HTWK
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 21. November 2025

Inhaltsverzeichnis

Abbildungsverzeichnis

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.

- **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.

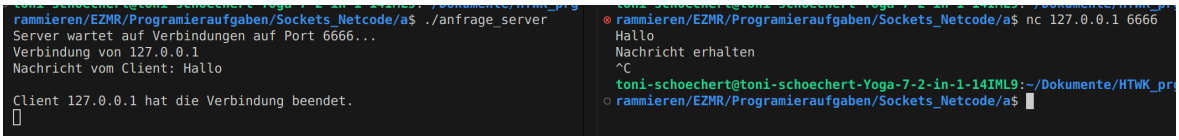

Abb. 2.1: Testlauf des Servers mit `netcat`: Verbindung, Nachrichtenaustausch und Verbindungsende

Dieser Test diente 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.

2.1.2 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.

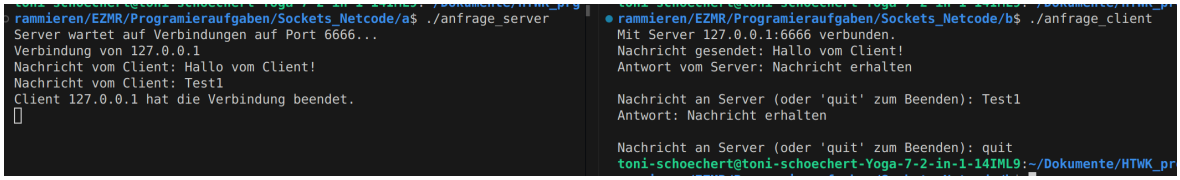

Abb. 2.2: Interaktive Kommunikation zwischen Client und Server über TCP

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-

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.

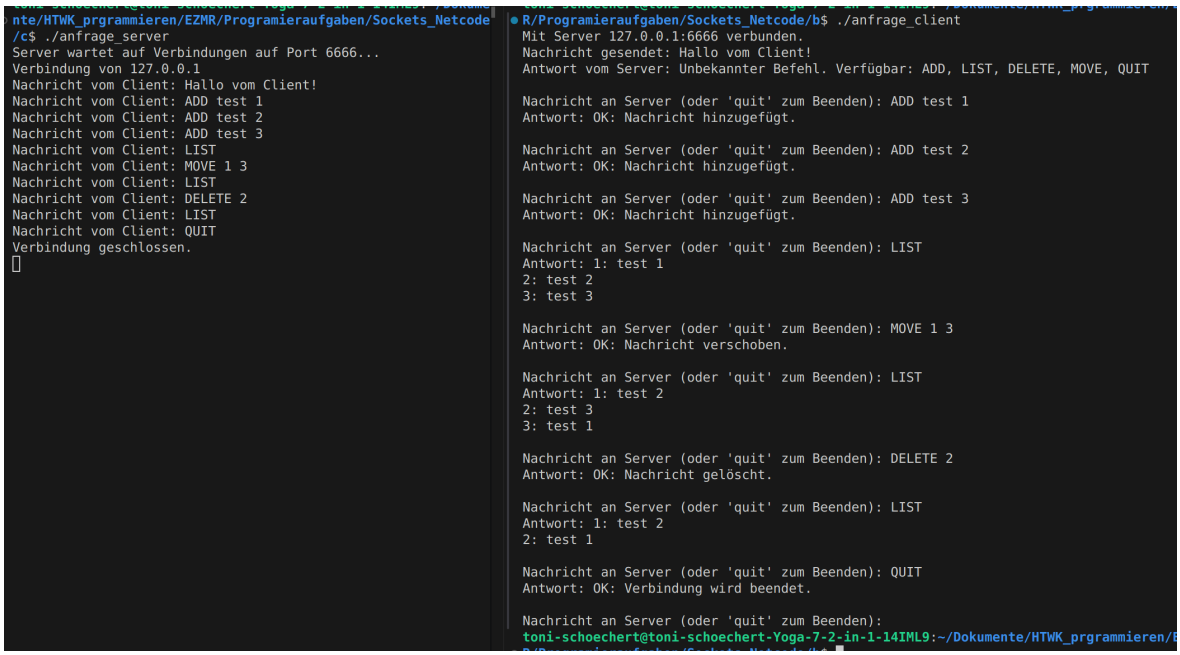

Abb. 2.3: Kommunikation zwischen Server (links) und Client (rechts) – Test der Befehle `ADD`, `LIST`, `DELETE`, `MOVE` und `QUIT`.

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.

- **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 angepassten 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.

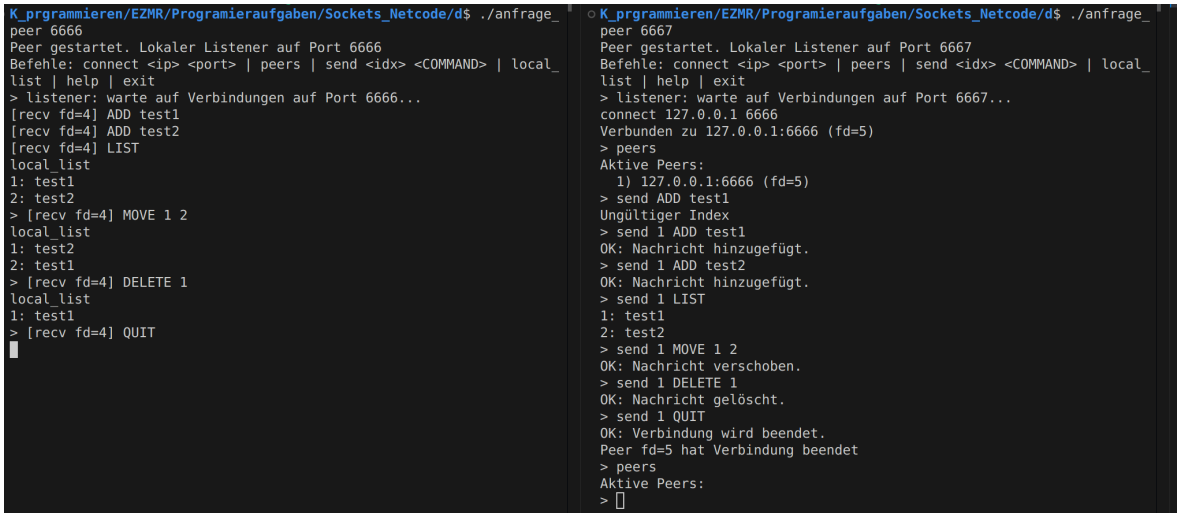

Abb. 2.4: Kommunikation zwischen zwei Peers: Peer 1 (links) lauscht auf Port 6666, Peer 2 (rechts) verbindet sich aktiv und sendet Befehle.

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.

2.2 IPC

In dieser Programmieraufgabe soll grundlegend die FUnktionsweise von **Interprocess Comunication** praktisch nachvollzogen werden. Die in den Teilschritten erstellten Programme sollen dabei nach dem Prinzip der Client-Server-Architektur und der Peer-to-Peer-Verbindung miteinander kommunizieren. Dabei soll die Kommunikation anders als in Sektion 2.1 zwischen Prozessen auf dem selben System stattfinden.
Als IPC-Varianten werden hier sowohl UNIX-Domain-Sockets sowie Shared Memory und Semaphore verwendet. Im Rahmen der Aufgaben wurden mehrere Programme erstellt, welche aufeinanander aufbauen und somit eine vollständige Kommunikationsstruktur ermöglichen.

- **Teilaufgabe a)**:
  - Erstellen einer IPC-Infrastruktur zur Entgegennahme von Anfragen.
  - Das Programm kann ggf. auf Anfragen antworten und tut dies auch. (Protokoll/PDUs)
- **Teilaufgabe b)**:
  - Eine oder mehrere Anwendungen sind zu erzeugen, welche Kontakt mit a) aufnehmen und Daten senden oder empfangen können. (Tests: ipcs ipcrm)
- **Teilaufgabe c)**:
  - Anwendung a) stellt eine Liste zur Verfügung, welche von b)-Anwendungen abgefragt und manipuliert werden kann.
  - Anwendungen können anhand dieser Liste auf Nachrichtenverläufe zugreifen (lesen/schreiben).
- **Teilaufgabe d)**:
  - Es soll keine Unterscheidung mehr zwischen a)- und b)-Anwendungen existieren.

2.2.1 Teilaufgabe a) - Implementierung des Servers

Hier wurde ein einfacher IPC-Server erstellt. Dieser wartet auf eingehende Nachrichten von Clients und antwortet darauf mit einer Bestätigung und Zustandsinformation.

Der Server erstellt hier als erstes mit `socket()` einen UNIX-DOMAIN-Socket, welcher mit `bind()` an einen Pfad im Dateisystem gebunden wird. in diesem Beispiel wird der Pfad `/tmp/ipc_example.sock` verwendet. Danach geht der Server in den `"Listen-Zustand"` und kann somit eingehende Verbindungen akzeptieren.

Ein Client, welcher sich verbinden möchte, erstellt ebenfalls einen Socket mit gleicher Addresse um mittels connect() den Server zu erreichen. Sobald eine Verbindung aufgebaut wurde, erzeugt der Kernel für diese Verbindung einen neuen Socket, den der Server über accept() erhält. Der Server erstellt dann über fork() einen Child-Process, der spezifisch für diese Client-Verbindung zuständig ist. Somit bleibt der Listening-Socket frei und der Server kann mehrere Clients bedienen.

In der Funktion `handle_client()` wird dabei der Umgang mit dem jeweiligen Client festgelegt. Diese liest zuerst ein 4-Byte-Längenfeld, danach die gesendete Nachricht und erzeugt eine Antwort. Diese wird ebenfalls mit einem Längenfeld versehen, sodass die Nachrichtenformate konstant bleiben. Nach Abschluss der Kommunikation wird der Child-Process beendet und die Verbindung wird geschlossen.

Der Server erkennt Empfangs-, Sende- und Allokationsfehler und beendet damit die Verarbeitungsschleife des Clients. Somit können durch Client-seitiges fehlerhaftes Verhalten entstandene Probleme verhindert werden. Der für jeden Client angelegte dynamische Speicher wird mit free() freigegeben und verbindungsspezifische Sockets werden geschlossen. Durch unlink() wird die UNIX-Socket-Datei beim Beenden des Servers gelöscht, sodass beim Neustart keine veralteten Socket-Dateien Konflikte verursachen.

2.2.2 Teilaufgabe b) - Implementierung des Clients

In dieser Teilaufgabe wurde eine Client-Anwendung entwickelt, welche sich mit dem in Teilaufgabe a) entwickelten Server verbindet und Daten mit diesem austauscht. Ziel ist es, die Funktionalität sowohl des neuen Clients als auch der bereits vorhandenen Server-Anwendung zu testen.

Der Client erstellt zunächst mit dem Aufruf `socket()` einen UNIX-Domain-Socket. Die Kommunikation erfolgt über den Pfad `/tmp/ipc_example.sock` , unter dem der Server erreichbar ist. Anschließend wird über `connect()` eine Verbindung zum Server hergestellt. Die Abläufe des Servers

werden in Sektion 2.2.1 beschrieben.

Für eine benutzerfreundliche Bedienung verwendet der Client eine interaktive Eingabe-Schleife. Der Nutzer kann mehrere Nachrichten hintereinander senden, ohne den Client neu starten zu müssen. ede eingegebene Nachricht wird auf ihre Länge überprüft, die Länge in Network Byte Order (`uint32_t`) umgewandelt und zusammen mit dem Payload an den Server gesendet. Der Server antwortet im gleichen Format: zuerst eine 4-Byte-Länge, dann die Payload. Die Antwort wird empfangen und dem Nutzer ausgegeben.
Der Client behandelt auch das Beenden der Verbindung: Wird eine leere Eingabe oder der Befehl `"quit"` eingegeben, sendet der Client eine Null-Länge-PDU an den Server, was als Signal zum Schließen der Verbindung interpretiert wird. Danach wird der Socket ordnungsgemäß geschlossen, und der Client beendet sich.

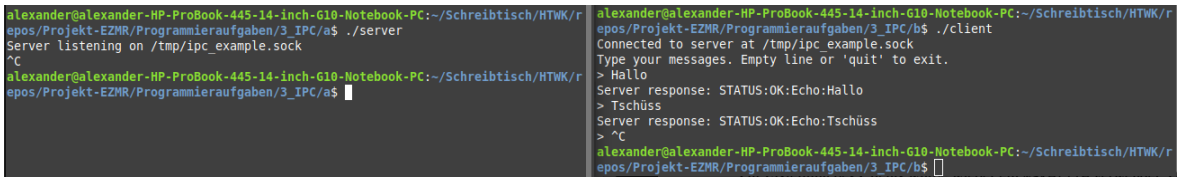

Abb. 2.5: Interaktion zwischen Server und Client bei Verwendung von IPC über UNIX-Domain-Sockets - Aufgabe 3 b)

In Abbildung 2.5 wird der Testlauf der Kommunikation von Server und Client dargestellt. Hier ist zu erkennen, dass eine eingehende Verbindung mit einem Client nicht beim Server angezeigt wird. Dagegen erhält der Client bei einer Nachricht eine Antwort bestehend aus dem Zustand des Servers sowie eine Wiederholung der Nachricht.

2.2.3 Teilaufgabe c) - Nachrichtenverwaltung im Server

In dieser Teilaufgabe werden der Server aus Teilaufgabe a) und der Client aus Teilaufgabe b) erweitert. Zusätzlich zur bereits genutzten Kommunikation über UNIX-Domain-Sockets kommen nun Shared Memory und Semaphoren zum Einsatz. Damit kann der Server einen globalen, von allen Kindprozessen gemeinsam nutzbaren Speicherbereich verwalten. Der Server stellt hierfür eine Liste von Textnachrichten bereit, die von allen Clients abgerufen und manipuliert werden kann. Zur Interaktion unterstützt der Server folgende Befehle:

- `LIST` – gibt alle aktuell gespeicherten Einträge aus
- `ADD:<text>` – fügt der Liste einen neuen Eintrag hinzu
- `CLEAR` – löscht alle Einträge
- `quit` – beendet den Client und schließt die Verbindung

Damit entsteht eine einfache textbasierte Steuerung, bei der jeder Befehl eine konkrete serverseitige Operation auslöst. Auf Client-Seite wurde hierzu die Funktion `send_cmd()` ergänzt, welche Befehle im geforderten PDU-Format versendet, die Antwort des Servers empfängt und anschließend ausgibt. Der Befehl `quit` beendet ausschließlich den Client; er wird nicht an den Server weitergeleitet.

Im Vergleich zu Abschnitt 2.2.1 wurde der Server deutlich erweitert. Es wird nun ein Shared-Memory-Segment initialisiert, das eine gemeinsame Datenstruktur zur Nachrichtenverwaltung enthält. Mithilfe eines binären Semaphors wird der Zugriff aller parallelen Server-Kindprozesse synchronisiert, um Race Conditions zu vermeiden. Die Funktion `process_command()` übernimmt die Auswertung der empfangenen Befehle und führt die zugehörigen Operationen auf der globalen Nachrichtenliste aus.

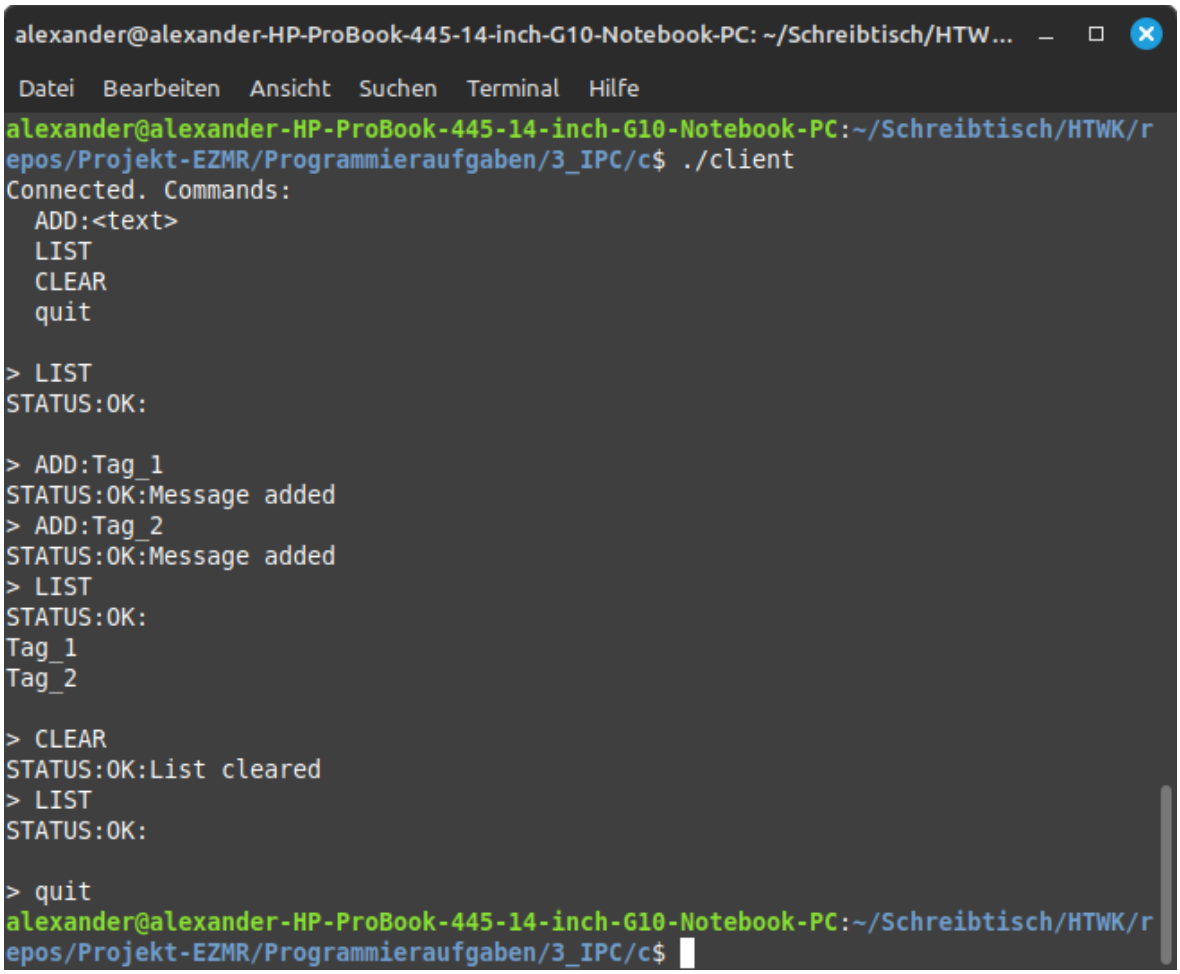

Abb. 2.6: Textbasierte Steuerung über den Client- Aufgabe 3 c)

Die Abbildung 2.6 zeigt eine beispielhafte Kommunikation zwischen Client und Server. Hier werden durch ADD neue Einträge zur Liste hinzugeführt und wieder entfernt. Da sich die Ausgabe vom Server

nicht zur Abbildung 2.5 änder, wird dieser hier nicht noch einmal gezeigt.

2.2.4 Teilaufgabe d) - Vereinheitlichung von CLient und Server

In dieser Teilaufgabe wurde die anfängliche Trennung zwischen Client- und Serveranwendung aufgehoben. Anstelle zweier separater Programme wurde hier eine kombinierte Anwendung erstellt, die innerhalb eines einzelnen Prozesses sowohl die Server- als auch die Clientfunktion übernimmt.

Das Programm `peer.c` baut dabei auf der Funktionalität von Sektion 2.2.3 auf und kombiniert dabei die Inhalte von `server.c` und `client.c`. Die größte funktionale Änderung liegt hierbei, dass beim Start des Programms kein `shm->count = 0;` ausgeführt wird. Dadurch bleiben Einträge aus vorherigen Programmläufen im Shared Memory bestehen, während `server.c` bei jedem Start mit einer leeren Liste begonnen hat.
Die übrigen Funktionen entsprechen den vorherigen Aufgaben: Serverseitig werden erneut Shared Memory und Semaphoren verwendet und die Funktion `process_command()` übernimmt die Auswertung eingehender Kommandos. Der Server läuft in einem eigenen Thread `server_ thread` , der einen Unix-Domain-Socket erzeugt und im lokalen Dateisystem bereitstellt.
`"Clientseitig"` wird wieder die Funktion `send_cmd()` verwendet, um Befehle an den lokalen Server zu schicken. Dabei wird für jedes Kommando eine eigene Verbindung zum Socket aufgebaut.

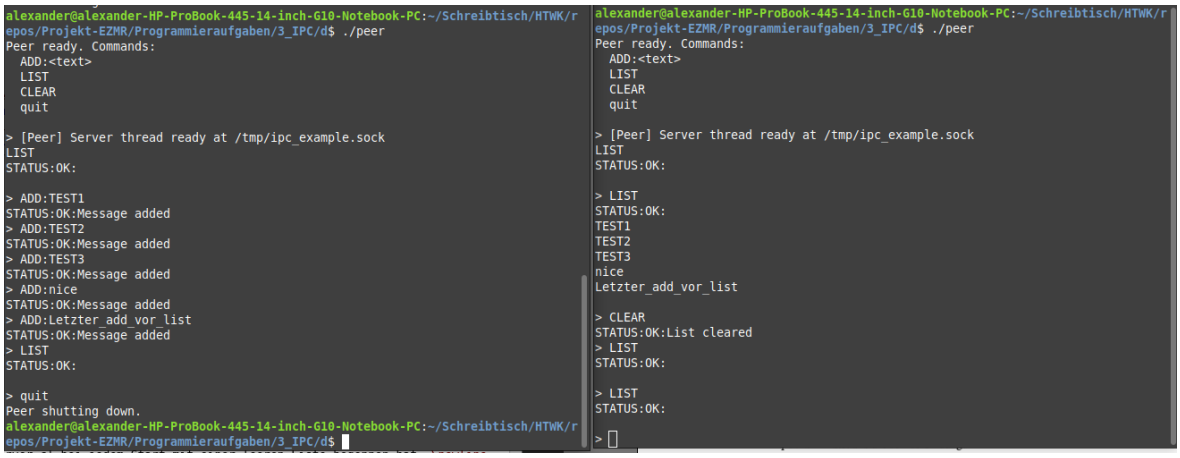

Abb. 2.7: Testversuch mit peer.c - Aufgabe 3 d)

In Abbildung 2.7 wird eine beispielhafte Kommunikation zwischen zwei Ausführungen des Programms `peer.c`. Hierbei ist zu sehen, dass beide Ausführungen auf die Liste zugreifen und diese bearbeiten können und dabei unabhängig voneinander weiter laufen können.

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
```

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

```
<depend>rclcpp</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;
```

```
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.

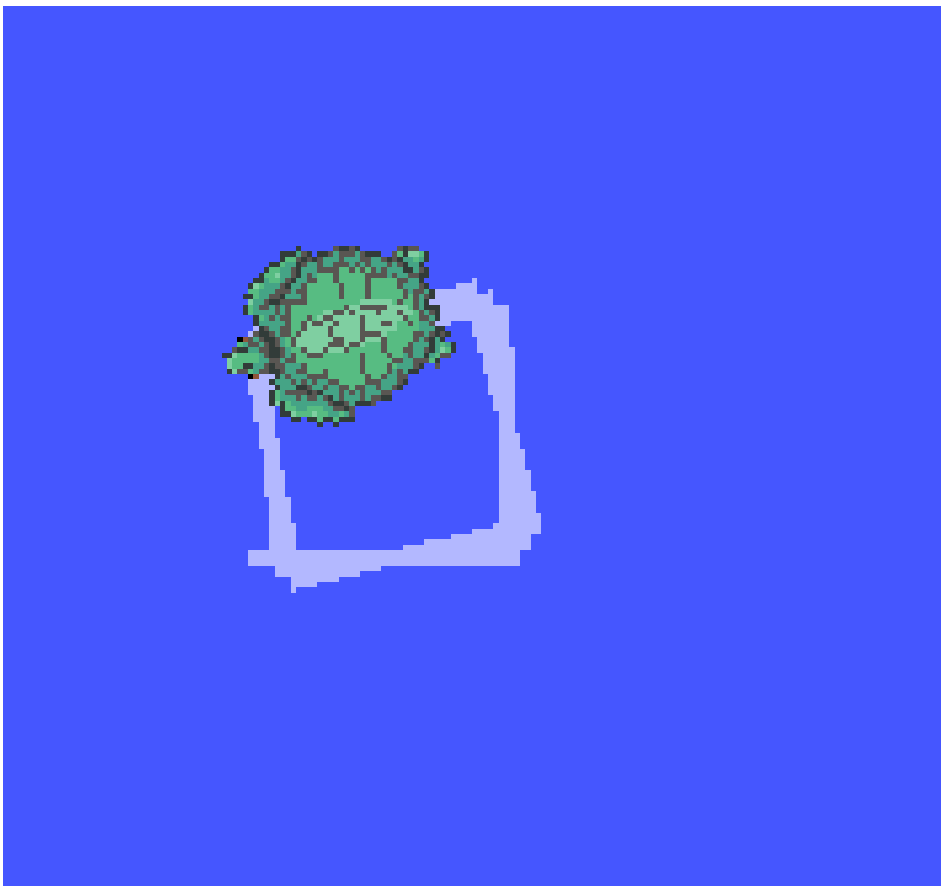

Abb. 3.1: Quadratische Fahrbewegung in der `turtlesim`-Simulation

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:

- 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

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: geradeaus fahren, anschließend drehen und erneut geradeaus 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 und Fazit

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.

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

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 Erweiterung durch Sensorik und eigenes Projekt

Nachdem die Grundbewegung des Roboters funktionierte, sollte im Rahmen der Aufgabe 2.4 mindestens ein Sensor des Roboters in ein eigenes Projekt integriert werden. Ziel war es, den Sensor mithilfe des ESP32 auszulesen, die Messwerte in das ROS 2-Netzwerk zu publizieren und auf dem Host-PC weiterzuverarbeiten.
In diesem Projekt werden die beiden *Bumper-Sensoren* des Roboters sowie der *Buzzer* genutzt. Die Bumper dienen als einfache kollisionsbasierte Hinderniserkennung und lösen abhängig von der Fahrtrichtung links oder rechts ein eigenes Verhalten aus. Der Buzzer wird dabei genutzt, um akustische Rückmeldungen zu erzeugen.
Der vollständige Quellcode für ESP32 (micro-ROS) und den ROS-2-Node befindet sich im zugehörigen GitHub-Repository.

3.4.1 Aufbau des Systems

Der Roboter verfügt über zwei vorne montierte Bumper-Schalter (links und rechts), welche im ESP32 als digitale Eingänge mit `INPUT_PULLUP` konfiguriert wurden. Ein gedrückter Bumper erzeugt ein `LOW`-Signal. Beide Sensoren werden nicht per Interrupt, sondern über einen regelmäßig laufenden micro-ROS-Timer (50 ms) abgefragt.
Zusätzlich wurde der vorhandene Buzzer an einem PWM-fähigen Pin angebunden. Dieser kann über das Topic `/buzzer` mit einer Frequenz angesteuert werden, sodass der Roboter akustisch auf erkannte Hindernisse reagiert.
Auf dem Host-PC wurde ein eigener ROS-2-Node implementiert, der die Bumper-Signale auswertet und daraufhin Bewegungsbefehle über `/cmd_vel` an den Roboter sendet. Somit entsteht eine einfache Hindernisvermeidung.

3.4.2 Funktionsweise des ESP32-Programms

Der ESP32 übernimmt drei zentrale Aufgaben:

1. **Auslesen der Bumper-Sensoren**

- Polling im 50 ms-Takt
- Software-Debouncing zur Unterdrückung von Prellen
- Veröffentlichung von `/bumper/left` und `/bumper/right` als `std_msgs/Bool`

2. **Ansteuerung der Motoren über `/cmd_vel`**
   - differentialbasierte Umsetzung der Twist-Nachricht
   - Begrenzung und Umrechnung der Werte auf das serielle Motorprotokoll
   - Debug-Ausgabe über `/debug`
3. **Buzzer-Steuerung über `/buzzer`**
   - PWM-Signal mit frei wählbarer Frequenz
   - automatisches Abschalten nach Ablauf der Dauer

Durch die Kombination dieser Funktionen wird der ESP32 zu einem reinen micro-ROS-Sensorknoten, während die höhere Logik auf dem PC ausgeführt wird.

3.4.3 Funktionsweise des ROS2-Nodes auf dem Host-PC

Der ROS-2-Node `bumper_react` übernimmt die komplette Entscheidungslogik. Sein Verhalten lässt sich wie folgt zusammenfassen:

- Standardzustand: **geradeaus fahren**
- Wenn der linke Bumper ausgelöst wird:
  - ein höherer Ton wird über `/buzzer` gesendet,
  - der Roboter fährt kurz rückwärts,
  - anschließend dreht er nach rechts,
  - danach setzt er die Vorwärtsfahrt fort.
- Wenn der rechte Bumper ausgelöst wird:
  - ein tiefer Ton wird ausgelöst,
  - kurze Rückwärtsfahrt,
  - Drehung nach links,
  - anschließend Weiterfahrt.
- Zu beginn der Rückwärtsfahrt wird der Buzzer zusätzlich mehrfach aktiviert, um das Manöver akustisch zu signalisieren.

Die Steuerung erfolgt über einen Zustandsautomaten mit den Zuständen `FORWARD`, `REVERSE`, `STOP` und `TURN`. Ein 200 ms-Timer führt die Bewegungslogik aus, wobei mehrfach hintereinander Bewegungsbefehle mit einer Schleife gesendet werden, um Übertragungsverluste zu kompensieren.
Beim Beenden des Programms wird automatisch ein Stopp-Befehl gesendet.

3.4.4 Beobachtungen und Herausforderungen

Während der praktischen Versuche zeigte sich Folgendes:

- Die Hinderniserkennung über die Bumper funktionierte zuverlässig.
- Der Debounce-Mechanismus war notwendig, da die Bumper ohne Entprellung gelegentlich mehrfach auslösten.
- Die Bewegungsabläufe (*rückwärts, drehen, vorwärts*) funktionierten im Normalfall korrekt.
- Es kam vereinzelt vor, dass der Roboter von den vorgesehenen Bewegungen abwich. Dies hängt vermutlich mit der Mikro-ROS-Übertragung zusammen:
  - einzelne `/cmd_vel`-Nachrichten gingen verloren,
  - Motorbefehle wurden manchmal verzögert verarbeitet.
- Durch die Mehrfachübermittlung von Bewegungsbefehlen innerhalb der Zustandsmaschine konnten diese Fehler deutlich reduziert werden.

3.4.5 Fazit

Das Projekt zeigt, wie sich einfache Sensordaten (Bumper) nutzen lassen, um ein autonomes Verhalten zu erzeugen. Die Trennung zwischen ESP32 als micro-ROS-Sensor-/Aktorknoten und PC-seitiger Logik hat sich klar bewährt. Gleichzeitig verdeutlichen die Versuche, dass ohne Rückkanäle oder Positionssensorik kleine Übertragungs- oder Timingfehler dazu führen können, dass die Trajektorie des Roboters nicht vollständig reproduzierbar ist.
Das vorgestellte System bietet jedoch eine robuste und erweiterbare Grundlage, um weitere Sensordaten oder komplexere Reaktionen zu integrieren.

3.5 Arbeiten mit dem TurtleBot 4

In diesem Abschnitt wird die praktische Arbeit mit dem professionellen mobilen Roboter *TurtleBot 4* dokumentiert. Der TurtleBot 4 basiert auf ROS,2 und stellt zahlreiche Sensoren, Aktoren sowie Navigations- und SLAM-Funktionalitäten bereit. Ziel der Aufgabe war es, die grundlegenden Funktionsweisen des Roboters kennenzulernen, die SLAM- und Navigationspipeline praktisch zu erproben

und anschließend ein eigenes kleines ROS-2-Programm auf den TurtleBot anzupassen. Wir orientieren uns in diesem Abschnitt an dem Benutzerhandbuch zum Turtlebot4 **?**

3.5.1 Aufgabe 1

Ziel dieser 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 über das Topic cmd_vel gesteuert werden kann.

3.5.2 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.

3.5.3 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, diese Karte ist in Abbildung 3.2 zu sehen.
Dabei kam es beim ersten Versuch dazu, dass eine nicht existierende Wand eingezeichnet wurde. Diese enstand 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'"
```

3.5.4 Aufgabe 4

Um die autonome Navigation des TurtleBot 4 zu ermöglichen, wurden zunächst die notwendigen ROS 2 Launch-Dateien gestartet:

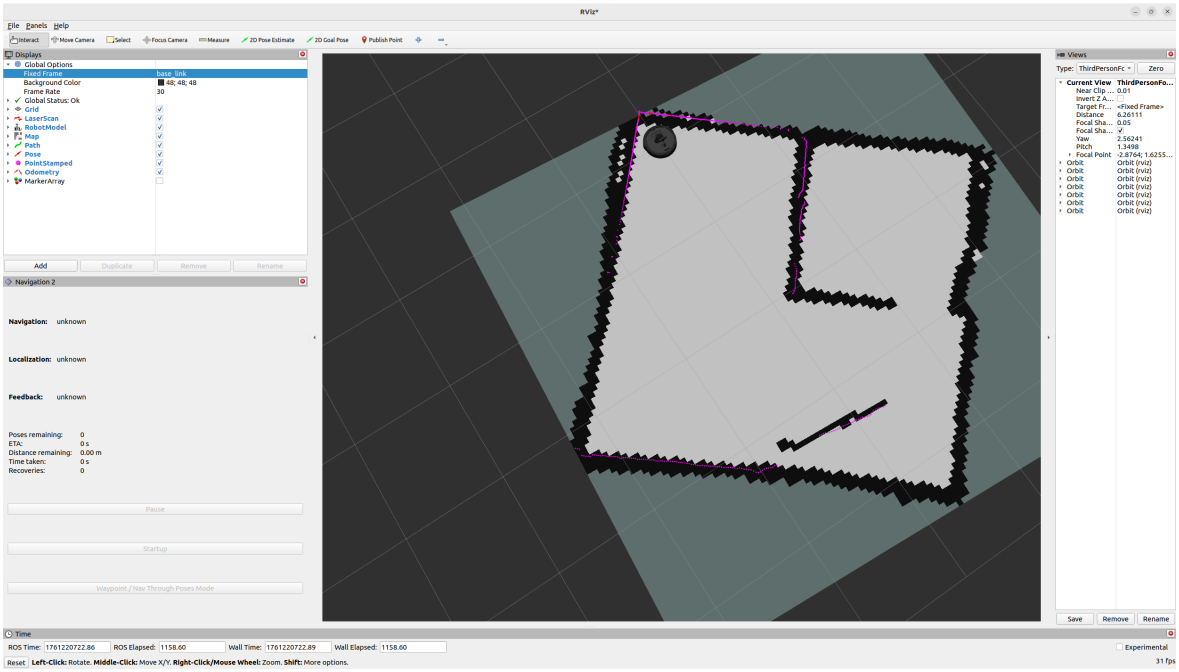

Abb. 3.2: Mit Rviz erzeugte Karte

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

Damit wird die zuvor aufgezeichnete Karte in RViz geladen und gleichzeitig der Nav2-Stack inklusive Lokalisierung aktiviert. Anschließend wurde mit der RViz-Funktion *2D Pose Estimate* die ungefähre Startposition des Roboters angegeben. Über die Funktion *Nav2 Goal* wurde daraufhin ein Zielpunkt in der Karte gesetzt.

Nach dem Setzen des Ziels beginnt der Roboter automatisch mit der Pfadplanung und Navigation in Richtung des angegebenen Punktes. Dabei wich er Hindernissen selbstständig aus. In unserem Versuch musste der Roboter zunächst ein Stück in die entgegengesetzte Richtung fahren, bevor er die optimale Route zum Ziel nahm. Beim Erreichen des Ziels drehte er sich zweimal um die eigene Achse, was typisch für den Nav2-Controller ist, da der Roboter am Endpunkt orientiert wird.

Während der Fahrt kam es mehrfach zu kurzen Stopps. Diese entstehen, wenn der Roboter seine Sensordaten mit der Karte abgleicht und seine Trajektorie aktualisiert. Ein Beispiel der Navigation in RViz mit der geplanten Route ist in Abbildung 3.3 dargestellt.

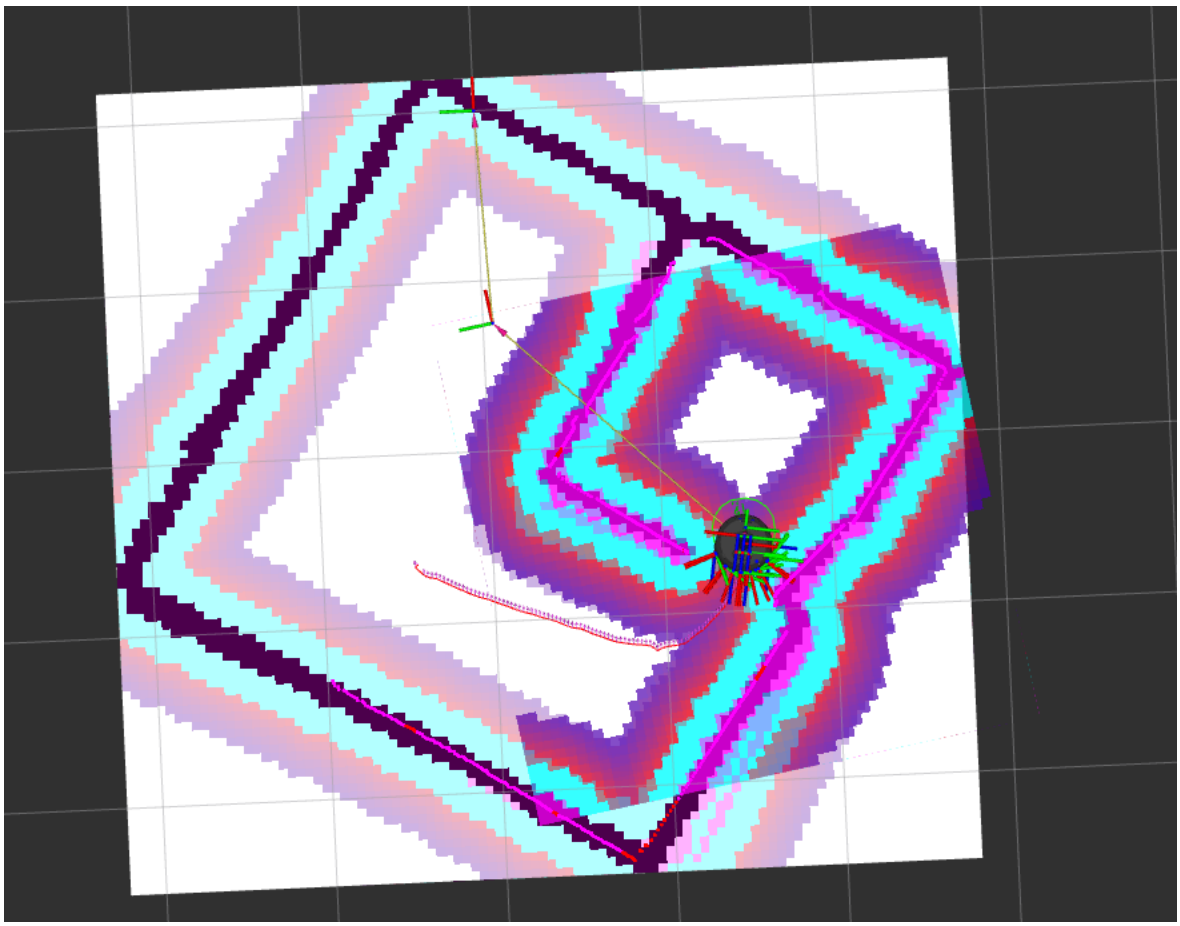

Abb. 3.3: Navigation auf der Karte in RViz

3.5.5 Aufgabe 5

Als letzte Aufgabe sollte eine eigene Anwendung umgesetzt werden. Hierzu wurde das zuvor für den selbstgebauten Roboter entwickelte ROS-2-Programm übernommen und an die Struktur des TurtleBot angepasst.
Anstelle der einfachen Bumper-Signale aus der ESP32-Übung wurde für diese Aufgabe das TurtleBot4 eigene Topic `/hazard_detection` verwendet. Dieses Topic publiziert Nachrichten des Typs `HazardDetectionVector`, welcher eine Liste verschiedener Ereignisse enthält (u.a. Bumper, Cliff-Sensoren, Wheel Drop). Für das hier umgesetzte Verhalten wurden ausschließlich die Bumper-Ereignisse ausgewertet.
Jedes Element der detections-Liste besitzt ein Typfeld sowie einen zugehörigen Frame-Namen (z.B. `base_left_bumper` oder `base_right_bumper`). Die Bumper-Erkennung erfolgt, indem alle enthaltenen Ereignisse iteriert und diejenigen ausgewählt werden, deren Typ dem Bumper-Sensor entspricht. Anhand des Frame-Namens wird zwischen linkem und rechtem Bumper unterschieden. Da-

durch konnte das zuvor verwendete einfache ESP32-Signal durch die integrierten TurtleBot-Sensoren ersetzt werden.
Auch die Ansteuerung des Buzzers musste gegenüber den vorherigen Aufgaben angepasst werden. Der TurtleBot4 akzeptiert keine einfachen numerischen Befehle, sondern erwartet eine Nachricht des Typs `AudioNoteVector`. Dieses Format besteht aus einer Liste von `AudioNote`-Einträgen, wobei jeder Eintrag eine Frequenz sowie eine maximale Abspieldauer enthält.
Für die Umsetzung wurde beim Erkennen eines Bumper-Events programmgesteuert ein entsprechender Ton erzeugt und als `AudioNoteVector` publiziert. Dabei wurde ein wie zuvor ein unterschiedlicher Ton für linken und rechten Bumper verwendet. Zusätzlich erzeugt der Roboter während des automatischen Rückwärtsfahrens mehrere kurze Signaltöne, um das Manöver akustisch kenntlich zu machen.
Der angepasste Code ist vollständig im GitHub-Repository dokumentiert. Das grundlegende Verhalten:

- Standard: Roboter fährt langsam geradeaus.
- Linker Bumper: kurzer Rückwärtsgang, Drehung nach rechts, hoher Ton.
- Rechter Bumper: kurzer Rückwärtsgang, Drehung nach links, tiefer Ton.
- Beim Rückwärtsfahren zu beginn: periodischer Warnton.

Die Fahrdistanzen und Drehungen wurden verkürzt.
Die Anpassung funktionierte gut. Der TurtleBot führte das Ereignis-basierte Verhalten zuverlässig aus:

- Die Hazard-Detection erkannte Kollisionen korrekt.
- Buzzer-Signale wurden wie vorgesehen ausgeführt.
- Rückwärtsfahrt und Drehmanöver liefen stabil.
- Kurze Stopps oder minimale Verzögerungen entstanden durch Nachrichtenschwankungen, hatten jedoch keinen Einfluss auf das Gesamtverhalten.

Insgesamt zeigte sich, dass ein zuvor entwickeltes ROS-2-Programm mit wenigen Änderungen erfolgreich auf einen professionellen Roboter portiert werden kann, sofern Topics und Nachrichtenformate entsprechend angepasst werden.