HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Semesterprojekt in den Studiengängen

IT-Produktmanagement & Allgemeine Informatik Neu

Im Sommersemester 2018

Umgebungserfassung mit LIDAR und Turtlebot 2

Referent	Prof. Dr. Elmar Cochlovius
Vorgelegt am	12.07.2018

Projektleitung	Christina Maier, 254779, christina.maier@hs-furtwangen.de
Projektentwicklungsteam	Maxim Balsacq, 25463, maxim.balsacq@hs-furtwangen.de
	Artur Dick, 252211, artur.dick@hs-furtwangen.de
	Alexandra Garber, 251583, alexandra.garber@hs-furtwangen.de
	Jonathan Merkel, 251554, jonathan.merkel@hs-furtwangen.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

[Abkürzung]	[Ausgeschriebene Abkürzung]
LIDAR	Light Detection And Ranging
SLAM	Simultanious Localization and Mapping
ROS	Robot Operating System

1 Einleitung

Ein großes Thema in der heutigen Zeit, ist das autonome Fahren. Dieses soll ermöglichen, dass Fahrzeuge sich größtenteils ohne manuelle Steuerung fortbewegen können. Durch angebrachte Sensoren wird das Umfeld der Fahrzeuge zeitnah wahrgenommen und ausgewertet. So können Hindernisse frühzeitig erkannt und ihnen ausgewichen werden. Da diese Thematik auch in der Zukunft eine große Rolle spielen wird, wollen wir mit unserem Semesterprojekt ebenfalls ein System entwickeln, das dem Turtlebot ermöglicht seine Umgebung möglichst genau zu erfassen und zu analysieren. Durch die Auseinandersetzung mit diesem Thema, wollen wir die Funktionsweisen eines LIDAR-Sensors (Light Detection and Ranging) besser verstehen und nachvollziehen können.

2 Projektauftrag

2.1 Aufgabenstellung

Die Aufgabenstellung des Projektes befasst sich mit der Einarbeitung in die SLAM-Thematik (Simlutanious Localization and Mapping), sowie Projekt-Planung und -Organisation. Die zwei Hardwarekomponenten, LIDAR-Sensor und Turtlebot, werden kombiniert und auf ihre Funktionsweisen untersucht. Im Rahmen des Projektes soll ein Roboter mit Softwarepaketen realisiert werden, der eine Karte erstellen kann und auf dieser navigieren kann. Idealerweise soll die Kartenerstellen autonom stattfinden.

2.2 Zielsetzung

Z1: Entwicklung eines Prototyps zur Umgebungserfassung (SLAM) in Echtzeit.

Z2: Selbständiges und zielgerichtetes Fahren/Bewegen im Raum des Prototyps.

Z3: Analysieren der Umgebung und Erstellen einer abstrakten Karte durch das zu entwickelnde System.

Z4: Selbständiges Navigieren des Turtlebot zu einem gewählten Punkt auf der Karte.

3 Methodisches Vorgehen

Projektorganisation und Projektteam

Entwicklungs-team	Vertreter	Rolle	Aufgabenbereich
Jonathan Merkel (AIN)	Alexandra Garber	Navigator-Experte	Inbetriebnahme der Navigationsnodes.
Alexandra Garber (AIN)	Maxim Balsacq	Sensoriker	LIDAR-Sensor in den Turtlebot einbinden, verknüpfen und konfigurieren.
Artur Dick (AIN)	Jonathan Merkel	Integrator	Erstellung eines Grundsystems mit Hilfe eines Raspberry Pis. Dient der Steuerung des Turtlebots und verbindet die Hardwarekomponenten miteinander.
Maxim Balsacq (AIN)	Alexandra Garber	Systemad-ministrator	System Instandhaltung und Pflege, WLAN aufsetzten, Erzeugung von Startskripten
Christina Maier (ITP)	Artur Dick	Projektleiter	Projektplanung, -organisation und -dokumentation

Tabelle 1: Projektorganisation und Projektteam

3.1 Projektstrategie

Um unser Projekt zufriedenstellend abzuschließen, werden wöchentlich Besprechungen im Team durchgeführt. Eine aktive Kommunikation zwischen allen Teammitgliedern soll Missverständnisse von Beginn an vermeiden und entgegenwirken. Zusätzlich findet alle zwei Wochen ein Treffen mit dem Betreuer satt, in dem wir Ihn über unseren aktuellen Stand informieren. Durch klar strukturierte Arbeitspakete werden Aufgaben an das jeweils zuständige Projektmitglied vergeben und auf Erfüllung kontrolliert, damit der Zeitplan möglichst genau eingehalten werden kann. Zur zusätzlichen Überprüfung werden Meilensteine definiert, an denen wir messen können in welcher Phase sich unser Projekt befindet.

3.2 Meilensteine

Meilenstein	Beschreibung	Termin
M1	Pflichtenheft	13.04.18
M2	Softwareinstallation (Alpha Version) und Test	23.05.18
M4	Softwareerweiterung (Beta Version/Basisversion) und Test	31.05.18 ->14.06.18
M6	Release Version und Test	15.06.18-> 25.06.18
M8	Fertigstellung Handbuch	22.06.18-> 30.06.18
M9	Abschluss und Freigabe	25.06.18-> 27.06.18
M10	Präsentation	06.07.18

Tabelle 2: Meilensteine

Aufgrund von langsamen Vorrankommen bei der Umsetzung der Arbeitspakete, mussten die Phasen und Meilensteine des Projektes auf andere Termine verschoben werden. Die aufgetretenen Komplikationen werden im Verlauf der Dokumentation genauer erläutert.

3.3 Strukturplan und Arbeitspakete

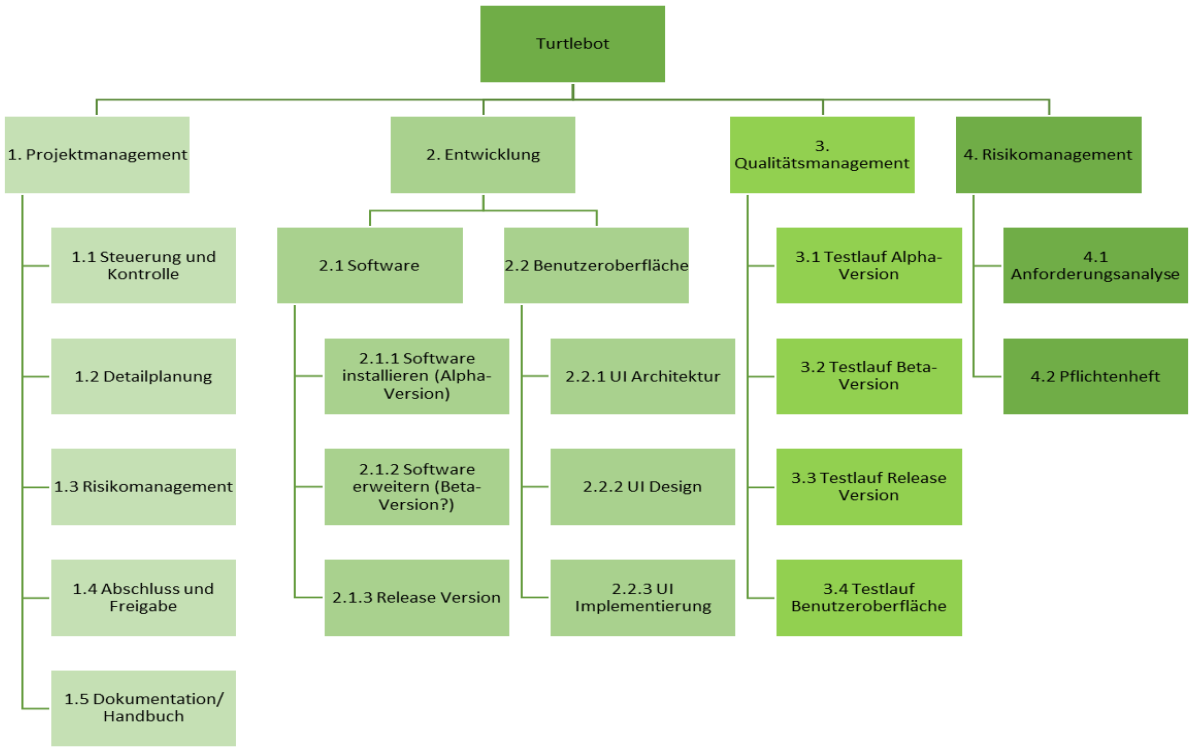


Abbildung 1: PSP zu Beginn

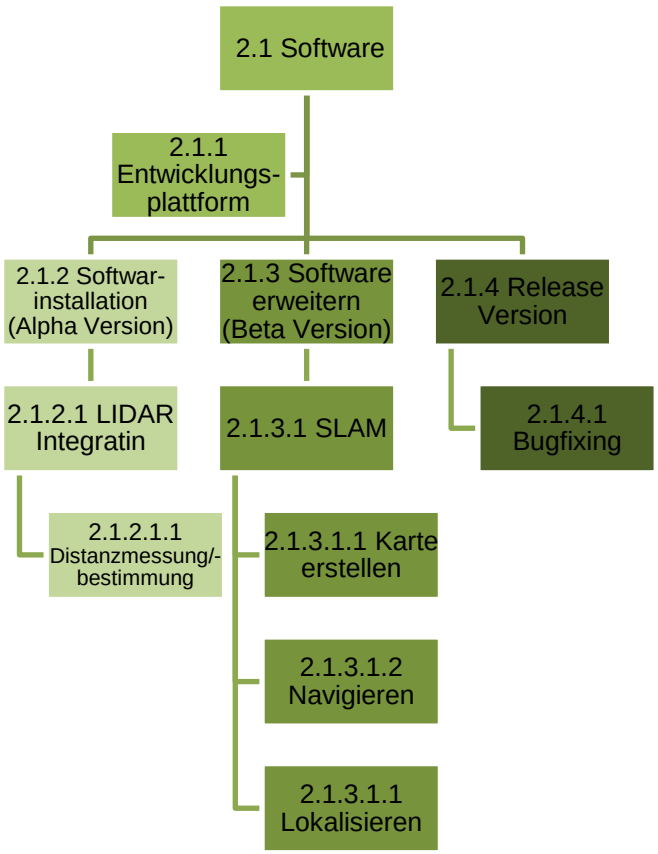


Abbildung 2: PSP Software

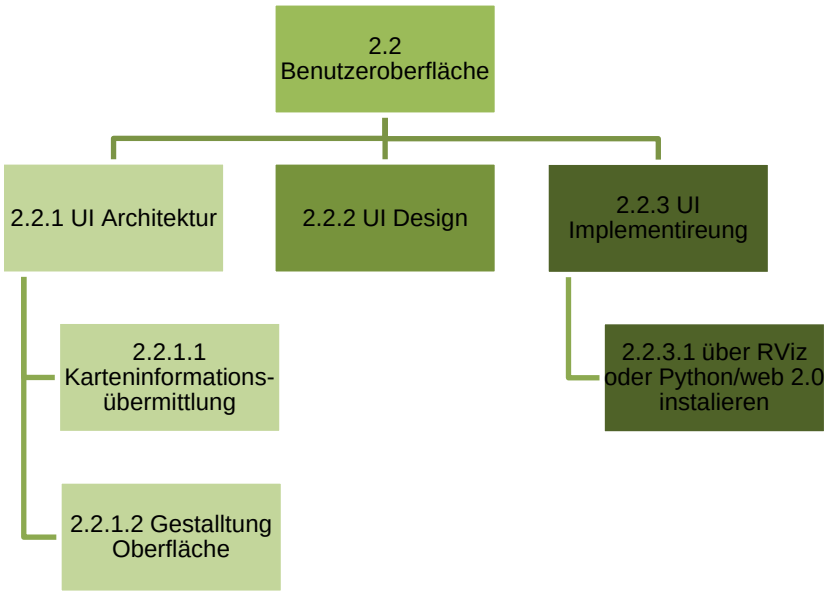


Abbildung 3: PSP Benutzeroberfläche

Da sich die einzelnen Phasen in die Länge gezogen haben, entschlossen wir uns die Arbeitspakete „2.2.1: UI Architektur“, „2.2.2: UI Design“ und „2.2.3: UI Implementierung“ durch „2.2.1: RViz installieren“ zu ersetzen. Somit konnte keine eigene Benutzeroberfläche erstellt werden. Der Ursprüngliche PSP mit dem Zweig „Benutzeroberfläche“ musste gestrichen werden. Der aktuelle PSP ist nun wie folgt:

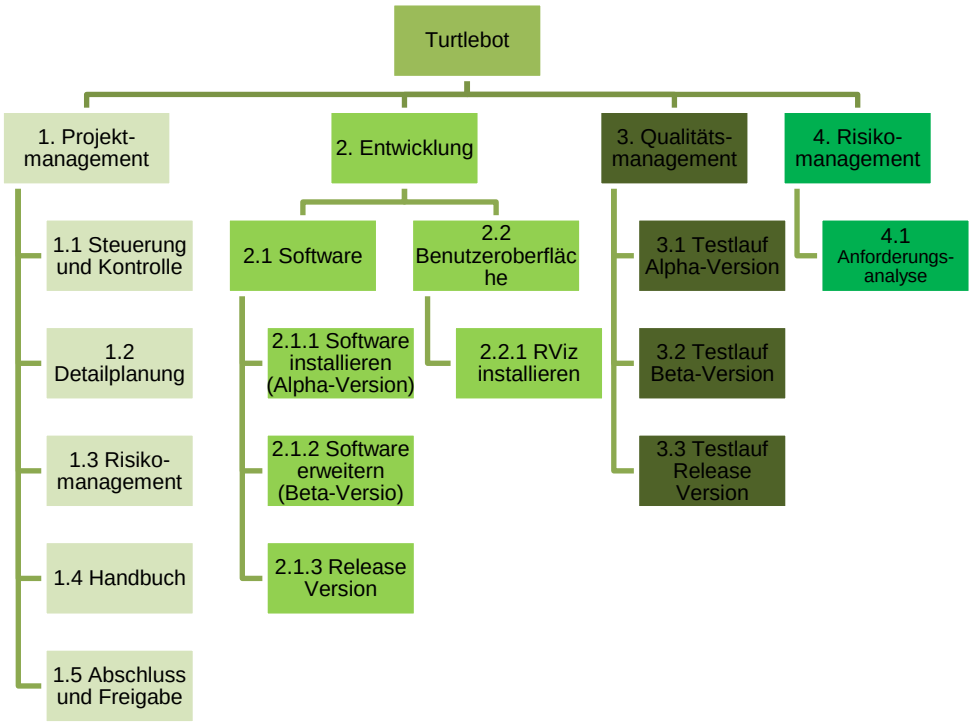


Abbildung 4: PSP Aktuell

1.1 Steuerung und Kontrolle

Beschreibung	Leitung, Steuerung und Kontrolle während des gesamten Projektverlaufes.
Durchführung	Projektleiter
Notwendige Fähigkeiten	Fähigkeit ein Team zu leiten und das Projekt organisiert durchzuführen. Bei Komplikationen Lösungen finden.
Aufwand	60 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Überwachtes und kontrolliertes Projekt
Voraussetzungen	Keine

Tabelle 3: Steuerung und Kontrolle

1.2 Detailplanung

Beschreibung	Analysieren der Anforderungen aus dem Lastenheft und anschließende Erstellung eines detaillierten Pflichtenheftes und Projektplans.
Durchführung	Projektleiter
Notwendige Fähigkeiten	Organisation und Aufwandschätzung der Mitarbeiter
Aufwand	2PT
Leistungserbringung	Eigenleistung
Liefergegenstände	PSP, Pflichtenheft
Voraussetzungen	Lastenheft

Tabelle 4:Detailplanung

1.3 Risikomanagement

Beschreibung	Dokumentation aller möglichen Risiken, welche im Verlauf des Projektes auftreten können. Verfolgung des Projekts, um aufkommende Risiken schnellstmöglich zu erkennen und Lösungen zu finden.
Durchführung	Projektleiter
Notwendige Fähigkeiten	Risiken einschätzen können und gegebenenfalls Planänderung vornehmen.
Aufwand	60 PT
Leistungserbringung	Mögliche Risiken und Notfallpläne
Liefergegenstände	Eigenleistung
Voraussetzungen	Keine

Tabelle 5:Risikomanagement

1.4 Handbuch

Beschreibung	Erstellen eines Benutzerhandbuchs
Durchführung	Projektleiter, Entwicklungsteam
Notwendige Fähigkeiten	Fachwissen über den Projektverlauf
Aufwand	5 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Benutzerhandbuch
Voraussetzungen	Projektplan, Pflichtenheft

Tabelle 6:Handbuch

1.5 Abschluss und Freigabe

Beschreibung	Finale Überprüfung der Release Version und anschließende Freigabe der Software.
Durchführung	Projektleiter, Entwicklungsteam
Notwendige Fähigkeiten	Zuverlässigkeit, Genauigkeit
Aufwand	3 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Finale Software
Voraussetzungen	Release Version

Tabelle 7: Abschluss und Freigabe

2.1.1 Entwicklungsplattform

Beschreibung	Implementierung des ROS-Betriebssystems (Robot Operating System). Installieren von ROS und notwendigen Paketen (RP-LIDAR und nav2d)
Durchführung	Navigator, Sensoriker, Integrator
Notwendige Fähigkeiten	Grundwissen Programmierung
Aufwand	3 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Arbeitsumgebung
Voraussetzungen	Ubuntu

Tabelle 8: Entwicklungsplattform

2.1.3.1.1 Karte erstellen

Beschreibung	LIDAR-Sensor integrieren und Karte erstellen mit nav2d.
Durchführung	Sensoriker
Notwendige Fähigkeiten	Linux-Kenntnisse, ROS-Kenntnisse
Aufwand	7 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Darstellung einer Umgebung auf dem Bildschirm
Voraussetzungen	ROS-Betriebssystem, LIDAR-Sensor

Tabelle 9: Karte erstellen

2.1.3.1.2 Navigieren

Beschreibung	Autonome Navigation für den Turtlebot einrichten und konfigurieren.
Durchführung	Navigator und Integrator
Notwendige Fähigkeiten	Linux-Kenntnisse, ROS-Kenntnisse
Aufwand	1 PT
Leistungserbringung	Eigenleistung/Fremdleistung
Liefergegenstände	Navigation
Voraussetzungen	ROS-Betriebssystem, nav2d, LIDAR-Sensor

Tabelle 10: Navigieren

2.1.3.1.3 Lokalisieren

Beschreibung	Lokalisieren des eigenen Punktes im Raum.
Durchführung	Navigator und Integrator
Notwendige Fähigkeiten	Linux-Kenntnisse, ROS-Kenntnisse
Aufwand	6 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Lokalisierung
Voraussetzungen	ROS-Betriebssystem, LIDAR-Sensor, Datensicherung

Tabelle 11: Lokalisieren

2.1.2.1 LIDAR Integration

Beschreibung	LIDAR-Sensor am Turtlebot anbringen und anschließen.
Durchführung	Integrator
Notwendige Fähigkeiten	ROS-Kenntnisse
Aufwand	3 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Integration des LIDAR-Sensors
Voraussetzungen	ROS-Betriebssystem, LIDAR-Sensor

Tabelle 12: LIDAR Integration

2.1.2.1.1 Distanzmessung/-bestimmung

Beschreibung	Integration LIDAR-Sensor, Einrichtung der Distanzmessung
Durchführung	Integrator
Notwendige Fähigkeiten	ROS-Kenntnisse
Aufwand	1 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Korrekte Distanzwerte, erste graphische Darstellung der Sensor-daten
Voraussetzungen	ROS-Betriebssystem, LIDAR-Sensor, RViz

Tabelle 13: Distanzmessung/-bestimmung

2.1.3 Software erweitern

Beschreibung	Erweiterung der Alpha-Version zur Beta-Version
Durchführung	Navigator, Integrator, Sensoriker
Notwendige Fähigkeiten	ROS-Kenntnisse, Linux-Kenntnisse
Aufwand	10 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Beta-Version
Voraussetzungen	Abgeschlossene Alpha Version

Tabelle 14: Software erweitern

2.1.3.1 Bug fixing

Beschreibung	Fehlerbehebung der vorherigen Version
Durchführung	Navigator, Integrator, Sensoriker
Notwendige Fähigkeiten	ROS-Kenntnisse, Linux-Kenntnisse
Aufwand	2 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Eine bereinigte Beta-Version
Voraussetzungen	Beta-Version

Tabelle 15: Bug fixing

2.1.4.1 Release Version / Bug fixing

Beschreibung	Erweiterung und Fehlerbehebung der Beta-Version zur Release Version
Durchführung	Navigator, Integrator, Sensoriker
Notwendige Fähigkeiten	ROS-Kenntnisse, Linux-Kenntnisse
Aufwand	3 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Release Version
Voraussetzungen	Fertige Beta-Version

Tabelle 16: Release Version/ Bug fixing

2.2.1 RViz installieren

Beschreibung	Das Softwareprogramm RViz installieren.
Durchführung	Integrator, Sensoriker, Navigator
Notwendige Fähigkeiten	ROS-Kenntnisse
Aufwand	2 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Benutzeroberfläche
Voraussetzungen	ROS von Vorteil

Tabelle 17: RViz installieren

3.1 Testlauf Alpha-Version

Beschreibung	Testen der funktionalen Anforderungen und Fehlerfindung, inklusive Dokumentation, sodass diese von den Softwareentwicklern behoben werden kann.
Durchführung	Navigator, Integrator, Sensoriker
Notwendige Fähigkeiten	Genauigkeit, Fachwissen über Qualitätsrichtlinien
Aufwand	2 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Fehler Alpha-Version
Voraussetzungen	Alpha-Version

Tabelle 18: Testlauf Alpha-Version

3.2 Testlauf Beta-Version

Beschreibung	Testen der funktionalen Anforderungen und Fehlerfindung, inklusive Dokumentation, sodass diese von den Softwareentwicklern behoben werden kann.
Durchführung	Navigator, Integrator, Sensoriker
Notwendige Fähigkeiten	Genauigkeit, Fachwissen über Qualitätsrichtlinien
Aufwand	2 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Fehler Beta-Version
Voraussetzungen	Beta-Version

Tabelle 19: Testlauf Beta-Version

3.3 Testlauf Release Version

Beschreibung	Testen der funktionalen Anforderungen und Fehlerfindung, inklusive Dokumentation, sodass diese von den Softwareentwicklern behoben werden kann.
Durchführung	Navigator, Integrator, Sensoriker
Notwendige Fähigkeiten	Genauigkeit, Fachwissen über Qualitätsrichtlinien
Aufwand	2 PT
Leistungserbringung	Eigenleistung
Liefergegenstände	Fehler Release Version
Voraussetzungen	Release Version

Tabelle 20: Testlauf Release Version

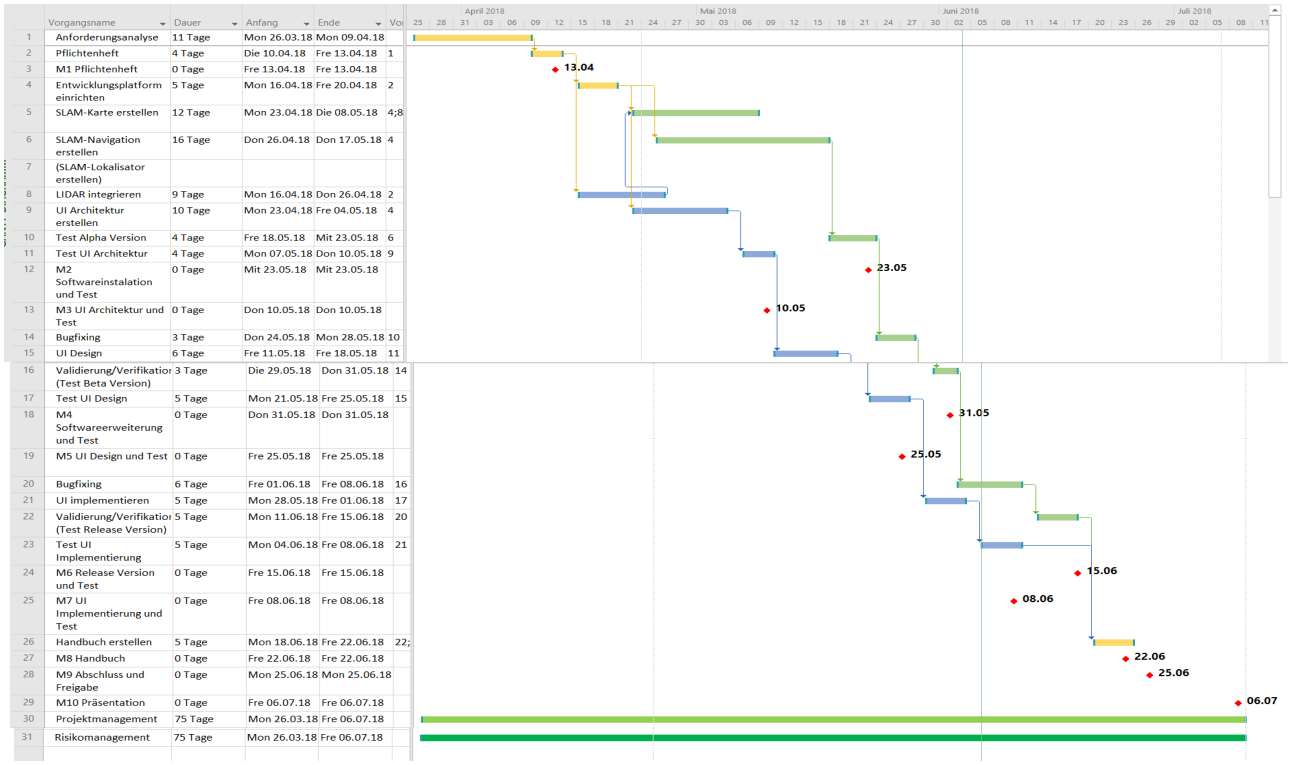
4.1 Anforderungsanalyse

Beschreibung	Anforderungen analysieren und bestimmen
Durchführung	Projektleiter
Notwendige Fähigkeiten	Fachwissen über Requirements Engineering
Aufwand	1 PT
Leistungserbringung	Eigenleistung/Fremdleistung
Liefergegenstände	Klar definierte Anforderungen
Voraussetzungen	Lastenheft

Tabelle 21: Anforderungsanalyse

3.4 Aktivitätenzeitplanung

3.4.1 Gantt-Chart



	Vorgangsname	Dauer	Anfang	Ende	Vo
1	Anforderungsanalyse	11 Tage	Mon 26.03.18	Mon 09.04.18	
2	Pflichtenheft	4 Tage	Die 10.04.18	Fre 13.04.18	1
3	M1 Pflichtenheft	0 Tage	Fre 13.04.18	Fre 13.04.18	
4	Entwicklungsplatform einrichten	5 Tage	Mon 16.04.18	Fre 20.04.18	2
5	SLAM-Karte erstellen	12 Tage	Mon 23.04.18	Die 08.05.18	4;8
6	SLAM-Navigation erstellen	16 Tage	Don 26.04.18	Don 17.05.18	4
7	(SLAM-Lokalisator erstellen)				
8	LIDAR integrieren	9 Tage	Mon 16.04.18	Don 26.04.18	2
9	UI Architektur erstellen	10 Tage	Mon 23.04.18	Fre 04.05.18	4
10	Test Alpha Version	4 Tage	Fre 18.05.18	Mit 23.05.18	6
11	Test UI Architektur	4 Tage	Mon 07.05.18	Don 10.05.18	9
12	M2 Softwareinstalation und Test	0 Tage	Mit 23.05.18	Mit 23.05.18	
13	M3 UI Architektur und Test	0 Tage	Don 10.05.18	Don 10.05.18	
14	Bugfixing	3 Tage	Don 24.05.18	Mon 28.05.18	10
15	UI Design	6 Tage	Fre 11.05.18	Fre 18.05.18	11
16	Validierung/Verifikation (Test Beta Version)	3 Tage	Die 29.05.18	Don 31.05.18	14
17	Test UI Design	5 Tage	Mon 21.05.18	Fre 25.05.18	15
18	M4 Softwareerweiterung und Test	0 Tage	Don 31.05.18	Don 31.05.18	
19	M5 UI Design und Test	0 Tage	Fre 25.05.18	Fre 25.05.18	
20	Bugfixing	6 Tage	Fre 01.06.18	Fre 08.06.18	16
21	UI implementieren	5 Tage	Mon 28.05.18	Fre 01.06.18	17
22	Validierung/Verifikation (Test Release Version)	5 Tage	Mon 11.06.18	Fre 15.06.18	20
23	Test UI Implementierung	5 Tage	Mon 04.06.18	Fre 08.06.18	21
24	M6 Release Version und Test	0 Tage	Fre 15.06.18	Fre 15.06.18	
25	M7 UI Implementierung und Test	0 Tage	Fre 08.06.18	Fre 08.06.18	
26	Handbuch erstellen	5 Tage	Mon 18.06.18	Fre 22.06.18	22;
27	M8 Handbuch	0 Tage	Fre 22.06.18	Fre 22.06.18	
28	M9 Abschluss und Freigabe	0 Tage	Mon 25.06.18	Mon 25.06.18	
29	M10 Präsentation	0 Tage	Fre 06.07.18	Fre 06.07.18	
30	Projektmanagement	75 Tage	Mon 26.03.18	Fre 06.07.18	
31	Risikomanagement	75 Tage	Mon 26.03.18	Fre 06.07.18	

Abbildung 5: Gantt-Chart veraltet

Da die einzelnen Phasen während dem Verlauf des Projektes länger verlaufen sind als geplant, mussten Termine verschoben werden und bestimmte Phasen vorgezogen bzw. entfernt werden. Somit hat sich auch die Gantt-Chart verändert.

3.4.2 Abhängigkeiten

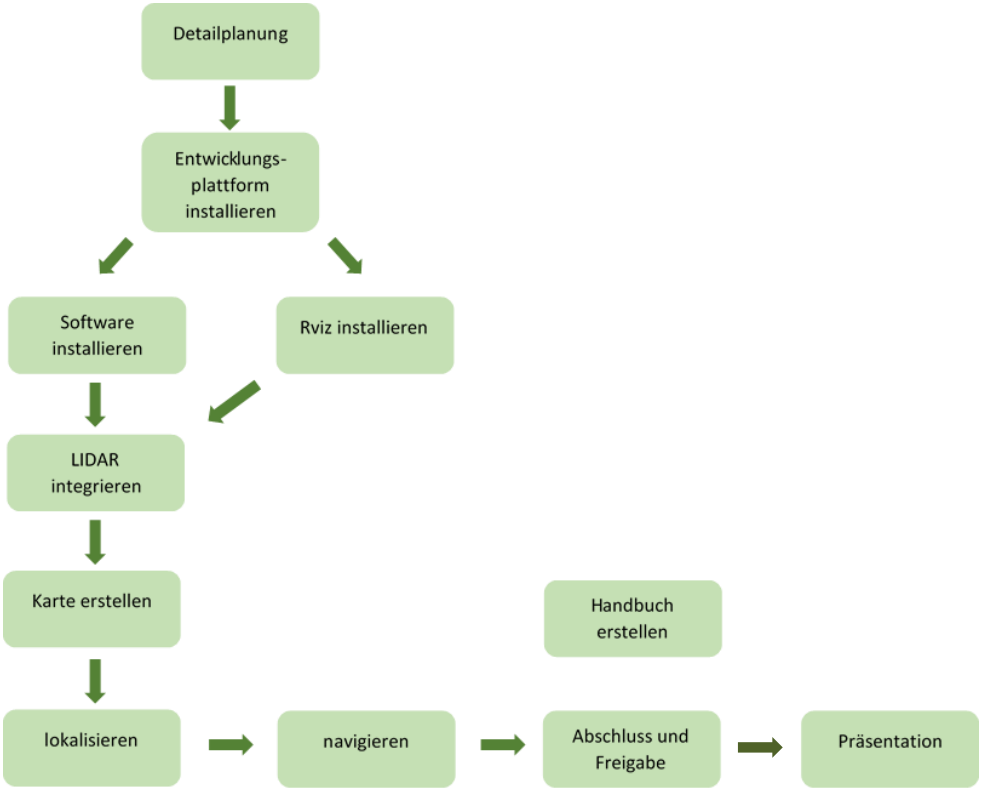


Abbildung 6: Abhängigkeiten

3.5 Risikomanagement

3.5.1 Risikoliste

Risiko Nr.	Eintritts Datum	Kurzbeschreibung des Risikos	SH	WH	Priori-tät
1	28.03.18	Fehlende Zugangsberechtigung für das Labor Zusätzliche Verzögerung durch Ostern	3	0,6	1,8
2	Nicht eingetreten	Turtlebot/LIDAR funktioniert nicht mehr	10	0,35	3,5
3	Nicht eingetreten	Systemabsturz	7	0,35	2,45
4	26.03.18	Wenig wissen über Funktionen und benötigte Ressourcen	4	0,65	2,6
5	05.04.18	Material (Notebook) nicht verfügbar, auch bei regelmäßigen Nachfragen und erinnern daran	5	0,7	3,5
6	21.05.18-25.05.18	Ferien und Feiertage verkürzen die Arbeitszeit	7	0,8	5,6
7	04.06.18	WLAN Ausfall im SmartHome Labor	2	0.4	0.8
8	30.05.18	Entwickler kommen nicht voran/ bleiben bei bestimmten Arbeitspaketen hängen	10	0,6	6
9	28.03.18	Projektmanagement zu oberflächlich/ Leitung zu locker	6	0,5	3
10	Nicht eingetreten	Präsentationstermin kann nicht eingehalten werden	10	0.4	4

Tabelle 22: Risikoliste

3.5.2 Risikoportfolio

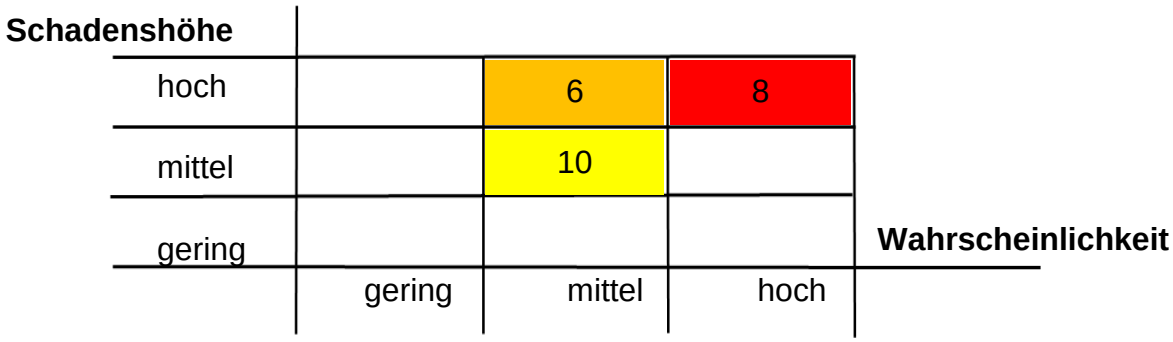


Abbildung 7: Risikoportfolio

3.5.3 Maßnahmenplan

Risiko Nr.	Maßnahme	Erledigt am
1	Selbständige Organisation der Zugangsberechtigung	05.04.18
4	Einarbeiten und recherchieren in das Thema	02.04.18
5	Da das Material nicht zur Verfügung stand war die Alternative die eigenen Notebooks über einen Raspberry PI mit dem Turtlebot und LIDAR-Sensor zu verbinden	16.04.18
6	Arbeiten während den Ferien	23.05.18
7	Alternatives WLAN Netz aufgebaut	04.06.18
8	Vorziehen einzelner Arbeitspakete und parallele Bearbeitung	04.06.18
8	Plan B überlegen	18.06.18
9	Feste Termine definieren	06.04.18

Tabelle 23: Maßnahmenplan

3.5.4 Schwierigkeiten im Projekt

Wie man der Risikoliste entnehmen kann, lagen die größten Schwierigkeiten darin, dass die Arbeitspakete „lokalisieren“ und „navigieren“ mehr Zeit als geplant in Anspruch genommen haben. Die selbstständige Lokalisierung des Turtlebot sorgte für einen großen Zeitverlust und verhinderte jegliche Fortschritte. Da die Navigation auf der Lokalisierung aufbaut, gab es ebenfalls Probleme in diesem Schritt weiterzukommen. Der Turtlebot konnte seine Position nicht bestimmen und erstellte daher Routen, die nicht befahren werden konnten. Auf der erstellten Karte war zusehen, dass der Turtlebot stets in die entgegengesetzte Richtung fuhr, als auf der Karte angezeigt wurde. Zu Beginn waren wir der Meinung das es ein Fehler im Code sei und die Komponenten

deshalb nicht richtig funktionierten. Da sich das Problem nach ein paar Wochen immer noch nicht gelöst hatte, standen wir unter Zeitdruck und beschlossen notfalls auf gewisse Anforderungen zu verzichten und diese für den Ausblick zu verwenden. Ebenfalls machten wir uns Gedanken einen Plan B zu entwickeln, damit das Projekt vorankommt und wir eine Demo für die Präsentation haben. Nach kurzer Überlegung beschlossen wir als Alternative den Turtlebot über einen Controller steuern zu lassen, verwarfen diese Idee aber wieder recht schnell. Der Grund hierfür war, dass unser eigentliches Problem bei der Lokalisierung gelöst wurde, nachdem der Sensor andersrum platziert war. Der Turtlebot konnte seine Position nun bestimmen und die selbständige Navigation funktionierte zum großen Teil. Nach diesem Durchbruch entschieden wir, uns wieder auf das Ursprüngliche Vorhaben zu konzentrieren.

4 Fachlicher Hintergrund

4.1 Rahmenbedingungen

4.1.1 organisatorische Rahmenbedingungen

Der Turtlebot soll sich in Kombination mit dem LIDAR-Sensor möglichst unbeaufsichtigt in Räumen auf dem Boden bewegen können. Durch die vielseitige Anwendung können verschiedene Zielgruppen angesprochen werde. In unserem Projekt beziehen wir uns unter anderem auf Gastronomievorgänge, in denen der Turtlebot als Kellner eingesetzt werden kann. Die tägliche Betriebszeit kann je nach Akkukapazität und Anwendungsbereich variieren. In unserem Projekt beträgt diese ca. 30min – 1h.

4.1.2 Technische Rahmenbedingungen

Die Software läuft auf einem Ubuntu System mit dem ROS-Framework. Für die graphische Darstellung wird RViz verwendet. Zusätzlich wird ein Turtlebot benötigt, der mit einem Notebook und LIDAR-Sensor ausgestattet wird. Die Betriebszeit soll mindestens 4 Stunden halten, danach wird der Akku erneut geladen.

4.1.3 Anforderungen an die Entwicklungsumgebung

Betriebssystem:	Ubuntu 16.04
Framework:	ROS Kinetic Kame
Programmiersprache:	C++/ Python
Hardware:	Turtlebot, LIDAR-Sensor, Notebook; Raspberry Pi, Power Bank
Sonstiges:	Eigenes WLAN Netzwerk

4.2 Systemkontext und Überblick

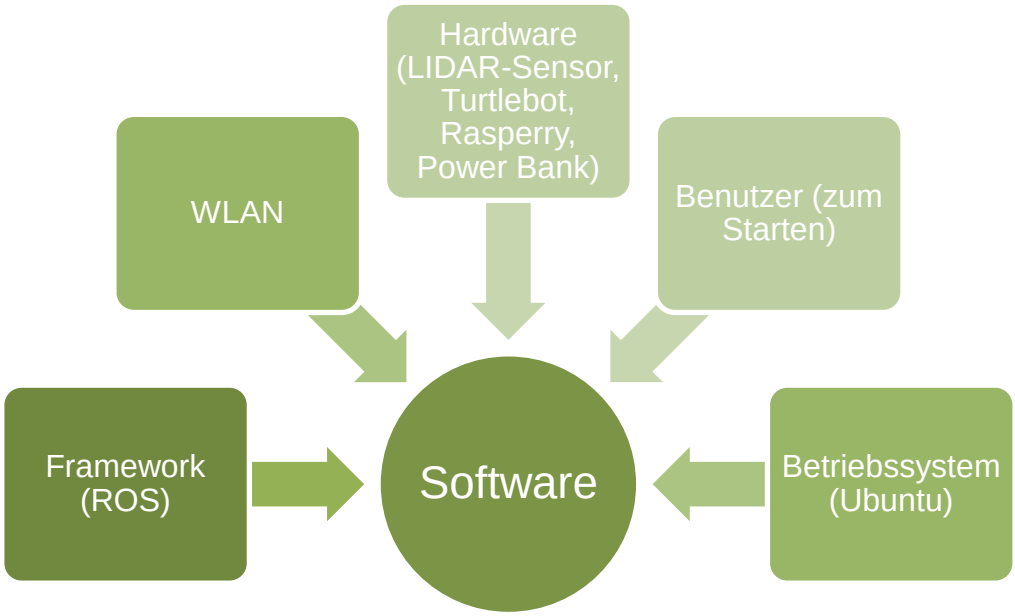


Abbildung 8: Systemkontext

4.2.1 Deployment Diagramm

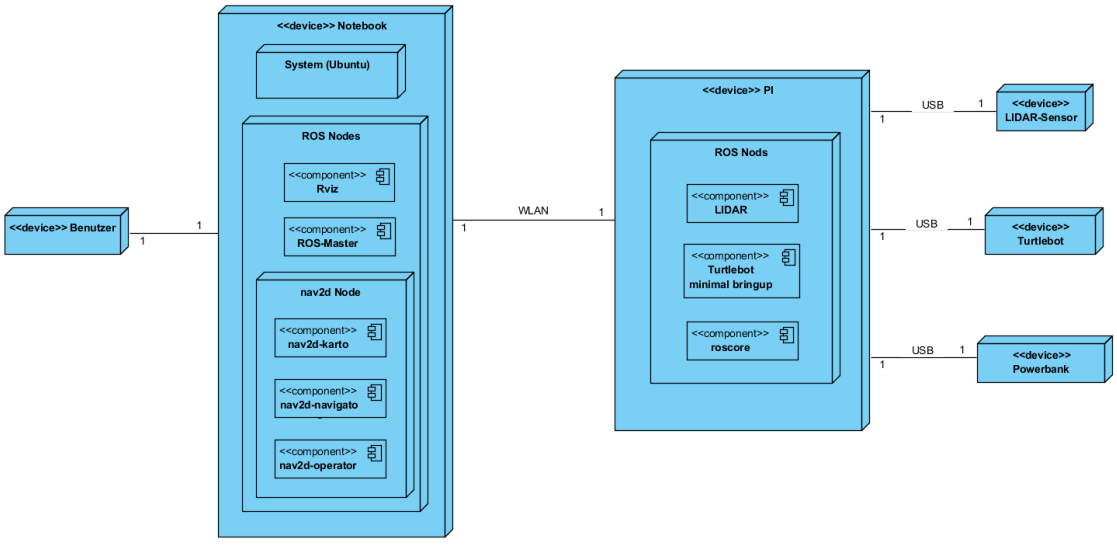


Abbildung 9: Deployment Digramm

4.3 Funktionale Anforderungen

FA1: Der Turtlebot muss fahren und sich dabei in alle Richtungen drehen können.

FA2: Der Turtlebot muss Hindernisse erkennen.

FA3: Der Turtlebot soll Hindernisse dynamisch erkennen können.

FA4: Der Turtlebot muss Hindernisse in die Karte einfügen und wieder löschen können.

FA5: Der Turtlebot muss sich mit dem Notebook verbinden können.

FA6: Der Turtlebot muss sich mit dem LIDAR verbinden können.

FA8: Der Turtlebot muss sich selbstständig im Raum bewegen können

FA7: Der Turtlebot muss sich zielgerichtet autonom im Raum bewegen können.

FA9: Der Turtlebot soll sich selbst navigieren können.

FA10: Der Turtlebot soll seine eigene Position bestimmen können.

FA11: Der Turtlebot soll sich an einen gegebenen Punkt navigieren können.

FA12: Der Turtlebot muss seine Umgebung durch einen LIDAR wahrnehmen können.

FA13: Der Turtlebot muss seine Umgebung analysieren können.

FA14: Der Turtlebot muss sich seine Umgebung merken können.

FA15: Der Turtlebot soll seine Umgebung für unbegrenzte Zeit speichern, bis sie gelöscht wird.

FA16: Der Turtlebot muss die Karte auf einem externen Display darstellen können.

4.4 Qualitätsanforderungen

QA1: Das System muss seine dynamisch ändernde Umgebung beherrschen können.

QA2: Das System kann auf Raspberry Pi laufen.

QA3: Soll die Karte des kompletten Raums innerhalb einer annehmbaren Zeit erstellen (Referenz von 8min im SmartHome-Labor).

4.5 Fachliche Lösung

4.5.1 Use Case 1: Karte erstellen

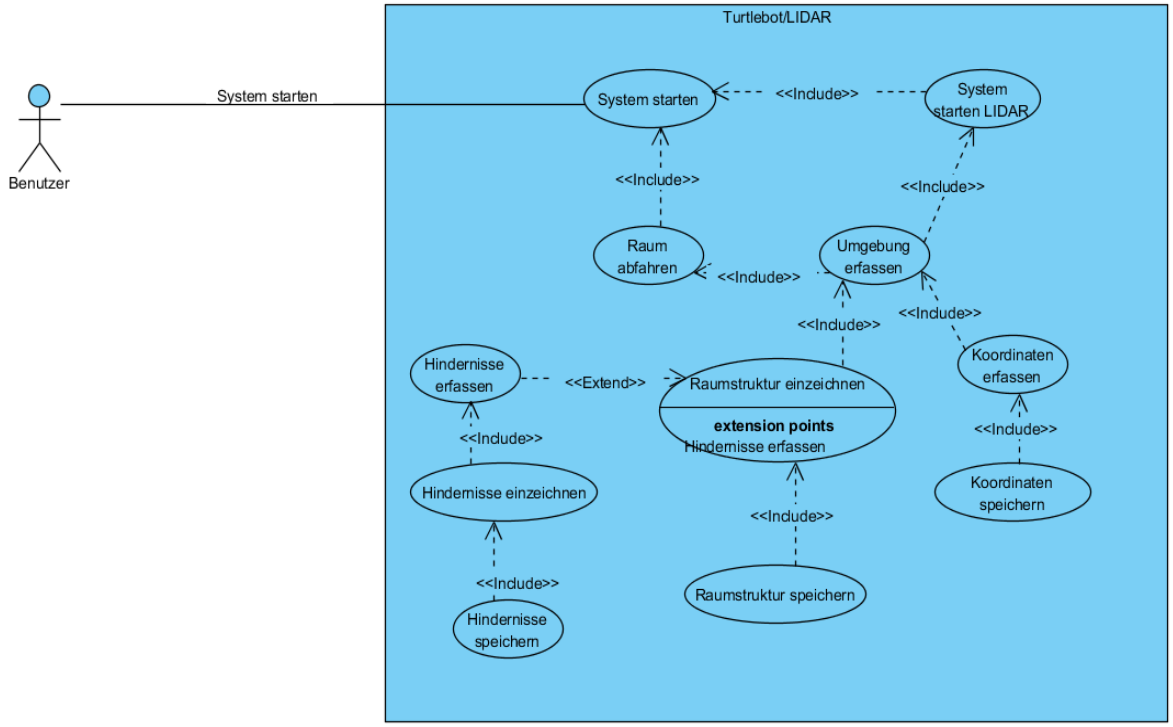


Abbildung 10: Use Case1 Karte erstellen

4.5.2 Use Case 2: navigieren

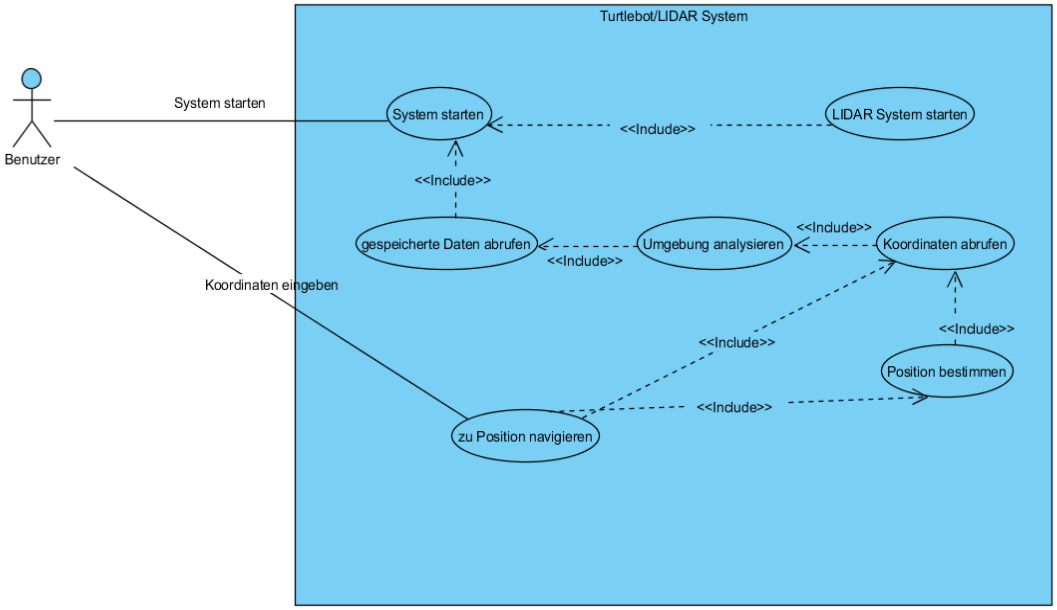


Abbildung 11: Use Case2 navigieren

4.5.3 Skizze Benutzeroberfläche

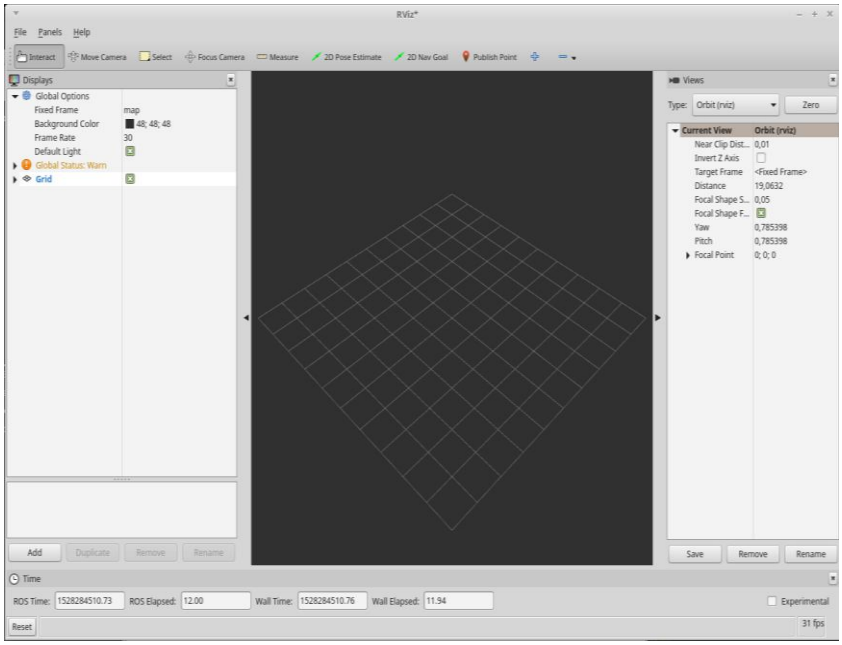


Abbildung 12: Benutzeroberfläche

5 Fallbeispiel Gastronomie

Use Case Gastronomie:

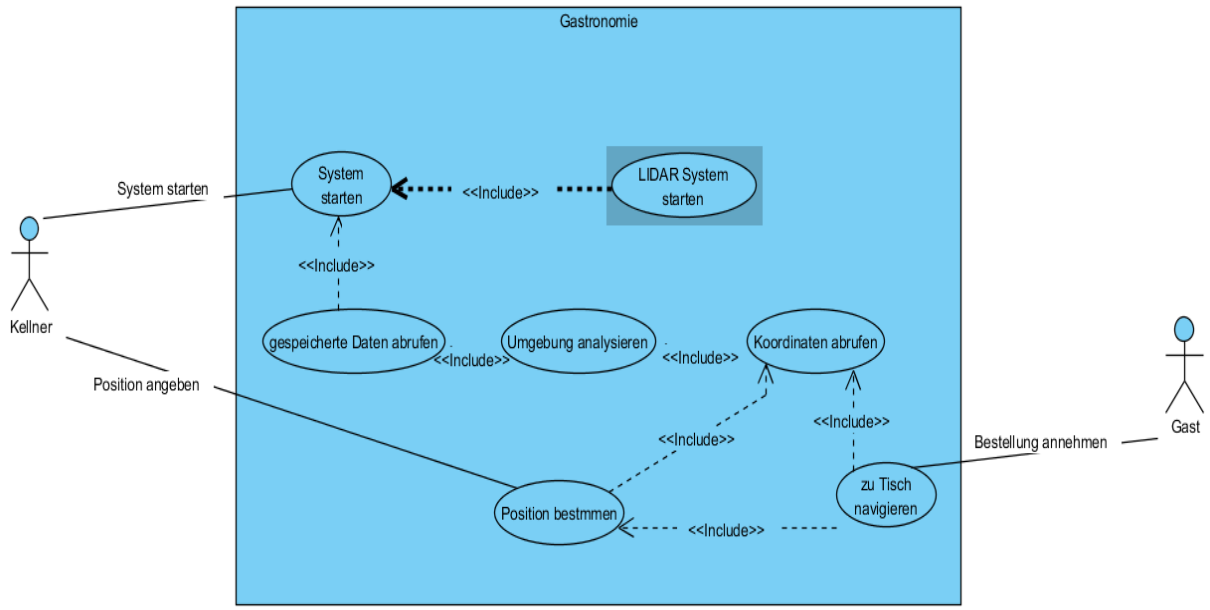


Abbildung 13: Use Case Fallbeispiel Gastronomie

Da sich die autonome Navigation, wie auch der Gebrauch von Robotern, in jeglichen Bereichen anwenden lässt, haben wir uns entschieden unser Fallbeispiel auf die Gastronomie zu beziehen. Auch hier wird seit einigen Jahren das Bedienungssystem durch Apps und Roboter ausgeführt. Der Turtlebot eignet sich gut als eine Alternative für den Kellner. Die obere Tragefläche kann als Tablett zum Servieren verwendet werden. Die autonome Navigation ermöglicht es den Turtlebot selbständig an den gewünschten Platz fahren zu lassen.

In unserer Simulation befindet sich ein Gast im SmartHome Labor, dieser nimmt an einem frei gewählten Punkt Platz und bestellt bei einem Mitglied unseres Teams ein Getränk.

Daraufhin wird auf die Servierplatte des Turtlebots das gewünschte Getränk platziert, das System gestartet und die Position des Gastes über die erstellte Karte auf dem Bildschirm ausgewählt. Der Turtlebot bewegt sich nun mit der Bestellung auf die Zielposition zu und kann somit den Gast bedienen.

Wie das System genau funktioniert und welche Komponenten zum Einsatz kommen, wird im Folgenden beschrieben:

6 Technische Implementierung

6.1 Einführung Turtlebot und Lidar-Sensor

Der Turtlebot ist ein kostengünstiger, mit einem Akku ausgestatteter, quelloffener Roboter der sich dank seiner integrierten Motoren bewegen lässt. In der Front ist ein Tastsensor verbaut, durch welchen ein Auffahren auf ein Hindernis erkannt werden kann. Gesteuert wird dieser über einen Computer, welcher per USB verbunden werden kann. Wird der Steuercomputer in ein drahtloses Netzwerk eingebunden, kann der Turtlebot auch drahtlos aus der Ferne gesteuert werden.
Auf der Rückseite befinden sich Anschlüsse für USB Verbindungen, als auch zur Energieverwaltung angeschlossener Geräte. Am unteren Teil einer der Seiten ist der Power Schalter verbaut, welcher den Turtlebot aktiviert. Gleich daneben ist der Stecker für das Netzteil.
Mit Platten und einem Gestänge können verschiedene Ebenen für Hardware oder sonstiges auf dem Turtlebot montiert und genutzt werden.

Der Turtlebot wurde außerdem mit einem Lidar-Sensor ausgestattet, der die Entfernungen zu Hindernissen erkennt. Ist der Lidar-Sensor in Betrieb, dreht er sich dauerhaft und sendet Laserstrahlen, um Hindernisse zu erkennen und die Entfernungen zu diesen zu berechnen. Wichtig ist dabei, den Lidar-Sensor richtig zu positionieren, dazu später mehr.

In der Abbildung 14: Turtlebot "Harry" sieht man den Turtlebot2, sowie die verschiedenen Komponenten Raspberry Pi, Lidar-Sensor und WLAN-Stick. Abbildung 2 und 3 zeigen den Lidar-Sensor selber und das Schema, wodurch die Positionierung des Lidars besser verstanden werden kann.

Abbildung 14: Turtlebot "Harry"

Die Nachrichten, welche der Lidar Treiber verschickt, sind im sensor_msgs/LaserScan Message Format und beinhalten folgende Informationen:

```
Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Das folgende Bild enthält die Schematische Funktionsweise des Lidar-Sensors:

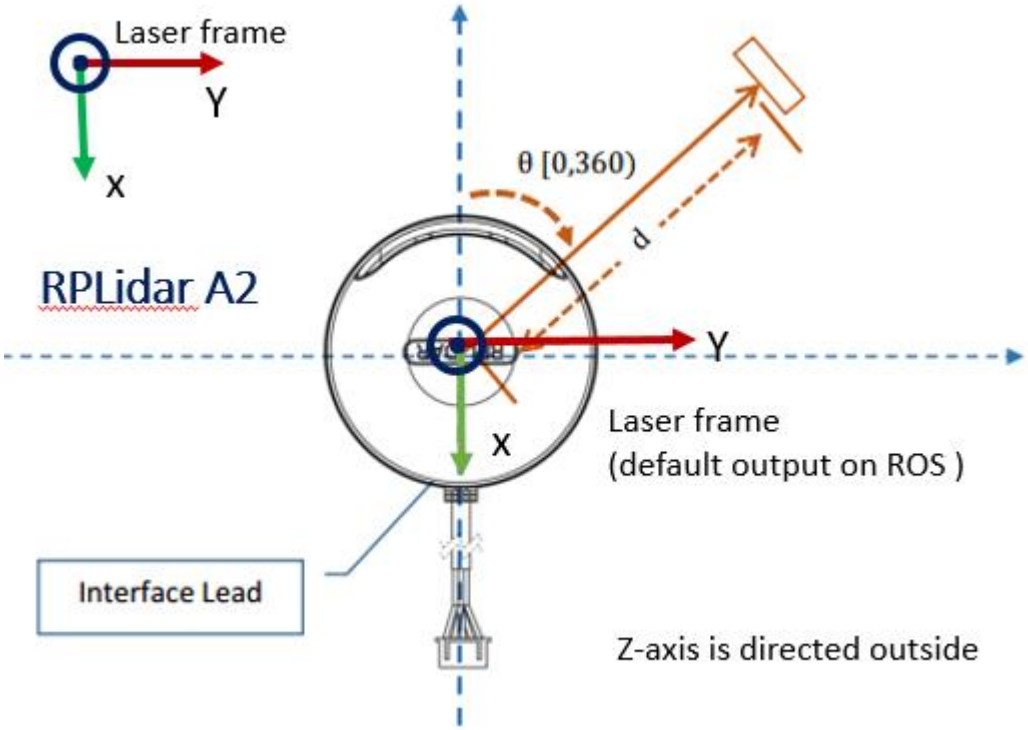


Abbildung 15: Schema Lidar

Dieses Bild zeigt den Lidar-Sensor, wie er auf dem Turtlebot befestigt ist.

Abbildung 16: Lidar-Sensor

```
<?xml version="1.0"?>

<launch>
<param name ="/use_sim_time" value="false"/>
  <!-- Map server-->
  <arg name="map_file" default="/Pfad/zur/Map/MapName.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" output="screen"/>
  <arg name="initial_pose_x" default="0"/>
<!-- Use 17.0 for willow's map in simulation -->
  <arg name="initial_pose_y" default="0"/>
 <!-- Use 17.0 for willow's map in simulation -->
  <arg name="initial_pose_a" default="0"/>
 <include file="$(find turtlebot_navigation)/launch/includes/amcl/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>
  <include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml"/>

 <node pkg="tf" type="static_transform_publisher" name="baselink_laser_bc" args="0 0 0 0 0 0 base_link laser 100" />
 <node pkg="tf" type="static_transform_publisher" name="map_odom" args="0.0 0.0 0.0 0.0 0.0 0.0 map odom 100" />
</launch>
```

6.2 Einführung ROS

Das ROS Software-Framework läuft unter Betriebssystemen wie Microsoft Windows, Linux und macOS und dient als eine Art Baukasten für die Programmierung von Robotern welcher:

- Die Hardware von Robotern abstrahiert
- Gerätetreiber zur Verfügung stellt
- Die Kommunikation zwischen einzelnen Bestandteilen des Systems regelt

Das Framework steht unter der BSD-Lizenz frei zur Verfügung und lässt sich, wie zahlreiche seiner Module, kostenfrei im Quelltext herunterladen. Die Grundidee hinter ROS ist ein Gerüst zur Verfügung zu stellen, mit dem Teile eines Roboters und Software-Programme direkt (peer to peer) miteinander verbunden sind und zusammenarbeiten können. Dabei gibt es Nodes, welche Informationen veröffentlichen (publisher) oder empfangen (subscriber) und diese verarbeiten.

6.2.1 ROS Core

Der Hauptbestandteil des Frameworks liegt im Roscore, der eine Art Backbone für die Kommunikation der einzelnen Systembestandteile (siehe Abbildung Graph des ROS Cores) erstellt. Alle Programme registrieren sich über eine Umgebungsvariable ROS_MASTER_URI zum Roscore und melden diesem, welche Informationen sie veröffentlichen und über welche Sie benachrichtigt werden möchten.

Im Bild sieht man verschiedene Teile eines ROS Systems welche miteinander kommunizieren. Die Linien zwischen den Teilen wird dabei vom Roscore verwaltet. Die großen rechteckigen Kästen sind dabei Namespaces, also Gruppierungen, verschiedener Nodes, wie der Operator Namespace, der selbst verschiedene innere Nodes startet. Topics sind dabei mit vorangehenden Schrägstrich gekennzeichnet.

6.2.2 ROS Nodes

Ein ROS Node ist ein ausführbares Programm, welches Informationen verarbeiten, veröffentlichen, als auch Informationen von anderen Nodes empfangen kann. Für die Kommunikation gilt das Talker/Listener Prinzip. Der Node teilt dem Roscore zu Beginn mit, unter welchem Namen (Topic) er welches Nachrichtenformat veröffentlicht, aber auch für welche Topics dieser sich registrieren möchte. Ein Rosnode wird über den Befehl *rosrun* mit seinen entsprechenden Parametern gestartet, oder kann über eine launch Datei per *roslaunch* gestartet werden. Der Vorteil bei den launch Dateien liegt darin, dass man mit einem Befehl mehrere Nodes gleichzeitig initialisieren und starten kann. Dabei wird auch ein eventuell fehlender Roscore gleich mit gestartet.

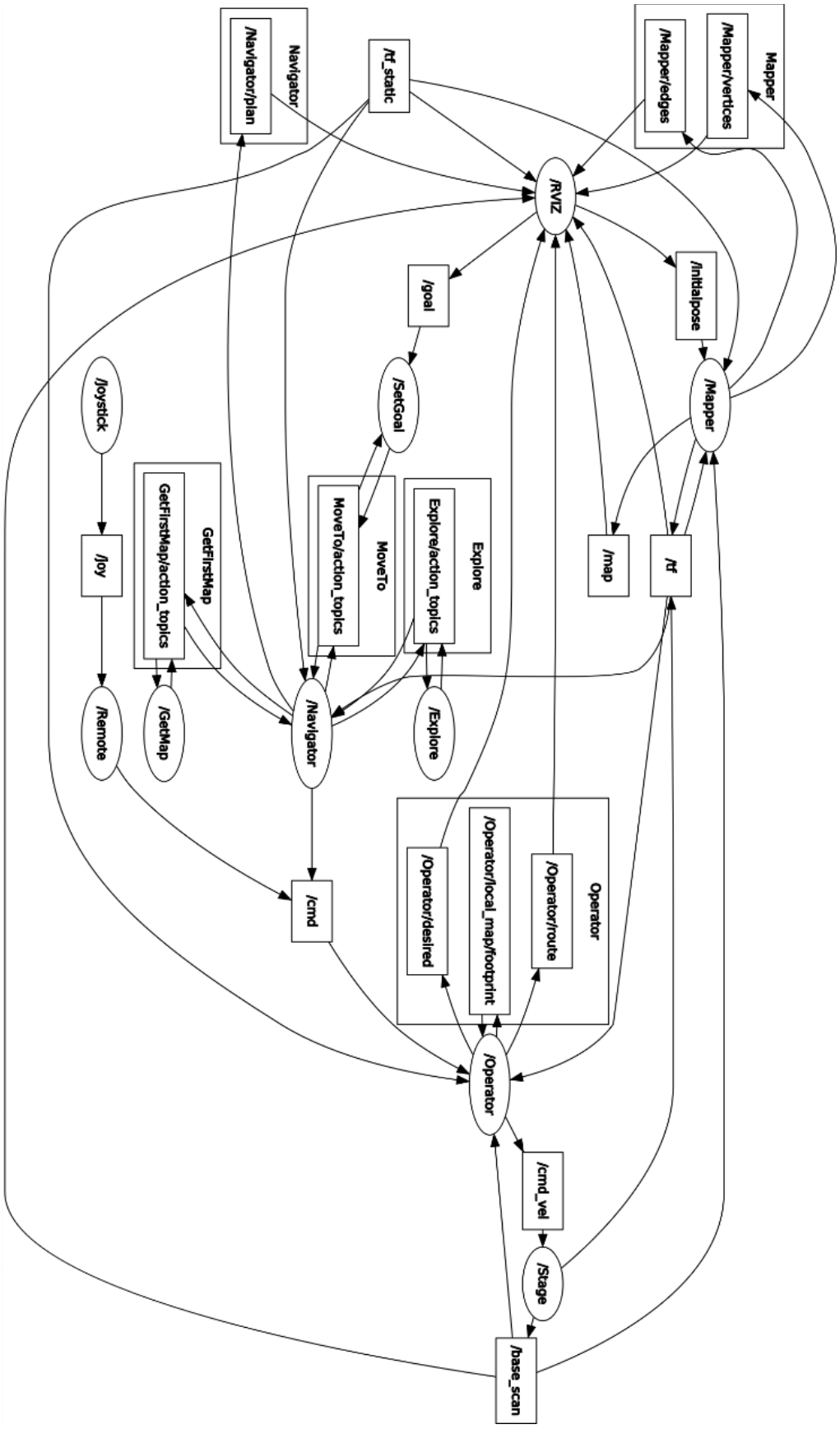


Abbildung 17: ROS-Graph

6.2.3 ROS Topics

Ein ROS Node verarbeitet Informationen und stellt diese anderen Nodes über Topics bereit. Der Node teilt dem Roscore mit, welche Topics dieser mit welchen Informationen zur Verfügung stellt. Als Beispiel dient der Lidar-Node, der die gemessenen Umgebungsentfernungen in einer LaserMSGS (Winkel und Distanz) zur Verfügung stellt. Andere Nodes können sich für solch ein bereitgestelltes Topic registrieren und die Informationen aus diesem empfangen.

6.2.4 RViz

RVIZ ist eine 3D grafische Oberfläche für das ROS Framework. Mit dieser Software können Sensordaten, wie die Umgebungsentfernungen oder erstellte Karten, visuell dargestellt werden.

6.3 Systemarchitektur

Da der Turtlebot einen Computer als Steuergerät benötigt wird ein Raspberry Pi in Verbindung mit einem WLAN-Stick und einer Powerbank am Roboter direkt angebracht. Der Raspberry Pi läuft mit einem Ubuntu Betriebssystem und hat die ROS Pakete aus den ROS Paketquellen installiert. Der Raspberry Pi wurde nach der Anleitung "https://wiki.ubuntu.com/ARM/RaspberryPi" mit einem offiziellen Ubuntu 16.04 aufgesetzt.

Der Raspberry Pi wird in das drahtlose Labor Netzwerk angemeldet und der Roscore gestartet. Über USB sind der Turtlebot und der Lidar-Sensor mit dem Raspberry Pi verbunden und die jeweiligen Nodes (mobile_base und rpLidarnode) werden über udev Regeln beim Anstecken der USB Anschlüsse gestartet. Somit läuft die Verteilung der Sensordaten und die Ansteuerung des Roboters direkt über den Raspberry Pi. Auf einer Workstation wird die ROS_MASTER_URI Variable auf den Raspberry Pi mit dem gestarteten Roscore gesetzt.

```
1. export ROS_MASTER_URI=http://192.168.0.36:11311
```

Dadurch können die Nodes der Workstation mit den beiden Nodes des Steuergeräts kommunizieren.

Auf der Workstation läuft dann die Software, welche mit Hilfe der berechneten Laser Werte eine Umgebungskarte erstellt. Dabei gilt: eine weiße Fläche ist

frei und Hindernisse werden mit der Farbe schwarz eingezeichnet. Weitere Programme dienen dem eigenen Auffinden in der Karte, das Lokalisieren. Das Programmpaket für die Navigation berechnet einen möglichen Weg zwischen der aktuellen Position und einem Ziel, welches von Hand oder vom Exploration Programm vorgegeben wird. Das autonome Erforschen des Raumes geschieht dabei nach einem "Frontier" Algorithmus, welcher jeweils zum nächstgelegenen unerforschten Punkt fahren will um diesen genauer zu betrachten.

6.4 Eingesetzte Komponenten

Für das Arbeiten mit ROS wird das Betriebssystem Ubuntu mit der Version 16.04 benötigt. Es gibt verschiedene Möglichkeiten, wie man das System aufsetzt und ROS für sich nutzbar macht. In der Durchführung wurden verschiedene Möglichkeiten eingesetzt:

- Ubuntu 16.04
- Linux Mint, welches auf Ubuntu 16.04 basiert
- Eine virtuelle Maschine mit Ubuntu 16.04

6.4.1 Entwicklungsumgebung

Damit kann man die ROS-Komponenten, die man benötigt, installieren und verwenden kann, wurde für die Installation der Komponenten und dem Simulator Gazebo dieses Video benutzt: https://www.youtube.com/watch?v=9U6GDonGFHw.
Die Vorgehensweise lässt sich auch durch folgende Befehle ausführen:

```
1. sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -
   sc) main" > /etc/apt/sources.list.d/ros-latest.list'
2. sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-
   key 421C365BD9FF1F717815A3895523BAEEB01FA116
3. sudo apt-get update
4. sudo apt-get install ros-kinetic-desktop-full
5. sudo rosdep init
6. rosdep update
7. echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
8. source ~/.bashrc
9. sudo apt-get install python-rosinstall python-rosinstall-generator python-
   wstool build-essential ros-kinetic-turtlebot-bringup ros-kinetic-turtlebot-
   teleop
```

Als Netzwerk wird das SHLAB02 des SmartHome Labors verwendet.

6.4.2 SLAM Umsetzung

Für die Umsetzung der SLAM Thematik werden die Pakete von nav2d (https://github.com/skasperski/navigation_2d), welche vom Deutschen Institut

für künstliche Intelligenz entwickelt werden, benutzt. Im ROS Wiki (http://wiki.ros.org/nav2d) ist dabei beschrieben wie diese funktionieren und zu nutzen sind. Anhand von vier Tutorials können verschiedene Aufgaben gelöst werden. Für die autonome Umgebungserfassung und Kartenerstellung kommt das Tutorial 3 zum Einsatz.

Mit dem Befehl:

```
1. roslaunch nav2d_tutorials tutorial3.launch
```

wird ein Simulator gestartet welcher eine festgelegte Umgebung für einen Roboter bereit stellt.

Dabei werden dem System, im Simulator per ROS Service Aufrufe, verschiedene Kommandos mitgeteilt. Im zu Beginn ruhenden System kann so mit

```
1. rosservice call StartMapping
```

das Erstellen einer initialen Map veranlasst werden. Der Roboter fährt eine kurze Strecke und macht dann eine 360 Grad Drehung. Eine erste, noch nicht ganz vollständige, Karte ist erstellt worden.

Für ein autonomes Erforschen, der noch nicht entdeckten Stellen der Karte, wird das Kommando

```
1. rosservice call StartExploration
```

an das System geschickt und der Roboter bewegt sich von alleine, fährt durch die ganze simulierte Karte und erforscht diese.

Mit dieser Grundlage muss jetzt ein eigenes ROS Paket geschnürt werden, welche nicht den Simulator nutzt, sondern einen Turtlebot2 und Lidar Sensor.

6.4.3 Harry ROS-Paket

Die Ordnerstruktur wurde von den nav2d_tutorials übernommen und damit ein eigenes Paket namens Harry erstellt, welches das Tutorial 3 auf den Turtlebot bringen soll.

Folgende harry.launch-Datei wurde für die Demonstration des lauffähigen Prototypen verwendet, Erklärungen siehe Kommentare:

```
1.  <launch>
2.      <!-- Lade Grundkonfiguration -->
3.      <rosparam file="$(find harry)/param/ros.yaml"/>
4.      <!-- Definiere Laser_Frame -->
5.      <param name="/laser_frame" value="base_footprint" />
6.
7.      <!-- Operator, der den Turtlebot ansteuert -->
8.      <node name="Operator" pkg="nav2d_operator" type="operator" >
9.          <remap from="/cmd_vel" to="/mobile_base/commands/velocity" />
10.         <rosparam file="$(find harry)/param/operator.yaml"/>
11.         <rosparam file="$(find harry)/param/costmap.yaml" ns="lo-
    cal_map" />
12.     </node>
13.
14.     <!-- Mapper der die Karte aufgrund der Laserwerte erstellt -->
15.     <node name="Mapper" pkg="nav2d_karto" type="mapper">
16.         <rosparam file="$(find harry)/param/mapper.yaml"/>
17.     </node>
18.
19.     <!-- Navigator, der Routen auf der Karte berechnen kann -->
20.     <node name="Navigator" pkg="nav2d_navigator" type="navigator">
21.         <rosparam file="$(find harry)/param/navigator.yaml"/
22.     ></node>
23.
24.     <!--  GetMap erstellt die initiale Karte durch eine kurze Bewe-
    gung und 360 Grad Drehung -->
25.     <node name="GetMap"  pkg="nav2d_navigator" type="get_map_client" />
26.     <!-- Stellt den Exploration Algorithmus zur Verfuegung -->
27.     <node name="Explore" pkg="nav2d_navigator" type="explore_client" />
28.     <!-- Schickt dem Navigator ein Ziel -->
29.     <node name="SetGoal" pkg="nav2d_navigator" type="set_goal_client" />
30.
31.     <!-- Remote Joystick und Controller (todo: test without this)-->
32.     <node name="Joystick" pkg="joy" type="joy_node" />
33.     <node name="Remote" pkg="nav2d_remote" type="remote_joy" />
34.
35.     <!-- RVIZ zur grafischen Darstellung der Karte und der Navigationsda-
    ten -->
36.     <node name="RVIZ" pkg="RViz" type="RViz" args=" -
    d $(find harry)/param/tutorial3.RViz" />
37.     <node pkg="tf" type="static_transform_publisher" name="base_to_la-
    ser_broadcaster"
38.         args="0 0 0 0 0 0 base_link laser 100" />
39.
40. </launch>
```

Das laufende ROS System mit der harry.launch bildet folgenden Graphen:

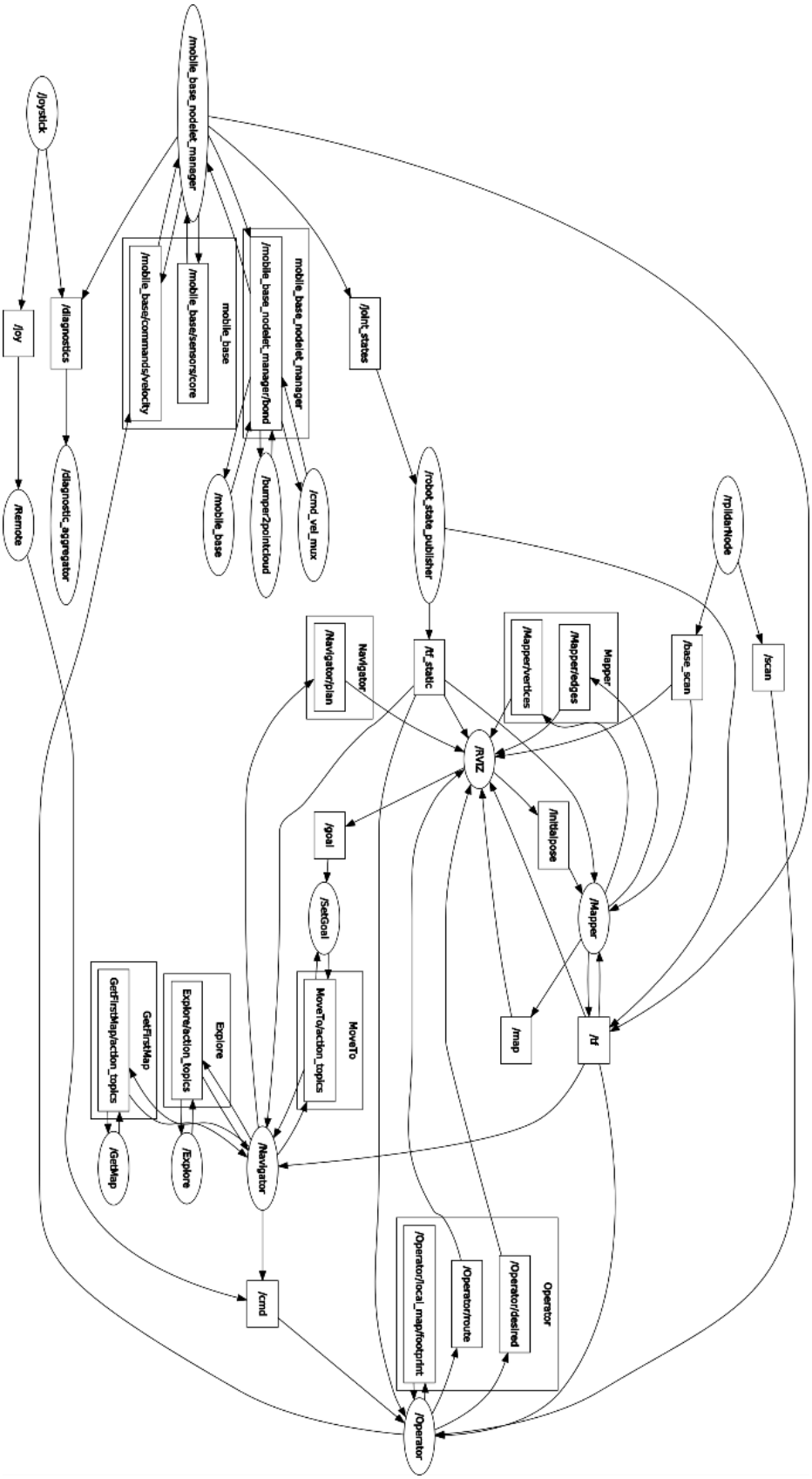


Abbildung 18: Rosgraph des Prototypen

Hierbei sind die einzelnen Nodes (Ellipsen) gut zu erkennen. Nodes verteilen Informationen über Topics (Rechtecke), welche wiederum von anderen Nodes abonniert werden. Bsp. der rpLidarNode in der oberen Hälfte. Dieser bietet die Topics /scan und /base_scan welche von den Nodes RVIZ und Operator gelesen werden. Die großen Rechtecke, welche mehrere Topics umfassen sind Namespaces, also zusammengefasste Topics aus demselben Namensraum.

Die launch-Datei bindet beim Ausführen verschiedene Parameter Dateien mit ein, siehe Anhang auf der CD.

Für eine eigene Demo, den Harry Projektordner direkt nach /opt/ros/kinetic/share/ kopieren und dann per

```
1.  Roslaunch harry harry.launch
```

starten (Vorausgesetzt sind ein laufender rpLidar Node der Laser Werte übermittelt, sowie ein aktiver Turtlebot Node).
Durch den noch nicht ganz verstandenen Aufbau von ROS und Frames hat beim ersten Prototypen das Erstellen der Map nicht ganz funktioniert, weil ein Frame von base_scan gefehlt hatte. Dies wurde behoben, indem der rpLidar Treiber seine Scan Werte auf dem Topic /scan und /base_scan verteilt. Zu Beginn gab es nur den /scan Topic.
Mit diesem zweiten Topic (Zeile 13 im Codelisting vom Lidar Treiber) konnte die Karte richtig erstellt und angezeigt werden. Es ist nicht ausgeschlossen, dass eine einfache Umkonfiguration ebenfalls zu diesem Ziel geführt hätte.

```
1.  int main(int argc, char * argv[]) {
2.      ros::init(argc, argv, "rpLidar_node");
3.
4.      std::string serial_port;
5.      int serial_baudrate = 115200;
6.      std::string frame_id;
7.      bool inverted = false;
8.      bool angle_compensate = true;
9.      float max_distance = 8.0;
10.     int angle_compensate_multiple = 1;//it stand of angle compen-
    sate at per 1 degree
11.     ros::NodeHandle nh;
12.     ros::Publisher scan_pub = nh.advertise<sensor_msgs::La-
    serScan>("scan", 1000);
13.     ros::Publisher base_scan_pub = nh.advertise<sensor_msgs::La-
    serScan>("base_scan", 1000);
14.     ros::NodeHandle nh_private("~");
15.     nh_private.param<std::string>("serial_port", se-
    rial_port, "/dev/ttyUSB0");
```

```
16.     nh_private.param<int>("serial_baudrate", serial_bau-
    drate, 115200/*256000*/);//ros run for A1 A2, change to 256000 if A3
17.     nh_private.param<std::string>("frame_id", frame_id, "laser_frame");
18.     nh_private.param<bool>("inverted", inverted, false);
19.     nh_private.param<bool>("angle_compensate", angle_compensate, false);
20.
21.     ROS_INFO("RPLIDAR running on ROS package rpLidar_ros. SDK Ver-
    sion:"RPLIDAR_SDK_VERSION"");
22.     ...
```

Ein weiteres Problem war die falsche Montage des Lidars auf dem Turtlebot. Dieser war um 180 Grad gedreht auf den Turtlebot aufgebracht und hat zur Odometrie falsche Werte geliefert. Der Treiber bietet zwar die Möglichkeit des Invertierens der Werte, aber das konnte nicht zufrieden stellend getestet werden. Ein einfaches Drehen des Sensors auf dem Turtlebot war für den Prototypen ausreichend. Somit konnte der Turtlebot autonom den Raum erkunden und die Karte, welche man auf folgendem Bild sieht, erstellen.

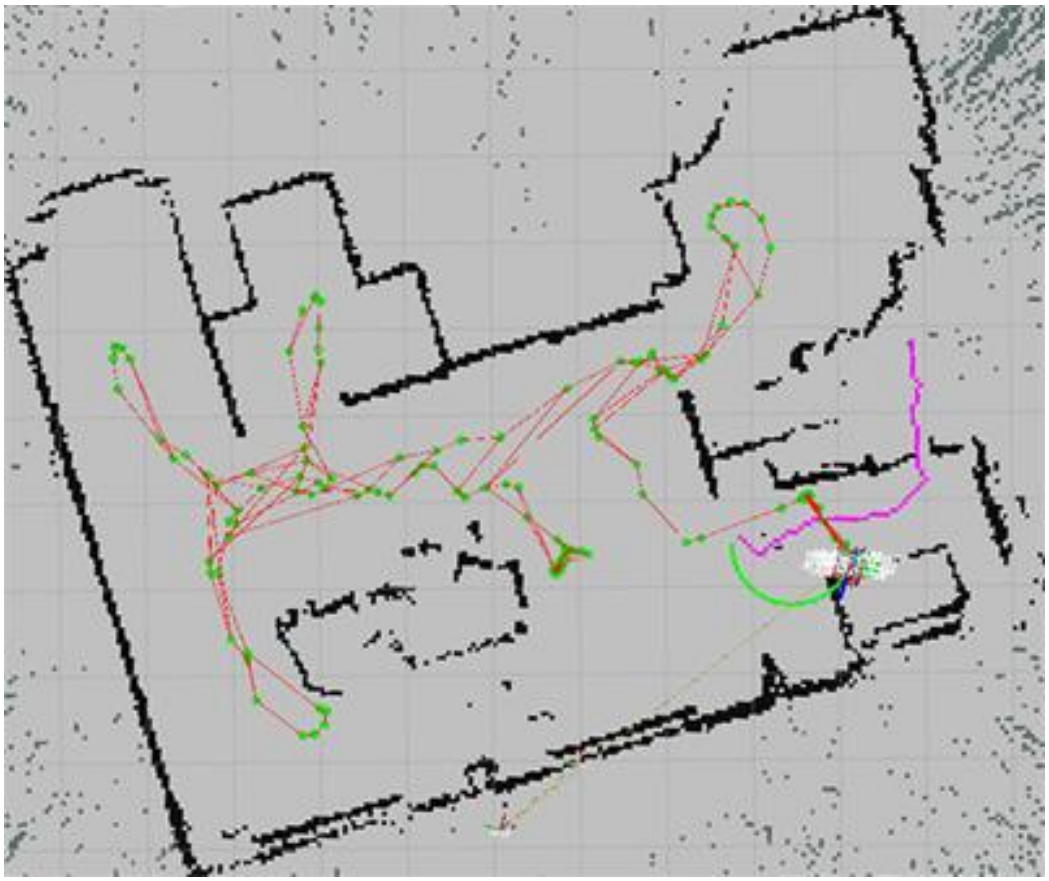


Abbildung 19: Autonom erstellte Karte

Auf der erstellten Karte kann direkt in RViz navigiert werden. Dafür wählt man „2D Nav Goal“ aus und wählt auf auf der Karte einen Punkt und die Orientierung aus. Der Turtlebot navigiert dann zu diesem Punkt und dreht sich in die richtige Richtung.

6.5 Fehlgeschlagene erste Implementierung

Der Ansatz mit dem nav2d-Paket war erst unsere zweite Lösung. Zuerst war geplant die Karte mit Hector_Mapping und einer manuellen Steuerung umzusetzen. Leider funktionierte die Implementierung nicht, da sich der Roboter nicht auf der erstellten Karte orientieren konnte.

6.5.1 Hector_Mapping

Hector_Mapping ist ein Node, der dazu benutzt wird, um die Sensordaten des Lidar-Sensors auszuwerten und daraus eine Karte der Umgebung erstellen zu können.

Hector_Mapping funktioniert ohne Odometrie und benötigt nur wenig Ressourcen, was es ermöglicht den Node auf schwachen Geräten, wie zum Beispiel dem Raspberry-Pi laufen zu lassen.

Die Installation funktioniert über den catkin_workspace. Dafür wird das Git-Repository in den src-Ordner geklont.

```
1. git clone https://github.com/tu-darmstadt-ros-pkg/hector_slam.git cat-
   kin_ws/src/
```

Anschließend wird das Paket vom catkin_ws\-Ordner aus gebaut:

```
1. catkin_make
```

In der Datei mapping.default.launch im Ordner catkin_ws/src/hector_slam/hector_mapping/launch muss folgende Zeile ergänzt werden:

```
1. <node pkg="tf" type="static_transform_publisher" name="base_to_laser_broad-
   caster"
2. args="0 0 0 0 0 0 base_link laser 100" />
```

Damit wird das base_link-Topic in das Laser-Topic konvertiert.

Gestartet werden kann Hector_Mapping dann mit:

```
1. roslaunch hector_slam_launch tutorial.launch
```

Kommt es zu Fehlermeldungen, dann kann dies damit zusammenhängen, dass die simulierte Zeit verwendet wird. Um den Fehler beheben zu können, muss in der Datei /catkin_ws/src/hector_slam/hector_slam_launch/launch/tutorial.launch der Parameter "use_sim_time" auf "false" gesetzt werden.

Wenn der Node funktioniert sollte sich RVIZ öffnen und man sollte bereits erste Ansätze für eine Karte sehen.

In einem weiteren Terminal muss nun die Steuerung per Tastatur gestartet werden:

```
1. roslaunch turtlebot_teleop keyboard_teleop.launch
```

Durch das Herumfahren kann in RVIZ beobachtet werden, wie sich der Roboter bewegt und die Karte langsam aufgebaut wird.

Ist die Karte fertig, kann diese mit folgendem Kommando gespeichert werden:

werden:

```
1. rosrun map_server map_saver -f /ein/Pfad/MapName
```

6.5.2 Navigation mit turtlebot_navigation

Um die erstellte Map benutzen und auf ihr navigieren zu können wird ein Navigation-Stack benötigt. Dafür kann der vorhandene Navigation-Stack des Paketes turtlebot_navigation verwendet werden. Gestartet wird der Node mit einer separaten Datei turtlebot_nav.launch. Die Datei sieht wie folgt aus:

```
1. <?xml version="1.0"?>
2.
3. <launch>
4. <param name ="/use_sim_time" value="false"/>
5.   <!-- Map server-->
6.   <arg name="map_file" default="/Pfad/zur/Map/MapName.yaml"/>
7.   <node name="map_server" pkg="map_server" type="map_server" args="$(arg ma
   p_file)" output="screen"/>
8.   <arg name="initial_pose_x" default="0"/> <!-- Use 17.0 for wil-
   low's map in simulation -->
9.   <arg name="initial_pose_y" default="0"/> <!-- Use 17.0 for wil-
   low's map in simulation -->
10.  <arg name="initial_pose_a" default="0"/>
11.  <include file="$(find turtlebot_navigation)/launch/in-
   cludes/amcl/amcl.launch.xml">
12.    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
13.    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
14.    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
15.  </include>
```

```
16.   <include file="$(find turtlebot_navigation)/launch/in-
    cludes/move_base.launch.xml"/>
17.
18.  <node pkg="tf" type="static_transform_publisher" name="baselink_la-
    ser_bc" args="0 0 0 0 0 0 base_link laser 100" />
19.  <node pkg="tf" type="static_transform_pub-
    lisher" name="map_odom" args="0.0 0.0 0.0 0.0 0.0 0.0 map odom 100" />
20. </launch>
```

Zusätzlich muss in der Datei /opt/ros/kinetic/turtlebot_navigation/param/costmap_common_params.yaml der Robot-Radius auf 0.18 gesetzt werden und der Footprint gegebenenfalls auskommentiert werden.

Gestartet wird das Ganze dann aus dem Ordner der turtlebot_nav.launch-Datei mit:

```
1.  roslaunch turtlebot_nav.launch
```

Navigiert werden kann dann indem man RVIZ mit folgendem Kommando startet:

```
1.  roslaunch turtlebot_rviz_launchers view_navigation.launch --screen
```

Mit "2D-Pose-Estimate" kann die ungefähre Position des Turtlebots bestimmt werden und mit "2D Nav Goal" kann ein Ziel auf der Karte gewählt werden, zu dem sich der Turtlebot schließlich navigiert.

Leider funktionierte dieser Node nicht richtig. Der Roboter konnte seine Position nicht genau bestimmen, beziehungsweise es sprang die Position immer zwischen einem Initialen Punkt und der gewählten Position hin und her, was eine Navigation unmöglich machte.

Die Ursachen für das Problem konnten nicht gefunden werden, weswegen in der finalen Lösung das nav2d-Package benutzt wurde.

7 Nicht/erreichte Ergebnisse

7.1 Produkte und Projektergebnisse

Das Ergebnis unseres Projektes ist wie geplant ein Turtlebot 2 der mit dem LIDAR-Sensor ausgestattet wurde. Der Turtlebot ist in der Lage den Raum durch manuelle wie auch eigenständige Navigation zu befahren und sich selbst im Raum zu lokalisieren. Durch den Sensor ist es möglich die Umgebung während der Fahrt zu analysieren und eine Karte auf einem externen Bildschirm zu erstellen. Die Karte zeigt den aufgenommenen Raum an und macht eine Verfolgung des sich bewegenden Turtlebot möglich. Ebenfalls lässt sich nach Kartenerstellung ein Punkt auf der Karte auswählen der vom Turtlebot autonom angefahren wird.

7.2 Nicht erreichte Ergebnisse

Da die Zeit knapp wurde und technische Schwierigkeiten in der Lokalisierung wie auch eigenständiger Navigation viel Arbeit und Zeit in Anspruch nahmen, konnten die Anforderungen „FA:3“, „FA:4“ und „FA:15“ nicht umgesetzt werden. Ebenfalls fiel das Erstellen einer eigenen Benutzeroberfläche weg.

8 Ausblick und Fazit

8.1 Fazit

Die meiste Zeit stand das Projektteam der Projektarbeit positiv gegenüber. Schon von Anfang an waren alle Mitglieder sehr motiviert sich auf das Thema einzulassen und die verschiedenen Komponenten und Funktionen kennen zu lernen. Dadurch, dass die erste Lösung mit Hector_Mapping nicht funktioniert hatte wurde die Einstellung jedoch zunehmend getrübt. Erst als die alternative Lösung erfolgreich umgesetzt wurde besserte sich die Stimmung wieder.

Die größten Stärken des Teams waren die gute Zusammenarbeit und Kommunikation zwischen den Projektmitgliedern. Die Thematik ROS in Verbindung mit dem Turtlebot und dem Lidar-Sensor ist hoch interessant und hat mit Sicherheit sogar genug Potential für eine WPV mit einem Praktikum. Im Großen und Ganzen war es ein tolles Projekt mit vielen Facetten und Vertiefungsmöglichkeiten.

8.2 Ausblick

Da uns die Zeit nicht gereicht hat alle geplanten funktionalen Anforderungen umzusetzen, haben wir bestimmte Anforderungen gestrichen, diese können in der Zukunft zu einem neuen Release hinzugefügt werden:

FA3: Der Turtlebot soll Hindernisse dynamisch erkennen können.

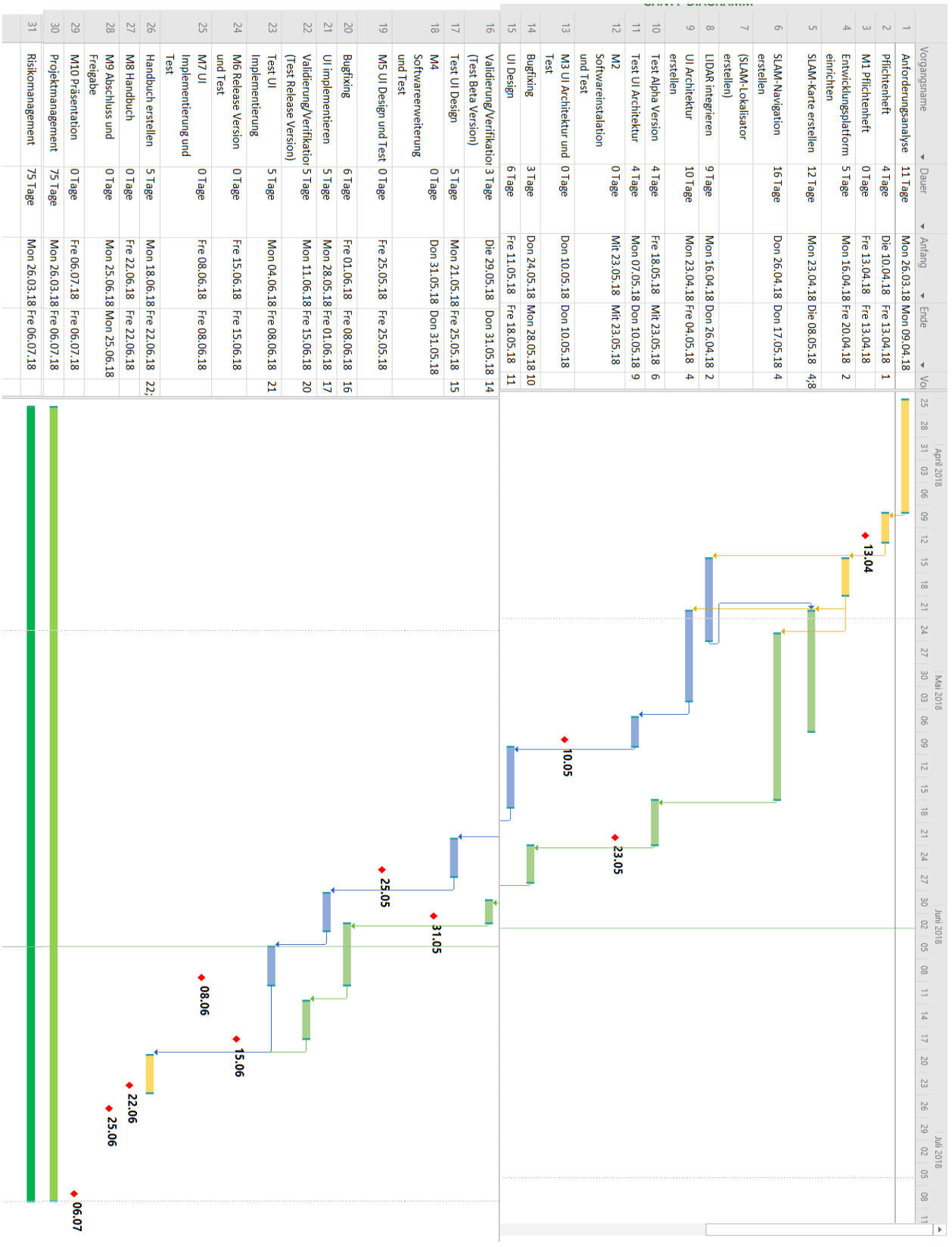
FA4: Der Turtlebot muss Hindernisse in die Karte einfügen und wieder löschen können.

FA15: Der Turtlebot soll seine Umgebung für unbegrenzte Zeit speichern, bis sie gelöscht werden.

Ebenfalls kann durch ein anderes/erweitertes Sensorsystem eine 3D Karte erstellt werden, die das Zurechtfinden auf der Karte vereinfacht und realistischer darstellt.

9 Anhang

9.1 Gantt-Chart



9.2 Deployment Diagramm