HAW HAMBURG

Bachelorarbeit

Leo Peters

# Vergleich von verteilten Architekturansätzen zur Steuerung autonomer Roboter am Beispiel des Ninebot Loomo

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department Computer Science*

Leo Peters

# Vergleich von verteilten Architekturansätzen zur Steuerung autonomer Roboter am Beispiel des Ninebot Loomo

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 26. August 2024

**Leo Peters**

**Thema der Arbeit**

Vergleich von verteilten Architekturansätzen zur Steuerung autonomer Roboter am Beispiel des Ninebot Loomo

**Stichworte**

Ninebot Loomo, Verteilte Systeme, Middleware, ROS, RPC, Java

**Kurzzusammenfassung**

In dieser Arbeit werden zwei Middleware-Lösungen zum Einsatz in autonomen Robotersystemen untersucht: das Robot Operating System (ROS) und die selbst entwickelte Middleware RoboLink. Ziel der Arbeit ist es, die Stärken und Schwächen beider Ansätze unter verschiedenen Aspekten, wie der Leistung, Effizienz und Benutzerfreundlichkeit zu untersuchen. Während ROS ein standardisiertes, weit verbreitetes Framework mit einem umfangreichen Ökosystem darstellt, wurde RoboLink entwickelt um Schwachstellen von ROS zu adressieren, wie etwa den Single Point of Failure und die fehlende Unterstützung für eine asynchrone, bidirektionale Kommunikation. Die Analyse beinhaltet verschiedene Tests zur Bewertung der Umlaufzeit, des Durchsatzes und des Ressourcenverbrauchs der beiden Middleware-Lösungen. Die Tests wurden auf allgemeingültigen Systemen durchgeführt, um Erkenntnisse zu gewinnen, die auf verschiedene Roboterplattformen übertragbar sind. Die Ergebnisse zeigen, dass RoboLink insbesondere bei der Leistung und Effizienz überlegen ist, ROS hingegen bietet durch ein breites Ökosystem klare Vorteile für Projekte, die viel zusätzliche Funktionalität erfordern. Die Arbeit diskutiert zum Abschluss die Limitationen und die Übertragbarkeit der Ergebnisse auf den Loomo. Außerdem gibt sie einen Ausblick auf zukünftige Forschungsfelder, wie die Integration weiterer Roboterplattformen und die Ausführung von Tests direkt auf dem Loomo.

**Leo Peters**

**Title of Thesis**

"Comparison of Distributed Architecture Approaches for Controlling Autonomous Robots using the Example of the Ninebot Loomo

**Keywords**

Ninebot Loomo, Distributed systems, Middleware, ROS, RPC, Java

**Abstract**

This thesis examines two middleware solutions for use in autonomous robotic systems: the Robot Operating System (ROS) and a custom middleware RoboLink. The objective of the work is to investigate the strengths and weaknesses of both approaches under various aspects, such as performance, efficiency and usability. While ROS is a standardized, widely adopted Framework with an extensive ecosystem, RoboLink was developed to address ROS's weaknesses, such as the single point of failure and the lack of support for asynchronous, bidirectional communication. The analysis includes different tests to evaluate the round trip time, throughput and resource consumption of the two middleware solutions. The tests were conducted on general purpose machines to gain insights that can be transferred to different robotic platforms. The results show that RoboLink is superior in terms of performance and efficiency while ROS offers clear advantages due to its broad ecosystem for projects that require a lot of additional functionality. The thesis discusses the limitations and transferability of the results to the Loomo. It also provides an outlook on future research, for example the integration of other robotic platforms and the conducting of tests directly on the Loomo.

# Inhaltsverzeichnis

# Abkürzungen

**CPU** Central Processing Unit.

**DDS** Data Distribution Service.

**JVM** Java Virtual Machine.

**LAN** Local Area Network.

**NTP** Network Time Protocol.

**RAM** Random Access Memory.

**ROS** Robot Operating System.

**RPC** Remote Procedure Call.

**SDK** Software Development Kit.

**TCP** Transmission Control Protocol.

**UDP** User Datagram Protocol.

# 1 Einleitung: Vergleich von Middleware-Lösungen für autonome Roboter

## 1.1 Hintergrund, Motivation und Zielsetzung

Autonome Roboter revolutionieren verschiedenste Sektoren der Industrie, von Fertigungsanlagen über Operationssäle bis hin zu unseren Straßen, sind sie überall zu finden. Besonders in den letzten zwei Jahrzehnten hat die Anwendung solcher Roboter stark zugenommen und sie sind in der Lage, immer komplexere Aufgaben zu bewältigen. Autonomie in Robotersystemen wird definiert durch die Fähigkeit eines Roboters, eigenständig Entscheidungen zu treffen und Aufgaben ohne externe Steuerung auszuführen. Damit solch komplexe Mechanismen möglich werden, wird ein System benötigt, das den Roboter befähigt effizient zu arbeiten. Teile eines solchen Systems laufen oft auf mehreren Maschinen um die Last aufzuteilen oder mehrere Roboter zugleich zu steuern. Ein wichtiger Bestandteil eines solchen verteilten Systems ist die Middleware. Sie ist verantwortlich für die Kommunikation und den Datenaustausch zwischen den verschiedenen Software-Komponenten. Das Robot Operating System (ROS), welches im nächsten Kapitel eingehend vorgestellt wird, ist ein weit verbreiteter Standard zum Einsatz als Middleware in Robotersystemen. In dieser Arbeit soll erforscht werden, welche Vor- und Nachteile ROS gegenüber einer selbst entwickelten Middleware (RoboLink) für die Steuerung autonomer Roboter bietet.

Abbildung 1.1: Der Ninebot Loomo

## 1.2 Der Ninebot Loomo

Der Ninebot Loomo, zu sehen in Abbildung 1.1, dient in dieser Arbeit als repräsentatives Beispiel für einen autonomen Roboter. Der Loomo ist ein zweirädriges, innovatives, persönliches Transportgerät. Er ist eine Kombination aus einem Segway und einer Roboterplattform, die über KI-Fähigkeiten und autonome Funktionen verfügt. Einige der wichtigsten Eigenschaften und Funktionen des Loomo sind unter anderem:

- **Autonomes Fahren:** Der Loomo kann sich autonom fortbewegen und dabei Hindernissen ausweichen. Durch seine verschiedenen Sensoren wie mehrere Kameras, einen Ultraschall- und einen Infrarotsensor kann er seine Umgebung wahrnehmen und sich ihr anpassen. Da er sich nur auf zwei Rädern fortbewegt, nutzt er hierfür zusätzlich seine Funktion, sich selbstständig auszubalancieren.

- **Personenverfolgung:** Er kann Personen ohne direkte Steuerung verfolgen, zum Beispiel um Videos zu drehen oder Dinge zu transportieren.
- **Gesten- und Sprachsteuerung** Mit Gesten und Sprache kann der Loomo einfache Befehle entgegennehmen, zum Beispiel um Fotos zu schießen oder eine Videoaufnahme zu starten.
- **Entwicklungsplattform:** Das Loomo Software Development Kit (SDK) ist ein Entwicklungstool für den Loomo und wird von Segway-Ninebot bereitgestellt. Durch das SDK ist es möglich den Roboter mit benutzerdefinierten Sensoren oder Software-Lösungen zu erweitern, außerdem bietet sie eine Schnittstelle, welche Zugriff auf die Funktionen und die Hardware des Loomo erlaubt. Die Basis für den Loomo ist eine Android Applikation, mithilfe derer zum Beispiel auch ROS mit dem SDK integriert werden kann.

Als Mitglied der CaDS (Communication and Distributed Systems) Forschungsgruppe an der HAW Hamburg, die den Loomo im Kontext von ROS-basierten Projekten untersucht, bietet sich dieser Roboter als idealer Kandidat für die Evaluation verschiedener Middleware-Lösungen an.

In dieser Arbeit wurde der Loomo jedoch nicht direkt für die Tests des Vergleichs der Middleware-Lösungen verwendet. Stattdessen wurden die Tests auf allgemeingültigen Systemen durchgeführt, um hardwarebedingte Einschränkungen zu vermeiden. Außerdem können die Ergebnisse dadurch besser auf andere Roboterplattformen und autonome Systeme übertragen werden. Dennoch bieten die in dieser Arbeit erlangten Erkenntnisse wertvolle Hinweise darauf, wie Middleware-Systeme auf ressourcenbeschränkten Umgebungen wie dem Loomo eingesetzt werden können. Zukünftige Arbeiten könnten diese Annahmen durch Tests auf dem Loomo selbst weiter vertiefen und validieren.

## 1.3 Forschungsziele und Forschungsfragen

Das Ziel dieser Arbeit ist es, die Herausforderungen und Probleme bei der Verwendung von ROS in Robotikanwendungen zu identifizieren und zu analysieren. Die Implementierung von RoboLink, welche im dritten Kapitel näher erläutert wird, zielt darauf ab, diese bekannten Schwachstellen zu adressieren und eine Lösung zu bieten, welche die Zuverlässigkeit von Robotikanwendungen insgesamt verbessert. Für die Analyse zwischen

ROS und RoboLink werden bestimmte Aspekte der beiden Lösungen verglichen und diskutiert. Der Vergleich fokussiert sich auf die Untersuchung der Leistung, der Effizienz und der Benutzerfreundlichkeit jedes Ansatzes. Die folgenden Untersuchungen sind die Hauptziele dieser Forschung:

- **Leistungsbewertung:** Analyse der Umlaufzeit und des Durchsatzes beider Middleware-Ansätze in kontrollierter Umgebung.
- **Auswertung der Effizienz:** Beurteilung der Recheneffizienz und des Ressourcenverbrauchs von ROS und RoboLink unter unterschiedlicher Belastung.
- **Benutzerfreundlichkeitsanalyse:** Bewertung der Benutzerfreundlichkeit bei der Einrichtung, der Konfiguration und dem Einsatz, wobei Herausforderungen und Vorteile jedes Systems hervorgehoben werden.
- **Best Practices:** Bereitstellung von Empfehlungen für die Auswahl und Implementierung von Middleware-Lösungen, basierend auf empirischen Erkenntnissen.

## 1.4 Bedeutsamkeit der Studie

Diese Arbeit untersucht Herausforderungen und Verbesserungsmöglichkeiten von ROS, denn trotz der Weiterentwicklung in den letzten Jahren und der Einführung neuer Versionen, bleiben viele der grundlegenden Probleme weiterhin von Bedeutung. Im nächsten Kapitel werden die Probleme von ROS detailliert aufgeführt und analysiert. Die Implementierung von RoboLink bietet wertvolle Einblicke, wie alternative Middleware-Architekturen gestaltet werden können um die gefundenen Probleme zu vermeiden. Außerdem können Entwickler, durch ein tieferes Verständnis der Stärken und Limitationen von ROS, informierte Entscheidungen treffen um die Funktionalität ihrer Systeme zu verbessern.

Ein weiterer Aspekt ist die Analyse, wie die Optimierungen von RoboLink verwendet werden können, um die Effizienz in verteilten Systemen zu verbessern. Diese Analyse liefert Erkenntnisse, die sowohl relevant für ROS, als auch allgemeingültig interessant für die Entwicklung von Middleware-Architekturen sein können. Zusätzlich hebt diese Forschung die Herausforderungen bei der Entwicklung einer Middleware hervor und bietet wertvolle Erfahrungen für die breite Community von Entwicklern.

Darüber hinaus ist die Studie relevant für die CaDS Forschungsgruppe, da die vorgeschlagene Middleware möglicherweise als Ersatz oder Ergänzung zu ROS dienen könnte. Obwohl die RoboLink Middleware derzeit noch nicht vollständig für den Einsatz mit dem Loomo geeignet ist, bietet sie eine Grundlage für weitere Forschungsprojekte, die auf dieser Arbeit aufbauen und die Middleware weiterentwickeln können.

## 1.5 Struktur der Arbeit

Diese Arbeit ist in die folgenden Kapitel unterteilt:

- **Kapitel 2: Das Robot Operating System (ROS):** Dieses Kapitel bietet einen Überblick über ROS, dessen Funktionalität und Architektur und beschreibt die Unterschiede zwischen den ROS Versionen. Das Verständnis der Funktionsweise von ROS liefert die Basis für die Evaluierung im vierten Kapitel dieser Arbeit.
- **Kapitel 3: Die RoboLink Middleware:** In diesem Kapitel wird die für diese Forschung implementierte Middleware, RoboLink, vorgestellt. Es diskutiert die Gründe hinter den getroffenen Designentscheidungen und erklärt die Funktionsweise mit dem Loomo.
- **Kapitel 4: Vergleich von ROS und RoboLink:** Dieses Kapitel beschreibt die Vorgehensweise beim Vergleich von ROS und RoboLink, präsentiert die Ergebnisse der durchgeführten Tests und analysiert die entsprechenden Stärken und Limitationen.
- **Kapitel 5: Fazit und Ausblick:** Das letzte Kapitel fasst die wichtigsten Ergebnisse der Forschung zusammen, diskutiert die einhergehenden Implikationen und schlägt Richtlinien für zukünftige Forschung vor.

# 2 Das Robot Operating System (ROS)

In diesem Kapitel wird ROS vorgestellt und dessen wichtigste Eigenschaften erläutert. Außerdem wird die Architektur und Arbeitsweise, sowie deren Limitationen beleuchtet. Es wird über die verschiedenen Versionen und die Verwendung von ROS in dieser Arbeit aufgeklärt. Ziel ist es ein grundlegendes Verständnis für ROS aufzubauen um die Vergleichsanalyse und deren Ergebnisse im späteren Teil dieser Arbeit relevanter nachvollziehen zu können.

## 2.1 Grundlagen von ROS

ROS ist ein öffentliches Framework zur Entwicklung von Software für Roboter. Es findet Anwendung in jeglichen Bereichen, in denen Roboter zu finden sind. Ursprünglich wurde es im Jahr 2007 vom Stanford Artificial Intelligence Laboratory (SAIL) mit der Unterstützung des Stanford AI Robot Projektes entwickelt.[3] Mittlerweile ist das Framework weit verbreitet und hat eine große Community aufgebaut. Funktionen in ROS sind meist generalisiert für die Anwendung mit verschiedenen Robotern, vor allem wegen dieser Vorgehensweise wurde eine Vielzahl von Bibliotheken und Funktionen öffentlich bereitgestellt, die heute mit als Hauptgrund dafür gelten, eine Applikation basierend auf ROS zu entwickeln. Einige Beispiele solcher Bibliotheken und Funktionen sind zum Beispiel SLAM, Objekterkennung, Werkzeuge für 3D Visualisierung (RVIZ) und die Möglichkeit, Daten zu sammeln und offline erneut abzuspielen (rosbag). Regelmäßige Updates und die Unterstützung der großen Community, helfen Nutzern dabei ROS Applikationen zu schreiben, zu testen und einzusetzen. ROS-basierte Applikationen sind sprachunabhängig und können in verschiedenen, gängigen Programmiersprachen entwickelt werden, zum Beispiel C++, Python, Java und LISP [5].

ROS existiert zurzeit in zwei unterschiedlichen Versionen, ROS 1 und ROS 2, beide Versionen werden in diesem Kapitel noch genauer vorgestellt. ROS 1 ist die Basis vieler

bestehender Anwendungen, während ROS 2 wiederum der logische Nachfolger ist und eine moderne Art der Kommunikation bietet. Der Schwerpunkt dieser Arbeit liegt allerdings bei ROS 1, da diese Version zurzeit von der CaDS Gruppe an der HAW Hamburg als Umgebung für den Loomo verwendet wird. Im Folgenden wird daher der Begriff ROS ausschließlich für ROS 1 verwendet, wenn auf ROS 2 verwiesen werden soll, wird dies explizit angegeben.

## 2.2 Architektur und Komponenten von ROS

Die Komplexität einer ROS Applikation ist in verschiedene Knoten unterteilt. Typischerweise wird nach dem Designprinzip Separation of Concerns gearbeitet, so dass ein Knoten nur für eine spezifische Aufgabe zuständig ist. Jeder von ihnen läuft in einem eigenen Prozess und kann daher auch auf einer separaten Maschine ausgeführt werden. Die Knoten kommunizieren miteinander meist über ein Publish/Subscribe-Modell, welches ihnen erlaubt asynchron Daten zu übermitteln. Hierfür bietet ROS Topics, auf welche die Knoten Nachrichten senden (Publish) oder sich registrieren (Subscribe) können, um Nachrichten zu empfangen. Ein weiteres mögliches Modell basiert auf Request/Reply, dies ist eine synchrone Datenübertragung mithilfe von Services. Hierbei werden Daten von einem Knoten bei einem weiteren Knoten angefragt (Request), dieser verarbeitet die Anfrage und schickt eine Antwort zurück (Reply). Unabhängig vom Kommunikationsmodell ist der ROS Master ein zentraler Bestandteil eines ROS Systems. Bei ihm können Knoten ihre Topics und Services registrieren, damit andere Knoten diese wiederum abfragen können. Wichtig ist zu bemerken, dass die tatsächliche Kommunikation, unabhängig vom Modell, zwischen den Knoten direkt ist, also Nachrichten direkt von einem Knoten zum anderen übertragen werden. Zusätzlich bietet der ROS Master die Möglichkeit Parameter als Key/Value entgegenzunehmen und zur Laufzeit bereitzustellen. Dies ist hilfreich um Konfigurationen zentral zu speichern und im System verteilen zu können.[2]

## 2.3 Technische Limitationen von ROS

Auch wenn ROS in vielen Roboter-Applikationen eingesetzt wird, weist das Framework in einigen Bereichen Einschränkungen auf. Die Kommunikation über verlustbehaftete Verbindungen wie zum Beispiel WLAN oder Satellitenverbindungen können von ROS

teilweise nicht lückenlos übertragen werden. Außerdem gibt es keine eingebauten Sicherheitsmechanismen. Ein weiterer Nachteil ist der ROS Master, durch ihn entsteht im System ein Single Point of Failure. Das bedeutet, wenn der ROS Master aus irgendeinem Grund nicht erreichbar sein sollte, kann das System nicht mehr zuverlässig Nachrichten übertragen, in diesem Fall muss das gesamte System oft vollständig rekonstruiert werden. Das gesamte System ist darauf angewiesen, dass der ROS Master zu jeder Zeit erreichbar ist.[7, S. 330] Mit zusätzlichen Erweiterungen, die von der ROS Community entwickelt wurden, existieren Versuche diese Probleme zu lösen. In nahezu allen Fällen mussten jedoch durch Limitierungen in der Architektur oder andere technische Gründe Kompromisse eingegangen werden. Diese Erweiterungen haben dazu beigetragen, dass die Nutzbarkeit von ROS verbessert, beziehungsweise verlängert wurde, konnten die Kernprobleme allerdings nicht lösen. [1]

Ein weiterer Punkt sind die Möglichkeiten der Kommunikation, in ROS stehen ausschließlich die beiden oben vorgestellten Kommunikationsvarianten zur Verfügung. Durch das Publish/Subscribe-Modell ist eine asynchrone, unidirektionale Kommunikationsart gegeben. Das bedeutet der Publisher wartet nach dem Senden der Nachricht nicht auf eine Antwort und der Nachrichtenfluss ist einseitig, da nur Anfragen und keine Antworten gesendet werden. Anders ist es beim Request/Reply-Modell, bei welchem der Anfragende blockiert, bis er eine Antwort erhalten hat, dementsprechend werden auch Nachrichten in beide Richtungen versendet, damit bietet ROS auch eine synchrone, bidirektionale Kommunikationsart. Um die Skalierbarkeit in einem verteilten System zu gewährleisten, kann es sinnvoll sein ein Modell zu verwenden, das nicht blockiert, wenn eine Nachricht versendet wird aber dennoch eine Antwort zurücksendet, also eine asynchrone, bidirektionale Kommunikationsart. Vor allem wenn Nachrichten über weite Entfernungen versendet werden, kann ein System mit einem solchen Modell deutlich effizienter arbeiten, da Wartezeiten verringert werden.[7, S. 9-13] Um einen Teil dieses Problems zu umgehen, kann das Senden der Nachricht vom aufrufenden Thread entkoppelt werden. Dadurch können Nachrichten zwar parallel versendet werden, allerdings muss für jede Nachricht ein Thread existieren, der auf die Antwort wartet. Je nach Programmiersprache unterscheiden sich die Implementierungen von ROS, während in Java (ROSJava) das Request/Reply-Modell automatisch asynchron ausgelagert wird, muss dies in C++ (roscpp) selbst übernommen werden. Da jedoch bei dieser Variante für jede Nachricht ein Thread auf die Antwort warten muss, ist die Skalierbarkeit hierbei fraglich.

Um mit ROS Nachrichten zu versenden, die über die standardisierten Datentypen hinausgehen, muss eine Datei angelegt werden, welche die Nachricht definiert. Anschließend

muss diese Datei kompiliert werden, dabei werden automatisch passende Dateien für die jeweilig verwendete Programmiersprache generiert. Mithilfe dieser generierten Dateien kann die benutzerdefinierte Nachricht in die Implementierung eingefügt werden, dabei muss beachtet werden, dass diese Dateien allen Knoten zur Verfügung stehen müssen, die an der Kommunikation beteiligt sind. Zusätzlich müssen Nachrichten,beziehungsweise Services, für das Publish/Subscribe- und das Request/Reply-Modell separat definiert werden. Dieser Vorgang ist aufwendig und benötigt für jeden neuen Nachrichtentyp eine neue Konfiguration.

## 2.4 Weiterentwicklung von ROS: ROS 2

ROS 2 wurde entwickelt, um einige Limitierungen und Herausforderungen von ROS 1 zu adressieren. Die zentrale Master-Architektur entfällt und wird ersetzt durch ein Data Distribution Service (DDS), einem anerkannten Standard für Echtzeitkommunikation. Diese Umstellung ermöglicht es, ohne zentralen Knoten auszukommen, was die Fehlertoleranz und Skalierbarkeit deutlich verbessert. Mit dieser Änderung eignet sich ROS 2 für Echtzeit- und verteilte Systeme, die in sicherheitskritischen Anwendungen wie autonomen Fahrzeugen und industriellen Steuerungssystemen eingesetzt werden. Außerdem wird ROS 2 auf einer Vielzahl von Plattformen unterstützt, einschließlich Linux, Windows, macOS und Echtzeitbetriebssystemen. Die allgemeine Leistung und der Ressourcenverbrauch sind ebenfalls optimiert. Eine Aktualisierung von ROS 1 auf ROS 2 bedeutet jedoch für die meisten Applikationen eine immense Anzahl an Änderungen, da sich die Schnittstelle und die Funktionsweise der Middleware in ihrem Kern wesentlich verändert hat.[4]

## 2.5 Fazit zur Analyse von ROS

In diesem Kapitel wurde die wesentliche Funktionsweise und Architektur von ROS vorgestellt. Im Folgenden werden die wichtigsten Punkte noch einmal zusammengetragen. ROS benötigt einen zentralen Master-Knoten, welcher eine Schwachstelle für das gesamte System bedeutet. Zudem unterstützt es keine asynchrone, bidirektionale Kommunikation auf inhärente Weise, was bestimmte Kommunikationsmuster einschränkt. Benutzerdefinierte Nachrichten müssen kompiliert und auf allen beteiligten Knoten vorkonfiguriert werden. Außerdem ist die Nachrichtenübertragung über verlustbehaftete Verbindungen nicht immer fehlerfrei. Der Wechsel zu ROS2 ist jedoch aufwendig, da es erhebliche Änderungen in der Schnittstelle mit sich bringt. Trotzdem bietet ROS den Vorteil eines umfangreichen Zugriffs auf eine Vielzahl von Bibliotheken und Funktionen, die über viele Jahre hinweg entwickelt wurden und für viele Anwendungen unverzichtbar sind. Im nächsten Kapitel wird eine alternative Middleware-Lösung vorgestellt, welche versucht die Schwachstellen von ROS zu adressieren.

# 3 Die RoboLink Middleware

## 3.1 Einführung in RoboLink

In diesem Kapitel wird die benutzerdefinierte Middleware RoboLink vorgestellt, die als universelle Bibliothek für die Kommunikation in verteilten Systemen entwickelt wurde. Bei der Entwicklung von RoboLink wurden verschiedene Schwachstellen von ROS adressiert und damit Verbesserungsmöglichkeiten für Middleware-Systeme vorgeschlagen, außerdem wurde diese mit Hinblick auf den Loomo optimiert. Zur Veranschaulichung der Funktionalität wurde zusätzlich eine Beispielanwendung, der Connector4loomo, entwickelt. Diese dient dazu die Integration von RoboLink mit dem spezifischen Anwendungsfall des Loomo zu demonstrieren. Der Connector4loomo ist hierbei keine vollständige Lösung zur Verwendung mit dem Loomo, sondern lediglich ein Lösungsansatz, um eine grundlegendes Framework zur Verfügung zu stellen, die Funktion mit dem Loomo wurde allerdings über mehrere Tests sichergestellt.

## 3.2 Architektur von RoboLink

RoboLink ist eine dezentrale Middleware, die mit dem Prinzip des Remote Procedure Call (RPC) arbeitet, das in diesem Kapitel noch ausführlich ausgeführt wird. RoboLink ist als Java-Bibliothek implementiert und stellt eine flexible und skalierbare Infrastruktur für die Kommunikation in verteilten Systemen bereit. RoboLink ist sprachunabhängig konzipiert, das bedeutet die Kommunikationsschicht ist so konstruiert, dass die Implementierung eines Gegenstücks für die Middleware in einer anderen Programmiersprache möglich ist, ohne den bestehenden Quellcode zu verändern.

Die RoboLink Middleware ist in Java implementiert, durch den Fakt, dass die Infrastruktur des Loomo auf einer Android-Applikation basiert und das Android SDK Java

offiziell als native Sprache unterstützt, ist die Integration von RoboLink und dem Loomo besonders einfach. Des weiteren ist Java plattformunabhängig, daher kann RoboLink auch für andere autonome Roboter, auf anderen Plattformen eingesetzt werden. Zusätzlich ist Java kompatibel mit Kotlin, bietet eine große Menge an Bibliotheken und eine große Community. Diese Gründe sind ausschlaggebend für die Auswahl von Java als Programmiersprache für RoboLink.

Im Gegensatz zu ROS ist RoboLink dezentral entworfen, daher wird kein zentraler Knoten benötigt, der die Kommunikation zwischen den Knoten orchestriert. Dies verhindert die Entstehung eines Single Point of Failure in der Middleware. Stattdessen sind alle Knoten zum Start des Programms vorkonfiguriert, das bedeutet alle Knoten und deren Adressen sind allen anderen Knoten schon vorab bekannt. Diese Entscheidung wurde getroffen, um die Komplexität der dynamischen Registrierung von Knoten zu vermeiden; diese kann in einem dezentralen System eine erhebliche Herausforderung darstellen. Ein Mechanismus, der automatisch die Entdeckung und Verwaltung von Knoten im System übernimmt, hätte den Implementierungsaufwand deutlich erhöht. Durch diese Vorkonfiguration ist sichergestellt, dass die Kommunikation im System reibungslos ablaufen kann.

Ein weiterer wichtiger Aspekt ist die entkoppelte Nachrichtenübertragung. Eingehende und ausgehende Nachrichten werden nicht sofort versendet, sondern jeweils in einer Queue abgelegt. Diese Nachrichten werden von der Middleware in einem eigenen Thread aus der Queue genommen und verschickt. Durch dieses Prinzip wird die Nachrichtenübertragung vom Rest des Systems entkoppelt. Das hat verschiedene Vorteile, zum einen wird die Nachricht über die Queue an einen anderen Thread übergeben, der aufrufende Thread wird dadurch schneller frei um Aufgaben der Applikationsebene zu übernehmen. Zum anderen können hoch konzentrierte Nachrichtenmengen besser abgefangen werden, denn in der Realität werden Nachrichten in verteilten Systemen meist nicht immer gleich verteilt gesendet, sondern Nachrichten treten gebündelt auf. Ohne eine Queue könnte es vorkommen, dass Nachrichten nicht schnell genug verarbeitet werden und daher je nach verwendetem Transportprotokoll Paketverluste oder Verzögerungen bei der Übertragung auftreten. Die maximale Größe einer Queue ist konfigurierbar, wenn eine Queue voll ist und eine neue Nachricht hinzugefügt werden soll, wird die älteste Nachricht entfernt.[6][8]

Vor der Übertragung einer Nachricht mit RoboLink, wird die Nachricht serialisiert, ebenso wird eine Nachricht nach dem Erhalt deserialisiert. Durch diesen Vorgang werden die Daten standardisiert, sodass sie von verschiedenen Systemen und Plattformen verwen-

det werden können. Außerdem hilft die Serialisierung dabei die Datenmenge in einer Nachricht zu verringern, was wiederum den Transfer über das Netzwerk optimiert. Für RoboLink wird eine Serialisierung mithilfe von MessagePack verwendet. MessagePack ist ein effizientes, binäres Format zur Serialisierung und verfügbar für viele gängige Programmiersprachen. Im Gegensatz zu ROS können benutzerdefinierte Objekte ohne Vorkompilierung übertragen werden. Das Objekt muss hierfür lediglich auf beiden Knoten identisch definiert sein, um eine korrekte Deserialisierung zu gewährleisten. Durch die Sprachunabhängigkeit von MessagePack besteht die Möglichkeit ein Gegenstück für RoboLink in einer anderen Programmiersprache zu implementieren.

RoboLink bietet verschiedene Kommunikationsarten und verwendet dafür zwei verschiedene Transportprotokolle, das Transport Control Protocol (TCP) und das User Datagram Protocol (UDP). UDP ist ein verbindungsloses Protokoll, Nachrichten werden sofort versendet und bekommen keine Bestätigung für das erfolgreiche Erhalten zurück. Im Gegensatz dazu steht TCP, dieses Protokoll baut für die Kommunikation zuerst eine Verbindung zum anderen Endpunkt auf, Nachrichten werden im Anschluss über diese Verbindung versendet. Um TCP in einer Middleware effizient einzusetzen, muss eine aufgebaute Verbindung wiederverwendet werden, denn der Neuaufbau von Verbindungen kann sehr teuer sein.[7, S. 38] RoboLink baut eine Verbindung zu jedem anderen Knoten auf, sobald das erste mal eine Nachricht dorthin geschickt wird, diese Verbindungen werden aufrechterhalten, solange beide Knoten gegenseitig erreichbar bleiben.

### 3.2.1 Implementierung von Remote Procedure Calls in RoboLink

Einfach erklärt, ist ein RPC der Aufruf einer Funktion auf einem nicht-lokalen Knoten. Für den Entwickler soll der Aufruf dieser Methode so wirken, als wäre er ein normaler, lokaler Methodenaufruf. Durch das Prinzip des RPC wird die Komplexität der Nachrichtenübertragung komplett in der Middleware gekapselt. Um dies zu verwirklichen wird dem Entwickler ein Stub zur Verfügung gestellt, dieser definiert die Methoden, welche auf dem anderen Knoten zur Verfügung stehen. Wenn eine Funktion über diesen Stub aufgerufen wird, ist die Middleware dafür verantwortlich den Funktionsaufruf mit den dazugehörigen Parametern zum anderen Knoten zu übermitteln. Auf diesem Knoten existiert ein Serverstub, auch Skeleton genannt, welcher als Gegenstück für den Stub auf dem ersten Knoten fungiert. Die Funktion wird über den Skeleton aufgerufen und ein möglicher Rückgabewert wird über die Middleware zurück an den ersten Knoten gesendet und dort an den Entwickler zurückgegeben.[7, S. 126-135]

RoboLink verwendet ausschließlich das RPC Prinzip, allerdings ist es möglich einen RPC auf verschiedene Weise zu übertragen. Hierfür sind in RoboLink drei verschiedene Strategien implementiert:

- **Request/Response**: Die Request/Response-Strategie ist synchron, das bedeutet sie blockiert, bis eine Antwort zurückgesendet wurde, das unterliegend verwendete Transportprotokoll ist TCP. Der Einsatzbereich für diese Strategie sind Fälle, in denen die Applikation eine Antwort benötigt um weiterzuarbeiten. Beispiele für die Verwendung dieser Strategie sind Abfragen von Sensordaten oder eines Status.
- **Request/Future**: Die Request/Future-Strategie ist asynchron, wenn eine Nachricht mit dieser Strategie versendet wird, gibt die Middleware umgehend ein Future Objekt zurück. Dieses Objekt wird mit der Antwort angereichert sobald diese zurückgesendet und erhalten wurde. Um die Antwort zu erhalten, kann beim Future Objekt zum Beispiel zu einem beliebigen Zeitpunkt angefragt werden, ob die Antwort bereits vorhanden ist. Auch diese Strategie verwendet TCP um sicherzustellen, dass die Nachrichten korrekt übertragen werden. Eingesetzt wird die Strategie wenn die Applikation die Antwort nicht sofort benötigt um weiterzuarbeiten. Hierdurch können Aufgaben parallel abgearbeitet und dadurch der Durchsatz erhöht werden. Ein Beispiel für die Anwendung sind Operationen die sehr viel Zeit in Anspruch nehmen, wie die Verarbeitung von Videodaten oder Anfragen über weite Entfernungen.
- **Request/No Response**: Die Request/No Response-Strategie ist unidirektional, es wird nur eine Nachricht versendet aber keine Antwort zurückgegeben. Für diese Art der Kommunikation wird das UDP Protokoll verwendet, daher ist es möglich, dass Nachrichten verloren gehen nachdem sie versendet wurden. Sie kommt zum Einsatz wenn die Übertragung einer Nachricht nicht sicherheitsrelevant ist und keine Antwort erforderlich ist. Die Verwendung dieser Strategie ist zum Beispiel sinnvoll um einen Datenstrom zu senden, bei dem Verluste akzeptabel sind, wie wiederholte Bewegungskommandos eines Roboters oder eine regelmäßige Aktualisierung von nicht-kritischen Sensordaten.

Das Aufrufen der Funktionen im Skeleton ist mithilfe von Java-Reflection implementiert. Die Funktionen werden durch den Methodennamen und die verschiedenen Typen der Parameter zugeordnet. Durch einen Caching-Mechanismus werden bereits aufgerufene Methoden zwischengespeichert um die Ausführungszeit zu verkürzen. Mit der Remote

Method Invocation (RMI), bietet Java einen eigenen Mechanismus um RPC zu ermöglichen. Dieser wurde bei RoboLink nicht verwendet, da RMI nicht direkt kompatibel mit anderen Programmiersprachen ist, damit wäre eine Erweiterung der Middleware auf andere Programmiersprachen nicht mehr ohne weiteres möglich.

## 3.3 Spezifische Anpassungen von RoboLink für den Loomo

Die Loomo Applikation basiert auf Android, die passende Java Version für die Entwicklung mit dem Loomo ist Java 8, daher wurde auch RoboLink mit dieser Version implementiert. Für die Steuerung des Loomo muss RoboLink in der Lage sein, Sensordaten effizient zu übertragen. Vor allem beim Versenden von Kameradaten des Loomo können große Datenpakete anfallen. In vielen Fällen müssen regelmäßig und in kurzer Zeit verschiedene Sensoren abgefragt und transferiert werden. RoboLink bietet durch die Möglichkeit der asynchronen Verarbeitung von Nachrichten ein optimales Modell, um Sensorergebnisse effizient abzurufen. Ebenfalls kann die Übertragung von Sensordaten über UDP durchgeführt werden, dadurch kann der Overhead bei der Nachrichtenübertragung für nicht-kritische Sensordaten minimiert werden. Außerdem kann die Anzahl der Threads für die Nachrichtenverarbeitung angepasst werden, um die Middleware optimal zu skalieren, die Hardware des Loomo ist begrenzt, daher ist es wichtig dessen Ressourcen minimal auszulasten. Ebenso kann für jede Queue die Menge der maximalen Elemente gesetzt werden, um die benötigte Speichermenge zu reduzieren. Diese Optimierungen von RoboLink tragen dazu bei, dass die Middleware effizient mit dem Loomo zusammenarbeiten kann.

## 3.4 Entwicklung und Einsatz des Connector4loomo

RoboLink alleine ist nur eine Bibliothek und noch keine Anwendung, um die Middleware zu testen muss eine Beispielimplementierung entwickelt werden. Hier kommt der Connector4loomo zum Einsatz. Die Implementierung setzt die Basis für eine Applikation zur Steuerung spezifisch für den Ninebot Loomo. Sie zeigt gleichzeitig auch, wie RoboLink für verschiedene andere Roboterplattformen integriert werden kann. Mithilfe des Connector4loomo wurden im nächsten Kapitel Tests durchgeführt, um verschiedene Aspekte der Implementierung zu testen.

## 3.5 Fazit zur RoboLink Middleware

In diesem Kapitel wurde eine selbst entwickelte Middleware-Bibliothek, RoboLink, vorgestellt. Die Middleware basiert auf dem RPC Mechanismus und bietet synchrone, asynchrone und unidirektionale Kommunikationsmöglichkeiten, um eine flexible Lösung für autonome Roboter zu bieten. RoboLink wurde mit Blick auf erforschte Limitationen im Umgang mit ROS entwickelt und die Architektur versucht dessen Schwächen zu adressieren. Der Connector4loomo verwendet RoboLink als Middleware zur Kommunikation zwischen dem Loomo und einem steuernden System. Im nächsten Kapitel wird der Connector4loomo verwendet um bestimmte Aspekte von RoboLink zu evaluieren und mit ROS zu vergleichen.

# 4 Vergleich von ROS und RoboLink

In diesem Kapitel werden die beiden Middleware-Lösungen, ROS und RoboLink verglichen. Hierfür wurde eine Reihe von spezifisch definierten Tests durchgeführt, welche die Leistung bewerten und funktionelle Aspekte der Implementationen beleuchten sollen. Jeder dieser Tests soll tiefere Einblicke bringen, die dabei helfen, beide Lösungen einzuschätzen und bewerten zu können. Die untersuchten Aspekte in diesem Kapitel sind die Umlaufzeit, der Durchsatz, die Effizienz und die Benutzerfreundlichkeit. Im Folgenden wird zuerst das Vorgehen und der Aufbau der Tests vorgestellt und anschließend deren Ergebnisse diskutiert. Der Connector4loomo wird eingesetzt um die Tests für die RoboLink Middleware durchzuführen, im Folgenden bezieht sich der Vergleich allerdings auf RoboLink und nicht auf die spezifische Implementierung. Um die Vergleichbarkeit nicht zu beeinträchtigen, wurde für die Implementierung von ROS ebenfalls Java verwendet, damit beide Middleware-Lösungen unter den gleichen Bedingungen getestet werden können.

## 4.1 Testaufbau und Durchführung

Zur Analyse der zwei Middleware-Lösungen, wurden mehrere individuelle Tests entwickelt. Die Tests wurden nicht auf dem Loomo, sondern auf zwei unabhängigen Systemen durchgeführt. Dies hat mehrere Hintergründe, zum Einen wird so die Leistungscharakteristik von den Loomo spezifischen Faktoren, wie den Hardwarespezifikationen und dem Android Betriebssystem, isoliert. Zum anderen sind die Ergebnisse von universellen Maschinen einfacher zu generalisieren und auf verschiedene Roboterplattformen und Applikationen anwendbar. Aus diesen Gründen ist die Testumgebung ohne den Loomo relevanter für zukünftige Forschung und Entwicklung. In Tabelle 4.1 sind die Hardwarespezifikationen der verwendeten Testsysteme aufgelistet.

| Hardware | Testsystem 1 | Testsystem 2 |
|---|---|---|
| Prozessor | Intel Core i5-8400, 2.8GHz, 64-bit | Intel Core i7-1165G7, 2.8GHz, 64-bit |
| Arbeitsspeicher | 16GB DDR4 2400 MHz | 32GB DDR4 3200MHz |
| Grafikkarte | NVIDIA GeForce GTX 1060 6GB | Intel Tigerlake-LP GT2 |
| Festplatte | Crucial P3 SSD 1TB, M.2 NVMe | SK Hynix PC711 1TB, M.2 NVMe |
| Netzwerk | Gigabit Ethernet (1Gb/s) | Gigabit Ethernet (1Gb/s) |
| Betriebssystem | Windows 10, 22H2 | Ubuntu 20.04 LTS, Kernel 5.15 |

Tabelle 4.1: Hardwarespezifikationen

### 4.1.1 Test 1: Umlaufzeit

*Zielsetzung:* Messung der Zeit, die benötigt wird um eine Nachricht zwischen zwei Knoten hin- und wieder zurück zu senden. *Vorbereitung:* Ursprünglich sollte dieser Test die Latenz der Kommunikation zwischen zwei Knoten messen, hierfür müssen allerdings einige Dinge beachtet werden. Um eine korrekte Zeitmessung durchzuführen, wurden die Systeme zunächst mit dem Network Time Protocol (NTP) synchronisiert. Da diese Methode zu ungenau ausfiel, wurde ein weiterer Versuch unternommen, die Zeit über einen externen Server abzugleichen. Auch dieser Ansatz erwies sich als unzureichend. Um dennoch einen aussagekräftigen Test durchzuführen, wurde daher für jede Nachricht eine Antwort zum ersten Knoten zurück gesendet um damit die Umlaufzeit zu bestimmen. Die Zeit wurde an zwei Zeitpunkten auf dem ersten Knoten gemessen, das erste mal, kurz bevor die Anfrage versendet wurde und ein weiteres mal nach der Ankunft der Antwort. Ein weiterer Punkt, den es zu beachten gilt, ist die Reihenfolge der Nachrichtenübertragung. Nicht jede der Kommunikationsmöglichkeiten stellt sicher, dass Nachrichten in der gleichen Reihenfolge beim Empfänger ankommen, wie sie beim Sender losgeschickt wurden. Daher wurde jeder Nachricht eine eindeutige ID beigefügt, mithilfe derer das Paar aus Anfrage und Antwort einander zugeordnet werden konnte. Ein ebenfalls wichtiger Aspekt ist die Größe, also die Anzahl von Bytes, in den verwendeten Nachrichten. Um die Umlaufzeit umfänglich zu testen, wurde den Nachrichten ein Payload mit variabler Größe hinzugefügt. In realen Bedingungen werden große Datenmengen meist nur in eine Richtung versendet, daher beinhaltet in diesem Test auch nur die Anfrage einen zusätzlichen Payload, die Antwort wurde nicht mit zusätzlichen Daten angereichert. Alle Tests wurden über ein Local Area Network (LAN) durchgeführt.

- **ROS Middleware:** Für die Verwendung von ROS wurde das Ubuntu-Testsystem mit einer ROS Noetic Umgebung ausgestattet, diese ist notwendig um den ROS

Master zu starten und eigene Nachrichten bzw. Services erstellen zu können. Zusätzlich wurde auf beiden Systemen ein ROS Knoten gestartet, für diese wurde jeweils ein Publisher und ein Subscriber implementiert, sodass Nachrichten über Topics ausgetauscht werden konnten. Zusätzlich wurde eine konforme Implementierung für das Request/Reply-Modell aufgesetzt, der zweite Knoten definiert einen Service-Server und der erste Knoten verwendet einen Service-Client um den Server abfragen zu können.

- **RoboLink:** Der Testaufbau für RoboLink benötigt lediglich eine Java Umgebung. Wie auch bei ROS stellt RoboLink verschiedene Modelle mit unterschiedlichen Kommunikationseigenschaften bereit, für jedes dieser Modelle wurde der Test für die Umlaufzeit einzeln durchgeführt. Der Stub und der Skeleton des Connector4loomo wurde um entsprechende Testmethoden angereichert. Da das Request/No Response-Modell unidirektional kommuniziert, musste bei dieser Variante zusätzlich ein RPC in die andere Richtung aufgerufen werden um eine Antwort senden zu können. Für den Test wurde die Standardkonfiguration genutzt, die der Connector4loomo zur Verfügung stellt.

*Durchführung:* Für den Umlauftest wurden mehrere Durchgänge mit kontinuierlich steigender Nachrichtengröße von $10^2$ bis $10^7$ Byte durchgeführt, um die Auswirkungen der verschiedenen Größen auf die Umlaufzeit zu analysieren. Pro Test wurden 1000 Nachrichten versendet, wobei vor dem Versenden einer neuen Nachricht mit ausreichend Wartezeit sichergestellt wurde, dass die Verarbeitung der vorherigen Nachricht vollständig abgeschlossen ist. Dadurch wurde sichergestellt, dass nur eine Nachricht auf einmal von der Middleware verarbeitet wurde. Die Java Virtual Machine (JVM) alloziert bei Applikationsbeginn eine bestimmte Menge des Arbeitsspeichers (RAM), für den Umlauftest wurde dieses maximale Limit auf 10 GB für alle Systeme festgelegt. Beim Senden einer hohen Anzahl großer Nachrichten, kann die Auslastung des Arbeitsspeichers signifikant ansteigen, daher wurde das Limit für die JVM erhöht um Abstürzen vorzubeugen und sicherzustellen, dass genügend Speicher für die Nachrichten verfügbar ist.

### 4.1.2 Test 2: Durchsatz

*Zielsetzung:* Messung wie viele Nachrichten pro Sekunde von einem Knoten zu einem anderen hin- und zurückgeschickt werden können. *Vorbereitung:* Für den Durchsatztest wurde der gleiche Versuchsaufbau verwendet wie für die Umlaufzeit. Die Unterscheidung

liegt lediglich in der Durchführung. *Durchführung:* Der Durchsatztest wurde ebenfalls für Nachrichtengrößen von $10^2$ bis $10^7$ Byte durchgeführt, ebenso wurden der JVM die gleichen Argumente gesetzt. Für jeden Test wurden 10.000 Nachrichten ohne Wartezeit, also direkt nacheinander, verschickt, die Zeit wurde gemessen bevor die erste Nachricht versendet und nachdem die letzte Nachricht vollständig verarbeitet war. Zur Errechnung des Durchsatzes wurde die Anzahl der Nachrichten durch die gemessene Zeit geteilt.

### 4.1.3 Test 3: Recheneffizienz

*Zielsetzung:* Auswertung der verwendeten Ressourcen der Central Processing Unit (CPU) und des Random Access Memory (RAM) während der Verwendung der Middleware. *Vorbereitung:* Beide Testsysteme wurden mit dem Tool VisualVM ausgestattet, mit diesem Tool können Java Applikationen zur Laufzeit analysiert und profiliert werden. Es eignet sich hervorragend um die CPU Auslastung und die Verwendung des RAM über die Zeit eines Applikationslaufs zu überwachen. *Durchführung:* Die Verwendung der Systemressourcen wurde auf beiden Systemen gemessen, während erneut Nachrichten mit Payload-Größen von $10^2$ bis $10^7$ Byte versendet wurden. Zwischen dem Versenden der Nachrichten wurde keine Wartezeit eingefügt, daher wurden Nachrichten mit der größtmöglichen Geschwindigkeit versendet. Die JVM wurde erneut mit einem erhöhten RAM-Limit von 10 GB parametrisiert. Die ersten 10 Sekunden nach dem Starten jedes Tests wurden in den Ergebnissen herausgefiltert, da der Aufwand der Initialisierung nicht in die Ergebnisse einfließen sollte. Anschließend wurde eine Minute lang die Auslastung der CPU und des RAM aufgezeichnet.

### 4.1.4 Test 4: Benutzerfreundlichkeit

*Zielsetzung:* Vergleich des Aufwands der erstmaligen Einrichtung, der Konfiguration und der Bereitstellung, mit einem Fokus auf die Benutzerfreundlichkeit in der Praxis. *Durchführung:*

- **ROS Middleware:** Die Einrichtung von ROS, insbesondere ROS Java Noetic, stellte mehrere Herausforderungen dar. Zum einen war Ubuntu 20.04 das einzig unterstützte Betriebssystem und selbst diese Unterstützung war nicht offiziell, sondern nur durch die Community bestätigt. Eine weitere Hürde war die Verwendung

von benutzerdefinierten Nachrichten. Die notwendige Maven-Abhängigkeit um eigene Nachrichten zu erstellen war im ROS Java Paket zwar enthalten, aber nicht funktionsfähig. Mehrere Versuche die fehlenden Pakete mit älteren Versionen manuell zu überschreiben sind gescheitert. Um die Testfähigkeit dennoch sicherzustellen, wurden daher einzig vordefinierte Standardnachrichten verwendet. Die Einrichtung des ROS Services fügte dem Prozess weitere Komplexität hinzu, da für alle Systeme über einen aufwendigen Prozess, Dateien kompiliert werden mussten. Zusätzlich zu den Schwierigkeiten der Installation benötigte die Verwendung von ROS weitere Recherche. Die Dokumentation war an vielen Stellen lückenhaft oder veraltet und es musste auf Informationen der Community zurückgegriffen werden, um grundlegende Probleme der Einrichtung und Konfiguration zu lösen.

- **RoboLink:** Die Benutzerfreundlichkeit von RoboLink wurde anhand der Connector4loomo Implementierung getestet, die als Beispiel für den Einsatz von RoboLink mit dem Loomo dient. Die Installation und Konfiguration von RoboLink erwiesen sich im Vergleich zu ROS als erheblich einfacher. Durch die Java-basierte Umgebung, war die Einrichtung auf verschiedenen Betriebssystemen unkompliziert. Die gesamte Applikation ist dokumentiert und eine Anleitung zur Verwendung, sowie eine Beispielimplementierung, sind vorhanden.

## 4.2 Ergebnisse und Analyse

### 4.2.1 Umlaufzeit

Abbildung 4.1 zeigt die Umlaufzeit pro Nachrichtengröße für die unterschiedlichen Kommunikationsvarianten der Middleware Lösungen. Die Request/No Response-Strategie von RoboLink verwendet das UDP-Protokoll, welches bei der Nachrichtengröße auf 65.536 Byte beschränkt ist, in diesem Diagramm sind daher keine Daten dieser Strategie für Nachrichten von 100 KB oder größer vorhanden.

Für kleine Nachrichtengrößen, von 100B bis 1KB, liegen alle Varianten mit Werten von 0,4 ms bis 4 ms, sehr nah beieinander. Ab einer Nachrichtengröße von 100 KB steigt die Umlaufzeit der beiden ROS-Varianten gegenüber den RoboLink-Varianten deutlich an, der Mittelwert beträgt an dieser Stelle bereits mehr als das Doppelte. Die Spreizung der gemessenen Werte ist ebenfalls deutlich weiter, was auf eine Instabilität bei der Übertragung hindeutet. Bei Nachrichten von 1 MB ist der Mittelwert der ROS-Varianten

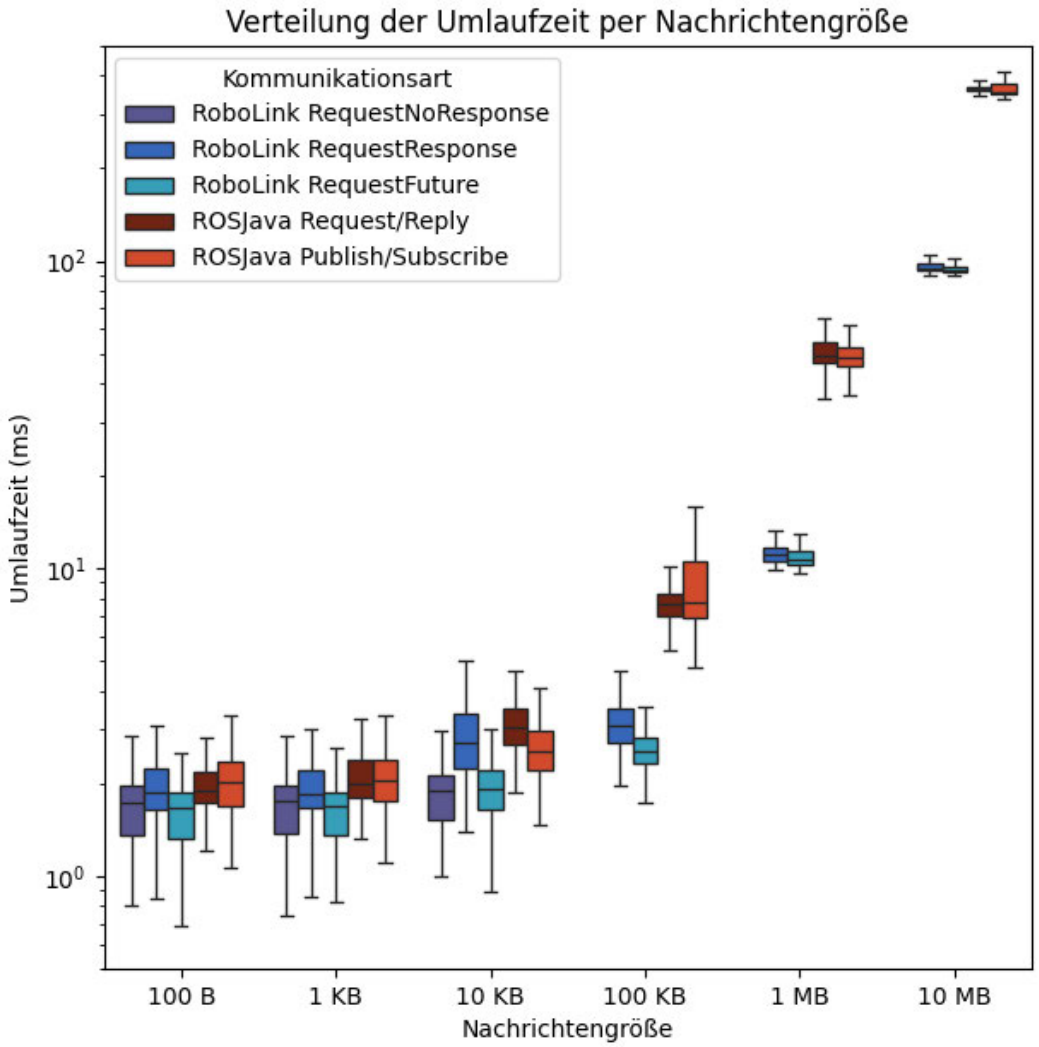


Abbildung 4.1: Umlaufzeit Testergebnis

bereits auf das dreifache gegenüber den RoboLink-Varianten angestiegen. Auch für 10 MB Nachrichten ist die Umlaufzeit für die Varianten von ROS im Mittel etwa drei bis vier mal höher. Zusätzlich konnte bei dem Publish/Subscribe-Modell von ROS für den Test mit 10 MB Nachrichtengröße nur etwa die Hälfte der Nachrichten übertragen werden, bevor der Arbeitsspeicher der Anwendung vollständig aufgebraucht war und die Applikation abgestürzt ist. Mehrere Wiederholungen des Tests zeigten dasselbe Ergebnis. Insgesamt lässt sich zusammenfassen, dass beide Middleware-Lösungen effizient mit Nachrichtengrößen von 100 B bis 10KB umgehen können, RoboLink jedoch deutlich bes-

ser mit größeren Nachrichten zurechtkommt. Auch die Stabilität der Umlaufzeit bleibt bei RoboLink merklich konstanter.

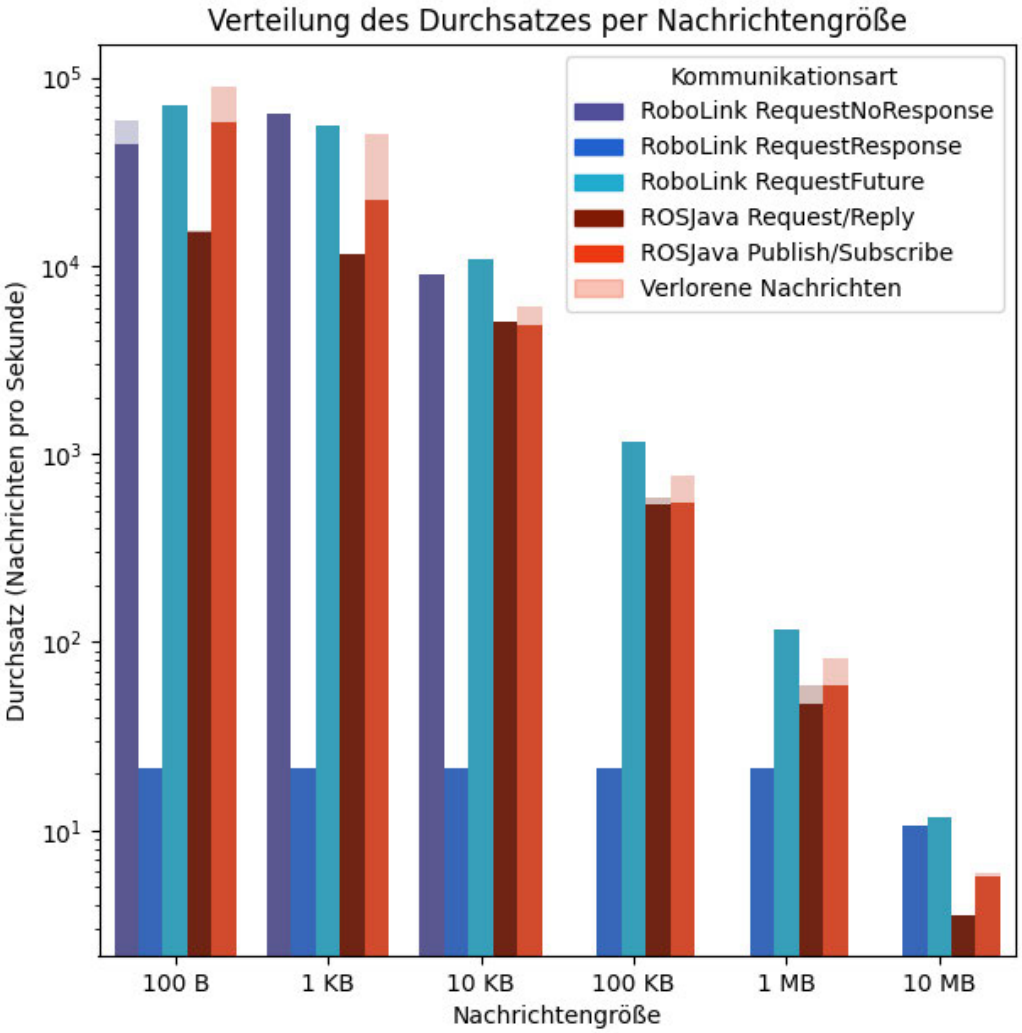


Abbildung 4.2: Durchsatz Testergebnis

### 4.2.2 Durchsatz

In Abbildung 4.2 ist die Verteilung des Durchsatzes für alle getesteten Varianten und Nachrichtengrößen abgebildet. Auch für diesen Test sind keine Daten der Request/No Response-Strategie des RoboLink für Nachrichtengrößen über 100 KB verfügbar. Einige Varianten konnten nicht alle Nachrichten korrekt übertragen, am oberen Ende des

jeweiligen Balkens werden diese verloren gegangenen Nachrichten leicht transparent angezeigt.

Die RoboLink-Strategie Request/No Response ist für die Nachrichtengrößen, die sie übertragen kann sehr gut aufgestellt und kann mit dem Durchsatz der anderen Varianten gut mithalten. Für 1 KB Nachrichten wurde sogar der höchste Durchsatz aller Varianten gemessen. Allerdings besteht das Risiko von Nachrichtenverlusten, denn bei Nachrichten mit einer Größe von 100 B sind in diesem Test fast ein Viertel der Nachrichten nicht vollständig übertragen worden.

Die Request/Response-Strategie von RoboLink ist synchron, daher ist der Durchsatz mit 20 Nachrichten pro Sekunde sehr gering gegenüber den anderen Varianten. Allerdings gingen dafür im gesamten Test keine Nachrichten verloren. Zusätzlich interessant ist, dass der Durchsatz für diese Strategie sehr konstant bleibt und bei Nachrichten von 10 MB mit den asynchronen Strategien mithalten kann und diese sogar teilweise übertrifft. Die Ergebnisse des Umlauftests haben gezeigt, dass die Request/Response-Strategie im Mittel maximal 2 ms für das Übertragen einer Nachricht von bis zu 100 KB braucht. Das würde bedeuten, dass der erwartete Durchsatz in diesem Test bei etwa 500 Nachrichten pro Sekunde liegen sollte. Der gemessene Durchsatz liegt damit sehr weit unter dem erwarteten Wert, Wiederholungen des Tests führten jedoch zu demselben Ergebnis. Eine erweiterte Nachforschung zeigte, dass eine Umlaufzeit von 2 ms nur erreicht werden kann, wenn zwischen dem Versenden von Nachrichten mindestens 50 ms gewartet wird. Da dies im Umlauftest der Fall war, konnte dieser Test diese Anomalie nicht aufdecken. Die echte Umlaufzeit für das Versenden aufeinanderfolgender Nachrichten scheint bei etwa 40 ms bis 50 ms zu liegen, ob dies auf einen Fehler der Implementierung oder andere Gründe zurückzuführen ist, wurde in dieser Arbeit nicht weiter untersucht.

Die Request/Future-Strategie zeigt für fast alle Nachrichtengrößen den höchsten Durchsatz, teilweise sogar mit einem deutlichen Abstand. Zusätzlich sind alle gesendeten Nachrichten vollständig übertragen worden. Diese Strategie sticht im Vergleich eindeutig als Variante mit dem besten Durchsatz heraus. Für die kleinsten Nachrichten, zeigt diese Variante einen Durchsatz von etwa 70.000 Nachrichten pro Sekunde.

Das Request/Reply-Modell von ROSJava schneidet trotz seiner asynchronen Implementierung in fast allen Fällen am schlechtesten gegenüber den anderen asynchronen Varianten ab. Eine Erklärung hierfür könnte sein, dass die Asynchronität des Modells nur nutzerseitig ausfällt, effektiv muss für jede Nachricht ein Thread auf den Erhalt der Antwort warten. Dieser Overhead könnte den verringerten Durchsatz erklären. Obwohl der

Durchsatz gegenüber den anderen Varianten gering ist, gehen bei größeren Nachrichten, ab 100 KB, auch bei diesem Modell Nachrichten verloren. Für 10 MB Nachrichten ist der Durchsatz mit 1,5 Nachrichten pro Sekunde nicht einmal halb so hoch wie bei der nächst besseren Variante.

Das zweite Modell von ROS, Publish/Subscribe, verliert von allen Varianten die meisten Nachrichten. Für jede Nachrichtengröße sind in diesem Test Nachrichten verloren gegangen. Der Durchsatz ist jedoch in fast allen Fällen höher gegenüber der ersten Variante von ROS. Auch in diesem Test konnten nicht alle 10 MB Nachrichten übertragen werden, da die Applikation vor Ende des Tests, aufgrund von nicht ausreichendem Arbeitsspeicher, abgestürzt ist.

Der Durchsatztest zeigt, dass die Auswahl der Kommunikationsart je nach Verwendungszweck unterschiedlich ausfallen kann. RoboLink zeigt insgesamt jedoch deutlich weniger verlorene Nachrichten gegenüber ROS und stellt mit der Request/Future-Strategie die leistungsstärkste Variante im Bezug auf den Durchsatz. ROS zeigte Schwächen insbesondere bei großen Nachrichten, die in manchen Fällen sogar zum Absturz der Applikation führten.

### 4.2.3 Recheneffizienz

- **ROS Middleware:** Der Ressourcenverbrauch von ROS zeigte einen deutlichen Unterschied zwischen den beiden Kommunikationsvarianten auf. Während die CPU-Auslastung des Request/Reply-Modells auf beiden Systemen zwischen 5% und 10% lag, stieg die Auslastung des Publish/Subscribe-Modells, vor allem bei der Verarbeitung von großen Nachrichten auf 40% bis 50% an. Auch für die Auslastung des RAM wurden deutlichen Unterschiede gemessen. Das Request/Reply-Modell zeigte auf beiden Systemen einen durchschnittlichen Speicherverbrauch von 50 MB bis 500 MB für alle Nachrichtengrößen. Das Publish/Subscribe-Modell maß für kleinere Nachrichten, von 100 B bist 1 KB, ähnliche Werte. Ab einer Nachrichtengröße von 10 KB zeigte der Speicherverbrauch extrem hohe Werte. Über die getestete Minute hinweg stieg der Speicherverbrauch auf bis zu 6 GB für 10 KB Nachrichten und auf bis zu 9 GB für 1 MB Nachrichten. Die Tests der Nachrichtengröße von 10 MB konnten nicht vollständig durchgeführt werden, da das RAM-Limit erreicht wurde und die Applikation abstürzte, bevor eine Minute vergangen war. Auch für

Nachrichten der Größe 100 KB und 1 MB zeichnete sich ein stetig steigender Speicherverbrauch ab und die Applikation stürzte innerhalb weniger Minuten ab.

- **RoboLink:** RoboLink zeigte eine deutlich geringere Auslastung der CPU zwischen 2% und 10% für die unterschiedlichen Kommunikationsvarianten und Nachrichtengrößen. Die Unterschiede zwischen den einzelnen Varianten sind hierbei nur sehr gering. Der einzige herausstehende Wert, war die Auslastung der Request/Response-Strategie, hier lag der Wert für die CPU stets unter 2%. Dies lässt sich durch die synchrone Verarbeitung erklären, da die Middleware durch das Übertragen einer einzelnen Nachricht zur Zeit nur wenig bis gar nicht ausgelastet wurde. Auch die Speicherauslastung ergab für alle Strategien sehr ähnliche Werte, insgesamt lag die Auslastung des RAM zwischen 100 MB und 400 MB für alle Varianten und Nachrichtengrößen.

### 4.2.4 Benutzerfreundlichkeit

Die Ersteinrichtung der ROS Middleware wurde durch mehrere Hindernisse erschwert. Ein explizites Betriebssystem musste eingerichtet werden, fehlende oder veraltete Dokumentation führten zu umfassenden Nachforschungen in externen Quellen. Zusätzlich waren nicht alle Funktionen verfügbar, da Abhängigkeiten nicht korrekt eingebunden waren. Das Kompilieren der Service Dateien brachte eine zusätzliche Komplexitätsebene und die Notwendigkeit der Auseinandersetzung mit einem weiteren Prozess mit sich. Die gesamte Einrichtung dauerte in etwa fünf bis sechs Stunden. Die Einrichtung von RoboLink vereinfachte den Prozess erheblich, insgesamt dauerte die Einrichtung auf einem neuen System weniger als eine Stunde. Die umfassende Dokumentation und Beispielimplementierung hob sich positiv von der Komplexität und dem Zeitaufwand der Einrichtung von ROS ab.

## 4.3 Fazit zur Vergleichsanalyse

Die soeben vorgestellte, detaillierte Vergleichsanalyse zwischen ROS und RoboLink unterstreicht die Wichtigkeit für eine informierte Auswahl bei der Suche nach einer Middleware-Lösung. Die gewählte Middleware sollte den spezifischen Anforderungen der gewünschten Applikation entsprechen. RoboLink ragt mit seiner Leistung und Benutzerfreundlichkeit heraus, allerdings muss bedacht werden, dass ROS ein umfangreiches Ökosystem bietet. Applikationen, die lediglich eine Middleware zur Nachrichtenübertragung benötigen, könnten von der Leistung und Effizienz RoboLinks profitieren. Wenn die Applikation jedoch den Funktionsumfang bestimmter Bibliotheken aus dem ROS Ökosystem benötigt, könnte die Wahl auf ROS fallen.

# 5 Fazit und Ausblick

In dieser Arbeit wurden verteilte Architekturansätze zum Kontrollieren autonomer Roboter am Beispiel des Ninebot Loomo erforscht. Die Untersuchung zielte darauf, verschiedene Aspekte der zwei Middleware-Lösungen, ROS und RoboLink, zu vergleichen. Es wurden Einsichten in die Stärken und Schwächen beider Ansätze aufgezeigt. In diesem Kapitel werden die wichtigsten Argumente zusammengefasst, die daraus folgenden Implikationen für die Auswahl einer passenden Middleware abgeleitet, Limitationen der Vorgehensweise herausgearbeitet und ein Ausblick für zukünftige Nachforschungen gegeben.

## 5.1 Wichtigste Erkenntnisse

### 5.1.1 Leistung

RoboLink übertraf ROS in Bezug auf die Umlaufzeit, den Durchsatz und die Effizienz, insbesondere bei großen Nachrichten. Dies kann der leichtgewichtigen Architektur zugesprochen werden, die spezifisch hierfür optimiert werden konnte. Applikationen, die eine hohe Performanz und minimale Verzögerungen erfordern, können davon profitieren. Außerdem benötigt RoboLink weniger RAM und CPU-Ressourcen und ist daher besser für Systeme geeignet, die in der Hardware eingeschränkt sind, was im Bereich der autonomen Roboter ein klarer Vorteil sein kann.

### 5.1.2 Benutzerfreundlichkeit

Die Benuzterfreundlichkeit von RoboLink war durch die einfache Installation und beigelegte Dokumentation deutlich höher, während ROS aufgrund von unzureichender Dokumentation und Kompatibilitätsproblemen Schwierigkeiten bereitete. Die Lernkurve und

die Ressourcen zur Unterstützung der Implementierung sollten daher bei der Auswahl der Middleware ebenfalls eine Rolle spielen.

### 5.1.3 Ökosystem

Trotz der technischen Schwächen bietet ROS ein umfassendes Ökosystem mit zahlreichen Bibliotheken und Werkzeugen, die den Entwicklungsprozess für spezifische Anwendungen erheblich beschleunigen können.

## 5.2 Implikationen für die Auswahl einer Middleware

Die Vergleichsanalyse in dieser Arbeit betont den Bedarf eines strategischen Ansatzes zur Auswahl einer Middleware für autonome Roboter. Die Wahl zwischen einem standardisierten System wie ROS oder einer maßgeschneiderten Lösung sollte durch spezifische Projektanforderungen gesteuert werden. Für Applikationen, die eine hohe Performanz und minimale Latenz erfordern, kann ein benutzerdefinierter Ansatz vorteilhaft sein. ROS wiederum könnte für Projekte eingesetzt werden, die von umfangreichen Bibliotheken und Werkzeugen profitieren.

## 5.3 Einschränkungen der Arbeit

Während diese Untersuchung wertvolle Einblicke bietet, ist es wichtig bestimmte Limitationen hervorzuheben.

- **Fokus der Middleware:** Die Ergebnisse dieser Arbeit fokussieren sich auf die Leistung und die Benutzerfreundlichkeit der Middleware-Lösungen, ohne tiefere Einblicke in die Applikationsebene und spezifische Anwendungsfälle für autonome Roboter. Die verschiedenen Middleware-Lösungen könnten für ein Multi-Roboter System eingesetzt und erweitert verglichen werden um neue Einsichten zu gewinnen.
- **Subjektivität:** Die Evaluation der Unterstützung der Community und der Dokumentation von ROS ist subjektiv und basiert auf Erfahrungen, die während der Implementationsphase gemacht wurden. Tatsächlich kann die Unterstützung für andere Versionen und andere Konfigurationen unterschiedlich sein.

- **Spezifikation ROSJava:** ROSJava wird seit mehreren Jahren nicht mehr weiterentwickelt und ist daher nicht mit neueren ROS-Versionen kompatibel. Ein Vergleich mit einer Implementierung zum Beispiel in C++ (roscpp) mit einer neueren Version könnte weitere Erkenntnisse liefern.
- **Lokale Tests über LAN:** Obwohl die Tests dieser Arbeit über LAN durchgeführt wurden, ermöglichen sie eine saubere Analyse der Middleware-Lösungen unter idealen Bedingungen. Allerdings werden viele autonome Roboter und verteilte Systeme im WLAN eingesetzt. Eine erneute Durchführung der Tests über WLAN könnte Einblicke in die Leistung und Effizienz beider Middleware-Lösungen über verlustbehaftete Verbindungen bieten.
- **Loomotests:** Mit der Durchführung von erweiterten Tests auf dem Loomo selbst, könnten plattformspezifische Erkenntnisse gewonnen werden, die dazu beitragen die Limitationen des Loomo und die Leistung verschiedener Middleware-Systeme in ressourcenbeschränkten Umgebungen besser zu verstehen.

## 5.4 Erweiterung der RoboLink Middleware

Die in dieser Arbeit vorgestellte RoboLink-Middleware bietet eine robuste Grundlage für die Kommunikation in verteilten Systemen. Dennoch gibt es einige Bereiche, in denen RoboLink weiter optimiert werden könnte:

- **Verteilung der Konfiguration:** Eine wichtige Erweiterung von RoboLink ist die Möglichkeit die Konfiguration dynamisch zu verteilen. Aktuell müssen Knoten manuell vorkonfiguriert werden, was in großen verteilten Systemen mit vielen Knoten nur umständlich umsetzbar ist. Die Skalierbarkeit könnte mit dieser Erweiterung daher deutlich verbessert werden.
- **Parallele TCP Verbindungen:** Eine weitere sinnvolle Erweiterung der RoboLink Middleware wäre die Unterstützung von parallelen TCP-Verbindungen, um die Kommunikation zwischen den Knoten zu beschleunigen und den Durchsatz weiter zu erhöhen. Dies ist besonders nützlich für Applikationen die große Mengen von Daten übertragen müssen.
- **Alternative zu MessagePack:** MessagePack ist zwar eine effiziente Methode zur Serialisierung, bei großen Datenmengen, wie Videodaten, könnte jedoch eine

alternative Methode implementiert werden, die zum Beispiel H.264 unterstützt. Dieses Format kann die Daten deutlich besser komprimieren.

- **Python- und C++-Implementierung:** Eine Implementierung von RoboLink in Python und C++ könnte die Unterstützung für verschiedene Roboterplattformen verbessern.
- **Fehlertoleranz:** RoboLink verwendet eine dezentrale Architektur, daher ist es wichtig die Fehlertoleranz im System zu erhöhen. Dies könnte zum Beispiel durch die Einführung von Mechanismen zur Wiederherstellung von Knoten erreicht werden. Dieser könnte sicherstellen, dass Knoten automatisch neu gestartet oder ersetzt werden, falls es zu Ausfällen kommt.

## 5.5 Zukünftige Forschungsperspektiven

Aufbauend auf den Resultaten dieser Arbeit, könnte weitere Forschung in einigen Gebieten der autonomen Roboter entstehen:

- **Erweiterte Plattformtests:** Eine Erweiterung der Analyse um weitere Roboter-Plattformen, zum Beispiel ROS 2, könnte eine Generalisierung verbessern und ein tieferes Verständnis für verschiedene Architekturen liefern.
- **Erweiterte Anwendungsfälle:** Die Untersuchung von komplexeren und vielseitigeren Anwendungsfällen, wie zum Beispiel Multi Roboter Koordination, kann tiefere Einsichten in die Stärken und Limitationen der Middleware-Lösungen in Realsituationen einbringen.
- **Langfristige Studien:** Mit der Durchführung langfristiger Studien zur Beurteilung der Langzeitauswirkungen auf die Zuverlässigkeit oder die Wartung der Middleware-Systeme, könnten weitere wichtige Aspekte untersucht werden.
- **Skalierbarkeit:** Die Skalierbarkeit ist ein wichtiger Faktor bei der Entwicklung verteilter Systeme. Eine Erweiterung der bestehenden Tests um die Skalierbarkeit der Middleware-Lösungen zu analysieren könnte die Aussagekraft des Vergleichs erhöhen.

## 5.6 Schlussfolgerungen und abschließende Gedanken

Diese Arbeit hat die Wichtigkeit der Auswahl einer passenden Middleware für autonome Robotersysteme demonstriert. Durch die Evaluierung der Leistung, Effizienz und Benutzerfreundlichkeit der beiden vorgestellten Middleware Systeme, ROS und RoboLink, stellt diese Forschung eine Grundlage für eine informierte Entscheidungsfindung bei der Entwicklung von Robotersystemen bereit. Während der primäre Fokus dieser Arbeit technischer Natur ist, sollte die Bedeutung ethischer Bedenken, die mit der Weiterentwicklung autonomer Roboter einhergehen, nicht ungeachtet bleiben. Zukünftige Forschung sollte untersuchen, wie sichergestellt werden kann, dass diese Innovationen verantwortlich und positiv in unsere Gesellschaft einwirken.

# Literaturverzeichnis


[1] Macenski, Steven ; Foote, Tully ; Gerkey, Brian ; Lalancette, Chris ; Woodall, William: Robot Operating System 2: Design, architecture, and uses in the wild. In: *Science Robotics* 7 (2022), Mai, Nr. 66. – URL http://dx.doi.org/10.1126/scirobotics.abm6074. – ISSN 2470-9476

[2] Malavolta, Ivano ; Lewis, Grace A. ; Schmerl, Bradley R. ; Lago, Patricia ; Garlan, David: How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2020), S. 31–40

[3] Martinez, Aaron ; Fernndez, Enrique: *Learning ROS for Robotics Programming*. Packt Publishing, 2013. – ISBN 1782161449

[4] Maruyama, Yuya ; Kato, Shinpei ; Azumi, Takuya: Exploring the performance of ROS2. In: *Proceedings of the 13th International Conference on Embedded Software*. New York, NY, USA : Association for Computing Machinery, 2016 (EMSOFT '16). – URL https://doi.org/10.1145/2968478.2968502. – ISBN 9781450344852

[5] Portugal, David ; Iocchi, Luca ; Farinelli, Alessandro: *A ROS-Based Framework for Simulation and Benchmarking of Multi-robot Patrolling Algorithms*. S. 3–28. In: Koubaa, Anis (Hrsg.): *Robot Operating System (ROS): The Complete Reference (Volume 3)*. Cham : Springer International Publishing, 2019. – ISBN 978-3-319-91590-6

[6] Sanika, R.: *"Performance Comparison of Message Queue Methods*, University of Nevada, Las Vegas (UNLV), Master's Thesis, 2019. – URL http://dx.doi.org/10.34917/16076287. – UNLV Theses, Dissertations, Professional Papers, and Capstones, Paper 3746

[7] Tanenbaum, A.S. ; Steen, M. van: *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007. – URL https://books.google.de/books?id=DL8ZAQAAIAAJ. – ISBN 9780132392273

[8] Yongguo, Jiang ; Qiang, Liu ; Changshuai, Qin ; Jian, Su ; Qianqian, Liu: Message-oriented Middleware: A Review. In: *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, 2019, S. 88–97

# A Anhang

https://gitlab.com/LeoPeters/bachelorleopeters

# Glossar

**Central Processing Unit** Zentraler Prozessor eines Computers, der Anweisungen von Software oder dem Betriebssystem ausführt.

**Data Distribution Service** Ein Standard für die Datenverteilung in Echtzeitsystemen.

**Durchsatz** Anzahl der erfolgreich verarbeiteten Nachrichten pro Zeiteinheit in einem System.

**H.264** Ein Videokompressionsstandard mit hoher Kompressionsrate.

**Java Virtual Machine** Eine Laufzeitumgebung, in der Java-Programme ausgeführt werden.

**Java-Reflection** Eine Methodik der Programmiersprache Java um zur Laufzeit Informationen über Klassen zu erhalten und diese zu manipulieren.

**Knoten** Ein Dienst oder eine Einheit in einem verteilten System, die eine spezifische Aufgabe erfüllt.

**MessagePack** Binäres Serialisierungsformat zur effizienten Übertragung von Daten.

**Middleware** Teil einer Software, die als Vermittler von Nachrichten zwischen verschiedenen Diensten fungiert.

**Network Time Protocol** Ein Protokoll zur Synchronisierung der Uhren von verschiedenen Systemen über ein Netzwerk.

**Publish/Subscribe** Ein Kommunikationsprinzip, bei dem Sender (Publisher) Nachrichten über Topics an Empfänger (Subscriber) übertragen.

**Queue** Eine Datenstruktur, welche Elemente in einer Reihenfolge abspeichert, wobei Elemente die zuerst abgelegt-, auch zuerst verarbeitet werden.

**Random Access Memory** Der Arbeitsspeicher eines Computers, er speichert Daten und Programme vorübergehend, während sie in Gebrauch sind.

**Remote Procedure Call** Aufruf einer Funktion auf einem entfernten System, als wäre der Aufruf lokal.

**Request/Future** Ein Kommunikationsmuster, bei dem ein Client eine Anfrage an einen Server sendet und ein Future-Objekt erhält, mithilfe dessen zu einem späteren Zeitpunkt das Ergebnis abgerufen werden kann.

**Request/No Response** Ein Kommunikationsmuster, bei dem ein Client eine Anfrage an einen Server sendet, aber keine Antwort erwartet.

**Request/Response** Ein Kommunikationsmuster, bei dem ein Client eine Anfrage an einen Server sendet und auf eine Antwort wartet.

**Robot Operating System** Framework zur Entwicklung von Robotersoftware. Stellt Werkzeuge und Bibliotheken zur Verfügung um Entwickler zu unterstützen und fungiert als Middleware.

**Separation of Concerns** Ein Prinzip der Softwarearchitektur, bei dem Systeme in Module unterteilt werden, die für eine spezifische Aufgabe verantwortlich sind.

**Single Point of Failure** Eine Komponente in einem System, bei deren Ausfall der Rest des System nicht mehr arbeitsfähig ist.

**Software Development Kit** Sammlung von Werkzeugen für die Entwicklung von Software für eine bestimmte Plattform.

**Thread** Ein Ausführungskontext in einem laufenden Programm, der parallel zu anderen Threads Aufgaben ausführen kann.

**Topic** Ein Kommunikationskanal, auf den Nachrichten gesendet und diese empfangen werden können.

**Transport Control Protocol** Ein verbindungsorientiertes Kommunikationsprotokoll zur Übertragung von Paketen über ein Netzwerk.

**Umlaufzeit** Zeit die beim Versenden von Nachrichten benötigt wird um eine Antwort auf eine Anfrage zu erhalten.

**User Datagram Protocol** Ein verbindungsloses Kommunikationsprotokoll zur Übertragung von Paketen über ein Netzwerk.

**Verteilte Systeme** Ein Netzwerk aus eigenständigen Computereinheiten, die koordiniert arbeiten um ein gemeinsames Ziel zu erreichen.

**VisualVM** Ein Werkzeug zur Überwachung und Profilierung von Java-Anwendungen.

## Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original