DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Pepper VR – Teleopperation eines humanuiden Roboter auf Basis der Analyse menschlicher Bewegung

Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik / Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Matthias Schuhmacher und Marlene Rieder

Abgabedatum 20. Mai 2024

Bearbeitungszeitraum	300 Stunden
Matrikelnummer	4128647 und 8261867
Kurs	tinf21b3 und tinf21b5
Gutachter der Studienakademie	Prof. Dr. Marcus Strand

Erklärung

Wir versicheren hiermit, dass wir unsere Studienarbeitmit dem Thema: »Pepper VR – Teleopperation eines humanuiden Roboter auf Basis der Analyse menschlicher Bewegung« selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versicheren zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort Datum

Unterschrift

Zusammenfassung

Das Ziel der vorliegenden Studienarbeit ist es, eine Verbindung zwischen einer Virtual-Reality Brille und dem humanoiden Roboter Pepper herzustellen. Das Kamerabild des Roboters soll auf der Brille angezeigt werden, ebenfalls soll es möglich sein, den Roboter mit Hilfe der Controller der Brille zu steuern.

Inhaltsverzeichnis

1 Einleitung **8**

2 Grundlagen **9**
2.1 Pepper . . . 9
2.2 Publish and Subscribe . . . 10
2.3 NAOqi . . . 11
2.3.1 Was ist NAOqi? . . . 11
2.3.2 NAOqi Vorgehensweise . . . 14
2.3.3 NAOqi Module . . . 15
2.3.4 NAOqi Speicherverwaltung . . . 16
2.4 Middleware . . . 17
2.5 Robot Operating System 2 . . . 18
2.5.1 Nodes . . . 18
2.5.2 Topics . . . 19
2.5.3 Interfaces . . . 20
2.5.4 Parameter . . . 22
2.5.5 Start . . . 24
2.5.6 Nutzer Bibliotheken . . . 24
2.6 Programming Languages . . . 24
2.6.1 C++ . . . 24
2.6.2 Python . . . 24
2.7 Virtual Reality . . . 24
2.8 Entwicklung für Virtual Reality . . . 24
2.9 TCP . . . 24

3 Technologieauswahl **25**
3.1 VR-Brillen . . . 25
3.2 Entwicklung für VR-Brillen . . . 25

4 Umsetzung **26**
4.1 Pepper . . . 26
4.2 MetaQuest3 . . . 26
4.3 Verbindung . . . 26

5 Anwendungsgebiete **27**
5.1 Pflege 27

6 Fazit **28**

7 Fortsetzung des Projekts **29**

Literaturverzeichnis **30**

Abbildungsverzeichnis

2.1 NAOqi Framework Sprachübergreifend 12
2.2 NAOqi Framework Introspektion 13
2.3 NAOqi Framework Kommunikation 14
2.4 NAOqi Broker Bibliotheken Module 14
2.5 NAOqi Broker Modul Methoden 15
2.6 NAOqi Framework Speicherverwaltung 17

Tabellenverzeichnis

Liste der Algorithmen

2.1 ROS 2 Message 21
2.2 ROS 2 Service 21
2.3 ROS 2 Action 22

Formelverzeichnis

Abkürzungsverzeichnis

SAS Société par actions simplifiée 9
API Application Programming Interface 9
SDK Software Development Kit 10
URL Uniform Resource Locator 13
IP Internet Protocol 13
RPC Remote Procedure Call 13
LPC Local Procedure Call 13
ACM Association for Computing Machinery 10
JMS Java Message Service 11
DDS Data Distribution Service 11
MOM Message Oriented Middleware 11
ROS Robot Operating System 11
IDL Interface Definition Language 20
IMU Inertial Measurement Unit 20
CLI Command Line Interface 23

Kapitel 1

Einleitung

Der technologische Fortschritt steht nie still, deshalb ist es auch uns ein wichtiges Anliegen daran teilzuhaben. Mit dieser Arbeit

Kapitel 2

Grundlagen

2.1 Pepper

Pepper ist ein humanoider Roboter, der entwickelt wurde, um die Gefühle und Gesten von Menschen zu analysieren und basierend auf diesen, darauf zu reagieren. Das Projekt entstand durch eine Zusammenarbeit des französischen Unternehmens Aldebaran Robotics Société par actions simplifiée (SAS) und des japanischen Telekommunikations- und Medienkonzerns SoftBank Mobile Corp. Ziel diese Projektes war es, einen humanuiden "Roboter-Gefährten" oder einen "persönlichen Roboter-Freund" zu schaffen, der zunächst im Gewerbesektor in Verkaufsräumen, an Empfangstischen oder in Bildungs- und Gesundheitseinrichtungen eingesetzt werden sollte. Die Produktion wurde jedoch aufgrund geringer Nachfrage bis auf Weiteres pausiert.
Das Konzept von Pepper distanziert sich von herkömmlichen Industrierobotern und reinen Spielzeugrobotern, indem er als informativer und kommunikativer Begleiter konzipiert wurde. Sein Aussehen, das im etwa an die Größe eines Kindes angelehnt ist, sowie ein freundliches Gesicht und eine kindliche Stimme sind im ästhetischen Konzept von "kawaii" (japanisch für "niedlich" oder auch "liebenswert") gehalten.
Pepper wurde im Rahmen einer Präsentation am 5. Juni 2014 als der "erste persönliche Roboter der Welt mit Emotionen" vorgestellt. Die Vermarktung begann damit, dass SoftBank Pepper-Geräte in ihren Verkaufsräumen einsetzte, um Kunden zu unterhalten und zu informieren. Die Roboter sollte dabei den Umgang mit Kunden erlernen, um zukünftige Anwendungsmöglichkeiten zu erforschen. Verkauft wurde offiziell ab dem 3. Juli 2015 zu einem Preis von 198.000 Yen pro Einheit, zuzüglich monatlicher Gebühren für Zusatzleistungen. Im Laufe der Zeit wurde Pepper auch für den Einsatz in weiteren Unternehmen und Einrichtungen verfügbar gemacht.
Pepper wird mit einer Grundausstattung an Anwendungen geliefert, jedoch sind für spezifische Anwendungen, individuell entwickelte Softwarelösungen erforderlich wie auch zum Beispiel in diesem. SoftBank ermöglichte unabhängigen Entwicklern durch die Veröffentlichung der Schnittstellen den Zugang zu einem Interface für Applikationsprogramme, um zusätzliche Anwendungen für Pepper zu erstellen. Das NAOqi-Framework welches für diesen Nutzen bereitgestellt wurde, beinhaltet eine Application Programming In-

terface (API), eine Software Development Kit (SDK) und weitere Tools, welche in den Sprachen Python und C++ uneingeschränkten Zugriff auf die Komponenten, Sensoren und Aktoren des Roboters bieten, dazu später Ausführlicheres in Abschnitt 2.3. Mit Hilfe diese Interfaces haben verschiedene Unternehmen integrierte Lösungen entwickelt, die Pepper beispielsweise bei der Kundenberatung unterstützen können.
Das Design von Pepper ist dem Menschen ähnlich und umfasst einen Kopf mit integrierten Mikrofonen und Kameras sowie einen Torso mit weiteren Sensoren für Stabilität und Sicherheit. Der Roboter verfügt über verschiedene Mechaniken, die es ihm ermöglichen, sich flüssig zu bewegen und mit Personen zu interagieren. Durch die Verwendung von Kameras und bereitgestellter Software ist Pepper in der Lage, Emotionen bei seinen Gesprächspartnern zu erkennen und darauf zu reagieren, obwohl er selbst keine Mimik besitzt. Sicherheitsvorkehrungen wie Abstandssensoren und Stabilisatoren gewährleisten einen sicheren Einsatz von Pepper in verschiedenen Umgebungen. Diese können jedoch bedingt durch den Entwickler deaktiviert werden, um den Roboter in komplexeren oder laborähnlichen Umgebungen zu betreiben.

2.2 Publish and Subscribe

Bevor wir uns mit den Technologien und Herangehensweisen des Pepper Roboters beziehungsweise dessen Betriebssystem beschäftigen, müssen wir uns mit dem Publish-Subscribe-Modell auseinandersetzen.
Das Publish-Subscribe-Modell ist ein Paradigma für einen effektiven Nachrichtenaustausch in verteilten Systemen. Erstmals publiziert in einem Paper der Association for Computing Machinery (ACM) von 1987[Wikipedia o. D.], ermöglicht es die flexible Kommunikation zwischen verschiedenen Komponenten, indem es einen Mechanismus bereitstellt, über den Nachrichten von einem Sender, dem Publisher, an einen oder mehrere Empfänger, den Subscribern, verteilt werden können. Ein zentrales Element dieses Modells ist dabei Nachrichtenbroker oftmals auch nur als Borker referenziert, der als Vermittler zwischen Publishern und Subscribenden fungiert.
Der Broker empfängt Nachrichten vom Publisher und organisiert sie in verschiedene Kategorien, welche als Topics bezeichnet werden. Diese Topics dienen als thematische Selektionsmerkmale, die den Inhalt der Nachrichten beschreiben. Sie können in den verschiedensten Datentypen erfolgen, einfache Ganzzahlen, Texte bis hin zu Bildern oder weitaus komplexeren Datenstrukturen. Durch diese Kategorisierung in den Topics, wird eine gezielte Auswahl und Weiterleitung der Nachrichten ermöglicht.
Subscriber können bestimmte Topics abonnieren, die ihren Gewünschten entsprechen. Dadurch erhalten sie nur die Nachrichten, die zu den von ihnen gewählten Themen gehören. Dieser Prozess der Nachrichtenauswahl und -verarbeitung wird als Filterung bezeichnet und kann auf zwei Arten erfolgen: themenbasiert und inhaltsbasiert.
Im themenbasierten Ansatz erhalten Abonnenten alle Nachrichten zu den Topics, welche sie abonniert haben. Der Publisher definiert die verfügbaren Topics, aus welchen die Abonnenten selbst wählen können. Im inhaltsbasierten Ansatz dagegen, werden Nachrichten

nur an die Abonnenten weitergeleitet, wenn sie den vom dessen festgelegten Kriterien entsprechen. Dabei ist der Abonnent für die Spezifikation dieser Kriterien verantwortlich[ITWissen.info o. D.]
Das Publish-Subscribe-Modell bietetdurch seine Herangehensweise eine hohe Flexibilität und Skalierbarkeit, und wird daher in verschiedenen Anwendungsbereichen eingesetzt. Es ermöglicht eine Entkopplung von Nachrichtenerzeugung und -verarbeitung, was gerade in der Entwicklung verteilter Systeme Prozesse erleichtert. Die vermutlich bekannteste Implementierung des Modell ist das MQTT-Protokoll, welches konzipiert wurde um Telemetriedeaten zwischen Sensoren und Servern zu übertragen speziell in unzuverlässigen Umgebungen[Elektronik-Kompendium.de o. D.]
Weiter findet das Publish-Subscribe-Modell Anwendung in einer Vielzahl von weiteren Systemen und Technologien. Beispiele dafür sind der Java Message Service (JMS), der Data Distribution Service (DDS) oder wie im Fall diese Projektes die Message Oriented Middleware (MOM) zu denen auch Robot Operating System (ROS) gehört.

2.3 NAOqi

Im folgenden Abschnitt wird das NAOqi-Framework, welches auf dem Pepper Roboter läuft, genauer erläutert.

2.3.1 Was ist NAOqi?

NAOqi ist die Bezeichnung für die Hauptsoftware, die auf dem Pepper Roboter ausgeführt wird und ihn intern steuert. Diese kann mit persönlichen Modulen weiterentwickelt und angepasst werden. Dazu können mit Hilfe der Aldebaran SDK auch NQOqi-SDK genannt, eigene Module oder Bibliotheken entwickelt werden. Diese NAOqi-SDK ist die Basis des NAOqi Frameworks und ist in C++ implementiert[University Year of access].
Das NAOqi Framework ist das Programmiergerüst, welches zur Programmierung von NAO und Pepper Robotern verwendet wird. Es implementiert alle allgemeinen Anforderungen der Robotik, einschließlich: Parallelität, Ressourcen-Management, Synchronisation und Ereignisse. Dieses Framework ermöglicht eine homogene Kommunikation zwischen verschiedenen Modulen wie etwa die Bewegung, Audio oder Video sowie eine homogene Programmierung und einen homogenen Informationsaustausch. Das Framework ist:

- plattformübergreifend, wie bereits erwähnt, basiert die NAOqi-SDK auf C++ und bietet damit die Möglichkeit auf Windows, Linux oder sogar auch Mac zu entwickeln.
- sprachübergreifend, mit einer identischen API für C++ und Python. Weitere Details dazu sind in Abschnitt 2.3.1 aufgeführt.

- fähig auf Introspektion, was bedeutet, dass das Framework weiß, welche Funktionen in den verschiedenen Modulen verfügbar sind und wo. Für Details diesbezüglich siehe Abschnitt 2.3.1.

Sprachübergreifend

Software kann in C++ und Python entwickelt werden. Eine Übersicht über die Sprachen selbst in den Abschnitten Unterabschnitt 2.6.1 und Unterabschnitt 2.6.2. In allen Fällen sind die Programmiermethoden genau die gleichen, alle vorhandenen APIs können unabhängig von den unterstützten Sprachen aufgerufen werden:

- Wird ein neues C++-Modul erstellt, können die C++-API-Funktionen von überall aus aufgerufen werden,
- Sind sie richtig definiert, können auch die API-Funktionen eines Python-Moduls von überall aus aufgerufen werden.

Normalerweise werden die Verhaltensweisen in Python und Ihre Dienste in C++ entwickelt.

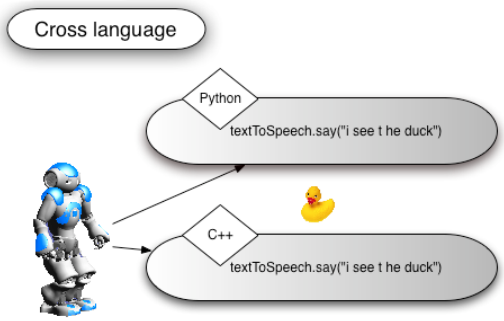

Abbildung 2.1: NAOqi Framework Sprachübergreifend

Introspektion

Die Introspektion ist die Grundlage der Roboter-API, der Fähigkeiten, der Überwachung und der Maßnahmen bei überwachten Funktionen. Der Roboter selbst kennt alle verfügbaren API-Funktionen. Wird eine Bibliothek entladen, werden die entsprechenden API-Funktionen automatisch ebenfalls entfernt. Eine in einem Modul definierte Funktion kann der API mit einem `BIND_METHOD` hinzugefügt werden.
Wird eine Funktion gebunden, werden automatisch folgende Funktionen ausgeführt:

- Funktionsaufruf in C++ und Python, wie in Abschnitt 2.3.1 beschrieben

- Erkennen der Funktion, wenn sie gerade ausgeführt wird
- Funktion lokal oder aus der Ferne, z.B. von einem Computer oder einem anderen Roboter, ausführen weiter im Detail beschrieben in Abschnitt 2.3.1
- Generierung und Aufruf von `wait`, `stop`, `isRunning` in Funktionen

Die API wird im Webbrowser angezeigt wenn auf das Gerät per Uniform Resource Locator (URL) oder Internet Protocol (IP)-Addresse auf dem Port 9559 zugegriffen wird. In dieser Übersicht, zeigt der Roboter seine Modulliste, Methodenliste, Methodenparameter, Beschreibungen und Beispiele an. Der Browser zeigt auch parallele Methoden an, die überwacht, zum Warten veranlasst und gestoppt werden können.
Die Introspektion und derer Implementation im NAOqi-Framework, ist also ein nützliches Werkzeug, welches es ermöglicht, die Roboter-API zu verstehen und zu verwenden aber auch zu überwachenund zu steuern.

Abbildung 2.2: NAOqi Framework Introspektion

Verteilter Baum und Kommunikation

Eine Echtzeitanwendung kann aus einer einzelnen ausführbaren Datei oder einem Baum von mehreren Systemen wie etwa Robotern, Prozessen oder Modulen bestehen. Unabhängig davon sind die Aufrufmethoden immer dieselben. Eine ausführbare Datei kann durch eine Verbindung mit einem anderen Roboter mit IP-Adresse und Port verbunden werden, sodass alle API-Methoden von anderen ausführbaren Dateien sind auf die gleiche Weise verfügbar sind, genau wie bei einer lokalen Methode. NAOqi trifft dabei selbst die Wahl zwischen schnellem Direktaufruf Local Procedure Call (LPC) und Fernaufruf Remote Procedure Call (RPC).

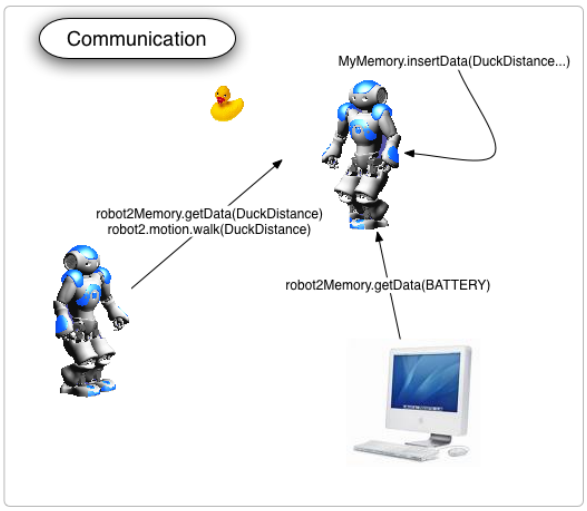

Abbildung 2.3: NAOqi Framework Kommunikation

2.3.2 NAOqi Vorgehensweise

Die NAOqi Software, welche auf dem Roboter läuft, ist ein Broker. Wenn dieser startet, lädt er eine Voreinstellungsdatei in den Speicher, in der festgelegt ist, welche Bibliotheken in dieser Konfiguration geladen werden sollen. Jede Bibliothek enthält ein oder mehrere Module, die den Broker benutzen, um ihre Methoden bereitzustellen.

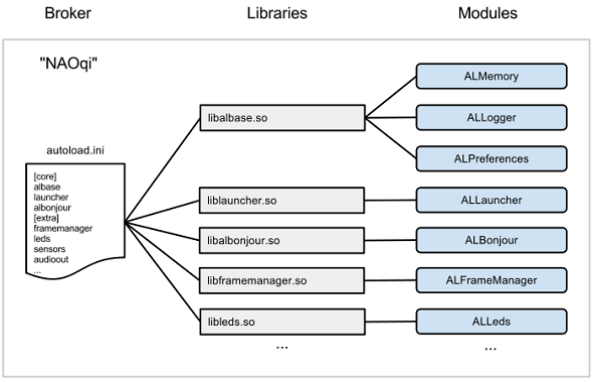

Abbildung 2.4: NAOqi Broker Bibliotheken Module

Der Broker selbst bietet Nachschlagdienste an, so dass jedes Modul im Baum oder im Netzwerk jede Methode finden kann, die an dem Broker bekannt gegeben wurde.
Das Laden von Modulen bildet dann einen Baum von Methoden, die mit Modulen verbunden sind, und von Modulen, welche mit dem Broker verbunden sind.

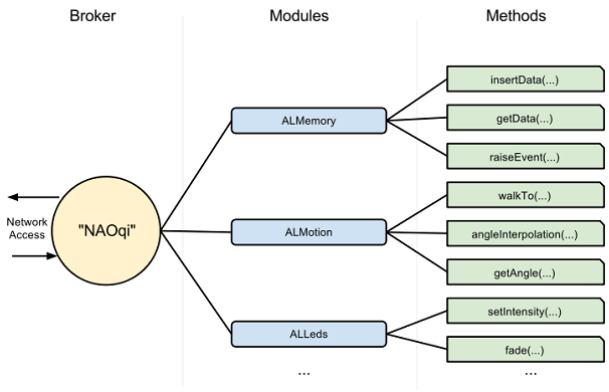

Abbildung 2.5: NAOqi Broker Modul Methoden

NAOqi Proxy

Ein weiterer wichtiger Bestandteil des NAOqi Broker ist der Proxy welcher grundlegend die Aufgabe eines Modules, näher beschrieben im folgenden Unterabschnitt 2.3.3, repräsentiert. Dies kann lokal oder entfernt enstehen der einzige Unterschied dabei ist, dass bei einer entfernten Refernz die IP des entfernten Broker mit angegeben werden muss, gleich bleibt aber im internen, dass der Proxy die Methoden des Moduls an den Broker weiterleitet.

2.3.3 NAOqi Module

Module sind die Grundbausteine der personalisierten Gestaltung von NAOqi. In ihnen kann der Nutzer eigene funktionaitäten implementieren und diese dem Broker bereitstellen. Ein Modul kann dabei aus einer oder mehreren Klassen bestehen, die wiederum Methoden enthalten.
Standardmäßig ist jedes Modul eine Klasse innerhalb einer Bibliothek, welche über eine `autoload.ini`-Datei geladen wird, worauf die Modulklasse automatisch instanziiert wird. Weiter können auch Module von übergeordneten Module abgeleitet werden, ähnlich wie etwa bei der Vererbung in der objektorientierten Programmierung. Wird eine Klasse voneiner anderen abgeleitet, können die Methoden gebunden werden, wodurch ihre Namen und MethodenSignaturen direkt dem Broker bekannt gemacht werden.
Weiter können, wie bereits erwähnt, Module sowohl lokal als auch entfernt implementiert werden. Ist es ein solches entferntes Modul, wird es als ausführbare Datei kompiliert, und kann auch außerhalb des Roboters ausgeführt werden.
Unabhängig jedoch von der Lagerung der Module, enthält jedes Modul eine Bandbreite von Methoden, welche wie im Abschnitt 2.3.1 beschrieben, gebunden werden können und damit nach außen hin nutzbar gemacht werden. Es ist also unabhängig der Lagerung der Module möglich, diese in der gleichen Weise aufzurufen. Module passen sich also automatisch selbst an.

Entfernte Module

Entfernte Module sind Module, die über das Netzwerk kommunizieren, dies kann auf demselben Robotter sein, also auch auf einem anderen Gerät im Netzwerk. Ein entferntes Modul braucht einen entfernten Broker, um mit anderen Modulen zu kommunizieren, da es selbst keinen eigenen besitzt. Broker sind dann für den gesamten Netzwerkteil verantwortlich. Über diesen Umweg über das Netzwerk können keine schnellen Zugriffe über Remote-Module durchgeführt werden wie etwa direkte Speicherzugriffe.
Diese entfernten Module sind trivialer in der Handhabung, da sie außerhalb des Systems entwickelt und debugged werden. Dagegen sind sie aber in der Laufzeit selbst deutlich schwächer in den Punkten Geschwindigkeit und Speichernutzung, gegenüber iherer lokalen Gegenstücke. Diese Funktionalität der entfernten entwicklung spielt speziell in der Implementierung dieses Projektes eine grundlegende Rolle.

Lokale Module

Lokale Module werden als Bibliotheken kompiliert und nur auf dem Roboter verwendet. Dadurch sind sie schneller und effizienter in der Laufzeit als auch in der Speicherverwaltung, dagegen ist die Entwicklung und das Debugging auf dem Roboter selbst deutlich aufwendiger.
Sie bestehen aus zwei oder mehr Module, welche auf dem Roboter im selben Prozess gestartet werden. Sie kommunizieren miteinander über denselben Broker.
Da sich lokale Module im selben Prozess befinden, können diese Variablen gemeinsam nutzen und die Methoden des jeweils anderen ohne Serialisierung oder Vernetzung aufrufen. Dies ermöglicht die schnellstmögliche und maximal effiziente Kommunikation zwischen den Modulen.

2.3.4 NAOqi Speicherverwaltung

Die Speicherverwaltung in NAOqi wird von dem Modul `ALMemory` übernommen. Dieses Modul ist ein Speicher welcher sowohl Daten als auch Ereignisse beinhaltet. Dieser Speicher wird von allen Modulen geteilt und ermöglicht es, Daten zwischen den Modulen auszutauschen. Zusätzlich werden in diesem Speicher auch alle Ereignisse zwischengespeichert, auf welchen dann die abonierten Module zugreifen können, sobald das Ereigniss ausgelöst wurde. Alle Module haben auf diesen Bereich sowohl Lese- als auch Schreibzugriff. Jedoch ist das `ALMemory` Modul kein Echtzeitsynchronisationstool. Abonieren auf Bewegungsdatendaten oder Echtzeitvariablen in diesem Speicher ist also risikobehaftet.
Das Modul selbst ist ein Array aus sogenannten `ALValue`-Objekten, auf welche threadsicher zugegriffen werden kann. In diese Speicherbausteine können alle gängigen Datentypen gelegt werden. Dazu zählen einfache Datentypen wie einzelne Bits, Ganzzahlen, Fließkommazahlen und Zeichenketten, aber auch komplexere Datenstrukturen wie Arrays, Matrizen und Binaries.

Standardmäßig wird zwischen drei verschiedenen Datentypen unterschieden:

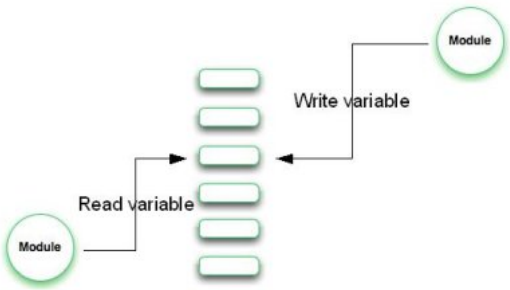

Abbildung 2.6: NAOqi Framework Speicherverwaltung

- Daten von Sensoren und Gelenken: diese Daten werden vom Roboter selbst in den Speicher geschrieben und haben keine Historie. Sie sind nicht größer als 32-Bit und können direkt per Pointer aufgerufen werden.
- Ereignisse: auf welche aboniert werden muss. Sobald eines dieser Ereignisse ausgelöst wird, wird es im Speicher gespeichert und kann von den abonierten Modulen abgerufen werden. Im Gegensatz zu den Daten von Sensoren und Gelenken sowie den Micro-Ereignissen, haben diese Ereignisse eine Historie und können auch nach dem Auslösen noch abgerufen werden.
- Micro-Ereignisse: Diese Ereignisse sind sehr kurzlebig und werden sofort nach dem Auslösen wieder gelöscht. Sie werden nicht im Speicher gespeichert und haben daher keine Historie.

2.4 Middleware

Middleware ist eine Software, die zwischen Anwendungen und Betriebssystemen vermittelt. Sie ermöglicht die Kommunikation zwischen verschiedenen Anwendungen, die auf unterschiedlichen Plattformen oder in unterschiedlichen Netzwerken ausgeführt werden. Middleware bietet eine Reihe von Diensten, die die Verbindung von Anwendungen erleichtern, die eigentlich nicht dafür vorgesehen sind. Dadurch, sowie durch Bereitstellung von weiteren Diensten wie etwa Sicherheit, Skalierbarkeit und Zuverlässigkeit, rationalisiert Middleware die Entwicklung, Bereitstellung und Wartung von Anwendungen.
Middleware existiert in verschiedenen Formen, wie etwa Nachrichtenbroker oder Transaktionsverarbeitungsmonitore, welche jeweils auf spezifische Kommunikationsformen zugeschnitten sind. Andere, wie etwa Webanwendungsserver oder Middleware für mobile Geräte, bieten ein breites Spektrum an Kommunikations- und Konnektivitätsfunktionen, die für die Entwicklung bestimmter Anwendungen im Mobilebereich unabdinglich sind. Wieder andere, wie etwa der Enterprise Service Bus (ESB), fungiert als zentraler Integrationsknotenpunkt, der alle Komponenten in einem Unternehmen miteinander verbindet.

Zusätzlich können aber auch persönliche Middleware selbst entwickelt werden, für personalisierte Anwendungen.
Der Begriff "Middleware"wurde geprägt, da die erste Generation dieser oft als "Vermittlerßwischen einem Anwendungs-Frontend zum Beispiel eines Clients und einer Backend-Ressource, in welcher Daten gespeichert oder verarbeitet werden, fungierte.
Die moderne Middleware heutzutage beinhaltet jedoch durchaus einen weitaus breiteren Aufgabenbereich. Beispielsweise umfasst Portal-Middleware sowohl das Frontend der Anwendung als auch Tools für die Backend-Konnektivität, während Datenbank-Middleware in der Regel einen eigenen Datenspeicher enthält [Microsoft Azure Accessed: 2024-05-05].
In diesem Projekt wird die Middleware ROS 2 genutzt, welche speziell für die Entwicklung von Robotersoftware entwickelt wurde und daher eine Vielzahl von Funktionen und Diensten bietet, die speziell auf die Anforderungen derer zugeschnitten sind.

2.5 Robot Operating System 2

ROS 2 ist eine Middleware zur entwickeln von Software für Roboter. Sie ist stark typisiert und basiert auf einem anonymen Publish-Subscribe-Mechanismus, welcher die Weitergabe von Nachrichten zwischen verschiedenen Prozessen ermöglicht.
Das Herzstück eines jeden ROS 2-Systems ist der ROS-Graph. Er bezieht sich auf das Netzwerk von Knoten in einem ROS-System und die Verbindungen zwischen ihnen, über welche sie komunizieren [ROS Documentation Contributors Accessed: 2024-05-05[e]].
Im Folgenden wird näher auf die in diesem Projekt verwendeten Konzepte und Technologien von ROS 2 eingegangen.

2.5.1 Nodes

Eine Node, auch Knoten genannt, ist eine Nutzer-Bibliothek zur Kommunikation mit anderen Nodes. Diese können mit anderen Nodes, innerhalb eines Prozesses, in einem anderen Prozess oder auf einem anderen Rechner kommuniziert werden. Sie sind häufig die Einheit der Berechnung in einem ROS-Graphen. Dabei sollte jede Node eine logische Aufgabe erfüllen.
Nodes haben die Möglichkeit, bestimmte Topics zu veröffentlichen, um Daten an andere Nodes zu übermitteln oder bestimmte Themen zu abonnieren (subscriben), um Daten von anderen Nodes zu erhalten. Es ist möglich, dass Sie entweder als Service-Client agieren, um eine Berechnung in ihrem Namen durchzuführen, oder als Service-Server, um anderen Nodes Funktionen zur Verfügung zu stellen. Eine Node kann als Aktionsrechner fungieren, um eine weitere Node zu beauftragen, die Berechnung in ihrem Namen durchzuführen oder als Aktionsserver für wiederum weitere Nodes als Funktionen bereitzustellen. Nodes können zusätzlich konfigurierbare Parameter bereitstellen, um das Verhalten während der Laufzeit individuell anzupassen.

Eine Node ist oft eine komplexe Kombination aus Publishern, Subscribern, Service-Servern, Services-Clients, Action-Servern und Action-Servern und werden über einen verteilten Erkennungsprozess identifiziert [ROS Documentation Contributors Accessed: 2024-05-05[b]].
Eine solche Node wird in diesem Projekt für die Kommunikation und Übersetzung verschiedener Daten für den Pepper Roboter entwickelt.

2.5.2 Topics

Topics sind ein Mechanismus, um Nachrichten zwischen Nodes zu übertragen. Sie sind asynchron, strikt typisiert, anonym und ermöglichen die Kommunikation zwischen Nodes, ohne dass diese voneinander Kenntnis haben müssen. Dies ist möglich durch die Nutzung des Publish-Subscribe-Prinzip. Dieses wurde zwar schon in den Abschnitten Abschnitt 2.3 und speziell Abschnitt 2.2 erläutert, wird jedoch hier noch einmal im Kontext von ROS 2 aufgegriffen.
In ROS 2 können Nodes sowohl Produzenten sein und Daten in ein Topic veröffentlichen, als auch Konsumenten, die Daten aus einem Topic abonnieren. Die Nodes selbst können dabei beliebig viele Topics abonnieren oder veröffentlichen.
Dabei ist der Begriff Topic lediglich der Name unter dem die Nachrichten veröffentlicht und auch abgerufen werden können. Die Nachrichten selbst sind dabei in einem strikt typisierten Format, welches als Message bezeichnet und definiert wird und im Abschnitt 2.5.3 ganuer erläutert wird.
Werfen wir jedoch zunächst einen genaueren Blick auf die Deteils der Bedeutungen der Anonymität und der strikten typisierung im Kontext von ROS 2:

Anonymität

Die Anonymität in ROS 2 bezieht sich darauf, dass die Nodes, die Daten in ein Topic veröffentlichen oder von einem Topic abonnieren, nicht wissen, welche anderen Nodes ebenfalls auf dieses Topic zugreifen und primär von welcher Node sie überhaupt stammen. Dies ermöglicht eine lose Kopplung zwischen den Nodes, da sie nicht auf die Existenz oder den Zustand der anderen Nodes angewiesen sind. Dadurch werden ROS 2 Systeme flexibel und Nodes können durch die genannte Entkopplung völlig frei entwickelt, ausgetauscht und aus technischer Sicht auch gelöscht werden.

Strikte Typisierung

ROS 2 Nodes können zwar sowohl in Python als auch in C++ entwickelt werden, was zumindest auf Seiten der Python-Entwicklung eine dynamische Typisierung implizieren könnte, jedoch bildet sich diese Freiheit nicht auf die Nachrichten (Messages) ab, da diese auch im Raum der in C++ strikt typisiert ist, genutzt werden müssen. Dadurch ergeben

sich die in den Interfaces definierten, strikten Typen, die von den Nodes eingehalten werden müssen. Dies ermöglicht eine einfache und effiziente Kommunikation zwischen den Nodes, sollten diese auch in verschiedenen Sprachen implementiert sein, da die Nachrichten immer in einem festen Format vorliegen und die Nodes sich darauf verlassen können, dass die Nachrichten korrekt sind.
Weiter werden die Nachrichten auch semantisch strikt typisiert, was bedeutet, dass die Nachrichten auch inhaltlich korrekt sind und die Nodes sich darauf verlassen können, dass die Nachrichten die erwarteten Daten enthalten. Ein anschauliches Beispiel dafür sind die sogenannten Inertial Measurement Unit (IMU)-Messages. Sie enthalten als Datentyp ein dreidimensionales Array aus Fließkommazahlen, welche die Beschleunigung, die Winkelgeschwindigkeit und die Magnetfeldstärke eines Sensors in den drei Raumrichtungen beschreiben. Also ist nicht nur der Datentyp strikt festgelegt, sondern auch die einzelnen Werte und deren Reihenfolge, die in den Arrays enthalten sind [ROS Documentation Contributors Accessed: 2024-05-05[d]].

2.5.3 Interfaces

ROS 2 kommuniziert typischerweise über verschiedene Schnittstellen (Interfaces), die als Messages, Services und Actions bezeichnet werden. ROS 2 verwendet dabei eine vereinfachte Beschreibungssprache, um die Interfaces zu definieren, die als Interface Definition Language (IDL) bezeichnet wird. Diese Beschreibungssprache wird verwendet, um die Interfaces in verschiedenen Programmiersprachen zu generieren [ROS Documentation Contributors Accessed: 2024-05-05[a]].
Dabei bietet ROS 2 drei Typen von Interfaces welche folgend erläutert werden:

Messages

Messages werden in Topics genutzt und sind einfache Textdateien, die die Felder einer ROS-Nachricht beschreiben. Diese Nachrichten werden verwendet, um Source-Code für die Nachrichten zu generieren, die in verschiedenen Programmiersprachen verwendet werden können.
Die Messages sind eine Möglichkeit für Nodes, Daten über das Netzwerk mit weiteren Nodes auszutauschen, auch wenn keine Antwort erwartet wird. Als Beispiel kann eine Node eine Nachricht veröffentlichen, die die Daten eines Sensors enthält, und eine andere Node kann diese Nachricht abonnieren, ohne dass die erste Node eine Antwort erwartet oder überhaupt von der Existenz der zweiten Node weiß.
Die `.msg`-Dateien, welche die Messages beschreiben, werden in einem separaten Verzeichnis im ROS-Paket unter `msg/` abgelegt. Sie beinhalten die Felder der Nachricht und Konstanten. Inhalt dieser Felder können nahezu alle gängigen Datentypen sein, wie etwa Ganzzahlen (Integer), Fließkommazahlen (Float), einzelne Zeichen (Char), Zeichenketten (String) aber auch statische Arrays. Einzige Einschränkung sind hierbei die Regelungen (Conventions) von ROS 2, welche die Verwendung von alphabetischen Zeichen zu Beginn

und Unterstrichen zur Trennung einzelner Wörter im Namen der Felder vorschreiben.
Ein typischer Eintrag einer solchen Datei könnte, beispielhaft für die verschiedenen Felder, etwa wie folgt aussehen:

Algorithmus 2.1: ROS 2 Message

```
type name
int16 sensor_id
int32[] sensor_data

type name defaultvalue
bool enabled true

type CONSTANTNAME=constantvalue
string IP="192.168.100.11"
```

Services

Services sind im Gesensatz zu den Messages eine synchrone Kommunikation zwischen zwei Nodes. Ein Node sendet eine Anfrage an einen anderen Node und wartet auf dessen Antwort.
Sie werden ebenfalls in einem separaten Verzeichnis im ROS-Paket unter `srv/` als `.srv` Datei abgelegt und beschreiben die Anfrage und die Antwort, die von einem Service bereitgestellt wird. Auch hier folgt der Inhalt der Datei einem festgelegtem Schema ähnlich dessen der Message Dateien, mit dem Zusatz, dass hier getrennt von einer Zeile mit drei Bindestrichen als Inhalt `--`, auch die zu erwartende Antwort definiert wird.
Eine einfach gehaltene `.srv`-Datei könnte also etwa wie folgt aussehen:
Auch hier können wie bei den Messages auch komplexere Datenstrukturen, Konstanten

Algorithmus 2.2: ROS 2 Service

```
string myString
---
string yourString
```

und Standardwerte genutzt werden.

Actions

Actions sind eine Erweiterung der Services, die es ermöglichen, asynchrone, lang laufende Prozesse mit beidseitiger Kommunikation zu definieren. Im Gegensatz zu den Services, bei denen die Antwort direkt erwartet wird, können Actions mehrere Sekunden bis Minuten dauern. Diese Action kann in diesem Zeitraum auch unterbrochen und wieder aufgenommen werden, oder sogar ganz abgebrochen werden.
Auch Actions werden in einem separaten Verzeichnis im ROS-Paket unter `action/` als `.action` Datei abgelegt und beschreiben in einer ähnlichen Weise wie die Services und die Messages das erwartete Verhalten. In Gegensatz zu den beiden anderen Schnittstellen, wird hier jedoch nicht die Anfrage und die Antwort definiert, sondern ein Ziel, eine Rückmeldung und ein Feedback, welche im Schema gehalten durch einen Dreifachbindestrich `--` getrennt werden.
Nehmne wir zur Veranschaulichung die `Fibonacci .action`-Datei als Beispiel:
Es wird ein `Int32` abgeschickt und daraufhin ein Array aus wiederum `Int32` erwartet,

Algorithmus 2.3: ROS 2 Action

```
int32 order
---
int32[] sequence
---
int32[] sequence
```

mit den berechneten Werten. Während der Berechnung wird ein Feedback in Form eines `Int32` zurückgegeben, mit dem Stand bis zu einem bestimmten Zeitpunkt.

2.5.4 Parameter

Parameter in ROS 2 sind direkt verbunden mit den einzelnen Nodes. Sie werden genutzt, um die Konfiguration der Nodes beim Start un zur Laufzeit zu verwalten, ohne den darunterliegenden Code anpassen zu müssen. Die Lebenszeit der Parameter ist dabei an die Lebenszeit der Nodes gebunden, was bedeutet, dass die Parameter beim Start der Nodes geladen werden und beim Beenden der Nodes verfallen. Eine Node kann jedoch eine Art Persistenz implementieren um die Parameter zu speichern und wieder zu verwenden.
Jeder dieser Parameter besteht aus einem Schlüssel (Key), einem Wert (Value) und einer Beschreibung (Descriptor). Während der Schlüssel ein `String` ist und den Parameter identifiziert, ist der Wert der eigentliche Inhalt des Parameters und wird zunächst als Typ definiert. Die Beschreibung ist optional und Standardmäßig `NULL`. Sie dient dazu, den Parameter zu beschreiben und zu dokumentieren, wie Typ-Informationen, Wertebereiche, Nutzungsdeteils und weitere Informationen welche zur Nutzung relevant sein können.

Zu Beginn der Laufzeit muss einen Node definieren welche Parameter diese aktzeptiert, und ob diese optional oder zwingend sind. Dies reduziert die Anfälligkeit auf Fehler in der Konfiguration im späteren Verlauf der Laufzeit. Die Parameter können dann auch zur Laufzeit über die API der Nodes abgerufen, gesetzt und gelöscht werden. Einige Arten von Nodes können auch Parameter aktzeptieren bevor diese bekannt sind. Dazu stellt ROS 2 den Parameter `allow_undeclared_parameters` zur Verfügung, welcher es bei aktivierung ermöglicht, dass Parameter gesetzt werden können, auch wenn sie nicht explizit in der Parameterliste der Node definiert sind. Dies ist allerdings nur möglich bei neu hinzugefügten Parametern, bereits vorhandene Parameter können nicht zur Laufzeit verändert werden. Dies würde die Grundlage der strikten Typisierung und der semantischen Korrektheit der Parameter verletzen und damit das gesamte System instabillisieren. Versuche dies trotzdem zu tun, werfen einen Fehler und die Node wird sofort terminiert.
Soll der Wert eines Parameters zur Laufzeit geändert werden, gibt es dazu zwei Möglichkeiten:

- Die `set_parameters` Methode, welche über die API der Node aufgerufen wird und die Parameterwerte ändert. Diese Methode setzt die gewünschten Parameterwerte und gibt die tatsächlich gesetzten Werte zurück, auch wenn die Node die Werte nicht akzeptiert hat. Dies ermöglicht es, die Parameterwerte zu überprüfen und gegebenenfalls zu korrigieren oder gar abzulehnen.

- Die `on_parameter_event` Methode, welche aufgerufen wird, wenn sich ein Parameterwert ändert. Dadurch wird es der Node und dem User ermöglicht, auf Änderungen der Parameterwerte zu reagieren welche von anderen Nodes oder von der Command Line Interface (CLI) zur Laufzeit vorgenommen wurden.

Zur allgemeinen Interaktion mit den Parametern, bietet ROS 2 neben den bereits genannten, eine Vielzahl von API-Methoden, auch API-Services genannt, um die Parameter zu verwalten. Diese werden standardmäßig bei Initiierung einer Node geladen und bereitgestellt [ROS Documentation Contributors Accessed: 2024-05-05[c]].

2.5.5 Start

2.5.6 Nutzer Bibliotheken

2.6 Programming Languages

2.6.1 C++

2.6.2 Python

2.7 Virtual Reality

2.8 Entwicklung für Virtual Reality

2.9 TCP

Kapitel 3

Technologieauswahl

3.1 VR-Brillen

3.2 Entwicklung für VR-Brillen

Kapitel 4

Umsetzung

4.1 Pepper

4.2 MetaQuest3

4.3 Verbindung

Kapitel 5

Anwendungsgebiete

Das entstandene Produkt kann in verschiedenen Fällen eingesetzt werden.

5.1 Pflege

In Pflegeeinrichtungen können Beispielsweise Bewohner, die selbst nicht mehr so gut zu Fuß sind sich gegenseitig Besuchen und kommunizieren. Ebenfalls können Routinebesuche bei den Besuchern durch Pflegekräfte von einer zentralen Stelle aus getätigt werden, was die Pflegekräfte entlasten würde.

Kapitel 6

Fazit

Kapitel 7

Fortsetzung des Projekts

Literatur

Carnegie Mellon University [2024]. *NAOqi Documentation*. https://www.cs.cmu.edu/~cga/nao/doc/reference-documentation/dev/naoqi/index.html.

Dante [Jan. 2010]. *Webseite der* Deutschsprachige Anwendervereinigung TeX e.V. http://www.dante.de.

Elektronik-Kompendium.de [o. D.] *Publish-Subscribe-Modell*. https://www.elektronik-kompendium.de/sites/net/2204051.htm. [Online; accessed 22-April-2024] [siehe S. 11].

ITWissen.info [o. D.] *Publish-Subscribe-Modell (publish-subscribe)*. https://www.itwissen.info/Publish-Subscribe-Modell-publish-subscribe.html. [Online; accessed 22-April-2024] [siehe S. 11].

Knuth, Donald E. [1984]. *The TEXbook*. Addison-Wesley.

Lamport, Leslie [1995]. *Das LaTeX Handbuch*. Addison-Wesley.

Microsoft Azure [Accessed: 2024-05-05]. *What is Middleware?* https://azure.microsoft.com/en-gb/resources/cloud-computing-dictionary/what-is-middleware/ [siehe S. 18].

ROS Documentation Contributors [Accessed: 2024-05-05[a]]. *ROS - About Interfaces*. https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html [siehe S. 20].

— [Accessed: 2024-05-05[b]]. *ROS - About Nodes*. https://docs.ros.org/en/humble/Concepts/Basic/About-Nodes.html [siehe S. 19].

— [Accessed: 2024-05-05[c]]. *ROS - About Parameters*. https://docs.ros.org/en/humble/Concepts/Basic/About-Parameters.html [siehe S. 23].

— [Accessed: 2024-05-05[d]]. *ROS - About Topics*. https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html [siehe S. 20].

— [Accessed: 2024-05-05[e]]. *ROS - Basic Concepts*. https://docs.ros.org/en/humble/Concepts/Basic.html [siehe S. 18].

University, Carnegie Mellon [Year of access]. *Nao Developer Documentation: C++ SDK Installation Guide*. https://www.cs.cmu.edu/~cga/nao/doc/reference-documentation/dev/cpp/install_guide.html. Accessed: April 22, 2024 [siehe S. 11].

Wikipedia [o. D.] *Publish–subscribe pattern.* `https://en.wikipedia.org/wiki/PublishâĂŞsubscribe_pattern`. [Online; accessed 22-April-2024] [siehe S. 10].