Hochschule Bochum

SolarSwarm Networks & SolarSwarm Visualizer

Technische Dokumentation

von

Christopher Blümer (018384657)

Nils Steingräber (018395884)

Robin Brack (018101266)

Vincent Welsch (018390735)

Betreuer: Denny Falls-Rodriguez

Prüfer: Prof. Dr.-Ing Wolf Ritschel

Abgabedatum:06.02.2026

Inhaltsverzeichnis

1 Einleitung

1.1 Kurzfassung

SolarSwarm ist ein Projekt des Instituts für Elektromobilität an der Hochschule Bochum, an dem Teams aus Studierenden aus verschiedenen Fachbereichen über Projektarbeiten gemeinsam daran arbeiten, einen autonomen, mobilen Roboterschwarm für umweltfördernde Aufgaben wie das Sammeln von Müll, Pflanzen von Bäumen oder Pflegen von Grünflächen zu entwickeln. Die technische Umsetzung und die Analyse hinsichtlich der ökonomischen Auswirkungen dieses Projekts sollen durch die interdisziplinäre Zusammenarbeit gelingen. Im Rahmen einer Projektarbeit für das Modul Softwarepraktikum im Studiengang Informatik Bachelor wurde von unserem Team über das vierte und fünfte Fachsemester ein Softwareprojekt für SolarSwarm abgewickelt. Um eine Grundlage für den Roboterschwarm zu schaffen, sollte ein vom Internet getrenntes, lokales Netzwerk für Kommunikation, Kooperation, Überwachung und Steuerung der Roboter errichtet und eine benutzerfreundliche Web-Applikation entwickelt werden. In einer anfänglichen Einarbeitungsphase wurde sich mit Docker und dem Robot Operating System (ROS2) vertraut gemacht und eine Lösung für das Bilden eines eigenen Netzwerks gesucht. Mit vier zukünftig als Roboter vorgesehenen Linux-Computern als Mock-Roboter wurden regelmäßig Netzwerk- und Softwaretests durchgeführt, neue Bedürfnisse festgestellt und inkrementell auf eine Lösung hingearbeitet. Obwohl eine Grundlage besteht, wird zusätzlicher Aufwand und Verständnis über unser System erforderlich sein, um verbleibende Fehler zu beheben, um Mock-Daten durch echte Daten zu ersetzen oder das Datenmodell für spezifischere Zwecke zu erweitern. Wegen der zeitlichen Einschränkung konnten keine Feldtests mit Robotern gemacht werden. Somit ist nicht klar, wie leistungsstark ihre Hardware ist, ob das System für einen realen Einsatz brauchbar ist oder wie sich Störungen oder hohe Auslastung des Netzwerks auf das System auswirken. Eine besondere Herausforderung stellt der Verlust des Quorums im Schwarm dar. Die momentane Lösung dafür kann zur kurzzeitigen Unterbrechung mancher Dienste führen.

Diese Arbeit dokumentiert die zwei wesentlichen Ergebnisse dieser Projektarbeit, *SolarSwarm Visualizer* und *SolarSwarm Networks*, mit dem Ziel, weitere Teams aus möglicherweise unterschiedlichen Fachbereichen auf die Arbeit mit der von uns erarbeiteten Grundlage vorzubereiten. Dazu sollen die Architektur und Komponenten beschrieben, Entscheidungen begründet und Verwendung und Erweiterung erläutert werden, sowie Probleme und offene Aufgaben diskutiert werden.

1.2 Glossar

Begriff	Bedeutung	Pfleger
Host	Eine reale Maschine mit Hardware und Betriebssystem (etwa Linux), die ein Client oder Roboter werden kann oder bereits ist. Jeder Host hat eine eindeutige MAC-Adresse, die sich nur durch erheblichen Aufwand ändern lässt, es für dieses Projekt aber nicht sollte.	SolarSwarm
Client	Eine reale Maschine, die nicht als Roboter tätig ist, sondern von Menschen bedient wird oder passiv überwachend Teil des Schwarms oder nur des Netzwerks ist.	SolarSwarm
Roboter	Entweder ein mobiler Host, der später Aufgaben wie das Sammeln von Müll übernehmen soll und sich beim Routing in einem Ad-hoc Netzwerk beteiligt, oder ein Service auf der Anwendungs-Ebene, welcher Informationen über den Host (MAC-Adresse, Position, Zustand, etc.) für andere Hosts veröffentlicht.	SolarSwarm
Node/Knoten	Ein Host, der einem Docker Swarm beigetreten ist und von diesem bestimmt Docker Services startet.	Docker
Swarm	Ein Docker Swarm (siehe Kapitel 3.2.3). Gemeint ist die Verbindung von Docker über mehrere Hosts mit dem Zweck, Services unter ihnen zu orchestrieren.	Docker
Cluster	Ein Swarm mit einzigartiger Cluster-ID. Eine Node kann nur einem Cluster gleichzeitig angehören und Nodes kommunizieren nur mit anderen Nodes in diesem Cluster.	Docker
Schwarm	Entweder ein Swarm nach Docker oder mehrere Roboter, die Teil von SolarSwarm Networks sind.	SolarSwarm
Container	Ein laufendes, virtuelles Betriebssystem, in dem für einen Zweck oder eine Aufgabe nur das Nötigste vorhanden ist, damit diese unabhängig von der Hardware und dem Betriebssystem des Hosts, auf dem das virtuelle Betriebssystem läuft, erfüllt werden können. Gemeint sind hier vor allem Docker Container (siehe Kapitel 3.2.1).	SolarSwarm

Service	Abhängig vom Kontext. Auf der System-Ebene (siehe Kapitel 2.1) sind Systemd-Services gemeint, welche auf Linux vom Betriebssystem verwaltete Hintergrundprogramme sind. Auf der Docker- oder Anwendungs-Ebene könnte ein Docker Service gemeint sein, welcher als Definition eines Containers vor seinem Start verstanden werden kann (siehe Kapitel 3.2.2). Auf der Anwendungs-Ebene könnte auch ein ROS-Service gemeint sein, welcher ein Server ist, der eine Anfrage von einem Client annimmt, etwas macht und anschließend eine Antwort gibt.	SolarSwarm
Ad-hoc Netzwerk	Ein Netzwerk ohne vorhandene Infrastruktur. Stattdessen sind seine Teilnehmer für die Kommunikation und das Routing von Datenpaketen zuständig. Das Dokument *Dokumentation_neu/Einarbeitung_Geteilte_Netzwerke_und_BATMAN.md* befasst sich genauer mit der Entscheidung für MANETs als Typ und und BATMAN als Routing-Protokoll, sowie und dem Troubleshooting beim Einrichten eines Ad-hoc-Netzwerks mit BATMAN.	SolarSwarm
BATMAN/ batman_adv	Abkürzung für das Routing Protokoll B.A.T.M.A.N. (better approach to mobile ad-hoc networking) bzw. B.A.T.M.A.N. Advanced von Open-Mesh.	SolarSwarm
ROS/ROS2	Kurz für Robot Operating System. SolarSwarm verwendet momentan die Jazzy Distribution, welche jedoch während des Projekts LTS verlor.	ROS
Dependency Injection	Entwurfsmuster, welches einer Klasse Abhängigkeiten von außen bereitstellt, ohne diese selbst erzeugen zu müssen.	Backend
ASGI-Bibliothek	Das Asynchronous Server Gateway Interface ist eine Schnittstelle, welche Webservern ermöglicht mit Webanwendungen/Frameworks zu kommunizieren	
Datenbank Engine	Komponente bei Datenbanksystemen, die für das Speichern, Abrufen, Aktualisieren und Verwalten von Daten verantwortlich ist.	

2 Architektur

2.1 Übersicht

SolarSwarm Networks beschäftigt sich mit dem Errichten eines lokalen, dynamischen und selbstheilenden Netzwerks, welches durch seine Netzwerkteilnehmer gebildet wird und so unabhängig von vorhandener Infrastruktur und abgelöst vom Internet ist. Teilnehmer an sollen insbesondere Roboter mit unterschiedlicher Hardware sein, aber auch Clients, die sich mit dem Netzwerk verbinden, um den Roboterschwarm zu überwachen und zu kontrollieren. Da manche Services für Roboter mit geringerer Kapazität zu aufwendig sind oder zusätzliche Daten (etwa Kartendaten) benötigen, sollen Verantwortlichkeiten so verteilt werden, dass Roboter nicht überfordert werden. Welche Services genau auf den jeweiligen Robotern laufen sollen, kann je Service oder kategorisch nach Hardware entschieden werden.

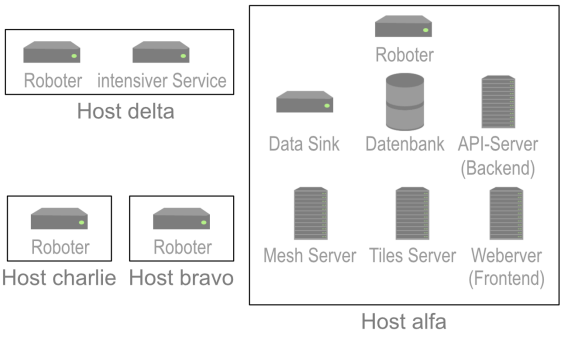
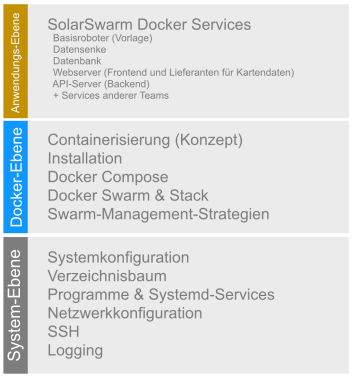


Abbildung 2a zeigt eine beispielhafte Verteilung von für den Schwarmbetrieb benötigten Services auf vier Robotern. Die Hosts bravo und charlie stellen darin leistungsschwächere Roboter dar und übernehmen daher nur notwendige Verantwortlichkeiten. Host delta übernimmt zusätzlich einen rechenintensiven Service. Häufig nehmen diese Anfragen anderer Roboter an und übernehmen eine Aufgabe wie Bildverarbeitung oder komplexe Berechnungen. Host alfa ist verantwortlich als Datensenke. Er soll Statusmeldungen von Robotern sammeln, speichern und über einen Webserver den SolarSwarm Visualizer ausliefern und Clients mit Daten versorgen. Dafür benötigt alfa umfangreiche Kartendaten, betreibt eine Datenbank und hostet einen Web- und mehrere API-Server. Da die Datensenke essenziell für Clients ist, sollte sie ein zuverlässiger, leistungsstarker Roboter (oder Nicht-Roboter) nahe der Clients möglichst vielen Robotern sein.

Der SolarSwarm Visualizer ist eine Webapplikation, welche die Visualisierung des Roboterschwarms auf einer Karte ermöglicht. Über diese sollen die Verteilung der Roboter, ihr Zustand und ihre ROS-Services zugänglich gemacht, sowie begrenzte Kontrolle ausgeübt werden.

SolarSwarm basiert auf mehreren Komponenten, von denen eine gewisse Grundmenge auf allen Roboten vorhanden sein muss. Um das Projekt besser beschreiben zu können, wird es in eine System-Ebene, eine Docker-Ebene und eine Anwendungs-Ebene (Abbildung 2b) unterteilt. Die dargestellten Begriffe je Ebene bieten nur eine thematische Eingrenzung für die Dokumentation. Die Zugehörigkeit einer Komponente zu einer Ebene entscheidet nicht darüber, wo sie in der Verzeichnisstruktur des Projekts liegt. Die Gesamtheit



aller Komponenten auf einer Ebene kann nur dann richtig funktionieren, wenn die Komponenten darunterliegenden Ebenen funktionieren. Kritische Bedingungen sind auf der System-Ebene der erfolgreiche Beitritt in ein Ad-hoc Netzwerk mit anderen Hosts und auf der Docker-Ebene der erfolgreiche Beitritt in einen Docker Swarm.

Da jeder Host einen anderen Nutzer hat, unterscheiden sich je Host auch die Home-Verzeichnisse und somit Pfade zu den zwei wesentlichen Projektverzeichnissen *solarswarm_setup/* und *solarswarm_run/*. In Kapitel 3.3.1 und der Installationsanleitung wird daher eine Voragbe für die Benennung von Hosts und deren primären Nutzer für den Schwarmbetrieb beschrieben. Ist der Host alfa, ist nach der Vorgabe sein Home-Verzeichnis /home/alfa und die Pfade zu den Projektverzeichnissen */home/alfa/solarswarm_setup/* und */home/alfa/solarswarm_run/*. Die absoluten Pfade sind für fast alle Skripte der System-Ebene relevant und sollten nicht ohne gleichzeitige Anpassung der Skripte angepasst werden. Diese definieren momentan eigene, deckungsgleiche Konstanten zu Beginn des Skripts. Falls die Pfäde notwendiger angepasst werden sollte, wird geraten, gemeinsame Konstanten in eine Datei mit Umgebungsvariablen zu verschieben, aus der alle Skripte lesen. Dies würde weitere Anpassungen erleichtern.

Zur Abkürzungen der Verzeichnisnamen werden fortan sw_setup und sw_run verwendet. Aus sw_run gehören alle Inhalte zur Anwendungs- oder Docker-Ebene. Dieses Verzeichnis soll alle für Docker Services benötigten Dateien enthalten. Aus sw_setup gehören alle Inhalte zur System- oder Docker-Ebene. Für Systemd-Services werden aber auch Dateien in */home/alfa/.ssh/*, */etc/systemd/system/* und */usr/local/bin/* hinterlegt.

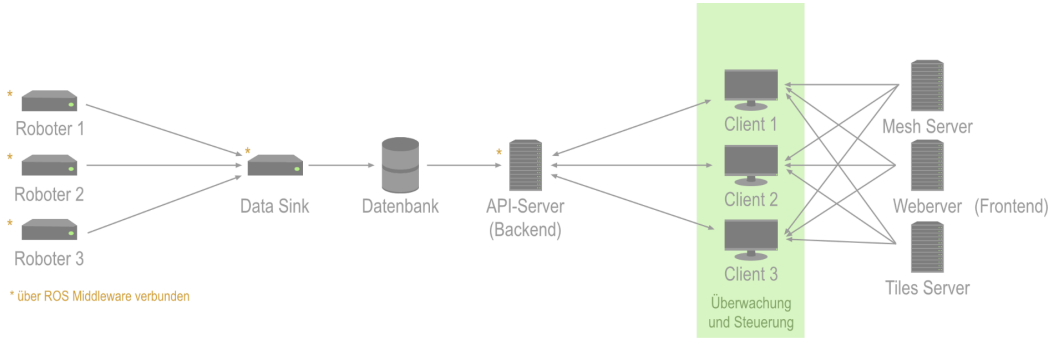
2.2 Ebenen

Die System-Ebene beinhaltet betriebssystemnahe Komponenten und Konfigurationen, die hauptsächlich in der Skriptsprache Bash umgestzt werden. Für die System- und Netzwerkkonfiguration, sowie Aktivierung der Komponenten gibt es ein Helfer-Skript *system_helper.bash*, welches der Wartbarkeit von Robotern beiträgt und bei Ergänzungen auf der System-Ebene angepasst werden sollte, um besonders das Setup und Cleanup auf einem Roboter zu erleichtern. Für automatisierte Abläufe und Systemstarts sind Systemd-Services zuständig. Ohne diese müssten alle Komponenten auf der Docker- und Anwendungs-Ebene nach jedem Reboot durch einen Menschen manuell gestartet oder bei Problemen eingegriffen werden. Da es mehrere Roboter gibt und diese auf dem Feld nicht mit einem Bildschirm, einer Computermaus und einer Tastatur verbunden sind und sich eventuell noch bewegen, sind manuelle Eingriffe höchst unpraktisch. Solange ein Roboter gesund ist, können manuelle Eingriffe über die Webapplikation remote geschehen.

Die Docker-Ebene umfasst die Vermittlung von Konzepten und des Verhaltens von Docker. Vor dem Start eines Docker Services oder Docker Swarms dienen Dateien im YAML-Format und Dockerfiles der Konfiguration und Definition. Für den Start und das Managen eines Swarms hat

Docker Kommandozeilenbefehle, welche frequent in den Systemd-Services *docker_init* und *docker_leader* verwendet werden.

Die Anwendungs-Ebene umfasst die Dokumentation aller definierten Docker Services und ROS-Packages. Abbildung 2d zeigt ein Schema für den Datenfluss zwischen diesen Services. Die Clients sind Hosts, die über einen Browser auf den vom Frontend ausgelieferten SolarSwarm Visualizer zugreifen.



3 Komponenten

3.1 ROS 2 – Grundlagen, Konzepte und Einsatz im Projekt

3.1.1 Motivation für ROS 2

ROS 2 (Robot Operating System 2) ist eine Middleware, die speziell für die Entwicklung modularer, verteilter Robotersysteme konzipiert wurde. Im Gegensatz zu monolithischen Ansätzen verfolgt ROS 2 das Ziel, komplexe Robotersysteme aus vielen unabhängigen, klar abgegrenzten Funktionseinheiten aufzubauen. Diese Funktionseinheiten werden als eigenständige Prozesse ausgeführt und über standardisierte Kommunikationsmechanismen miteinander verbunden.

Im Kontext moderner Robotik ist diese Herangehensweise insbesondere deshalb relevant, da Roboter zunehmend aus heterogenen Komponenten bestehen, die verteilt ausgeführt werden. Dazu zählen etwa Steuerungssoftware, Sensordatenerfassung, Auswertungs- und Diagnosekomponenten oder externe Systeme zur Datenpersistenz. ROS 2 bietet hierfür eine einheitliche Abstraktionsschicht, welche die Kommunikation zwischen diesen Komponenten übernimmt und die Komplexität verteilter Systeme reduziert.

Im Projekt SolarSwarm wird ROS 2 eingesetzt, um den Austausch von Roboterstatusdaten über unterschiedliche Systemgrenzen hinweg zu ermöglichen. Dazu gehört sowohl die Kommunikation zwischen mehreren Docker-Containern auf einem einzelnen Host als auch die Kommunikation zwischen mehreren Robotern innerhalb eines gemeinsamen Netzwerks. ROS 2 bildet damit die Grundlage für den Betrieb von Mehrroboter- und Schwarmszenarien.

3.1.2 Zentrale Konzepte von ROS 2

3.1.2.1 Nodes

Eine Node stellt die grundlegende Recheneinheit innerhalb eines ROS-Systems dar. Technisch handelt es sich um einen eigenständigen Prozess, der eine klar definierte Aufgabe erfüllt. Die Kommunikation zwischen Nodes erfolgt ausschließlich über die von ROS bereitgestellten Mechanismen, wodurch eine strikte Trennung der einzelnen Komponenten erreicht wird.

Im Rahmen dieses Projekts sind sowohl der Mock-Roboter als auch der Data Sink jeweils als eigene ROS-2-Nodes implementiert. Der Mock-Roboter übernimmt dabei die Rolle eines Datenproduzenten, während der Data Sink als verarbeitende und aggregierende Komponente fungiert. Diese Trennung ermöglicht es, beide Komponenten unabhängig voneinander zu entwickeln, zu testen und bei Bedarf auszutauschen.

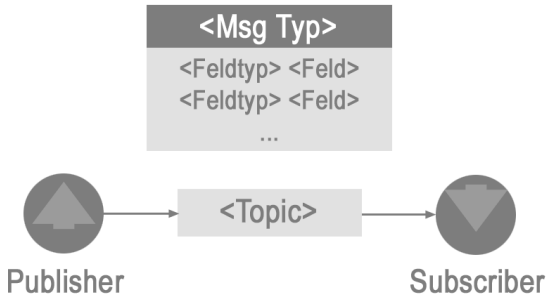
3.1.2.2 Topics und Publish-Subscribe

ROS 2 nutzt für den Großteil der Kommunikation ein Publish-Subscribe-Modell. In diesem Modell veröffentlichen Nodes Nachrichten auf sogenannten Topics, während andere Nodes

diese Topics abonnieren und die Nachrichten empfangen können. Publisher und Subscriber sind dabei vollständig entkoppelt und müssen keine Kenntnis voneinander besitzen.

Dieses Kommunikationsmodell eignet sich insbesondere für kontinuierlich anfallende Daten, wie sie in Robotersystemen typischerweise auftreten. Dazu zählen unter anderem Systemzustände wie Batteriestand oder CPU-Last, geometrische Informationen wie Position und Orientierung sowie Netzwerk- oder Nachbarschaftsinformationen.

Im Projekt veröffentlicht der Mock-Roboter diese Daten regelmäßig auf verschiedenen Topics. Der Data Sink abonniert diese Topics und empfängt die Statusmeldungen aller aktiven Roboter. Durch die asynchrone Kommunikation ist sichergestellt, dass der Ausfall oder Neustart einzelner Komponenten keinen unmittelbaren Einfluss auf andere Teile des Systems hat.



3.1.2.3 Services

Neben Topics unterstützt ROS 2 auch Services, die eine synchrone Kommunikation nach dem Anfrage-Antwort-Prinzip ermöglichen. Services werden dann eingesetzt, wenn eine gezielte Interaktion erforderlich ist, bei der eine unmittelbare Rückmeldung erwartet wird.

Im Projektkontext werden Services beispielsweise genutzt, um den Aktivitätszustand eines Mock-Roboters zu verändern oder Informationen über dessen interne Konfiguration abzufragen. Im Gegensatz zu Topics, die für kontinuierliche Datenströme vorgesehen sind, eignen sich Services insbesondere für seltene, klar abgegrenzte Operationen.

3.1.2.4 Actions

Neben Topics und Services stellt ROS 2 mit sogenannten *Actions* ein weiteres Kommunikationsparadigma zur Verfügung. Actions sind für lang andauernde, zustandsbehaftete Operationen vorgesehen, bei denen neben einer initialen Anfrage auch Zwischenstände sowie ein abschließendes Ergebnis übermittelt werden sollen. Sie kombinieren damit Eigenschaften von Publish-Subscribe- und Service-Kommunikation.

Eine Action besteht konzeptionell aus drei Bestandteilen:
einem Ziel (Goal), das vom Client an den Action-Server gesendet wird,
regelmäßigen Rückmeldungen (Feedback) über den Fortschritt der Ausführung sowie
einem Ergebnis (Result), das nach Abschluss der Aktion zurückgegeben wird.

Im Gegensatz zu Services blockiert ein Action-Aufruf den aufrufenden Client nicht dauerhaft. Stattdessen kann der Client den Fortschritt der Aktion beobachten, sie bei Bedarf abbrechen oder parallel weitere Aufgaben ausführen. Dieses Verhalten ist insbesondere für Robotersysteme relevant, da viele Aktionen – etwa Bewegungsabläufe oder komplexe Arbeitsprozesse – eine nicht vernachlässigbare Zeit in Anspruch nehmen.

Typische Anwendungsfälle für Actions in der Robotik sind unter anderem:
Bewegungsaufträge, die über mehrere Sekunden oder Minuten ausgeführt werden,
Navigationsaufgaben mit kontinuierlichem Fortschrittsfeedback oder
komplexe Systemoperationen, die während der Ausführung überwacht oder abgebrochen werden können.

Im Projekt SolarSwarm Networks werden Actions derzeit nicht aktiv eingesetzt. Die Architektur berücksichtigt dieses Kommunikationsmodell jedoch bewusst. Insbesondere die Simulation von Bewegungsabläufen oder autonomen Aufgaben des Mock-Roboters stellt ein naheliegendes Einsatzszenario für Actions dar.
Eine zukünftige Erweiterung könnte beispielsweise darin bestehen, Fahr- oder Missionsaufträge nicht mehr über Services, sondern über Actions zu steuern, um Fortschritt und Abschluss explizit abbilden zu können.

Durch die klare Trennung zwischen Topics, Services und Actions bleibt die Kommunikationsstruktur des Systems übersichtlich und an reale Robotersysteme angelehnt.

3.1.2 Middleware, Discovery und Netzwerk

Die Übertragung der ROS-Nachrichten erfolgt über die ROS-Middleware (RMW). ROS 2 basiert hierbei auf dem Industriestandard DDS/RTPS, welcher für die automatische Entdeckung von Kommunikationspartnern, die Serialisierung der Nachrichten sowie den Transport der Daten verantwortlich ist.

In diesem Projekt wird die Standardkonfiguration von ROS 2 Jazzy mit Fast DDS verwendet. Die sogenannte Discovery-Funktionalität erlaubt es Nodes, sich automatisch im Netzwerk zu finden, solange sie sich in derselben ROS_DOMAIN_ID befinden. Dadurch entfällt die Notwendigkeit, Publisher und Subscriber manuell zu konfigurieren.

3.1.3 Verteilte Kommunikation

Ein wesentliches Merkmal von ROS 2 ist die native Unterstützung verteilter Systeme. ROS-Nodes können unabhängig davon miteinander kommunizieren, ob sie im selben Prozess, im selben Container, in unterschiedlichen Containern oder auf verschiedenen physischen Robotern ausgeführt werden.

Im Projekt wird diese Eigenschaft gezielt genutzt, um Mock-Roboter und Data Sink in separaten Docker-Containern zu betreiben. Die zugrunde liegende ROS-Middleware abstrahiert dabei die Netzwerkkommunikation, sodass sich die Applikationslogik ausschließlich auf die Verarbeitung der Daten konzentrieren muss.

3.1.4 Einsatz von ROS 2 im Projektkontext

3.1.4.1 Modularität und lose Kopplung

Der Einsatz von ROS 2 verfolgt im Projekt das Ziel, eine klare Trennung der Systemkomponenten zu erreichen. Durch die Aufteilung in einzelne Nodes können Mock-Roboter, Data Sink und potenzielle Erweiterungen unabhängig voneinander entwickelt und betrieben werden.

Die lose Kopplung durch Topics und Services stellt sicher, dass Änderungen an einer Komponente keine unmittelbaren Auswirkungen auf andere Komponenten haben. Insbesondere kann der Data Sink erweitert oder ersetzt werden, ohne dass Anpassungen am Mock-Roboter erforderlich sind.

3.1.4.2 Zeitgesteuerte Statusveröffentlichung

Der Mock-Roboter nutzt ROS-Timer, um Statusinformationen in festen Intervallen zu veröffentlichen. Unterschiedliche Arten von Daten werden dabei mit unterschiedlichen Frequenzen aktualisiert. Dieses Vorgehen orientiert sich an realen Robotersystemen, bei denen Sensoren und Systemzustände ebenfalls nicht synchron erfasst werden.

Durch diese zeitliche Entkopplung entsteht ein realistischer Datenstrom, der sich gut für Tests, Auswertungen und Skalierungsexperimente eignet.

3.1.5 Mock-Roboter im ROS-System

3.1.5.1 Rolle im Gesamtsystem

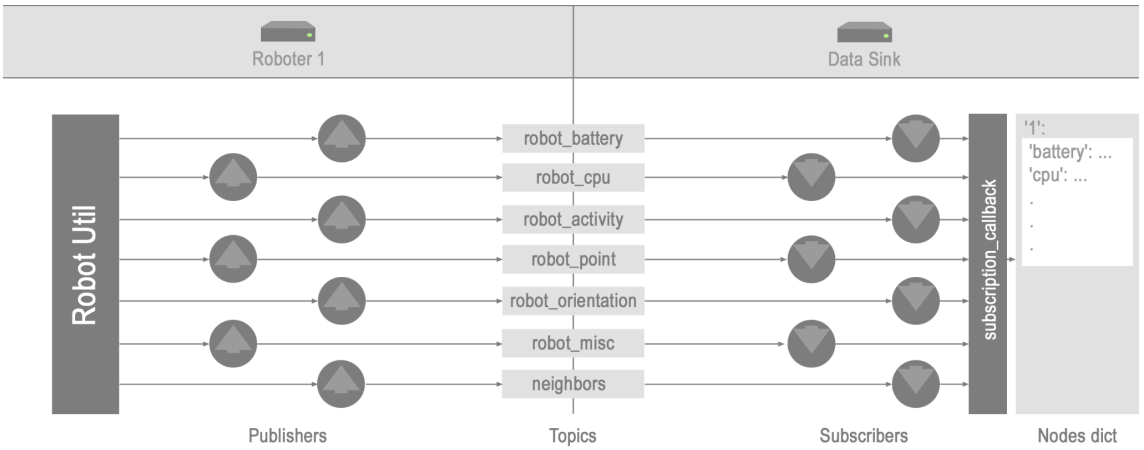
Der Mock-Roboter simuliert einen realen Roboter, indem er kontinuierlich Status- und Umgebungsinformationen erzeugt und über ROS-Topics verteilt. Aus Sicht anderer Nodes ist er nicht von einem realen Roboter zu unterscheiden.

Diese Eigenschaft erlaubt es, das Gesamtsystem frühzeitig zu testen und zu validieren, auch wenn keine physische Hardware zur Verfügung steht.

3.1.6 Identifikation und Mehrroboterszenarien

Jede vom Mock-Roboter veröffentlichte Nachricht enthält eine eindeutige Node Identifier (NID), die im Nachrichten-Header mitgeführt wird. Diese Identifikation ermöglicht den parallelen Betrieb mehrerer Mock-Roboter innerhalb desselben ROS-Systems.

Der Data Sink nutzt diese NID, um eingehende Statusmeldungen eindeutig zuzuordnen und die Daten mehrerer Roboter zu aggregieren. Dadurch ist das System von Beginn an auf Mehrroboter- und Schwarmszenarien ausgelegt.



3.1.7 Simulation von Aktivität und Bewegung

Der Mock-Roboter simuliert unterschiedliche Betriebszustände, die den typischen Lebenszyklus eines Roboters abbilden. Abhängig vom Aktivitätszustand verändert sich das Verhalten der Node, insbesondere in Bezug auf Bewegung und Statusänderungen.

Positionsdaten werden nur dann kontinuierlich aktualisiert, wenn sich der Roboter in einem aktiven Zustand befindet. In inaktiven Zuständen bleibt die Position konstant, was realistische Tests stationärer Szenarien ermöglicht.

Wurde der Mock-Roboter als ROS2 Node gebaut, kann er mit `ros2 run mock_robot mock_data` manuell gestartet werden. Zur Laufzeit wird mit `ros2 node list` die Node als `mock_robot_status_pub_<nid>` angezeigt. Sie verfügt über die drei Parameter `system_intervall` (default 3.0), `geo_intervall` (default 3.0) und `misc_intervall` (default 20.0), welche bestimmen, wie häufig Timer-Callback-Funktionen Nachrichten veröffentlichen. Da die Timer bei start der Node initialisiert werden, müssen die Parameter zur Launch-Time festgelegt werden.

Diese Tabelle stellt die Zugehörigkeit der geforderten Topics zu ihren Interfaces und den drei Timern dar:

Topic	Interface	Gruppe	Bedeutung
robot_battery	RobotBattery	system	Batteriestand data in % zwischen 0.0 und 100.0 oder gleich -1.0 (falls keine Informationen vorhanden sind) als 64-bit float
robot_cpu	RobotCpu	system	Durchschnittliche CPU-Last data aller Kerne in der letzten Minute in % zwischen 0.0 und 100.0 als 64-bit float
robot_activity	RobotActivity	system	Aktuelle Aktivität als String: 'auto' (Mock-Roboter fährt Route ab) 'manual' (Punkt wird angefahren), 'idle', 'recharge'

robot_point	RobotPoint	geo	Drei Koordinaten x, y und z als 64-bit floats
robot_orientation	RobotQuaternion	geo	Vier Werte x, y, z, w als 64-bit floats
robot_misc	RobotMisc	misc	IPv4, IPv6 und MAC als Strings
neighbors	NeighborList	geo	Liste an NIDs von Nachbarn und parallele Liste an Indikatoren (ein float je Nachbar; Einheit `dBm`) für die Verbindungsqualität

3.1.8 Data Sink als aggregierende Komponente

3.1.8.1 Aufgaben und Verarbeitung

Der Data Sink fungiert als zentraler Empfänger aller relevanten Statusmeldungen. Er abonniert die Topics der Mock-Roboter und speichert die empfangenen Nachrichten zunächst in einer internen Datenstruktur.

Die Weiterverarbeitung erfolgt batchweise in regelmäßigen Intervallen. Dieses Vorgehen reduziert Lastspitzen und ermöglicht eine konsistente Sicht auf den aktuellen Zustand des Gesamtsystems.

3.1.9 Weiterleitung und Persistenz

Abhängig von der Konfiguration können die aggregierten Daten entweder zu Debug- und Testzwecken ausgegeben oder persistent in einer Datenbank gespeichert werden. Die Trennung zwischen ROS-Kommunikation und externer Persistenz stellt sicher, dass ROS-Nodes selbst zustandsarm bleiben.

Neue Roboter werden automatisch erkannt, sobald deren erste Statusmeldungen empfangen werden.

3.1.10 Erweiterbarkeit der ROS-Anwendungs-Ebene

Die Architektur von SolarSwarm Networks ist so ausgelegt, dass neue Funktionalitäten schrittweise ergänzt werden können. Neue Statuswerte oder Funktionalitäten werden über zusätzliche ROS-Nachrichten, Topics oder Services integriert.

Neue Nachrichtentypen werden in separaten Interface-Packages definiert und bilden die verbindliche Schnittstelle zwischen Publishern und Subscribern. Änderungen an diesen Interfaces erfordern ein erneutes Bauen aller abhängigen Docker-Images, um Konsistenz sicherzustellen.

Neue Services werden analog eingeführt und sollten ausschließlich für klar abgegrenzte, synchrone Interaktionen genutzt werden. Erweiterungen, die für den Data Sink relevant sind, erfordern zusätzliche Subscriptions sowie Anpassungen der internen Datenstruktur und Verarbeitungslogik.

Durch die zentrale Aggregation im Data Sink bleibt der Anpassungsaufwand jedoch lokal begrenzt, während die restliche Systemarchitektur unverändert bleiben kann.

3.2 Docker-Ebene

3.2.1 Was ist Docker?

Docker ist eine Open-Source-Plattform für Containerisierung. Als Container wird allgemein ein laufendes, virtuelles Betriebssystem bezeichnet, in dem für einen Zweck oder eine Aufgabe nur das Nötigste vorhanden ist, damit diese unabhängig von der Hardware und dem Betriebssystem des Hosts, auf dem das virtuelle Betriebssystem läuft, erfüllt werden kann (IBM, n.d.).

Container basieren auf Images, welche einen Ausgangszustand für das virtualisierte Betriebssystem festhält. Ein Image kann durch eine Momentaufnahme oder das Bauen eines Images, basierend auf einem weiteren, entstehen. Das Verzeichnis *solarswarm_run/* enthält mehrere Dockerfiles, welche sequenziell auszuführende Anweisungen für das Bauen eines Images enthalten. Unsere Docker Services db und backend basieren beispielsweise auf öffentlichen Images postgres:13 und ros:jazzy-ros-base. Die entsprechenden Dockerfiles fügen während des Bauens der Images db und backend durch COPY dem jeweiligen Ausgangszustand lokale Dateien hinzu oder führen durch RUN Installationen aus. Wird ein Container mit dem Image db oder backend gestartet, herrschen darin die jeweiligen Ausgangszustände.

Ist beim Bauen eines Images oder Starten eines Containers ein auf hub.docker.com vorhandenes Image nicht lokal vorhanden, wird es zunächst heruntergeladen. Eigene Images sind nur dann bekannt, wenn sie gebaut oder in einem Repository abgelegt wurden. Neben Anbietern wie Docker gibt es auch die Möglichkeit, ein eigenes Repository zu hosten. Dies würde es ersparen, auf jedem Roboter dasselbe Image selbst bauen zu müssen. Aufgrund der zeitlichen Beschränkung dieser Projektarbeit wurde kein eigenes Repository erstellt, wäre jedoch eine sinnvolle Erweiterung in zukünftigen Projekten unter SolarSwarm.

3.2.2 Docker Services

In der Compose-Datei *solarswarm_run/docker-compose.yaml* definieren wir die Docker-Services base_robot, data_sink, db, adminer, backend, und frontend, sowie mehrere Server für Kartendaten und Beispielservices, welche ROS-Services anbieten sollen. Anstatt alle Images einzeln zu bauen, können mit `docker compose build [Services]` alle oder eine Auswahl an Images gebaut werden. Da das Bauen von ROS-Packages (besonders custom_interfaces) länger braucht, werden sie nur für base_robot gebaut. Alle anderen Services, die die Packages benötigen, basieren auf base_robot. Wegen dieser Abhängigkeit und da beim Bauen keine Reihenfolge angegeben werden kann, muss base_robot separat vor auf ihm aufbauenden Services gebaut werden.

In einer Compose-Datei lassen sich zusätzlich vereinfacht volumes (mit Host geteilte Verzeichnisse), networks (Netzwerke zwischen Containern) und Abhängigkeiten zwischen Services definieren. Mit `docker compose up [Services]` können alle oder eine Auswahl an Services gestartet werden. Dies eignet sich jedoch nur zum Testen oder für den Betrieb auf einem einzigen Roboter. Für den Schwarmbetrieb sollte die Compose-Datei zusammen mit der Deploy-Datei *solarswarm_run/stack-deploy.yaml* verwendet werden. Diese ergänzt die Services um für Docker Swarm spezifische Angaben und Overlay-Netzwerke (Treiber hostübergreifende Networks). Docker bietet die Möglichkeit, YAML-Dateien zu mergen (Docker Merge, n.d.). Momentan werden sie bereits automatisch kurz nach Initialisierung eines Clusters durch den Systemd-Service docker_init gemergt. Die Compose- und Deploy-Dateien beinhalten zusätzliche Hilfen in Form von Kommentaren.

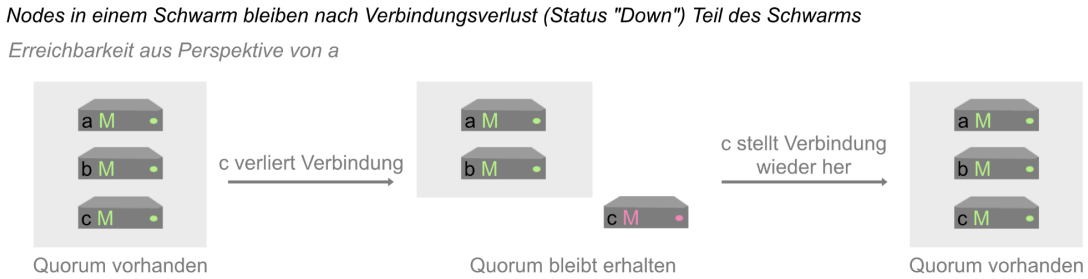
3.2.3 Docker Swarm

In einem Docker Swarm oder Cluster gibt es Manager- und Worker-Nodes. Während Worker Last (in Form von Services) übernehmen, übernehmen Manager administrative Aufgaben und Last nur optional. In SolarSwarm sind Manager auch Roboter und übernehmen daher auch Last. Daher sollen nur leistungsfähigere Roboter Manager werden können. Angestrebt wird eine momentan in docker_init angegeben ideale Anzahl an Managern. Diese sollte stets ungerade sein (Docker Admin Guide, n.d.), damit es eine Mehrheit von Managern geben kann, die sich über administrative Handlungen einigen können. Kann sich mehr als die Hälfte aller bekannten Manager austauschen, besteht ein Quorum. Nur Manager mit Quorum können beispielsweise Nodes beitreten lassen, entfernen oder Services auf ihnen replizieren lassen. Tritt ein Host einem gesunden Cluster bei, ist dieser allen erreichbaren Managern mit seiner ID bekannt. Verliert eine Node die Verbindung zum Quorum, ändert sich aus Sicht der Manager ihr Status zu „Down“ (davor „Ready“). Wird die Verbindung wiederhergestellt, bleibt der Host dieselbe Node im Cluster und ihr Status ist wieder „Ready“. Verlässt ein Host hingegen explizit das Cluster mit `docker swarm leave`, bleibt die dazugehörige Node „Down“. Tritt der Host dem Cluster wieder bei, erhält er eine andere Node-ID. Die alte Node muss von einem Manager entfernt werden.

Der Verlust des Quorums stellt für SolarSwarm eine große Herausforderung dar, da alle Manager gleichzeitig mobile Roboter sind und es immer nur ein Quorum geben kann. Ein Roboter wird im Falle eines Verlusts nicht am Arbeiten gehindert, sobald er im Cluster ist und

seine Services gestartet hat. Jedoch können Services und Nodes nicht bearbeitet, hinzugefügt oder entfernt werden. Fällt ein Roboter mit einem einmaligen, wichtigen Service aus, während kein Quorum vorhanden ist, kann er nicht auf einem weiteren Roboter gestartet werden.

Der Swarm hat mit größerer Manageranzahl eine größere Ausfalltoleranz. Einzelne Manager können kurzzeitig ausfallen, sollten bei längerem Fehlen aber demotet und vom Cluster entfernt werden, damit sich solche Fälle nicht anhäufen, bis ein Quorum-Verlust auftritt (siehe docker_leader). Abbildung 3d zeigt den kurzzeitigen Ausfall eines Managers von insgesamt drei Managern. Das graue Rechteck symbolisiert die Erreichbarkeit anderer Nodes von a aus und wird so auch in weiteren Abbildungen in diesem Kapitel verwendet.



Da Nodes auch nach Verbindungsverlust Teil desselben Clusters bleiben, kann ein Quorum wiederhergestellt werden, wenn genügend Manager wieder erreichbar werden. Wenn das Verhalten von Robotern in zukünftigen Projektarbeiten implementiert wird, sollten Manager sich nicht vom Cluster isolieren und einander aufsuchen, wenn kein Quorum besteht. Eine erste Grundlage dafür könnte die Funktion check_for_health() aus docker_init.bash sein (siehe Kapitel 3.3.4.4).

Unter Screenshots befinden sich Ergebnisse aus Tests zum Verhalten von Docker Swarm vom 01.08.2026, welche zu einem besseren Verständnis verhelfen könnten. Zu diesem Zeitpunkt wurden die Hostnamen jedoch noch nicht richtig konfiguriert. Der Hostname „IFE“ entspricht charlie und „pc-NUC8i3BEK“ entspricht bravo.

3.2.3.1 Node Labels und Constraints

Docker Swarm Services sollen später nur auf bestimmten Hosts laufen, um leistungsschwächere Roboter nicht zu überlasten, systemspezifische Eingenschaften wie CPU-Architekturen zu berücksichtigen und Daten auf dem Roboter vorauszusetzen. Beispielsweise ist es nur sinnvoll für einen Roboter, einen Service zum Ausliefern von Kartendaten zu starten, wenn er eine lokale Kopie dieser Daten hat. Docker Services können Constraints erhalten, sodass Services nur bedingt auf einer Node gestartet werden können. Für Constraints werden Node Labels verwendet, welche einer Node durch einen Manager zugewiesen werden können.

Labels sind einzelne Wörter, können aber auch Key-Value-Paare sein (Docker Admin Guide, n.d.). Aus Tests ergab sich jedoch, dass constraints in der Deploy-Datei mit einem Value angegeben werden müssen. Wenn Manager die Labels einer Node auflisten, werden auch Labels ohne Value als Key-Value-Paar mit leerem Value angezeigt. Bei binären Labels wie has_web nutzen wir daher stets Key-Value-Paare. Wir mögen etwa `--label-add has_web=true`

als Option für `docker node update` und `has_web==true` als constraint verwenden. Werden mehrere Constraints eingetragen, werden die Wahrheitswerte aller Einträge mit einem logischen UND verbunden.

Welche Labels eine Node haben muss, um einen Swarm Service replizieren zu können, wird in der Deploy-Datei festgelegt. Die von uns verwendeten Labels werden in solarswarm_setup/*docker/labels_usage.md* aufgelistet. Beide Dateien sollten erweitert werden, wenn neue Labels eingeführt werden. Von besonderer Bedeutung ist das Label can_become_manager, da durch den Systemd-Service Nodes nur als Worker beitreten können und nur dann zu Managern promotet werden, wenn ein Mangel festgestellt wird und sie dieses Label haben.

Welche Labels eine Node erhalten soll, muss in ihrer Labels-Datei stehen. Diese liegt in *solarswarm_setup/docker/* und ist benannt nach der Identität mit der Endung „.labels“. Ihr Inhalt sind alle zuzuweisenden Labels, die mit Zeilenumbrüchen voneinander getrennt sind. Labels-Dateien werden von einer neuen Node an alle bekannten Manager geschickt, da ihr unbekannt ist, welcher Manager Leader ist.

3.2.3.2 Leader und Designated Leader

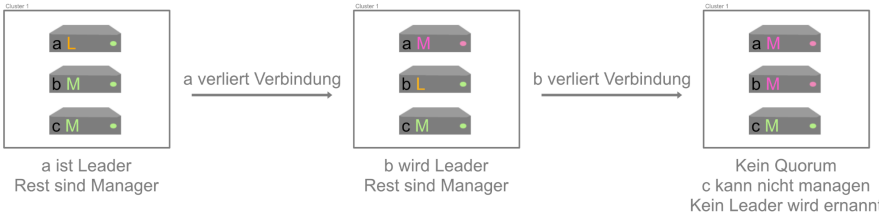
Der Leader (Anführer) in einem Docker Swarm ist ein vom Quorum bestimmter Manager. In einem neuen Cluster ist es immer diejenige Node, die es initiiert hat. Er stellt das erste Worker- und Manager-Token aus, mit denen andere Nodes beitreten können. Fällt ein Leader aus Sicht des Quorums aus, wird ein neuer Manager mit Quorum der neue Leader. Somit gibt es nur dann einen Leader, solange es ein Quorum im Cluster gibt.

Um kritische Aufgaben erteilen und Selbstheilung bei Quorum-Verlust umsetzen zu können, definieren wir einen Designated Leader. Dieser ist diejenige Node, dessen Hostname (bekannte Identität) in der Leader-Datei solarswar_setup/docker/leader steht. Während Leader nur ein Manager-Status in einem Docker Swarm und flüchtig ist, gilt der Status Designated Leader auch auf System-Ebene. Kapitel 3.3.4.5 behandelt den Systemd-Service docker_leader, welcher für die zusätzliche Aufgaben für Leader und Designated Leader zuständig ist.

Alle Hosts versuchen, über SCP die Leader-Datei von anderen Hosts zu kopieren. Dazu werden alle Identitäten der Namensliste nach abgefragt. Mindestens ein Host (bestenfalls der Designated Leader selbst) muss manuell eine Leader-Datei erhalten. Bis zum Abschluss dieser Projektarbeit ist alfa der Designated Leader.

3.2.3.3 Quorum Management

Abbildung 3e veranschaulicht, weshalb mehr als die Hälfte aller Manager einstimmig sein muss, um eine Entscheidung treffen zu können. Im Beispiel kann alfa nicht unterscheiden, ob bravo und charlie ausgefallen sind oder alfa selbst isoliert ist.



In Abbildung 3g fallen mehrere Manager einzeln aus. Solange ein Quorum vorhanden ist, kann ein neuer Leader ernannt werden.

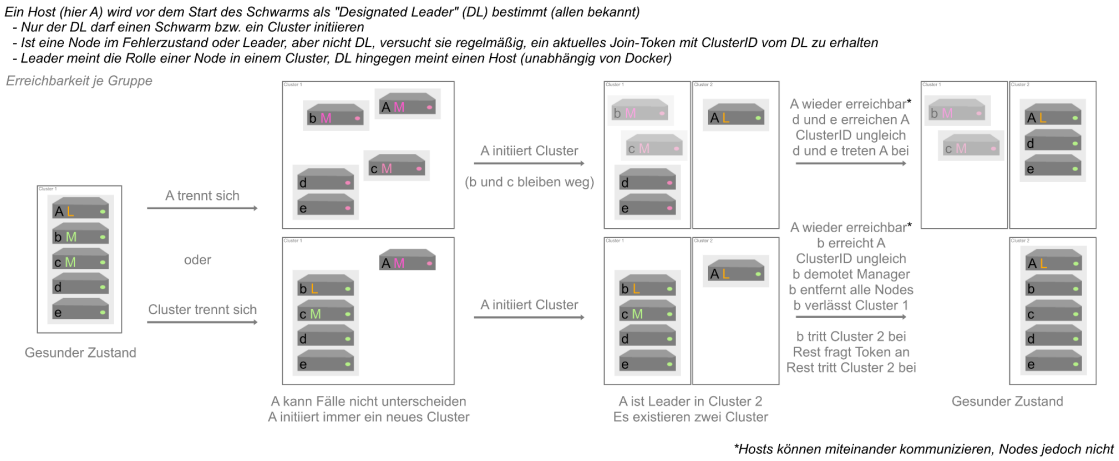


Abbildung 3h zeigt unsere Strategie für die Erholung von einem Quorum-Verlust. Stellt der Designated Leader fest, dass er für einen längeren Zeitraum kein Quorum hat, wechselt er das Cluster. Andere Nodes folgen ihm und treten als Worker bei. Es gilt zu berücksichtigen, dass Nodes selbstständig den Swarm-Modus verlassen und dadurch Services gestoppt werden. Dabei werden zwei Szenarien abgedeckt, da alfa (der Designated Leader) nicht unterscheiden kann, ob er vom Cluster isoliert ist oder andere Manager ausgefallen sind. Beide Szenarien setzen voraus, dass Worker erkennen, dass sich das Cluster in einem Fehlerzustand befindet. Für das obere Szenario wurde nicht getestet, welcher Fehler in den Logs der Worker erscheint, wenn Worker einen Manager ohne Quorum erreichen. Bis die Funktion check_for_health() im Systemd-Service docker_init um diesen Fall erweitert wird, vergleicht ein Worker seine Cluster-ID mit der des Designated Leaders, wenn mehrmals der Code „255“ (ausgegeben „-1“) zurückgegeben wird.

3.3 System-Ebene

SolarSwarm sieht vor, dass alle Roboter über eine Ubuntu 24.04 (Linux) Distribution verfügen, weswegen Komponenten der System-Ebene nur auf dieser Distribution getestet wurden. Weiter sollen Roboter unkompliziert eingerichtet werden können. Daher beinhaltet das Projektverzeichnis *solarswarm_setup/* alle für das Einrichten eines Roboters benötigten Dateien, sowie eine Anleitung. Um einheitliche Prozesse definieren und Teile dieser automatisieren zu können, sowie unerwartetes Verhalten und zusätzlichen Aufwand für individuelle Anpassungen zu minimieren, geben wir ein festes Schema vor, welches in diesem Kapitel erläutert wird.

3.3.1 Systemkonfiguration

Einen Roboter auszeichnend sind sein Hostname, sein Username, seine MAC-Adresse und eine IP-Adresse in einem Netzwerk. Da Hostname, Username (und Passwort) und IP-Adresse frei wählbar oder flüchtig sind, hängen diese von derjenigen Person ab, die den Roboter einrichten. Da wechselnde Teams an Studierenden an mehreren Robotern arbeiten sollen, müssen diese leicht zu identifizieren und unkompliziert mit ihnen zu arbeiten sein. Dazu wurden ***Identitäten*** definiert, welche die genannten Werte vorgeben.

Eine Identität in SolarSwarm Networks ist ein eindeutiger Name mit einer vorgegebenen, zugehörigen IPv4-Adresse, die im Roboterschwarm statisch ist. Alle erlaubten Namen sind im Verzeichnis *solarswarm_setup/ssh_identities/* in den Dateien *names* und *names_with_ip* vorgegeben. Diese Listen erfüllen zwei Zwecke: In erster Linie werden sie von Programmen genutzt, um über Hosts zu iterieren, Namen zu validieren und von Namen auf IP-Adressen (und umgekehrt) zu schließen. Zusätzlich bieten sie Studierenden eine Referenz, um Roboter besser unterscheiden und Testergebnisse zuordnen zu können. Für die Vergebung von Identitäten sollte sich jedoch mit einem Betreuer abgesprochen werden, da das Projekt keine Liste mit vergebenen Namen beinhaltet. Dafür eignen sich physikalische Listen eher, da Roboter ihre Listen synchronisieren müssten. Identitäten gelten jedoch nur in robotereigenen Ad-hoc-Netzwerken von SolarSwarm, um Konflikte zu vermeiden, wenn IP-Adressen dynamisch an sonstige Netzwerkteilnehmer vergeben werden (beispielsweise im Hochschul-Gastnetzwerk). Die von uns vorgegebenen Namen richten sich nach dem NATO-Alphabet in Kleinschrift (alfa, bravo, charlie, …). Die Listen können jedoch erweitert oder ersetzt werden, sollten dann aber paarweise einstimmig und auf allen Robotern gleich sein. Die IP-Adressen zählen von 192.168.1.1/24 bis 192.168.1.26/24 auf.

Jeder Roboter muss einen Nutzer mit dem Namen der ausgewählten Identität haben. Dieser soll über Administratorrechte verfügen und automatisch eingeloggt werden, wenn die Maschine hochfährt. Dadurch ist anderen Robotern stets der Nutzer und das Home-Verzeichnis bekannt, wenn sie auf Dateien (etwa beim Austausch von Logs oder Docker Join-Tokens) zugreifen wollen. Eine Maschine, die als Roboter eingerichtet wurde, sollte mit seiner Identität beschriftet werden. Die Anleitung rät zusätzlich dazu, Passwörter nach dem Schema „solarswarm-a“ für alfa, „solarswarm-b“ für bravo und „solarswarm-c“ für charlie zu wählen. Dies gilt nur, solange sich die Anforderungen an die Sicherheit nicht verschärfen, da jeder Roboter angreifbar ist, wenn das Schema bekannt ist.

3.3.2 SSH und SCP

SSH und SCP sollen es Studierenden und wichtigen Systemd Services ermöglichen, auf andere Roboter zuzugreifen und Dateien zu transferieren. Dazu authentifiziert sich ein SSH-Client bei einem auf der Zielmaschine laufendem SSH-Server mit einem Passwort oder einem Schlüssel. Die Schlüsselauthentifizierung erleichtert wiederholte Anmeldeversuche und ermöglicht Automatisierung in Skripten. Damit diese stattfinden kann, muss der SSH-Server der Zielmaschine konfiguriert (siehe Anleitung zur Einrichtung) und ein Schlüsselpaar generiert werden.

Ein Schlüsselpaar besteht aus einem öffentlichen Schlüssel und einem privaten Schlüssel (etwa alfa.pub und alfa). Jede Identität sollte nur ein Schlüsselpaar nach diesem Muster haben. Das Service-Helfer-Skript generiert den privaten Schlüssel in *solarswarm_setup/ssh_identities/* und sammelt seinen und andere öffentliche Schlüssel in *solarswarm_setup/ssh_identities/keys/*. Dieses Verzeichnis ist eine Sammlung von öffentlichen Schlüsseln. Ihre Anwesenheit bedeutet jedoch nicht, dass sie registriert sind. Nur wenn ein öffentlicher Schlüssel in *.ssh/authorized_keys/* enthalten ist, ist er registriert. Der Besitzer des privaten Schlüssels kann sich auf einem Host mit seinem registrierten, öffentlichen Schlüssel authentifizieren lassen. Daher muss ein Roboter seinen öffentlichen Schlüssel auf allen anderen Robotern registrieren, damit die Schlüsselauthentifizierung bei jedem gelingt. Dies ist besonders für die Systemd-Services docker_init und docker_leader wichtig.

Eine individualisierte Kopie der SSH-Hosts-Datei *solarswarm_setup/ssh_identities/config* wird beim Einrichten der Identität mit Hilfe des Service-Helfer-Skripts in das SSH-Verzeichnis *.ssh/* kopiert werden, um den Umgang mit SSH zu erleichtern. Ein darin definierter Host kann direkt über den vergebenen Namen angesprochen werden. User, IP-Adresse und der richtige private Schlüssel (auch IdentityFile genannt, welche hier immer der eigene private Schlüssel ist) werden durch die SSH-Host-Datei ermittelt. Abbildung 3a zeigt einen Eintrag auf alfa zu bravo.

```
Host bravo
        HostName 192.168.1.2
        User bravo
        IdentityFile ~/.ssh/alfa
```

Host alfa kann sich in einem Terminal mit dem Befehl `ssh bravo` auf bravo einloggen, wenn alfas öffentlicher Schlüssel dort registriert und Schlüsselauthentifizierung erlaubt ist.

Zu beachten ist, dass SSH-Host nur SSH bekannt ist. Der Befehl ping bravo würde das Ziel nicht erreichen, da Hosts unter Linux in /etc/hosts enthalten sind. Falls erwünscht, kann diese Datei in späteren Projektarbeiten um die Identitäten ergänzt werden. Außerdem wird das SSH-Verzeichnis nicht gefunden, wenn SSH oder SCP mit „sudo“ ausgeführt werden.

3.3.3 Service-Helfer-Skript

Das Service-Helfer-Skript *service_helper.bash* übernimmt viele aufwendige und repetitive Abläufe und hilft insbesondere dabei, die Systemd-Services auf dem Linux-System einzurichten. Das Skript informiert bei seiner Ausführung über gültige Optionen und Parameter.

Da für viele Aufgaben Administratorrechte benötigt werden, das Skript selbst aber nicht schreibgeschützt ist, wird zu Beginn immer eine Warnung ausgesprochen. Die wichtigsten Optionen des Skripts sind `wlandev`, `ssh` und `setup`, welche benötigt werden, um einen Roboter einzurichten.

`wlandev` setzt die Umgebungsvariable WLANDEV auf den übergebenen String. Dies ist wichtig für den Systemd-Service batman_adv_setup, welcher ein gültiges WLAN-Netzwerkinterface braucht, um mit anderen Robotern ein Ad-hoc-Netzwerk bilden zu können.

`ssh` setzt die Umgebungsvariablen MESH_IDENTITY und MESH_IP und finalisiert so die Identität des Roboters. Zusätzlich kann ein SSH-Schlüsselpaar generiert werden. Wird dies wiederholt, können wahlweise alte Paare überschrieben werden, solange nicht festgestellt wird, dass möglicherweise ein anderer Host bereits seinen öffentlichen Schlüssel zur neuen Identität abgelegt hat.

`setup` ändert formal den Hostnamen der Maschine zur Identität und kopiert für alle in *solarswarm_setup/system\ services/* enthaltenen Systemd-Services. Unit Files (*.service*) werden nach */etc/systemd/system/* kopiert und der Platzhalter „own_name“ wird ersetzt. Skripte (*.bash*) wernden nach */usr/bin/local/* kopiert und ausführbar gemacht. `cleanup` entfernt diese Dateien wieder und eignet sich gut, um Änderungen an Services wirkend zu machen.

`status` bietet eine Übersicht darüber, welche Umgebungsvariablen gesetzt wurden, welche Dateien an ihrer intendierten Stelle liegen, welche Services „active“ sind (laufend) und welche Services „enabled“ sind (Autostart nach Boot). `restart`, `stop`, `enable` und `disable` helfen dabei, Services während Tests zu kontrollieren.

`managed` tritt aus einem Ad-hoc-Netzwerk aus, sodass sich ein Roboter wieder mit einem Access Point verbinden kann. Dies ist notwendig, wenn er eine Internetverbindung braucht.

`send` und `collect` dienen dem Schicken und Holen von Dateien wie SSH-Schlüsseln oder Logs. Zusätzliche Parameter bestimmen, welche Dateien mit welchen Hosts geteilt werden sollen. Transferierte Schlüssel können mit dem Parameter „register“ direkt auf dem Zielhost registriert werden. Alternativ kann ein Host mit der Option `register` alle lokalen öffentlichen Schlüssel bei sich registrieren.

3.3.4 Systemd-Services

Systemd-Services auf Linux sind (vereinfacht dargestellt) Hintergrundprogramme, die in SolarSwarm zur Automatisierung des Starts des Roboters und für Healthchecks gebraucht werden. Das Verzeichnis *solarswarm_setup/system\ services/* beinhaltet je Service eine Unit File (*.service*), die den Service beschreibt, und ein Skript (*.bash*), welches durch den Service ausgeführt wird. Gewöhnlich wird das Skript in der Unit File angegeben ein Service gilt als „active“ oder „running“, wenn das Skript läuft. Daher beinhalten die meisten Skripte im

genannten Verzeichnis eine Endlosschleife. Wird ein Skript verlassen, wird ein Service beendet. In der Unit File kann jedoch angegeben werden, wann ein Skript neugestartet werden soll. Alternativ kann ein Timer (*.timer*) verwendet werden, um Services regelmäßig zu starten. Für unsere Zwecke genügt jedoch eine Endlosschleife mit `sleep <Dauer>`. System-Services werden mit dem Standardprogramm systemctl kontrolliert und gesteuert. Das Service-Helfer-Skript vereinfacht die Nutzung mit den Optionen `status`, `restart`, `stop`, `enable` und `disable`.

Einige Skripte benötigen Administratorrechte. Diese sind gegeben, wenn ein Service keine explizite Angabe eines Users in der Unit File erhält und standardmäßig vom User „root“ ausgeführt wird. Da jedoch SSH-Hosts im Home-Verzeichnis eines Users liegen, wird das Home-Verzeichnis für jeden Service geändert: `Environment=HOME=/home/own_name` enthält einen Platzhalter, der beim Einrichten der Services mit der `setup`-Option des Service-Helfer-Skripts ersetzt wird.

Das auszuführende Skript für einen Service wird in der Unit File mit `ExecStart` angegeben. Damit ein Service nicht inaktiv wird oder ständig neustartet, wenn das Skript keine Schleife enthält, kann es mit `ExecStartPre` gestartet werden. Da auf Linux jedes Programm eine Datei ist, kann anschließend das Programm `sleep` angegeben werden: `ExecStart=/bin/sleep infinity`.

3.3.4.1 iw_dump

In einem Ad-hoc-Netzwerk sind häufig nicht alle Knoten direkter miteinander verbinden, sondern nur über andere Knoten. Damit ein Roboter (gemeint ist hier eine ROS-Node) einer Datensenke mitteilen kann, welche Hosts seine direkten Nachbarn sind, wird regelmäßig mit dem Programm iw ein Station Dump in die Datei *solarswarm_run/tmp/iw_dump/iw_dump.txt* geschrieben. Um Probleme bei gleichzeitigen Zugriffen auf die Datei zu vermindern, wird der Output erst in eine temporäre Datei geschrieben, welche dann *iw_dump.txt ersetzt*. Zurzeit werden nur MAC-Adresse und durchschnittliche Signalstärke vom Roboter aus dem Output gelesen (siehe `Util.get_neighbors()` des ROS-Packages *mock_robot*).

3.3.4.2 rx_copy

Dieser Service dient dazu, den Umgang mit durch SCP erhaltenen Dateien zu bestimmen. Wird eine Datei mit SCP transferiert, kann sie an jedem ungeschützten Ort abgelegt werden. Holt sich ein Host von einem anderen Host eine Datei, kann er sich aussuchen, wo er sie ablegt. Schickt ein Host eine Datei an einen anderen Host, ist es erwünscht, dass diese in dem Verzeichnis solarswarm_setup/rx/ abgelegt wird. Wir geben bisher zwei Unterverzeichnisse in rx/ vor: *ssh/* und *docker/*. Beispielsweise kopieren Hosts ihre öffentlichen SSH-Schlüssel auf anderen Hosts in *rx/ssh/* oder ihre Labels-Dateien in *rx/docker/*.

3.3.4.3 batman_adv_setup und batman_adv_healthcheck

Der Service batman_adv_setup konfiguriert das Netzwerkinterface WLANDEV und versetzt es in den Ad-hoc-Modus, sodass es einem selbstdefinierten Netzwerk beitreten kann. Zum Routing wird das unter Linux als Kernel-Modul vorinstallierte Routing-Protokoll batman-adv verwendet. Als Grundlage für das Skript diente der offizielle Guide von Open-Mesh (open-mesh, 2016). Dieser verwendet das Programm avahi-autoipd für die dynamische Zuweisung von IP-Adressen, welches durch die statischen IP-Adressen der Identitäten ersetzt wurde. Da es bei genauem Befolgen des Guides nicht gelang, Roboter in ein Ad-hoc-Netzwerk zu integrieren, mussten über mehrere Iterationen Anpassungen vorgenommen werden.

Während späterer Tests wurde festgestellt, dass Host bravo einen Defekt hat und regelmäßig die Verbindung zum Ad-hoc-Netzwerk verliert. Dies kann nur durch einen Neustart des Services behoben werden. In einem früheren Fall wurde beobachtet, dass sich zwei Hosts (jetzt alfa und bravo) nach wenigen Minuten nicht mehr sehen konnten. Da Ausgaben des Tools batctl ließen jedoch nicht erkennen, dass es ein Problem gab. Der zusätzliche Service batman_adv_healthcheck soll daher batman_adv_setup neustarten, wenn das Tool batctl länger keine Nachbarn findet oder einen Fehlercode zurückgibt.

3.3.4.4 docker_init

Dieser Service sollte auf allen Robotern während ihrer Arbeit laufen. Er dafür zuständig, (a) einen Docker Swarm zu initiieren und ein Join-Token auszustellen, wenn ein Roboter Leader ist, (b) den aktuellen Leader zu erfragen und seinem Swarm beizutreten und (c) seinen eigenen Zustand als Docker Swarm Node zu überwachen. Aufgaben (a) und (b) werden beim Start des Skripts nur dann ausgeführt, wenn ein Roboter sich nicht bereits als Teil eines Swarms sieht, der Swarm-Modus als „inactive“ ist. Tritt ein Roboter einem Docker Swarm bei, sieht er sich als Teil dessen, bis er ihn explizit mit `docker swarm leave` verlässt. Aufgabe (c) wird nach Betreten einer Endlosschleife bis zum Stopp des Services wiederholt.

Der Service bricht frühzeitig ab, wenn die Identität des Roboters nicht bestimmt wurde. Ist ein Roboter nicht im Swarm-Modus, muss er im Skript zuerst den Designated Leader (DL) bestimmen. Dazu schaut er lokal in *solarswarm_setup/docker/* oder fragt in einer Endlosschleife nach einer Leader-Datei von anderen Hosts, bis er eine Datei findet und den darin angegebenen Host erreichen kann. Ist ein Roboter selbst DL, initiiert er einen Swarm und erstellt eine Datei *worker_token* mit einem Worker-Join-Token und der aktuellen ClusterID im selben Verzeichnis. Sonst fragt er so lange worker_tokens vom Leader oder anderen Hosts an, bis ihm der Beitritt in ein Cluster gelingt. Beitretende Roboter sind zunächst Worker und pausiert und übernehmen daher keine Aufgaben im Swarm. Der aktuelle Leader (nicht unbedingt DL) muss zunächst eine Labels-Datei (siehe Kapitel 3.2.3) vom jeweiligen Roboter empfangen und den Nodes vorgesehene Labels zuweisen (siehe Kapitel 3.3.4.5). Dennoch beginnt im Skript die Überwachung.

Der Zustand der eigenen Node sollte stets „active“ sein. Sonst wird das Skript verlassen und der Service neugestartet. „active“ sagt jedoch nichts über die Node im Verhältnis zu anderen Nodes aus. Daher wird in diesem Fall geprüft, ob die Node ein Manager oder sogar ein Leader ist und

andere Manager erreichen kann. Der aktuelle Leader und der DL sollen stets den Service docker_leader ausführen. Sind Leader und DL nicht gleich, übernimmt der DL nur einen Teil seiner Aufgaben in docker_leader.

Da nur Manager einen Überblick über Nodes im Swarm haben, filtern Worker mit dem Standardprogramm `journalctl` Logs von Docker, um zu erkennen, ob sie dem DL in ein anderes Cluster folgen müssen. Siehe Abbildung 3b für ein Aktivitätsdiagramm.

3.3.4.5 docker_leader

Dieser Service sollte nur durch docker_init auf einem Leader oder dem DL gestartet werden. Er ist dafür zuständig, (a) doppelte Hostnamen zu entfernen, (b) als DL bei längerem Verlust des Quorums die Bildung eines neuen Clusters zu erzwingen, (c) Labels zu neuen Nodes hinzuzufügen und sie zu aktivieren, (d) länger unerreichbare Manager zu entfernen und (e) fähige Worker zu Managern zu erheben, wenn es einen Mangel an Managern gibt.

Die genannten Aufgaben werden sequenziell in einer Endlosschleife abgearbeitet. Zuvor wird jedoch geprüft, ob der Roboter noch Leader und der Schwarm in einen gesunden Zustand ist. Ist ein Roboter nicht mehr Leader, wird der Service beendet, solange er nicht DL ist. Hat der Roboter kein Quorum mehr, wartet er ab oder bildet ein neues Cluster, wenn er DL ist. Nur der DL darf ein neues Cluster bilden, wonach ihm alle Nodes folgen müssen. Der DL führt zu diesem Zweck auch dann den docker_leader Service aus, wenn er selbst nicht mehr Leader ist (beispielsweise wegen eines vorrübergehenden Verbindungsabbruchs). Nur der aktuelle Leader ist für Aufgaben (c) bis (e) zuständig.

Die ideale Anzahl an Managern und abzuwartende Dauer, bis ein unerreichbarer Manager entfernt oder ein neues Cluster gebildet wird, sind über Konstanten im Skript bestimmbar. An Managern sollte es stets eine ungerade Zahl geben, welche sich nach Größe des Clusters richtet, um Ausfalltoleranz zu gewährleisten (Docker Admin Guide, n.d.). Momentan macht das Skript Worker nach Möglichkeit zu Managern, auch wenn es dadurch eine ungerade Zahl an Managern im Swarm gibt. Siehe Abbildung 3c für ein Aktivitätsdiagramm.

4 Verwendung

Zur Einrichtung eines Roboters liegt *Dokumentation_neu/* eine Anleitung bei. Für den Start eines Roboters können mit dem Service-Helfer-Skript alle Systemd-Services gestartet werden. Werden enabled, startet der Roboter automatisch nach Systemneustart.

Zum Testen können die Skripte der Systemd-Services auch manuell aus *solarswarm_setup/system\ services/* gestartet werden. Werden Änderungen daran vorgenommen, sollten mit dem Service-Helfer-Skript ein Cleanup und anschließend ein Setup durchgeführt werden, um die Services zu aktualisieren. Die in *solarswarm_run/docker-compose.yaml* definierten Docker Services können auch ohne Swarm-Betrieb auf einem Roboter getestet werden. Der Befehl `docker compose up -d` startet alle gleichzeitig. Ohne die Option `-d` werden Log-Nachrichten aller Services direkt in der Konsole angezeigt. Services können auch wahlweise gestartet werden: `docker compose up base_robot data_sink db` startet nur die drei aufgelisteten Services. Um Constraints auf einem Roboter zu testen, kann dieser mit `docker swarm init` ein eigenes Cluster initiieren und einen Stack deployen (siehe *solarswarm_run/stack-deploy.yaml*).

Hinweis: Den Systemd-Service docker_init manuell zu stoppen, beendet nicht den Swarm-Modus auf einem Host. Das Service-Helfer-Skript hingegen verlässt ein Cluster, wenn der Service mit der Option `stop` beendet wird. Dazu muss der Swarm explizit mit `docker swarm leave -f` verlassen werden. Die Option `-f` ist nur für Manager-Nodes nötig, wenn nach dem Verlassen nicht genügend Manager im Swarm sind.

Hypothese zur späteren Nutzung

Angenommen, vier Roboter seien eingerichtet und verfügen bereits über grundlegende Funktionen zur Navigation und Bewegung. Die Systemd-Services der Roboter sind enabled, sodass sie automatisch gestartet werden, wenn Studierende die Maschinen einschalten am Institut für Elektromobilität einschalten, damit die Roboter arbeiten. Die Roboter verlassen selbstständig die Werkstatt und gehen ihren Aufgaben nach. Nach einigen Stunden werden sie zurückgerufen und sammeln sich wieder in der Werkstatt. Von einem Roboter oder Client (etwa ein Laptop eines Studierenden) aus, der sich im Ad-hoc-Netzwerk befindet, werden mit dem Service-Helfer-Skript alle Logs eingesammelt. Anschließend werden alle Roboter ausgeschaltet und für den nächsten Tag zum Laden zu ihren Ladestationen gebracht.

5 Erweiterung

5.2 Docker-Ebene

Momentan treten alle Roboter dem Docker Swarm als Worker bei. Manager können jedoch auch Manager-Token ausstellen. Wenn klar ist, welche Roboter Manager werden und sie sofort als solche beitreten sollen, können diese ein Manager-Join-Token verwenden. Dazu sind Anpassungen an den Systemd-Services docker_init und eventuell docker_leader nötig. Da Manager sich selbst Labels zuteilen können, könnten sie jedoch auch mit Availability „Active“ beitreten und sich selbst Labels zuteilen. Dann wären keine Anpassungen an docker_leader nötig. Eventuell sollten dann aber zusätzliche Checks eingebaut werden, damit nicht zu viele Manager einem Cluster beitreten. Dies stellt an sich keine Gefahr dar, könnte aber dazu führen, dass eine ungerade Anzahl an Managern im Cluster ist und bei einer Spaltung zwei Hälften ohne Quorum existieren.

Womit sich nicht genauer beschäftigt wurde, sind Raft-Snapshots. Wie in Kapitel 3.2.3.3 angesprochen, werden Services auf Nodes beendet, wenn sie das Cluster wechseln. Zukünftige Projektarbeiten könnten herausfinden, ob man über Raft Snapshots die Verteilung der Services auf Nodes zu rekonstruieren kann, damit dieselben Roboter ihre Aufgaben wieder aufnehmen, insofern dies erwünscht ist. Alternativ könnten Services mit Constraints genauer eingegrenzt werden, damit sie stets auf denselben Robotern gestartet werden.

Um zentral Images bauen und in ein Repository pushen zu können, sodass sich Roboter ihre Images daraus pullen können, könnte ein eigenes Repository gehostet werden. Die Compose-Datei in *solarswarm_run/* enthält Verweise auf Quellen, die dabei helfen könnten.

5.3 System-Ebene

Ein umfangreicher Aspekt, den die Aufgabenstellung an diese Projektarbeit nicht voraussetzte, ist Sicherheit. Das robotereigene Ad-hoc-Netzwerk ist nicht geschützt. Jeder Host kann ihm beitreten. Da die Passwörter aller Roboter recht unsicher sind und SSH mit Passwortauthentifizierung aktiv ist, sind die Roboter aus der Nähe sehr angreifbar. Die Sicherheit wurde in der Aufgabenstellung vernachlässigt, da die Umsetzbarkeit von SolarSwarm im Vordergrund stand und Roboter keine Anbindung zum Internet haben sollten. Für den Betrieb im Freien wäre es jedoch sinnvoll, grundlegende Maßnahmen umzusetzen. Ein Anfang könnte sein, SSH-Hosts besser zu authentifizieren. Außerdem sind sicherheitskritische Informationen wie SSH-Schlüssel, IP-Adressen, Koordinaten von Testergebnissen und das Passwort der Datenbank im öffentlichen Git-Repository (github.com) einsehbar. Besondere Gefahrenquellen für die Roboter sind das Service-Helfe-Skript und die Systemd-Services, welche mit Administratorrechten ausgeführt werden. Da der Projektstand in einem öffentlichen Repository liegt und alle Roboter diesen klonen, können schadhafte Anpassungen alle Roboter beeinflussen.

Werden Anpassungen am Service-Helfer-Skript vorgenommen, sollte auch die Übersicht der Optionen aktualisiert werden. Werden zusätzliche Systemd-Services hinzugefügt, sollte das Service-Helfer-Skript ebenfalls aktualisiert werden, damit `setup`, `celanup`, `restart`, etc. diese berücksichtigen.

6 SolarSwarm Visualizier

6.1 Backend

6.1.1 Architekturübersicht

Das Backend ist als schlanke, modulare Service-Applikation angelegt, deren Architektur aus wenigen klar abgegrenzten Komponenten besteht: einer FastAPI-basierten HTTP und JSON Schicht, einer ORM (Object Relational Mapping) Schicht mit SQLAlchemy, einer PostgreSQL Datenbank und dem ROS2 Workspace, die innerhalb eines Docker Containers per Shell Subprocess angesprochen werden. Zentrales Element ist die FastAPI Anwendung, die mittels Pydantic Modelle die Eingaben validiert und die HTTP-Endpunkte für Abfragen und Operationen bereitstellt. Die Pydantic Modelle spiegeln dabei die Datenstrukturen wieder, die später als Tabellen im Datenbankschema erstellt werden. Die Tabellen werden durch ORM Klassen von SQLAlchemy mit Python Klassen definiert, welche SQL Queries zur Tabellenerstellung in Form von Klassen in Python bereitstellt. Beim Start der App werden die Modell Metadaten mit der konfigurierten Datenbank Engine verknüpft, so dass die notwendigen Tabellen in der PostgreSQL Instanz angelegt werden. Zur Verwaltung von Datenbankzugriffen wird für jeden Request eine Session bereitgestellt, welche über eine Dependency-Injection erzeugt und automatisch wieder geschlossen wird. Dieses Verfahren in Kombination mit der Validierung von Pydantic sorgt für saubere Transaktionsgrenzen und reduziert das Risiko von Verbindungsabbrüchen. Die API ist bewusst durch GET-Requests für das lesen implementiert, denn für jede Klasse in der Datenbank existieren Endpunkte, die die gespeicherten Informationen abfragen und als JSON zurückliefern, während spezielle Endpunkte die Liste verfügbarer ROS2-Services ermitteln und Service Aufrufe an einzelne Roboter übermitteln. Technisch werden die ROS2 Befehle nicht direkt in die Anwendung eingebettet, sondern in einem gesourcten ROS Workspace als Subprocess via Docker aufgerufen, wodurch die Laufzeitumgebung isoliert innerhalb des Containers bleibt und die ROS spezifischen Abhängigkeiten nicht lokal installiert sein müssen. Die Initialisierung der Datenbasis erfolgt durch ein separates Setup-Skript, das Standardzustände eingepflegt und serverseitig einen Trigger zur automatischen Aktualisierung des Roboter Status anlegt. Abschließend sorgt ein Docker basiertes Deployment dafür, dass sowohl die ROS2 Laufzeit als auch die Python Anwendung reproduzierbar und portabel zur Verfügung stehen. Die Containerisierung kapselt Umgebungsvariablen, Abhängigkeiten und Startsequenzen, sodass konsistente Laufzeit Bedingungen gewährleistet sind.

6.1.2 Komponenten & Dateien

Das Backend setzt sich aus vier Python Skripten und einer Dockerfile zusammen, die gemeinsam die komplette Server Funktionalität bereitstellen. Jede dieser Dateien übernimmt

eine klar abgegrenzte Aufgabe, sodass Datenbankanbindung, Datenmodell, API-Logik und Initialisierung voneinander getrennt bleiben und das System leichter wartbar ist.

Die *main.py* bildet das zentrale Element der Anwendung und stellt die eigentliche FastAPI Anwendung bereit. In dieser Datei werden alle HTTP Endpunkte definiert, über welche das Frontend oder andere Systeme mit dem Backend kommunizieren können. FastAPI selbst bringt Starlette, eine ASGI-Bibliothek und Uvicorn (ein ASGI Webserver, für die Kommunikation zwischen Webserver und Framework), um die Funktion von FastAPI zu gewährleisten. In der Dockerfile wird auch per CMD Befehl der Webserver Uvicorn gestartet, damit FastAPI funktioniert und abrufbar ist. Mit Pydantic zur Datenvalidierung überprüft FastAPI eingehende Daten anhand von definierten Modellklassen, bevor diese weiterverarbeitet oder in der Datenbank gespeichert werden. Die verwendeten Modelle orientieren sich an den Tabellenstrukturen der Datenbank und stellen sicher, dass nur vollständige und korrekt strukturierte Daten verarbeitet werden. Zu Beginn der Anwendung wird die FastAPI Instanz erzeugt, die anschließend sämtliche Endpunkte verwaltet. Danach werden die Datenbankverbindung und die ORM-Modelle eingebunden. Über die Metadaten der definierten Modelle werden die notwendigen Tabellen automatisch mit der Datenbank Engine verknüpft und bei Bedarf in der PostgreSQL Datenbank angelegt. Für jeden eingehenden Request wird über eine Dependency Injection Funktion eine Datenbank Session bereitgestellt, die nach Abschluss der Anfrage wieder geschlossen wird, wodurch stabile Datenbankzugriffe und saubere Transaktionen ermöglicht werden. Zusätzlich enthält die *main.py* die Logik zur Ausführung von ROS2 Befehlen. Hierfür wird innerhalb des Containers ein Subprocess gestartet, der den ROS Workspace lädt und anschließend ROS Services ausführen kann. Die bereitgestellten Endpunkte sind überwiegend als GET Requests umgesetzt und dienen dazu, die Inhalte der einzelnen Tabellen auszulesen und für das Frontend als JSON zurückzugeben. Ergänzend existieren spezielle Endpunkte, über die verfügbare ROS Services eines Roboters abgefragt und anschließend direkt aufgerufen werden können. Zusätzlich gibt es Debug Endpunkte, die Services ohne Datenbankabfrage direkt über den ROS Workspace ausführen, was besonders für Tests und Fehlersuche hilfreich ist.

```python
app = FastAPI()
models.Base.metadata.create_all(bind=engine)

# Pydantic Modelle
class StatusBase(BaseModel):
    robot_id: int
    battery: Optional[float] = None
    cpu_1: Optional[float] = None
    point: Optional[Dict[str, Any]] = None
    orientation: Optional[Dict[str, Any]] = None
    last_heard: int

class ServiceCall(BaseModel):
    service_name: str
    service_type: str
    arguments: Optional[Any] = Field(default=None, description="dict or raw string")
    timeout: Optional[int] = 10

class StatusResponse(StatusBase):
    status_id: int

    class Config:
        from_attributes = True

class RobotBase(BaseModel):
    nid: str
    state_id: Optional[int] = None
    display_name: Optional[str] = None
    ipv4: Optional[str] = None
    ipv6: Optional[str] = None
    mac: Optional[str] = None

class RobotResponse(RobotBase):
    robot_id: int

    class Config:
        from_attributes = True
```

Die *models.py* enthält die Definition der Datenbankstrukturen mithilfe des SQLAlchemy ORMs. SQLAlchemy erlaubt es, Datenbanktabellen als Python Klassen zu definieren, wodurch keine direkten SQL Befehle geschrieben werden müssen. Die Datei importiert dazu die Base Klasse aus der Datenbank Konfiguration und nutzt diese als Grundlage, um Python Klassen zu erstellen, die automatisch als Tabellen interpretiert werden. In diesen Klassen werden Tabellenname, Attribute und Beziehungen zwischen Tabellen beschrieben. Innerhalb des Projekts werden Tabellen für den Roboter Zustand, die Roboter Identität, Statusdaten, Informationen über den Nachbars Roboter und Zustandsänderungen definiert. Dadurch entsteht eine strukturierte Datenbasis, in der sowohl technische Informationen wie IP-Adressen als auch Statuswerte wie Batteriestand oder CPU Auslastung gespeichert werden können. Zusätzlich wird über die Tabelle für Zustandsänderungen eine Historie aufgebaut, die später zur Analyse oder Überwachung genutzt werden kann.

```python
# models.py
from sqlalchemy import Column, Integer, String, Float, ForeignKey, BigInteger, JSON
from database import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class State(Base):
    __tablename__ = "State"

    state_id = Column(Integer, primary_key=True, index=True)
    state = Column(String(64))

    robots = relationship("Robot", back_populates="state")
    state_changes = relationship("StateChange", back_populates="state")

class Robot(Base):
    __tablename__ = "Robot"

    robot_id = Column(Integer, primary_key=True, index=True)
    nid = Column(String(64), unique=True, index=True)
    state_id = Column(Integer, ForeignKey("State.state_id"))
    display_name = Column(String(64))
    ipv4 = Column(String(16))
    ipv6 = Column(String(40))
    mac = Column(String(24))

    statuses = relationship("Status", back_populates="robot")
    state_changes = relationship("StateChange", back_populates="robot")
    state = relationship("State", back_populates="robots")

class Status(Base):
    __tablename__ = "Status"
```

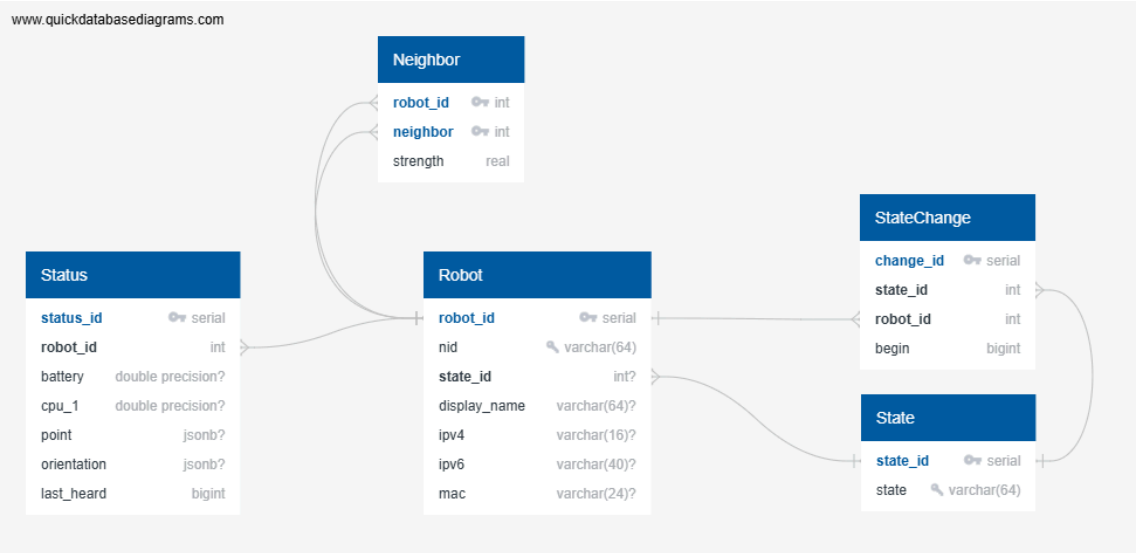
Die *database.py* übernimmt die Aufgabe, die Verbindung zur PostgreSQL Datenbank bereitzustellen. Hier wird die Datenbank Adresse definiert, die aus Datenbanktyp, Benutzername, Passwort, Host, Port und Datenbankname besteht. Diese Informationen werden in der Regel über Umgebungsvariablen an den Container übergeben, sodass der Container den Zugriff auf die Datenbank gewährleisten kann. Mithilfe dieser Verbindungsinformationen erzeugt SQLAlchemy eine Engine, über die später alle Datenbankoperationen ausgeführt werden. Zusätzlich wird eine Session erzeugt, die für jede Anfrage eine Verbindung zur Datenbank herstellt. Die Datei definiert außerdem die Base Klasse, die benötigt wird, um Python Klassen als Datenbankmodelle zu registrieren. Ohne diese Basisklasse könnten die Tabellen nicht automatisch erzeugt oder verwaltet werden.

Die *setup.py* dient zur Initialisierung der Datenbank und wird üblicherweise erst nach dem Start der Umgebung ausgeführt. Das Skript öffnet eine Datenbank Session und trägt Standardzustände in die State Tabelle ein, damit Roboter bereits bekannte Betriebszustände verwenden können. Zusätzlich wird ein Datenbank Trigger eingerichtet, der automatisch einen Eintrag in der Tabelle für Zustandsänderungen erzeugt, sobald sich der Status eines Roboters ändert. Dadurch muss diese Historie nicht manuell durch die Anwendung gepflegt werden, sondern wird direkt von der Datenbank verwaltet, was die Anwendung vereinfacht und Fehler reduziert.

Gemeinsam bilden diese Dateien die Grundlage des Backends, wobei jede Komponente eine klar definierte Rolle übernimmt. Durch die Aufteilung in API Logik, Datenmodell, Datenbankanbindung und Initialisierung bleibt die Architektur übersichtlich und ermöglicht eine einfache Erweiterung oder Anpassung des Systems.

6.1.3 Datenbankmodell

Die Anbindung der Datenbank basiert im Backend hauptsächlich auf dem Zusammenspiel der Dateien *models.py* und *database.py*, welche gemeinsam die Struktur der Daten und die Verbindung zur PostgreSQL Datenbank bereitstellen. Die *models.py* ist dabei entscheidend für die Definition aller Tabellen, die zur Verwaltung der ROS Roboter benötigt werden. Mithilfe des SQLAlchemy ORMs werden hier Tabellen nicht direkt über SQL Befehle, sondern über Python Klassen beschrieben. Jede dieser Klassen repräsentiert eine Tabelle innerhalb der Datenbank und enthält die jeweiligen Attribute sowie die Beziehungen zu anderen Tabellen. Auf diese Weise werden alle benötigten Datenstrukturen zentral im Code definiert und können von der Anwendung direkt genutzt werden. Im Projekt werden Tabellen für Zustände der Roboter, die eigentlichen Roboter Einträge, Statusinformationen, Nachbarschaftsbeziehungen zwischen Robotern sowie die Historie von Zustandsänderungen erstellt. Jede dieser Klassen wird über eine gemeinsame Basisklasse registriert, wodurch SQLAlchemy erkennt, dass es sich um ein Datenbankmodell handelt und die entsprechenden Tabellen automatisch in der Datenbank erzeugt werden können.



Die *database.py* stellt die eigentliche Verbindung zur PostgreSQL Datenbank her. Hier wird die Datenbank Adresse konfiguriert, welche Angaben zum verwendeten Datenbanksystem, Benutzer, Passwort, Host, Port und dem Namen der Datenbank enthält. Auf Grundlage dieser Verbindungsinformationen erzeugt SQLAlchemy eine sogenannte Engine, die später alle Datenbankoperationen ausführt. Zusätzlich wird eine Session Konfiguration erstellt, die es der

Anwendung ermöglicht, für jede Anfrage eine eigene Verbindung zur Datenbank zu öffnen und nach Abschluss wieder zu schließen. Dadurch wird sichergestellt, dass mehrere Anfragen parallel verarbeitet werden können, ohne dass Verbindungen dauerhaft offen bleiben oder blockiert werden.

Ergänzend dazu wird die *setup.py* verwendet, um die Datenbank nach dem ersten Start mit grundlegenden Informationen zu befüllen. Da es schwierig war, mithilfe der SQLAlchemy eigenen DDL Funktionen einen Trigger zur automatischen Verarbeitung von Zustandsänderungen zu definieren, wurde dieser Schritt direkt über SQL Befehle umgesetzt, die über die bereits vorhandene Engine an die Datenbank gesendet werden. Das Skript trägt dabei zunächst die benötigten Standardzustände in die entsprechende Tabelle ein und richtet anschließend einen Trigger auf den Status des Roboters ein. Dadurch muss diese Logik nicht in der Anwendung selbst umgesetzt werden, sondern wird direkt durch die Datenbank übernommen.

Die Dockerfile stellt sicher, dass die gesamte Anwendung in einem Container läuft, der bereits den ROS2 Jazzy Workspace enthält. Dadurch können die Python Skripte direkt innerhalb dieser ROS Umgebung ausgeführt werden, ohne dass auf dem Hostsystem eine separate ROS Installation notwendig ist. Während der Entwicklung traten hierbei Probleme mit Berechtigungen und dem Auffinden des ROS Workspaces auf, da unterschiedliche Pfade und Zugriffsrechte berücksichtigt werden mussten.

6.1.4 API-Referenz & ROS-Integration

Die API des Backends stellt die Schnittstelle zwischen Datenbank, ROS Umgebung und externen Anwendungen dar und ermöglicht den Zugriff auf gespeicherte Roboter Informationen sowie die Ausführung von ROS Services über HTTP Anfragen. Technisch ist die Anwendung so aufgebaut, dass innerhalb des Containers ein Subprocess gestartet wird, der zuvor den ROS Workspace lädt und das Workspace sourced. Dieser Prozess erlaubt es der Anwendung, ROS Service Befehle auszuführen, ohne dass diese direkt in Python implementiert werden müssen. Dadurch bleibt die ROS Umgebung gekapselt innerhalb des Containers und kann unabhängig vom eigentlichen Anwendungscode genutzt werden.

Die bereitgestellten Endpunkte sind hauptsächlich als GET Requests umgesetzt und dienen dazu, die Inhalte der einzelnen Datenbanktabellen auszulesen. Da das Backend selbst keine Daten in die Datenbank schreibt, sondern lediglich die Struktur bereitstellt und vorhandene Daten ausliest, ermöglichen diese Endpunkte eine einfache Kontrolle, ob Tabellen korrekt befüllt wurden. Für jede Tabelle existiert ein eigener Endpunkt, über den die gespeicherten Informationen abgefragt und anschließend als JSON Daten an das Frontend oder andere Anwendungen zurückgegeben werden. Auf diese Weise kann der aktuelle Zustand der Roboter, deren Statusinformationen sowie weitere gespeicherte Daten zentral über die API eingesehen werden.

Zusätzlich existieren spezielle Endpunkte zur Interaktion mit ROS Services. Ein Teil dieser Endpunkte greift zunächst auf die Datenbank zu, um die Zuordnung eines Roboters zu ermitteln und anschließend die für diesen Roboter verfügbaren Services über den ROS Workspace auszulesen oder aufzurufen. Diese Endpunkte nutzen intern die Funktion, welche den Subprocess mit der ROS Umgebung startet und anschließend die entsprechenden Service Befehle ausführt. Ergänzend dazu existieren weitere Endpunkte mit gleicher Funktionalität, die jedoch direkt mit dem ROS Workspace arbeiten, ohne zuvor die Datenbank zu verwenden. Diese Variante dient hauptsächlich zu Debugging Zwecken, da Services sofort abgefragt oder aufgerufen werden können, selbst wenn noch keine Datenbankeinträge vorhanden sind. Auf diese Weise bietet die API sowohl eine reguläre Betriebsform über gespeicherte Roboter Informationen als auch eine direkte Zugriffsmöglichkeit für Tests und Fehlersuche.

6.1.5 Deployment

Das Deployment des Backends erfolgt vollständig containerisiert über Docker bzw. der Dockerfile und docker compose, wodurch sichergestellt wird, dass alle benötigten Abhängigkeiten, Laufzeitumgebungen und Dienste reproduzierbar gestartet werden können. Grundlage des Backends ist ein Docker Image, das auf einem ROS2 Jazzy Basisimage aufbaut. Dadurch steht die komplette ROS Umgebung bereits im Container zur Verfügung und muss nicht separat auf dem Hostsystem installiert werden. Innerhalb des Containers wird zunächst ein Arbeitsverzeichnis definiert, in das anschließend alle benötigten Python Abhängigkeiten installiert werden. Danach werden die Python Anforderungen aus der Projektstruktur geladen und installiert, bevor der eigentliche Anwendungscode in den Container kopiert wird. Zusätzlich wird der von der FastAPI Anwendung verwendete Netzwerkport freigegeben, sodass die API später von außen erreichbar ist. Beim Start des Containers wird zunächst der ROS Workspace geladen und anschließend der FastAPI Server gestartet, wodurch sowohl ROS Funktionen als auch die Web API innerhalb derselben Umgebung verfügbar sind.

Das Zusammenspiel der einzelnen Dienste wird anschließend über docker compose organisiert, wodurch mehrere Container gleichzeitig gestartet und miteinander verbunden werden können. Neben dem Backend selbst existieren zusätzliche Container, die unterschiedliche Aufgaben innerhalb des Gesamtsystems übernehmen. Dazu gehört ein Container, der als Basis für Roboter Funktionen dient und beispielsweise Netzwerkdaten oder Logdateien bereitstellt. Ein weiterer Container übernimmt die Rolle eines Data Sink und sammelt Statusinformationen oder leitet diese an die Datenbank weiter. Die eigentliche Persistenzschicht wird über einen separaten Datenbankcontainer bereitgestellt, der PostgreSQL ausführt und seine Daten in einem persistenten Volume speichert, sodass Informationen auch nach einem Neustart der Container erhalten bleiben. Ergänzend dazu wird ein Adminer Container bereitgestellt, der eine Weboberfläche zur Verfügung stellt, über die Tabellen und Datenbankinhalte komfortabel im Browser betrachtet und bearbeitet werden können.

Der Backend Container selbst erhält über Umgebungsvariablen die notwendigen Informationen zur Verbindung mit der Datenbank sowie die Angabe der verwendeten ROS Distribution. Durch diese Struktur lässt sich das gesamte System mit einem einzigen Befehl starten, wodurch Datenbank, Backend, Roboterdienste und Verwaltungsoberflächen automatisch bereitgestellt werden und eine einheitliche Laufzeitumgebung entsteht

6.1.6 Betrieb & Troubleshooting

In der Regel reicht es aus, den Startbefehl einmal auszuführen, wodurch Backend, Datenbank und weitere benötigte Komponenten gemeinsam hochfahren und die Anwendung anschließend erreichbar ist. Im Ordner von SolarSwarmNetworks/solarswarm_run/

```
docker compose build base_robot && docker compose up —-build
```

Sollte es jedoch während der Entwicklung oder nach mehreren Tests zu kleineren Problemen kommen, kann es hilfreich sein, die bestehende Umgebung zunächst vollständig zu stoppen und die zugehörigen Volumes zu entfernen, bevor die Container erneut aufgebaut werden.

```
docker compose down -v && docker compose up —-build
```

6.1.6.1 Troubleshooting

Während der Entwicklung und beim Testen der Docker Umgebung kam es häufiger vor, dass Container oder Images nicht korrekt beendet wurden und dadurch Netzwerkports, insbesondere der PostgreSQL Port, weiterhin blockiert waren. In solchen Fällen konnte die Anwendung nicht erneut gestartet werden, da Docker meldete, dass bestimmte Adressen bereits verwendet werden. Um diese Probleme schnell beheben zu können, wurde eine kleine Kombination aus Docker-Befehlen zusammengestellt, die nicht mehr benötigte Container, Images, Volumes und Builddaten entfernt. Diese Befehle können direkt in der Kommandozeile ausgeführt werden und helfen dabei, die Docker Umgebung wieder in einen sauberen Zustand zu versetzen, sodass die Container anschließend problemlos neu gestartet werden können.

```
docker compose down
```

```
docker system prune -a -f
```

```
docker volume prune -f
```

```
docker builder prune -a -f
```

Zusätzlich kann es vorkommen, dass weiterhin die Meldung erscheint, dass eine Adresse bereits verwendet wird. In diesem Fall kann überprüft werden, welche Container noch aktiv sind, und diese anschließend manuell gestoppt werden. Dadurch werden blockierte Ports wieder freigegeben und die Anwendung kann erneut gestartet werden.

```
docker ps -l
```

```
docker stop process
```

Sollte das Problem weiterhin bestehen, kann es außerdem hilfreich sein, den lokal laufenden PostgreSQL Dienst auf dem Hostsystem zu stoppen, da dieser ebenfalls den Datenbank Port belegen kann.

```
systemctl stop postgresql
```

6.1.7 Probleme

Im Verlauf der Entwicklung des Backends traten mehrere technische Schwierigkeiten auf, die vor allem durch das Zusammenspiel von ROS, Docker, Datenbank und Web Framework entstanden sind. Viele dieser Probleme zeigten sich erst während praktischer Tests, da mehrere voneinander abhängige Komponenten gleichzeitig korrekt funktionieren mussten. Im Folgenden sind die wichtigsten Probleme sowie deren Ursachen und Lösungen aufgelistet.

- Zu Beginn der Entwicklung war die Anwendung davon abhängig, dass auf dem jeweiligen System bereits eine funktionierende ROS Installation vorhanden war. Wenn diese fehlte oder falsch konfiguriert war, konnten keine ROS Befehle ausgeführt werden und die Anwendung war nicht nutzbar. Besonders bei Tests auf unterschiedlichen Rechnern führte dies zu Problemen. Die Lösung bestand darin, die komplette Anwendung inklusive ROS in Docker Container auszulagern. Dadurch wird die ROS Umgebung direkt im Container bereitgestellt und die Anwendung funktioniert unabhängig von der lokalen Installation des Nutzers.

- Ein weiteres Problem entstand bei der Integration der FastAPI Anwendung in eine Docker Umgebung, die gleichzeitig den ROS Workspace enthalten musste. Zu Anfang war geplant, alles über einzelne Dockerfile zu nutzen, jedoch zeigte sich schnell, dass sowohl Python-Abhängigkeiten als auch ROS Komponenten korrekt eingerichtet werden mussten. Dadurch kam es zu Startproblemen der Anwendung. Gelöst wurde dies durch eine angepasste Container Struktur, in der die ROS Umgebung korrekt geladen wird, bevor der FastAPI Server gestartet wird. Der Container startet dabei zuerst den ROS Workspace und führt anschließend den Webserver aus, sodass beide Systeme zuverlässig zusammenarbeiten.

- Während der Entwicklungsphase wurden häufig Container gestartet, gestoppt und neu gebaut. Dabei kam es regelmäßig vor, dass Container nicht vollständig beendet wurden und bestimmte Netzwerk Ports weiterhin blockiert waren. In solchen Fällen erschien die Fehlermeldung, dass eine Adresse bereits verwendet wurde und der neue Container nicht starten konnte. Dieses Problem wurde durch die oben genannten Bereinigungs Befehle behoben, um nicht mehr verwendete Container und Images zu entfernen.

- Ein weiteres Problem bestand darin, dass die FastAPI Anwendung teilweise schneller startete als die Datenbank, wodurch beim Start noch keine Verbindung zur Datenbank aufgebaut werden konnte. Dies führte dazu, dass Tabellen nicht erstellt oder Abfragen nicht korrekt ausgeführt wurden. Die Lösung bestand darin, ein separates Setup Skript einzusetzen, das die Datenbank initialisiert und notwendige Tabellen sowie Startdaten anlegt. Durch die weitere Entwicklung wurde das Skript aber verworfen.

- Während der Entwicklung trat mehrfach der Eindruck auf, dass Tabellen nicht erzeugt wurden oder die Datenbank leer sei. Bei genauer Überprüfung stellte sich jedoch heraus, dass die Tabellen vorhanden waren, jedoch über eine andere Portkonfiguration der Datenbankinstanz angesprochen wurden und zwar Port 5433. Wurde nun auch wieder gelöst, indem die Dockerfile bearbeitet und die Portzuweisung korrigiert wurde.

- Ein weiteres Hindernis ergab sich bei der Umsetzung von Datenbank-Triggern. SQLAlchemy bietet keine komfortable Möglichkeit, solche Trigger direkt über die ORM Konfiguration zu definieren. Da jedoch Zustandsänderungen von Robotern automatisch aktualisiert werden sollten, musste ein alternativer Weg gefunden werden. Aus diesem Grund wurde das Setup Skript ersetzt, sodass Trigger direkt über SQL Befehle in der Datenbank erstellt werden. Dadurch konnte die gewünschte Funktionalität umgesetzt werden.

6.2 Frontend - Visualizer

Im Verlauf des Projekts wurde ersichtlich, dass das Projekt in seinem ursprünglichen Umfang zu gering erschien. Spontanes experimentieren hat dann ein erster Prototyp hervorgebracht, mit der Absicht, die im praktischen Geplanten Einsätze verwalt- und steuerbar zu machen.

6.2.1 VueJS

Bei VueJS handelt es sich um ein Web-Framework zum Entwerfen von UI und Webanwendungen. Unter den Rahmenbedingungen des Projekts wurde VueJS gewählt. Vue bietet Nutzenden die Möglichkeit, gebündelte Komponenten zu entwerfen, den *Component.vue* Dateien. Innerhalb dieser ist eine klare Struktur kodiert: Einen Abschnitt, der zur logischen Programmierung dieser Komponente zuständig ist (<*script*>), gefolgt von <*template*>, in welchem das HTML-Konstrukt gebaut werden kann. Das Konstrukt kann dabei von den zuvor berechneten Werten aus dem <*script*>-Block beeinflusst werden. So können beispielsweise Listen von Strings berechnet werden, welche dann innerhalb der <*template*>-Blöcke benutzt werden können, um Listen darzustellen. Innerhalb der <*style* > Blöcke können CSS-Werte definiert werden.

Gepaart mit der weiten Verbreitung und der daraus entstehenden Bekanntheit gibt es rund um VueJS eine Vielzahl an Informationen und Tipps. Es ist ein leichtes und umfangreiches Framework mit viel Support und einer Vielzahl an Bibliotheken.

Um Letzten Endes Drohnen verwalten zu können, ist es unabdingbar, dass eine räumliche Visualisierung der Umgebung notwendig ist.

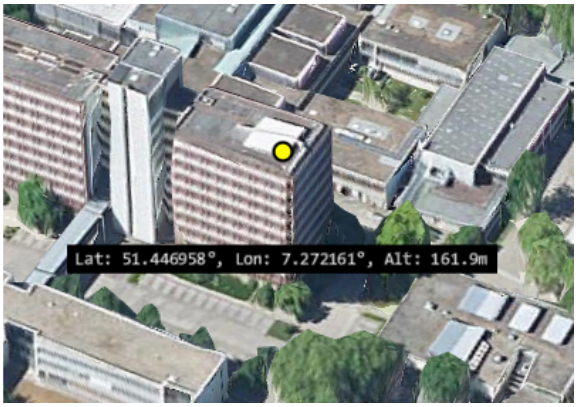
6.2.2 Visualisierung

Im Rahmen dessen sah die erste Version des Visualizers eine schlichte, Interaktive Karte mit OpenStreetMaps-Daten vor, auf welcher Roboter und Drohnen angezeigt werden würden. Die Daten sind öffentlich erhältlich, können frei genutzt und bearbeitet werden. Eine Implementierung in purem 2D fand jedoch nicht statt, weil wir ziemlich schnell auf die Bibliothek „Cesium“ Aufmerksam wurden. Cesium bietet die Möglichkeit, die gesamte Erde als 3D-Model, samt Erhöhungen, anzeigen zu lassen. Dazu lassen sich viele Objekte in die Welt setzen, es können Distanzen gemessen werden – es kann also beliebig mit der Welt interagiert werden. Diese Möglichkeit würde es uns außerdem

erlauben, sowohl OpenStreetMaps-Karten als auch 3D-Meshes und Höhen-Karten darzustellen. Der Visualizer benutzt drei Datensätze, um die Umgebung aufzubauen:

1) Ein Base-Layer in Form eines OpenStreetMap-Layers, also Karten-Texturen auf einer geodätischen Kugel (gute Erdannäherung, jedoch keinerlei Höheninformationen)
2) Elevation-Layer, was dem Base-Layer Höhendaten hinterlegt. So sind Berge und Höhenunterschiede erkennbar, außerdem stimmen die gemeldeten Koordinaten von GPS-Geräten nun mit der korrekten Höhe überein (Empfänger “schwebt” ansonsten in der Luft)
3) 3D-Mesh-Tiles, große Meshes, die über das Gelände gelegt werden. Sie erlauben in den meisten Fällen die genauesten Darstellungen von Gebäuden, Bäumen und anderen Hindernissen, die besonders mit autonomen Robotern eine Rolle spielen.

Mit diesen 3D-Meshes ist es nun zum Beispiel möglich, beispielsweise die Höhe von Gebäuden auszurechnen. Die 3D-Meshes bieten eine sehr hohe Datengenauigkeit, woraus wir die Höhe des Zentralcampus der Hochschule Bochum mit einer Höhe von etwa 34 Metern bestimmen könne,



Die 3D-Meshes werden vom

6.3.1 Datenverwaltung

Die Verwaltung oder das Betreiben von autonomen Robotern und Netzen kann insbesondere durch die hohe Menge anfallender Datensätze und Informationen schwierig werden. Angenommen, in einem Netzwerk befinden sich 10 Roboter, die

jeweils alle 3 Sekunden Daten versenden, so gehen im Backend rund 1500 Datensätze pro Minute ein. Rechnet man mit einigen wenigen Kilobyte entstehen schnell hunderte Megabyte an Daten.

Laut Konzeption und Lastenheft war es deshalb wichtig, eine möglichst reaktive Anwendung zu schaffen, die Problemlos mit hunderten Datenpunkten umgehen kann, vor allem im Hinblick auf die Modularität des Projekts, was eine höhere Menge an Robotern erlauben würde.

Das Frontend bzw. die Webanwendung benutzt aus diesem Grund ein System aus In-Memory-Cache, um einen schnellen Datenzugriff zu ermöglich. Damit auch länger persistente Daten aufrufbar sind, bildet die IndexedDB-Datenbank eine Art „Zwischenschicht“. Sie empfängt sämtliche eingehenden Daten und speichert diese persistent ab. Dabei handelt es sich um eine Transaktionelle Datenbank, in der Key-Value-Paare abgelegt werden können [1]. Sie bleiben auch nach dem Schließen des Browsers bestehen und eignen sich so als einen lokalen, unabhängigen Speicher.

Die Anwendung bezieht ihre Daten ausschließlich aus dem Backend. In regelmäßigen Abständen werden Anfragen von dem Frontend (*DronePollingService.ts*) aus an die Datenbank geschickt. Wie in anderen Kapiteln erklärt, stellt das Backend Endpunkte bereit, die dann angesprochen werden. In Abbildung 1 ist zu sehen, wie Anfragen an das Backend (*Database*) geschickt werden, welche darauf hin mit neuen Daten antwortet. Diese Neuen Daten werden dann wiederrum an die IndexedDB weitergegeben, wo sie dann entsprechend gelagert werden können. Anschließend gibt es innerhalb des *DroneHistoryStore.ts* den genannten Cache. In diesem ist nur ein kurz Ausschnitt aller Daten der IndexDB zu finden, die Daten sind dafür jedoch extrem schnell zugreifbar.

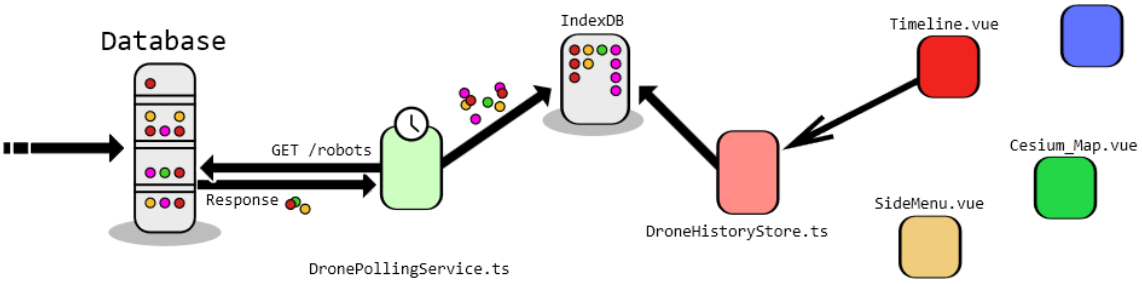
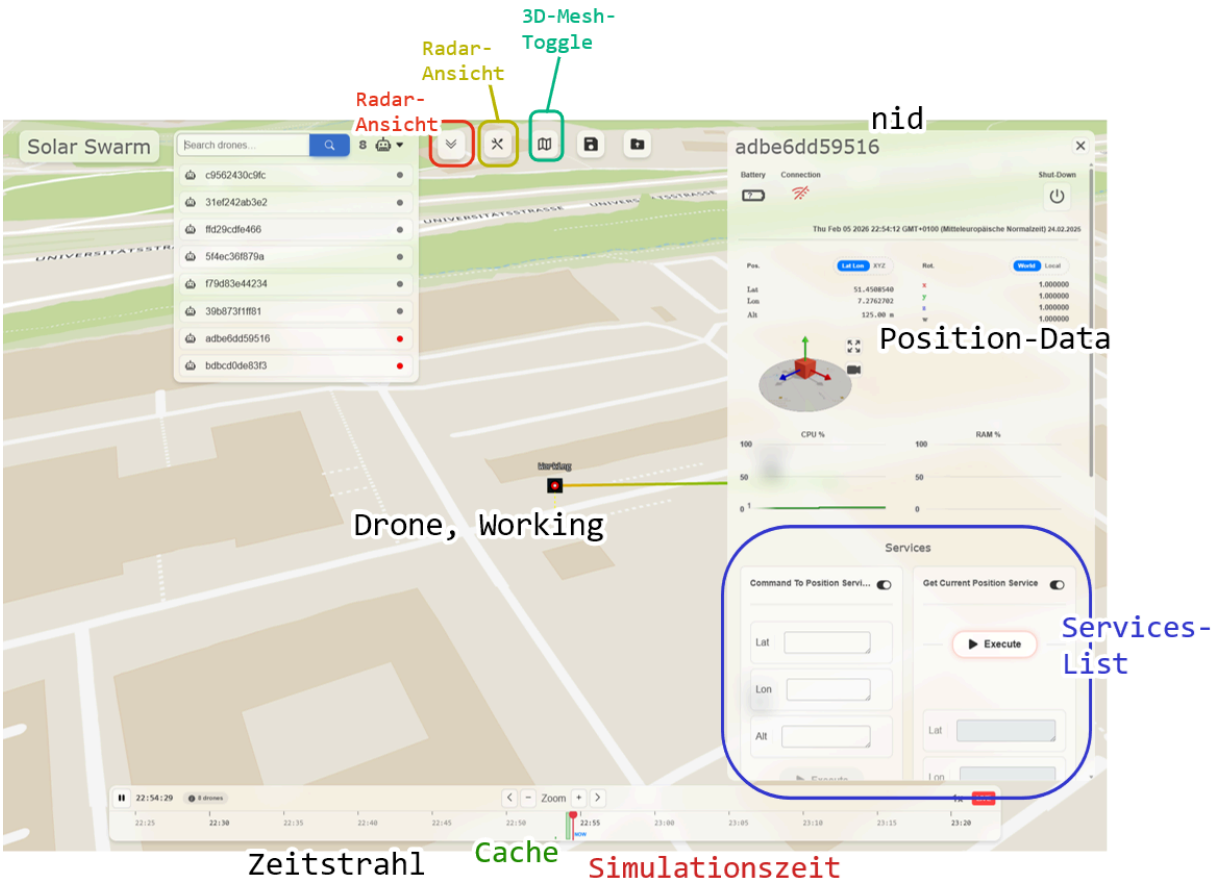


Abbildung 1 Data-Flow

User Interface

Das User Interface ist wie in der folgenden Abbildung zu sehen ist aufgebaut. In der oberen Leiste ist eine Suchleiste zu finden, mit der man direkt zu den gesuchten Robotern springen kann. Dann folgen (v. L. n. R.): Schalter zum Wechseln zwischen Radar und Cesium Ansicht, der Button zum Öffnen der GeoTools, ein Button zum Ein- und Ausschalten des 3D-Meshes, ein Speichern der gesamten IndexedDB sowie ein entsprechendes erneutes Einspielen dieser Daten.



Das rechte Seitenmenü zeigt die gerade ausgewählte Drohne bzw. den ausgewählten Knoten an. Zu sehen sind Informationen über die Akkuleistung, den Empfang, die aktuelle Position und Rotation, die aktuelle CPU-Auslastung sowie am unteren Ende des Menüs die geladenen Services, die der ausgewählte Knoten ausführen kann.

Der Zeitstrahl am unteren Rand des Bildschirms zeigt die aktuelle Simulationszeit in Form eines roten Strichs auf dem Zeitstrahl an. Ein weiterer, blauer Strich zeigt dabei die reale Zeit an. Der grüne Bereich auf dem Zeitstrahl zeigt an, dass der oben erläuterte Cache in diesem Zeitraum Inhalte geladen hat.

In der Mitte, die eigentliche 3D-Umgebung, werden die Drohnen dargestellt. Ein Doppelklick auf diese fokussiert sie und lässt das Seitenmenü erscheinen.

Quellenverzeichnis

Administer and maintain a swarm of Docker Engines. (o. D.). Docker Documentation. https://docs.docker.com/engine/swarm/admin_guide/

B.A.T.M.A.N. Advanced quick start guide. (o. D.). Open Mesh. https://www.open-mesh.org/projects/batman-adv/wiki/Quick-start-guide

Susnjara, S., & Smalley, I. (2024, Juni 6). *Was ist Docker und wie wird es benutzt?* IBM. https://www.ibm.com/de-de/think/topics/docker

Eric Roby*, How to build a FastAPI app with PostgreSQL (11.06.2023)*. YouTube. https://www.youtube.com/watch?v=398DuQbQJq0

FastAPI Dokumentation. (o. D.). Tiangolo. https://fastapi.tiangolo.com/

ROS Robot Operating System. (o.D.) Open Robotics. https://docs.ros.org/en/jazzy/index.html

Docker Dokumentation (o,D,). Docker Inc. https://docs.docker.com/

PostgreSQL Dokumentation. (o. D.). The PostgreSQL Global Development Group. https://www.postgresql.org/docs/

SQLAlchemy Dokumentation. (21.01.2026) Michael Bayer. https://docs.sqlalchemy.org/en/20/