technische universität dortmund

Fraunhofer IML

Bachelorarbeit

Dezentrale Echtzeitdatenverarbeitung in heterogenen Systemen

Matthias Wrede

Januar 2019

Gutachter:
Dr. Thomas Liebig
Prof. Dr. Michael ten Hompel

Technische Universität Dortmund
Fakultät für Informatik
Informatik VIII (LS-8)
www-ai.cs.uni-dortmund.de

In Kooperation mit:
Fraunhofer IML
Bereich I: Materialflusssysteme

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Hintergrund . . . 1

1.1.1 Programmiersprachen als Werkzeug zur Code-Implementierung . . . 2

1.1.2 Wissensentdeckung in ubiquitären Systemen - "KDubiq" . . . 5

1.1.3 Frameworks als Werkzeug zur Softwareentwicklung . . . 5

1.1.4 Heterogene Systeme . . . 6

1.2 Ziele der Arbeit . . . 6

1.2.1 Ein Framework zur Programmierung von verteilten Systemen . . . 6

1.2.2 MicroPython-Code in heterogenen Systemen . . . 7

1.3 Aufbau der Arbeit . . . 7

2 Verwandte Arbeiten und Grundlagen **9**

2.1 Das Robot-Operating-System (ROS) . . . 9

2.1.1 ROS-Catkin-packages . . . 10

2.1.2 Gazebo zur Simulation von Robotern . . . 10

2.2 Programmierung heterogener Roboterschwärme mit Buzz . . . 10

2.3 Verteilte Systeme . . . 12

2.3.1 Die logische Lamport Uhr . . . 12

2.3.2 Synchronisation globaler Ressourcen . . . 14

2.4 Verteilte Speicher . . . 15

2.4.1 Der verteilte Speicher in Buzz . . . 15

2.4.2 Der Paxos-Mechanismus . . . 16

3 Das MicroPython-Framework **19**

3.1 Der Name `uBabble` . . . 19

3.2 Architektur des Frameworks . . . 20

3.2.1 Eine logische Lamport-Uhr . . . 20

3.2.2 Steuerungsfunktionen . . . 21

3.2.3 Schnittstelle zur Kommunikation . . . 21

3.2.4 Eine Nachbar-Datenstruktur . . . 21

3.2.5 Ein verteilter Speicher . . . 22
3.3 Hardwarekontext . . . 22
3.3.1 Threading in uBabble . . . 22
3.3.2 Vorhandene Rechenleistung . . . 23
3.4 Anbindung an ROS . . . 23
3.4.1 Gemeinsame Daten . . . 24
3.4.2 Physikalische Schnittstelle . . . 24
3.5 Nachrichtenaustausch in uBabble . . . 25

4 Eine Bibliothek in zwei Systemen 27
4.1 Das Unix-System . . . 27
4.2 Befehlssatz-Architekturen . . . 27
4.3 MicroPython-Code in heterogenen Systemen . . . 28
4.3.1 Externe Bibliotheken im Unix-Port . . . 29
4.3.2 Das Mikrocontroller-System . . . 29
4.3.3 Externe Bibliotheken im ARM-Port . . . 30
4.4 uBabble auf dem Unix-Port . . . 30
4.5 uBabble auf dem ARM-Port . . . 30

5 Eine algorithmische Problemstellung 33
5.1 Mathematische Spezifikation . . . 34
5.2 Mögliche Anwendungsbeispiele . . . 35
5.2.1 Sensoren . . . 36
5.2.2 Paketdrohnen . . . 36
5.2.3 Weltraumteleskop . . . 37
5.3 Mögliche Lösungsansätze . . . 37
5.3.1 Ein intuitiver Ansatz . . . 37
5.3.2 Clustering mit K-Means . . . 37
5.3.3 Potentialfeld . . . 38
5.4 Besonderheiten in Gazebo . . . 40

6 Experiment in Gazebo 41
6.1 Versuch mit Unix-Systemen . . . 41
6.2 Versuch mit ARM-Systemen . . . 42
6.3 Interpretation der Versuche . . . 44

7 Diskussion und Evaluation 45
7.1 Schwachpunkte im Framework . . . 45
7.1.1 Kopplung von Klassen . . . 45
7.1.2 Die verteilte Uhr . . . 46

7.1.3 Authentizität 46
7.2 Positive Aspekte des Frameworks 47
7.2.1 Systemvoraussetzungen 47
7.2.2 Technologieunabhängiges Fluten 47
7.3 Programmierung heterogener Systeme mit MicroPython 47

8 Zusammenfassung und Ausblick **49**
8.1 Eine Gazebo-Simulation mit heterogenen Geräten 49
8.2 Verteiltes maschinelles Lernen 50
8.3 Eine Blockchain 50

A Weitere Informationen **51**
A.1 Installationen 51
A.1.1 MicroPython installieren 51
A.1.2 Robot-Operating-System (ROS) 51
A.1.3 ARM-Emulation 52
A.2 Ausführen der Simulationen 52
A.2.1 Simulation für den Unix-Port 52
A.2.2 Simulation für den ARM-Port 52
A.3 Ein C-Modul für MicroPython 53
A.3.1 Makefile 53
A.3.2 mpconfigport.h 53
A.3.3 Die Moduldatei 54
A.3.4 Innere Klasse eines Moduls 54
A.4 Hinweise zur Unicorn-Emulation 54

Abbildungsverzeichnis **57**

Algorithmenverzeichnis **59**

Literaturverzeichnis **61**

Erklärung **65**

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Mit Beginn des 21. Jahrhunderts stehen Industrie und Arbeitswelt vor einem Wandel. Der Begriff Industrie 4.0 wird öffentlich diskutiert und die Anzahl der IT-Unternehmen in Deutschland nimmt zu [37]. Die Digitalisierung ist weltweit zu beobachten. Nach dem Network-Readiness-Index ist Singapur im Jahr 2015 weltweit führend und bietet die besten Voraussetzungen zur internationalen Wettbewerbsfähigkeit in der IT-Branche [41]. Die schnelle Entwicklung begünstigt weiterhin den Einzug verschiedener IT-Systeme in Unternehmen aller Branchen. Digitale Systeme sollen heute die Überwachung und Steuerung von Prozessabläufen, Verwaltung, Kommunikation und vieles mehr ermöglichen. Auch im privaten Alltag wird der technische Wandel und eine oft damit einhergehende Komplexität wahrgenommen: Smartphones, Tablets, klassische Computer, aber auch eingebettete Systeme sind in privaten Haushalten allgegenwärtig und schon seit einiger Zeit werden Robotersysteme, die Produkte vernetzt und eigenständig erzeugen können, für viele Industriezweige immer interessanter [1]. Im Allgemeinen wächst die öffentliche Aufmerksamkeit im Kontext von eingebetteten Systemen wie Robotern, Sensoren und Drohnen. Aufgrund der Vielzahl an Herstellern in diesem Bereich steht verschiedene Hardware zur Programmierung und Anwendung im gewünschten Kontext bereit. Häufig sollen die Komponenten optimal mit fremder Hardware zusammenspielen können. Die Integration der unterschiedlichen Hardware gestaltet sich jedoch vor allem im Rahmen von Echtzeitsystemen als schwierig, da Latenzen und unterschiedliche Rechenleistung die praktische Umsetzung erschweren. Zukünftige Systeme sollen zudem Ausfallsicherheit gewährleisten und die Privatsphäre der Nutzer umfassend wahren. Dies führt zu einer zunehmenden Vermeidung von zentralen Komponenten im Systementwurf, da sich Nutzer, über die Kommunikationsverbindung und Datenspeicherung des zentralen Service, personifizieren lassen. Außerdem entsteht hierbei eine architekturbedingte Schwachstelle, weil im Falle eines Ausfalls das gesamte System unbrauchbar ist. Zudem hängt die Geschwindigkeit der Informationsver-

arbeitung maßgeblich von diesem zentralen Server ab. Daher soll im Folgenden ein System von ausschließlich gleichberechtigten Kommunikationspartnern betrachtet werden, die unter Echtzeitbedingungen arbeiten. Auf diese Weise müssen auch rechenleistungsbedingte Beschränkungen berücksichtigt werden. Wegen der Inhomogenität der heute verfügbaren Hardware sprechen wir von den verschiedenen, miteinander kommunizierenden Geräten als heterogene Geräte.

1.1.1 Programmiersprachen als Werkzeug zur Code-Implementierung

Die Implementierung von Programmcode in verschiedene Systeme kann sich als sehr schwierig erweisen. Hardware muss in einer für Menschen intuitiv verständlichen Sprache programmierbar sein. Zwecks praktikablen Systementwurfs entstanden und entwickelten sich somit im Laufe der Geschichte zahlreiche Programmiersprachen mit verschiedenen Ansätzen und Zielen [21]. Mit dem Versuch, Sprachen durch Compiler immer stärker von der Hardware zu abstrahieren und der menschlichen Sprache anzunähern, sind jedoch viele Programmiersprachen entstanden, die auf spezifische Anwendungsbereiche zugeschnitten sind. Diese Sprachen sind für kontextübergreifende Programmierung unbrauchbar und werden als 'domain-specific languages' (DSL) bezeichnet. Demgegenüber stehen allgemeine Programmiersprachen wie C, C++, Java oder Python. Diese werden als 'general-purpose languages' (GPL) bezeichnet und sind für die Einbindung in verschiedene Systeme und Anwendungskontexte interessant.

Die Programmiersprache C

Keine andere Sprache hat die Entwicklung von Betriebssystemen so sehr geprägt wie das mit Beginn der siebziger Jahre entwickelte C. Es ist die weltweit am häufigsten eingesetzte Programmiersprache [26]. Abstrakt und hardwarenah zugleich ist es optimal zur Implementierung von ressourcensparendem Quellcode in eingebettete Systeme geeignet. C-Code kann jedoch nur in Systemen mit vollständigem Compiler für die Hardware ausgeführt werden. Der Bau von Compilern für eine neue Hardware ist aber aufwendig. Zudem ist C als prozedurale Sprache, die kein namespacing unterstützt, für die Implementierung von objektorientierten Systementwürfen ungeeignet. Durch den verhältnismäßig engen Bezug des Quellcodes zur Rechnerarchitektur bietet C dem Programmierer allerdings viele Möglichkeiten zur Optimierung der Performance unter Teilkenntnis der Funktionsweise des Compilers [5]. Somit ist C insbesondere zur Implementierung von Algorithmen mit Echtzeitschranken qualifiziert.

Die Programmiersprache Java

Mit Java ist eine Sprache entstanden, die für die Problematik der aufwendigen Compiler eine clevere Lösung gefunden hat: Der Java-Quelltext wird in maschinellen Zwischencode, den sogenannten Bytecode, übersetzt und kann von einer virtuellen Maschine, der 'Java Virtual Machine' (JVM), ausgeführt werden. Dieser Bytecode kann auf allen Systemen ausgeführt werden, für die auch eine JVM als Interpreter zur Verfügung steht. Die Implementierung einer solchen virtuellen Maschine ist einfacher als das Schreiben eines vollständigen Compilers vom Quelltext bis zum Assembler-Code. Die JVM muss dafür in das System implementiert werden können, sodass der kompilierte Java-Bytecode von der Maschine interpretiert werden kann. Die Leistungsanforderungen an die Hardware für eine Java-Maschine sind jedoch hoch. Für Java 8 sind mindestens 128 MB Arbeitsspeicher und eine Intel Pentium CPU mit 266 MHz Taktfrequenz erforderlich [31]. Eine solche Rechenleistung ist aber bereits für eine CPU in Mikrocontrollern, wie den ARM Cortex-M3, zu hoch. Des Weiteren zielt die JVM auf eine Einbindung der vollständigen Java-Bibliothek. Dazu gehören dann auch Funktionen wie `Runtime.exec()`, um einen neuen Prozess zu starten. Parallele Prozesse sind in eingebetteten Systemen jedoch problematisch, da diese oft auf eine hardwarebasierte Memory-Management-Unit, kurz MMU, verzichten. MMUs sind eine sehr praktikable Lösung zur Übersetzung logischer Adressen in physikalische Adressen und schützen Prozesse dadurch vor illegalen Speicherzugriffen. Bei fehlender MMU müsste diese also in irgendeiner Form emuliert werden. Es ist zudem sehr wahrscheinlich, dass viele eingebettete Systeme weitere hardware-spezifische Besonderheiten enthalten, die eine JVM-Portierung zusätzlich erschweren. Damit ist Java als Programmiersprache in heterogenen Systemen mit Mikrocontrollern ungeeignet und wird im Rahmen dieser Arbeit nicht weiter betrachtet.

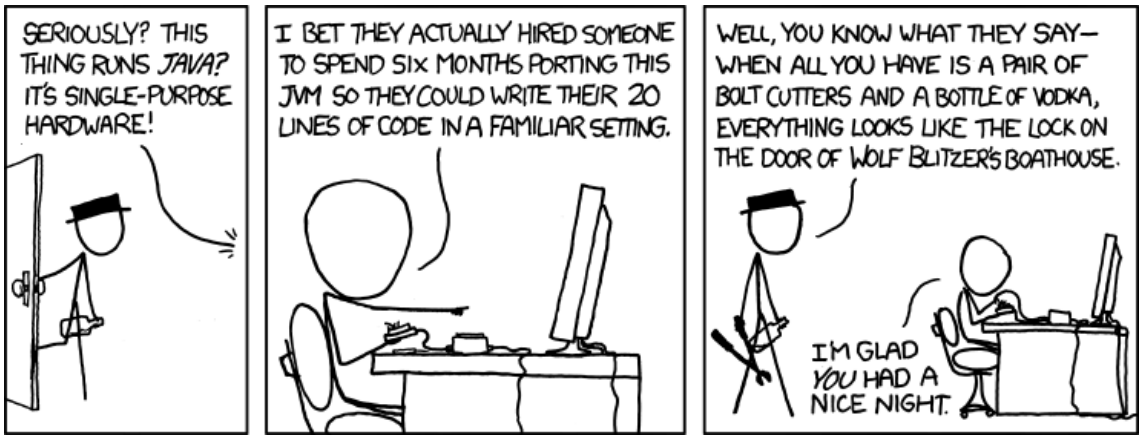

Abbildung 1.1: Golden Hammer [3]

Die Programmiersprache MicroPython

MicroPython ist eine Programmiersprache, die, vollständig in C-geschrieben, von der Syntax indentisch zu Python 3 ist. Als interpretierte Sprache führt MicroPython stark von der Hardware abstrahierten Code aus. Debugging ist hier vergleichsweise einfach: Das Programm wird mit dem Starten des Interpreters sofort ausgeführt und läuft solange, bis es terminiert oder ein Fehler auftritt. MicroPython generiert grundsätzlich keinen maschinellen Zwischencode. Skripte werden einfach auf das System kopiert und vom Interpreter ausgeführt. Dahinter liegen kompilierte C-Module, dessen Code dann tatsächlich in der Hardware implementiert ist. Mit interpretierten Sprachen geht jedoch im Allgemeinen der Nachteil der schwächeren Performance einher, da der Interpreter jede Instruktion einzeln ausführt. Damit ist MicroPython, als stark von der Hardware abstrahierte Sprache, für eine Verarbeitung von Daten mit Echtzeitschranken ungeeignet. Praktisch ist aber, dass MicroPython, da es in C geschrieben ist, mit C-Modulen erweitert werden kann [24]. Wie oben bereits erwähnt ist C als hardwarenahe Sprache mit Compiler besonders gut für Echtzeitdatenverarbeitung geeignet. An dieser Stelle empfiehlt es sich also, zeitkritische Berechnungen in ein C-Modul auszulagern, während das Strukturmodell des Programms durch den darüber liegenden Micropython-Code klar ersichtlich bleibt. Darüber hinaus verfügt MicroPython über alles Weitere, was der Sprache C fehlt. Von Strings bis zu closure-Funktionen und Klassen in Packages, isoliert in eigenen Namespaces, können objektorientierte Systementwürfe mit MicroPython in eingebettete Hardware implementiert werden. Programmcode kann in Mikrocontrollern mit lediglich 256KB Code-Speicher und 16KB RAM ausgeführt werden [14]. Aufgrund der Unterschiedlichkeit eingebetteter Systeme und ihrer Anwendungsfelder, ist MicroPython für verschiedene Ports verfügbar. Die Ports unterscheiden sich auch in den verfügbaren Standardbibliotheken. Beim Wechsel des Ports können Module nicht oder nur unzureichend implementiert sein, sodass entsprechende Module in C neu geschrieben oder umgeschrieben werden müssen. Wegen der gleichen Syntax kann ein Skript aber sowohl von einem Python3- als auch einem MicroPython-Interpreter ausgeführt werden, sofern die importierten Module in allen Schichten identisch sind. Somit könnte auf leistungsstarken Rechnern zu Python gewechselt werden, um eine Bibliothek in neue Anwendungskontexte zu implementieren, für die bereits Python-Bibliotheken verfügbar sind.

Die Programmierung der Hardware im Rahmen von Echtzeitdatenverarbeitung erfordert detailliertes Wissen über die verwendete Programmiersprache, aber auch Kenntnisse, die über reine Syntax und Umsetzung des Compilers einer Sprache hinausgehen. Neben rein funktionalen Anforderungen ist bspw. auch die Betrachtung sogenannter Liveness relevant, um zu prüfen, ob ein notwendiges Ereignis überhaupt eintritt [2]. Eine Analyse von verteilten Echtzeitsystemen ist darüber hinaus noch schwieriger.

1.1.2 Wissensentdeckung in ubiquitären Systemen - "KDubiq"

Neue Forschungsfelder wie "Knowledge Discovery in ubiquitous systems (KDubiq)" zielen darauf ab, strukturierte Ansätze zur Entwicklung von heterogenen Echtzeitsystemen zu schaffen. KDubiq versteht sich dabei als die Form von Wissensentdeckung, die Muster aus Datenströmen erkennt, welche innerhalb einer dynamischen und verteilten Infrastruktur von miteinander interagierenden Geräten generiert wurden [23]. Gegenstand der Forschung ist dabei eine Hardware, die im System folgende Eigenschaften aufweist:

1. Das Gerät befindet sich in einer räumlich und zeitlich veränderlichen Umwelt,
2. kann den Ort wechseln, verschwinden oder hinzugefügt werden,
3. kann Informationen verarbeiten,
4. kennt nur die eigene lokale Sicht auf die Umwelt,
5. arbeitet unter Echtzeitbedingungen,
6. die Geräte tauschen Informationen untereinander aus, arbeiten also kollaborativ.

1.1.3 Frameworks als Werkzeug zur Softwareentwicklung

Eine Besonderheit bei KDubiq ist unter anderem die Entwicklung von neuen Algorithmen, um ein Vorhersageproblem im verteilten Lernprozess auf den einzelnen Geräten lösen zu können. Ein Anwendungsbeispiel hierfür ist das "Vehicle Data Stream Mining System", kurz VEDAS. Hier wurden bereits existierende Algorithmen in weniger rechenintensive und approximierende Entwürfe überführt. Die Entwicklung neuer Algorithmen ist oft schwierig. Die Hürden zur Einbindung in Softwareprojekte sind im Allgemeinen essentiell geringer, wenn Algorithmen und grundlegende Systemmechanismen bekannt sind. Interpreter und Compiler vereinfachen dann die Einbindung in programmierbare Hardware, wie in Abschnitt 1.1.1 dargestellt. Das Nutzen von Methoden aus dem Bereich der Softwarekonstruktion bietet Unternehmen in der Softwareentwicklung darüber hinaus die Möglichkeit, einen gewünschten Qualitätsstandard des Produkts zu sichern und die Erfolgswahrscheinlichkeit des Projekts zu erhöhen [9]. Insbesondere aber die Kombination von individuellem Programmcode mit etablierten und existierenden Lösungen in Form von Softwarebibliotheken verschafft Projektgruppen einen entscheidenden Vorteil gegenüber der unwissenden Konkurrenz. Somit sind Frameworks ein unabdingbares Werkzeug zur Softwareentwicklung in der heutigen Zeit. Dieses Vorgehen ist effizient und schafft Synergien, anstatt das Rad ständig neu zu erfinden.

1.1.4 Heterogene Systeme

" The word heterogeneous is an adjective that means composed of different constituents or dissimilar components. "[15] „Das Wort heterogen ist ein Adjektiv, das die Komposition von verschiedenen Bestandteilen oder unähnlichen Komponenten bezeichnet.“ Dem entsprechend werden unter heterogenen IT-Systemen miteinander vernetzte Komponenten verstanden, dessen Unterschiedlichkeit für das System charakteristisch ist. Die einzelnen Komponenten sind wiederum Systeme, die Informationen verarbeiten und über Schnittstellen mit der Außenwelt kommunizieren. Das Internet ist ein Netz von Rechnern, indem zahlreiche unterschiedliche Systeme arbeiten. Man kann das Internet in gewisser Weise auch als heterogenes System betrachten. Jedes Gerät, das mit dem Internet verbunden ist, verfügt über eine damit kompatible Schnittstelle. In lokalen Netzwerken sind bspw. Ethernet- und Wifi- Schnittstellen in Endgeräten verbreitet. Das Austauschen von Informationen zwischen den Endgeräten ist durch einen Transportschicht-Protokollstack realisiert. Darin sind weitere Schichten, unter anderem die IP-Schicht, enthalten. Das Konzept der Schichten ist hier interessant. Es wird verwendet, um nach Außen in beabsichtigter Weise kommunizieren zu können. Schicht-Konzepte sind auch Gegenstand der objektorientierten Programmierung, wie beispielsweise in Form von Vererbung der Attribute und Methoden einer Klasse an seine Unterklasse. Eine verteilte Verarbeitung von Daten in heterogenen Systemen unter Echtzeitschranken ist jedoch, wie bereits zu Beginn des Kapitels skizziert, problematisch. Sowohl unterschiedliche Rechenleistung als auch Latenzen und Ausfälle können Entwurf und Entwicklung eines solchen Systems stark erschweren.

1.2 Ziele der Arbeit

1.2.1 Ein Framework zur Programmierung von verteilten Systemen

Weil dezentrale Systeme viele Sicherheiten bieten, wird eine Bibliothek zur verteilten Koordination von Aufgaben zwischen heterogenen Geräten erstellt. Die Bibliothek soll mittels Abstraktion und Reduktion die Implementierung einer Klasse von verteilten Algorithmen erleichtern. Alle Aufgaben werden dabei dezentral gelöst, sodass ein Prozess mit Sonderfunktionen weder erlaubt noch notwendig ist. Jeder Algorithmus wird protokoll-basiert auf gleichberechtigten Instanzen innerhalb eines heterogenen Systems ausgeführt. Ein Gerät mit einer für das System relevanten Zusatzfunktion macht die Ausfallsicherheit des gesamten Systems hinfällig. Vollständig dezentrale Systeme sind nicht zwingend für eine KDubiq-Architekur charakteristisch, da dort zentrale Instanzen durchaus an der Informationsverarbeitung beteiligt sein dürfen. Die sechs oben, in Abschnitt 1.1.2, erwähnten Eigenschaften von KDubiq-Geräten sollen dennoch auch in diesem Kontext erfüllt sein. Zur Visualisierung der Arbeit von verteilten Algorithmen wird mit dem 3D-Simulator Gazebo [27] und ROS [28], einem Framework zur Programmierung einzelner Roboter, gearbeitet.

Eine AbstractionLayer, siehe 3.4, verbindet Framework und ROS, sodass Roboter über das Framework gesteuert werden können. Gazebo visualisiert dazu ein Phänomen, das später konkret spezifiziert wird. Hierfür soll im weiteren Verlauf dieser Arbeit die Koordination von Schwarmverhalten im Vordergrund stehen, da eigenständig arbeitende Geräte ein globales Ziel verfolgen. Die programmatische Umsetzung eines solchen Verhaltens ist schwierig, weil sich die Spezifikation des individuellen Verhaltens an dem Phänomen der Gruppe orientieren muss. Ohne Nutzung eines Frameworks führt dies zu einem aufwendigen und fehleranfälligen Entwicklungsprozess.

1.2.2 MicroPython-Code in heterogenen Systemen

Wie in Abschnitt 1.1.1 dargestellt, bietet die Programmiersprache MicroPython eine vielversprechende Möglichkeit, um Programmcode praktikabel in verschiedene Hardware zu implementieren. Das Framework wird daher in MicroPython geschrieben, sodass Quellcode auch in eingebettete Hardware wie Mikrocontrollern implementiert werden kann. Es ist aber noch unklar, wie gut mit einer MicroPython-basierten Implementierung Kommunikation zwischen verschiedenen Komponenten oder Systemen realisiert werden kann. Als Hochsprache ist MicroPython-Quellcode vor allem für den abstrakten Kern des Frameworks geeignet. Bei der Implementierung in verschiedene Systeme sind aber Probleme zu erwarten. Daher wird das Framework in zwei verschiedenen Systemen auf STM32- und Unix-Basis entwickelt, um eine Aussage zur Eignung von MicroPython, als Sprache zur Programmierung in heterogenen Systemen, treffen zu können.

1.3 Aufbau der Arbeit

Im Rahmen dieser Arbeit werden MicroPython-Module gebaut, die verteilte Mechanismen implementieren. Dazu existieren bereits ausführliche Arbeiten, dessen Konzepte verwendet werden können. Viele Arbeiten basieren allerdings auf einer Grundlage, die für ein allgemeines Anwendungsfeld zu eingeschränkt oder unpraktikabel sind. Kapitel 2 stellt daher mögliche Grundlagen und verwandte Arbeiten vor.

Kapitel 3 beschreibt alle wesentlichen Komponenten des Frameworks sowie dessen strukturellen Kontext in Verbindung mit ROS. Die Komponenten liegen in Form von Packages, Klassen und Methoden vor. Ein UML-Klassendiagramm stellt dazu das Strukturmodell des Frameworks übersichtlich dar. Einige Klassen implementieren Funktionen, die für das Abstrahieren verteilter Problemstellungen hilfreich sind. Die übrigen Klassen dienen zur Abstraktion gerätespezifischer Funktionalitäten. Da in dieser Arbeit ROS verwendet wird, ist eine Anbindung des Frameworks an ROS über diese Module erforderlich. Des Weiteren sind aufgrund der verwendeten Technologien innerhalb der Bibliothek ein paar Besonderheiten für die späteren Experimente mit Gazebo in Kapitel 6 zu berücksichtigen.

Kapitel 4 stellt zwei grundlegend verschiedene Systeme vor, in die MicroPython-Programmcode zusammen mit der Bibliothek implementiert wird. Problematisch hierbei ist, dass der Quellcode des Frameworks auf weiteren Programmmodulen basiert. Verschiedene eingebettete Systeme sind jedoch für völlig unterschiedliche Anwendungsszenarien ausgelegt, sodass bei Portierung eines Programms viele Module möglicherweise nicht mehr verfügbar sind.

Kapitel 5 stellt eine verteilte Problemstellung vor, dessen Lösung anhand des Frameworks umgesetzt wird. Dazu werden mögliche Anwendungsbeispiele im realen Kontext genannt sowie geeignete Ansätze zur Lösung diskutiert.

In Kapitel 6 demonstrieren zwei Gazebo-Simulationen die Implementierung eines Algorithmus aus Kapitel 5 in Mikrocontroller- und Unix-Systemen.

Kapitel 7 diskutiert die Architektur des Frameworks und beurteilt den Entwurf hinsichtlich der softwaretechnischen Umsetzung und Eignung. Zudem wird MicroPython, als Sprache zur Programmierung heterogener Echtzeitsysteme, im Allgemeinen evaluiert.

Kapitel 8 fasst die Erkenntnisse der Arbeit kurz zusammen und erläutert mögliche weiterführende Arbeiten.

Kapitel 2

Verwandte Arbeiten und Grundlagen

2.1 Das Robot-Operating-System (ROS)

Im Rahmen dieser Arbeit ist das Robot-Operating-System, kurz ROS, ein zentraler Bestandteil. MicroPython-Code soll in die zwei Systeme STM32 und Unix implementiert werden. Beide Systeme kommunizieren mit ROS. Das Robot-Operating-System ist, anders als der Suffix " Operating-System" im Namen vermuten lässt, eine Sammlung von Bibliotheken auf Linux-Basis. Es handelt sich also nicht um ein Betriebssystem. ROS wurde erstmals am 2. März 2010 als „ROS Box Turtle“ Distribution veröffentlicht [29]. Entwickelt wurde es vom Robotik-Forschungslabor Willow Garage, das bis heute maßgeblich zur Entwicklung der Open-Source-Software beiträgt [40]. ROS vereinfacht als flexibles Framework die Programmierung von "general-purpose robot software"[28] enorm. Es zielt auf den Entwurf von Robotern in isolierter Betrachtung, ist also ein " single-robot-framework". [32] Mit der Standardinstallation kann das gesamte Framework in Python oder C++ verwendet werden. Zentrales Element in ROS ist der ROS-Master. Dieser ermöglicht einen Datenaustausch zwischen darin enthaltenen ROS-Instanzen über zwei wesentliche Abstraktionen, den ROS- `Nodes` und `Topics`. Hierbei erfolgt ein subscribe oder publish eines Nodes auf ein bestimmtes Topic, um Informationen zu erhalten oder zu veröffentlichen. Des Weiteren verwendet ein Roboter ROS-Nodes um bspw. Programme auszuführen, Steuerungselemente über Topics anzusprechen und Services bereitzustellen oder zu nutzen. Zum Arbeiten mit Robotern sind weitere Konzepte wie " transformation-frame-trees"(tf-package) und "joints" erforderlich. Für genauere Informationen sind die " Core ROS Tutorials" im ROS-Wiki [30] sehr hilfreich. Kenntnis über Nodes und Topics sind jedoch für das Verständnis der Funktionsweise von ROS im weiteren Verlauf ausreichend, da die Verknüpfung zwischen MicroPython-Framework und ROS hierauf beruht.

2.1.1 ROS-Catkin-packages

Roboter können in ROS seit 2012 mit der Version ROS-Groovy in einem sogenannten " Catkin-package " programmiert werden. Catkin ist das build-system für workspaces in ROS. Jedes Catkin-package beinhaltet eine `package.xml`, welche die Author- und Lizeninformation beschreibt, sowie eine `CMakeLists.txt`, die für das Bauen von Softwarepaketen verwendet wird. Jedes Catkin-package wird nach einem festen Schema funktional in verschiedene Ordner unterteilt:

1. config: Hier befinden sich Parameter für Controller externer Programme.
2. launch: XML-Dokumente, die die Ausführung des Befehls 'roslaunch' beschreiben.
3. nodes: Hier befinden sich Programme, die als ROS-Nodes ausgeführt werden sollen.
4. urdf: XML-Dokumente, die den Aufbau der vorhandenen Roboter beschreiben.

Es gibt weitere Ordnerbezeichnungen, bspw. map, rviz und scripts, die mit ROS verknüpft sind. Dadurch ist das Paket wohlstrukturiert, sodass über verschiedene ROS-Befehle mit dem Paket gearbeitet werden kann.

2.1.2 Gazebo zur Simulation von Robotern

Gazebo ist ein 3D Simulator, der mit eigener Physics Engine, qualitativ hochwertiger Grafik und grafischem Userinterface ausgestattet ist [27]. Gazebo wird hauptsächlich zur Simulation von Robotern auf Basis von ROS verwendet. Es können komplexe Welten modelliert werden. Das Gewicht jedes Objekts und Roboters ist dabei genau definiert ebenso wie alle Flächenträgheitsmomente jeder Teilgeometrie eines Roboters. Gazebo kommuniziert die Positionen und Orientierungen aller Roboter über die Topic " /gazebo/model_states" mit dem ROS-master. Damit kann jeder Roboter seine eigene Position in der Gazebo-Welt ermitteln. Alle Roboter sind vorab also quasi mit einem GPS-Gerät über Gazebo ausgestattet, ein nützliches Werkzeug zur Umsetzung einer verteilten Problemstellung und gegenseitiger Kommunikation von Positionen zwischen den Robotern.

2.2 Programmierung heterogener Roboterschwärme mit Buzz

Die Programmiersprache Buzz wurde entwickelt, um Schwarmverhalten von Robotern in heterogenen Systemen implementieren zu können. Unter Schwärmen werden hierbei miteinander vernetzte Roboter verstanden, die Bewegungen, Aufgaben und vieles mehr dezentral untereinander koordinieren. Dazu wird der Programmierer mit zwei Objektklassen, dem Roboter und dem Schwarm, konfrontiert. Auf diese Weise ermöglicht Buzz die Programmierung von Schwärmen über einen 'top-down'-Ansatz in Kombination mit einem 'bottom-up'-Ansatz zur Programmierung spezifischer Funktionen des einzelnen Roboters [32]. Dar-

über hinaus enthält Buzz weitere Abstraktionen wie die `neighbors` und `stigmery` Datenstrukturen. Die `neighbors`-Datenstruktur speichert alle lokal vorhandenen Nachbarn ab. Die `stigmery`-Struktur, auch "virtual stigmery"[32] genannt, ist ein verteilter (key, value)-Speicher. Dieser ermöglicht die Verteilung von einzelnen Aufgaben zwischen den Robotern, wobei es sich hierbei nicht um dauerhaft einzigartige Funktionen handelt, die von einem Gerät ausgeführt werden. Einträge innerhalb des Speichers sind nicht synchronisiert und können jederzeit von einem beliebigen Gerät geändert werden. Die Ausfallsicherheit des gesamten Systems ist von Aufgaben dieser Art nicht abhängig. `neighbors`-, `stigmery`- und `swarm`-Datenstruktur sollen die schwierige Verknüpfung zwischen individuellem und daraus resultierendem gruppenbasiertem Verhalten erleichtern. Buzz ist jedoch eine domänenspezifische Sprache (DSL, siehe 1.1.1) und wurde entwickelt, um auf existierende Systeme wie das Robot-Operating-System, siehe Abschnitt 2.1, aufgesetzt zu werden. Auf diese Weise kann Buzz in gewissem Umfang ROS-Funktionalitäten wie Steuerung und Kommunikation mitnutzen. Buzz-Code wird hierfür in einer virtuellen Maschine, der BuzzVM, ausgeführt. Zur Kopplung von Buzz und ROS wird eine sogenannte `ROSBuzz Abstraction Layer` [36] genutzt, denn ROS ermöglicht bereits die Programmierung heterogener Roboter, wie Fahrzeuge und Drohnen. Die `ROSBuzz Abstraction Layer` soll daher allgemeine Operationen abstrahieren, denn eine Drohne verwendet andere Topics und benötigt weitere Prozeduren um bspw. einen Abhebevorgang zu starten. Neue Datenstrukturen und Funktionen werden in Buzz als C-closures eingebunden. Diese können z.B. verwendet werden, um Sensorwerte des darunterliegenden Systems zu speichern. Somit ist Buzz auch eine in gewissem Rahmen erweiterbare Sprache, die jedoch stark von C-Code abhängig ist. Das Skript des Programmierers, auch Buzz-Skript genannt, wird iterativ im Rahmen einer festen Schrittfolge ausgeführt. Hierfür ist eine Programmschleife fest in der BuzzVM implementiert. Diese ist in fünf Phasen unterteilt: Zuerst werden Sensorwerte eingelesen, dann eingegange Nachrichten gespeichert, daraufhin wird das Buzz-Skript ausgeführt, dann Nachrichten aus der Warteschlange versendet und schließlich Aktoren bedient. Die BuzzVM ist eine sehr kleine virtuelle Maschine mit einer Größe von lediglich zwölf Kilobyte. Der Quellcode wird zudem in Zwischencode kompiliert und ermöglicht dadurch eine schnelle Ausführung der Buzz-Schleife. An dieser Stelle wird jedoch deutlich, dass die Programmierung in Buzz aufgrund des domänenspezifischen Kontextes stark eingeschränkt ist. Externe Bibliotheken können nur sehr umständlich verwendet werden. Zudem ist der Umfang der Anwendungsmöglichkeiten aufgrund der Einbettung von Quellcode in eine Schleife unnötig eingschränkt. Hier liegen die Vorteile eines Frameworks! Sie können modular in bereits existierende Systeme eingebunden werden und sind unabhängiger von der Systemumgebung verwendbar. Weil Buzz bereits vielversprechende Demonstrationen und eine klare Struktur vorweisen kann, sind die grundlegenden Konzepte des MicroPython-Frameworks von Buzz inspiriert.

2.3 Verteilte Systeme

Eine verteilte IT-Datenverarbeitung findet heute in vielen Bereichen statt. Hauptgegenstand sind Computer-Cluster. Hier arbeiten mehrere Rechner an einer gemeinsamen Aufgabe. Beispielsweise werden Webservices oft von Clustern bereitgestellt, um gegen DDOS-Attacken geschützt zu sein und Ausfällen im Allgemeinen vorzubeugen. Cluster sind auch, wie in Abschnitt 1.1.2 angedeutet, interessant im Bereich des maschinellen Lernens und Dataminings. Vorhersagen über Verkehrsaufkommen im Straßenverkehr sind beispielsweise für Navigationsgeräte interessant. Auch hier spielen Privatsphäre und permanente Systemverfügbarkeit eine wesentliche Rolle. Wird das Bestimmen von Vorhersagen in räumlich verteilte Messstationen einer Stadt ausgelagert, können lokale Vorhersagen durch Kommunikation mit Nachbarstationen getroffen werden. Ein Broadcasting aller Beobachtungen belastet das Netz jedoch stark. Diese Auslastung kann minimiert werden, indem man labelling-Strategien anwendet. Das Kommunizieren aggregierter Labels verringert die Netzauslastung und erschwert ein Tracking von Fahrzeugen zwischen Knotenpunkten. [19] Das Beispiel fällt auch in den Bereich ubiquitärer Wissensentdeckung, wobei die Messensoren statisch an Verkehrskreuzungen positioniert sind und den Ort somit nicht aktiv wechseln. Anders sähe eine Portierung des Systems in Fahrzeuge aus. In diesem Fall wären alle Aspekte einer ubiquitären Wissensentdeckung vorhanden. Wie in Abschnitt 1.1 und 1.1.4 erläutert, arbeiten die heterogenen Geräte eines verteilten Systems aufgrund verschiedener Rechenleistungen unterschiedlich schnell. Es ist möglich, dass Gerät A mehrere Rechenphasen durchführt, während Gerät B gerade einmal eine Phase beendet. Es ist ineffizient, wenn A aufgrund von B ausgebremst wird, weil die Informationen von B bereits völlig veraltet sind. Das langsamste Gerät könnte zum Flaschenhals werden. Zudem wäre eine Verfälschung beider Berechnungen möglich. Buzz löst diese Problematik, indem die Buzz-Schleife aller virtuellen Maschinen an ein festes Zeitfenster gebunden ist und allen Maschinen eine einheitliche Systemuhrzeit bekannt ist. Eine gemeinsame Uhr ermöglicht die totale Ordnung aller Ereignisse eines verteilten Systems. Eine zentrale Uhr ist jedoch unzulässig, siehe 1.2.1. Eine physikalische Uhr in jedem Gerät wäre ebenfalls fehlerbehaftet, da Uhren auf Dauer asynchron werden. Ein verteilter Mechanismus zur Synchronisation von Uhren könnte dieses Problem jedoch lösen. Buzz verwendet die sogenannte " LamportClock". Hierbei handelt es sich um einen vollständig dezentralen Mechanismus, der vom US-amerikanischen Mathematiker und Informatiker Leslie Lamport entwickelt wurde. Der von Lamport entwickelte Algorithmus [17] ist heute Grundlage zur Implementierung von verteilten Systemen.

2.3.1 Die logische Lamport Uhr

„Ein verteiltes System besteht aus einer Sammlung von getrennten Prozessen, die räumlich verteilt sind und miteinander durch Nachrichtenaustausch kommunizieren“[17]. Die

Tatsache, dass zwischen Versand und Ankunft einer Nachricht ebenso Zeit vergeht wie zwischen zwei internen Ereignissen eines Prozesses, entspricht der konzeptuellen Idee einer Uhr. Lamport entwickelt eine verteilte logische Uhr, die an diesen Grundgedanken ansetzt und eine totale Ordnung aller Ereignisse bei globaler Betrachtung des Systems ermöglicht. Mithilfe dieser Relation beschreibt er einen Mechanismus zur verteilten Synchronisation, auf dessen Basis weitere Algorithmen wie zum Beispiel verteilte Speicher aufbauen können. Problematisch ist jedoch die Ausfallsicherheit von verteilten Systemen, die auf einer logischen Uhr basieren, weil die Synchronisation verteilter Ressourcen eine Beteiligung aller Prozesse erfordert, siehe 2.4.1. Ob eine Komponente ausgefallen ist, kann durch eine logische Uhr nicht beurteilt werden. Außerdem können Zeitdifferenzen zwischen zwei innerhalb des Systems unabhängigen Ereignissen stark im Widerspruch zur tatsächlichen bzw. physikalischen Zeitdifferenz des Systems stehen. Daher kann die logische Uhr durch eine physikalische Uhr ersetzt werden, um Fehler zu minimieren. Bei der logischen Uhr handelt es sich lediglich um eine lokale Funktion innerhalb eines Prozesses, die jedem Ereignis einen diskreten Wert zuweist. Zwischen jedem Ereignis wird dieser Wert erhöht. Ein Ereignis kann lokal, bspw. in Form einer internen Berechnung, oder durch Einfluss von außen, dem Empfang einer Nachricht, auftreten. Die " Clock-Condition"[17] ist dabei eine Invariante, die eine partielle Ordnung $\rightarrow$ zwischen voneinander abhängigen Ereignissen definiert. Auf dieser Basis kann eine totale Ordnung aller Ereignisse im System festgelegt werden. Die Clock-Condition legt fest, dass ein Event b, das auf ein Event a folgt, zu höherer Uhrzeit als a auftritt, sodass die partielle Ordnung $a \rightarrow b$ gilt. Für lokale Ereignisse ist diese Bedingung durch das „Ticken“ ,bzw. Erhöhen des Variablenwerts, bereits erfüllt. Damit die Invariante auch für abhängige Ereignisse zwischen verschiedenen Prozessen gilt, senden alle Prozesse in jeder Nachricht einen Zeitstempel, die aktuelle Uhrzeit, mit. Die Empfänger-Uhr wird dabei nur aktualisiert, sofern die Uhr des Absenders schneller tickt. Ist der Zeitstempel des Absenders größer oder gleich dem aktuellen Wert der Uhr, muss die lokale Uhr über den Wert des Zeitstempels hinaus erhöht werden, weil eine Nachricht nur empfangen werden kann, nachdem diese versendet wurde. Dadurch bleibt die Clock-Condition erfüllt. Die daraus abgeleitete partielle Ordnung setzt ausschließlich lokal voneinander abhängige Ereignisse, also interne Ereignisse und durch Nachrichten verknüpfte Events, in Relation. Die Clock-Condition hält diese Ordnung lokaler Ereignisse aufrecht und ermöglicht das Festlegen einer beliebigen totalen Ordnung unter Berücksichtigung der partiellen Ordnung. Zur Festlegung einer totalen Ordnung definiert Lamport die Relation => . Eine Gleichzeitigkeit von Ereignissen kann dabei vermieden werden, indem die Wertebereiche aller Uhren disjunkt mit Hilfe von IDs implementiert sind.

Veranschaulichung der totalen Ordnung

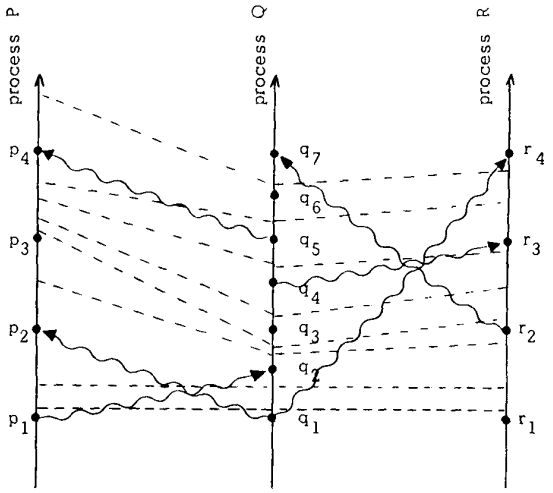

Abbildung 2.1: Raum-Zeit Diagramm [17]

Bild 2.1 zeigt das Auftreten lokaler Ereignisse, als indizierte Punkte `p`, `q` und `r`, innerhalb der Prozesse `P`, `Q` und `R`. Der kontinuierliche Verlauf der Zeit ist dabei durch die vertikalen Prozesslinien dargestellt, während dessen räumliche Trennung durch die Horizontale repräsentiert wird. Nachrichten sind durch geschlängelte Pfeile dargestellt, während gestrichelte Linien das Ticken der logischen Uhren repräsentieren. Bei Betrachtung der lokalen Ereignisse fällt auf, dass die Clock-Condition hierfür erfüllt ist: zwischen jedem lokalen Ereignis wurde die logische Uhr mindestens einmal erhöht. Man betrachte nun Prozess `Q`, der in Ereignis q_5 eine Nachricht zum Zeitpunkt sechs an Prozess `P` sendet. Angenommen `P` beendet Ereignis p_3 zum dargestellten Zeitpunkt Drei und erhöht die Uhr auf Vier. Jetzt trifft die Nachricht von `Q` mit dem Zeitstempel Sechs ein. Durch die Nachricht sind q_5 und p_4 voneinander abhängig. q_5 kann aber nicht nach p_4 auftreten, da so die Clock-Condition verletzt wäre. Somit erhöht `P` seine Uhr auf den Wert Sieben, sodass die partielle Relation $q_5 \rightarrow p_4$ gemäß der Clock-Condition erfüllt ist. Bei Betrachtung aller Nachrichten in Bild 2.1 fällt auf, dass die partiellen Ordnungen aller Ereignisse korrekt dargestellt sind: Eine gestrichelte Linie schneidet jeden Nachrichtenpfeil und verläuft zwischen jedem lokalen Ereignis. Angenommen es gilt nun `P < Q < R` und die Zeitpunkte aller Ereignisse sind dem Bild entsprechend festgelegt. Dann entsteht folgende totale Ordnung: $p_1 => q_1 => r_1 => p_2 => q_2 => p_3 => r_2 => q_3 => q_4 => q_5 => r_3 => p_4 => q_6 => q_7 => r_4$. Es fällt auf, dass das Ticken der Uhren willkürlich implementiert ist, weil zwischen q_1 und q_2, ebenso wie zwischen r_1 und r_2, mehrere Zeiteinheiten vergehen, obwohl eine Zeiteinheit zur Erfüllung der Clock-Condition ausreichend gewesen wäre. Damit ist auch die totale Ordnung willkürlich, weil z.B. sowohl $p_3 => r_2$ als auch $r_2 => p_3$ gelten kann.

2.3.2 Synchronisation globaler Ressourcen

Lamport nutzt die mit Hilfe der Clock-Condition entstehende totale Ordnung zur Beschreibung eines Protokolls, um die Belegung verteilter Ressourcen zu synchronisieren. Die Belegung einer Ressource s erfolgt unter folgenden Bedingungen: Ein Prozess muss Ressource s freigeben, bevor ein anderer Prozess die gleiche Ressource belegen kann. Unterschiedliche Anfragen belegen s in der Reihenfolge, in der diese getätigt wurden und jeder Prozess wird

die belegte Ressource s irgendwann wieder freigeben. Die letzte Bedingung entspricht einer Liveness-Bedingung, wie in 1.1.1 skizziert. Unter bestimmten technischen Voraussetzungen beweist Lamport die Korrektheit eines Algorithmus, der eine solche verteilte Synchronisation von gemeinsamen Ressourcen ermöglicht. Die Voraussetzungen hierfür sind, dass alle Prozesse Informationen direkt miteinander austauschen können und Nachrichten in der Reihenfolge empfangen werden, in der diese versendet wurden. Der Algorithmus funktioniert auf Basis einer Lamport-Uhr grob skizziert nun so, dass ein Prozess P_i eine Ressourcenbelegungsanfrage s_i an alle Prozesse versendet und s_i in einer Warteschlange lokal abspeichert. Ein Prozess P_j, der diese Anfrage empfängt, speichert die Anfrage s_i ebenfalls und sendet eine Bestätigung an P_i zurück, sofern keine ältere Anfrage an s vorliegt. P_i kann s schließlich belegen, wenn eine Bestätigung aller anderen Prozesse eingegangen ist und s_i an erster Stelle der Warteschlange steht. Zum Freigeben der Ressource löscht P_i s_i aus der Warteschlange und broadcastet eine entsprechende Nachricht, woraufhin alle Prozesse s_i aus der Warteschlange zur Belegung von s entfernen. Mit diesem Mechanismus kann eine Synchronisation von verteilten Speichern realisiert werden.

2.4 Verteilte Speicher

Ein verteilter Speicher ist eine Menge von gemeinsamen Ressourcen bzw. Variablen, die in einem Netz von Geräten global bekannt sind. Es gibt verschiedene Mechanismen, um verteilte Speicher zu implementieren.

2.4.1 Der verteilte Speicher in Buzz

Wie in Abschnitt 2.2 vorgestellt, verfügt Buzz über einen verteilten Speicher in Form der `stigmery`-Datenstruktur. Buzz verwendet jedoch keine verteilte Synchronisation. Somit ist auch keine globale Konsistenz der `stigmery`-Struktur zu jedem Zeitpunkt garantiert. Der technische Kontext von Buzz erschwert jedoch auch eine Synchronisation des verteilten Speichers, denn Roboter können Nachrichten nur innerhalb einer gewissen Reichweite broadcasten. Räumlich zu weit entfernte Roboter können also nicht direkt miteinander kommunizieren. Direkte Kommunikation zwischen allen Geräten ist laut Lamport jedoch Voraussetzung für den in 2.3.2 skizzierten Algorithmus. Buzz realisiert dennoch einen globalen Broadcast durch „Fluten“ [16] des Systems mit Nachrichten. Hierbei broadcasten Nachbargeräte erneut, wenn auch diese die stigmery-Struktur aufgrund der entsprechenden Nachricht aktualisiert haben. Auf diese Weise wird der neue Eintrag in einer beliebig großen zusammenhängenden Menge von Robotern propagiert. Zwischenzeitlich kann derselbe key aber durch einen aktuelleren Eintrag überschrieben werden, während andere Geräte noch mit einem alten Eintrag arbeiten. Der Algorithmus von Lamport kann auch in diesem Kontext alle Zugriffe auf verteilte Ressourcen synchronisieren. Stellen zwei räumlich weit entfernte Prozesse Anfragen an eine Ressource, werden beide zunächst Bestätigungen

erhalten, aber nur einer wird eine positive Antwort aller Teilnehmer bekommen. An dieser Stelle fallen jedoch zwei Probleme auf: Jedem Gerät müssen alle weiteren Geräte bekannt sein und kein Gerät darf ausfallen. Das erste Problem könnte dadurch behoben werden, dass jeder neue Roboter die eigene ID global propagiert und schließlich eine Liste aller Roboter erhält. Für das zweite Problem kann nur ein zusätzlicher Mechanismus auf Basis physikalischer Uhren herangezogen werden, sodass der Ausfall erkannt und die Belegung fortgesetzt wird. Eine Synchronisation verteilter Ressourcen in Buzz ist jedoch im Allgemeinen unpraktisch. Roboter sind dynamisch im Raum verteilt. Somit können einzelne Geräte die kommunikativ zusammenhängende Menge aller Roboter jederzeit verlassen oder mehrere unabhängige Mengen davon bilden. Das ist vermutlich der Grund, warum Buzz einen unsynchronisierten (key, value)-Speicher verwendet, der von Linda [11] inspiriert ist. Die `stigmery`-Struktur kann zu beliebigen Zeitpunkten beschrieben werden. Ein aktueller Eintrag überschreibt stets einen älteren.

2.4.2 Der Paxos-Mechanismus

Lamport verwendete seine verteilte Uhr zur Entwicklung einer allgemeinen Konsens-Strategie in verteilten Systemen, dessen Mechanismus als „Paxos“ bezeichnet wird [18]. Die hier vorhandenen Variablen werden jeweils einmalig beschrieben. Es liegt quasi das Gegenteil der "virtual stigmery" aus Buzz vor. In Paxos sind Prozesse durch drei unterschiedliche Rollen am Konsens beteiligt. Sogenannte " Proposer" schlagen Werte für die gemeinsame Variable vor, „Akzeptoren“ verwalten die Werte der Variablen und " Learner" ermitteln deren Inhalt. An dieser Stelle sei die Funktionsweise sehr grob veranschaulicht: Das Paxos-Protokoll ist in zwei Phasen unterteilt. Für beide Phasen ist ein einziger Prozess im System als Proposer und Learner qualifiziert, um Liveness zu garantieren. Dieser Prozess muss als Proposer einen " prepare request" an eine Mehrzahl aller Akzeptoren mit einer Zahl n, i. d. R. dem Wert der Lamport-Uhr, senden. Jeder Akzeptor, der diese Anfrage empfängt, antwortet mit einem " promise", sofern dieser bisher keinen " prepare request" mit einer höheren Zahl als n erhalten hat, und dem bisher höchsten akzeptierten proposal, falls dieser überhaupt vorhanden ist. Ein proposal ist in diesem Fall eine Zahl $m < n$ und ein Wert v. Falls der Proposer einen " promise" von der Mehrzahl aller Akzeptoren erhalten hat, beginnt die zweite Phase. Der Prozess sendet dazu einen " accept request" mit der Zahl n und einem Wert v. v ist hierbei der Wert des bisher höchst-nummerierten proposals unter den Antworten oder ein beliebiger Wert, falls kein proposal in einem " promise" vorhanden war. Empfängt ein Akzeptor nun den " accept request", akzeptiert er diesen, sofern dieser nicht bereits einem höher-nummerierten " prepare request" geantwortet hat. Der dedizierte Prozess ist Proposer und Learner zugleich, weil beide Rollen analog miteinander verknüpft sind. Ein Proposer, der einen " promise" mit Wert v erhält, wird automatisch zum Learner. Ein Wert wird nur ermittelt oder gesetzt, sofern ein " promise" von der Mehrzahl aller

Akzeptoren empfangen wurde. Es ist jedoch möglich, dass eine ausreichend große Gruppe von Prozessen den Konsens bewusst manipuliert, sodass die verteilte Variable inkonsistent ist und damit zwei oder mehr Gruppen unterschiedliche Werte speichern. "Byzantine-fault-tolerant systems" können gegenüber solchen Prozessgruppen bis zu einer bestimmten Größe robust sein. Paxos ist jedoch nicht "Byzantine-fault-tolerant" und nimmt an, dass jeder Prozess den Konsens anstrebt. Prozesse dürfen in Paxos aber abstürzen, neustarten und mit unterschiedlichen Geschwindigkeiten arbeiten. Nachrichten dürfen zudem doppelt übertragen werden und verloren gehen. Problematisch bei Paxos ist allerdings, dass die Implementierung abhängig vom System sehr aufwendig sein kann. Eine Alternative zu Paxos ist „Raft“[4]. Der Algorithmus verspricht, eine sehr verständliche Konsens-Strategie zu verfolgen. Raft wird im Rahmen dieser Arbeit jedoch nicht genauer analysiert.

Kapitel 3

Das MicroPython-Framework

Verteilte Speicher sind ein mächtiges Werkzeug zur Umsetzung von Problemstellungen in verteilten Systemen. Variablen sind dabei in räumlich getrennten Speichermedien vorhanden, während die Abstraktion ein zentrales Speichermedium suggeriert. Das MicroPython-Framework stellt im Wesentlichen zwei Klassen zur Abstraktion von verteiltem Verhalten zur Verfügung. Bei einer der beiden Klassen handelt es sich um einen solchen verteilten Speicher, siehe 3.2.5. Die zweite Klasse ist eine Nachbardatenstruktur, siehe 3.2.4, die Funktionen zum Umgang mit lokalen Ereignissen implementiert. Eine weitere Klasse abstrahiert Funktionen zur Steuerung der Roboter in ROS, siehe 3.2.2. Weitere in ROS verfügbare Hardware wird über das Framework nicht angesprochen, da im Rahmen dieser Arbeit ein Robotermodell mit ausschließlich grundlegenden Eigenschaftsmerkmalen gebaut wird. Es besteht aus einem Quader als Grundkörper und vier drehbar gelagerten Zylindern. Darüber hinaus gibt es eine weitere Klasse, die als Schnittstelle zur Kommunikation zwischen den Geräten dient, siehe 3.2.3.

3.1 Der Name `uBabble`

Der Name des Frameworks ist von Buzz inspiriert und lautet „`uBabble`“. Dieser soll den Charackter verteilter Kommunikation hervorheben und stammt vom englischen Wort "babble". Die Kommunikation zwischen den Robotern ist jedoch, anders als der Name suggeriert, logisch und strukturiert. "Babble" wurde um den Präfix „u“ erweitert. Im Allgemeinen handelt es sich hierbei um Module, die aus Python für MicroPython portiert wurden und durch den „u“-Präfix als unvollständig gekennzeichnet sind [13]. Die in diesem Kapitel vorgestellte Bibliothek entstand allerdings allein im Rahmen dieser Arbeit und ist in keiner weiteren Sprache verfügbar.

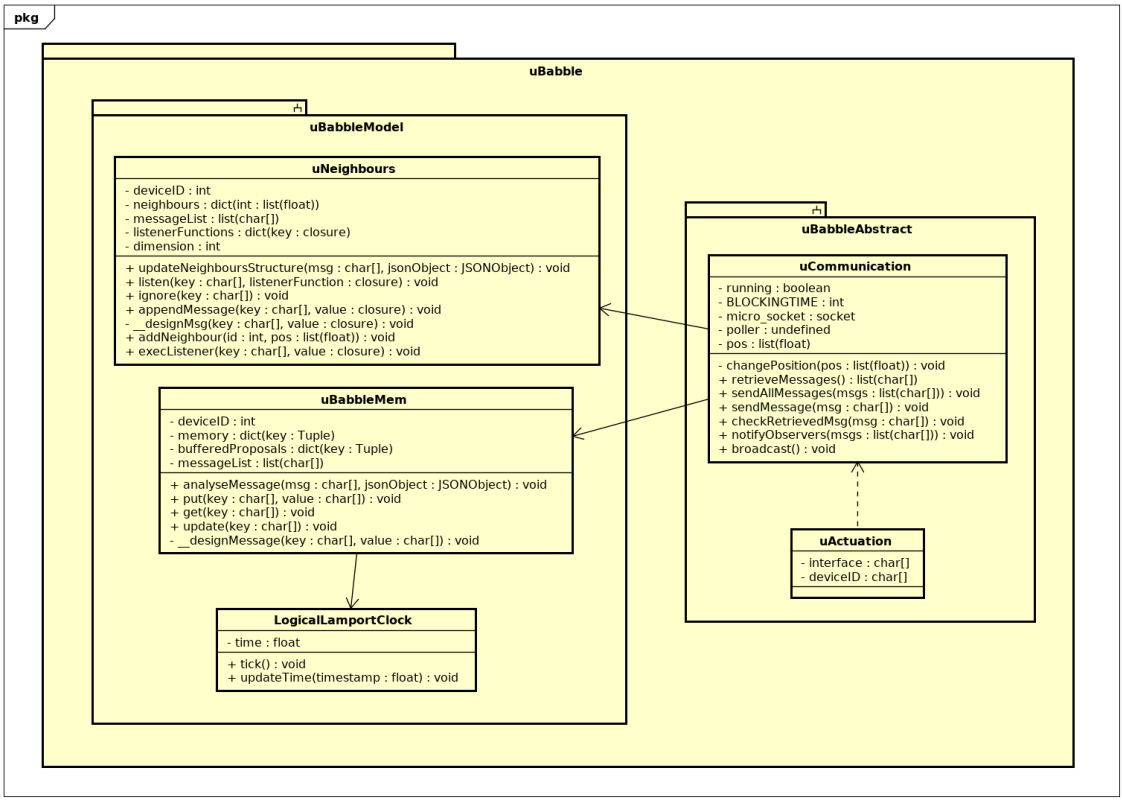

Abbildung 3.1: uBabble-Strukturmodell

3.2 Architektur des Frameworks

Das UML-Klassendiagramm in Bild 3.1 veranschaulicht die grundlegende Kernstruktur des Frameworks. Es zeigt die Klassen zur Abstrahierung verteilter Algorithmen im Paket `uBabbleModel` und ein weiteres Paket namens `uBabbleAbstract`. Dieses implementiert roboterspezifische Funktionalität. Jeder Roboter muss im Framework über eine Geräte-ID eindeutig identifizierbar sein.

3.2.1 Eine logische Lamport-Uhr

Die Uhr-Funktion der logischen Lamport-Uhr aus Abschnitt 2.3.1 wird mit der Klasse `LogicalLamportClock` implementiert. Die Methode `tick` erhöht den Wert der Uhr zwischen jedem lokalen Ereignis, während `updateTime` die lokale Uhr auf Basis eingegangener Nachrichten synchronisiert. Der Wertebereich der Uhr ist als Fließkommazahl implementiert, um durch Nachkommastellen die Wertebereiche aller Uhren disjunkt zu halten. Die Nachkommastellen entsprechen dabei der Geräte-ID. Auf diese Weise werden Konflikte vermieden.

3.2.2 Steuerungsfunktionen

Die Klasse `uActuation` implementiert Steuerungsfunktionen der Geräte. Im Paket `uBabble-Abstract` befindet sich ein von ROS abstrahierter Entwurf. Konkrete Funktionalität wird erst in einer darunterliegenden AbstractionLayer, vergleiche Abschnitt 1.1.4 und 2.2, definiert. Auf diese Weise kann ROS durch ein anderes System substituiert werden, indem eine entsprechende Klasse innerhalb der AbstractionLayer neu geschrieben wird und `uActuation` als Interface implementiert.

3.2.3 Schnittstelle zur Kommunikation

Die in Bild 3.1 dargestellte Klasse `uCommunication` ist Schnittstelle zur Kommunikation nach außen. Sämtliche Nachrichten werden innerhalb einer definierten Reichweite als Broadcast an naheliegende Geräte versandt. Eingegangene Nachrichten sind gepuffert und können über die Methode `retrieveMessages` abgerufen werden. Hierin werden zudem weitere Klassen im Paket `uBabbleModel` als Beobachter über die Funktion `notifyObservers` informiert. Es handelt sich dabei nicht um das klassische Beobachter-Entwurfsmuster der dynamischen Form, da lediglich zwei Klassen statisch als Beobachter vordefiniert sind. Die Methode `broadcast` leitet in den Beobachtern zwischengespeicherte Nachrichten über die Funktion `sendAllMessages` bzw. `sendMessage` an Nachbargeräte weiter.

3.2.4 Eine Nachbar-Datenstruktur

Die Klasse `uNeighbours` ist Beobachter der Klasse `uCommunication` und hält im Attribut `neighbours` ein Wörterbuch, in dem alle aktuellen Nachbarn gespeichert sind. Das Wörterbuch ist über Geräte-IDs indiziert und bildet auf die aktuellen Positionen der Nachbarn ab. Das Attribut `messageList` dient als Nachrichtenausgang aller innerhalb der Klasse generierten Nachrichten. Diese werden über die Methode `appendMessage` zur `messageList` hinzugefügt. Nachrichten aus der `uNeighbours`-Struktur werden, wie oben erwähnt, über die Methode `broadcast` der Klasse `uCommunication` an alle Geräte innerhalb der Reichweite weitergeleitet. Ein benachbartes Gerät verarbeitet die eingegangene Nachricht ebenfalls in der `uNeighbours`-Struktur mit der Methode `updateNeighboursStructure`. Hier wird das `neighbours`-Attribut aktualisiert. Des Weiteren werden mit der Nachricht über einen Key verknüpfte Listener-Funktionen ausgeführt. Dafür sind Nachrichten in `uNeighbours` json-strukturiert und enthalten in der Variable "data" ein «key, value»-Paar. Listener-Funktionen werden jedoch nur ausgeführt, sofern Funktionen in der Variable `listener-Functions` definiert sind. Eine Listener-Funktion wird über die Methode `listen` zum `listenerFunctions`-Attribut hinzugefügt, indem ein Key und eine Closure-Variable als Parameter übergeben werden. Die closure-Variable ist dabei die mit dem Key verknüpfte Funktion. `updateNeighboursStructure` führt alle Funktionen aus, die mit dem Key in Data verknüpft sind und übergibt zudem den dahinter liegenden Value als Parameter

an die Funktionen weiter. Über einen Aufruf der Methode `ignore` können alle mit einem Schlüssel verknüpften Listener-Funktionen gelöscht werden.

3.2.5 Ein verteilter Speicher

Der in `uBabble` verfügbare verteilte Speicher heißt `uBabbleMem` und ist, ebenso wie die `uNeighbours`-Struktur, maßgeblich von Buzz inspiriert [32]. Die Datenstruktur ist im Attribut `memory` als Wörterbuch implementiert. Universelle Schlüssel werden auf ein Tupel abgebildet. Dieses enthält Zeitstempel, Geräte-ID und einen Wert. Der Zeitstempel ist der Wert der Lamport-Uhr zum Zeitpunkt des lokalen Schreibvorgangs. Die Geräte-ID dient als Referenz auf das Gerät, das den Eintrag geschrieben hat. So soll mit jedem Schreibvorgang eine Identität verknüpft sein. Ebenso wie in Buzz, überschreibt jeder neue Eintrag einen bereits vorhandenen älteren. Als Variablen des Speichers dienen die Schlüssel des Wörterbuchs. Ein neuer Wert wird über die Methode `put` in den Speicher geschrieben. Falls der dabei angegebene Schlüssel noch nicht vorhanden ist, wird dieser angelegt. Die Methode `get` gibt den Wert eines Schlüssels aus. Dabei wird der lokal bekannte Wert sofort zurückgegeben. Weil dieser veraltet sein kann, werden naheliegende Geräte im Anschluss nach einem aktuelleren Wert gefragt. Die Methode `analyseMessage` überprüft eingegangene Nachrichten, die an den verteilten Speicher addressiert sind. Der Aufruf erfolgt analog zur `uNeighbours`-Struktur über eine Beobachter-Variable in der `uCommunication`-Schnittstelle. `analyseMessage` informiert Nachbarn mit veralteten Einträgen und aktualisiert das Speicher-Attribut. Beim Überschreiben eines Schlüssels werden Nachbarn allerdings nur informiert, wenn der eigene Eintrag älter als der aktuelle ist. Auf diese Weise werden Speichereinträge global propagiert. Der Mechanismus ist identisch zum Protokoll in Buzz, siehe 2.4.1, und die naheliegendste Lösung, da Nachrichten ausschließlich an alle Geräte im lokalen Umfeld versendet werden.

3.3 Hardwarekontext

3.3.1 Threading in uBabble

Wie in Abschnitt 1.2.2 dargestellt, soll MicroPython-Code unter anderem in ein System auf STM32-Basis implementiert werden. Dazu wird ein Mikrocontroller mit ARM-CPU verwendet. Bei STM32- bzw. Mikrocontroller-Systemen handelt es sich jedoch in der Regel um Einzweck-Hardware, die kein Multithreading unterstützt. Daher wird das uBabble-Strukturmodell in Bild 3.1 für eine Ausführung in sequentiellen Systemen ausgelegt. Das Beobachtermuster der Klassen `uBabbleMem` und `uNeighbours` mit `uCommunication` liegt hier nahe, da eingegangene Nachrichten sofort an die entsprechenden Beobachterfunktionen `analyseMessage` und `updateNeighboursStructure` weitergeleitet werden. Einem parallelen System würde ein Producer-Consumer-Entwurf zusprechen. Dieser benötigt weniger

Speicher und wäre daher insbesondere in einem Mikrocontroller zu bevorzugen. Die in dieser Arbeit verwendete ARM Cortex-M3 CPU unterstützt sogar Multithreading. Threads sind allerdings nicht auf dem entsprechenden MicroPython-Port verfügbar.

3.3.2 Vorhandene Rechenleistung

Der ARM Cortex-M3 kommt hauptsächlich in Mikrocontrollern vor und ist für Echtzeitaufgaben optimiert. Einem üblichen System stehen allerdings nur wenige Kilobyte Arbeitsspeicher bei einer Taktfrequenz von wenigen Megahertz zur Verfügung. Das Framework muss daher klein genug sein, um in ein System dieser Größe zu passen.

3.4 Anbindung an ROS

ROS-Bibliotheken können in der aktuellen Version "Melodic Morenia" nur in C++, Python 2 oder Java verwendet werden. Ein Portierung von ROS nach MicroPython ist sehr aufwendig und umständlich. Zudem wird das benötigte Linux-System auf der verwendeten ARM-CPU nicht ausführbar sein. Die Anbindung von ROS erfolgt im Rahmen dieser Arbeit daher über zwei Schnittstellen. Auf MicroPython-Seite wird das Modul `uActuation` im Paket `uBabbleROSAbstractionLayer` verwendet. Dieses implementiert Funktionen des `uActuation`-Interfaces aus Abschnitt 3.2.2 und sendet Steuerungsbefehle an einen ROS-Node namens `vehicle_interface`. Der Node besteht aus zwei parallel arbeitenden Threads in den Klassen `VehicleControl` und `GazeboListener`, siehe Bild 3.2, und nutzt ROS-Bibliotheken. Beide Klassen können daher beliebige Informationen abrufen sowie die Motorensteuerung ansprechen. Über eine physikalische Schnittstelle zur Inter-

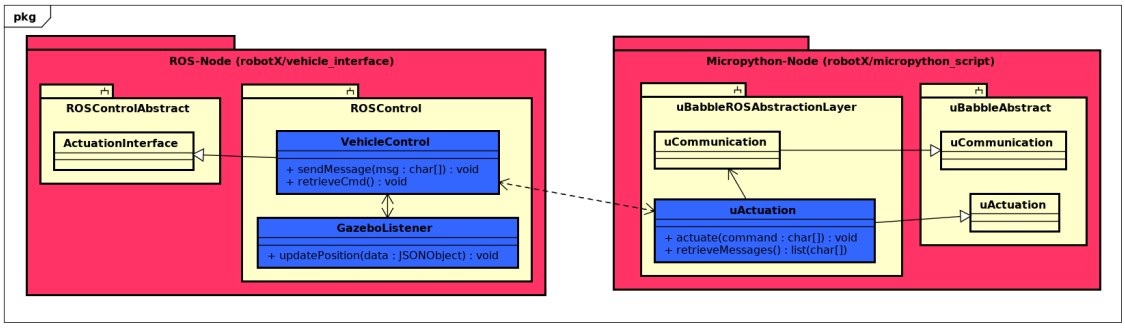

Abbildung 3.2: uBabble-ROS Schnittstelle

Prozess-Kommunikation, kurz IPC, tauschen ROS und MicroPython-Framework Daten aus. Die Schnittstellen sind als Attribut in den Klassen `uActuation` und `VehicleControl` vorhanden. Das Framework sendet Steuerungsparameter an `VehicleControl`, während der `GazeboListener` das Framework über die aktuelle Position und Orientierung des Roboters in der Gazebo-Welt informiert.

3.4.1 Gemeinsame Daten

Koordinaten

Alle Roboter befinden sich im Rahmen dieser Arbeit in einer Gazebo-Simulation, siehe 2.1.2. Da der `GazeboListener` die dem Framework bekannte Roboterposition regelmäßig aktualisiert, sendet die `uCommunication`-Schnittstelle diese Position als Headerinformation in allen Nachrichten mit. Somit wird auf Sensoren verzichtet und Roboter kennen zu jeder eingegangenen Nachricht die Koordinaten des Absenders. In der hier vorliegenden Gazebo-Simulation basiert die Authentizität dieser Informationen daher auf Vertrauen zwischen den Geräten. In der Realität werden oft sogenannte " situated communication devices"[32] verwendet, die den Ursprungsort eines Signals bestimmen können.

Steuerungsparameter

Die `uActuation`-Steuerungsschnittstelle sendet alle in ROS benötigten Parameter zur Klasse `VehicleControl`. Diese leitet die Parameter über eine ROS-Topic an den "differential drive controller"[30] weiter. Dieser übersetzt den Befehl in eine Bewegung, bei der "joints" bedient werden, die im urdf-file des Roboters, siehe 2.1.1, definiert sind. Die Steuerungsparameter müssen also der Abstraktionsschicht des Frameworks bekannt sein. Bei Verwendung eines neuen Roboters in Kombination mit einem anderen Steuerungsprogramm muss die AbstractionLayer des Frameworks im `uActuation`-Modul angepasst werden. Zur Vereinfachung der Anpassung dienen die Abstraktionsschichten `uBabbleAbstract` auf MicroPython-Seite und `ROSControlAbstract` auf ROS-Seite. Beide Pakete sind jedoch schwach abstrahiert und erfüllen insbesondere auf ROS-Seite keine bedeutenden Funktionen, weil soweit nur eine homogene Menge von Robotern des gleichen Typs verwendet wird.

3.4.2 Physikalische Schnittstelle

Die physikalische Anbindung des Frameworks an ROS ist durch Netzwerksockets realisiert. Hierbei handelt es sich um IPv4-Sockets, sodass Framework und ROS, entsprechend der Argumentation in 3.4, in zwei verschiedenen netzwerk-fähigen Systemen ausgeführt werden können. Denn wie in Abschnitt 3.3.2 beschrieben, muss das System klein genug sein, um von einem Mikrocontroller ausgeführt zu werden. Eine Aufteilung von ROS und MicroPython-Code in unterschiedliche Hardware bietet sich daher an. Zum Transport der Nachrichten wird das TCP-Protokoll verwendet. Der Grund hierfür wird in Kapitel 4.3.3 erläutert. Des Weiteren ist die in Bild 3.2 dargestellte Verbindung zwischen `uActuation` und `VehicleControl` nur indirekt vorhanden. Hintergrund ist die parallele Simulation mehrerer Roboter auf einem Ubuntu-Hostsystem. Alle Programme belegen dabei die Ports des Host-Systems, sodass vorab keine Standardports festgelegt werden können. Mit dem ROS-

Master startet daher ein zentraler Server names `connection_server` als ROS-Node. Über einen Anmeldevorgang identifiziert dieser zusammengehörige ROS- und MicroPython- bzw. uBabble-Instanzen. Ist der Anmeldevorgang erfolgreich abgeschlossen, werden Nachrichten zwischen beiden Prozessen vom Server direkt weitergeleitet. Sobald aber jedem Roboter und Mikrocontroller eine eigene Hardware zur Verfügung steht, kann dieser Server wegfallen.

3.5 Nachrichtenaustausch in uBabble

Mithilfe der `uCommunication`-Schnittstelle, siehe 3.2.3, informiert jeder Roboter nahegelegene Geräte über einen Broadcast. Die Klasse `uCommunication` nähert so die Kommunikationweise zwischen den Geräten in der Simulation einer Anwendung im realen Umfeld an. In der Realität könnten " situated communication devices" auch durch eine Kombination von WLAN und GPS-Sensor ersetzt werden, sodass ebenfalls eine Form der " situated communication" [32] vorliegt. Innerhalb der Gazebo-Simulation wird jedoch keine spezielle Hardware verwendet. Der Nachrichtenaustausch findet, ebenso wie in Abschnitt 3.4, über TCP-Sockets statt. Jedes Gerät sendet dabei Nachrichten an einen weiteren zentralen Server, der ebenfalls mit dem ROS-Master als `babble_server`-Node gestartet wird. Dieser analysiert die Position einer eingegangenen Nachricht und berechnet alle dazugehörigen Nachbarn auf Basis der Kommunikationsreichweite des Absenders. Dem Server sind hierfür die Signalreichweiten und Positionen aller Geräte bekannt. Mit jeder eingegangenen Nachricht werden alle Nachbarn des Absenders identifiziert und die Nachricht an diese weitergeleitet. An dieser Stelle sei angemerkt, dass weder der `connection_server` aus 3.4.2 noch der `babble_server` weitere Aufgaben übernehmen und insbesondere in keiner Weise die Prozesse bei verteilten Analysen oder Lernprozessen unterstützen. Die einzige Aufgabe beider Server liegt darin, einen Datenaustausch über Netzwerksockets im Rahmen der Gazebo-Simulation zu ermöglichen.

Kapitel 4

Eine Bibliothek in zwei Systemen

4.1 Das Unix-System

Die MicroPython-Bibliothek soll im Rahmen dieser Arbeit in heterogenen Geräten bzw. zwei verschiedenen Systemen verfügbar sein. Dazu wurde das Framework zunächst auf dem Unix-Port von MicroPython in einem Ubuntu-System entwickelt. Der Port kann auf allen heutigen Betriebssystemen verwendet werden, die Unix-Konzepte implementieren. Unix ist ein Betriebssystem dessen Entwicklung im Jahr 1969 begann. Es hat später die modulare Struktur von IT-Systemen sowie die grundlegenden Technologien zur Entstehung weiterer Betriebssysteme begründet [35]. Nahezu alle heutigen Mehrzweck-Betriebssysteme implementieren Unix-Konzepte. Zur Technologie gehören beispielsweise das hierarchische Dateisystem, virtuelle Dateisysteme, Gerätetreiber und Pipes. Der Unix-Port des Frameworks wird daher in einem Ubuntu-System entwickelt. Dieses stellt als System der Linux-Familie alle Unix-Technologien bereit und kann zudem ROS verwenden.

4.2 Befehlssatz-Architekturen

Betriebssysteme stellen einer Computersoftware Abstraktionen der Hardware zur Verfügung und verwalten diese. Ein Betriebssystem muss dafür vollständig mit der Hardware kompatibel sein. Zentrales Element einer Computerhardware ist der Prozessor. Dieser koordiniert weitere angeschlossene Hardware sowie sämtliche Berechnungen. Es gibt für CPUs unterschiedliche Paradigmen der Befehlssatz-Architektur, kurz ISA - Englisch für " Instruction-Set-Architecture". Die Befehlssatz-Architektur eines Prozessors definiert den Maschinenbefehlssatz und die Ausführung von CPU-Anweisungen im Steuer- und Rechenwerk. Programmiersprachen mit Compiler können Programmcode bis zum Binärcode dieser Maschinenbefehle übersetzen. Die Befehle sind Anweisungen der Programmiersprache Assembler. Die Menge dieser Anweisungen ist durch die ISA einer CPU festgelegt. ISAs werden nach unterschiedlichen Kategorien klassifiziert. Im Bereich der klassischen

Personal-Computer sind CISC-Architekturen verbreitet. Hier benötigt ein Maschinenbefehl mehrere Bus-Zyklen zur Ausführung im Prozessor. CISC-Architekturen eignen sich daher vor allem in Geräten, in denen das Rechenwerk deutlich schneller als der Hauptspeicher ist [22]. Durch viele Bus-Zyklen kann so die Last in Richtung CPU verlagert werden. In eingebetteten Systemen ist diese Diskrepanz hingegen nicht sehr stark ausgeprägt, sodass Prozessoren mit einer sogenannten RISC-Architektur verbreitet sind. Hier werden maschinelle Anweisungen in sehr wenigen Rechenzyklen ausgeführt. Bekannt für die Implementierung von RISC-Architekturen sind ARM-Prozessoren, wohingegen CISC-Anweisungen vor allem durch AMD- und Intel-Prozessoren verbreitet sind [10]. Darüber hinaus gibt es weitere Klassifikationen von Befehlssatz-Architekturen wie VLIW, LIW, MISC, OISC und EPIC [39]. Das vorliegende Ubuntu-System verfügt über eine Intel Core i7 CPU in CISC-Architektur. Somit muss MicroPython-Code im Rahmen dieser Arbeit auf dem Unix-Port in einer CISC-ISA interpretierbar sein. Für eine Ausführung von MicroPython-Code in einer RISC-Architektur kann das Pyboard mit ARM-Prozessor herangezogen werden. Hierbei handelt es sich um einen Mikrocontroller mit vielen nützlichen Funktionen, der mit MicroPython über den STM32-Port programmierbar ist. Das Schreiben von Compilern für RISC-Architekturen ist im Allgemeinen allerdings schwieriger, da entsprechende Befehlssätze in der Regel deutlich kleiner und fundierter sind. Somit erhöht sich die Zahl maschineller Anweisungen für eine RISC-Architektur im Vergleich zu einem CISC-Befehlssatz [6]. Dennoch haben RISC-Architekturen große Vorteile gegenüber CISC: Sie sind energiesparender und besser für Echtzeitdatenverarbeitung geeignet, da die Instruktionen mehr Spielraum für Compiler bieten, um Programmimplementierungen zu optimieren - vergleiche Abschnitt 1.1.1. Außerdem kann eine RISC-Pipeline mehrere Anweisungen parallel ausführen, wenn jeder Maschinenbefehl lediglich einen Zyklus benötigt und das Rechenwerk in isolierte Bereiche unterteilt ist. Moderne Prozessoren sind nur noch bedingt als RISC oder CISC kategorisierbar, da in heutigen Architekturentwürfen das Optimum aus beiden Welten gesucht wird.

4.3 MicroPython-Code in heterogenen Systemen

Programmbibliotheken basieren im Allgemeinen auf weiteren Bibliotheken, sodass eine komplexe Bibliothek aus einer Hierarchie und einem Netz von externen Modulen zusammengesetzt sein kann. Die Programmiersprache Java ist vor allem deshalb weit verbreitet, weil alle Bibliotheken jedem Java-fähigen System verfügbar gemacht werden können. Beim Entwurf einer Java-Bibliothek für heterogene Hardware muss lediglich beachtet werden, dass eine JVM für alle Systeme verfügbar ist, siehe 1.1.1. MicroPython-Code wird hingegen in keiner virtuellen Maschine ausgeführt und hält zudem auf verschiedenen Ports unterschiedliche Bibliotheken bereit. Der Umfang aller MicroPython-Standardbibliotheken liegt allerdings im überschaubaren Bereich. Zudem sind ausschließlich grundlegende Mo-

dule wie `math`, `json`, `os`, `sys`, `time` oder `select` enthalten. Wie in Abschnitt 3.1 erwähnt, implementieren viele dieser Module außerdem lediglich eine Teilfunktionalität vergleichbarer Python-Module. Das Framework verwendet nur vier externe Bibliotheken und ist damit gut auf zwei verschiedene Systeme portierbar. Das in Kapitel 3 vorgestellte Framework wird daher in ein System mit CISC-Architektur auf dem Unix-Port und RISC-Architektur auf dem STM32-Port implementiert. STM32 bezeichnet eine Architektur von 32-Bit Schaltkreisen. In dieser Arbeit wird eine abgewandelte Form des Ports verwendet, der im Folgenden schlicht als ARM-Port bezeichnet wird.

4.3.1 Externe Bibliotheken im Unix-Port

Die vier auf dem Unix-Port verwendeten externen Bibliotheken sind die MicroPython-Module `usocket`, `ujson`, `utime` und `math`. Alle sind bereits standardmäßig auf dem Unix-Port von MicroPython verfügbar. Die Module `math` und `utime` werden für Berechnungen und Wartezeiten verwendet. Das Modul `ujson` ermöglicht die Speicherung von Informationen in JSON-Objekten. JSON ist ein Kürzel für die Bezeichnung „JavaScript Object Notation“ und ermöglicht die Konvertierung von Objekten in Strings und anders herum. Auf diese Weise können Browser-Skripte Objekte mit Servern austauschen, ohne dass Programmierer komplizierte Parsing-Methoden schreiben müssen. JSON kann in jeder Programmiersprache verwendet werden und benötigt eine geringere Bandbreite zum Austausch von strukturierten Daten über Netzwerke als XML-codierte Nachrichten. Das `usocket`-Modul implementiert Socket-Funktionalitäten. Jedes moderne Unix-System kann Sockets zur Kommunikation zwischen Prozessen in verschiedenen Transportsystemen nutzen. Die Klassen `uCommunication` und `uActuation` nutzen daher im Rahmen der Gazebo-Simulation die `usocket`-Bibliothek zum TCP-basierten Datenaustausch über die Server aus Abschnitt 3.4.2 und 3.5.

4.3.2 Das Mikrocontroller-System

Mit dem ARM- bzw. STM32-Port ist MicroPython-Code auch in einer RISC-Architektur ausführbar. Dafür wird eine Emulation des Pyboards verwendet [33]. Das Pyboard ist ein Mikrocontroller mit einer ARM Cortex-M4 CPU, dessen ISA „Thumb“ eine RISC-Architektur ist. Die CPU-Emulation wird in der virtuellen Umgebung eines Webbrowser-Prozesses durch den „Unicorn-CPU-Emulator“[25] ermöglicht. Dafür wird das Github-Projekt " MicroPython on Unicorn"[33] verwendet und um die uBabble-Bibliothek erweitert. Das Github-Projekt stellt eine Webseite mit der Emulation des Pyboards und eines MicroPython-Terminals zur Verfügung. In der Webseite wird allerdings kein ARM Cortex-M4 Prozessor, sondern ein ARM Cortex-M3 emuliert. Dieser ist einem Cortex-M4 sehr ähnlich und implementiert die gleiche ISA.

4.3.3 Externe Bibliotheken im ARM-Port

Das MicroPython-Framework benötigt auch auf dem ARM-Port die externen Module des Unix-Ports. Die Bibliotheken `utime` und `math` sind im ARM-Port bereits vorhanden, allerdings kann weder `ujson` noch `usocket` importiert werden. Die Funktionalität beider Module muss dem Framework allerdings verfügbar gemacht werden. Die `ujson`-Bibliothek ist unproblematisch, da diese nur Parsing-Funktionalitäten implementiert. Das Modul `usocket` stellt allerdings ein Problem dar, da es eine Netzwerkschnittstelle benötigt. Auf dem Pyboard ist eine solche Schnittstelle allerdings nicht vorhanden. Trotzdem muss Inter-Prozess-Kommunikation auch in irgendeiner Weise über das Pyboard möglich sein. Das Problem wird im Rahmen dieser Arbeit dadurch umgangen, dass das Pyboard in der JavaScript-Umgebung eines Webbrowsers emuliert wird. Als Netzwerkschnittstelle können hier JavaScript-WebSockets verwendet werden. Aus Sicherheitsgründen steht der virtuellen Umgebung eines Webbrowsers mit der Klasse WebSocket allerdings nur der TCP-Stack zur Verfügung. Deswegen sind auch die physikalischen Schnittstellen des Frameworks sowie `connection_server` und `babble_server` TCP-basiert implementiert. Somit kann der ARM-Port um Funktionalität der Module `ujson` und `usocket` erweitert werden. Auf dieser Basis wird das Framework in die Emulation eingebunden und kann dank des `connection_servers` erneut ROS-Systeme in Gazebo steuern.

4.4 uBabble auf dem Unix-Port

Das Framework wird, wie in Abschnitt 4.1 beschrieben, mit dem Unix-Port in eine Linux-Umgebung implementiert, auf der auch ROS verfügbar ist. Daher wird MicroPython-Code direkt in ROS ausgeführt, weil es ebenfalls Unix-kompatibel ist. Der Port kann gegenüber dem ARM-System ein deutlich höheres Leistungspotenzial abrufen. Der gesamten Simulation steht die Leistung einer Intel Core i7 CPU mit 16 Gigabyte Arbeitsspeicher zur Verfügung.

4.5 uBabble auf dem ARM-Port

Die fehlenden Funktionen der `ujson` und `usocket` Bibliothek werden dem Port nachträglich hinzugefügt. Das `ujson`-Modul kann durch ein Makro im Header-file des MicroPython-Ports aktiviert werden, da dieses als port-unabhängiges Modul bereits zur Verfügung steht. Die Socket-Funktionalität wird mit dem Modul `uBabbleSocket` neu implementiert. Das Modul wird über eine C-Datei, ein Headerfile und einen Eintrag im `mpconfigport`-headerfile zur MicroPython-Bibliothek hinzugefügt. Es stellt in der Klasse `socket`, verfügbar durch eine weitere C-Datei, die eigentliche Funktionalität bereit, indem eine Referenz des Moduls auf die Klasse verweist. Darin sind vier Funktionen definiert, die der MicroPython-Interpreter als Methoden oder Konstruktor der Klasse `socket` verwenden

kann. Zum Senden und Empfangen von Daten sind die Methoden `send` und `recv` definiert. Beim Aufruf der Socket-Methode `send` schreibt das Objekt Daten in den Speicher der Unicorn-CPU. Diese führt daraufhin eine Listener-Funktion in der JavaScript-Umgebung aus, welche die Informationen an einen WebSocket weiterleitet, der die Daten schließlich versendet. Ein Aufruf von `recv` erfolgt analog, indem eine Variable gelesen wird, mit dessen Speicherbereich ebenfalls eine JavaScript-Listener-Funktion der Unicorn-CPU verknüpft ist. Die JavaScript-Funktion prüft, ob Daten im WebSocket-Objekt zwischengespeichert sind und gibt diese an die C-Funktion zurück. Der Aufruf von `recv` ist hierbei nicht blockierend. Bild 4.1 veranschaulicht die Verknüpfungen zwischen Unicorn-Emulator und den umliegenden Systemen. Der Datenaustausch ist durch bidirektionale Pfeile dargestellt. Das Bild skizziert auf der linken und rechten Seite zwei CPU-Emulationen, die über WebSockets mit den Servern aus 3.4.2 und 3.5 verknüpft sind. In der ROS-Komponente sind alle ROS-Nodes enthalten die über Topics mit dem ROS-Master verknüpft sind. Der `babble_server` wird zwar mit dem ROS-Master gestartet, ist jedoch ansonsten in keiner Weise mit ROS verknüpft und wird daher in Bild 4.1 außerhalb von ROS dargestellt. Die Skizze

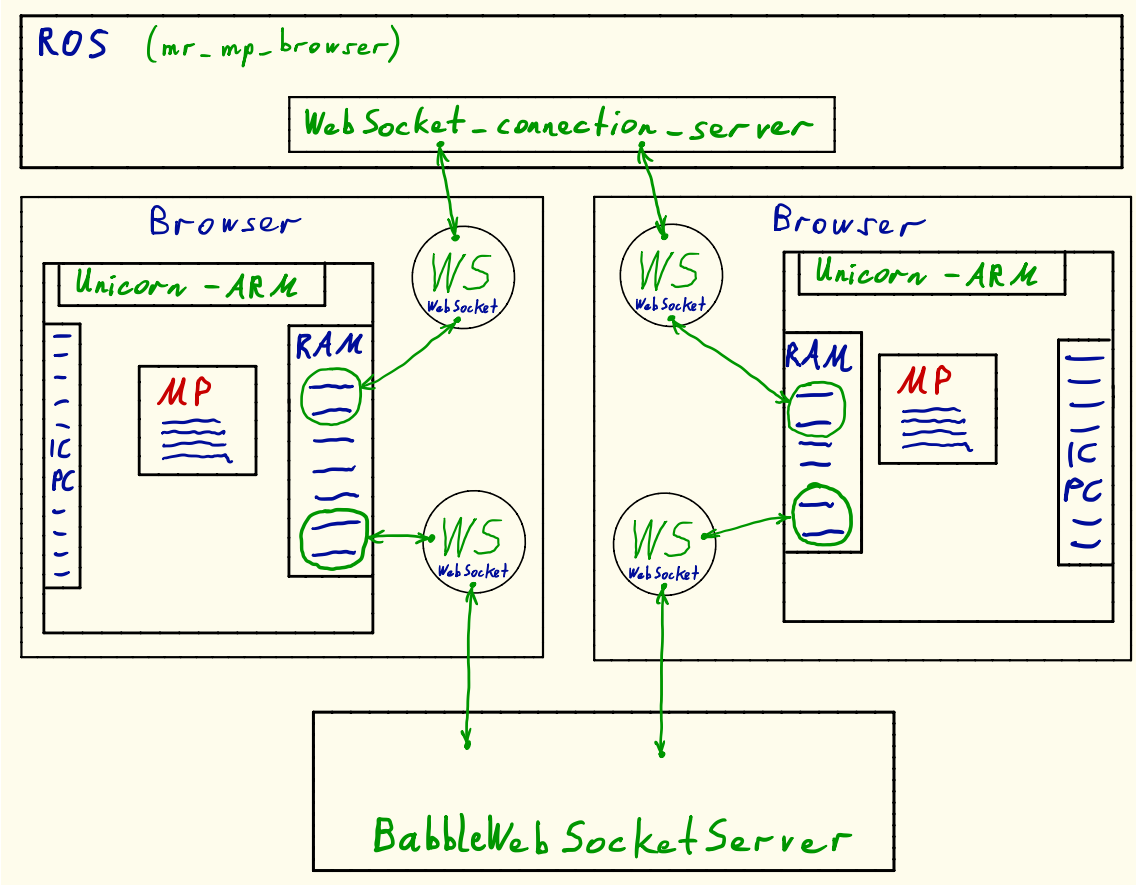

Abbildung 4.1: System der Unicorn-Emulation

stellt den MicroPython-Quellcode in den rot durch `MP` markierten Feldern dar. Der darun-

terliegende C-Code des Moduls `uBabbleSocket` kann auf, konkret mit der Unicorn-CPU verknüpfte, Speicherbereiche zugreifen, sodass die JavaScript-Listener-Funktionen Daten zwischen RAM und WebSockets austauschen. Der Vorgang ist im Bild durch die grünen Pfeile zwischen den `RAM`- und `WebSocket`-Komponenten dargestellt.

ARM- und Unix-Implementierung werden in Kapitel 6 durch zwei getrennte Gazebo-Simulationen demonstriert. Dazu soll im Folgenden eine konkrete Problemstellung zur Visualisierung eines verteilten Phänomens herangezogen werden. Die Simulation für den ARM-Port befindet sich im Catkin-package „mr_mp_browser“, während das Catkin-package für die Simulation im Unix-Port „mr_MP_ROS“ heißt. Im Anhang befindet sich unter Punkt A.2 eine kurze Anleitung zum Starten der in Kapitel 6 durchgeführten Simulationen. Dort befindet sich zudem eine detaillierte Beschreibung darüber, wie die Unicorn-Emulation um das `uBabbleSocket`-Modul für MicroPython erweitert wurde - siehe auch Quelle [24].

Kapitel 5

Eine algorithmische Problemstellung

Mit dem in Kapitel 3 beschriebenen Framework kann eine verteilte Problemstellung abstrahiert werden. Das gesamte bisher skizzierte System kann die algorithmische Umsetzung eines solchen Problems demonstrieren, indem der dazugehörige Programmcode in die beiden Umgebungen aus Kapitel 4 implementiert wird. Für eine Demonstration soll nun die gleichmäßige Verteilung von Robotern innerhalb einer leeren Gazebo-Welt herangezogen werden. Dazu wird das Problem im Folgenden aber zunächst unabhängig von Anwendung, Technologie und Kontext erläutert. Auf diese Weise soll eine Menge von wirtschaftlich relevanten Anwendungsbeispielen ermittelt werden. Im Folgenden soll die „gleichmäßige“ Verteilung von heterogenen Geräten im Raum betrachtet werden. Die Bezeichnung „gleichmäßig“ ist allerdings unpräzise und wird daher mathematisch formuliert. Zudem ist unklar, unter welchen Voraussetzungen die Geräte verteilt werden sollen. Um an dieser Stelle Rahmenbedingungen zu setzen, soll der Raum, in dem sich die Objekte verteilen, beliebige Dimensionen haben und zusammenhängend sein. Des Weiteren könnte neben einer rein räumlichen Verteilung auch eine Verteilung über diskrete mehrdimensionale Datenpunkte auf Intervallskalenniveau erfolgen. Durch eine „gleichmäßige“ Verteilung von dedizierten Objekten über einen solchen Datensatz können Muster in den vorliegenden Daten erkannt werden, sodass eine Datenmenge bspw. anhand einer Fragestellung interpretiert werden kann. Dieser Vorgang wird auch als Clustering von Daten bezeichnet und ermöglicht unter anderem die Implementierung moderner Spracherkennungssysteme. Im Kontext dieser Arbeit soll jedoch eine Demonstration der Verteilung von Robotern im Raum, ohne die Berücksichtigung weiterer Daten, ausreichen.

5.1 Mathematische Spezifikation

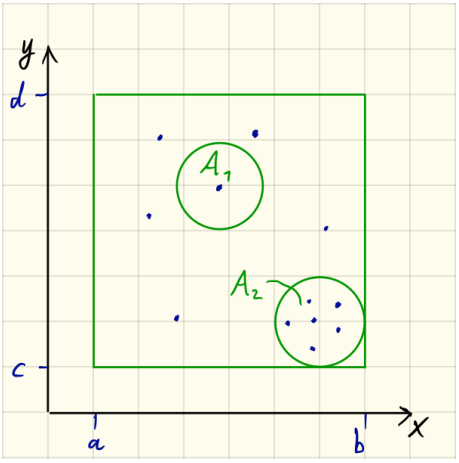

Abbildung 5.1: Unhomogene Verteilung

Zunächst wird die Bezeichnung „gleichmäßig“ genauer formuliert, sodass die Problemstellung spezifizierbar wird. Dazu wird das Problem in folgender Weise abstrahiert: n Bausteine bzw. Punkte sollen sich auf einer zweidimensionalen Karte möglichst „gleichmäßig“ in einem mathematisch schlichten Gebiet G der Grenzen $\bar{x} = [a, b] \wedge \bar{y} = [c, d]$ verteilen. Doch wie kann eine solche Verteilung aussehen? Man könnte z.B. eine gleichmäßige Verteilung im Sinne einer chemisch homogenen Verteilung fordern. Betrachtet man Abbildung 5.1 wird klar, dass der dargestellte Zustand nicht dem intuitiven Verständnis einer Gleichmäßigkeit entspricht. Die dargestellte Punktverteilung ist „nicht homogen, da die Bausteine [...] keine gleichmäßige Raumfüllung aufweisen“[38]. Die Fläche A_2 des Flächeninhalts ΔA enthält deutlich mehr Punkte als die ebenso große Fläche A_1. Versteht man als Kriterium für eine homogene Verteilung also beliebige Flächen A_i der Größe ΔA, die eine immer gleiche Anzahl z an Punkten enthalten, stellt man fest, dass eine chemisch homogene Verteilung nicht realisierbar ist, weil die Bausteine punktuell auf den Flächen verteilt sind und den Raum nicht vollständig ausfüllen. Somit findet sich immer mindestens eine Teilfläche, die mehr Punkte als die übrigen enthält. Für den zweidimensionalen Raum werden jetzt zwei Möglichkeiten betrachtet, um die Punkte gleichmäßig zu verteilen. Die erste Möglichkeit ist ein gleicher Abstand aller Punkte untereinander in x- und y-Richtung, wie in Abbildung 5.2 dargestellt.

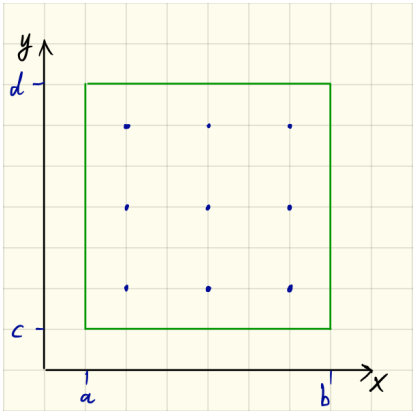

Abbildung 5.2: Gleicher Abstand in x- und y- Richtung

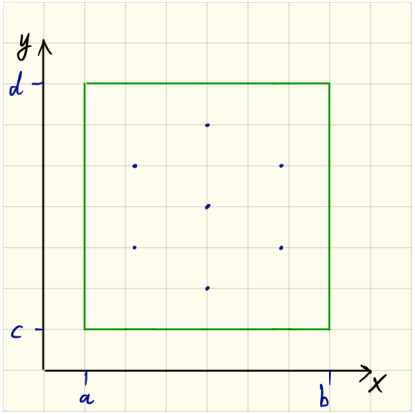

Abbildung 5.3: Äquidistante Verteilung

Hier müssen alle Punkte einen konkreten Abstand in x- oder y-Richtung zu benachbarten Punkten einhalten. Die Punkte müssen sich hierfür aber an einem globalen Koordinatensystem orientieren. Eine solche Orientierung ist in vielen Anwendungen schwer realisierbar. Ein dezentral einfacherer Ansatz ist es, euklidische Abstände für die Punkte untereinander festzulegen, sodass eine äquidistante Verteilung, siehe Abbildung 5.3, der Punkte entsteht. Weil es sich bei der Fläche und Anzahl der zu verteilenden Objekte in der Regel um gegebene Größen handelt, ist der Abstand zwischen allen benachbarten Bausteinen die gesuchte Größe. Es ist aber für den einzelnen Punkt umständlich, alle konkreten Nachbarn festzulegen oder überhaupt zu identifizieren, bei welchen Punkten es sich jetzt oder in Zukunft um Nachbarn handelt bzw. handeln wird. Damit wird die Bedingung an die gesuchte Lösung abgeschwächt und nur noch der größtmögliche Mindestabstand gesucht. Eine solche Verteilung ist mathematisch gut formulierbar:

$$\forall P_i, P_j \in G, \exists d_{min} \,|\, i, j \in \{1, ..., n\}, i \neq j \wedge G = [\bar{x}, \bar{y}] : euclideanDist(P_i, P_j) \geq d_{min} \tag{5.1}$$

Alle Punkte müssen den Mindestabstand einhalten, sodass kein Paar von Punkten P_i, P_j einen Abstand geringer als d_{min} zueinander hat. Weiterhin muss gelten, dass sich im Umkreis jeder beliebigen Position $s \in G$ mindestens ein Punkt P_i mit dem Abstand $a \leq \frac{d_{min}}{2}$ befindet:

$$\forall s \in G, \exists P_i \in G \,|\, i = \{1, ..., n\}, G = [\bar{x}, \bar{y}] : euclideanDist(s, P_i) \leq \frac{d_{min}}{2} \tag{5.2}$$

In einem verteilten System muss diese Aufgabe dezentral gelöst werden können. Für die hier betrachteten Punkte bzw. heterogenen Geräte sollen im Rahmen einer algorithmischen Lösung außerdem alle Eigenschaften gelten, die weitestgehend auch für KDubiq-Geräte charakteristisch sind, siehe hierzu Abschnitt 1.1.2. Dem entsprechend befindet sich jedes Gerät in einer zeitlich und räumlich veränderlichen Umwelt. Das heißt in diesem Fall, dass sich die Ränder des Gebiets ändern können. Weiterhin werden die Geräte eigenständig den Ort wechseln, dem verteilten Prozess beitreten oder diesen Verlassen. Zudem sind durch den Informationsaustausch zwischen den Geräten Echtzeitbedingungen gegeben, denn alle Bewegungen beeinflussen sich gegenseitig. Die Eigenschaft der Lokalität soll ebenfalls erfüllt sein, sodass eine Lösung allein mithilfe von Nachbarinformationen gefunden werden kann.

5.2 Mögliche Anwendungsbeispiele

Im Folgenden sollen ein paar Vorschläge für praktische Anwendungszwecke die Bedeutung des zu entwerfenden Algorithmus unterstreichen.

5.2.1 Sensoren

Sensoren können auf verschiedenen Frequenzen Ereignisse der realen Welt, wie beispielsweise Erschütterungen, Objekte und vieles mehr, erfassen. Durch die Verwendung mehrerer Sensoren, können Ereignisse präziser erfasst werden, indem bspw. das zu beobachtende Frequenzband unter den Sensoren aufgeteilt wird, siehe Abbildung 5.4. Gegebenenfalls können bestimmte Frequenzen durch mehrere Sensoren überprüft werden, sodass Messfehler erkennbar sind. Eine erfolgreiche Implementierung des gesuchten Algorithmus könnte eine diskrete Aufteilung der Frequenzbereiche unter den Sensoren realisieren, sodass jeweils ein Sensor auf seinen Frequenzbereich sensibilisiert ist, das gesamte Spektrum aber durch die Gesamtheit aller Sensoren abgedeckt wird. Die Technik könnte im Straßenverkehr Anwendung finden. Interessant ist hier auch die Fragestellung, wie Sensoren geographisch verteilt und mit unterschiedlichen Messwerten und Charakteristiken ein globales Ereignis erkennen können.

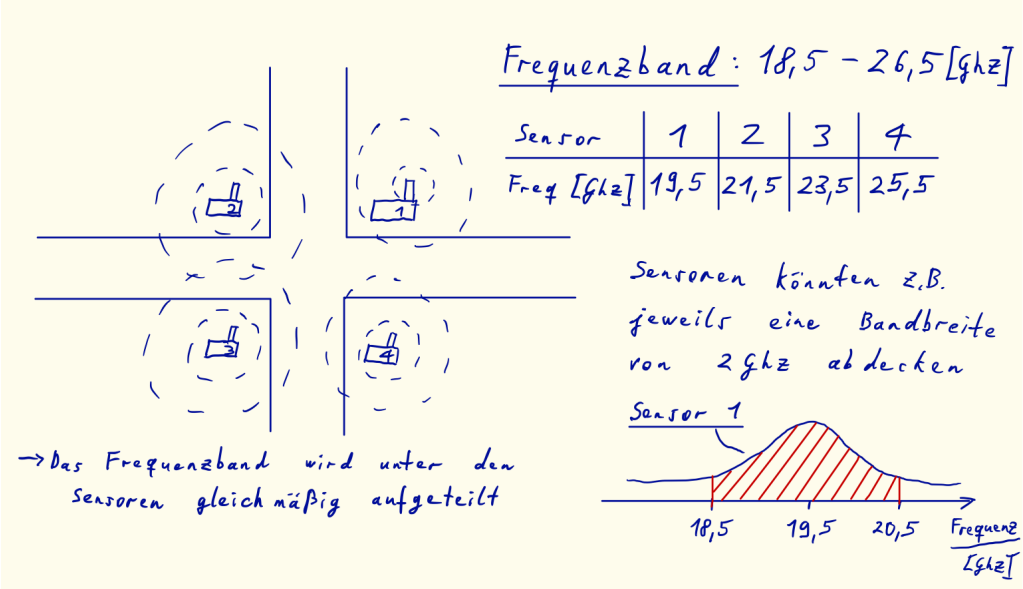

Abbildung 5.4: Frequenzband - Aufteilung

5.2.2 Paketdrohnen

In der Logistik könnte eine Verteilung von Ladestationen für Paketdrohnen berechnet werden, sodass diese optimal ausgelastet und Lieferwege möglichst kurz sind. Hierbei berechnet eine neue Drohne eigenständig ihre Position. Beispielsweise könnten Ladestationen so positioniert werden, dass diese gleichmäßig in einer Stadt verteilt sind und so für den Kunden minimale Wartezeiten entstehen. Diese Anwendung ist bereits mithilfe von Buzz oder auch über das Framework praktisch umsetzbar, da in beiden Systemen Abstände zu allen benachbarten Geräten bekannt sind. Das hier vorliegende Problem müsste aber im Gegensatz

zum Beispiel 5.2.1 auf eine Art Standortoptimierungsproblem erweitert werden, da die konzeptuelle Modellierung dieses Problems schwieriger ist. Es ist analog zur Berechnung von optimalen Positionen für Krankenhäuser oder Supermarktketten. Eine rein örtliche und äquidistante Verteilung ist für den praktischen Bezug nicht sinnvoll. Die geographische Verteilung der Bevölkerung ist in diesem Fall ein geeigneteres Maß. Schließlich hängt die Menge der erwarteten Bestellungen von der Einwohnerzahl im Servicefeld der Drohne ab.

5.2.3 Weltraumteleskop

Weltraumteleskope können an einem Observatorium, Forschungssatelliten oder sogar über den gesamten Globus verteilt sein. Um nun möglichst viele Objekte im Weltraum beobachten zu können, müssen Teleskope so ausgerichtet werden, dass sie einen möglichst weiten Sichtkegel erhalten. Die sich hier ergebende Fragestellung könnte eine Ausrichtung von Teleskopen zur Beobachtung an einem gewünschten Himmelsausschnitt mit maximaler Vergrößerung sein.

5.3 Mögliche Lösungsansätze

5.3.1 Ein intuitiver Ansatz

Bei Betrachtung der Bedingungen 5.1 und 5.2 ist die Ermittlung des gesuchten Abstandes durch einen äußeren Betrachter aufgrund der Anzahl aller Punkte und des vorliegenden Gebietes naheliegend. Ein solcher Betrachter ist für ein verteiltes System gemäß Abschnitt 1.2.1 jedoch unzulässig, sodass jede Instanz einen eigenen Abstand ermitteln muss. Weil aber mithilfe des verteilten Speichers eine gemeinsame Beobachtung propagiert werden kann, können alle Recheninstanzen zu einem einheitlichen Ergebnis kommen. Problematisch ist jedoch, dass es sich bei dem in 3.2.5 beschriebenen Speicher um einen unsynchronisierten Mechanismus ohne Konsens handelt, sodass zusätzliche Prozeduren erforderlich sind. Jedes Gerät könnte weiterhin seine aktuelle Position in einer Variable des verteilten Speichers propagieren, damit alle Instanzen solange am verteilten Prozess beteiligt sind, bis die in Abschnitt 5.1 definierten Bedingungen erfüllt sind. In diesem Ansatz wäre die vierte KDubiq-Eigenschaft, siehe 1.1.2, allerdings zu jedem Zeitpunkt verletzt. Zudem ist unklar, wie ein konkreter Algorithmus für diesen Lösungsansatz aussehen soll.

5.3.2 Clustering mit K-Means

Clustering ist eine bewährte Methode im Bereich des maschinellen Lernens, um Zusammenhänge in Daten zu identifizieren. Wie zu Beginn des Kapitels erwähnt, können Punkte als Repräsentant eines Clusters auch über eine Menge von Daten verteilt werden. Ein bekannter Algorithmus zum Clustering von Daten ist K-Means. Hierbei handelt es sich um einen Algorithmus der Komplexitätsklasse NP. K-Means initialisiert k Clusterschwerpunkte zu

Beginn zufällig im mathematischen Raum des Datensatzes und versucht anschließend, die Varianz zwischen allen Punkten eines Clusters und dessen Schwerpunkt zu minimieren. Der Minimierungsschritt besteht aus zwei Schritten: Zunächst wird jeder Datenpunkt einem Cluster zugeordnet, indem der Algorithmus über die gesamte Datenmenge iteriert und für jeden Datenwert den naheliegendsten Schwerpunkt ermittelt. Auf diese Weise wird jedem Datenpunkt ein Cluster zugewiesen. Anschließend erfolgt eine Korrektur aller Schwerpunkte, indem das arithmetische Mittel aller Clusterpunkte für jede Dimension berechnet wird. Dadurch ist die Varianz im alten Cluster minimiert, jedoch haben sich die Schwerpunkte verändert und damit auch die Zuweisungen von Datenpunkten zu ihren Clustern. Somit muss der Minimierungsschritt erneut durchgeführt werden. Dies geschieht solange, bis die Schwerpunkte konvergieren. Es gibt zahlreiche Varianten für K-Means. Zudem ist der Algorithmus parallelisierbar, indem mehrere Prozesse die Prozedur mit unterschiedlichen Initialclustern ausführen [34]. Im Anschluss muss jedoch jeder Prozess zentral mit allen bekannten Datenpunkten arbeiten. Der K-Means Algorithmus ist zwar approximativ, vergleiche Abschnitt 1.1.3, aber nicht innerhalb eines verteilten Systems im Anschluss an die Initialisierung parallelisierbar. Die Clusterschwerpunkte sind im Kontext der MicroPython-Bibliothek analog zu den einzelnen Geräte-Instanzen. Eine hiervon ausgehende Form der Parallelisierung ist nicht sinnvoll. Jeder Prozess muss über den gleichen Datensatz vollständig iterieren, sodass diese Aufgabe genauso gut einem einzigen Prozess überlassen werden kann. Eine Parallelisierung im Anschluss an die Initialisierung macht nur Sinn, wenn die Datenmenge gleichmäßig unter den Prozessen aufgeteilt werden kann und jeder Prozess alle Schwerpunkte kennt, sodass die Cluster verteilt berechnet werden können. Es ist aber unklar, wie genau die vergleichsweise einfache Problematik der äquidistanten Verteilung aus Abschnitt 5.1 über einen vollständig dezentralen Clusteringalgorithmus gelöst werden soll. Schließlich erfolgt beim Clustering keine Verteilung im mathematisch stetigen Raum, sondern über diskrete Datenpunkte.

5.3.3 Potentialfeld

Zur Umsetzung in einem verteilten System soll nun der im Folgenden skizzierte Algorithmus ausreichend sein: Im Rahmen der Gazebo-Simulation sind jedem Roboter die Positionen aller benachbarten Geräte bekannt. Ein Roboter kann auf dieser Basis eine Richtung berechnen, mit der die Entfernung zu den momentanen Positionen aller Nachbargeräte nicht verringert wird. Dazu stelle man sich alle Roboter als gleich geladene elektrische Teilchen vor, sodass ein Potentialfeld mit gegenseitiger Abstoßung entsteht. Der daraus resultierende Potentialvektor berechnet sich als Summe aller Differenzvektoren aus der eigenen Position und den Positionen benachbarter Roboter gewichtet mit dem Kehrwert des Abstands. Mithilfe des verteilten Speichers soll es ausreichen, das Gebiet als Rechteck durch zwei Variablen festzulegen. Wegen der losen Datenkopplung können die Grenzen des Gebiets zu

jeder Zeit beliebig geändert werden. Ein Roboter muss allerdings umkehren, wenn dieser das Gebiet verlässt. Des Weiteren soll der verteilte Vorgang terminieren. Eine Lösung hierfür kann willkürlich festgelegt werden. Algorithmus 5.1 berechnet den resultierenden Vektor im Wesentlichen mithilfe der `uNeighbours`-Datenstruktur. Für jeden Nachbarn wird dabei der Differenzvektor in der Variable `dirVector` berechnet und über den Abstand gewichtet auf die `resultVector`-Variable addiert. Der hier vorliegende Lösungsansatz ist dezentral, erfüllt alle KDubiq-Eigenschaften und wird daher für eine Demonstration in Gazebo verwendet.

```
newClosestNeighbour ← MAX_VALUE
i ← 0
while i < length of uNeighbours.neighbours do
    neighbourPos ← uNeighbours.neighbours(i)
    selfPos ← uCommunication.pos
    dirVector ← calculateDirectionVector(neighbourPos, selfPos)
    distance ← calculateDistance(neighbourPos, selfPos)
    if distance > 0.0 then
        if distance < newClosestNeighbour then
            newClosestNeighbour ← distance
        end if
        j ← 0
        while j < length of dirVector do
            dirVector(j) ← dirVector(j)/distance ∗ 50
            resultVector(j) ← resultVector(j) + dirVector(j)
            j ← j + 1
        end while
    end if
    i ← i + 1
end while
orientation ← math.fabs(uActuation.orientation)
if newClosestNeighbour ≥ closestNeighbourDistance then
    reward ← True
else
    reward ← False
end if
```

Algorithmus 5.1: resultierender Vektor

5.4 Besonderheiten in Gazebo

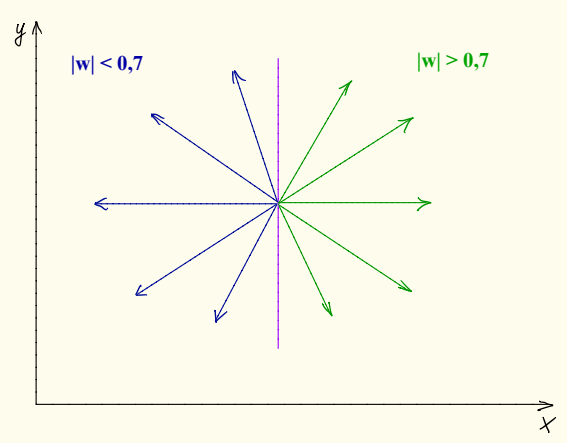

Abbildung 5.5: Vorwärtsbewegungen im Bezug zur Gazebo-Orientierung

Der Lösungsansatz aus Abschnitt 5.3.3 wird innerhalb des in Kapitel 3 beschriebenen Systems mit dem Algorithmus 5.1 umgesetzt. Es erfolgt damit eine Verteilung von Robotern im zweidimensionalen Raum. Wie in Abschnitt 2.1.2 beschrieben, publiziert Gazebo die Position und Orientierung eines Roboters in der ROS-Topic `/gazebo/model_states/` mit zentral einheitlichem Koordinatensystem. Über konkrete Steuerungsmethoden der Klasse `uActuation`, siehe Abschnitt 3.2.2 und 3.4, im Paket `uBabbleROSAbstractionLayer` ändern sich die Koordinaten eines Roboters in der Gazebo-Welt. Sei nun w die Orientierung eines Roboters in der Variable `pose[{Gazebo-Roboter-Index}].orientation.w`. Dann bedeutet die Vorwärtsbewegung eines Roboters über die Methode `forward()` der Klasse `uActuation` eine Bewegung in negative x-Richtung, wenn für die Orientierung gilt: $|w| < 0,7$. Eine Bewegung in positive x-Richtung erfolgt für $0,7 < |w| \leq 1,0$. Ein geeigneter ROS-Parameter zur Orientierung für die Bewegung in y-Richtung konnte im Rahmen dieser Arbeit nicht identifiziert werden. Das erschwert eine Bewegung der Roboter in die beabsichtigte Richtung. Des Weiteren werden alle Roboter als ROS-Nodes mit dem Befehl `roslaunch`, siehe Abschnitt 2.1.1, gestartet. Nachträglich werden in einer laufenden Gazebo-Simulation daher keine Roboter hinzugefügt, sodass die zweite KDubiq-Eigenschaft aus Abschnitt 1.1.2 nur bedingt erfüllt ist. Bei Betrachtung des Algorithmus 5.1 fällt auf, dass eine `reward`-Variable verwendet wird. Der Grund hierfür liegt in der nicht eindeutigen Orientierung des Roboters bezüglich seiner Bewegung in y-Richtung. Es kann sein, dass ein Roboter, zwar den korrekten Richtungsvektor berechnet, aber in die falsche Richtung fährt. Dann hat sich die Lösung mit der letzten Bewegung verschlechtert und die Bewegungsvorgänge müssen entsprechend angepasst werden.

Kapitel 6

Experiment in Gazebo

Der verteilte Algorithmus aus Abschnitt 5.3.3 und das Framework werden mithilfe des MicroPython-Interpreters in die Unix- und ARM-Systeme aus Kapitel 4 implementiert. Die Funktionalität des Algorithmus 5.1 ist im Modul `GradientReinforcementLearning` umgesetzt. Die hierin enthaltene Implementierung ist wegen der in Abschnitt 5.4 beschriebenen Problematik etwas umfangreicher, da für jeden berechneten Vektor der passende Steuerungsbefehl geschätzt werden muss. Die Unix- und ARM-Systeme sind durch zwei Simulationen voneinander getrennt. In beiden Fällen wird eine leere Gazebo-Welt mit mehreren Robotern geladen, in der sich alle Geräte in einer ungekennzeichneten viereckigen Fläche verteilen sollen.

6.1 Versuch mit Unix-Systemen

Das catkin-package `mr_MP_ROS` startet eine Simulation mit acht Robotern und führt alle MicroPython-Implementierungen im Unix-Port aus. Jeder Roboter ist zu Beginn willkürlich innerhalb der Gazebo-Welt positioniert, siehe Bild 6.1. Nach kurzer Zeit bilden sich, wie in Bild 6.2 erkennbar, zwei Gruppen, die in entgesetzte Richtungen voneinander wegfahren. Es entstehen jedoch schnell einseitige Polarisierungen und einige Roboter weichen in zufällige Richtungen aus, sodass sich die Verteilung in Bild 6.3 nach fast drei Minuten ergibt. Zum Abschluss ensteht eine Verteilung gemäß Bild 6.4. Hier liegt allerdings keine zufriedenstellende Lösung, entsprechend der mathematischen Bedingungen aus Abschnitt 5.1, vor. Weil das Gebiet G zudem ein Quadrat ist, kann der Raum durch die Roboter nicht vollständig ausgefüllt sein. Die in Bild 6.4 sichtbare Lösung ist daher noch weit von der geforderten äquidistanten Verteilung entfernt.

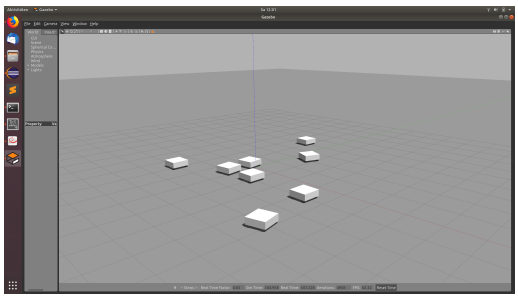

Abbildung 6.1: Startzustand der Simulation mit Unix-Systemen

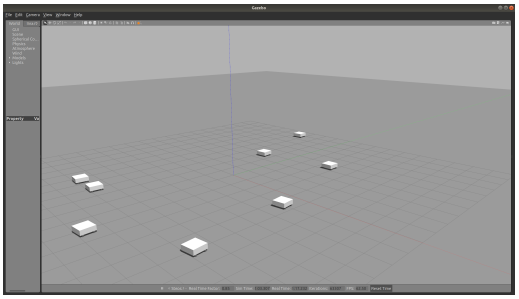

Abbildung 6.2: Perspektivische Ansicht nach 1:03

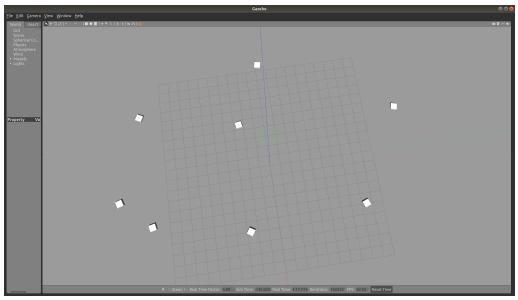

Abbildung 6.3: Verteilung nach 2:45

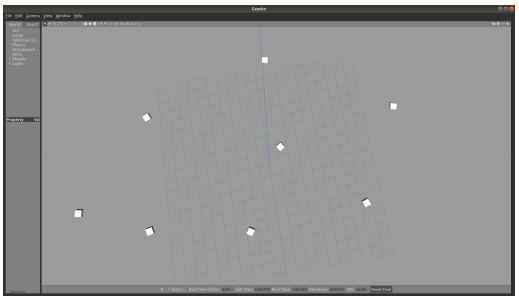

Abbildung 6.4: Verteilung nach 3:43

6.2 Versuch mit ARM-Systemen

Mit dem catkin-package `mr_mp_browser` wird eine Simulation in Gazebo mit drei Robotern bereitgestellt. Für jeden Roboter muss das MicroPython-Skript extern über den Browser gestartet werden. Dazu wird ein Servlet im modifizierten Github-Projekt " MicroPython on Unicorn" gestartet. Für jedes Skript muss ein eigenes Browser-Fenster geöffnet werden. Nachdem die MicroPython-Bibliotheken über die `AppendScript`-Buttons in drei `Unicorn-CPUs` eingebunden wurden, können die Programme gestartet werden. Im Anhang ist hierzu eine genauere Beschreibung unter Punkt A.2.2 vorhanden. Bild 6.5 zeigt die Simulation am rechten Bildrand und drei weitere Browserfenster mit den Oberflächen der verwendeten ARM-Emulation. Dem MicroPython-Skript steht in der Emulation wesentlich weniger Leistung als im Unix-Port zur Verfügung. Die Mikrocontroller takten mit einer Frequenz von ca. 5 Megahertz sehr langsam. Das MicroPython-Skript lässt sich dennoch ausführen. Im Bild ist erkennbar, dass den Emulationen nur 256 Kilobyte RAM zur Verfügung stehen. Weil nun lediglich drei Roboter simuliert werden, kann mit Bild 6.6 nur schwer eine Aussage darüber getroffen werden, wie „gleichmäßig“ die Roboter abschließend im Raum verteilt sind.

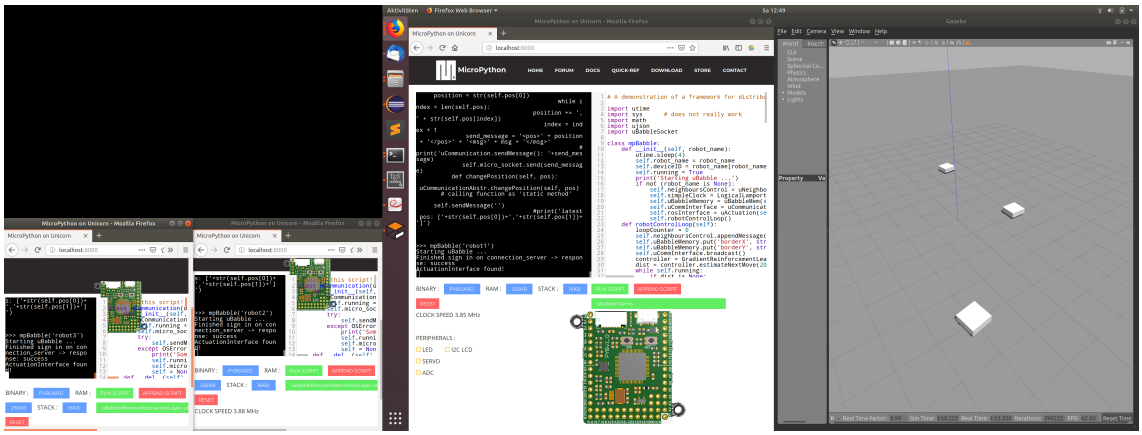

Abbildung 6.5: ARM-Emulationen mit Kopplung an Gazebo

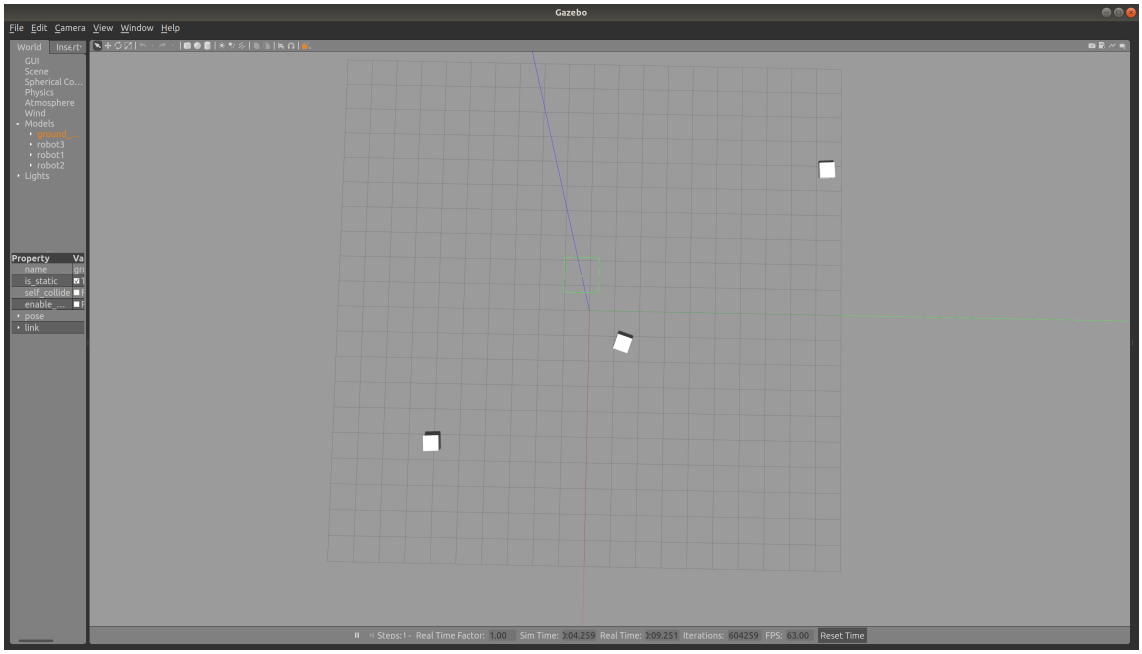

Abbildung 6.6: Verteilung nach einiger Zeit

6.3 Interpretation der Versuche

Die Verteilung der Roboter ist in beiden Versuchen nicht zufriedenstellend. Der Grund für das mangelhafte Ergebnis ist die ungenaue Abschätzung eines passenden Steuerungsbefehls und die frühzeitige Terminierung der MicroPython-Skripte, da jedes Skript beim Überschreiten der Gebietsgrenzen sofort terminiert. Eine exakte Lösung der Problemstellung aus Kapitel 5 soll im Rahmen dieser Arbeit aber eine sekundäre Rolle spielen. Die hier durchgeführten Versuche demonstrieren in erster Linie, dass die Implementierung verteilter heterogener Systeme mit MicroPython-Quellcode möglich ist, ohne aufwendige hardwarebedingte Anpassungen im Quellcode vorzunehmen. Auffällig ist, dass die Simulationen sehr langsam ablaufen. Das liegt nicht an der verwendeten Hardware oder den Emulatoren. Die Programmschleifen sind bewusst durch Sleep-Funktionen stark verlangsamt, da die Position und Orientierung eines Roboters nur zwei mal pro Sekunde mit dem Framework synchronisiert wird. Zudem wird so eine sehr hektische Bewegung der Roboter vermieden.

Kapitel 7

Diskussion und Evaluation

Das `uBabble`-Framework öffnet viele Möglichkeiten, um Problemstellungen in vollständig verteilten Systemen mit geringem Aufwand umzusetzen. Der Algorithmus aus Kapitel 5 umfasst mithilfe der Abstraktionen des Frameworks lediglich 140 Zeilen Code, von denen über 50 Zeilen auf die Steuerungsproblematik in Abschnitt 5.4 zurückzuführen sind. Darüber hinaus ist der Quellcode in zwei verschiedenen Betriebssystemen mit unterschiedlicher Rechnerarchitektur ausführbar. Der allgemeine Funktionsumfang zum Datenaustausch mit ROS ist jedoch sehr gering und beschränkt sich auf Steuerungsfunktionen. Die Programmbibliothek ist als objektorientierter Entwurf aber modular aufgebaut und kann daher gegenüber einem domain-spezifischen Entwurf mit geringerem Aufwand erweitert werden.

7.1 Schwachpunkte im Framework

Frameworks streben jedoch häufig eine Verwendung in bereits sehr spezifischen Anwendungsfeldern an. Komplexität und innere Vernetzungen erschweren die Nutzung und damit insbesondere Möglichkeiten zur Erweiterung oder Portierung.

7.1.1 Kopplung von Klassen

Objektorientierte Metriken [7] sind ein geeignetes Mittel zur Bewertung von Softwarearchitekturen. Es fällt bei Betrachtung von Bild 3.2 auf, dass `uActuation` und `uCommunication` in der `uBabbleROSAbstractionLayer` miteinander gekoppelt sind. Die Abstraktionsschicht dient allerdings zur möglichst einfachen Anbindung an weitere Systeme. Das Umschreiben der Klassen vereinfacht sich, wenn gemäß der objektorientierten CBO-Metrik nur wenige Abhängigkeiten vorhanden sind. Vor diesem Hintergrund ist eine Auslagerung der Kopplung beider Klassen in die darüberliegende Schicht `uBabbleAbstract` besser geeignet. Weiterhin ist die Kopplung der logischen Lamport-Uhr an den verteilten Speicher ein Problem, siehe Bild 3.1. Es macht keinen Sinn, die Klasse `LogicalLamportClock` einzig an die Klasse `uBabbleMem` anzuhängen, denn so kann die Uhr nur über Nachrichten des verteilten

Speichers synchronisiert werden. Besser wäre es, die Uhr, ebenso wie `uBabbleMem` und `uNeighbours`, als Beobachter an `uCommunication` zu koppeln. Auf diese Weise kann die Uhr mit jeder eingehenden Nachricht synchronisiert werden. Damit erhöht sich zwar die Klassenkopplung im Modul `uCommunication`, die Uhr kann dafür aber mit jeder eingehenden Nachricht synchronisiert werden. Alternativ kann die Uhr auch als Attribut in beiden Klassen des `uBabbleModel`-Packages referenziert sein.

7.1.2 Die verteilte Uhr

Die verteilte Uhr ist lediglich als logische Uhr-Funktion implementiert. Eine Erweiterung der Funktion zur physikalischen Uhr würde die Implementierung verteilter Synchronisationsmechanismen in einem nicht-ausfallsicheren System ermöglichen [17]. Des Weiteren erfolgt das Ticken der logischen Uhren im Rahmen des Experiments lediglich nach jedem Durchlauf der Programmschleife. So wird ein gesamter Schleifendurchlauf als Event betrachtet. Die Implementierung der verteilten Uhr ist damit strenggenommen falsch und führt zu Konflikten, wenn ein Gerät innerhalb eines Durchlaufs mehrere Nachrichten absetzen will. Die Uhr muss also mit jedem Methodenaufruf oder zwischen jeder Zeile des Quellcodes erhöht werden. Weiterhin werden die Wertebereiche aller Uhren nur disjunkt generiert, wenn alle Geräte-IDs gleich viele Zahlenstellen aufweisen. Eine Kombination aus `80` und `8` ist unzulässig. Stattdessen benötigt ID `8` eine vorangehende Null, sodass die Kombination `08` und `80` zulässig ist.

7.1.3 Authentizität

Alle MicroPython-Instanzen versenden regelmäßig ihre Geräte-ID, um im verteilten Prozess identifiziert zu werden, siehe Abschnitt 3.2. Soweit vertrauen alle Geräte den Nachrichten ihrer Nachbarn und nehmen an dass jeder Teilnehmer im verteilten System glaubwürdig bzw. authentisch ist. Ein bösartiger Prozess kann jedoch das gesamte System stören, indem ausgewiesene Identitäten bewusst manipuliert sind. In einem technisch sicheren System darf daher keinem Kommunikationspartner grundlos vertraut werden. Authentizität wird in verteilten Systemen über digitale Signaturen und vertrauenswürdige Certificate-Authorities realisiert. Bei einer CA handelt es sich allerdings um eine zentrale Instanz. Dennoch können digitale Signaturen verhindern, dass ein Prozess einem anderen Prozess in einem laufenden verteilten System die Identität stiehlt. Soweit sind universell eindeutige und authentische Geräte-IDs im Framework allerdings nicht zwingend notwendig, da fehlerhafte IDs keine Abstürze verursachen können.

7.2 Positive Aspekte des Frameworks

Ein großer Vorteil der Bibliothek ist die geringe Größe und Komplexität sowie die verwendete Programmiersprache. Es können an dieser Stelle alle in Abschnitt 1.1.1 beschriebenen Vorteile der Sprache MicroPython bestätigt werden. In Kapitel 4 konnten alle im ARM-Port fehlenden Module vollständig in C implementiert werden. Die in Kapitel 3.2 beschriebenen abstrakten Klassen können zudem eine flexible Verwendung des Frameworks in neuen Kontexten ermöglichen.

7.2.1 Systemvoraussetzungen

Mit der Durchführung des Versuchs aus Abschnitt 6.2 wurde eine Ausführung des Frameworks in Mikrocontroller-Hardware demonstriert. Der Quellcode ist daher für eine Ausführung in sehr ressourcenschwachen Systemen geeignet. Lediglich 32 Kilobyte Arbeitsspeicher und 4 Kilobyte Stapelspeicher sind ausreichend, um den Versuch in der Mikrocontroller-Emulation aus Abschnitt 4.3.2 durchzuführen.

7.2.2 Technologieunabhängiges Fluten

Alle Nachrichten zwischen MicroPython-Instanzen werden als lokaler Broadcast ausgetauscht. Die globale Propagierung von Einträgen des verteilten Speichers erfolgt daher durch Fluten des Systems mit Nachrichten. Diese Lösung bietet sich auch dann an, wenn Framework-Instanzen über ein IP-Netz gekoppelt sind, da am verteilten Prozess beteiligte Geräte nicht miteinander bekannt sein müssen. Dem entsprechend lässt sich aber auch keine verteilte Synchronisation, wie in Abschnitt 2.3.2 dargestellt, realisieren. Fluten hat jedoch den Nachteil, dass im gesamten System sehr viele Nachrichten entstehen und das Netz somit sehr stark belastet wird. `uBabble` flutet allerdings nur dann, wenn Einträge im verteilten Speicher beschrieben oder gelesen werden, sodass keine periodische Aktualisierung des Speichers erfolgt. Auf diese Weise konzentrieren sich entsprechende Nachrichten nur auf aktuell relevante Variablen [32].

7.3 Programmierung heterogener Systeme mit MicroPython

Objektorientierte Systementwürfe können mit MicroPython in heterogener Hardware umgesetzt werden, ohne im Vorfeld detaillierte hardwarespezifische Besonderheiten berücksichtigen zu müssen. Wie in Abschnitt 4.5 skizziert, besteht die Möglichkeit technische Komplikationen durch C-Module im Nachhinein anzupassen. Die zunächst für den Unix-Port entwickelte Bibliothek konnte daher anschließend in eine ARM-Umgebung eingebunden werden. Zudem ermöglichen C-Module die Implementierung von echtzeitkritischem Code. Für eine Verwendung von MicroPython-Quellcode in heterogenen Systemen müssen Grundlagen verfügbarer Hardwarekomponenten allerdings bereits in der Entwurfsphase

berücksichtigt werden. Schließlich kann grundlegende Hardwarefunktionalität nicht nachträglich durch Software ersetzt werden. In Kapitel 4 waren Sockets nur aufgrund der verknüpften JavaScript-Umgebung verfügbar. Ein echtes Pyboard muss hingegen um eine passende Hardware-Schnittstelle erweitert werden.

Kapitel 8

Zusammenfassung und Ausblick

Mit dieser Arbeit wurde die Implementierung eines MicroPython-basierten Frameworks in zwei verschiedene Systeme demonstriert. Die Sprache MicroPython ist dafür auf unterschiedlichen Ports für verschiedene Hardware verfügbar und flexibel erweiterbar. Über 500 Zeilen Quellcode konnten in einer sehr ressourcenschwachen Umgebung ausgeführt werden. Dabei war das Robot-Operating-System als externes System an jede Geräte-Instanz gekoppelt. Durch Kommunikation und gegenseitig bedingte Interaktionen mussten alle Systeme unter weichen Echtzeitbedingungen an der verteilten Lösung eines globalen Problems arbeiten. Dazu wurden zwei verteilte Systeme unterschiedlicher Leistungsklassen in Gazebo simuliert.

8.1 Eine Gazebo-Simulation mit heterogenen Geräten

Mit den in Kapitel 4 vorgestellten Systemen kann der gleiche Programmcode in unterschiedlicher Hardware ausgeführt werden. Das Experiment in Kapitel 6 simuliert homogene Hardware in zwei voneinander getrennten Versuchen. Es bietet sich daher an, eine Simulation mit beiden Systemen zu entwerfen, in der die Leistungsunterschiede der verschiedenen Befehlssatzarchitekturen technisch relevant werden. Dafür müssen allerdings die verteilten Uhren häufiger inkrementiert werden und mit allen ein- und ausgehenden Nachrichten verknüpft sein. Darüber hinaus muss die MicroPython-Implementierung für den ARM-Port in Browser-Prozessen automatisiert werden, da soweit mehrere Codesegmente manuell geladen werden müssen. Das Zusammenspiel der Komponenten kann anschließend anhand von Benchmarks beurteilt werden. Zudem kann eine detaillierte Analyse weiterer verteilter Prozesse mit neuen Problemstellungen erfolgen. Es empfiehlt sich allerdings, das in Abschnitt 4.5 vorgestellte Modul `uBabbleSocket` zu optimieren, da die Laufzeit der Methoden `send` und `recv` stark verbessert werden kann.

8.2 Verteiltes maschinelles Lernen

Die Problemstellung aus Kapitel 5 ist für reale Anwendungskontexte im Bereich des maschinellen Lernens nicht besonders relevant. Interessanter ist die Frage nach verteiltem Clustering einer Menge von zweidimensionalen Datenpunkten durch einen Algorithmus, der mithilfe der MicroPython-Bibliothek abstrahiert wird.

8.3 Eine Blockchain

Der verteilte Speicher `uBabbleMem` ist unsynchronisiert und inkonsistent. Daher können alle Prozesse die Variablen ständig überschreiben. Für verteilte Speicher wie bspw. einer Blockchain gibt es unterschiedliche Techniken, um ein beliebiges Überschreiben oder Erweitern des Speichers zu verhindern. Bisher konnte sich die Technik des Proof-of-Work bewähren: Ein Prozess, der in den Speicher schreiben will, muss zunächst ein „kryptographisches Puzzle“[20] lösen, um zum Schreibvorgang autorisiert zu sein. In einer Blockchain ist dieses Puzzle durch die Lösung eines vorausgegangenen Problems eindeutig bestimmt. Technisch wird das kryptographische Puzzle in der Regel durch Reverse-Hash Rätsel realisiert. Hashfunktionen werden in IT-Systemen genutzt, um einen für eine große Datenmenge repräsentativen Wert zu definieren. Für einen Hashwert der Größe k realisiert eine Hashfunktion die Abbildung $\{0,1\}^* \to \{0,1\}^k$. Stark kollisionsresistente Hash-Funktionen müssen ein mathematisches Kriterium erfüllen, nachdem es schwierig ist, zwei Datenpaare a und b mit gleichem Hashwert zu finden. Viele Proof-of-Work Techniken nutzen genau diese Komplexität, um Prozesse für Schreibvorgänge zu autorisieren. Der verteilte Speicher des Frameworks könnte um ein Konzept erweitert werden, das ein ständiges Beschreiben der Variablen verhindert. Hierzu exisitiert eine Arbeit über eine energieeffiziente Variante des Proof-of-Work, die in eingebetteten Systemen nutzbar ist [20].

Anhang A

Weitere Informationen

Die Gazebo-Simulationen aus Kapitel 6 sind auf der CD in den Ordnern `/mr_MP_ROS/` und `/Browser/mr_mp_browser/` enthalten. Im Pfad `/Browser/micropython-unicorn` befindet sich die ARM-Emulation zur Ausführung im Browser.

A.1 Installationen

Zur Ausführung der Simulationen sind eine aktuelle Version von MicroPython, eine vollständige Installation des Robot-Operating-Systems, ein aktueller Browser sowie Python in den Versionen 2 und 3 erforderlich. Die hier vorliegende Beschreibung zielt auf eine Installation aller Komponenten in Ubuntu LTS 18.04 als Hostsystem.

A.1.1 MicroPython installieren

Zur Installation von Micropython kann der Anleitung im Github-Wiki gefolgt werden [12]. Da innerhalb von ROS Micropython-Skripte ausgeführt werden, muss die Datei `../ports-/unix/micropython` durch die PATH-Umgebungsvariable ausführbar sein. Dazu wird das Repository am besten nach /opt kopiert und im Terminal mit `sudo ln -s /opt/micro-python/ports/unix/micropython /usr/bin/micropython` sowie `sudo chmod a+x /usr-/bin/micropython` ausführbar gemacht. Der Micropython-Interpreter lässt sich jetzt mit dem Befehl `micropython` ausführen.

A.1.2 Robot-Operating-System (ROS)

Die Installation von ROS erfolgt über die ROS-Homepage: `www.ros.org`. Die für Ubuntu 18.04 unterstützte Version ist „ROS Melodic Morenia“. Nach abgeschlossener Installation wird ein Catkin-Workspace bspw. im Home-Verzeichnis des Nutzers mit folgenden Befehlen erstellt: `mkdir -p ~/catkin_ws/src` , `cd ~/catkin_ws/` , `catkin_make` . Um Elemente innerhalb der Pakete des Workspace verwenden und ausführen zu können, muss jedes Terminal das zum Workspace zugehörige setup.bash - file kennen. Dazu wird `source`

`~/catkin_ws/devel/setup.bash` ausgeführt. Die Pakete „mr_MP_ROS“ und „mr_mp-_browser“ werden in den src-Ordner des Workspace kopiert und `catkin_make` neu durchgeführt. Die Simulationen können jetzt mit `roslaunch {Paketname} main.launch` ausgeführt werden. Wichtig ist, dass jedes neue Terminal das "../devel/setup.bash" -file sourcen muss, weil ROS sonst nicht weiß, in welchem Workspace gearbeitet wird. Es kann sein, dass die Python-Skripte im Ordner `nodes` nicht ausgeführt werden. Dann müssen diese mit `chmod +x {Skript}.py` ausführbar gemacht werden.

A.1.3 ARM-Emulation

Die für den Browser verfügbare ARM-Emulation ist eine Erweiterung des Github-Projekts " MicroPython on Unicorn"[33]. Die Emulation kann im Pfad `/Browser/micropython-unicorn/www-emu/build` gestartet werden. Eine Option zur Ausführung ist die Index-Datei. Die flexiblere Alternative sind jedoch Servlets wie beispielsweise der Http-Server in Python, welcher über den Befehl `python3 -m http.server {Portnummer}` gestartet werden kann. Die JavaScript-Oberflächen des Pyboards dürfen allerdings erst nach dem Starten der entsprechenden Gazebo-Simulation aufgerufen werden.

A.2 Ausführen der Simulationen

A.2.1 Simulation für den Unix-Port

Das Starten der Gazebo-Simulation mit Unix-basierten MicroPython Skripten ist sehr einfach:

1. Linux-Terminal starten
2. Eingabe: `source ~/catkin_ws/devel/setup.bash`
3. Eingabe: `roslaunch mr_MP_ROS main.launch`

A.2.2 Simulation für den ARM-Port

Die Unicorn- bzw. ARM-Emulation muss in Browser-Prozessen ausgeführt werden und kann daher nicht mit ROS gestartet werden. Zuerst wird aber ebenfalls eine Gazebo Simulation, genauso wie oben, gestartet:

1. Linux-Terminal starten
2. Eingabe: `source ~/catkin_ws/devel/setup.bash`
3. Eingabe: `roslaunch mr_mp_browser main.launch`

Anschließend wird ein Servlet mit dem Befehl `python3 -m http.server 8000` im Pfad `/Browser/micropython-unicorn/www-emu/build` gestartet. Das Servlet kann auch vor der Simulation aufgerufen werden. Im nächsten Schritt wird die Webseite geöffnet. Auf dem gleichen Gerät kann dies über die Adresse `localhost:8000` geschehen. Es müssen drei Webseiten in jeweils eigenen Browserfenstern aufgerufen werden. Über den Button `APPEND SCRIPT` werden die Skripte „uBabble Demo“, „uBabbleModel“, „uBabbleAbstract“, „uBabbleBrowserAbstractionLayer-uActuation“ und „uBabbleBrowserAbstractionLayer-uCommunication“ in das Terminal eingebunden. Abschließend können die MicroPython-Skripte durch Eingabe des Befehls `mpBabble('{Robotername}')` im Terminal gestartet werden. Die in Gazebo verfügbaren Roboter heißen "robot1", "robot2" und "robot3".

A.3 Ein C-Modul für MicroPython

Das Nutzen von MicroPython in neuer Hardware kann nachträgliche Anpassungen erfordern, die nur in C oder Assembler verfügbar sind [24]. Jeder MicroPyton-Port hält dafür eine Konfiguration bereit, die eine Erweiterung des Ports um C-Module ermöglicht, sodass C-Funktionen auch in MicroPython aufgerufen werden können. Im Rahmen dieser Arbeit wurde ein benutzerspezifischer MicroPython-Port um das Modul `uBabbleSocket` erweitert, welches Inter-Prozess-Kommunikation auch im Kontext einer ARM-Emulation in Browser-Prozessen ermöglicht.

A.3.1 Makefile

Im Makefile befindet sich eine Liste aller C-Dateien, die für diesen Port kompiliert werden. Soll ein Modul zum Port hinzugefügt werden, muss der C-Quellcode der Datei im Makefile referenziert sein. Das Makefile der MicroPython-Module für die ARM- bzw. Unicorn-Emulation befindet sich im Pfad `/Browser/micropython-unicorn/unicorn`. Hier sind zudem alle für den Port spezifischen C-Dateien definiert. Im Makefile des ARM- bzw. Unicorn-Ports wird das Modul `uBabbleSocket` mit der Datei `moduBabbleSocket.c` implementiert. Die Definition der dazugehörigen Socket-Klasse befindet sich in einer seperaten Datei namens `uBabbleSocket_socket.c`.

A.3.2 mpconfigport.h

Alle Module eines MicroPython-Ports sind im `mpconfigport`-headerfile definiert. Der Unicorn-Port erweitert das headerfile um eine pyboard-spezifische Definition in der Datei `mpconfigport_pyboard.h`. Dort befindet sich eine Referenz auf das Modul `uBabbleSocket` vom Typ `mp_obj_module_t`, welches der C-Typ aller MicroPython-Module ist. Der Eintrag `{ MP_ROM_QSTR(MP_QSTR_uBabbleSocket), MP_ROM_PTR(&uBabbleSocket) }` im Makro `MICROPY_PORT_BUILTIN_MODULES` zeigt auf eine Adresse, die ein C-struct vom Typ `mp_obj-`

_module_t in der Variable uBabbleSocket definiert. Hierbei handelt es sich um den Kern der Definition des Moduls uBabbleSocket. Der Name des Moduls ist durch den Schlüssel MP_QSTR_uBabbleSocket über den Präfix MP_QSTR_ als „uBabbleSocket“ festgelegt.

A.3.3 Die Moduldatei

Ein MicroPython-Modul besteht grundsätzlich aus dem oben erwähnten C-struct und einer Modultabelle, die statisch globale Variablen des Moduls festlegt. Das C-struct ist, wie oben bereits erwähnt, vom Typ mp_obj_module_t und definiert statische Eigenschaften. Dafür sind zwei Variablen notwendig: .base und .globals. Die Variable .globals ist ein Pointer, der über ein Wörterbuch auf die Modultabelle verweist. Die Tabelle ist erneut ein Wörterbuch vom Typ mp_map_elem_t, das die Adressen weiterer globaler Variablen enthält. Das Modul uBabbleSocket enthält hier eine Referenz auf die Klasse socket mit dem Eintrag { MP_OBJ_NEW_QSTR(MP_QSTR_socket), (mp_obj_t)&uBabbleSocket_socket_type }. Der Wörterbucheintrag bildet auf eine Adresse der Definition uBabbleSocket_socket_type ab. Diese ist in einem C-struct der Datei uBabbleSocket_socket.c definiert und über das in beide C-Dateien importierte Headerfile moduBabbleSocket.h deklariert. Damit erfolgt die Referenz des Moduls uBabbleSocket auf seine innere Klasse socket in gleicher Weise, wie auch der Port auf das Modul verweist, wobei die Klasse über ein C-struct vom Typ mp_obj_type_t definiert ist.

A.3.4 Innere Klasse eines Moduls

Die Definition der Klasse socket besteht neben dem C-struct ebenfalls aus einer Tabelle vom Typ mp_rom_map_elem_t. Hier sind Methoden für Objekte der Klasse festgelegt. Für socket-Objekte sind in der Datei uBabbleSocket_socket.c die zwei Methoden send und recv definiert. Ein weiteres C-struct „uBabbleSocket_socket_t“ legt das Attribut port für Objekte der Klasse fest. Der Konstruktor wird über die Funktion uBabbleSocket_socket_make_new implementiert und im uBabbleSocket_socket_type-struct über die Eigenschaft .make_new referenziert. Beide Methoden der Klasse socket werden mit der Klassentabelle „uBabbleSocket_socket_locals_dict_table“ jeweils über das Makro MP_DEFINE_CONST_FUN_OBJ_2 auf die entsprechende C-Funktion abgebildet. Das Makro gibt dabei die Zahl der Parameter an.

A.4 Hinweise zur Unicorn-Emulation

Der C-Quellcode für den Datenaustausch über WebSockets im micropython-unicorn-Projekt ist im Pfad /Browser/micropython-unicorn/unicorn in den Dateien uBabbleSocket_socket.c und unicorn_mcu.h enthalten. Die JavaScript-Gegenseite ist im Pfad /Browser/micropython-unicorn/www-emu in den Dateien mp_unicorn.js und ubabble-

`client.js` definiert. Die Server aus Abschnitt 3.4.2 und 3.5 sind im catkin-package `mr_mp_browser` als eventbasierte WebSocket-Server [8] implementiert und werden jeweils in einem einzigen Prozess ohne Thread ausgeführt. Der `babble_server` führt allerdings viele Berechnungen durch und ist daher in der WebSocket-Variante sehr langsam. Die TCP-Server im Paket `mr_MP_ROS` starten für jeden verbundenen Client einen neuen Thread und sind damit deutlich schneller.

Abbildungsverzeichnis

1.1 Golden Hammer [3] 3

2.1 Raum-Zeit Diagramm [17] 14

3.1 uBabble-Strukturmodell 20
3.2 uBabble-ROS Schnittstelle 23

4.1 System der Unicorn-Emulation 31

5.1 Unhomogene Verteilung 34
5.2 Gleicher Abstand in x- und y- Richtung 34
5.3 Äquidistante Verteilung 34
5.4 Frequenzband - Aufteilung 36
5.5 Vorwärtsbewegungen im Bezug zur Gazebo-Orientierung 40

6.1 Startzustand der Simulation mit Unix-Systemen 42
6.2 Perspektivische Ansicht nach 1:03 42
6.3 Verteilung nach 2:45 42
6.4 Verteilung nach 3:43 42
6.5 ARM-Emulationen mit Kopplung an Gazebo 43
6.6 Verteilung nach einiger Zeit 43

Algorithmenverzeichnis

5.1 resultierender Vektor 39

Literaturverzeichnis

[1] Aachener Werkzeugmaschinen-Kolloquium: *Internet of Production für agile Unternehmen*. Web: `http://www.awk-aachen.de/`, 2017. abgerufen am 10.12.2018.

[2] Alpern, B. und F. B. Schneider: *Recognizing safety and liveness*. Springer, 1987.

[3] anonym: *Golden Hammer*. Web: `https://xkcd.com/801/`. abgerufen am 03.12.2018.

[4] anonym: *The Raft Consensus Algorithm*. Web: `https://raft.github.io/`. abgerufen am 10.12.18.

[5] Bryant, R. E. und D. R. Hallaron: *Computer Systems - A Programmer's Perspective*, Kapitel Optimizing Program Performance. Pearson, Massachusetts, 2. Auflage, 2011.

[6] Chen, C., G. Novick und K. Shimano: *risc vs. cisc*. Web: `https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/`. abgerufen am 19.12.2018.

[7] Chidamber, S. R. und C. F. Kemerer: *TOWARDS A METRICS SUITE FOR OBJECT ORIENTED DESIGN*. ACM Digital Library, 1991.

[8] Dave, P.: *A Simple Websocket Server written in Python*. Web: `https://github.com/dpallot/simple-websocket-server`. abgerufen am 03.01.2019.

[9] Düdder, B.: *Softwarekonstruktion - Einleitung*. Vorlesung: TU Dortmund, 2016. Vorlesungsfolie auf Seite 27.

[10] Elektronik Kompendium: *CISC und RISC*. Web: `https://www.elektronik-kompendium.de/sites/com/0412281.htm`. abgerufen am 05.01.2019.

[11] Gelernter, D.: *Generative Communication in Linda*. ACM Transactions on Programming Languages and Systems, 7(1), Januar 1985.

[12] George, D. P.: *The MicroPython project*. Web: `https://github.com/micropython/micropython`. abgerufen am 03.01.2019.

[13] George, D. P., P. Sokolovsky und contributors: *MicroPython documentation.* Web: `http://docs.micropython.org/en/latest/`, 2018. abgerufen am 15.12.2018.

[14] George Robotics Limited: *Micropython.* Web: `http://www.micropython.org`, 2018. abgerufen am 03.12.2018.

[15] Helmentstine, A. M.: *Heterogeneous Definition.* Web: `https://www.thoughtco.com/heterogeneous-definition-and-example-606355`. abgerufen am 03.12.2018.

[16] Kowalk, W.: *Fluten.* Web: `http://einstein.informatik.uni-oldenburg.de/rechnernetze/fluten.htm`. abgerufen am 17.12.2018.

[17] Lamport, L.: *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 21:558 – 565, Juli 1978.

[18] Lamport, L.: *Paxos made simple.* ACM Digital Library, 2001.

[19] Liebig, T., M. Stolpe und K. Morik: *Distributed Traffic Flow Prediction with Label Proportions: From in-Network towards High Performance Computation with MPI.* CEUR-WS, 2015.

[20] Lundbaek, L., D. J. Beutel, M. Huth, S. Jackson, L. Kirk und S. Schwerin: *Xain - Practical Proof of Kernel Work & Distributed Adaptiveness.* Yellow Paper: Version 1.2.

[21] Lévénez, E.: *Computer Languages History.* Web: `https://www.levenez.com/lang/`. abgerufen am 30.11.2018.

[22] Marwedel, P.: *Rechnerstrukturen, Teil 2*, Seiten 35 – 36. TU Dortmund, 2013.

[23] May, M., B. Berendt, A. Cornuejols, J. Gama, F. Giannotti, A. Hotho, D. Malerba, E. Menesalvas, K. Morik, R. Pedersen, L. Saitta, S. Yücel, A. Schuster und K. Vanhoof: *Research Challenges in Ubiquitous Knowledge Discovery.* researchgate, 2009.

[24] Naumann, Stefan: *Adding a module to Micropython.* Web: `https://www.stefannaumann.de/de/2017/07/adding-a-module-to-micropython-2/`. abgerufen am 04.12.2018.

[25] Nguyen, A. Q. und H. V. Dang: *Unicorn - The ultimate CPU emulator.* Web: `http://www.unicorn-engine.org/`. abgerufen am 05.12.2018.

[26] Online Computer Magazin: *Die wichtigsten Programmiersprachen unserer Zeit.* Web: `http://www.commag.org/die-wichtigsten-programmiersprachen-unserer-zeit/`. abgerufen am 30.11.2018.

[27] Open Source Robotics Foundation: *Gazebo - Robot simulation made easy.* Web: `http://gazebosim.org/`. abgerufen am 05.12.2018.

[28] Open Source Robotics Foundation: *ROS.* Web: `http://www.ros.org/`. abgerufen am 04.12.2018.

[29] Open Source Robotics Foundation: *ROS Box Turtle.* Web: `http://wiki.ros.org/boxturtle`. abgerufen am 04.12.2018.

[30] Open Source Robotics Foundation: *ROS Tutorials.* Web: `http://wiki.ros.org/`. abgerufen am 05.12.2018.

[31] Oracle: *Welche Systemvoraussetzungen gelten für Java?* Web: `https://www.java.com/de/download/help/sysreq.xml`. abgerufen am 30.11.2018.

[32] Pinciroli, C., A. Lee-Brown und G. Beltrame: *Buzz: An extensible Programming language for Self-Organizing Heterogenous Robot Swarms.* arXiv, 2015.

[33] Pseudonym 'flowergrass': *MicroPython on Unicorn.* Web: `https://github.com/micropython/micropython-unicorn`. abgerufen am 04.12.2018.

[34] scikit-learn developers: *Clustering.* Web: `https://scikit-learn.org/stable/modules/clustering.html#k-means`. abgerufen am 28.12.2018.

[35] Spinellis, D.: *Unix Architecture Evolution from the 1970 PDP-7 to the 2018 FreeBSD Important Milestones and Lessons Learned.* Web: `https://www.youtube.com/watch?v=FbDebSinSQo`, auf FOSDEM 2018. abgerufen am 19.12.2018.

[36] St-Onge, D., V. S. Varadharajan, G. Li, I. Svogor und G. Beltrame: *ROS and Buzz: consensus-based behaviors for heterogeneous teams.* arXiv, 2017.

[37] statista: *Anzahl der Unternehmen in der IT-Branche in Deutschland von 2008 bis 2016.* Web: `https://de.statista.com/statistik/daten/studie/189870/umfrage/anzahl-der-unternehmen-in-der-it-branche-in-deutschland/`. abgerufen am 10.12.2018.

[38] Wikipedia: *Homogenität.* Web: `https://de.wikipedia.org/wiki/Homogenität`. abgerufen am 27.12.2018.

[39] Wikipedia: *Instruction set architecture.* Web: `https://en.wikipedia.org/wiki/Instruction_set_architecture`, auf FOSDEM 2018. abgerufen am 19.12.2018.

[40] Willow Garage: *About Willow Garage.* Web: `http://www.willowgarage.com/pages/about-us`. abgerufen am 04.12.2018.

[41] World Economic Forum: *The Global Information Technology Report 2015*. Web: https://www.youtube.com/watch?time_continue=0&v=8rwzXrK07YE, 2015. abgerufen am 10.12.2018.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 7. Januar 2019

Matthias Wrede