



Hochschule für Technik,
Wirtschaft und Kultur Leipzig



Echtzeit Systeme und Mobile Robotik

BELEGARBEIT

Alexander Gründling
Toni Volker Schoechert

GitHub-Repository:
<https://github.com/Toni-Volker-Schoechert/Projekt-EZMR>

Leipzig, den 13. November 2025

Inhaltsverzeichnis

Inhaltsverzeichnis	I
Abbildungsverzeichnis	II
1 Einleitung	1
2 Programmieraufgaben	2
2.1 Sockets/Netcode	2
2.1.1 Teilaufgabe a) – Aufbau eines einfachen TCP-Servers	3
2.1.2 Teilaufgabe b) – Implementierung eines TCP-Clients	4
2.1.3 Teilaufgabe c) – Nachrichtenverwaltung im Server	4
2.1.4 Teilaufgabe d) – Vereinheitlichung von Client und Server	6
3 Praktikum	9
3.1 ROS 2 Grundlagen und Arbeitsumgebung	9
3.2 Erstellung eines eigenen ROS 2-Pakets	9
3.2.1 Einrichtung des Arbeitsbereichs	10
3.2.2 Struktur des ROS 2-Pakets	10
3.2.3 Erstellung des eigenen Pakets	10
3.2.4 Implementierung Programm	11
3.2.5 Build und Test	12
3.2.6 Ergebnis und Beobachtungen	12
3.3 Programmierung eines eigenen Roboters	13
3.3.1 Aufbau und Funktionsweise	13
3.3.2 Durchführung	14
3.3.3 Beobachtungen	15
3.3.4 Diskussion	15
3.3.5 Fazit	15
3.4 Durchführung der Aufgaben vor Ort	16
3.4.1 Aufgabe 1	16
3.4.2 Aufgabe 2	16
3.4.3 Aufgabe 3	17
3.4.4 Aufgabe 4	17
3.4.5 Aufgabe 5	17

Abbildungsverzeichnis

2.1	Server Test 2a)	3
2.2	Server und Client Test 2b)	4
2.3	Server und Client Test 2c)	6
2.4	Kommunikation zwischen zwei Peers Test 2d)	7
3.1	Quadratische Fahrbewegung in der turtlesim-Simulation	13

1 Einleitung

Im Rahmen des Moduls Echtzeitsysteme und Mobile Robotik wurden zwei größere praktische Aufgaben bearbeitet. Ziel war es, grundlegende Konzepte der Prozesskommunikation sowie den Aufbau und die Funktionsweise von Robotersystemen mit ROS 2 zu verstehen und eigenständig umzusetzen. Beide Themen wurden durch eigenständige Programmierung, Tests und Dokumentation praktisch vertieft.

Der erste Teil beschäftigte sich mit der Kommunikation zwischen Prozessen und Anwendungen. Dazu wurden Programme entwickelt, die mithilfe von *Sockets* und *Interprozesskommunikation (IPC)* Daten austauschen können. Die Aufgaben umfassten die Erstellung von Server- und Client-Programmen, den Aufbau von Verbindungen über IP und Port sowie die Verwaltung gemeinsamer Datenstrukturen. Ziel war es, ein Verständnis dafür zu entwickeln, wie Prozesse in Echtzeitsystemen Informationen austauschen.

Im zweiten Teil stand die praktische Arbeit mit dem Robot Operating System 2 (ROS 2) im Mittelpunkt. Zunächst wurden die Grundlagen anhand der turtlesim-Simulation erlernt, um die Kommunikation zwischen Publishern und Subscribern zu verstehen. Anschließend wurde ein eigenes ROS-Paket entwickelt, das Bewegungsbefehle an die Turtle sendet und einfache Fahrbewegungen automatisch ausführt. Dieses Wissen wurde danach auf einen realen mobilen Roboter übertragen, der mit einem ESP32-Mikrocontroller ausgestattet ist und über micro-ROS mit dem ROS-Netzwerk kommuniziert. Der Roboter wurde so programmiert, dass er auf Bewegungsbefehle reagiert und Sensordaten erfassen kann. Abschließend wurde mit dem TurtleBot 4 gearbeitet, um Funktionen wie Kartierung, Navigation und autonome Bewegung kennenzulernen.

Der Beleg dokumentiert die Umsetzung beider Aufgaben, beschreibt die wichtigsten technischen Schritte und fasst die gewonnenen Erkenntnisse zusammen. Außerdem werden Herausforderungen und Lösungsansätze beschrieben, die während der Bearbeitung aufgetreten sind. Der vollständige Quellcode der Programme ist in einem öffentlichen GitHub-Repository ?? abgelegt und ergänzt die hier dargestellten Ergebnisse.

2 Programmieraufgaben

In diesem Kapitel werden die im Rahmen des Moduls durchgeführten Programmieraufgaben beschrieben. Ziel der Aufgaben war es, grundlegende Mechanismen der Prozess- und Netzwerkkommunikation unter Linux praktisch zu erarbeiten und zu verstehen. Dabei wurden zwei Schwerpunkte gesetzt:

- **Sockets/Netcode:** Kommunikation zwischen Programmen über das Netzwerk (TCP/IP).
- **Interprocess Communication (IPC):** Kommunikation zwischen Prozessen auf demselben System.

Jede dieser Aufgaben umfasst mehrere Teilaufgaben, die schrittweise aufeinander aufbauen. Im Folgenden wird zunächst die Socket-basierte Kommunikation beschrieben, während die IPC-Komponenten im Anschluss im zweiten Teil behandelt werden.

2.1 Sockets/Netcode

In dieser Programmieraufgabe sollte die grundlegende Funktionsweise der Netzwerkkommunikation praktisch nachvollzogen werden. Dazu wurden in mehreren Teilschritten Programme entwickelt, die über das TCP/IP-Protokoll miteinander kommunizieren. Ziel war es, die Prinzipien der Client-Server-Architektur sowie deren Erweiterung zu gleichberechtigten Peer-to-Peer-Verbindungen zu verstehen und zu implementieren.

Die Kommunikation zwischen den Anwendungen erfolgt in dieser Programmieraufgabe über Sockets. Sockets stellen eine Methode der Interprozesskommunikation (IPC) dar, die es ermöglicht, Daten zwischen Prozessen auszutauschen, auf demselben Rechner oder über ein Netzwerk. (? , Kap. 56)

In der Praxis wird unter Linux ein Socket durch den Systemaufruf `socket()` erzeugt, der einen Datei-Deskriptor zurückgibt. Dieser Deskriptor kann anschließend wie ein Ein- oder Ausgabekanal verwendet werden, um Daten zu senden oder zu empfangen (? , man 2 socket).

Im Rahmen dieser Aufgabe wurden mehrere Programme erstellt, die aufeinander aufbauen und schrittweise eine vollständige Kommunikationsstruktur realisieren:

- **Teilaufgabe a):** Aufbau eines Servers, der eine Netzwerkverbindung (Port/IP) zur Entgegennahme von Anfragen erstellt und gegebenenfalls auf diese antwortet.
- **Teilaufgabe b):** Entwicklung einer oder mehrerer Client-Anwendungen, die sich mit dem Server verbinden und Daten austauschen können.
- **Teilaufgabe c):** Erweiterung des Servers um eine gemeinsame Liste, die von den Clients abgefragt und manipuliert werden kann.

2 Programmieraufgaben

- **Teilaufgabe d):** Aufhebung der Trennung zwischen Server und Client. Jede Anwendung agiert sowohl als Server als auch als Client, wodurch eine symmetrische Peer-to-Peer-Kommunikation entsteht.

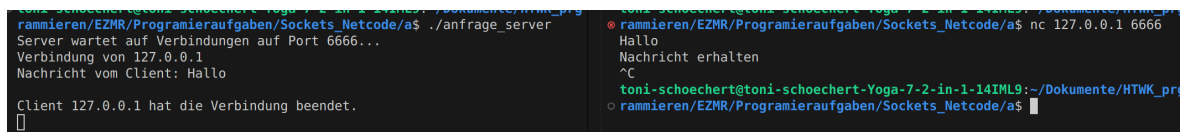
2.1.1 Teilaufgabe a) – Aufbau eines einfachen TCP-Servers

In der ersten Teilaufgabe wurde ein einfacher Server implementiert, der über TCP/IP auf eingehende Verbindungen wartet, Nachrichten von Clients entgegennimmt und entsprechende Bestätigungen sendet. Ziel war es, die grundlegenden Systemaufrufe für den Aufbau einer Netzwerkverbindung unter Linux praktisch nachzuvollziehen.

Der Server erstellt zunächst mit dem Systemaufruf `socket()` einen Kommunikationsendpunkt und bindet diesen über `bind()` an eine lokale IP-Adresse und einen Port (in diesem Fall 6666). Anschließend wird der Socket mit `listen()` in den passiven Zustand versetzt, um eingehende Verbindungen zu akzeptieren. Mit `accept()` werden neue Clients entgegengenommen, und über `recv()` und `send()` erfolgt der Nachrichtenaustausch zwischen Server und Client. (?, man 2 socket) (?, man 2 bind) (?, man 2 listen) (?, man 2 accept), (?, man 2 recv)

Der Server lauscht auf allen verfügbaren Netzwerkinterfaces (`INADDR_ANY`) und ist somit unter der Adresse `127.0.0.1:6666` lokal erreichbar. Zur Überprüfung der Funktion wurde der Server zunächst mit dem Tool `netcat (nc)` getestet. Dabei wurde in einem separaten Terminal eine Verbindung zum Server aufgebaut, eine Nachricht gesendet und anschließend die Verbindung beendet.

Abbildung 2.1 zeigt den Testablauf in zwei Terminals. Links ist die Ausgabe des Servers zu sehen, rechts die des Clients, der über `netcat` eine Nachricht sendet. Der Server bestätigt den Erhalt der Nachricht und meldet den Verbindungsabbruch, sobald der Client die Verbindung beendet.



```
rammieren/EZMR/Programieraufgaben/Sockets_Netcode/a$ ./anfrage_server
Server wartet auf Verbindungen auf Port 6666...
Verbindung von 127.0.0.1
Nachricht vom Client: Hallo
Client 127.0.0.1 hat die Verbindung beendet.
^C

rammieren/EZMR/Programieraufgaben/Sockets_Netcode/a$ nc 127.0.0.1 6666
Hallo
Nachricht erhalten
^C
rammieren/EZMR/Programieraufgaben/Sockets_Netcode/a$
```

Abb. 2.1: Testlauf des Servers mit `netcat`: Verbindung, Nachrichtenaustausch und Verbindungsende

Dieser Test diene in erster Linie der Überprüfung der Serverfunktionalität. In der folgenden Teilaufgabe b) wird ein eigener Client implementiert, der denselben Server gezielt ansteuert und eine strukturierte Kommunikation ermöglicht. Damit lässt sich die Funktion des Servers unter realistischen Bedingungen erneut überprüfen und weiter ausbauen.

Das Ergebnis dieser Teilaufgabe zeigt, dass die grundlegende Client-Server-Kommunikation über TCP erfolgreich implementiert wurde. Der Server kann mehrere Verbindungen nacheinander annehmen und reagiert korrekt auf empfangene Nachrichten.

2 Programmieraufgaben

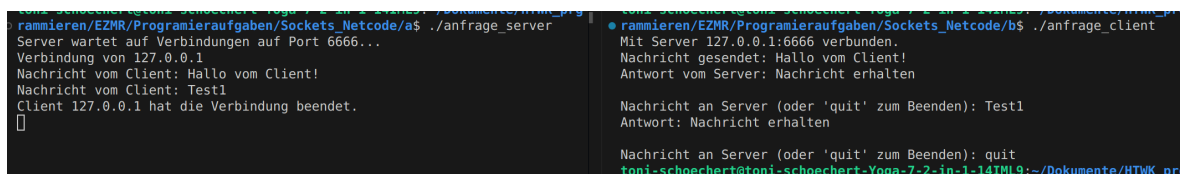
2.1.2 Teilaufgabe b) – Implementierung eines TCP-Clients

In dieser Teilaufgabe wurde eine Client-Anwendung entwickelt, die sich mit dem in Teilaufgabe a) implementierten Server verbindet und Daten mit diesem austauscht. Ziel war es, die Client-Seite der TCP-Kommunikation praktisch nachzuvollziehen und den zuvor erstellten Server zu testen.

Der Client erstellt zunächst mit dem Systemaufruf `socket()` einen eigenen Kommunikationsendpunkt und konfiguriert die Zieladresse des Servers mithilfe der Struktur `sockaddr_in`. Die IP-Adresse (127.0.0.1) wird dabei über den Aufruf `inet_pton()` in eine binäre Netzwerkdarstellung umgewandelt, und der Port 6666 wird durch `htons()` in Network Byte Order gebracht (? , man 3 `inet_pton`). Die Verbindung zum Server wird anschließend mit `connect()` aufgebaut, wobei der Socket-Deskriptor des Clients an den Server gebunden wird (? , man 2 `connect`). Nach erfolgreicher Verbindung kann der Client Daten mit dem Server austauschen. Dies geschieht über die Systemaufrufe `send()` und `recv()`, welche den Versand und Empfang von Nachrichten über den TCP-Datenstrom ermöglichen (? , man 2 `send`) (? , man 2 `recv`).

Nach dem Aufbau der Verbindung sendet der Client eine erste Nachricht ("Hallo vom Client!") an den Server und wartet auf dessen Antwort. Anschließend bleibt die Verbindung offen, sodass der Benutzer interaktiv weitere Nachrichten senden kann. Die Eingabe `quit` beendet die Verbindung. Der Ablauf verdeutlicht den bidirektionalen Charakter der TCP-Kommunikation, bei der beide Seiten sowohl Daten senden als auch empfangen können.

Die Implementierung wurde zunächst mit dem in Teilaufgabe a) entwickelten Server getestet. In Abbildung 2.2 ist der Testablauf dargestellt: Der Client stellt eine Verbindung her, sendet eine Nachricht, erhält eine Bestätigung vom Server und kann anschließend weitere Nachrichten austauschen. Durch Eingabe von `quit` wird die Verbindung ordnungsgemäß geschlossen.



```
rammieren/EZNR/Programieraufgaben/Sockets_Netcode/a$ ./anfrage_server
Server wartet auf Verbindungen auf Port 6666...
Verbindung von 127.0.0.1
Nachricht vom Client: Hallo vom Client!
Nachricht vom Client: Test1
Client 127.0.0.1 hat die Verbindung beendet.
[]

rammieren/EZNR/Programieraufgaben/Sockets_Netcode/b$ ./anfrage_client
Mit Server 127.0.0.1:6666 verbunden.
Nachricht gesendet: Hallo vom Client!
Antwort vom Server: Nachricht erhalten

Nachricht an Server (oder 'quit' zum Beenden): Test1
Antwort: Nachricht erhalten

Nachricht an Server (oder 'quit' zum Beenden): quit
toni-schoechert@toni-schoechert-Yoga-7-2-in-1-14IML9:~/Dokumente/HTWK_pr
```

Abb. 2.2: Interaktive Kommunikation zwischen Client und Server über TCP

Zur Erstellung des Clients wurde wieder ein `Makefile` verwendet. Die erfolgreiche Kommunikation zwischen Client und Server bestätigt die korrekte Umsetzung der grundlegenden TCP-Mechanismen (Socket-Erzeugung, Verbindungsaufbau, Datenübertragung und Verbindungsabbau).

2.1.3 Teilaufgabe c) – Nachrichtenverwaltung im Server

In dieser Teilaufgabe wurde der bestehende Server aus Teilaufgabe a) so erweitert, dass er eine einfache Nachrichtenliste im Arbeitsspeicher verwaltet. Clients können über die bestehende TCP/IP-

2 Programmieraufgaben

Verbindung Befehle an den Server senden, um diese Liste abzufragen oder zu verändern.

Die Kommunikation erfolgt weiterhin über Sockets, wie im Abschnitt 2.1.1 bereits beschrieben. Grundlage bilden die dort erläuterten Systemaufrufe `socket()`, `bind()`, `listen()`, `accept()`, `recv()` und `send()`, die hier unverändert verwendet werden. Der Server nimmt dabei Textbefehle entgegen und führt entsprechende Operationen aus.

Unterstützte Befehle:

- **ADD** `<Text>` – fügt eine neue Nachricht an das Ende der Liste an.
- **LIST** – gibt alle gespeicherten Nachrichten mit Indexnummern aus.
- **DELETE** `<id>` – löscht die Nachricht mit der angegebenen Indexnummer.
- **MOVE** `<from>` `<to>` – verschiebt eine Nachricht innerhalb der Liste.
- **QUIT** – beendet die aktuelle Verbindung.

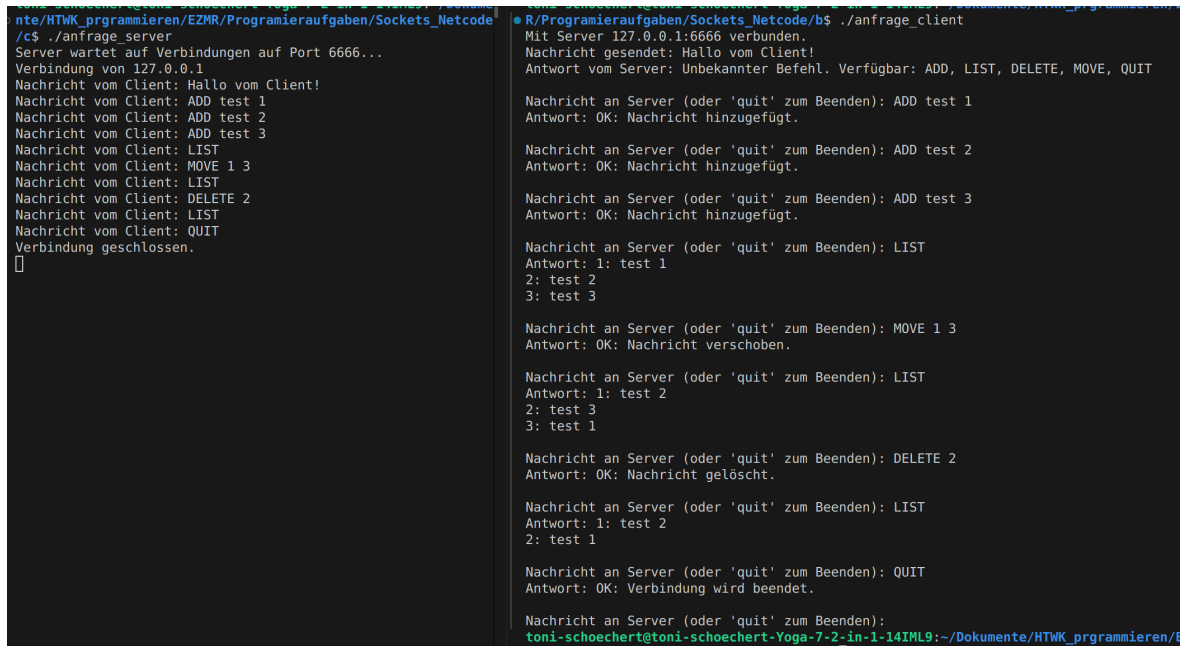
Damit entsteht eine einfache textbasierte Steuerung, bei der jeder Befehl eine bestimmte Listenoperation auslöst. Die Nachrichten selbst werden in einem statischen Array im Hauptspeicher abgelegt. Jede Operation erfolgt sequentiell, d. h. der Server verarbeitet immer nur einen Client gleichzeitig.

Die Funktionsweise des erweiterten Servers ist wie folgt. Nach dem Start erstellt der Server zunächst ein Socket und bindet es an Port 6666, wie in 2.1.1 bereits geschildert. Der Socket wird über `listen()` in den passiven Modus versetzt, sodass eingehende Verbindungen akzeptiert werden können. Hier kommt hinzu, dass jede angenommene Verbindung über eine Schleife verarbeitet wird: der Server empfängt die Eingaben eines Clients über `recv()`, wertet sie per Stringanalyse (`strncmp()`, `strtok()`) aus und führt die passende Operation aus. Die Ergebnisse werden anschließend über `send()` zurückgegeben. Bei der Eingabe von **QUIT** wird die Verbindung geordnet beendet. Zahlenparameter (z. B. bei **DELETE** oder **MOVE**) werden mit `strtol()` in Integer-Werte konvertiert, wobei auf Gültigkeit und Bereichsüberschreitungen geprüft wird.

Der eigene interaktive Client aus Teilaufgabe b) kann für Tests dieses Servers ebenfalls verwendet werden. Dieser sendet beim Start automatisch die Begrüßungsnachricht „Hallo vom Client!“, was hier zu einer kurzen Fehlermeldung führt, da die Eingabe keinem gültigen Befehl entspricht. Anschließend können alle oben genannten Befehle direkt genutzt werden, um die Funktionalität zu prüfen.

Zur Veranschaulichung wurde der Server und der Client in separaten Terminals gestartet (Abbildung 2.3). Links ist die Serverausgabe zu sehen, rechts der Client, der die Befehle eingibt.

2 Programmieraufgaben



```
ntw/HTWK_prgrammieren/EZMR/Programieraufgaben/Sockets_Netcode | ● R/Programieraufgaben/Sockets_Netcode/b$ ./anfrage_client
/c$ ./anfrage_server
Server wartet auf Verbindungen auf Port 6666...
Verbindung von 127.0.0.1
Nachricht vom Client: Hallo vom Client!
Nachricht vom Client: ADD test 1
Nachricht vom Client: ADD test 2
Nachricht vom Client: ADD test 3
Nachricht vom Client: LIST
Nachricht vom Client: MOVE 1 3
Nachricht vom Client: DELETE 2
Nachricht vom Client: LIST
Nachricht vom Client: QUIT
Verbindung geschlossen.
[]

Mit Server 127.0.0.1:6666 verbunden.
Nachricht gesendet: Hallo vom Client!
Antwort vom Server: Unbekannter Befehl. Verfügbar: ADD, LIST, DELETE, MOVE, QUIT

Nachricht an Server (oder 'quit' zum Beenden): ADD test 1
Antwort: OK: Nachricht hinzugefügt.

Nachricht an Server (oder 'quit' zum Beenden): ADD test 2
Antwort: OK: Nachricht hinzugefügt.

Nachricht an Server (oder 'quit' zum Beenden): ADD test 3
Antwort: OK: Nachricht hinzugefügt.

Nachricht an Server (oder 'quit' zum Beenden): LIST
Antwort: 1: test 1
2: test 2
3: test 3

Nachricht an Server (oder 'quit' zum Beenden): MOVE 1 3
Antwort: OK: Nachricht verschoben.

Nachricht an Server (oder 'quit' zum Beenden): LIST
Antwort: 1: test 2
2: test 3
3: test 1

Nachricht an Server (oder 'quit' zum Beenden): DELETE 2
Antwort: OK: Nachricht gelöscht.

Nachricht an Server (oder 'quit' zum Beenden): LIST
Antwort: 1: test 2
2: test 1

Nachricht an Server (oder 'quit' zum Beenden): QUIT
Antwort: OK: Verbindung wird beendet.

Nachricht an Server (oder 'quit' zum Beenden):
```

Abb. 2.3: Kommunikation zwischen Server (links) und Client (rechts) – Test der Befehle ADD, LIST, DELETE, MOVE und QUIT.

Mit dieser Erweiterung kann der Server nicht mehr nur einfache Nachrichten empfangen, sondern verwaltet einen dynamischen Nachrichtenspeicher, der über einfache Textkommandos steuerbar ist.

2.1.4 Teilaufgabe d) – Vereinheitlichung von Client und Server

In dieser Teilaufgabe wird die bisherige Trennung zwischen Client- und Server-Anwendung aufgehoben. Anstelle zweier separater Programme wird eine einheitliche Peer-to-Peer Anwendung entwickelt, in der jede Instanz sowohl eingehende Verbindungen annehmen (Server-Funktion) als auch selbst Verbindungen initiieren (Client-Funktion) kann. Damit kann jede Anwendung gleichzeitig Anfragen empfangen und senden.

Das Programm `anfrage_peer.c` kombiniert die Funktionalitäten der vorherigen Teilaufgaben in einer einzigen Applikation. Es besteht im Wesentlichen aus drei Hauptkomponenten:

- **Listener-Thread:** Der Listener-Thread erstellt mittels `socket()`, `bind()` und `listen()` einen TCP-Server, der auf eingehende Verbindungen wartet. Jede angenommene Verbindung wird an einen separaten Thread weitergegeben, der über `pthread_create()` gestartet wird. (? , man 3 `pthread_create`)
- **Connection-Handler-Threads:** Jeder Thread bearbeitet genau eine aktive Verbindung. Über `recv()` werden eingehende Befehle empfangen, interpretiert und verarbeitet. Nach der Verarbeitung wird eine Antwort mit `send()` an den Peer zurückgeschickt.

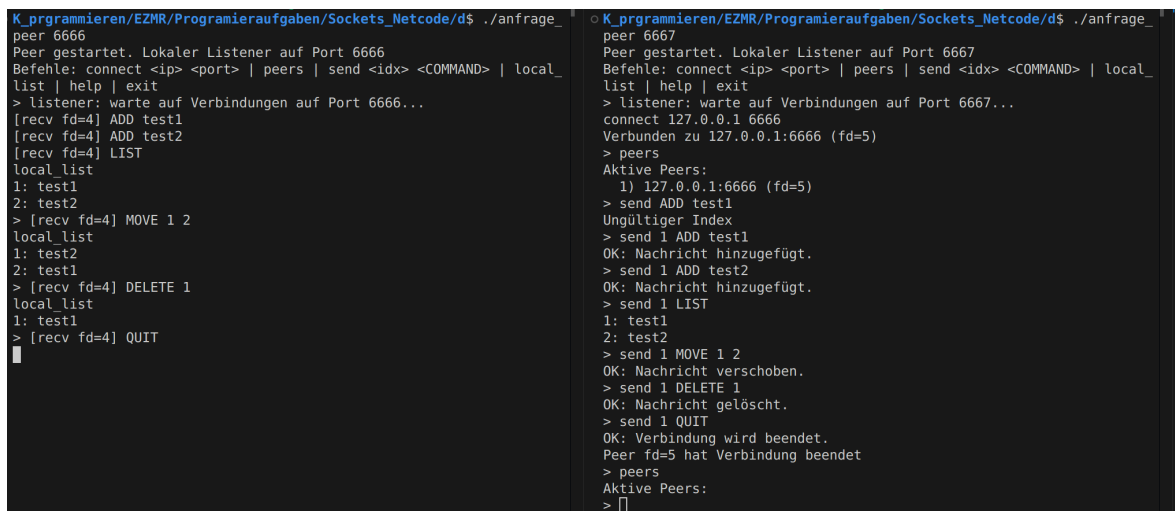
2 Programmieraufgaben

- **Kommandozeilen-Interface (Main Thread):** Das Hauptprogramm stellt eine interaktive CLI bereit, über die Verbindungen hergestellt und Befehle (vgl. Abschnitt 2.1.3) an andere Peers gesendet werden können:
 - connect <ip> <port> – Verbindet zu anderem Peer
 - peers – Listet alle aktiven Verbindungen
 - send <idx> <COMMAND> – Sendet Befehl an Peer mit Index
 - local_list – Zeigt lokale Nachrichtenliste
 - exit – Beendet den Peer

Der Zugriff auf die globale Nachrichtenliste (`messages[]`) und die Peer-Liste (`peers[]`) wird jeweils durch einen `pthread_mutex_t`-Lock geschützt. Dies stellt sicher, dass mehrere Threads nicht gleichzeitig auf dieselben Datenstrukturen zugreifen. (? , man 3 `pthread_create`), (? , man 3 `pthread_mutex_init`) Die Kommunikation erfolgt weiterhin über TCP-Sockets. Jede Nachricht wird als Textbefehl übermittelt und zeilenweise verarbeitet. Zur Verwaltung der Nachrichten werden die Kernfunktionen aus Abschnitt 2.1.3 angepasst und wiederverwendet.

Der Zugriff auf diese Funktionen ist ebenfalls mutex-geschützt.

Zur Überprüfung der Funktionalität wurden zwei Peers gestartet und miteinander verbunden. Peer 1 lauscht auf Port 6666, während Peer 2 sich aktiv verbindet und verschiedene Befehle sendet. Beide Peers laufen parallel in separaten Terminals und kommunizieren über TCP. Im Test werden Nachrichten zwischen den Peers gesendet, gelistet, verschoben und gelöscht, um die Funktionsfähigkeit der P2P-Kommunikation zu demonstrieren.



```
K_programmieren/EZMR/Programieraufgaben/Sockets_Netcode/d$ ./anfrage_
peer 6666
Peer gestartet. Lokaler Listener auf Port 6666
Befehle: connect <ip> <port> | peers | send <idx> <COMMAND> | local_
list | help | exit
> listener: warte auf Verbindungen auf Port 6666...
[recv fd=4] ADD test1
[recv fd=4] ADD test2
[recv fd=4] LIST
local_list
1: test1
2: test2
> [recv fd=4] MOVE 1 2
local_list
1: test2
2: test1
> [recv fd=4] DELETE 1
local_list
1: test1
> [recv fd=4] QUIT

K_programmieren/EZMR/Programieraufgaben/Sockets_Netcode/d$ ./anfrage_
peer 6667
Peer gestartet. Lokaler Listener auf Port 6667
Befehle: connect <ip> <port> | peers | send <idx> <COMMAND> | local_
list | help | exit
> listener: warte auf Verbindungen auf Port 6667...
connect 127.0.0.1 6666
Verbunden zu 127.0.0.1:6666 (fd=5)
> peers
Aktive Peers:
  1) 127.0.0.1:6666 (fd=5)
> send ADD test1
Ungültiger Index
> send 1 ADD test1
OK: Nachricht hinzugefügt.
> send 1 ADD test2
OK: Nachricht hinzugefügt.
> send 1 LIST
1: test1
2: test2
> send 1 MOVE 1 2
OK: Nachricht verschoben.
> send 1 DELETE 1
OK: Nachricht gelöscht.
> send 1 QUIT
OK: Verbindung wird beendet.
Peer fd=5 hat Verbindung beendet
> peers
Aktive Peers:
> []
```

Abb. 2.4: Kommunikation zwischen zwei Peers: Peer 1 (links) lauscht auf Port 6666, Peer 2 (rechts) verbindet sich aktiv und sendet Befehle.

2 Programmieraufgaben

Abbildung 2.4 zeigt den Verlauf der Interaktion: Nachdem Peer 2 eine Verbindung zu Peer 1 aufgebaut hat, werden nacheinander verschiedene Befehle ausgeführt (ADD, LIST, MOVE, DELETE, QUIT). Die Ausgabe des linken Terminals (Peer 1) zeigt die empfangenen Nachrichten und die interne Verarbeitung, während das rechte Terminal (Peer 2) die gesendeten Kommandos und die empfangenen Antworten darstellt.

Der Test zeigt, dass die Kommunikation zwischen beiden Peers bidirektional und unabhängig von der Verbindungsrichtung funktioniert.

Mit dieser Implementierung wurde eine funktionierende Peer-to-Peer-Struktur geschaffen, die Client- und Serverfunktionen in einer einzigen Anwendung vereint. Jeder Peer kann selbstständig Nachrichten austauschen, verwalten und gleichzeitig mehrere Verbindungen handhaben.

3 Praktikum

In diesem Kapitel werden die durchgeführten Praktikumsaufgaben zur mobilen Robotik dokumentiert. Ziel ist es, die praktischen Arbeiten mit ROS 2 nachvollziehbar darzustellen, die Umsetzung eigener Pakete zu beschreiben und die dabei gewonnenen Erkenntnisse zu reflektieren. Die Aufgaben sind in einzelne Abschnitte gegliedert, die jeweils eine kurze Einleitung, die Zielsetzung, die Umsetzung und die Auswertung enthalten.

3.1 ROS 2 Grundlagen und Arbeitsumgebung

Für die Durchführung des Praktikums wurde eine virtuelle Maschine mit Ubuntu 22.04 und der ROS 2-Distribution *Humble* ? verwendet. Die VM wurde speziell für den Einsatz mit ROS 2 vorbereitet, wodurch alle erforderlichen Abhängigkeiten bereits installiert waren.

Um die ROS-Umgebung bei jedem Start eines Terminals automatisch zu initialisieren, wurde folgender Befehl ausgeführt:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

Damit wird das Setup-Skript beim Öffnen eines neuen Terminals automatisch geladen. Dies erspart das manuelle Aktivieren des Arbeitsbereichs und vereinfacht die tägliche Arbeit mit ROS erheblich.
?

Zur Funktionsprüfung wurde die bekannte *turtlesim*-Simulation gestartet entsprechend der Quelle
?:

```
ros2 run turtlesim turtlesim_node  
ros2 run turtlesim turtle_teleop_key
```

Die Simulation ließ sich fehlerfrei ausführen, und sämtliche Tests verliefen wie erwartet. Über die *turtlesim*-Beispielanwendung konnten zudem die grundlegenden Kommunikationsmechanismen von ROS 2 nachvollzogen werden. Dabei kommunizieren einzelne *Nodes* über sogenannte *Topics*, über die beispielsweise Bewegungsbefehle in Form von Geschwindigkeitsnachrichten übertragen werden. Diese Erkenntnisse bilden die Grundlage für die Entwicklung eigener ROS 2-Pakete in den folgenden Arbeitsschritten.

3.2 Erstellung eines eigenen ROS 2-Pakets

In diesem Abschnitt wurde ein eigenes ROS 2-Paket erstellt, um ein Verständnis für die grundlegende Struktur und den Aufbau von ROS 2-Anwendungen zu entwickeln. Als Basis dient die offizielle Dokumentation zu Arbeitsbereichen und Paketen in ROS 2 ???.

3.2.1 Einrichtung des Arbeitsbereichs

Der verwendete Arbeitsbereich (`ros2_ws`) basiert auf der in Abschnitt ?? beschriebenen Umgebung. Der folgende Setup des Arbeitsbereiches basiert sich an dem Tutorial ?. ROS 2 nutzt ein sogenanntes *Underlay*- und *Overlay*-Konzept: Das Underlay umfasst die Basisinstallation (hier die ROS 2-Distribution *Humble*), während im Overlay eigene Pakete ergänzt werden können, ohne die Systeminstallation zu verändern.

Zunächst wurde ein neuer Arbeitsbereich mit zugehörigem `src`-Ordner angelegt:

```
mkdir -p ~/ros2_ws/src
```

Im `src`-Ordner können die neuen Pakete angelegt werden. Wenn ein neues Paket im `src`-Ordner angelegt wurde, kann das Overlay gebaut werden:

```
cd ~/ros2_ws  
colcon build
```

Anschließend sollte in einem neuen Terminal das Overlay gesourct werden:

```
cd ~/ros2_ws  
source install/local_setup.bash
```

3.2.2 Struktur des ROS 2-Pakets

Ein ROS 2-Paket dient der strukturierten Organisation und dem Wiederverwenden von Code. Es enthält für C++ folgende Elemente:

- `CMakeLists.txt` – beschreibt den Build-Prozess (CMake-Konfiguration),
- `package.xml` – enthält Metadaten, Abhängigkeiten und Lizenzinformationen,
- `src/` – Quellcode-Verzeichnis,
- `include/` – Header-Dateien (bei größeren Projekten).

Für die Entwicklung in C++ wird das Buildsystem `ament_cmake` verwendet, und als Build-Tool dient `colcon`.

3.2.3 Erstellung des eigenen Pakets

Das Paket `cpp_turtle_controller` wurde mit folgendem Befehl erzeugt:

```
cd ~/ros2_ws/src  
ros2 pkg create --build-type ament_cmake \  
--license Apache-2.0 cpp_turtle_controller
```

3 Praktikum

Damit wird automatisch die Grundstruktur des Pakets angelegt. In der Datei `package.xml` wurden anschließend die relevanten Abhängigkeiten ergänzt:

```
<depend>roscpp</depend>
<depend>geometry_msgs</depend>
```

sowie eine kurze Beschreibung und der Maintainer eingetragen. Alle geänderten Dateien dieses Paketes und der anderen Pakete liegen im Github Repository zu diesem Projekt.

3.2.4 Implementierung Programm

Im Verzeichnis `src/` des neuen Pakets wurde die Datei `turtle_square.cpp` erstellt. Sie implementiert eine ROS 2-Node, die periodisch `geometry_msgs/msg/Twist`-Nachrichten auf das Topic `/turtle1/cmd_vel` publiziert, um die Turtle in der `turtlesim`-Simulation automatisch ein Quadrat fahren zu lassen. Die Umsetzung und Beschreibung dieses Abschnitts orientiert sich an der Quelle [?](#). Die Steuerung erfolgt über einen zeitbasierten Zyklus:

- Für eine definierte Zeit fährt die Turtle geradeaus.
- Danach erfolgt eine Drehung um ca. 90°.
- Der Zyklus wiederholt sich kontinuierlich.

In Listing 3.1 ist ein Ausschnitt des Programmcodes dargestellt, der den zentralen Steuermechanismus der Node zeigt.

Der gezeigte Abschnitt bildet das Kernstück der Anwendung: Hier wird über eine einfache Zustandslogik (`step_`) zwischen zwei Bewegungsphasen unterschieden, einer Vorwärtsbewegung und einer anschließenden Drehung um etwa 90 Grad. Durch die wiederholte Ausführung dieser Logik entsteht die charakteristische quadratische Bewegung der Turtle in der Simulation.

Der restliche Programmaufbau (Publisher, Timer und Node-Initialisierung) folgt der Standardstruktur einer ROS 2-C++-Node und ist daher nicht im Detail dargestellt.

Listing 3.1: Node zur Steuerung der `turtlesim`-Turtle

```
1     publisher_ = this->create_publisher<geometry_msgs::msg::Twist
      >(
2     "/turtle1/cmd_vel", 10);
3
4     if (step_ < 20) {
5         msg.linear.x = 0.5;    // Vorwärtsbewegung
6         msg.angular.z = 0.0;
7     } else if (step_ < 36) {
8         msg.linear.x = 0.0;
```

3 Praktikum

```
9         msg.angular.z = 1.0;  // Drehung ca. 90 Grad
10     } else {
11         step_ = 0;
12     }
```

3.2.5 Build und Test

Nach der Implementierung wurde das Paket wie bereits allgemein in Abschnitt 3.2.1 beschrieben gebaut:

```
cd ~/ros2_ws
colcon build --packages-select cpp_turtle_controller
```

Anschließend in einem neuen Terminal wurde die turtlesim-Simulation gestartet und in einem separaten Terminal das Overlay mit dem neuen Paket gestartet und der Node ausgeführt.

```
cd ~/ros2_ws
source install/local_setup.bash
ros2 run cpp_turtle_controller turtle_square
```

3.2.6 Ergebnis und Beobachtungen

Das Ergebnis des Tests ist in Abbildung 3.1 dargestellt. Die Turtle bewegte sich wie erwartet entlang einer annähernd quadratischen Bahn. Aufgrund des fehlenden Positions-Feedbacks traten jedoch leichte Abweichungen in den Drehwinkeln auf, wodurch sich die Form des Quadrats über mehrere Zyklen leicht verzerrte.

Insgesamt funktionierte die Steuerlogik zuverlässig, und die Bewegungssequenzen wurden korrekt ausgeführt. Durch die Implementierung eines zusätzlichen *Subscribers*, der Positionsdaten aus der turtlesim-Simulation auswertet, könnten die auftretenden Ungenauigkeiten reduziert werden.

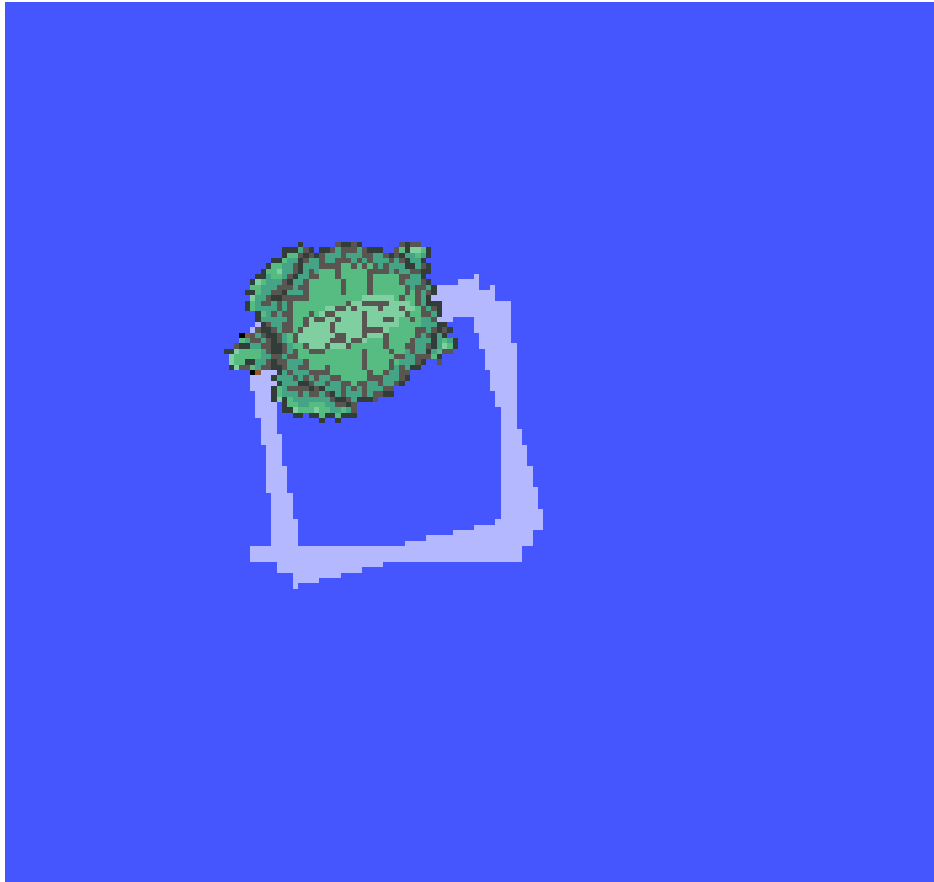


Abb. 3.1: Quadratische Fahrbewegung in der turtlesim-Simulation

3.3 Programmierung eines eigenen Roboters

Ziel dieser Aufgabe war es, den eigenen Roboter so zu programmieren, dass er über das ROS2-Topic `/cmd_vel` Geschwindigkeitsbefehle empfangen und seine Motoren entsprechend ansteuern kann. Dazu wurde ein ESP32-Mikrocontroller verwendet, der mithilfe der Bibliothek *Micro-ROS* direkt mit dem ROS2-Netzwerk kommuniziert. Der ESP32 fungiert dabei als Micro-ROS-Client und steht über eine serielle Verbindung mit dem auf dem Host-PC laufenden Micro-ROS-Agenten in Kontakt.

3.3.1 Aufbau und Funktionsweise

Die Programmierung erfolgte in der Entwicklungsumgebung *PlatformIO* unter Visual Studio Code. Das auf dem ESP32 ausgeführte Programm definiert eine Node mit dem Namen `esp32_cmd_vel_node`, welche zwei zentrale ROS-2-Kommunikationsobjekte beinhaltet:

3 Praktikum

- einen **Subscriber** auf das Topic `/cmd_vel` vom Typ `geometry_msgs/msg/Twist`, über den lineare und angulare Geschwindigkeitskomponenten empfangen werden,
- sowie einen **Publisher** auf das Topic `/debug` vom Typ `std_msgs/msg/String`, der zu Debug-Zwecken Status- und Bewegungsinformationen zurückmeldet.

Im Callback `cmd_vel_callback()` werden die empfangenen Geschwindigkeitsdaten in differenzielle Motorbefehle übersetzt. Hierfür wird eine einfache Differentialsteuerung verwendet, bei der aus der linearen und der Winkelgeschwindigkeit die Drehzahlen der linken und rechten Motoren berechnet werden. Die resultierenden Werte werden über eine UART-Schnittstelle an die Motorsteuerung gesendet. Zusätzlich werden die berechneten Werte als Textnachricht über das Topic `/debug` publiziert, um die Funktionsweise während der Laufzeit überwachen zu können.

Nachfolgend ist ein Ausschnitt des zentralen Steuerteils dargestellt:

Listing 3.2: Ausschnitt des Micro-ROS-Programms auf dem ESP32

```
1      void cmd_vel_callback(const void * msgin) {
2          const geometry_msgs__msg__Twist * received_msg =
3              (const geometry_msgs__msg__Twist *)msgin;
4
5          float linear = received_msg->linear.x;
6          float angular = received_msg->angular.z;
7
8          int16_t left_motor = linear - angular;
9          int16_t right_motor = linear + angular;
10
11         left_motor = constrain(left_motor, -100, 100);
12         right_motor = constrain(right_motor, -100, 100);
13
14         uint8_t left_dir = (left_motor >= 0) ? 1 : 0;
15         uint8_t right_dir = (right_motor >= 0) ? 0 : 1;
16
17         sendMotorCommand(1, 0, left_dir, abs(left_motor));
18         sendMotorCommand(2, 0, right_dir, abs(right_motor));
19     }
```

3.3.2 Durchführung

Nach dem Hochladen des Programms auf den ESP32 wurde der Micro-ROS-Agent auf dem Host-Rechner gestartet und der Mikrocontroller über USB erneut verbunden. Nach erfolgreicher Verbindung meldete sich der ESP32 im ROS-2-Netzwerk als Node an und begann, die über `/cmd_vel` gesendeten Bewegungsbefehle auszuwerten. Zur Überprüfung der Funktionalität konnten über die

3 Praktikum

folgenden Befehle in einem Terminal Testnachrichten gesendet und Rückmeldungen angezeigt werden:

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist \
"{linear: {x: 1.0}, angular: {z: 0.0}}"\part{ ros2 topic pub /cmd_vel geometry_msgs/msg/T
ros2 topic echo /debug
```

Anschließend wurde das zuvor in Abschnitt 3.2 erstellte ROS-Paket, das ursprünglich die *turtlesim*-Schildkröte steuerte, erneut ausgeführt. Damit der reale Roboter anstelle der Simulation auf dieselben Bewegungsbefehle reagieren konnte, musste im Programmcode von `turtle_square` das Topic, auf das die Befehle publiziert werden, von `/turtle1/cmd_vel` auf das allgemeine `/cmd_vel` geändert werden. Nach dieser Anpassung reagierte der Roboter auf die veröffentlichten Bewegungsbefehle des `turtle_square`-Nodes und führte die gleiche Bewegungssequenz aus wie in der Simulation.

3.3.3 Beobachtungen

Der Roboter führte die Bewegungssequenz grundsätzlich wie erwartet aus: gradeaus fahren, anschließend drehen und erneut gradeaus fahren. Auffällig war jedoch, dass die Drehungen kleiner ausfielen als in der Simulation. Anstelle 90° Drehungen, wie bei einem Quadrat, waren sie nur halb so groß, wodurch die Trajektorie eher einem unregelmäßigen Polygon entsprach.

Zeitweise zeigte der Roboter kurzzeitiges Zittern oder Oszillationen, was auf minimale Kommunikationsverzögerungen oder Jitter zwischen Agent und ESP32 hinweist. Trotz dieser Störungen blieb der Roboter weitgehend stabil und driftete nicht signifikant vom Ausgangsbereich ab.

3.3.4 Diskussion

Die Abweichungen zwischen der Simulation und dem realen Verhalten lassen sich durch mehrere Faktoren erklären:

- **Open-Loop-Steuerung:** Das Programm nutzt kein Sensorfeedback, sodass keine Korrektur von Abweichungen erfolgen kann.
- **Physikalische Unterschiede:** Reibung, Trägheit und Motordynamik wirken in der Realität, aber nicht in der idealisierten Simulation.
- **Quantisierung:** Durch die ganzzahlige Verarbeitung (`uint8_t`) werden kleine Geschwindigkeitsänderungen gerundet.

3.3.5 Fazit

Der Versuch zeigte, dass die Integration eines ESP32 mit Micro-ROS erfolgreich funktionierte und der Roboter über ROS2-Topics gesteuert werden konnte. Gleichzeitig verdeutlichte er die Grenzen

3 Praktikum

einer offenen Steuerung ohne Rückmeldung: kleine Abweichungen summieren sich über die Zeit und führen zu merkbaren Differenzen zwischen Soll- und Istbewegung.

Eine mögliche Verfeinerung dieses Programms bestünde darin, Sensordaten (z.B. IMU) rückzuführen, um die ausgeführten Bewegungen präziser steuern zu können. Auf diese Weise ließe sich die geplante Quadratbahn exakter abfahren. Im Rahmen dieser Aufgabe wurde jedoch bewusst auf Sensorfeedback verzichtet, um den Fokus auf die grundlegende Micro-ROS-Kommunikation und die Umsetzung der offenen Steuerung zu legen.

3.4 Durchführung der Aufgaben vor Ort

In diesem Abschnitt wird die Durchführung der praktischen Aufgaben beschrieben. Hierfür wurde die bereitgestellte Rechner mit einer Ubuntu 22.04 verwendet. Für die Bearbeitung wurde als Referenz die Anleitung ?? Pfirsich verwendet und wird hier folglich nicht mehr erwähnt.

3.4.1 Aufgabe 1

Ziel der Aufgabe ist es, eine Verbindung mit dem Turtlebot herzustellen und diesen manuell zu bewegen. Die Verbindung zwischen Rechner und Turtlebot wurde dabei automatisch hergestellt. Um die manuelle Bewegung zu ermöglichen, wurde zuerst der Befehl

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

ausgeführt. Dadurch öffnet sich im Terminal eine Auflistung an einfachen Eingaben, durch welche der Turtlebot gesteuert werden kann.

3.4.2 Aufgabe 2

Als zweiter Teil des Praktikums sollen mit dem Turtlebot visuelle Daten eingelesen und ausgegeben werden. Dafür wird das Programm RViz verwendet. Nach der Ausführung von

```
ros2 launch turtlebot4_viz view_robot.launch.py
```

wird das Programm gestartet und es wird ausgehend vom Sichtfeld des Roboters bereits eine Karte aufgezeichnet.

Zur Auslesung einzelner Werte wurde der Befehl

```
ros2 topic list
```

verwendet. Durch das Hinzufügen weiterer Argumente können hier Werte wie der Batteriestand oder der Ladezustand ausgegeben werden.

3.4.3 Aufgabe 3

Hier soll eine gesamte Karte aufgezeichnet und abgespeichert werden. Dafür wird der Roboter nach Aufgabe 2 manuell weiterbewegt, bis er alle Hindernisse erkannt hat. Das kann visuell mit RViz überprüft werden. Dabei kam es beim ersten Versuch dazu, dass eine nicht existierende Wand eingezeichnet wurde. Diese entstand vermutlich durch den Blindspot hinter dem Roboter, während sich dieser noch in der Startposition auf der Ladestation befand.

Das Abspeichern der Karte erfolgt daraufhin mit

```
ros2 service call /slam_toolbox/save_map slam_toolbox/srv/SaveMap
"name: data: 'map_name'"
```

3.4.4 Aufgabe 4

Um eine autonome Bewegung zu ermöglichen, wurden folgende Befehle durchgeführt:

```
ros2 launch turtlebot4_navigation localization.launch.py map:=office.yaml
ros2 launch turtlebot4_navigation nav2.launch.py
ros2 launch turtlebot4_viz view_robot.launch.py
```

Hiermit wird die vorher abgespeicherte Map in RViz geöffnet und eine Bewegung ermöglicht. Dafür wird mit den Funktionen 2D Point Estimate eine geschätzte Startposition angegeben und durch Nav2 Goal ein Ziel festgelegt. Des weiteren muss noch der Befehl

...

ausgeführt werden, wodurch die Lokalisierung des Roboters verbessert wird. Nachdem das Ziel nun gesetzt wird, beginnt der Roboter sich in die Richtung des Ziels zu bewegen und dabei Hindernisse zu umfahren. Bei unserem Durchlauf musste er dabei erst in die entgegengesetzte Richtung fahren, bevor er das Ziel erreichen konnte. Beim Erreichen des Ziels hat er sich noch zwei mal um sich selbst gedreht.

3.4.5 Aufgabe 5