung mehrerer Multicopter sowie Robotersteuerungssysteme genauer betrachtet. Im letzten Teil des Kapitels werden die dynamischen Aspekte modularer Hardware erörtert.

4.1 Aufbau eines Multicopters

Multicopter (kurz Copter) sind mehrrotorige Fluggeräte und bilden eine Untergruppe der Unmanned Aerial Systems (UAS, vgl. [103]). Die meist durch einen Lithium-Polymer Akku angetriebenen Geräte verfügen über mehrere an Auslegern (ugs. Arme) angebrachte Motoren an denen nach unten wirkende Luftschrauben (Propeller) angebracht sind (vgl. [48]). Das Gewicht herkömmlicher Copter variiert je nach Einsatzzweck zwischen einigen Gramm bis zu einigen wenigen Kilogramm (vgl. [41]). Sie werden von einer Funkfernbedienung gesteuert oder fliegen autonom. Dabei übernimmt ein Flugcontroller die Verarbeitung der Signale der Fernbedienung bzw. das Anfliegen von Wegpunkten. Zusätzlich regelt er durch Ansteuerung der Motoren Position und Lage des Copters in der Luft. Dafür greift er auf Sensoren zur Lage- und Positionsbestimmung zurück. Typische Sensoren für die Positionsbestimmung sind GPS-Module, Barometer und externe Trackingsysteme (vgl. [57]). Zur Lagebestimmung wird eine IMU (Inertial Measuring Unit) genutzt, welche die Werte eines Gyroskops, eines Magnetometers und eines Beschleunigungssensors zu einer Orientierung fusioniert. Durch die Regelung von Lage und Position ist es Multicoptern möglich, beliebige Punkte im dreidimensionalen Raum anzufliegen und an diesen zu verweilen. Im Forschungsbereich werden zusätzlich zum Flugcontroller oft Bordcomputer verbaut, welche dem Flugcontroller Sollpositionen mitteilen oder ihn anderweitig steuern. Diese Steuerbefehle werden aufgrund von an den Bordcoputer angeschlossenen Sensoren oder Kommunikationsgeräten abgeleitet, wodurch sensorbasierte Flüge oder Schwarmverhalten realisiert werden können (vgl. [61]). Die Anzahl der Propeller eines Copters ist entscheidend für die Namensgebung. Ein Multicopter mit drei, vier, sechs bzw. acht Luftschrauben wird Tri-, Quad-, Hexa-

oder Octocopter genannt, wobei die Quadcopter die am weitesten verbreitete Variante der Multicopter darstellen.

4.2 Steuerungssoftware für Multicopter

Die Steuerungssoftware für Multicopter wird meist vom Hersteller des Flugcontrollers bereitgestellt und ist daher an diesen gebunden (vgl. 2.1.1). Ausnahmen bilden Software wie UgCS (vgl. [92]), welche Systeme verschiedener Hersteller unterstützen. Der Aufbau und die Funktionsweise der Programme ist weitestgehend ähnlich. Über eine grafische Benutzeroberfläche verbindet sich der Nutzer entweder über Kabel oder Funk mit dem Flugcontroller und hat danach die Möglichkeit, diesen zu konfigurieren. Dabei können Parameter eingestellt werden, welche Regelung, Steuerung durch die Fernbedienung oder andere Eigenschaften des Copters verändern. Außerdem können Missionen geplant und an den Flugcontroller gesendet werden. Dies geschieht in der Regel durch die Definition von Wegpunkten auf einer Karte, welche in einer konfigurierbaren Reihenfolge abgeflogen werden können (vgl. Abbildung 2.1). Einige Systeme wie UgCS unterstützen das Erstellen von Wegpunkten durch Algorithmen, welche Wegpunktlisten zum Abfliegen eines definierbaren Areals automatisch generieren. An den Wegpunkten können vordefinierte Aktionen wie das Aufnehmen von Bildern, das Ansteuern eines Servos, das Verweilen des Copters an diesem Punkt oder das Kreisen um den Punkt ausgeführt werden. Diese Aktionen sind durch die Software vorgegeben und nur schwer durch den Nutzer erweiterbar. Außerdem ist die Oberfläche der Software primär für die Nutzung eines einzelnen oder von wenigen Coptern ausgelegt (vgl. [37]). Aus diesen Gründen wird im kommerziellen oder im Forschungsumfeld oft auf einen zusätzlichen, frei programmierbaren Bordcomputer und eine selbst entwickelte Bodenkontrollstation zurückgegriffen, um komplexere Missionen zu realisieren. Diese Software ist ebenfalls auf den jeweiligen Anwendungsfall wie z.B. die Überwachung Gebiets aus der Luft (vgl. [56], [12]) oder Search and Rescue Szenarien (vgl. [87]) zugeschnitten.

4.3 Multi-Quadcopter-Systeme

Multi-Quadcopter-Systeme bestehen aus mehreren Quadcoptern, die mit einer Bodenkontrollstation kommunizieren, welche von einem Bedienpersonal bedient wird (vgl. [12]). Dabei sendet die Bodenkontrollstation Missionsdaten und Steuerbefehle an die einzelnen Teilnehmer des Ensembles, die umgekehrt Statusnachrichten und Sensorwerte an die Bodenkontrollstation zurück senden. Das Bedienpersonal steuert und überwacht dabei die Ausführung der

Mission. Die Steuerung der Copter kann dabei über beliebige Funktechnologien wie WLAN, 433 oder 915MHz Telemetriemodule (vgl. [10]) oder GSM erfolgen. Die Reichweite der Funksignale wird oft durch Nutzen anderer Copter als Relay-Stationen für Signale erweitert. Dadurch kann auch mit Coptern, bei denen keine direkte Funkverbindung zur Bodenkontrollstation besteht, kommuniziert werden (vgl. [87]). Solche Systeme müssen sich nicht nur homogen aus Quadcoptern zusammensetzen. Auch heterogene Ensembles aus einer Kombination von Coptern und anderen Robotern sind möglich (vgl. [87], [36]). Dorigo u. a. zeigen im Swarmanoid Projekt einen heterogenes System aus Multicoptern, Bodenrobotern und Greifern zum Auffinden und Manipulieren von Objekten (vgl. [36]) Dabei übernehmen mit einer Kamera ausgestattete Quadcopter die Lokalisierung des Objekts. Ist dessen Position bekannt, transportieren zwei Bodenroboter einen Greifer zu dem Objekt, welcher dieses aufnehmen kann.

Die Kommunikation im Ensemble ist für die Synchronisation und Koordination der Teilnehmer untereinander und mit der Kontrollstation von großer Relevanz. Durch die Nutzung von Multiagentenframeworks lassen sich komplexe Aufgaben in einem Roboterschwarm koordinieren und ausführen (vgl. [55]) Eine einfache Möglichkeit zur Synchronisation der Ensembleteilnehmer untereinander, die bereits viele Anwendungsfälle abdeckt, ist die Nutzung von Barrieren (vgl. [101]). Bei diesem aus der parallelen Programmierung stammenden Konzept halten alle beteiligten Threads eines Programms an einer bestimmten Stelle der Programmausführung so lange an, bis der letzte Thread diese Stelle ebenfalls erreicht hat. Erst danach wird die Programmausführung aller Threads fortgesetzt. Diese Methodik lässt sich in Roboterensembles anwenden, um beispielsweise einen Roboter an einem bestimmten Punkt im Raum warten zu lassen, bis alle anderen diesen Punkt ebenfalls erreicht haben. Durch diese Vorgehensweise erfolgt eine einfache Synchronisierung der Roboter untereinander.

4.4 Roboter Middleware

Eine Middleware bietet eine Möglichkeit für den Datenaustausch zwischen Programmen (vgl. [60]). Middleware für Robotersysteme besteht neben einer Kommunikationsinfrastruktur oft auch aus Tooling, einer Visualisierungsumgebung und Bibliotheken zur Erstellung von Programmen für Roboter. Nachfolgend werden die Bestandteile dieser Frameworks genauer beschrieben und ein Überblick über existierende Projekte gegeben. Den Kern einer Roboter-Middleware bildet ein System zur Kommunikation einzelner Programmteile miteinander. Häufig wird ein nachrichtenbasierter Ansatz mit einer Publish / Subscribe Architektur genutzt. Dabei können Teile des Programms Nachrich-

ten in Kanälen (Topics, vgl. [34]) veröffentlichen (publish) und/oder Nachrichten anderer Programmteile abonnieren (subscribe). Topics haben einen definierten Datentyp, der die Struktur der Nachricht beschreibt. Durch diese nachrichtenbasierte Kommunikation können Datenströme wie Sensorwerte und Sollwertvorgaben versendet werden. Ein weiteres in vielen Frameworks vorhandenes Konzept ist das Anbieten und Nutzen von Services (vgl. [80]). Dabei definiert ein Programmteil einen Service, der von anderer Stelle aus genutzt werden kann (Service Call). Dadurch lassen sich Aktionen wie z.B. das Anfliegen eines Punktes oder das Schließen eines Greifers ausführen. Um das Kommunikationssystem und andere Bestandteile der Middleware zu nutzen, können Entwickler vom Herausgeber des Systems bereitgestellte Bibliotheken einbinden. Diese umfassen meist weitere Funktionalitäten wie die Transformation von Koordinatensystemen oder die Trajekorienberechnung. Ebenso werden Hardwaretreiber für Geräte wie Roboterarme oder Greifer zur Verfügung gestellt. Um Programme zu erstellen und zu debuggen, wird spezielles Tooling bereitgestellt, welches das Compilieren und Ausführen von Programmen, sowie das Überwachen und Manipulieren des Kommunikationsframeworks ermöglicht. Dabei helfen mitgelieferte Visualisierungstools wie RViz für ROS, welches unter anderem eine 3-dimensionale Darstellung von Sensorwerten und Roboterpositionen ermöglicht (vgl. [26]). Das in diesem Bereich primär genutzte Framework ist ROS in verschiedenen verfügbar. Es existieren aber auch andere, unabhängige Projekte, wie z.B. YARP (Yet Another Robot Platform, vgl. [102]). Die quelloffene Plattform bietet eine Peer-to-Peer Kominikationsinfrastruktur (Adaptiive Communication Environment, ACE) die verschiedenste Übertragungsschichten unterstützt. Zudem sind Bibliotheken für Signalverarbeitung und Treiber für verschiedene Endgeräte verfügbar.

Das Robot Operating System (ROS) ist eine quelloffene Sammlung von Software zum erstellen von Roboteranwendungen. Darin enthalten sind neben einem Kommunikationsframework auch Treiber für Hardware, eine Sammlung von Tools zum Debuggen und Testen von Programmen sowie Algorithmen zur Datenverarbeitung. Das ursprünglich vom Unternehmen Willow Garage gegründete Projekt wird inzwischen von einer Community und verschiedenen Industriepartnern weiterentwickelt (vgl. [76]). Ursprünglich für Serviceroboter gedacht, unterstützt es inzwischen eine Vielzahl weiterer Roboter wie Quadcopter, Roboterarme und Greifer. Kern des Frameworks bildet eine Publish / Subscribe Nachrichtenarchitektur, bei der Nachrichten in Topics veröffentlicht und abonniert werden können. Dabei ist sowohl die Nutzung vorgefertigter Nachrichtenformate für gängige Sensorwerte, Sollwertvorgaben und andere Datenformate als auch die Definition eigener Nachrichtentypen möglich. Ebenso ist eine Infrastruktur zum Anbieten und Nutzen von Services verfügbar. Services können Parameter entgegennehmen und nach Ausführung ein Ergebnis liefern. Auch hier ist die Nutzung vorhandener Ser-

vicetypen oder die Definition eigener Services möglich. Für die Durchführung von Aufgaben, die über einen längeren Zeitraum laufen (z.B. das Ausführen einer Bewegung eines Roboterarms) besteht die Möglichkeit, statt eines Services eine Action anzubieten bzw. zu nutzen. Diese liefert kontinuierlich den Fortschritt der ausgeführten Aktion und lässt sich während der Ausführung abbrechen. Durch die Community werden viele Treiber und Tools zur Verfügung gestellt und weiterentwickelt. Funktionalitäten und Programmteile sind in sogenannte Nodes gekapselt, welche über das Nachrichtensystem miteinander kommunizieren und in Pakete gegliedert sind. Das Erstellen dieser Nodes wird von ROS in den Sprachen C++ oder Python unterstützt. Die Community stellt aber auch Tools für die Entwicklung in anderen Sprachen wie Java oder C# zur Verfügung. Zur Bedienung stehen diverse Kommandozeilenwerkzeuge zur Verfügung, mit denen sich unter anderem Nodes starten, Informationen zu Services und Topics anzeigen und Probleme diagnostizieren lassen. ROS ist in zwei Versionen (ROS1 und ROS2) verfügbar, von denen wiederum verschiedene Releases existieren. Bei den Releases wird zwischen regulären und Long Term Support (LTS) Releases unterschieden, welche sich durch die Dauer der Unterstützung durch Updates unterscheiden. ROS1 bildet die erste Version des Frameworks und ist um den zentralen Kern, den roscore, aufgebaut. Dieser verwaltet die Verteilung von Nachrichten und die Koordination von Services und Actions (vgl. [33]). Die Nachrichtenkommunikation erfolgt über TCP / UDP und ein selbst entwickeltes Protokoll (vgl. [34]) Das unterstützte Betriebssystem ist für jedes Release jeweils ein konkretes Ubuntu Release (vgl. [2]). In der zweiten Version unterstützt ROS zusätzlich zu Ubuntu auch die Betriebssysteme OSX El Capitan und Windows 10. ROS2 befindet sich noch in einem frühen Stadium der Entwicklung und viele der in ROS1 vorhandenen Features sind noch nicht nach ROS2 portiert. Die größte Neuerung zu ROS1 ist die Umstellung des Kommunikationslayers auf das DDS Protokoll ([84]). Da dieses keinen festen Master benötigt, entfällt der roscore und bei der Verteilung von Nodes über das Netzwerk muss die IP des Masters nicht mehr bekannt sein. ROS Industrial ist eine von der Industrie getriebene Weiterentwicklung von ROS, die den Einsatz von ROS in der Produktion und im industriellen Umfeld ermöglichen soll (vgl. [76]). Weitere Anwendungsbereiche sind der Automotive-Sektor sowie die Luft- und Raumfahrt. Bei dem Projekt steht die Anbindung von Industrierobotern verschiedener Hersteller im Vordergrund.

Das letzte hier vorgestellte ROS-Derivat bildet H-ROS (Hardware Robot Operating System, vgl. [78]). Bei H-ROS liegt der Fokus auf der Definition einer einheitlichen Soft- und Hardwareschnittstelle für Roboterkomponenten. Durch ein Hardwaremodul, welches die Kommunikation mit ROS übernimmt, kann ein beliebiger Sensor oder Aktuator an ROS angebunden werden. Dieses

System ist der Versuch, modulare Hardware in der Robotik einzusetzen und wird deshalb im Abschnitt 4.5 im Detail aufgegriffen.

4.5 Modulare Hardware

Im Umfeld der Robotik ermöglicht modulare Hardware eine Veränderung der physikalischen Beschaffenheit eines Roboters zur Laufzeit (vgl.[85]). Diese Veränderung kann dabei durch den Roboter selbst oder von außen durchgeführt werden. Durch selbst durchgeführte Rekonfiguration kann ein Roboter andere Fähigkeiten erlangen (vgl. [36]) oder sich durch Tausch eines Moduls bei einem Defekt selbst reparieren. Die gängige Herangehensweise in Projekten zu modularer Roboterhardware ist, die Roboter aus einzelnen Bausteinen oder Modulen aufzubauen, welche jeweils eine konkrete Aufgabe erfüllen (vgl. [85], [36]) So gibt es im Cubelet Projekt (vgl. [14]) beispielsweise Module zum Ausführen von Aktionen wie Fahren (Action Cubelets), zur Verarbeitung von Daten (Think Cubelets) oder zur Wahrnehmung der Umwelt durch Sensoren (Sense Cubelets). Den Kern einer solchen modularen Architektur stellt eine Schnittstelle dar, welche den Modulen eine Kommunikation untereinander ermöglicht. Diese muss sowohl hardwareseitig implementiert, als auch in Software spezifiziert sein. Bei den Modulen von Tinkerforge (vgl. [90]) existiert eine standardisierte Schnittstelle zur Verbindung von Bricks mit Bricks und für die Verbindung zwischen Bricks und Bricklets. Diese zeichnet sich sowohl durch eine einheitliche Hardwareschnittstelle als auch durch ein von allen Modulen verwendetes Protokoll aus. Die Standardisierung der Schnittstelle ermöglicht Verschalten der Module auf beliebige Art und Weise. Dadurch, dass Tinkerforge keine Erweiterung der verfügbaren Module durch andere Hersteller zulässt, kann garantiert werden, dass für jeden möglichen Brick oder Bricklet eine Unterstzung durch Treiber und Bibliotheken in der aktuellen API gewährleistet ist. Dieses Vorgehen ermöglicht es Nutzern der Module sich auf die Implementierung der gewünschten Funktionalität des Gesamtsystems zu konzentrieren, ohne dabei die Details der Hardwareansteuerung beachten zu müssen. In H-ROS (vgl. [78]) ist die Integration von neuer Hardware über das Hinzufügen eines zusätzlichen Hardwaremoduls zu der zu integrierenden Hardwarekomponente gelöst. Dieses H-ROS SoM (System on Module, vgl. [62]) enthält einen Einplatinenrechner, welcher auf einem echtzeitfähigen Linux System die Middleware ROS2 ausführt und über Ethernet und TSN (vgl. [91]) andere Module anbindet. Das SoM steuert die zu integrierende Hardware (z.B. einen Greifer) über einen auf dem Gerät implementierten Treiber an und Kommuniziert nach außen hin ROS-kompatibel. ROS spezifiziert einheitliche Interfaces für viele Klassen von Roboterhardwa-

re, deren Nutzung die Austauschbarkeit von Komponenten (z.B. von einem Greifer der Firma Schunk durch einen der Firma Robotiq) ermöglicht.

5 Konzepte zur Missionsplanung und Smart-Components

Dieses Kapitel beschreibt die erarbeiteten Konzepte zur Missionsplanung in Multicopterensembles sowie die darin verwendeten Smart-Components. Die im Abschnitt 3 herausgearbeiteten Anforderungen für die Missionplanung werden konzeptuell in ein Framework gegossen und im ersten Teil des Kapitels vorgestellt. Zudem wird eine Strategie zum Erstellen von „smarten“ Sensoren aufgezeigt, welche Probleme bisheriger Lösungen verbessert. Dabei wird im Speziellen auf die Visualisierung von Messwerten und Zusatzinformationen zur Erkennung fehlerhafter Messwerte und defekter Sensoren eingegangen. Ebenfalls wird eine Methode erarbeitet, die das dynamische Austauschen von Sensoren ermöglicht und die Erweiterbarkeit des Systems um neue Sensoren behandelt. Außerdem wird ein Konzept zur Kalibrierung von Sensoren vorgestellt, bei dem die Kalibrierungsdaten auf dem Sensor gespeichert und verrechnet werden. Zunächst wird ein Überblick über den Aufbau eines typischen sensorbasierten Fluges und den daran beteiligten Geräten gegeben.

5.1 Systemaufbau Sensorflug

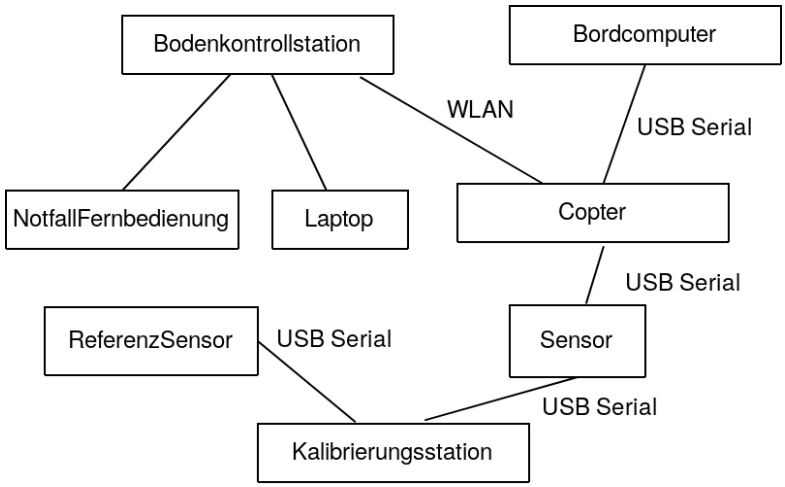

Abbildung 5.1: Schematische Darstellung des Systemaufbaus

Der Aufbau des erarbeiteten Konzepts für Messflüge ist in Abbildung 5.1 dargestellt. In einem Flug werden mehrere mit Sensoren für verschiedene Messgrößen ausgerüstete Copter von einer Bodenkontrollstation gesteuert. Diese besteht aus einem Laptop für den Start und das Deployment der Missionen und für die Überwachung des Fluges und einer Funkfernbedienung für die Copter um diese im Notfall stoppen zu können. Die verwendeten Sensoren können an einer Kalibrierstation kalibriert werden. Hierzu sind an der Kalibrierstation Referenzsensoren für die jeweilige Messgröße angebracht. An der Kalibrierstation errechnet der zu kalibrierende Sensor anhand er Referenzwerte eine Korrekturkurve für seine Messgröße.

5.2 Missionsplanung

Die Missionsplanung ermöglicht es, Abläufe für Teilnehmer des Ensembles zu modellieren und in wiederverwendbarer Art und Weise zu speichern. Hierfür wurde ein Baukastensystem erarbeitet, in dem eine Mission durch Verschalten verschiedener Blöcke definiert wird. Das System orientiert sich an Elementen der UML-Aktivitätsdiagramme (vgl. [93]) und Petrinetze (vgl. [39]). Es wird jeweils ein Teil des Funktionsumfangs der beiden genutzt. Die zu verschaltenden Blöcke lassen sich in zwei Kategorien einteilen. Funktionale Blöcke dienen zum Ausführen konkreter Funktionalitäten, wie z.B. dem Scharfschalten der Rotoren. Kontrollflussblöcke hingegen steuern und beeinflussen den Ablauf des Programms. Nachfolgend werden die einzelnen Blocktypen sowie ihre Funktion und Verwendung im Detail erklärt. Zudem wird ein Konzept zur Synchronisierung mehrerer Copter über Blöcke, die als Barriere fungieren, vorgestellt sowie eine Methodik zur Erweiterbarkeit des Funktionsumfangs über Plugins erläutert.

5.2.1 Block Definition Language (BDL)

Um Missionen einfach und in wiederverwendbarer Art und Weise definieren zu können, wird die Block Definition Language (BDL) eingeführt. Diese ermöglicht das Verschalten von Blöcken, welche verschiedene Aufgaben erfüllen, in einer Art Baukastensystem. In diesem wird für jeden Copter eine Mission im JSON-Format (JavaScript Object Notation, vgl. [22]) definiert.

Grundsätzlich wird zwischen Kontrollflussblöcken und funktionalen Blöcken unterschieden. Funktionale Blöcke enthalten die tatsächliche Funktionalität des Programms und tragen aktiv zur Ausführung der Mission bei. Kontrollflussblöcke dienen zur Steuerung und Manipulation des Missionsablaufs. Blöcke können dadurch sowohl sequenziell als auch parallel ausgeführt werden. Ebenso sind bedingte Verzweigungen sowie die Reaktion auf Fehler-

fälle möglich, um damit den Programmfluss zu steuern. Die verschiedenen Blöcke können unter der Beachtung der nachfolgenden Regeln beliebig miteinander verbunden werden und ermöglichen somit eine flexible Definition von Missionen. Die Verschaltung der Blöcke untereinander erfolgt über das Anlegen eines oder mehrerer Nachfolgeblöcke. Zur Laufzeit der Mission wird in jedem Block entweder ein Nachfolgeblock (sequenzielle Ausführung) oder mehrere Nachfolgeblöcke (parallele Ausführung) aus den möglichen Nachfolgern für die Ausführung im nächsten Schritt gewählt.

In UML-Aktivitätsdiagrammen wird der hier Programm- bzw. Kontrollfluss genannte Prozess über das Weiterreichen, Multiplizieren und Vernichten von Tokens modelliert. An einigen Stellen wird auf diese Sichtweise zurückgegriffen, um Zusammenhänge anschaulicher zu erklären.

Nachfolgend werden erst die Kontrollflussblöcke und anschließend die funktionalen Blöcke und ihre Fähigkeiten im Detail erklärt.

Kontrollflussblöcke

Kontrollflussblöcke steuern den Ablauf der Mission. Sie dienen lediglich zur Programmflussmanipulation und nicht zur Ausführung von konkreten Aufgaben in der Mission. Durch sie wird unter anderem die Parallelisierung von Programmflüssen und das bedingte Verzweigen in einer Mission ermöglicht. Im folgenden Abschnitt wird genauer auf die Funktionalität der einzelnen Kontrollflussblöcke eingegangen.

Start- und Endblöcke Zur Definition von Anfang und Ende einer Mission werden *Start*- und *End*-Blöcke benötigt. In einer Mission darf genau ein *Start*-Block existieren, an dem der Kontrollfluss beginnt. Ähnlich zur Definition in UML-Aktivitätsdiagrammen endet der Programmfluss beim Erreichen eines *End*-Blocks. In einem Programm dürfen mehrere *End*-Blöcke existieren. Die Mission wird jedoch beim ersten Erreichen eines *End*-Blocks beendet und eventuelle weitere parallele Kontrollflüsse werden terminiert. Abbildung 5.2 zeigt den Aufbau einer einfachen Mission mit einem *Start*- und zwei *End*-Blöcken. Direkt nach dem Start wird die Mission in zwei parallele Abläufe aufgeteilt. Dabei pausiert der rechte Ablauf für zehn Sekunden, wohingegen der linke lediglich fünf Sekunden im *Wait*-Block verweilt. Fünf Sekunden nach Programmstart erreicht der linke Ablauf den Endzustand, wodurch die Mission terminiert und der rechte Pfad der Mission frühzeitig abgebrochen wird.

Sequenzielle Ausführung Die sequenzielle Ausführung von Blöcken ist durch die Definition eines Nachfolgers für den jeweiligen Block möglich. Bis

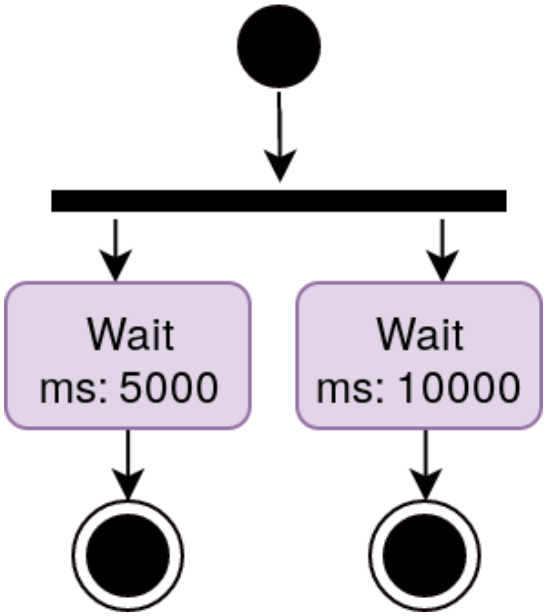

Abbildung 5.2: Mission mit einem *Start*- und zwei *End*-Blöcken

auf den *End*-Block hat jeder Block mindestens einen Verweis auf einen Nachfolger, zu welchem die Mission nach Beenden des aktuellen Blocks wechselt. Abbildung 5.3 zeigt eine einfache Mission mit zwei funktionalen Blöcken und je einem *Start*- und *End*-Block. Die Mission beginnt bei dem *Start*-Block, der keine interne Funktionalität besitzt. Deshalb folgt der Programmfluss sofort dem Verweis auf *Block1*. Nach dem Ausführen der in *Block1* hinterlegten Funktionalität, wird zum nachfolgenden Block (in diesem Fall *Block2*) gesprungen. Dieser Ablauf wiederholt sich solange, bis die Mission an einer Stelle einen *End*-Block erreicht, durch den die Mission terminiert.

Forks und Joins *Fork*-Blöcke ermöglichen die Parallelisierung des Programmflusses durch das gleichzeitige Ausführen mehrerer Nachfolgerblöcke. In der UML-Aktivitätsdiagrammsichtweise werden hier also Tokens vermehrt und gleichzeitig an alle Nachfolger weitergereicht. Die einzelnen Pfade werden solange unabhängig voneinander ausgeführt, bis ein *Join*-Block erreicht wird. In diesem Block enden die Pfade und der Nachfolger des Joins wird erst ausgeführt, wenn die Tokens von allen eingehenden Kanten angekommen sind. Abbildung 5.4 zeigt eine Mission, bei der zwei Blöcke parallel ausgeführt werden. Nach dem Startzustand wird das Token im *Fork*-Block aufgeteilt und parallel an *Block1* und *Block2* weitergegeben. Erst wenn beide Tokens am *Join*-Block angekommen sind, d.h. die Ausführung von *Block1* und *2* beendet ist, wird in den Endzustand gewechselt.

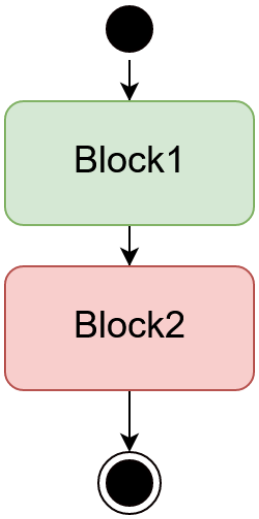

Abbildung 5.3: Beispiel für den sequenziellen Ablauf eines Programms

Decisions *Decision*-Blöcke führen nach Ausführung einen von mehreren Nachfolgern aus. Die Auswahl des Nachfolgers erfolgt dabei durch eine interne Berechnung im *Decision*-Block. So kann z.B. ein *DecisionRandom*-Block intern eine Zufallszahl berechnen und anhand dieser den Nachfolger bestimmen. Ebenso möglich ist das Erstellen eines *IsROSTopciActive*-Blocks, der das Expertenwissen für die ROS-Kommunikation beinhaltet und damit entscheiden kann, ob Nachrichtenverkehr in einem bestimmten Topic besteht. Mit diesem Wissen kann der Block einen Nachfolger für den Fall *Topic Aktiv* und *Topic Inaktiv* bestimmen und der Missionsablaufsteuerung mitteilen. Zu beachten ist, dass in Decision Nodes nur genau ein Nachfolger gewählt werden und somit kein Forking stattfinden kann. In Abbildung 5.5 ist der Aufbau einer einfachen Mission mit einer Decision dargestellt. Nach dem Start wird im *DecisionRandom*-Block einer von zwei möglichen Nachfolgepfaden zufällig gewählt und ausgeführt. Ein Merge erfolgt ohne gesonderten Block durch den Verweis beider alternativer Programmabläufe auf den selben Nachfolger. Ein eigener *Merge*-Block wurde nicht implementiert, da er keine Funktion außer das Aufrufen des Nachfolgers erfüllen würde. Zu beachten ist, dass ein impliziter Merge wie er in der Abbildung dargestellt ist, nicht der UML 2.5 Spezifikation für Aktivitätsdiagramme (vgl. [93]) entspricht. In dieser werden die zwei eingehenden Pfeile am Endknoten als impliziter Join gewertet, was zu einem Deadlock führt. Da ein Join in der BDL stets mit einem *Join*-Block erfolgen muss und ein impliziter Join nicht möglich ist, ist das Zusammenführen von Programmflüssen an einem Node als Merge definiert.

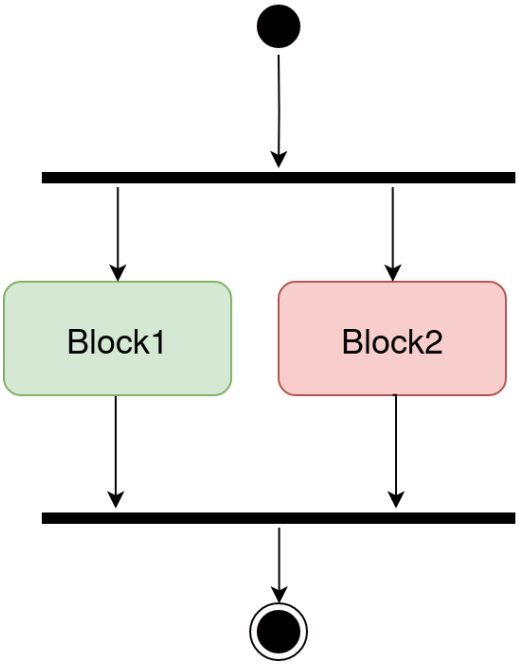

Abbildung 5.4: Beispiel für die Nutzung von *Fork*- und *Join*-Blöcken. Programmfluss wird direkt nach dem Start parallelisiert. Die Mission terminiert erst, nachdem beide Pfade am Join angekommen sind.

Funktionale Blöcke

Im Gegensatz zu den Kontrollflussblöcken, die den Missionsablauf steuern, führen funktionale Blöcke eine konkrete Aufgabe aus. Dies kann z.B. das Starten eines ROS-Nodes, das Warten auf ein bestimmtes Ereignis oder ein entfernter API-Aufruf sein. Um sie so generisch wie möglich zu halten, sollen in der Basisversion der Missionsplanung keine auf einen bestimmten Anwendungsfall zugeschnittenen Blöcke enthalten sein. Die konkrete Implementierung der Funktionalitäten wird in domänenspezifischen Plugins gekapselt, welche neue Blöcke bei der Missionsplanung registrieren (vgl. Abschnitt 5.2.2). Obwohl funktionale Blöcke nicht primär zur Steuerung des Programmflusses gedacht sind, ergibt sich an einigen Stellen die Notwendigkeit, in den Programmablauf einzugreifen. So soll beispielsweise beim Auftreten eines Fehlers ein anderer Pfad gewählt werden, als beim erfolgreichen Ausführen der Aktion. Aus diesem Grund können in funktionalen Blöcken zusätzliche Nachfolger definiert werden, die im Fall eines Fehlers als alternative Pfade für den Programmfluss dienen. Dadurch kann u.a. mit Timeouts, dem nicht erfolgreichen Starten eines Nodes oder dem Fehlen einer notwendigen Hardwarekomponente umgegangen werden. Ein Block kann zwar beliebig viele

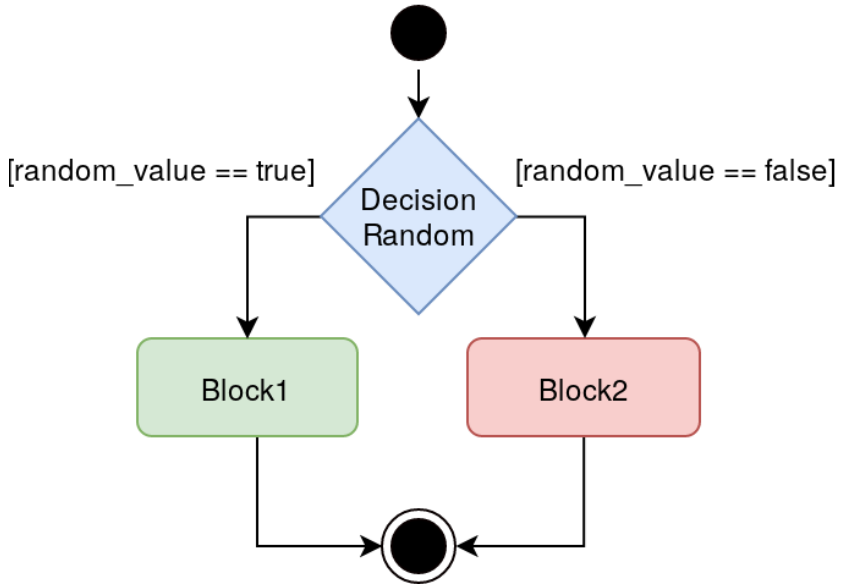

Abbildung 5.5: Beispiel für die Verwendung eines *Decision*-Blocks. Der Programmablauf verläuft, abhängig vom Inhalt des *Decision*-Blocks, entlang genau einem der aus dem Block ausgehenden Pfade.

Fehlerfälle behandeln, allerdings darf nur genau ein Nachfolger gewählt werden. Ein Fork ist nicht möglich. In Abbildung 5.6 ist eine einfache Fehlerbehandlung dargestellt. Anstatt den normalen Nachfolger direkt unterhalb zu wählen, wird im Fehlerfall der alternative, gestrichelte Pfad abgelaufen.

5.2.2 Erweiterbarkeit über Plugins

Die Missionsplanung ist generisch gehalten, um den Nutzer nicht an ein spezifisches Framework oder Tool zu binden. Ein Pluginsystem erlaubt das modulare Hinzufügen neuer Funktionen. Somit ist es möglich, ein Plugin für die Roboter Middleware ROS zu entwickeln und dieses in die Missionsplanung einzubinden. In dem Plugin können u.a. Blöcke zum Starten von Nodes oder dem Aufruf von Services implementiert werden. Auch *Decision*-Blöcke, wie das Testen auf Aktivität in einem Topic, sind möglich. Jeder Block hat einen global einzigartigen Typbezeichner und eine Callbackmethode, mit welchen er sich bei der Missionsplanung registriert. Sobald die Missionsplanung auf einen Block diesen Typs in einer Mission stößt, ruft sie die hinterlegte Callbackmethode auf. Sie übergibt dabei zusätzliche, für diesen Block in der Mission spezifizierte Parameter. Die Callbackfunktion führt die eigentliche Funktionalität aus und gibt, abhängig vom Ergebnis der Ausführung, einen

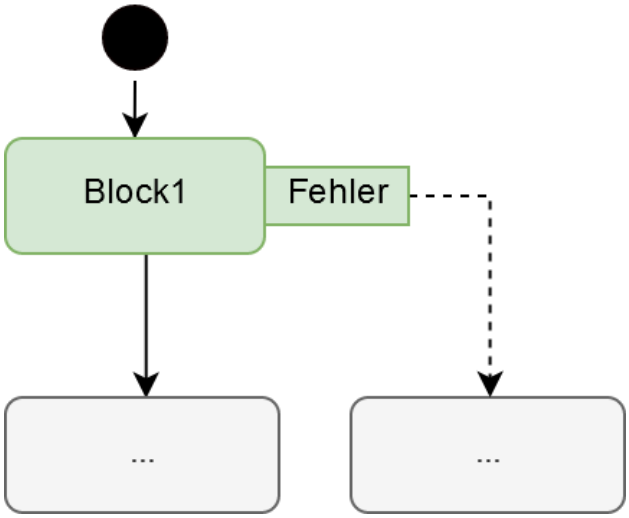

Abbildung 5.6: Beispiel für eine einfache Fehlerbehandlung. Tritt ein Fehler bei der Ausführung des Blocks auf, so kann dieser die Ausführung eines alternativen Programmpfads auslösen. Analog zum Verhalten des *Decision*-Blocks wird immer nur einer der ausgehenden Pfade gewählt.

Nachfolger zurück. Durch diese Architektur lassen sich sowohl funktionale als auch *Decision*-Blöcke erstellen. Die Erweiterung anderer Programmflussblöcke über das Pluginsystem ist nicht angedacht, da der bisherige Funktionsumfang ausreichend für die Ausführung der Mission aus dem Fallbeispiel ist und eine weitere Schnittstelle an dieser Stelle den Rahmen der Arbeit gesprengt hätte. Die Abbildung 5.7 zeigt den Ablauf der Registrierung und Nutzung eines Plugins. Zunächst registriert das Plugin einen Block mit dem Typbezeicher *StartNode* sowie einer Callbackmethode *callbackFunc* beim MissionPlayer. Anschließend kann der Nutzer eine Mission, die einen Block vom Typ *StartNode* beinhaltet, ausführen. Trifft der MissionPlayer auf einen *StartNode*-Block, so wird die hinterlegte Callbackmethode *callbackFunc* mit in der Mission definierten, zusätzlichen Parametern (*args*) aufgerufen.

5.2.3 Synchronisierung mehrerer Copter

Die Synchronisierung mehrerer Copter im Ensemble wird ebenfalls über ein Plugin gelöst, welches die Nutzung von Barrieren in den Missionen ermöglicht. In der Initialisierungsphase der Mission werden alle Copter im Ensemble ermittelt und gespeichert. Dies geschieht, indem auf ein copterspezifisches Signal (*Heartbeat*) gewartet wird. Dadurch können die Teilnehmer des Ensembles dynamisch bestimmt werden, die später an den Barrieren anwesend

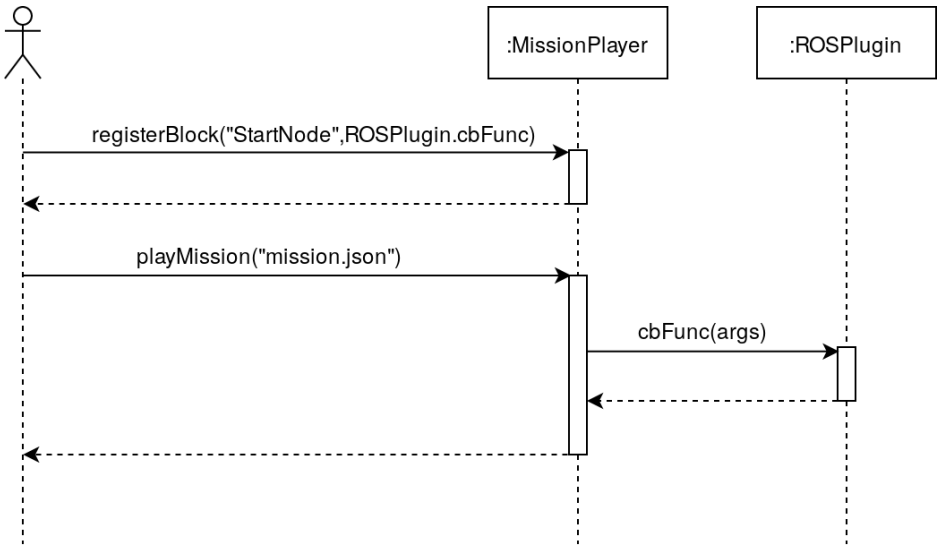

Abbildung 5.7: Registrierung und Nutzung eines Plugins

sind. Jede Barriere hat eine eindeutige ID, die sie von anderen unterscheidbar macht. Erreicht ein Copter eine Barriere, so teilt er das anderen Teilnehmern des Ensembles mit und wartet, bis auch diese die selbe Barriere erreicht haben. Für die Mission bedeutet das, dass der Copter im Block für die jeweilige Barriere verweilt, bis alle anderen Copter auch an dieser angekommen sind. Erst danach wird der Programmfluss synchronisiert bei allen Teilnehmern fortgesetzt. Abbildung 5.8 veranschaulicht die Synchronisation zweier Copter mittels Barrieren. Erreicht der *Copter1* beispielsweise die Barriere *Barrier1* vor *Copter2*, verweilt er so lange im *Barrier1*-Block, bis er erfährt, dass *Copter2* ebenfalls an dieser Barriere angekommen ist. Wenn die Programmausführung bis *Barrier2* im zweiten Copter schneller stattfindet, so erreicht dieser zuerst die zweite Barriere und wartet wiederum auf *Copter1*. Damit ist gewährleistet, dass die beiden Teilnehmer des Ensembles eine Barriere immer synchron verlassen.

5.3 Smart-Sensor

Der zentrale Aspekt des Smart-Sensors ist die Erweiterung eines klassischen Sensors um diverse „smarte“ Funktionen. Hierzu zählen das Anzeigen zusätzlicher Informationen zur einfacheren Fehlerdiagnose und Funktionsprüfung sowie ein erhöhter Automatisierungsgrad in der Handhabung. Ebenso werden sensorspezifische Funktionen im Sinne des Expert-Patterns der Softwaretechnik auf den Sensor ausgelagert. Hierzu zählt z.B. die Kalibrierung direkt auf dem Sensor. Durch einheitliche Schnittstellen wird zudem eine einfache Er-

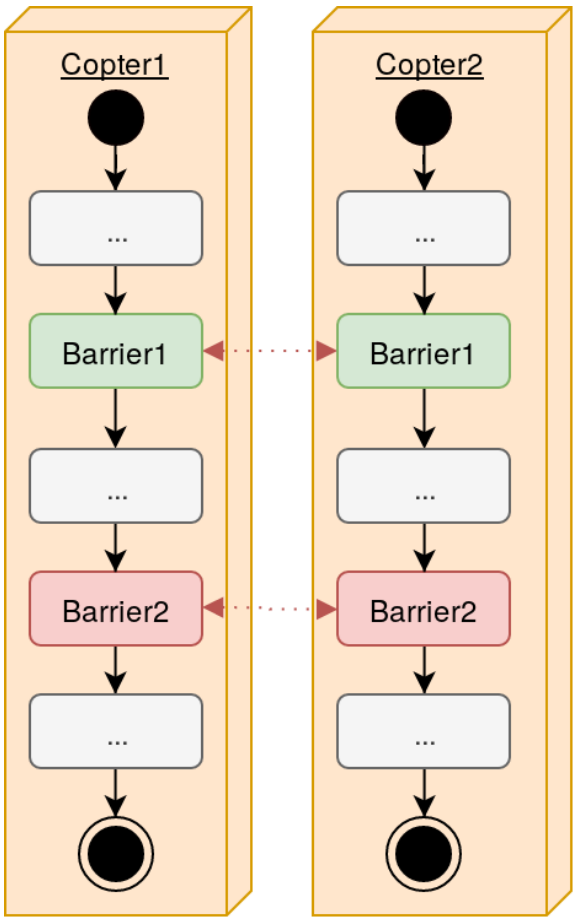

Abbildung 5.8: Synchronisierung mehrerer Copter mithilfe von eindeutig identifizierbaren Barrieren. Erst wenn alle Copter die jeweilige Barriere erreicht haben, läuft der Programmfluss weiter.

weiterbarkeit und Austauschbarkeit sichergestellt. Die hier angesprochenen Punkte werden in den nachfolgenden Abschnitten genauer erklärt.

5.3.1 Visualisierung von Sensorinformationen

Um beim Aufbau einer Mission, z.B. für eine Messkampagne, die Fehlerdiagnose zu erleichtern, kann die Anzeige umfangreicher Daten zu den Sensoren und anderen Komponenten hilfreich sein. Durch genaue Informationen zum angeschlossenen Sensor kann ein Defekt frühzeitig erkannt werden, wenn direkt ersichtlich ist, dass der gemessene Wert unrealistisch ist oder keine Kommunikation mit dem Controller stattfindet. Hierzu wurden einige Werte herausgearbeitet, die bei einem einfachen Sensor für die Fehlerdiagnose hilfreich sein können. Hierzu zählen

- der eigentliche Sensorwert
- der Kommunikationsstatus mit dem Controller
- die Aktualisierungsfrequenz des Sensorwerts
- die ID des Sensors, über die er im Controller gefunden werden kann
- der Status der Kalibrierung
- eine durch Expertenwissen des Sensors durchgeführte Plausibilitätsprüfung des Sensorwerts

Diese Werte sind teilweise auch in den verarbeitenden Systemen einsehbar, allerdings müssen diese in der Praxis oft mittels eines externen Rechners abgefragt werden, was die Diagnose umständlich macht. Als Lösungsansatz für dieses Problem bietet sich das Anzeigen der Informationen am Sensor selbst, beispielsweise mit visuellem oder akustischem Feedback an. Somit besteht kein Zweifel bei der Zuordnung von Werten bei mehreren Sensoren und die Abfrage der Werte gestaltet sich unkompliziert.

5.3.2 Austauschbarkeit der Sensoren

Sollte bei einem Sensor eine Fehlfunktion detektiert worden sein, ist es wichtig, dass der Sensor schnell und einfach ausgetauscht werden kann. So können definierte Intervalle bei Messflügen eingehalten werden. Dabei ist es hilfreich, wenn nach dem Tausch eines Sensors keine Änderungen am Programm vorgenommen werden müssen. Im besten Fall lässt sich ein Sensor durch einen anderen Sensortyp mit derselben Messgröße (z.B. Temperatur) tauschen. Um diesen Grad der Austauschbarkeit zu gewährleisten, müssen Schnittstellen auf Software- und Hardwareseite geschaffen werden. Es gilt, die Installation eines neuen Treibers beim Wechsel des Sensortyps zu umgehen. Daher wird softwareseitig ein einheitliches Protokoll definiert, worüber der Sensor mit dem Controller kommunizieren kann. Ebenso ist die Standardisierung von Einheiten der Messgrößen wichtig. So kann z.B. ein Sensor, der die Temperatur in Grad Celsius angibt, nicht ohne Änderung im Programm oder zusätzliche Informationen durch einen Sensor erstetzt werden, welcher in Grad Fahrenheit misst. Aus diesem Grund sind die Einheiten (außer Grad Celsius bei Temperatur) auf die Si Einheiten beschränkt. Eine mögliche Erweiterung könnte an dieser Stelle eine semantische Annotation der Sensorwerte mit der Einheit sein oder die Möglichkeit, eine Konvertierung des Werts in andere Einheiten auf dem Sensor vorzunehmen. Zusätzlich sollen neu angeschlossene Sensoren automatisch erkannt und ins System eingebunden werden, um

zeitaufwändige Neustarts des Systems oder einzelner Programmteile zu reduzieren. Um einen schnellen Tausch der Hardware zu ermöglichen, sollten Sensoren außerdem werkzeuglos wechselbar sein. Wird bei einem solchen System ein defekter Sensor gegen einen Neuen getauscht, kann er an derselben physikalischen Schnittstelle angeschlossen werden. Das System erkennt den Sensor automatisch und bindet ihn über einen generischen Treiber ein. Anschließend sind die Messwerte des Sensors im System zur Auswertung durch ein Programm verfügbar.

5.3.3 Erweiterbarkeit um neue Sensoren

Für eine einfache Integration neuer Sensoren in das System muss der Entwicklungs- und Programmieraufwand gering gehalten werden. Hierzu bietet es sich an, eine Hard- und Softwareabstraktionsschicht zwischen Sensor und Controller zu schalten, welche Sensoren verschiedenster Protokolle ansteuern und über ein einheitliches Protokoll anbinden kann. Vergleichbar mit einem Arduino Entwicklungs-Board, welches je nach Programmierung Daten von Schnittstellen wie I2C, Analogeingänge und OneWire über eine USB Schnittstelle an einen PC weitergeben kann, übernimmt der Hardwarewrapper das Auslesen der Sensoren. Dadurch wird die konkrete Treiberimplementierung auf dem Sensor gekapselt. Der Hardwarewrapper, welcher Bestandteil des „smarten“ Sensors ist, gibt die Daten aufbereitet und über einheitliche Hardwareschnittstellen und Protokolle an den Controller weiter. Dieser Hardwarewrapper muss - ähnlich wie ein Arduino - vielseitig aufgebaut sein, um mithilfe eines universellen Geräts möglichst viele Sensoren abzudecken und damit die Integration weiter zu erleichtern. Softwareseitig sollte eine Bibliothek, die für alle Sensortypen benötigte Funktionalität wie die Kommunikation mit dem Controller kapseln. Diese übernimmt ebenfalls die Speicherung und Errechnung von Kalibrierungswerten sowie die Ansteuerung von Geräten zur Visualisierung wie z.B. Displays. Das erspart die mehrfache Implementierung von identischen Features in verschiedenen Sensoren.

5.3.4 Kalibrierung

Die Kalibrierung und Errechnung von Korrekturwerten erfolgt nicht extern auf einem anderen Gerät, sondern direkt auf dem Sensor. Der Sensor kennt seine Messcharakteristik und die effizienteste Methode zur Errechnung seiner Korrekturkurve. Deshalb bietet es sich an, diese Funktionalität gemäß dem Expert-Pattern der Softwaretechnik auf den Sensor auszulagern.

Dazu ist es notwendig, dass der Sensor in einen Kalibrierungsmodus versetzt werden kann, in dem er Messwerte von einem Referenzsensor erhält. Diese werden als „ground truth“, also als Referenzwert für den tatsächlichen

Wert der zu messenden Größe, für die Messfehlerberechnung herangezogen. Letztere ergibt sich aus der Differenz des eigenen Messwerts und des Referenzsensorwerts. Wird dieser Fehler über einen größeren Messwertbereich ermittelt, kann der Sensor daraus eine Korrekturkurve errechnen, die für jeden gemessenen Sensorwert einen Korrekturwert bereitstellt. Dieser Korrekturwert berichtigt daraufhin den Messwert. Somit liegt dem Controller direkt der berichtigte Sensorwert vor. Der Vorteil der Verrechnung des Korrekturwerts auf dem Sensor ist eine verbesserte Austauschbarkeit der Sensoren, da sensorspezifische Korrekturen auf dem Sensor und nicht auf dem Controller durchgeführt werden. Eine Anpassung des Codes auf dem Controller, um korrigierte Sensorwerte zu erhalten, ist somit nicht notwendig.

6 Realisierung und Implementierung von Missionsplanung und Smart-Components

Dieses Kapitel behandelt die Implementierung der bereits in Kapitel 5 vorgestellten Konzepte zu Missionsplanung und die Realisierung von Soft- und Hardware der Smart-Sensors. Dafür wird zunächst ein Überblick zum Gesamtsystem gegeben, bevor die Missionsplanung sowie deren Erweiterbarkeit über Plugins und die Verteilung von Missionen behandelt wird. In den darauf folgenden Abschnitten ist die Hard- und Softwarearchitektur der Smart-Sensors beschrieben. Dabei wird im Speziellen auf das System zur Kalibrierung von Sensoren sowie auf die Erweiterbarkeit des Systems um weitere Sensoren eingegangen. Zuletzt wird der Aufbau einer einfachen Bodenkontrollstation zur Steuerung und Überwachung der Mission sowie zum zentralen Sammeln von Sensordaten beschrieben.

6.1 Systemaufbau

Das in der Arbeit realisierte System besteht aus einer Bodenkontrollstation und mehreren Quadrocoptern. Abbildung 6.1 beschreibt den Aufbau des Gesamtsystems. Um das Diagramm übersichtlich zu halten, ist nur ein Copter abgebildet. Auf jedem der Copter ist eine Autoquad M4 (vgl. [42]) Flugsteuerung und ein MissionController in Form eines Raspberry Pi 3B (vgl. [68]) verbaut. An dem Raspberry Pi können über die daran angeschlossen Smart-Sensor-Stecker bis zu zwei Smart-Sensor-Boards verbunden werden. In der Abbildung sind jeweils ein Board mit einem DHT22 (vgl. [97]) und einem DS18B20 (vgl. [18]) Sensor erkennbar. Der Flugcontroller der Copter ist über das DSMX Funkprotokoll (vgl. [51]) mit einer Spektrum DX9 Funkfernbedienung auf dem Boden verbunden, über die die Copter im Notfall manuell gelandet werden können. Der Raspberry Pi kommuniziert über einen WLAN Access Point mit einem Laptop, der zusammen mit der Funkfernbedienung die Bodenkontrollstation bildet.

In der Abbildung 6.2 ist die detaillierte Sicht auf die Artefakte auf den einzelnen Coptern dargestellt. Auf dem Raspberry Pi laufen die Frameworks ROS1 und ROS2, die über die ROS 1 Bridge miteinander kommunizieren.

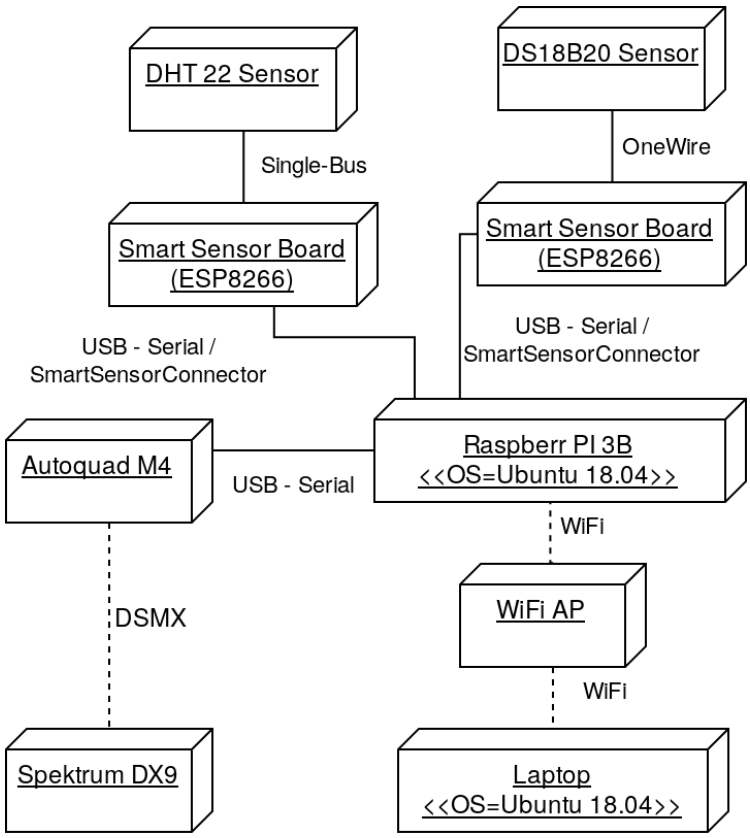

Abbildung 6.1: Verteilungsdiagramm für eine Mission mit einem Copter

In ROS1 wird das Paket *mavros* (vgl. [30]) ausgeführt, welches die Kommunikation mit dem Autoquad Flugcontroller übernimmt. Das SerialWatchdog Script wird automatisch nach dem Start gestartet und übernimmt die Initialisierung der Smart-Sensors nach dem Anstecken. Zusätzlich läuft auf dem MissionController der MissionPlayer, der die eigentliche Mission auf dem Copter ausführt. Auf den Smart-Sensor-Boards ist eine sensorspezifische Firmware aufgespielt, welche die Ansteuerung des Sensors und die Kommunikation mit dem MissionController übernimmt. Auf dem Laptop der Bodenkontrollstation ist ebenfalls das Release Dashing der ROS2 Middleware installiert (vgl. Abbildung 6.3). Dieses kommuniziert mit DDS (vgl. [64]) über WLAN mit den Coptern. Dadurch sind alle Daten und Messwerte der Copter auch auf der Bodenkontrollstation verfügbar. Diese Daten werden von dem von ROS bereitgestellten Tool *rosbag2* aufgezeichnet, welches von dem ebenfalls auf dem Laptop installierten MissionPlayer Skript bei der Ausführung der Mission gestartet wird. Die zentral aufgezeichneten und mit einem Zeitstempel versehenen Daten können nach der Ausführung der Mission ausgewertet werden. Zusätzlich sind auf der Bodenkontrollstation diverse

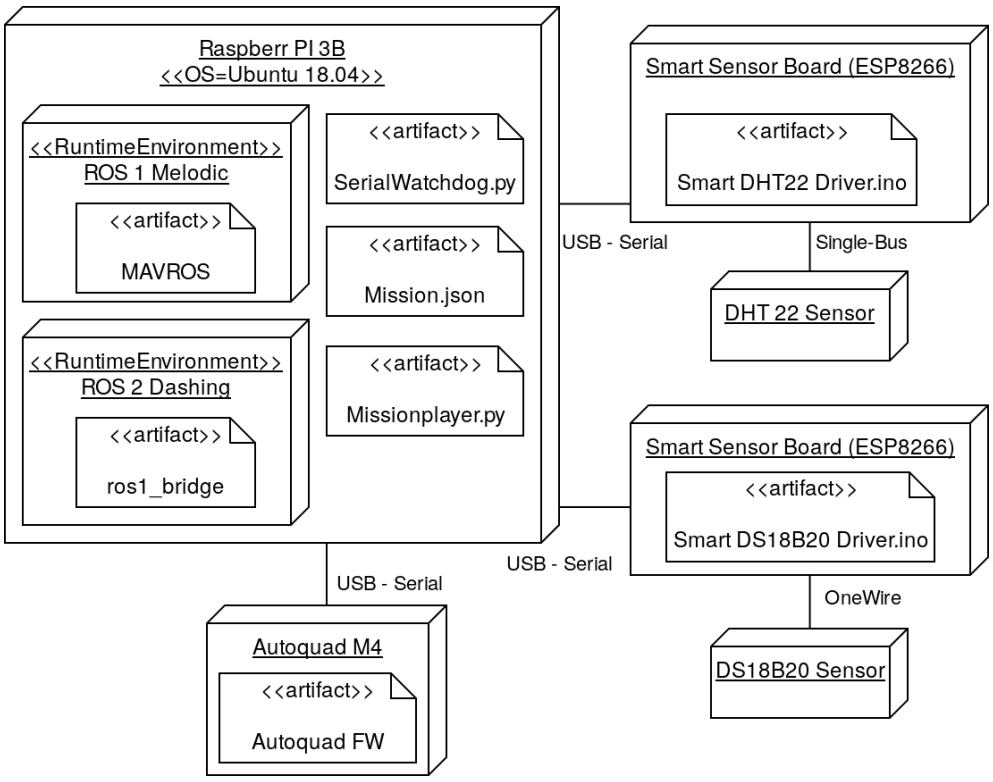

Abbildung 6.2: Verteilungsdiagramm des Copters

ROS-eigene Tools installiert, die das Überwachen der Mission zur Laufzeit erlauben.

6.2 Missionsplanung

Nachfolgend wird die Implementierung des Missionsplanungsframeworks näher beschrieben. Dabei wird zuerst der allgemeine Aufbau erläutert und dann auf einzelne Features eingegangen. Zu den Features zählen die Behandlung von Fehlerfällen, das Deployment und die Erweiterbarkeit über Plugins. Darüber hinaus werden die beiden implementierten Plugins zur Synchronisierung des Ensembles und zur ROS-Integration kurz vorgestellt.

6.2.1 Allgemeiner Aufbau

Das Missionsplanungsframework ist in Python geschrieben und dadurch plattformunabhängig. Da die beiden implementierten Plugins jedoch das ROS1- und ROS2-Framework benötigen, wird der volle Funktionsumfang nur mit ROS-kompatiblen Systemen erreicht. Auf jedem der Copter des Ensembles

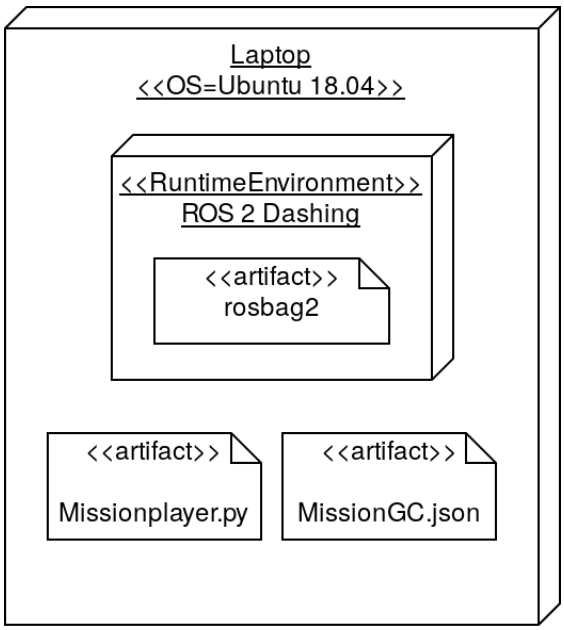

Abbildung 6.3: Verteilungsdiagramm des Rechners der Bodenkontrollstation

ist ein RaspberryPi 3B montiert, welcher Ubuntu 18.04 sowie ROS in der Version Melodic und Dashing ausführt (vgl. 6.1). Der MissionPlayer stellt den Kern des Missionplanungsframeworks dar. Er ist für die Ausführung von Missionen zuständig, welche in json Dateien definiert werden. Eine Mission besteht dabei aus einer Abfolge von Blöcken, welche wiederum durch Argumente parametrisiert werden können. Argumente können dabei verpflichtend angegeben werden müssen oder optional sein. Eine einfache Beispielmission ist in Listing 6.1 gezeigt und wird durch Abb. 6.4 grafisch dargestellt. Die Mission besteht lediglich aus einem *Start*-Block, einem darauffolgenden *CallService*-Block, welcher den ROS Service

`/example/service`

aufruft und einem *End*-Block. Die Abfolge der Mission wird über den *next*-Verweis in der Blockdefinition vorgegeben. Intern analysiert der MissionPlayer die json Datei und arbeitet diese Block für Block in der durch Verweise vorgegebenen Reihenfolge ab. Im MissionPlayer sind jedoch nur die nachfolgend behandelten Kontrollflussblöcke und der universell einsetzbare *Wait*-Block, welcher die Programmausführung eine vorgegebene Zeit blockiert, implementiert. Sollen weitere Fähigkeiten, wie die Interaktion mit ROS hinzugefügt werden, muss dafür ein Plugin geladen werden. Dieses registriert einen Blocktyp zusammen mit einer Callbackfunktion, die aufgerufen wird, sobald der

MissionPlayer diesen Blocktyp vorfindet (vgl. Abb. 5.7). Die eigentliche Implementierung der Funktionalität wird damit in das Plugin ausgelagert und der MissionPlayer kann um beliebige Funktionen erweitert werden. Neben der ROS-Integration ist die Synchronisierung des Ensembles über Barrieren als eigenes Plugin realisiert.

Listing 6.1: json Definition einer einfachen Mission

```
{
  "mission": {
    "b0": { // ID des Knotens
      "type": "Start",  // Knotentyp Startknoten
      "next": "b1"      // ID des Nachfolgerknotens
    },

    "b1":{
      "type": "CallService",
      "name": "/example/service", // Servicename
      "type": "std_srvs/SetBool", // Typ des Services
      "args": "{data: true}", // Parameter fuer Service
      "next": "b2"
    },

    "b2": {
      "type": "End"
    }
  }
}
```

Sequenzieller Programmablauf

Bei der sequenziellen Programmausführung werden die Blöcke in der vorher definierten Reihenfolge nacheinander ausgeführt. Dabei wird für jeden Block dessen zugehörige Callbackfunktion ausgeführt und die in der Mission definierten Parameter werden übergeben. Diese Funktion blockiert den Programmablauf bis zum Ende ihrer Ausführung an dem sie einen Nachfolgeblock zurück gibt. Dieser wird aus einem der in den Parametern definierten Nachfolgeblöcken bestimmt. Anschließend wiederholt sich der Prozess beim nächsten Block.

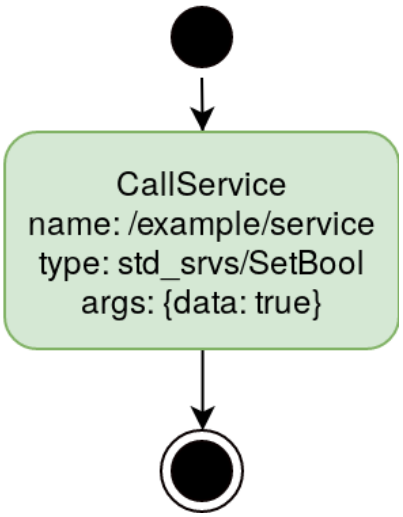

Abbildung 6.4: Grafische Repräsentation einer einfachen Mission

Parallelisierung

Bei der parallelen Programmausführung wird der Programmfluss an einer Stelle in mehrere, parallele Teilflüsse aufgeteilt (fork) und an späterer Stelle nach Terminierung aller Teilflüsse wieder zusammengeführt (join). In der Implementierung wird bei einem Fork für jeden der parallelen Programmflüsse ein Thread erzeugt, welcher den Teilfluss des Programms unabhängig von den anderen bearbeitet. Dadurch wird gewährleistet, dass die einzelnen Teilflüsse unabhängig voneinander ausgeführt werden.

Blöcke in Missionen halten lediglich eine Referenz auf ihren Nachfolger und nicht auf ihren Vorgänger. Das ist dadurch begründet, dass ein Block normalerweise keine Informationen über seinen Vorgänger benötigt und das manuelle Eintragen eines solchen Verweises das Erstellen der Mission unnötig verkomplizieren würde. Durch die fehlende Definition der Vorgängerblöcke ist es notwendig, eine Liste der Vorgänger eines Joins zur Laufzeit zu erstellen. Diese Liste wird benötigt, da der Programmablauf in einem Join erst fortgesetzt werden darf, wenn alle Vorgänger den Join erreicht haben. Zu diesem Zweck wird die Mission vor Beginn der Ausführung einmal komplett wie ein gerichteter Graph, dessen Kanten durch die next-Relationen gegeben sind, durchlaufen. Dabei wird eine Tabelle aller Vorgänger von allen Joins angelegt. Erreicht ein Teilfluss während der Mission einen Join, so aktualisiert er die Tabelle und vermerkt damit sein Antreffen am Join. Durch Vergleichen der angetroffenen und der erwarteten Vorgänger kann ermittelt werden, ob der Programmfluss fortgeführt werden kann. Sind alle Teilflüsse eines Joins angekommen, so wird mit der Ausführung des Nachfolgers des *Join*-Blocks

begonnen. Werden noch weitere Teilflüsse am Join erwartet, terminiert der Thread. Auf diese Art und Weise können simple Join- und Fork-Operationen durchgeführt werden, um den Programmfluss zu parallelisieren. Listing 6.2 zeigt die Definition einer einfachen Mission mit der Ausführung zweier paralleler Programmflüsse in json. Eine visuelle Repräsentation der Mission ist in Abbildung 6.5 dargestellt. Nach dem Startknoten wird der Programmfluss über einen *Fork*-Block gespalten. Dieser hat als *next*-Parameter eine Liste von Nachfolgern. Für jeden Nachfolger in dieser Liste wird ein neuer Thread erstellt, in dem dieser ausgeführt wird. Dadurch teilt sich der Programmfluss auf und die beiden in der Liste referenzierten *Wait*-Blöcke werden parallel ausgeführt. Nach der in b3 definierten Wartezeit von 100 ms wird der Nachfolger b6 (ein *Join*-Block) erreicht. Nachdem noch nicht alle für den Join erforderlichen Blöcke diesen erreicht haben (b2 muss den Join ebenfalls erreichen), terminiert der Thread. Nachdem der parallel ausgeführte Block b2 die vorgegebenen 5000 ms pausiert hat, erreicht auch dieser den Block b6. Damit sind alle benötigten Teilflüsse am Join angekommen und der Thread fährt mit der Ausführung von b19 fort.

Listing 6.2: Mission mit paralleler Ausführung zweier *Wait*-Blöcke

```
{
  "mission": {
    "b0": {
      "type": "Start",
      "next": "b1"
    },
    "b1": {
      "type": "Fork",
      // Liste von Nachfolgern
      "next": ["b2","b3"]
    },

    // Unabhaengiges Warten...
    "b2": {
      "type": "Wait",
      "ms": "5000",
      "next": "b6"
    },

    "b3":{
      "type": "Wait",
      // ...mit unterschiedlichen Wartezeiten
      "ms": "100",
```

```
      "next": "b6"
    },

    // Zusammenfuehren des Programmflusses
    "b6": {
      "type": "Join",
      "next": "b19"
    },

    "b19": {
      "type": "End"
    }
  }

}
```

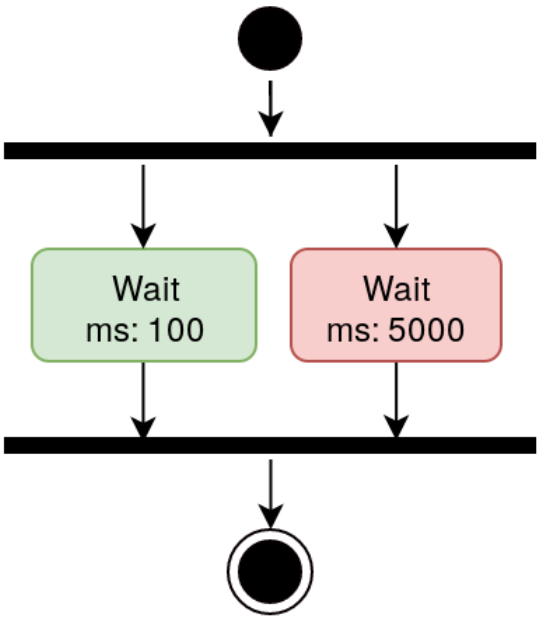

Abbildung 6.5: Grafische Repräsentation der Fork / Join Mission

Behandeln von Fehlerfällen

Zur Behandlung von Fehlerfällen in Blöcken kann ein alternativer Verweis auf einen Nachfolger in den Parametern des Blocks angegeben werden, welcher im Fehlerfall statt des *next*-Verweises ausgewählt wird. In der Implementierung

wird das durch Mitteilen des Nachfolgers im Rückgabewert des Callbacks erreicht. Im Listing 6.3 ist die Definition eines Blocks mit alternativem Nachfolger im Fehlerfall am Beispiel eines *WriteToFile*-Blocks dargestellt, welcher einen String in eine Textdatei schreibt.

Listing 6.3: json Definition des *WriteToFile*-Blocks

```
"b1"{                       // ID des Blocks
  "type": "WriteToFile"     // Typ des Blocks
  "file": "~/hello.txt"     // Name der Datei
  "text": "Hello World"     // Zu schreibender Text
  "next": "b2"              // Nachfolger im Erfolgsfall
  "nextError": "b4"         // Nachfolger im Fehlerfall
}
```

Wichtig ist hier der *nextError* Verweis, der zusätzlich zum normalen *next* existiert. In der Implementierung (vgl. Listing 6.3) wird im Erfolgsfall der Nachfolger des Parameters *next* zurückgegeben, bei Auftreten einer Exception in der Callbackfunktion wird der im Parameter *nextError* definierte Nachfolger zurückgegeben. Der MissionPlayer führt dann den Programmfluss mit dem zurückgegebenen Nachfolger fort.

Abbildung 6.6 veranschaulicht den Einsatz *WriteToFile*-Blocks grafisch und zeigt den alternativen Programmfluss im Fehlerfall als zusätzlichen Pfeil auf der rechten Seite.

Listing 6.4: Callbackfunktion des *WriteToFile*-Blocks

```
def block_write_to_file(args):
  try:
    f = open(args["file"], "w+")  # Datei oeffnen
    f.write(args["text"])         # Text in datei schreiben
    f.close()                     # Datei schliessen
    # Erfolgsfall: Nachfolger ist "next"
    return args["next"]
  except Exception:
    # Fehlerfall: Nachfolger ist "nextError"
    return args["nextError"]
```

Decisions

Decision-Blöcke dienen zur bedingten Änderung des Programmflusses (vgl. Abschnitt 5.2.1). Die Implementierung erfolgt analog zur Fehlerbehandlung in funktionalen Blöcken. Ein Unterschied zwischen einem funktionalen Block mit Fehlerbehandlung und einem *Decision*-Block besteht nur im Konzept,

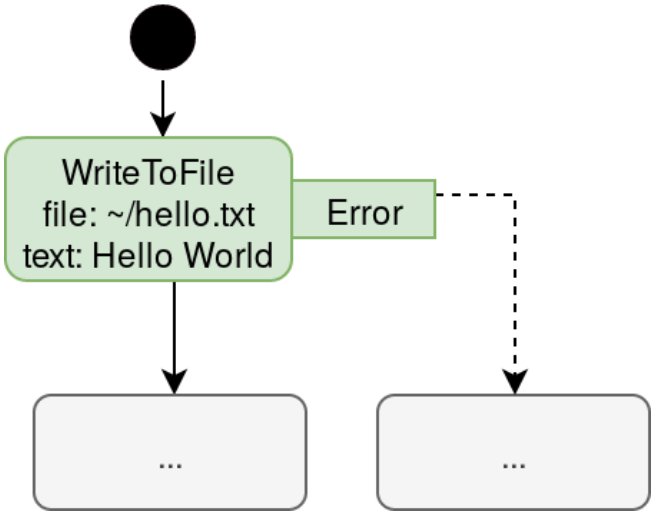

Abbildung 6.6: Alternativer Programmfluss bei einem Fehlerfall

nicht in der Implementierung. Um *Decision*-Blöcke für den Nutzer von anderen klar unterscheidbar zu machen, wurde die Konvention eingeführt, deren Typbezeichnung mit dem Präfix *Decision* zu kennzeichnen (z.B. *DecisionRandom*, *DecisionTopicActive*). Nachfolgend ist die Implementierung (vgl. Listing 6.6) und Benutzung (vgl. Listing 6.5) eines *DecisionRandom*-Blocks aufgeführt. Dieser entscheidet zufällig, welcher der beiden Nachfolger gewählt wird. Die Wahrscheinlichkeit für die Wahl des Nachfolgers *nextIfTrue* ist über den Parameter *chanceForTrue* einstellbar.

Listing 6.5: json Definition des *DecisionRandom*-Blocks

```
"b2": {                          // ID des Blocks
  "type": "DecisionRandom", // Typ des Blocks
  // Wahrscheinlichkeit fuer Nachfolger "nextIfTrue"
  "chanceForTrue": ".3",
  "nextIfTrue": "b5",  // Nachfolger im Fall "True"
  "nextIfFalse": "b3"  // Nachfolger im Fall "False"
},
```

In der Missionsdatei werden dem Block Parameter für die Wahrscheinlichkeit für den Nachfolger *nextIfTrue* sowie die Blöcke für die beiden Nachfolger angegeben.

Listing 6.6: Callbackfunktion des *DecisionRandom*-Blocks

```
def block_decision_random(args):
```

```
# Falls Zufallszahl kleiner als "chanceForTrue"
if random.random() < float(args['chanceForTrue']):
  # Nachfolger ist "nextIfTrue"
  return args['nextIfTrue']
else:
  # Nachfolger ist "nextIfFalse"
  return args['nextIfFalse']
```

Die Implementierung des Blocks generiert eine Zufallszahl zwischen 0 und 1. Liegt diese unter dem Wert des Parameters *chanceForTrue*, wird der im Parameter *nextIfTrue* definierte Nachfolger zurückgegeben. Andernfalls wird der in *nextIfTrue* angegebene gewählt.

6.2.2 Erweiterungen über Plugins

Um den Funktionsumfang des MissionPlayers nicht auf einen Anwendungsfall zu beschränken, wurde die Möglichkeit geschaffen, diesen über Plugins zu erweitern. Dabei bestehen Plugins aus einem Dictionary mit Blocktypen als Schlüssel und Callbackfunktionen als Werten. Das Dictionary wird im MissionPlayer geladen, welcher dadurch die Möglichkeit erhält, diese Blocktypen auszuführen. Die genaue Funktionsweise des Pluginsystems wird in nachfolgend beschrieben. Außerdem werden die beiden Erweiterungen für die ROS-Integration und die Synchronisierung des Ensembles genauer vorgestellt.

ROS Erweiterung

Die ROS-Erweiterung besteht aus dem Python-Plugin sowie einer Reihe von Shell Skripten, welche die cli-tools (Command Line Interface) von ROS1 und ROS2 ausführen. Diese Skripte werden von den Callbackfunktionen der Blöcke aufgerufen. Damit wird es möglich, die Middlewares ROS1 und ROS2 zu bedienen. Für die Ausführung des Proof of Concepts wurden die folgenden Blöcke implementiert:

- CallServiceROS2
- CallServiceROS1
- StartRoscore
- WaitForActiveTopicROS1
- WaitForActiveTopicROS2
- StartNodeROS2

- LaunchROS1
- StartROS1Bridge
- StartLogging
- StopLogging

Die *CallService*-Blöcke ermöglichen das Aufrufen von Services in ROS1 und ROS2. Hierfür wird aus dem Block heraus ein Shell Script aufgerufen, welches die jeweilige ROS-Umgebung lädt sowie anschließend das entsprechende Kommandozeilentool zum Aufruf des Services startet. In der Callbackfunktion des Blocks wird auf das Enden des Shell Skripts gewartet, bevor der durch die Parameter definierte Nachfolger zurückgegeben wird. Der Prozess des Startens eines Skripts aus der Callbackfunktion heraus wird bei den anderen Blöcken analog verwendet, weshalb nachfolgend nur noch die Funktion der Blöcke erklärt wird. Der *StartRoscore*-Block startet einen roscore Prozess, der in ROS1 zur zentralen Verwaltung aller Nodes und Nachrichten dient. Die *WaitForActiveTopic*-Blöcke existieren in einer Ausführung für ROS1 und ROS2. Diese blockieren den Programmfluss solange, bis in einem Topic Aktivität herrscht. Damit kann beispielsweise auf das Anstecken eines Sensors oder das Fertigstellen der Initialisierung eines Nodes gewartet werden. Für ROS2 wurde mit dem *StartNodeROS2*-Block die Möglichkeit geschaffen, Nodes zu starten. Dabei ist es sowohl möglich, die vorinstallierten Nodes zu nutzen, als auch selbst Entwickelte Nodes aus dem extra dafür angelegten Workspace zu laden. Das ROS1 Gegenstück für den *StartNodeROS2*-Block bildet *LaunchROS1*. Dieser Block ermöglicht das Aufrufen beliebiger in ROS1 verfügbarer Launch files. Zur Kommunikation zwischen ROS1 und ROS2 wird die ROS1 Bridge benötigt, welche über den *StartROS1Bridge*-Block gestartet werden kann. Zum Starten und Stoppen des Loggings werden die Blöcke *StartLogging* und *StopLogging* genutzt. *StartLogging* startet ein Skript, welches das Tool *rosbag2* ausführt und speichert eine Referenz auf diesen Prozess. *StopLogging* terminiert über diese Referenz den Prozess und beendet damit das Logging. Die ausführliche Dokumentation der Blöcke und deren Parameter kann im Detail in den DocStrings im Code des Python-Plugins eingesehen werden.

Swarm Erweiterung

Die Swarm Erweiterung nutzt diverse ROS2 Nodes sowie das ROS2 Nachrichtensystem, um Barrieren zur Synchronisierung der Copter zu realisieren. Das Plugin besteht aus den Blöcken *SyncInit* zum Initialisieren des Verwalters des Barrierensystems und *Barrier* zum Warten an einer Barriere. Diese Blöcke greifen wie im vorherigen Abschnitt beschrieben über Skripte auf selbst

entwickelte ROS2-Nodes zu, welche die eigentliche Synchronisierung implementieren. Außerdem bietet das Plugin noch noch *UserBreakpoint*-Blöcke an, welche den Programmablauf blockieren, bis eine Nutzereingabe durch das Schließen einer auf dem Bildschirm erscheinenden MessageBox erfolgt. Der *SyncInit*-Block startet den ROS2-Node, der die Synchronisierung zentral verwaltet. Dieser muss lediglich einmal pro Ensemble gestartet werden, weshalb es sich anbietet, ihn auf dem Telemetrierechner auszuführen. Der Node verwaltet Informationen über die im Ensemble enthaltenen Teilnehmer und publisht sie. In der Initialisierungsphase meldet sich jeder Teilnehmer durch das Publishen seiner ID in ein vom Verwaltungsnode abonniertes Topic an. Hierfür wird der Node *swarm_sync_client* gestartet, welcher den Prozess automatisiert. Der Verwaltungsnode erstellt damit eine Liste aller im Ensemble vorhandenen Teilnehmer und publisht diese. Durch aufrufen eines vom Verwaltungsnode angebotenen Services wird die Initialisierungsphase beendet und keine weiteren Copter mehr ins Ensemble aufgenommen. Der *Barrier*-Block startet den ROS2-Node *barrier_node*, welcher so lange läuft, bis die durch eine ID spezifizierte Barriere erreicht wurde. Um dieses Verhalten zu erreichen, kommuniziert der Block über Topics mit den *Barrier*-Blöcken der anderen Geräte sowie dem Verwaltungsnode und ermittelt dadurch, wann die Barriere von allen Teilnehmern erreicht wurde. Erst wenn die Barriere erreicht wird und der Node terminiert, hört auch der *Barrier*-Block auf zu blockieren und der Programmfluss wird fortgesetzt. Der Block *UserBreakpoint* pausiert den Programmfluss bis eine auf dem Bildschirm erscheinende MessageBox mit dem *OK* oder *Schließen* Knopf beendet wurde. Damit lässt sich der Programmfluss bis zum Erfolgen einer Nutzereingabe unterbrechen.

Hinzufügen neuer Erweiterungen

Das Missionsplanungsframework wurde generisch gehalten, um es nicht auf Quadcoptermissionen oder ROS zu beschränken. Durch ein Pluginsystem wurde eine Schnittstelle zur Erweiterung des Frameworks geschaffen. Um den Prozess des Hinzufügens eines neuen Plugins zu veranschaulichen und gleichzeitig die Flexibilität des Systems zu demonstrieren, wird nachfolgend ein Plugin für eine Kaffeemaschine mit IoT-Erweiterung vorgestellt. Die IoT-fähigkeit wird durch eine Erweiterung der Maschine um einen Microcontroller wie im Projekt von Thomas Witt (vgl. [88]) beschrieben hergestellt. Das Projekt ermöglicht die Steuerung eines Jura Impressa S95 Kaffeevollautomaten über eine Web API, welche durch das Plugin aufgerufen werden soll. Hierfür wird das eigentliche Plugin in der Datei coffeeplugin.py (vgl. 6.7) erstellt. Dort wird zunächst eine Funktion definiert, die die notwendigen Web-Requests zur Kaffeemaschine tätigt. Als Argument bekommt diese ein Dictionary aller im Block definierten Parameter übergeben. Darin enthalten sind

sowohl die URL der Kaffeemaschine als auch der Nachfolgeblock. Über die Requests-Library von Python wird eine Anfrage an die API der Kaffeemaschine geschickt, wodurch diese einschaltet wird. Nach zehn Sekunden Wartezeit wird ein weiterer Request verschickt. Dieser sorgt für das Kochen einer Tasse Kaffee. Am Ende der Methode gibt diese den im Parameter *next* definierten Nachfolgerblock zurück. Abschließend folgt die Definition der Blöcke in der *blocks*-Variable. Diese Variable ist ein Dictionary mit dem Typ des Blocks als Schlüssel und der Callbackfunktion für den Block als Wert. Die *blocks*-Variable wird später benötigt, um das Plugin im MissionPlayer zu laden. Da die Library nur einen Block enthält, ist auch nur dieser in der Variable definiert.

Listing 6.7: Implementierung des Kaffeemaschinen-Plugins (coffeeplugin.py)

```
import requests
import time

# Callback function fuer Block
def block_make_coffee(args):
  baseurl = args["baseurl"]

  # Turn the machine on
  url="%s/turn_on"%baseurl
  requests.get(url)

  time.sleep(10) # Wait for machine to turn on

  # Make some coffee
  url="%s/make_coffee"%baseurl
  requests.get(url)

  return args["next"]

# Dictionary with Blocktype -> Callback Function
blocks = {
  "MakeCoffee": block_make_coffee
}
```

In der Datei coffeemission.json (vgl. 6.8) wird nach dem *Start*-Block der *MakeCoffee*-Block definiert. Dieser enthält als Parameter die Basis-URL der Kaffeemaschine und in *next* den nachfolgenden *End*-Block. Im Listing wurden *Start*- und *End*-Block zur Vereinfachung weggelassen.

Listing 6.8: Verwendung des MakeCoffee Blocks (Auszug aus coffeemission.json)

```
"b2"{
  "type": "MakeCoffee",
  "baseurl": "http://coffeemaker.local/..."
  "next": "b4"
}
```

Im Listing 6.9 ist das Importieren des Plugins in den MissionPlayer und das Ausführen der Mission aufgezeigt. Nach dem Import des MissionPlayers und des Plugins für den Kaffeevollautomaten, wird das Plugin beim MissionPlayer Registriert. Anschließend wird die vorher definierte JSON Mission ausgeführt.

Listing 6.9: Importieren des Kaffeemaschinen-Plugins (coffeemission.py)

```
#!/usr/bin/env python
import mission_player as mp
import coffeeplugin

# Add plugins
mp.add_plugin(coffeeplugin.blocks)

# Play Mission
mp.play_mission("coffeemission.json")
```

Nach der Definition eines Blocks über eine Callbackfunktion und der Registrierung dieser beim MissionPlayer, lässt sich der Block in der Mission nutzen. Durch dieses Vorgehen lässt sich das Framework um nahezu beliebige Funktionalitäten erweitern.

6.2.3 Verteilen der Missionen

Das Verteilen der Mission erfolgt über ein git-Repository, auf dem alle Missionen zentral gespeichert sind. Nach dem Start lädt der Bordcomputer jedes Copters die Änderungen des Repos herunter. Im Repository befinden sich Ordner mit dem Namen der MAVLink System-ID (MAV-ID) jedes Copters, in denen jeweils eine Datei mit dem Namen mission.json liegt.

Der Vorteil dieses Deployments ist, dass das Kopieren von Missionsdateien auf die einzelnen Copter entfällt und das komplette Ensemble mit einem push in das git-Repo neue Missionen erhält.

Ein Nachteil dieser Verteilungsmethode ist die benötigte Internetverbindung, welche im Outdooreinsatz nur langsam oder überhaupt nicht verfügbar sein kann. Da die Missionen jedoch nur wenige Kilobytes groß sind, lassen sie

sich problemlos über eine mobile Internetverbindung z.B. über ein Smartphone mit WLAN-Tethering herunterladen. Sollte an Orten operiert werden, an denen kein mobiles Internet verfügbar ist, kann auf einen lokal operierenden git-Server der z.B. auf dem Telemetrierechner laufen kann, zurückgegriffen werden. Da jeder Copter seine eigene MAV-ID kennt, kann er anschließend die aktuelle Mission aus seinem Ordner ausführen. In einigen Fällen ist es erwünscht, dass der Copter nicht direkt startet, sondern auf ein manuelles Signal wartet. Dadurch kann ausgeschlossen werden, dass sich noch Bedienpersonal in den Gefahrenbereichen um die Copter herum aufhält. Hierzu führt der Telemetrierechner eine Mission aus, die eine Barriere bis zum Betätigen eines Start Buttons blockiert und an der alle Copter nach der Initialisierung warten.

6.3 Smart-Sensors

Die grundlegende Idee der smarten Sensoren ist die Erweiterung einfacher Sensoren um „smarte“ Funktionen (vgl. 5.3). Im Folgenden wird die konkrete Implementierung einer einheitlichen Hard- und Softwareschnittstelle sowie der Kapselung des Expertenwissens auf dem Sensor im Detail beschrieben. Dabei wird zunächst auf den Hardware- und Softwareaufbau eingegangen, bevor das integrierte Kalibrierungssystem und die Erweiterbarkeit des Systems um neue Sensoren erläutert wird.

6.3.1 Hardwareaufbau Smart-Sensor-Board

Der Hardwareaufbau teilt sich in den eigentlichen Sensor, den Mikrocontroller, die Anzeigen und den Steckverbinder auf. Der Sensor ist austauschbar, jedoch unterstützt das Framework in seinem jetzigen Stand nur die Übertragung von Fließkommawerten. Damit ist die Messung einfacher Messgrößen wie Temperatur, Helligkeit oder Luftfeuchtigkeit problemlos möglich. Komplexere Datentypen wie LIDAR-Punktwolken werden von der Schnittstelle unterstützt, wurden aber aufgrund des größeren Implementierungsaufwandes nicht in die Bibliothek integriert. Die Sensorwerte werden von einem ESP8266 Mikrocontroller-Board ausgelesen und verarbeitet. Dieses übernimmt zudem die Ansteuerung eines Displays, welches die Sensordaten sowie Zusatzinformationen zum Sensor anzeigt. Ebenso geschieht dort die Verrechnung des Sensors mit vorher ermittelten Kalibrierungsdaten. Nach der Korrektur werden die Werte über eine modifizierte USB-C Schnittstelle an eine ROS2 Installation, die auf dem MissionController eines Copters läuft, weitergereicht. In den folgenden Abschnitten werden die hier angesprochenen Features im Detail erklärt.

Kompatible Mikrocontroller

Die Auswahl des Mikrocontrollers ist durch ros2arduino (vgl. [70]), die verwendete Software-Library zur Kommunikation mit ROS2 und kompatible Boards eingeschränkt. Zu den offiziell unterstützten Boards zählen diverse STM32F103-basierte Entwicklungs-Boards des Herausgebers der Library (OpenCR, OpenCM9.04 (vgl. [69]), der Arduino MKR ZERO (vgl. [8]), der Arduino DUE (vgl. [7]) und der ESP32 (vgl. [38]). Nach eigenen Tests erwies sich der ESP8266 (vgl. [17] als ebenfalls kompatibel. Ebenso getestet wurden der Arduino Nano, welcher durch die schwache Leistung jedoch nur maximal einen Publisher erstellen kann und danach nur noch wenige Leistungsreserven hat, sowie der Arduino Mega 2560, welcher eine Modifikation der Library benötigte (vgl. [71]).

Aufgrund des niedrigen Preises von weniger als 3 € und der geringen Größe im Vergleich zu den teureren und teils größeren, offiziell unterstützten Boards wurde das ESP8266-basierte ESP-12F Board (vgl. [17]) für die Smart-Sensors ausgewählt. Auf dem von der Firma Ai-Thinker vertriebenen Board ist ein 32-Bit ESP8266 Prozessor mit einer Taktrate von 80MHz verbaut. Die Programmierung und Kommunikation mit dem MissionController erfolgt über einen seriellen Port am Board, der über einen USB-Serial-Chip auf eine USB-Schnittstelle umgesetzt und mit dem Controller verbunden werden kann. Um die Verbindung mit dem PC zu vereinfachen existieren Entwicklungs--Boards, auf denen ein ESP-12F Board zusammen mit einem 5V zu 3.3V Spannungswandler sowie ein USB-Serial Adapter mitsamt USB-Buchse verbaut ist. Eins dieser Boards stellt das NodeMCU Devkit V3 (vgl. [24]) dar, welches für Revision 1 des Hardware-Boards eingesetzt wurde und dessen Schaltplan als Basis für die in Revision2 entworfene Platine diente. Weitere Schnittstellen des Controllers sind unter anderem:

- IEEE 802.11 b/g/n WLAN (2,4 GHz)
- SPI
- I2C und I2S
- bis zu 12 Digitale I/O Pins
- ein 12-Bit Analog zu Digital Umsetzer
- 1-Wire
- 9x PWM (Pulse Width Modulation) Pins

Ein weiteres Feature ist die Unterstützung des Microcontrollers für das Arduino-Ökosysten. Somit können die zahlreichen von der Community erstellten

Softwarebibliotheken für viele Sensoren verwendet und dadurch der Implementierungsaufwand reduziert werden.

Display und LEDs

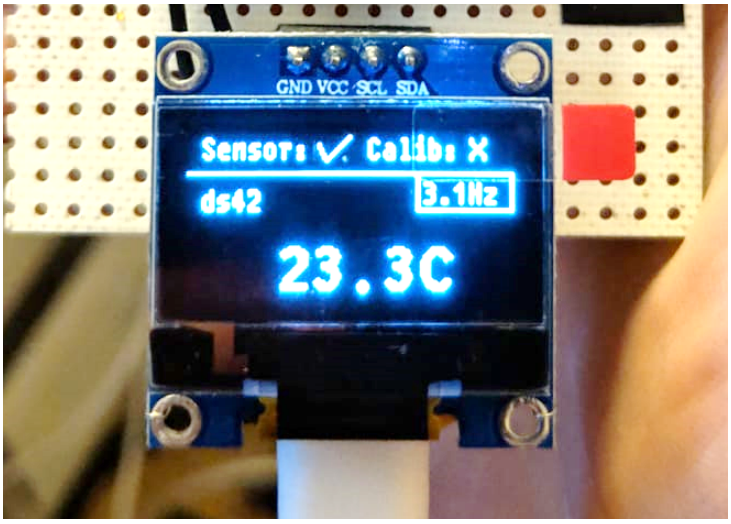

Abbildung 6.7: OLED Display mit Informationen über Sensorwert und Status der Kalibrierung

Zur Anzeige von relevanten Informationen zum Sensor wurde ein 0,96ÖLED Display mit I2C Schnittstelle gewählt. Dieses ist im Vergleich zu LCD Displays auch bei Dunkelheit und starker Sonneneinstrahlung gut lesbar. Durch die Architektur von I2C können weitere Endgeräte mit der Schnittstelle verbunden werden und das Display belegt sie nicht exklusiv. Abbildung 6.7 zeigt die nach dem Verbinden mit dem MissionController dargestellten Werte. In der obersten Zeile wird die Plausibilität des Sensorwerts sowie der Status der Kalibrierung durch einen Haken (für „Alles OK“) oder ein X (für „Fehler“) dargestellt. Die Plausibilitätsprüfung ermöglicht es dem Benutzer, einen defekten Sensor frühzeitig zu erkennen. Dabei kann der Integrator des Sensors mithilfe seines Expertenwissens selbst entscheiden, wann ein Sensorwert plausibel erscheint. Dies kann z.B. realisiert werden, indem geprüft wird, ob der Sensorwert in einem bestimmten Wertebereich liegt. Der Indikator für die Kalibrierung zeigt in der aktuellen Version lediglich an, ob eine Kalibrierung durchgeführt wurde. Denkbar ist jedoch, dass eine Kalibrierung nach einer bestimmten Anzahl von Starts oder einer gewissen Zeitspanne ihre Gültigkeit verliert. Die Starts könnten über einen Counter im integrierten EEPROM-Speicher des Chips mitgezählt und die Zeit über WLAN von einem NTP Server abgefragt werden. Aufgrund des Implementierungsaufwands

wurden diese Konzepte jedoch nicht umgesetzt. In der zweiten Zeile ist die ID des Sensors eingeblendet, mit dem dieser in ROS2 und den späteren Logs identifiziert werden kann. Wird nach mehreren Messungen festgestellt, dass der Sensor mit einer bestimmten ID fehlerhafte Werte liefert, können diese Werte auch nach der Messung zurückverfolgt und bei der Auswertung ignoriert werden. Neben der ID findet sich die Frequenz, in welcher der Sensor Werte an das System sendet. Darunter wird der eigentliche Messwert des Sensors sowie dessen Einheit angezeigt. Sollte der Sensor mehrere Messgrößen ermitteln, können diese durch Betätigen eines Knopfes auf dem Board durchgeschaltet werden. Zusätzlich sind auf den Boards zwei LEDs verbaut, die den Status der seriellen Kommunikation zum MissionController in Sende- und Empfangsrichtung melden. Mit ihnen kann schnell erkannt werden, ob der Sensor mit dem MissionController kommuniziert oder ob Verbindungsprobleme vorliegen.

Steckverbinder

Bei den Steckverbindern der Smart-Sensor-Boards handelt es sich um modifizierte USB-C Stecker und Buchsen. Diese wurden gegenüber anderen USB-Steckertypen bevorzugt, da sie durch ihren punktsymmetrischen Aufbau in zwei Orientierungen eingesteckt werden können. Über die am Smart-Sensor-Board angebrachte Buchse erfolgt sowohl die Kommunikation mit dem MissionController als auch die Programmierung am PC. Um beim Programmieren keinen speziellen Stecker zu benötigen, wurde darauf geachtet, dass der Steckverbinder weiterhin kompatibel zu standardisierten USB-C Kabel ist. Eine wesentliche Änderung ist das Hinzufügen von Magneten links und rechts von Stecker und Buchse, welche verhindern, dass sich der Sensor im Flug durch auftretende Vibrationen und Beschleunigungskräfte löst. Die verwendeten Neodym-Magnete bieten eine ausreichende Haltekraft, um kleine Sensoren sicher am Copter zu befestigen. Sollten schwerere Sensoren benötigt werden, wäre eine Erweiterung des Steckers um zusätzliche gefederte Plastikclips denkbar. Abbildung 6.8 zeigt den Steckverbinder jeweils als Buchse und Stecker. USB-C Buchse bzw. Stecker sitzen in der Mitte und sind von einem 3D-gedruckten Plastikteil umgeben, welches auch die Magnete hält. Dieses Plastikteil ist am Stecker mit dem Copter und am Sensor mit der Platine des Sensor-Boards verbunden.

Smart-Sensor-Board Rev1

Die erste Version des Smart-Sensor-Boards wurde auf einer Lochrasterplatine umgesetzt (vgl. 6.9). Um den Anschluss des ESP-12F Boards zu erleichtern, wurde das NodeMCU Devkit V3 (vgl. [24]) genutzt, welches eine 3,3 V Strom-

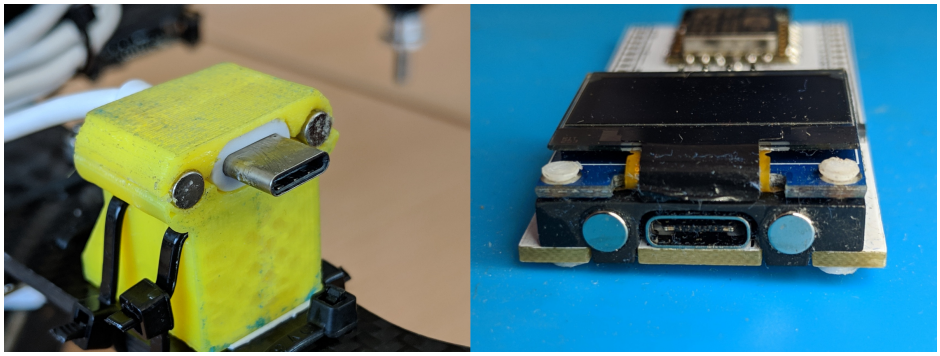

Abbildung 6.8: Steckverbinder des Smart-Sensor-Boards als Stecker am Quadcopter montiert (links) und als Buchse am Board (rechts)

versorgung sowie einen USB auf Seriell Wandler mitbringt. Dieses wird eigentlich mit LUA Skripten programmiert, aufgrund der höheren Performance und Anforderungen wurde in dieser Arbeit mit der ros2arduino Library mit C++ und der Arduino IDE gearbeitet. Da das Board jedoch statt der benötigten USB-C Buchse eine Micro-USB Verbindung besitzt, wurde zusätzlich das Pololu USB-C Breakout Board (vgl. [25]) verbaut. Die GND und VBUS Leitungen des Breakout-Boards wurden mit den GND und VIN Pins des NodeMCU Boards verbunden. Die Datenverbindungen für Data Plus und Data Minus des Breakout Boards mussten direkt an den Pins des CH340G USB-Serial Converter Chips auf dem NodeMCU Board angelötet werden. Das Breakout-Board wurde mithilfe eines Geradschleifers so bearbeitet, dass die Plastikhalterung für die Magnete angebracht werden konnte. Oberhalb des Steckverbinders ist das 0,96 Zoll OLED Display (vgl. [15]) verbaut, welches über I2C mit dem NodeMCU Board verbunden ist (s. Abbildung 6.9). Es ist über einen dünnen Draht mit dem Steckergehäuse verbunden, um ein Abfallen während des Fluges zu verhindern. Zusätzlich sind eine rote und eine grüne 5mm LED zur Anzeige von Aktivität auf der seriellen Verbindung in Empfangs- und Senderichtung angebracht. Diese wurden mit einem 1 kOhm Vorwiderstand versehen und direkt an die RX und TX Boards des NodeMCU Boards angeschlossen. Die Stromversorgung an den TX-Pins des NodeMCU boards und am CH340G Chip reichen aus, um die LEDs zu betreiben, ohne dass die serielle Kommunikation dadurch gestört wird. Durch den Aufbau auf der Lochrasterplatine ist es leicht möglich beliebige Sensoren auf dem Board zu integrieren. Nach Anbringen des Sensors können die benötigten Verbindungen zum NodeMCU Board verlötet werden.

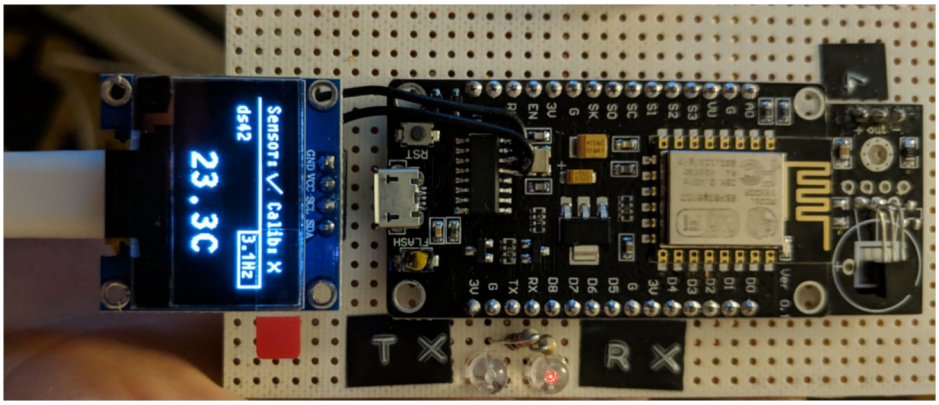

Abbildung 6.9: Aufbau des Smart-Sensor-Boards Rev1 auf einer Lochrasterplatine

Smart-Sensor-Board Rev2

Diese modulbasierte Bauweise des Rev1 Boards ist mit einfachen Mitteln und gut verfügbaren Teilen umzusetzen. Allerdings ist das resultierende Board groß und verhältnismäßig schwer für den Einsatz an Quadrocoptern mit beschränkter Nutzlast. Die USB-Verbindung mit dem CH340G Chip ist nicht optimal gelöst und das aus mehreren Modulen zusammengestellte Board wirkt unprofessionell. Aus diesem Grund wurde eine zweite Version mit einer speziell für den geplanten Einsatzzweck entworfenen Platine angefertigt. Die Basis des Designs ist der Schaltplan des NodeMCU Devkit in der Version 1.0 (vgl. [86]). Weil das QFN-Package (Quad Flat No Leads-Package) des dort verbauten CP2102 USB-Serial Converters sich nur schlecht mit einem konventionellen Lötkolben verbauen lässt, wurde dieser durch ein CH340G Modul, wie es im NodeMCU V3 vorhanden ist, ersetzt. Zusätzlich wurden die 5mm LEDs für die RX/TX Anzeige durch kleinere SMD LEDs im Formfaktor 0603 ausgewechselt und direkt auf das Board integriert. Durch die unterschiedlichen Farben der LEDs (Rot für RX und Grün für TX), sind diese trotz der wesentlich kleineren Gehäuse noch gut voneinander unterscheidbar. Der Micro-USB Verbinder wurde durch eine USB-C Buchse ersetzt und Aussparungen für die Magneten auf dem Board angebracht. Der I2C Bus und die Stromversorgung für das OLED Display wurden auf dem Board herausgeführt, um das Display direkt verlöten zu können. Außerdem wurden Bohrungen angebracht, um das OLED und Plastikteil des Steckverbinders fest mit dem Board verschrauben zu können. Dies kann über M2 Schrauben und Muttern geschehen oder indem aus PLA-Filament für 3D-Drucker Niete hergestellt werden, welche OLED, Plastikteil und Platine miteinander verbinden (vgl. [4], Abbildung 6.8). Abbildung 6.10 zeigt einen Render der erstellten Platine. Bei dem Entwurf des Boards wurde darauf geachtet, die

Pinkompatibilität mit dem NodeMCU V3 Board zu erhalten. Es ist angedacht, den eigentlichen Sensor auf einer kleinen Lochrasterplatine auf der Rückseite oder über dem Board anzubringen. Dadurch kann die Abmessung des Sensor-Boards im Vergleich zur Revision 1 von ca. 110x55x25 mm auf 74x31x30 mm reduziert werden. Das Gewicht eines Smart-Sensors mit einem DS18B20 Temperatursensors verringerte sich von 24 g in der Revision 1 auf 16 g. Durch die starke Orientierung an den NodeMCU Boards ist die Revision

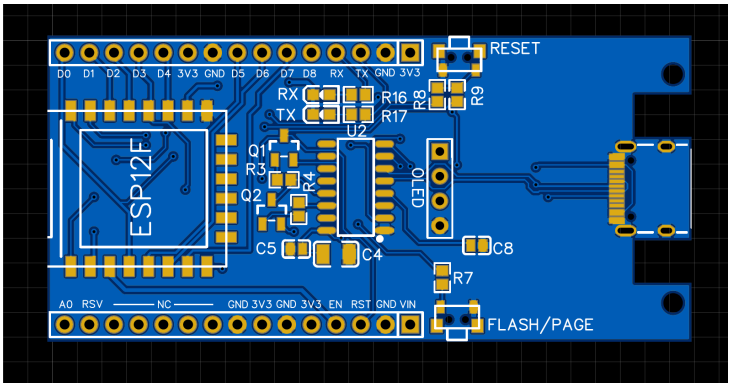

Abbildung 6.10: Platinenlayout des Smart-Sensor-Board Rev2

2 des Smart-Sensor-Boards kompatibel mit der Arduino IDE und dem NodeMCU-Boardpreset. Wird die Arbeitszeit für Entwicklung und Bestückung der Platine nicht betrachtet, ähnelt der Preis der speziell angefertigten Sensor-Boards dem der NodeMCU Boards. Dafür ist die Teilebeschaffung und der Zusammenbau anspruchsvoller und zeitaufwändiger als bei Revision 1.

6.3.2 Softwarearchitektur

In diesem Abschnitt wird die Softwarearchitektur für die Smart-Sensor-Boards vorgestellt und in diese die eingesetzten Technologien verortet. Abbildung 6.11 veranschaulicht den Softwareaufbau eines mit dem MissionController verbundenen Smart-Sensors. Auf dem Sensor übernimmt die Smart-Sensor-Library die Kommunikation mit dem MissionController und verwaltet die Kalibrierungsdaten. Die Software des Smart-Sensors nutzt dabei die ros2arduino Library (vgl. [70]), welche die Kommunikation zwischen Mikrocontroller und ROS2 herstellt. Diese verwendet den Micro-XRCE-DDS-Agent (vgl. [72]), ein Kommunikationsframework, welches Geräte mit beschränkten Ressourcen über das DDS-XRCE (DDS for eXtremely Resource Constrained Environments) an bestehende DDS-Infrastruktur anbindet. Die ros2arduino Library benutzt den Micro-XRCE-DDS-Agent, um mit dem DDS-basierten ROS2 Nachrichtensystem zu kommunizieren. Dieser Prozess wird im Ab-

schnitt 6.3.2 im Detail erklärt. Will ein Entwickler einen neuen Sensor integrieren, kann dieser die Smart-Sensor-Library nutzen und somit den Implementierungsaufwand für die Kommunikation mit dem MissionController reduzieren. Es muss lediglich der eigentliche Sensor ausgelesen und die Messwerte an die Library übergeben werden.

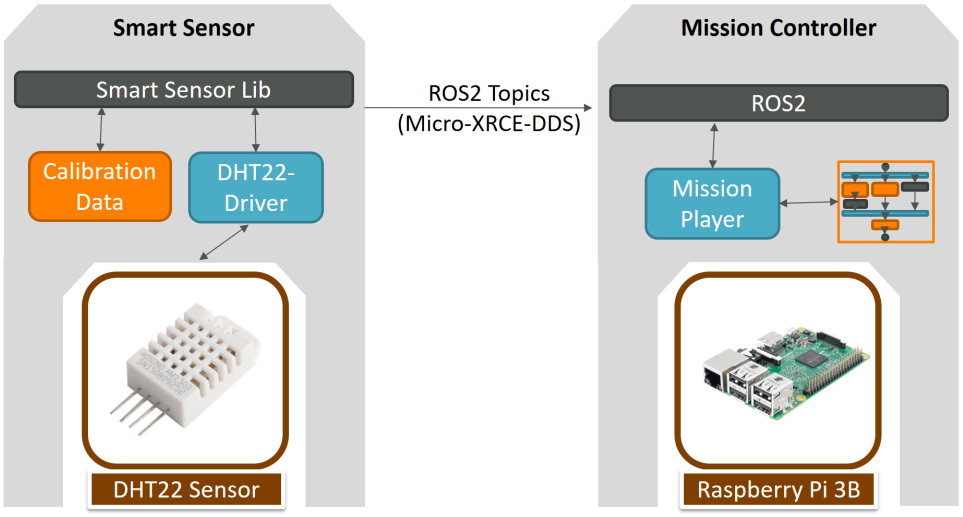

Abbildung 6.11: Softwarearchitektur des Smart-Sensors

ROS2 Anbindung

Zum Publishen eines Sensorwerts sendet der Smart-Sensor diesen über die serielle Schnittstelle an den ROS2-Rechner. Dort läuft der Micro-XRCE-DDS-Agent und nimmt Daten im XRCE-DDS-Format auf dem Port entgegen. Diese Daten wandelt der Agent in DDS-Pakete um und publisht sie stellvertretend für den Sensor. Das Empfangen von Nachrichten über einen Subscriber auf dem Mikrocontroller ist ebenfalls möglich. Um neue Sensoren automatisch ins System einzubinden, wurde ein Watchdog erstellt, welcher neue serielle Geräte auf dem MissionController automatisch erkennt und einen entsprechend konfigurierten Micro-XRCE-DDS-Agent startet. Dadurch werden vom Sensor zur Verfügung gestellte Sensorwerte automatisch in ROS2 verfügbar. Der Sensor publisht seine Messwerte in einem Topic, dessen Namensgebung einem definierten Konvention entspricht. Der Name des Topics richtet sich dabei nach dem folgenden Schema:

`/mav*MAV-ID*/sensor/*TYPE*/*DEVICE*/*ID*`

Dabei beschreibt **MAV-ID** die MAVLink System-ID des Copters, welche diesen eindeutig im Ensemble identifiziert. Diese teilt der Copter dem Sensor

in der Initialisierungsphase mit. Anstelle von **TYPE** wird die vom Sensor gemessene Messgröße eingesetzt. Beispielwerte hierfür wären *temperature*, *humidity* oder *distance*. Der Platzhalter **DEVICE** steht für die Bezeichnung des Sensors. Hierfür wird der Name des Sensors konform der Richtlinien für ROS-Topic Namen (vgl. [81]) angegeben um Messwerte von Sensoren verschiedenen Typs unterscheiden zu können. Für den DS18B20 Temperatursensor der Firma Maxim Integrated (vgl. [18]) wurde beispielsweise der Wert *DS18B20* gewählt. Das Feld **ID** schließt den Namen des Topics ab. Jeder Smart-Sensor ist mit einer eindeutigen ID ausgestattet, welche er auf seinem Display anzeigt. Sollte der Defekt eines Sensors nach einer Reihe von Messflügen festgestellt werden, so können die dazugehörigen Messwerte in den Aufzeichnungen der Messungen über diese ID eindeutig identifiziert und nicht in die Auswertung mit aufgenommen werden. Hiermit kann möglicherweise verhindert werden, dass eine komplette Reihe von Messungen unbrauchbar wird. Im MissionPlayer ist es möglich, Variablen zu nutzen um die copter-spezifische ID in den Missionen automatisiert zu ersetzen. Außerdem ist das Nutzen von regulären Ausdrücken (vgl. [89]) in einigen Parametern möglich. Soll in der Mission beispielsweise auf das Anstecken eines beliebigen Temperatursensors am eigenen Copter gewartet werden, so kann ein *WaitForActiveTopicROS2*-Block in der Missionsdatei wie im Listing 6.10 definiert werden.

Listing 6.10: Definition eines WaitForActiveTopic Blocks mit Wildcards

```
"b2"{
  "type": "WaitForActiveTopicROS2",
  "name": "\/$mavid\/sensor\/temperature\/"
  "next": "b4"
}
```

Die Variable $mavid wird hier durch die MAV-ID des Copters, auf dem die Mission ausgeführt wird ersetzt. Da die Auswertung des dadurch entstehenden regulären Ausdrucks arbiträre Endungen zulasst, wird der Programmfluss nach Anstecken eines beliebigen Temperatursensors fortgesetzt.

Einen Sonderfall bei der Namensgebung bilden die Referenzsensoren für die Kalibrierung. Diese publishen ihre Werte in ein Topic mit dem Namen

```
/calibration/*TYPE*
```

, wobei **TYPE** auch hier wieder für die Messgröße des Sensors steht. Jeder Nicht-Referenzsensor erstellt einen Publisher auf das Kalibrierungstopic seiner Messgröße und startet die Kalibrierung, sobald auf diesem Topic Werte gesendet werden (vgl. 6.3.3). Ob ein Sensor ein Referenzsensor ist, kann über ein *define* Statement im Code des Sensors zum Zeitpunkt des Kompilierens festgelegt werden.

6.3.3 Kalibrierung

Dieser Abschnitt beschreibt die Architektur und Arbeitsweise der Kalibrierungsfunktion der Smart-SensorLibrary. Dabei wird zunächst der Ablauf der Kalibrierung Beschrieben bevor auf die konkrete Implementierung der Funktion eingegangen wird. Der Kalibrierungsprozess läuft - wie bereits im Abschnitt 5.3.4 beschrieben - größtenteils im Smart-Sensor ab. Dennoch wird eine Kalibrierungsstation benötigt, welche aus einem Rechner mit ROS2 Installation und einem Referenzsensor besteht. Dafür muss die Möglichkeit geschaffen werden, verschiedene Messwerte bei den Sensoren messen können. Dies kann z.B. eine Klimakammer bei Temperatursensoren oder eine dimmbare Lichtquelle bei Lichtsensoren sein. Die Smart-Sensor-Library erstellt standardmäßig einen Subscriber für Kalibrierungdaten auf den Messwerttyp des Sensors (z.B. auf das Topic

`/calibration/temperature`

bei Temperatursensoren). Der Name des Topics für die Kalibrierungsdaten setzt sich dabei aus dem präfix *calibration* und der gemessenen Größe (z.B. *temperature* oder *humidity*) zusammen. Sobald ein Referenzsensor auf diesem Topic Referenzwerte publisht, beginnt auf dem Sensor der Kalibrierungsprozess. Die Library speichert beim Eintreffen eines Referenzwerts für eine Messgröße, die der Sensor selbst erfasst, den selbst gemessenen Wert und die Abweichung vom Referenzwert. Sobald der Referenzsensor keine weiteren Werte publisht oder wenn 500 dieser Wertepaare aufgenommen wurden, beginnt die Auswertung. Dabei wird auf dem Smart-Sensor mittels linearer Regression eine Korrekturkurve errechnet (vgl. 6.12). Diese Kurve liefert für jeden gemessenen Wert einen geschätzten Messfehler, der von dem Messwert abgezogen werden kann, um ein präziseres Ergebnis zu erreichen. Da sich die lineare Regression zur Erstellung der Korrekturkurve einiger der im Proof of Concept genutzten Sensoren gut eignet, wurde diese Methode gewählt. Denkbar wäre zudem, dem Integrator verschiedene generische Methoden zum Erstellen einer Korrekturkurve anhand der Fehlerdaten zur Verfügung zu stellen. Die Parameter der Korrekturkurve werden abschließend im EEPROM des ESP8266 Boards gespeichert, wodurch sie auch nach dem Neustart des Sensors verfügbar sind. Nach erfolgreicher Kalibrierung publisht der Sensor lediglich kalibrierte Werte. Ein Publishen der rohen Sensordaten oder des Zustands der Kalibrierung wäre hilfreich, um in den Logdateien zwischen kalibrierten und nicht kalibrierten Sensoren zu unterscheiden. Dies wurde allerdings aus Ressourcengründen nicht implementiert. Jeder Publisher benötigt einen verhältnismäßig großen Anteil des verfügbaren RAMs. Durch zusätzliches Übertragen der Rohdaten und des Zustands der Kalibrierung würde die Anzahl der verschiedenen übertragbaren Messgrößen pro Smart-Sensor auf

die Hälfte oder ein Drittel verringert. Da die Messung von Daten gerade bei kostengünstigeren Sensoren nur mit einem kalibrierten Sensor aussagekräftig ist, wird davon ausgegangen, dass nur kalibrierte Sensoren in einem Messflug verwendet werden. Da der Zustand der Kalibrierung über ein Symbol auf dem OLED Display angezeigt wird (vgl. Abschnitt 6.3.1), kann direkt beim Einstecken überprüft werden, ob ein kalibrierter Sensor verwendet wird.

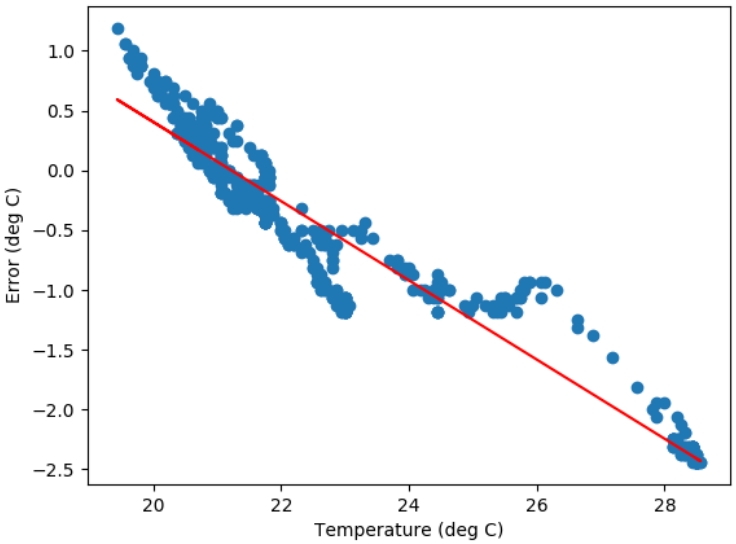

Abbildung 6.12: Messfehler (Blau) und Korrekturkurve (Rot) eines DS18B20 Sensors bei verschiedenen Temperaturen

6.3.4 Integrieren eines neuen Sensors

Für das Hinzufügen eines neuen Sensors muss ein Entwickler den Sensor mit dem Sensor-Board verbinden. Dafür stehen ihm die im Abschnitt 6.3.1 genannten Schnittstellen zur Verfügung. Durch die Kompatibilität des Rev1 und Rev2 Boards mit dem 2,54 mm Pinabstand, kann dies sowohl auf einer Lochrasterplatine als auch auf einem handelsüblichen Breadboard erfolgen. Softwareseitig muss die Smart-Sensor-Library eingebunden, Objekt der Klasse SmartSensor erstellt und mit Angaben zum Sensor initialisiert werden. Anschließend wird entweder der sensorspezifische Code für die Anbindung an den Mikrocontroller entwickelt oder eine entsprechende Library installiert. Durch die Nutzung der Arduino Bibliotheken existiert bereits eine Hardwareabstraktionsschicht (Hardware Abstraction Layer, HAL) für alle gängigen Schnittstellen. Zudem kann durch die Manipulation von CPU-Registern auf

Schnittstellen wie UART und SPI über vorgefertigte Bibliotheken zugegriffen werden. Ebenso werden Bibliotheken oder Beispielcode für das Auslesen vieler Sensoren durch die Community bereitgestellt und laufend aktualisiert. Dadurch konnte für zwei der drei im Proof of Concept in die Library integrierten Temperatursensoren konnte auf fertige Bibliotheken zurückgegriffen werden. Für den dritten Sensor fand sich ein Beispielcode, der durch Nutzung einer der Arduino-HAL-Librarys ein Auslesen des Sensorwerts in zehn Zeilen Code ermöglichte. Ein aufwändiges Studieren von Datenblättern und Schreiben eigener Treiber entfällt somit in den meisten Fällen. Nach dem Auslesen der Sensorwerte müssen diese lediglich an das SmartSensor-Objekt weitergegeben und die *tick()*-Methode des Objekts in der Hauptschleife periodisch aufgerufen werden. Diese Methode ist verantwortlich für die Kommunikation mit ROS, die Kalibrierung sowie die Ansteuerung des Displays. Nach der Initialisierung sind vom Nutzer keine weiteren Aktionen mehr erforderlich, um diese Funktionen zu nutzen.

6.4 Bodenkontrollstation

Um die Mission zu überwachen und zu koordinieren, wird ein Telemetrierechner eingesetzt, welcher die folgenden Aufgaben erfüllt:

- Erstellen, modifizieren und verteilen der Mission
- Starten der Mission durch Startfreigabe
- Diagnose während des Fluges
- Zentrales Sammeln von Sensor- und Telemetriedaten

Hierfür wurde auf dem Rechner eine Ubuntu 18.04 Installation mit ROS2 in der Version Dashing aufgesetzt. Der Rechner ist mit demselben WLAN wie die MissionController der Quadcopter verbunden, was die Kommunikation der verschiedenen ROS2-Installationen ermöglicht.

6.4.1 Logging

Das Logging der Messwerte erfolgt zentral auf dem Telemetrierechner. Dabei werden über das in ROS2 integrierte Tool *ros2 bag* (vgl. [28]) sämtliche im System gepublishten Nachrichten mitsamt ihrem Zeitstempel aufgezeichnet. Durch die Architektur von DDS ist es problemlos möglich, die Daten der Copter auf dem Telemetrierechner zu empfangen und zu speichern. Das Logging kann über den Befehl

```
ros2 bag record --all
```

gestartet werden. Dadurch wird der komplette Nachrichtenverkehr in einer .bag Datei gespeichert. Das Starten und Stoppen des Loggings kann ebenso über einen entsprechenden Block in der Mission des Telemetrierechners erfolgen. Durch den Befehl

```
ros2 bag play LOGDATEI.bag
```

kann der aufgezeichnete Flug wieder „abgespielt“ und dabei analysiert und ausgewertet werden.

Das zentrale Sammeln von Logdaten macht das nachträgliche Zusammentragen der Daten der einzelnen Copter überflüssig. Zudem sind die gesammelten Daten mehrerer Copter bereits in einem Format gespeichert, dass das gemeinsame Plotten und Auswerten der Daten mehrerer Copter ohne manuelles Zusammenfügen von Messwerten ermöglicht. Dadurch, dass jeder Messwert mit einem Zeitstempel versehen ist, ist auch die Auswertung vom Messwert abhängig von der Position der Copter oder anderen Werten möglich.

6.4.2 Diagnose und Telemetrie

Zur Diagnose während des Fluges können die in ROS2 vorhandenen Tools wie das 3D-Visualisierungstool RViz (vgl. [26]) oder das umfangreiche Kommandozeileninterface genutzt werden. RViz ermöglicht die Anzeige der Position der Copter in einer 3D-Visualisierung. Möglich wäre außerdem ein Plugin für die Visualisierung, welches zusätzliche Informationen zu den Coptern wie z.B. Akkustand und Sensorwerte direkt an einer 3D-Repräsentation des Copters anzeigt. Aufgrund des hohen Implementierungsaufwands wurde dies in der Arbeit allerdings nicht umgesetzt. Die als Topics verfügbaren Sensorwerte und Statusinformationen lassen sich jedoch über die ROS2 Kommandozeilentools ausgeben.

In diesem Kapitel wurde somit ein System erarbeitet, welches durch eine modulare Missionsplanung die flexible Ausführung von Missionen in Roboterensembles sowie die Synchronisierung der einzelnen Roboter ermöglicht. Dieses System ist an die Smart-Sensors angebunden, welche über eine einheitliche Schnittstelle und das Bereitstellen einer Bibliothek, die Integration verschiedenster Sensoren an das Gesamtsystem ermöglicht. Zuletzt wurde eine Bodenkontrollstation vorgestellt, welche die Überwachung von Missionen sowie die zentrale Akquise von Sensordaten ermöglicht.

7 Proof of Concept

Um die Funktion des Systems zu belegen, wurde mit diesem ein konkretes Szenario getestet. Dabei wurde sich an der in Abschnitt 3.1 bereits kurz vorgestellten ScaleX 2015-Mission orientiert. Nachfolgend wird die Architektur des Proof of Concept-Systems behandelt, welches eine weiterentwickelte Form der virtuellen Messtürme von ScaleX 2015 darstellt. Anschließend wird auf den genauen Aufbau der einzelnen Hardwareelemente eingegangen. Dazu zählen die eingesetzten Copter, die implementierten Sensoren die Bodenkontrollstation und der Kalibrierungsaufbau. Der letzte Abschnitt behandelt die Ergebnisse der Ausführung der Mission.

7.1 Systemarchitektur

Für das Proof of Concept wurden der grundlegende Aufbau, wie er im Abschnitt 6.1 beschrieben ist übernommen. Die Messungen wurden wie in ScaleX 2015 (vgl. 3.1) mit drei Quadcoptern durchgeführt. Jedoch wurden statt den vorher verwendeten Copter vom Typ Saphira von der Firma rOswewhite Multicopters drei Copter Typ Rachel des selben Herstellers verwendet (vgl. [99]). Jeder der Copter ist mit einem Raspberry Pi 3B ausgestattet, welcher als MissionController fungiert. An den MissionControllern befindet sich jeweils ein DS18B20- und DHT22-Smart-Sensor zum Messen der Temperatur. Neben den Coptern wird für den Versuchsaufbau eine Bodenkontrollstation benötigt. Diese besteht aus einem Laptop, einem WLAN Access Point zur Vernetzung der Copter und des Laptops untereinander sowie einer Funkfernbedienung, mit der die Quadcopter im Notfall gelandet werden können. Der Laptop übernimmt das zentrale Loggen der Positions- und Sensordaten, das Erteilen der Startfreigabe und ermöglicht die Anzeige der Copterpositionen und Messwerte.

Um möglichst exakte Werte von den Sensoren zu erhalten, werden diese vor Ausführung der Mission in einer Kalibrierungsstation kalibriert. Hierzu wird ein präziser Referenzsensor genutzt, welcher eine „ground truth“ für die Temperatur liefert. Dieser wird genauso wie der zu kalibrierende Sensor mit einem Mission Controller verbunden, wodurch die Kalibrierung selbstständig startet. Der gesamte Aufbau muss in einer Umgebung stattfinden, die während der Kalibrierung langsam ihre Temperatur variiert. Hierzu wurde ein

handelsüblicher Kühlschrank gewählt. Der genaue Ablauf der Kalibrierung wird im Abschnitt 7.6 genauer erklärt.

7.2 Erstellte Missionen

Die für das Proof of Concept entwickelte Beispielmission ist dem Aufbau der ScaleX 2015 nachempfunden (vgl. Abschnitt 3.1). Jeder Copter erhält dabei dieselbe Missionsdatei, deren Aufbau in Abbildung 7.1 im Detail dargestellt ist. Zu Beginn der Mission wird der Programmfluss in zwei parallele Abläufe aufgeteilt. Der Erste startet den für die Synchronisierung über Barrieren notwendigen *swarm_ sync_ client*-Node und wartet auf das Anschließen eines Temperatursensors. Dies wird über einen *WaitForActiveTopicROS2*-Block erreicht, welcher darauf wartet, dass Werte in einem über den Parameter *topic* festgelegten Topic gepublisht werden. Der zweite Ablauf startet die für die Initialisierung des Copters notwendigen Treiber und wartet über einen *WaitForActiveTopicROS1*-Block darauf, dass der Treiber vollständig gestartet wurde. Anschließend werden die Programmflüsse über einen *Join*-Block wieder zusammengeführt. Nach dem Join folgt eine Barriere, um den Fortschritt der Mission aller Copter nach der Initialisierung wieder zu synchronisieren. Nach dieser Barriere beginnt die eigentliche Mission mit einem Block zum Scharfschalten der Motoren des Copters gefolgt von dem Befehl zum Abheben. Um die Synchronisierung zu testen, steigen die Copter nicht, wie in ScaleX 2015, sofort auf eine Höhe von 100 Metern. Stattdessen wurden mehrere, durch Barrieren realisierte Zwischenhalte auf verschiedenen Höhen definiert. An diesen Punkten verweilen die Copter jeweils für einige Sekunden. Hierzu wird eine Kombination aus *Barrier*-, *Wait*- und *CallServiceROS1*-Blöcken genutzt. Die Barrieren nutzen jeweils einen alternativen timeout-Nachfolger um zu landen, falls die Barriere nicht von allen Coptern erreicht wird. Nach Erreichen der letzten Barriere an der höchsten Position fliegt der Copter wieder in einer Höhe von einem Meter über seinen Startplatz und landet. Zuletzt werden die Motoren abgeschaltet und die Mission endet.

Die Abbildung 7.2 zeigt den Aufbau der Mission für die Bodenkontrollstation. Nach dem Start werden direkt die Verwaltungsnodes für die Synchronisierung des Ensembles gestartet, bevor mit einem *UserBreakpoint* auf die Freigabe der Mission durch das Bedienpersonal gewartet wird. Dies erfolgt über das Schließen einer auf dem Bildschirm der Bodenkontrollstation erscheinenden MessageBox. Nach der Freigabe der Mission endet die Initialisierung der Synchronisierung und es werden keine neuen Teilnehmer mehr in das Ensemble aufgenommen. Zusätzlich wird über einen *StartLogging*-Block das Logging gestartet. Dieses Zeichnet alle ROS2 Nachrichten im gesamten Ensemble auf und muss aus diesem Grund nur in einer Mission gestartet

werden. Um die Missionen der Copter einheitlich zu halten, wurde die der Bodenkontrollstation zum Koordinator für die Synchronisierung und das Logging gewählt. Die Bodenkontrollstation verhält sich wie ein Teilnehmer im Ensemble. Sie muss also ebenfalls alle Barrieren erreichen, die von den Coptern erreicht werden müssen. Aus diesem Grund folgen in der Mission alle benötigten Barrieren in der korrekten Reihenfolge, sodass der Missionsablauf nicht weiter blockiert wird. Nach Erreichen der letzten Barriere wird das Logging beendet und die Mission terminiert.

7.3 Aufbau der Copter

Als Quadcopter kommen drei Copter vom Typ Rachel des Herstellers rOsewhite Multicopters zum Einsatz (vgl. [99]). Diese sind größer als die in ScaleX 2015 verwendeten Copter vom Typ Saphira. Sie besitzten größere Rotoren, welche effizienter arbeiten und dadurch höhere Lasten tragen und/oder eine längere Flugzeit erzielen können. An dem Copter ist ein RaspberryPi 3B Einplatinenrechner montiert (vgl. Abbildung 6.2). Auf diesem läuft ein Ubuntu 18.04 mit installiertem ROS1 Melodic und ROS2 Dashing. ROS1 ist dabei für die Ansteuerung des auf dem Copter verbauten Autoquad M4 Flugcontrollers (vgl. [42]) mittels des MAVLink Protokolls (vgl. [29]) zuständig. Da das für die Ansteuerung notwendige Paket *mavros* (vgl. [30] noch nicht auf ROS2 portiert wurde, läuft dieses in ROS1 und wird über die ROS1 Bridge (vgl. [31]) an ROS2 angebunden.

An dem Raspberry Pi sind zwei USB-A auf USB-C Kabel angeschlossen, bei denen der USB-C Verbinder zu einem kompatiblen Stecker für das Smart-Sensor-Board umgerüstet wurde. Diese Stecker sind mithilfe von Kabelbindern mit den Armen des Copters verbunden. An jedem Stecker ist ein Smart-Sensor angeschlossen. Für die Beispielmission wurde je ein Smart-Sensor-Board mit einem DS18B20 und eins mit einem DHT-22 Sensor verwendet.

Abbildung 7.3 zeigt einen der drei für die Mission verwendeten Copter. An den Auslegern befinden sich zwei Smart-Sensors, welche über die dort angebrachten Steckverbinder angeschlossen wurden. Diese sind mit dem Raspberry Pi verbunden, der im Zentrum des Rahmens unterhalb der Ausleger sitzt. Der Pi ist über USB ebenfalls mit dem darüberliegenden Autoquad Board verbunden. Auf der an oberster Stelle angebrachten GPS-Antenne wurde ein Label mit der MAV-ID des Copters angebracht, um diesen eindeutig identifizieren zu können.

Für die Ansteuerung der Copter per Software wurde ein ROS1-Paket zur Wegpunktnavigation erstellt. Hiermit kann ein Wegpunkt als Sollpunkt für die Copterposition vorgegeben werden, den der Copter eigenständig anfliegt. Die Kommunikation mit dem im Proof of Concept eingesetzten Coptern er-

folgt über das ROS1-Paket *mavros* (vgl. [30]) Für die Navigation erhält jeder Copter einen Wegpunkt relativ zu seiner Startposition, den er versucht zu erreichen. Beim Ändern des Wegpunkts kann durch eine große Distanz von neuem Punkt und aktueller Position ein hoher Regelfehler in der Positionsregelung des Copters entstehen. Die Regelung des Copters versucht in diesem Fall die Abweichung durch starkes Gegenregeln zu reduzieren. Dadurch kann es zu instabilem Flugverhalten oder Abstürzen kommen, weshalb eine Strategie zum Eingrenzen dieses Verhaltens implementiert wurde. Als Lösung versucht der Copter nicht den Wegpunkt selbst, sondern einen Punkt auf einer von einem anderen Node vorgegebenen Trajektorie anzufliegen. Dieser Node berechnet einen vom Copter zum Wegpunkt zeigenden Vektor und setzt den vom Copter anzufliegenden Sollpunkt in geringer Entfernung zum Copter in Richtung des Vektors. Diese Strategie glättet die Sprünge des Sollpunkts soweit, dass ein stabiler Flug des Copters zum Wegpunkt möglich wird. Von dem implementierten *waypoint_control*-Paket werden die Services *fly_to*, *takeoff* und *land* angeboten. Dem Service *fly_to* wird der Sollpunkt des Copters im 3D-Raum mitgegeben, wodurch mit der bereits beschriebenen Methode eine Trajektorie generiert wird, die den Copter zum Sollpunkt fliegen lässt. *takeoff* generiert einen Punkt an der aktuellen Position des Copters in einer Höhe von einem Meter und fliegt diesen an. Dadurch wird gewährleistet, dass ein am Boden stehender Copter senkrecht nach oben startet und nicht durch eine Seitwärtsbewegung mit Hindernissen am Boden kollidiert. Der *land*-Service erstellt ebenfalls einen Sollpunkt an der aktuellen Copterposition, jedoch ist die Höhe in diesem Fall knapp unter dem Startpunkt gewählt. Dadurch landet der Copter ohne Seitwärtsbewegung auf dem Boden und nach Ausführung des Services kann gewährleistet werden, dass die Motoren des Copters bedenkenlos abgeschaltet werden können.

7.4 Realisierte Sensoren

Für den Versuchsaufbau wurden drei verschiedene Sensoren mit dem Smart-Sensor-Board ins System integriert. Dazu zählen die verhältnismäßig günstigen, aber auch ungenauen DHT22 und DS18B20 Temperatursensoren, welche mit den präziseren PT1000 Temperatursensoren kalibriert wurden. Nachfolgend werden die einzelnen Sensoren und deren Anbindung an das Smart-Sensor-Board im Detail beschrieben.

DHT-22

Der DHT-22-Sensor (vgl. [97], [16]) liefert neben der Temperatur auch die aktuelle relative Luftfeuchtigkeit. Auf dem Sensor ist ein 8-Bit Mikrocontroller

verbaut, welcher das Auslesen eines NTC-Thermistors (Negative Temperature Coefficient Thermistor) für die Temperaturmessungen und eines Luftfeuchtigkeitssensors übernimmt. Außerdem regelt er die Kommunikation mit dem Smart-Sensor-Board über das Single-Bus Protokoll, welches nur eine Datenleitung benötigt.

Den Messbereich gibt der Hersteller mit 0 bis 100 % bei der Luftfeuchtigkeit und -40 bis 80 °C bei der Temperatur an. Die Genauigkeit der gemessenen Werte liegt bei ± 2 % bei der Luftfeuchtigkeit und ±0,1 °C bei der Temperatur. Die maximale Abtastrate des Sensors beträgt 0,5 Hz. Der Sensor wurde (wie im Abschnitt 6.3.4 beschrieben) mit dem Smart-Sensor-Board in das System integriert. Auf dem OLED Display werden sowohl die Temperatur, als auch die Luftfeuchtigkeit angezeigt. Jedoch ist aufgrund eines Fehlers in der ros2arduino library momentan nur das publishen eines Wertes möglich, weshalb nur die Temperatur übertragen wird.

Maxim DS18B20

Der DS18B20 Sensor von Maxim Integrated (vgl. [18]) misst die aktuelle Umgebungstemperatur in einem Bereich von -55 °C bis +125 °C. Dabei liegt die Genauigkeit der Messungen laut Hersteller bei ±0,5 °C im Bereich von -10 °C bis +85 °C. Der Sensor kommuniziert über das 1-Wire Protokoll mit dem Smart-Sensor-Board und wurde ebenfalls nach der in Abschnitt 6.3.4 beschriebenen Vorgehensweise integriert.

PT1000

Der dritte integrierte Sensor ist ein PT1000 Platin Messwiderstand (vgl. [19]). Dieser Sensortyp verändert seinen elektrischen Widerstand abhängig von der Temperatur und wird mittels eines Spannungsteilers und einem Analog-zu-Digital Umsetzers (Analog to Digital Converter, ADC) ausgelesen. Der 10-Bit ADC des NodeMCU Boards liefert lediglich eine Diskretisierung des Abtastsignals in 1024 Schritten, was für die Bestimmung der Temperatur unzureichend ist. Aus diesem Grund wurde ein TEMOD Temperaturmodul (vgl. [65]) genutzt, welches den Sensor an einen 14-Bit ADC anbindet und damit mit 16-facher Genauigkeit abtastet. Diese Werte werden intern in eine Temperatur konvertiert und über den i2c Bus dem Smart-Sensor-Board zur Verfügung gestellt. Durch die Anbindung des Sensors über i2c ist der ADC-Port des NodeMCU frei für andere Verwendungszwecke. Durch die unterschiedlichen Logikpegel des TEMOD-Boards und des ebenfalls über i2c angebundenen OLED-Displays lässt sich ohne i2c-Level-Shifter nur eines der beiden Geräte am i2c-Bus betreiben. Da der Sensor im Versuchsaufbau le-

diglich als Referenzsensor für die Kalibrierung der anderen beiden Sensoren dient, wurde das OLED-Display entfernt.

7.5 Aufbau der Bodenkontrollstation

Die Bodenstation besteht aus einem Notebook, einem WLAN Access Point und einer Funkfernbedienung für die Copter (vgl. Abbildung 6.3). Auf dem Laptop läuft eine virtuelle Maschine mit Ubuntu 18.04 mit ROS2 in der Version Dashing. Auf dem Laptop läuft ebenfalls eine Mission, welche das Starten und Stoppen des Loggings, die Verwaltung der Synchronisiation und das Geben des initialen Startsignals für die Mission übernimmt (vgl. Abbildung 7.2). Außerdem kann vom Laptop aus eine Verbindung zu den MissionControllern auf den Coptern hergestellt und diese darüber gewartet und gedebuggt werden. Als WLAN Access Point kommt der in die Roboterplattform Innok Heros (vgl. [52]) verbaute WLAN-Router zum Einsatz. Dieser verfügt über externe 2.4 GHz Outdoor Antennen, durch die er über die 802.11 Standards b, g und n eine hohe Reichweite erzielt. Als Funkfernbedienung wird eine Spektrum DX9 Black des Herstellers Horizon Hobby genutzt. Diese ist an alle eingesetzten Copter gebunden und ermöglicht das Abschalten aller Copter im Fehlerfall, sie ist daher einem Not-Aus Schalter ähnlich.

7.6 Kalibrierung der Sensoren

Da im Proof of Concept keine wissenschaftlich verwertbaren Daten gesammelt, sondern lediglich die Funktion des Kalibrierungssystems gezeigt werden soll, erfolgte die Kalibrierung der Sensoren nicht in einer Klimakammer, sondern in einem beheizten Raum, in dem durch öffnen eines Fensters die Temperatur langsam abgesenkt wurde. Für den eigentlichen Kalibrierungsprozess wurden der zu kalibrierende Sensor und ein Referenzsensor eine halbe Stunde in einen beheizten Raum auf dessen Innentemperatur aufgewärmt, um Messungenauigkeiten durch unterschiedliche Wärmekapazitäten der Messelemente weitestgehend zu reduzieren. Nachdem die Sensoren die Umgebungstemperatur angenommen hatten, wurde die Heizung ausgeschaltet und das Fenster des Raums geöffnet, um einen langsamen Abfall der Umgebungstemperatur zu erreichen. Zum Starten der Kalibrierung wurden beide Sensoren über ein USB-C Kabel mit einem Mission Controller, wie er auch auf den Coptern zum Einsatz kommt, verbunden. Dieser bindet die Sensoren automatisch ein, wodurch diese ohne weiteres Zutun des Nutzers in ROS ihre Messwerte publishen. Die Firmware des Referenzsensors wurde so parametrisiert, dass

er nicht das übliche Namensgebungsschema für Topics (vgl. Abschnitt 6.3.2) nutzt, sondern in das Topic

```
/calibration/temperature
```

publisht, auf das jeder Temperatursensor einen Listener erstellt. Der zu kalibrierende Sensor erkennt das Eintreffen von Korrekturdaten in dem Topic und beginnt automatisch mit seiner Kalibrierung.

Nachdem 500 Werte gesammelt wurden oder keine Korrekturwerte des Sensors mehr eintreffen, errechnet das Smart-Sensor-Board mittels linearer Regression eine Korrekturkurve (vgl. 6.3.3) und speichert sie im EEPROM ab. Eine solche Fehlerkurve für einen DS18B20 Sensor ist in der Abbildung 6.12 in rot dargestellt. In X-Richtung ist die gemessene Temperatur angezeichnet, für die in Y-Richtung jeweils der berechnete Messfehler (blau) ablesbar ist. Durch verrechnen des Messfehlers mit der gemessenen Temperatur kann die tatsächliche Temperatur angenähert werden. Nach erfolgreicher Kalibrierung werden vom Sensor die kalibrierten Werte gepublisht.

7.7 Ausführung der Mission und Ergebnisse

Der ROS-Treiber für Wegpunktflüge mit den eingesetzten Coptern befand sich zum Zeitpunkt der Durchführung der Mission noch in einem experimentellen Entwicklungsstadium. Aus diesem Grund wurden die Flugbewegungen der Copter während der Mission nur simuliert. Zu Beginn der Mission wurden die drei Copter in einer Reihe aufgestellt und die Akkus installiert. Nach dem Einschalten der Copter stellten diese automatisch eine Verbindung zum WLAN-Netzwerk her und bezogen ihre Mission aus dem dafür eingerichteten git-Repository. Die Bindung der Copter an die Notfallfernbedienung wurde überprüft und die Sensoren an die Copter angeschlossen. Die MissionController erkannten die Sensoren automatisch und starteten die Software zur Integration der Sensoren in ROS2. Auf dem Telemetrierechner wurde ebenfalls die aktuelle Mission heruntergeladen und gestartet. Durch eine Konsolenausgabe auf dem Rechner konnte verfolgt werden, welche der Copter bereits ihre Initialisierung abgeschlossen hatten und bereit für die Mission waren. Nach der Startfreigabe durch Bestätigen der MessageBox am Bildschirm des Telemetrierechners wurde die eigentliche Mission auf den Coptern und dem Telemetrierechner ausgeführt. Dabei war es möglich, den Verlauf der Mission am Telemetrierechner über die Kommandozeile und die Copterpositionen grafisch über das Tool RViz zu verfolgen. Parameter wie Werte der Sensoren und die Position der Copter waren verfügbar. Diese Werte wurden ebenfalls durch das *rosbag2*-Framework zentral auf dem Telemetrierechner gespeichert. Nach

dem Landen und Abstellen der Motoren endete die Ausführung der Mission auf den MissionControllern der Copter und auf dem Telemetrierechner.

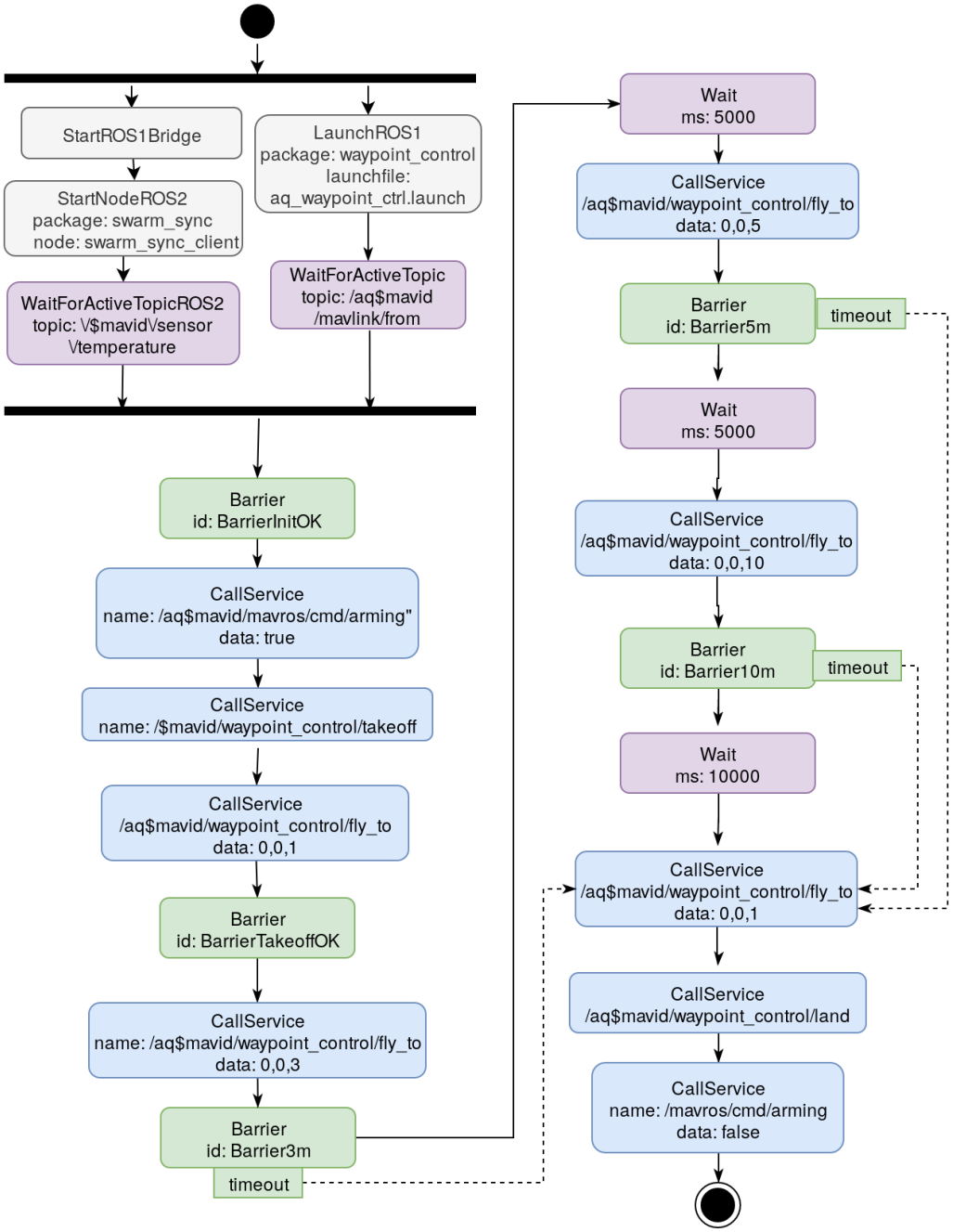

Abbildung 7.1: Beispielmission für die Copter in ScaleX Remastered

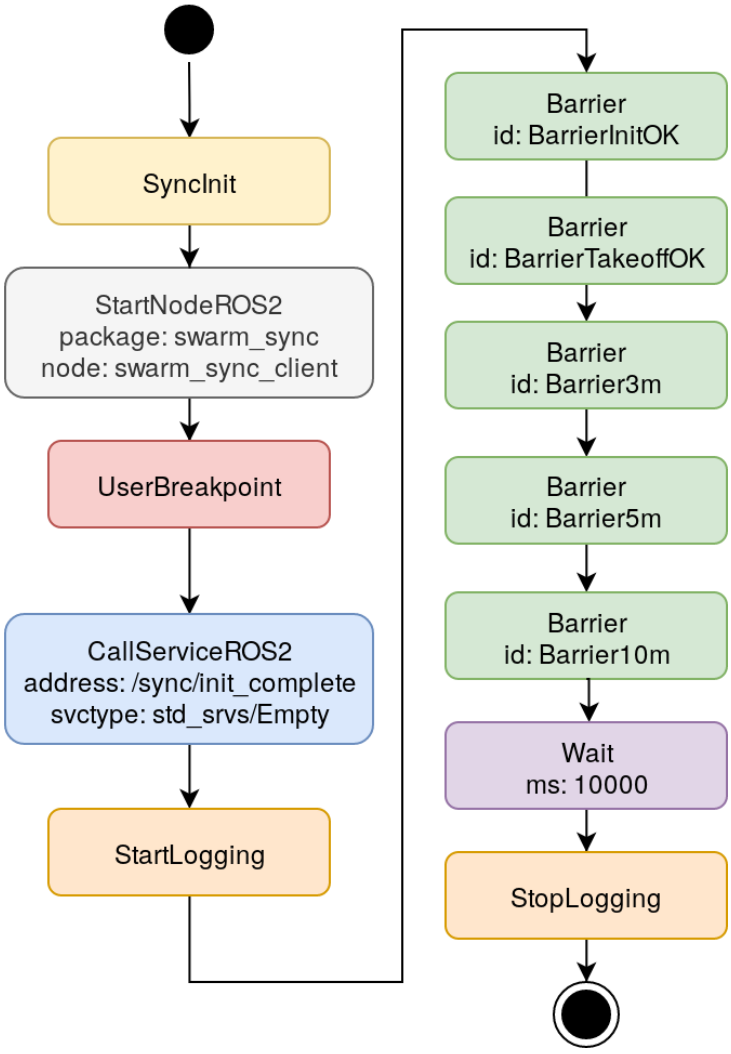

Abbildung 7.2: Beispielmission für die Bodenkontrollstation ScaleX Remastered

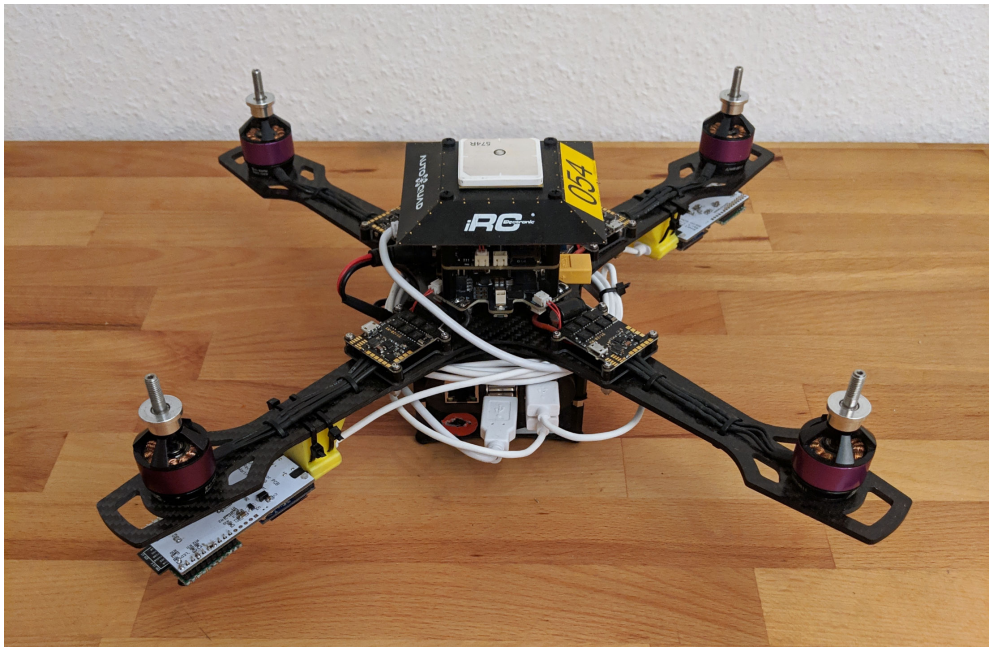

Abbildung 7.3: Aufbau eines Copters für die Durchführung der Beispielmission

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde zunächst ein Überblick über verwandte Arbeiten zu den Themen Missionssteuerung bei Roboterensembles und modulare Hardware gegeben. Anschließend wurden anhand des Fallbeispiels ScaleX 2015 Probleme von Systemen für sensorbasierte Flüge in Ensembles identifiziert und daraus eine Aufgabenstellung definiert. Nach der Erläuterung der Grundlagen von modularer Hardware und der Steuerung von Multicoptern wurde ein Konzept für ein Missionsplanungsframework für Roboterensembles mit einer Anbindung an modulare Sensorik erarbeitet. Dieses Konzept wurde in den darauf folgenden Kapiteln Implementiert und Realisiert sowie in Form eines Proof of Concept bei einem Messflug mit einem Ensemble bestehend aus drei Multicoptern und einer Bodenkontrollstation getestet. Hierfür wurde in zwei Iterationen Hardware entwickelt, welche das Einbinden einfacher Sensoren in das System ermöglicht und diverse Beispielsensoren integriert. Diese Hardware ermöglicht das einfache Erweitern des Systems um neue Sensoren, die Anzeige des Sensorzustands direkt am Sensor und die On-Board-Kalibrierung der Sensoren. Außerdem wurde ein Missionsplanungsframework entwickelt, welches die Definition und Ausführung von Missionen erlaubt und sich über Plugins erweitern lässt. Abschließend wurde in einem Proof of Concept ein Messflug implementiert und auf der entwickelten Hardware simuliert.

Bei der Implementierung und der Ausführung des Proof of Concepts zeigten sich einige Schwächen des Konzepts, welche im folgenden genauer beschrieben werden. Dabei wird ebenfalls auf mögliche Lösungen eingegangen. Die Nutzung des noch unausgereiften ROS2-Projekts bringt an vielen Stellen Instabilität und andere Probleme in das System ein. Das verwendete ros2arudino Plugin befindet sich in einem sehr frühen Stadium der Entwicklung und ist daher vom Funktionsumfang noch eingeschränkt. Beispielsweise konnten keine selbst erstellten Nachrichtentypen für die Kommunikation über die Schnittstelle verwendet werden und das Erstellen von mehr als einem Publisher war aufgrund eines Bugs im Plugin zeitweise nicht möglich. Das Projekt wird jedoch stark weiterentwickelt und für die Zukunft ist die Integration von Services angekündigt. Die im Vergleich zu ROS1 sehr träge reagierenden Kommandozeilentools für ROS2 waren bei der Entwicklung des ROS Plugins für den MissionPlayer problematisch. Diese benötigen oft über zehn Sekunden um einfache Funktionen wie das Aufrufen eines Services auszuführen. Die Implementierung von dauerhaft ausgeführten ROS2 Nodes, welche vom

MissionPlayer Befehle entgegennehmen und die von ROS2 bereitgestellte Python und C++ API nutzen, könnte eine performantere Lösung darstellen. Eine Schwachstelle des MissionPlayers selbst ist die fehlende Möglichkeit, neben dem Programmfluss auch den Datenfluss zu verschalten. Durch die Möglichkeit auch Daten direkt manipulieren zu können, würde die Missionsplanung ähnlich mächtig wie die bereits vorgestellte Blueprint Skriptsprache. Die Implementierung der für diese Funktionalität notwendigen Schnittstellen hätte jedoch den Rahmen der Arbeit gesprengt. Eine semantische Annotation der Sensordaten, um diese Daten in der Missionsplanung nutzen zu können würde die Einsatzmöglichkeiten des Frameworks ebenfalls erweitern. Damit wäre beispielsweise eine Reaktion auf Einbrüche in der Frequenz der Messwerte im Programm oder die automatische Konvertierung von Einheiten möglich. Auch die Implementierung dieses Features war im Rahmen dieser Arbeit nicht möglich. Beim Aufsetzen der Mission bieten die Smart-Sensors zwar Feedback über die integrierten Displays und LEDs, jedoch fehlt eine Statusanzeige für den MissionController. Hier würden zusätzliche LEDs oder ein Display dem Personal dabei helfen, den Status von Internetverbindung, Missionsausführung und anderen Parametern überwachen zu können. Auch die Lesbarkeit der auf der Bodenkontrollstation angezeigten Informationen könnte weiter verbessert werden. Plugins für die Visualisierungssoftware RViz sowie eine grafische Oberfläche für das Erstellen und Überwachen von Missionen wären sinnvolle Erweiterungen des Funktionsumfangs.

Abschließend lässt sich sagen, dass die herausgearbeiteten Anforderungen der Aufgabenstellung erfüllt wurden. Es wurde ein Tool zur Planung und Ausführung von Missionen für Roboterensembles erstellt. Zudem wurde eine Schnittstelle entwickelt und in das System integriert, die es ermöglicht, Sensorik auf modulare Art und Weise auszutauschen. Das System ermöglicht es, nach Implementierung der gerätespezifischen Treiber mit beliebigen Multicoptern eine Abfolge von Wegpunkten abzufliegen. Dabei ist es möglich, auf Fehlerfälle zu reagieren und sich mit anderen Teilnehmern des Ensembles zu synchronisieren. Sensorewerte und Telemetriedaten können während der Mission live eingesehen und für die spätere Auswertung aufgezeichnet werden. Durch den hohen Automatisierungsgrad ermöglicht das in dieser Arbeit vorgestellte Framework eine Reduktion des Personalaufwands im Vergleich zum betrachteten Fallbeispiel. Darüber hinaus lässt die Flexibilität des Systems vielseitige Einsatzzwecke zu und stellt einen Mehrwert für das Arbeiten mit Quadrocopterensembles dar.

Literatur

[1] *Amazon Prime Air Herstellerseite*. URL: https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011 (besucht am 30. 08. 2019).

[2] *Änderungen ROS2 zu ROS1*. URL: http://design.ros2.org/articles/changes.html (besucht am 03. 08. 2019).

[3] *Ankerstation Tharandter Wald - Teilfläche Ökomessfeld*. URL: https://tu-dresden.de/bu/umwelt/hydro/ihm/meteorologie/forschung/mess-und-versuchsstationen/tharandt-ankerstation-oekomessfeld (besucht am 05. 09. 2019).

[4] *Anleitung zum Vernieten von Plastikteilen durch PLA-Filament*. URL: https://www.instructables.com/id/How-to-Make-3D-Filament-Rivets/ (besucht am 29. 08. 2019).

[5] *APM Planner 2 offizielle Webseite*. URL: http://ardupilot.org/planner2/ (besucht am 02. 08. 2019).

[6] *APM Planner Dokumentation*. URL: http://ardupilot.org/planner/ (besucht am 02. 08. 2019).

[7] *Arduino DUE Produktspezifikation*. URL: https://store.arduino.cc/arduino-due-without-headers (besucht am 02. 08. 2019).

[8] *Arduino MKR ZERO Produktspezifikation*. URL: https://store.arduino.cc/arduino-mkrzero (besucht am 02. 08. 2019).

[9] *Arduino Nano Produktspezifikation*. URL: https://store.arduino.cc/arduino-nano (besucht am 03. 08. 2019).

[10] *Ardupilot Dokumentation zu 433 und 915MHz Telemetriemodulen*. URL: http://ardupilot.org/copter/docs/common-sik-telemetry-radio.html (besucht am 05. 09. 2019–).

[11] P Brisset u. a. „The Paparazzi Solution“. In: *MAV 2006, 2nd US-European Competition and Workshop on Micro Air Vehicles*. Sandestin, United States, 2006.

[12] A Bürkle, F Segor und M. Kollmann. „Towards Autonomous Micro UAV Swarms“. In: *Journal of Intelligent & Robotic Systems* 61.1 (2011), S. 339–353. ISSN: 1573-0409. DOI: 10.1007/s10846-010-9492-x. URL: http://dx.doi.org/10.1007/s10846-010-9492-x.

[13] P Chandhar und E G Larsson. „Massive MIMO for drone communications: Applications, case studies and future directions“. In: *arXiv preprint arXiv:1711.07668* (2017).

[14] *Cubelets Produktvorstellung*. URL: https://www.youtube.com/watch?v=4EDsLayRKQA (besucht am 02. 08. 2019).

[15] *Datenblatt des 0,96 Zoll OLED Displays*. URL: https://cdn.shopify.com/s/files/1/1509/1638/files/0.96inch_OLED_Datenblatt_8ee6f307-2d6e-4254-a5af-0a9b9bc773b4.pdf?17806221578424420126 (besucht am 21. 08. 2019).

[16] *Datenblatt DHT22 Temperatur und Luftfeuchtesensor*. URL: https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf (besucht am 21. 08. 2019).

[17] *Datenblatt ESP-12F*. URL: https://www.elecrow.com/download/ESP-12F.pdf (besucht am 21. 08. 2019).

[18] *Datenblatt Maxim DS18B20 Temperatursensor*. URL: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf (besucht am 21. 08. 2019).

[19] *Datenblatt PT1000 Temperatursensor*. URL: http://www.keb.com.tw/Files/Other/PDF/Data%20Sheet-PT1000-RW.pdf (besucht am 21. 08. 2019).

[20] *Datenblatt SHT75 Temperatursensor*. URL: http://www.mouser.com/ds/2/682/Sensirion_Humidity_SHT7x_Datasheet_V5-469726.pdf (besucht am 05. 09. 2019).

[21] *DHL Paketkopter Herstellerseite*. URL: https://www.dpdhl.com/de/presse/specials/dhl-paketkopter.html (besucht am 30. 08. 2019).

[22] *Dokumentation des JSON Formats*. URL: https://www.json.org/ (besucht am 29. 08. 2019).

[23] *Dokumentation des Lifecycle von ROS2 Nodes*. URL: https://design.ros2.org/articles/node_lifecycle.html (besucht am 06. 09. 2019).

[24] *Dokumentation des NodeMCU DEVKIT V1.0*. URL: https://github.com/nodemcu/nodemcu-devkit-v1.0 (besucht am 21. 08. 2019).

[25] *Dokumentation des Pololu USB-C Breakout Boards*. URL: https://www.pololu.com/product/2585 (besucht am 21. 08. 2019).

[26] *Dokumentation des RViz Projekts für ROS2*. URL: https://github.com/ros2/rviz/blob/ros2/README.md (besucht am 23. 08. 2019).

[27] *Dokumentation des smach Frameworks für ROS*. URL: http://wiki.ros.org/smach (besucht am 06. 09. 2019).

[28] *Dokumentation des Tools rosbag*. URL: http://wiki.ros.org/rosbag (besucht am 06. 09. 2019).

[29] *Dokumentation MavLink*. URL: https://mavlink.io/en/ (besucht am 29. 08. 2019).

[30] *Dokumentation mavros*. URL: http://wiki.ros.org/mavros (besucht am 29. 08. 2019).

[31] *Dokumentation ROS1 Bridge*. URL: https://github.com/ros2/ros1_bridge/ (besucht am 29. 08. 2019).

[32] *Dokumentation ROS1 Launch Files*. URL: http://wiki.ros.org/roslaunch (besucht am 06. 09. 2019).

[33] *Dokumentation ROS1 roscore*. URL: http://wiki.ros.org/roscore (besucht am 03. 08. 2019).

[34] *Dokumentation ROS1 Topics*. URL: http://wiki.ros.org/Topics (besucht am 03. 08. 2019).

[35] *Dokumentation ROS2*. URL: https://index.ros.org/doc/ros2/ (besucht am 06. 09. 2019).

[36] M Dorigo u. a. „Swarmanoid: A Novel Concept for the Study of Heterogeneous Robotic Swarms“. In: *IEEE Robotics Automation Magazine* 20.4 (2013), S. 60–71. ISSN: 1070-9932. DOI: 10.1109/MRA.2013.2252996.

[37] N Dousse, G Heitz und D Floreano. „Extension of a ground control interface for swarms of Small Drones“. In: *Artificial Life and Robotics* 21.3 (2016), S. 308–316. ISSN: 1614-7456. DOI: 10.1007/s10015-016-0302-9. URL: http://dx.doi.org/10.1007/s10015-016-0302-9.

[38] *ESP32 Produktspezifikation*. URL: https://www.espressif.com/en/products/hardware/esp32/overview (besucht am 02. 08. 2019).

[39] *Folien zur Einführungsvorlesung Petrinetze von Manuel Hertlein vom Lehrstuhl Theorie der Programmierung an der Humboldt Universität Berlin.* URL: https://www2.informatik.hu-berlin.de/top/lehre/WS05-06/se_systementwurf/Petrinetze-1.pdf (besucht am 05. 09. 2019).

[40] K A Ghamry, M A Kamel und Y Zhang. „Multiple UAVs in forest fire fighting mission using particle swarm optimization". In: *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*. Juni 2017, S. 1404–1409. DOI: 10.1109/ICUAS.2017.7991527.

[41] S Gupte, PIT Mohandas und JM Conrad. „A survey of quadrotor unmanned aerial vehicles". In: *Southeastcon, Proceedings of IEEE*. IEEE. 2012, S. 1–6.

[42] *Herstellerseite Autoquad M4.* URL: http://autoquad.org/wiki/wiki/m4-microcontroller/?lang=de (besucht am 29. 08. 2019).

[43] *Herstellerseite der DJI GCS.* URL: https://www.dji.com/de/ground-station-pro (besucht am 02. 09. 2019).

[44] *Herstellerseite der DJI Mavic Pro 2 Kameradrohne.* URL: https://store.dji.com/de/product/mavic-2?vid=45291 (besucht am 29. 08. 2019).

[45] *Herstellerseite des Pixhawk Flugcontrollers.* URL: http://pixhawk.org/ (besucht am 06. 09. 2019).

[46] *Herstellerseite DJI.* URL: https://www.dji.com/ (besucht am 06. 09. 2019).

[47] *Herstellerseite MathWorks Simulink.* URL: https://de.mathworks.com/products/simulink.html (besucht am 06. 09. 2019).

[48] G Hoffmann u. a. „The Stanford testbed of autonomous rotorcraft for multi agent control (STARMAC)". In: *Digital Avionics Systems Conference, DASC 04. The 23rd*. Bd. 2. IEEE. 2004, 12–E.

[49] *IBM Spezifikation für Plug and Play.* URL: ftp://ftp.software.ibm.com/rs6000/technology/spec/pnp.ps (besucht am 05. 09. 2019).

[50] *Industrial Skyworks - Building Inspections.* URL: http://industrialskyworks.com/drone-inspections-services/ (besucht am 30. 08. 2019).

[51] *Informationen zum DSMX Protokoll*. URL: https://www.spektrumrc.com/Technology/DSMX.aspx (besucht am 06. 09. 2019).

[52] *Innok Heros Produktseite*. URL: http://www.innok-robotics.de/produkte/heros (besucht am 29. 08. 2019).

[53] *Intel Shooting Star Herstellerseite*. URL: http://www.intel.de/content/www/de/de/technology-innovation/videos/drone-shooting-star-video.html (besucht am 07. 04. 2017).

[54] *Introduction to Blueprints*. URL: https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html (besucht am 06. 09. 2019).

[55] O Kosak u. a. „Decentralized Coordination of Heterogeneous Ensembles Using Jadex“. In: *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. Sep. 2016, S. 271–272. DOI: 10.1109/FAS-W.2016.65.

[56] S Leuchter u. a. „Karlsruhe generic agile ground station“. In: *Future Security. 2nd Security Research Conference*. 2007, S. 12–14.

[57] H Lim u. a. „Build your own quadrotor: Open-source projects on unmanned aerial vehicles“. In: *IEEE Robotics & Automation Magazine* 19.3 (2012), S. 33–45.

[58] R D Lorenz u. a. „Dragonfly: a Rotorcraft Lander Concept for scientific exploration at Titan“. In: *Johns Hopkins APL Technical Digest* 34 (2018), S. 374–387.

[59] *M-Blocks Projektseite*. URL: https://www.csail.mit.edu/research/m-blocks-modular-robotics (besucht am 02. 08. 2019).

[60] *Microsoft Azure: Was ist Middleware?* URL: https://azure.microsoft.com/de-de/overview/what-is-middleware/ (besucht am 05. 09. 2019–).

[61] K Mohta u. a. „Experiments in fast, autonomous, GPS-denied quadrotor flight“. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, S. 7832–7839.

[62] *News Artikel zum H-ROS SoM*. URL: https://blog.hackster.io/acutronic-robotics-unveils-new-h-ros-som-designed-for-developing-modular-robotic-components-28e685b439da (besucht am 05. 09. 2019–).

[63] *ODROID-XU4 Produktspezifikation*. URL: https://wiki.odroid.com/odroid-xu4/odroid-xu4 (besucht am 03. 08. 2019).

[64] *OMG Spezifikation zu DDS*. URL: https://www.omg.org/spec/DDS/ (besucht am 06. 09. 2019).

[65] *Operation Manual TEMOD I2C Temperaturmodul*. URL: https://shop.bb-sensors.com/out/media/Operation_manual_TEMOD-I2C_Temperature_module_Pt1000_new_1.pdf (besucht am 21. 08. 2019).

[66] J Penders u. a. „A robot swarm assisting a human fire-fighter“. In: *Advanced Robotics* 25.1-2 (2011), S. 93–117.

[67] *Produktbeschreibung LiPo Warner*. URL: https://www.amazon.de/ITGadgets24-Warner-Schutz-Checker-Voltage/dp/B014HAKD82 (besucht am 03. 08. 2019).

[68] *Produktspezifikation Raspberry Pi Modell 3B*. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (besucht am 06. 09. 2019).

[69] *Projektseite der ROBOTIS Mikrocontrollerboards*. URL: http://emanual.robotis.com/docs/en/parts/controller/opencm904/ (besucht am 21. 08. 2019).

[70] *Projektseite der ros2arduino library auf github*. URL: https://github.com/ROBOTIS-GIT/ros2arduino (besucht am 21. 08. 2019).

[71] *Projektseite des Mara robotic arms*. URL: https://github.com/AlaaAlassi/mara_arduino_remote_control_arduino_side (besucht am 21. 08. 2019).

[72] *Projektseite des Micro-XRCE-DDS-Agents auf github*. URL: https://github.com/eProsima/Micro-XRCE-DDS-Agent/tree/v1.1.0 (besucht am 21. 08. 2019).

[73] *Projektseite NASA New Frontiers Program*. URL: https://www.nasa.gov/planetarymissions/newfrontiers.html (besucht am 07. 09. 2019).

[74] *QGroundControl Dokumentation: Missionsplanung*. URL: https://docs.qgroundcontrol.com/en/PlanView/PlanView.html (besucht am 02. 09. 2019).

[75] J W Romanishin, K Gilpin und Rus D. „M-blocks: Momentum-driven, magnetic modular robots“. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Nov. 2013, S. 4288–4295. DOI: 10.1109/IROS.2013.6696971.

[76] *ROS Industrial Projektseite*. URL: https://rosindustrial.org/ (besucht am 03. 08. 2019).

[77] *ROS News Artikel von The Robot Report über Probleme bei H-ROS*. URL: https://www.therobotreport.com/acutronic-robotics-h-ros-robot-hardware-fails/ (besucht am 03. 08. 2019).

[78] *ROS News Artikel zu H-ROS*. URL: https://www.ros.org/news/2016/10/hardware-robot-operating-system-h-ros.html (besucht am 03. 08. 2019).

[79] *ROS Projektseite*. URL: http://ros.org (besucht am 05. 09. 2019–).

[80] *ROS Service Dokumentation*. URL: http://wiki.ros.org/Services (besucht am 05. 09. 2019–).

[81] *ROS Topic and Service name mapping to DDS*. URL: https://design.ros2.org/articles/topic_and_service_names.html (besucht am 05. 09. 2019–).

[82] *ROS1 actionlib Dokumentation*. URL: http://wiki.ros.org/actionlib (besucht am 02. 09. 2019).

[83] *ROS2 Dokumentation: Launching and monitoring multiple Nodes with Launch*. URL: https://index.ros.org//doc/ros2/Tutorials/Launch-system/ (besucht am 06. 09. 2019).

[84] *ROS2 on DDS*. URL: https://design.ros2.org/articles/ros_on_dds.html (besucht am 03. 08. 2019).

[85] S Sankhar Reddy Chennareddy, A Agrawal und KR Anupama. „Modular Self-Reconfigurable Robotic Systems: A Survey on Hardware Architectures“. In: *Journal of Robotics* 2017 (März 2017), S. 1–19. DOI: 10.1155/2017/5013532.

[86] *Schaltplan des NodeMCU DEVKIT V1.0*. URL: https://github.com/nodemcu/nodemcu-devkit-v1.0/blob/master/NODEMCU_DEVKIT_V1.0.PDF (besucht am 21. 08. 2019).

[87] *Swarmix Projektseite*. URL: http://www.swarmix.org/index.html (besucht am 02. 08. 2019).

[88] *Thomas Witt Blog: Control your Jura Impressa S95 Coffee Maker with Siri via Homekit and Arduino*. URL: https://thomas-witt.com/remote-control-your-jura-impressa-s95-coffee-maker-with-siri-via-homekit-and-arduino-60badb64ed8b (besucht am 21. 08. 2019).

[89] K Thompson. „Programming Techniques: Regular Expression Search Algorithm“. In: *Commun. ACM* 11.6 (Juni 1968), S. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: http://doi.acm.org/10.1145/363347.363387.

[90] *Tinkerforge Dokumentation*. URL: https://www.tinkerforge.com/de/doc/ (besucht am 05. 09. 2019).

[91] *Übersicht über die Standards zu TSN*. URL: https://1.ieee802.org/tsn/ (besucht am 05. 09. 2019).

[92] *UgCS Herstellerseite*. URL: https://www.ugcs.com/ (besucht am 05. 09. 2019–).

[93] *UML Spezifikation in der Version 2.5.1*. URL: https://www.omg.org/spec/UML/2.5.1/PDF (besucht am 05. 09. 2019).

[94] *Video zum Fallbeispiel "Finding Linda" des Swarmix Projekts*. URL: https://www.youtube.com/watch?v=nzcvKPbzgjc (besucht am 05. 09. 2019).

[95] *Video: Making of Intel Drone 100*. URL: https://www.youtube.com/watch?v=eZ-js5zn-I0 (besucht am 07. 04. 2017).

[96] J Wang u. a. „Taking Drones to the Next Level: Cooperative Distributed Unmanned-Aerial-Vehicular Networks for Small and Mini Drones“. In: *IEEE Vehicular Technology Magazine* 12.3 (Sep. 2017), S. 73–82. ISSN: 1556-6072. DOI: 10.1109/MVT.2016.2645481.

[97] *Weitere Angaben zum DHT22 Temperatur und Luftfeuchtesensor*. URL: https://lastminuteengineers.com/dht11-dht22-arduino-tutorial/ (besucht am 21. 08. 2019).

[98] B A White u. a. „Contaminant cloud boundary monitoring using network of UAV sensors“. In: *IEEE Sensors Journal* 8.10 (Okt. 2008), S. 1681–1692. ISSN: 1530-437X. DOI: 10.1109/JSEN.2008.2004298.

[99] *Wissenschaftliche Drohnen der Firma rOsewhite Multicopters*. URL: https://www.rosewhite.de/wissenschaftliche-drohnen (besucht am 29. 08. 2019).

[100] B Wolf u. a. „The SCALEX Campaign: Scale-Crossing Land Surface and Boundary Layer Processes in the TERENO-preAlpine Observatory“. In: *Bulletin of the American Meteorological Society* 98.6 (2017), S. 1217–1234. DOI: 10.1175/BAMS-D-15-00277.1. eprint: https://doi.org/10.1175/BAMS-D-15-00277.1. URL: https://doi.org/10.1175/BAMS-D-15-00277.1.

[101] M Wolfe. „Multiprocessor synchronization for concurrent loops“. In: *IEEE Software* 5.1 (Jan. 1988), S. 34–42. ISSN: 0740-7459. DOI: 10.1109/52.1992.

[102] *YARP Dokumentation: What is Yarp*. URL: http://yarp.it/what_is_yarp.html (besucht am 05. 09. 2019–).

[103] C Zhang und J M. Kovacs. „The application of small unmanned aerial systems for precision agriculture: a review“. In: *Precision Agriculture* 13.6 (Dez. 2012), S. 693–712. ISSN: 1573-1618. DOI: 10.1007/s11119-012-9274-5. URL: https://doi.org/10.1007/s11119-012-9274-5.