Freie Universität Berlin
Technische Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Entwurf und Entwicklung eines Systems zur Visualisierung und Steuerung von Industrierobotern auf mobilen Endgeräten

Friedrich Ueberreiter
friedrich@ueberreiter.com

28. Mai 2018

Betreuer:
Prof. Dr. Daniel Göhring
Prof. Dr. Jörg Krüger

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Datum Unterschrift

Abstract

Das System zur einfachen und sicheren Steuerung von Industrierobotern wurde entwickelt um eine flexible Kommunikation zwischen verschiedenen Diensten zu ermöglichen. Dabei ist besonderer Wert auf Einfachheit, Performance und Erweiterbarkeit gelegt worden. Die Bedienung der Roboter erfolgt über das Netzwerk mittels einer HoloLens. Diese zeigt zur Visualisierung Hologramme an mit deren Hilfe der Benutzer über definierte Gesten den Roboter ausrichten kann. Die Augmented Reality ermöglicht es intuitiv mit dem Roboter zu interagieren und ihn Aufgaben erledigen zu lassen. Letztlich wurde das Konzept mittels eines Versuches und Software-Tests validiert.

Inhaltsverzeichnis

1. Einführung

In den letzten Jahren hat sich die Art der Produktion von Gütern gewandelt. Viele Unternehmen haben ihre Produktionsstätten ins Ausland verlagert um ihre Produkte günstiger anfertigen zu können. Der Trend der eigenen Fertigung im Ausland entwickelte sich durch kostengünstigere Alternativen weiter. Immer mehr Unternehmen geben ihre Produktion vollständig an Drittunternehmen bzw. Auftragsfertiger ab. Diese produzieren für eine Vielzahl von Firmen die erwünschten Güter. Der aktuelle Markt fordert mehr qualitativ hochwertige und individuelle Produkte zu niedrigen Preisen. (AP, 2003; Creutzburg, 2015)

Weiterhin werden nahezu jährlich neue Versionen von Konsumgegenständen auf den Markt gebracht. Daraus resultieren kürzere Lebenszyklen der jeweiligen Waren. Dieser Wandel sorgt dafür, dass Auftragsfertiger ihre Produktionsstätte immer wieder überarbeiten und flexibel gestalten müssen ohne an Qualität zu verlieren. Die erwünschte Anpassungsfähigkeit soll sowohl in der Produktion als auch im Umgang mit wechselnden Kunden mit Hilfe von modernen Industrierobotern erfolgen. In immer mehr Bereichen kommen diese zum Einsatz, was sich in einer steigenden Gesamtzahl der weltweit verkauften Industrieroboter bemerkbar macht (Abb. 1). Der in der Grafik ersichtliche Einbruch im Jahr 2009 ist der weltweiten Wirtschaftskrise zuzuschreiben. (Gemma et al.)

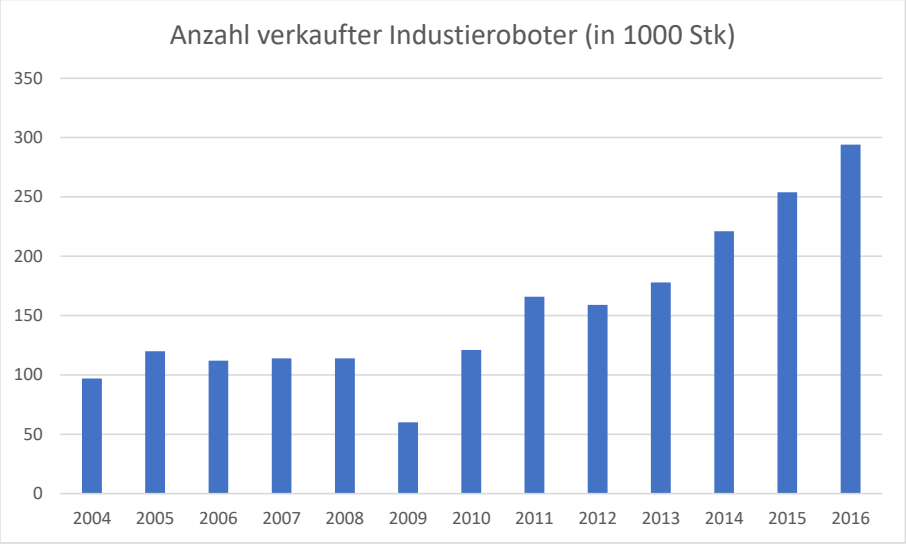

Abbildung 1: Anzahl verkaufter Industrieroboter zwischen den Jahren 2004 und 2016

Die Unternehmen sollen selbst in der Lage sein, bei Auftragsfertigern verteilte Industrieroboter flexibel einzusetzen. Diese müssen zum einen einfach und gezielt gesteuert und zum anderen unkompliziert programmiert werden können.

Im Rahmen dieser Arbeit wird der Prototyp eines solchen Systems mittels Augmented Reality umgesetzt. Durch den indirekten Kontakt soll Mitarbeitern ein sicherer Umgang mit Industrierobotern ermöglicht werden. Um den unterschiedlichsten Gegebenheiten gerecht zu werden, müssen die Industrieroboter und zahlreiche Dienste miteinander kommunizieren. Diese Mehrwertdienste vereinfachen die Steuerung der Roboter und führen

Aufgaben wie bspw. eine Bahnplanung aus. Eine flexible Netzwerkarchitektur, die es ermöglicht verschiedene Dienste für die Roboter und Endgeräte zur Steuerung und Überwachung bereitzustellen, ist gefordert. Dabei werden im Prototyp unter anderem die Aspekte Zugangsschutz sowie Verschlüsselung vernachlässigt. Diese Themen sind für den produktiven Einsatz unablässig und müssen vor Verwendung des Systems implementiert werden.

Zu Beginn der Ausarbeitung werden grundlegende Informationen zum weiteren Verständnis aufgeführt. Anschließend werden bereits bestehende Systeme der Technischen Universität Berlin analysiert. Komponenten wie Steuerungssysteme werden im entwickelten Netzwerk integriert und betrieben. Ein Konzept, das die Kommunikation zwischen den Teilnehmern sowie die Visualisierung und Steuerung der Industrieroboter ermöglicht, wird erarbeitet. Im darauf folgenden Kapitel wird das Konzept umgesetzt und implementiert. Zur Validierung des Systems wird zum einen der Industrieroboter mittels einer HoloLens programmiert einen Gegenstand von einer Position auf eine andere zu platzieren. Zum anderen werden Softwaretests durchgeführt. Im Fazit wird auf weitere und zukünftige Entwicklungsmöglichkeiten eingegangen.

2. Grundlagen

2.1. Industrielle Roboter

„Industrieroboter sind universell einsetzbare Bewegungsautomaten mit mehreren Achsen, deren Bewegungen hinsichtlich Bewegungsfolge und Wegen bzw. Winkeln frei programmierbar (d.h. ohne mechanischen Eingriff vorzugeben bzw. änderbar) und gegebenenfalls sensorgeführt sind. Sie sind mit Greifern, Werkzeugen oder anderen Fertigungsmitteln ausrüstbar und können Handhabe- oder andere Fertigungsaufgaben ausführen.“ (Weber, 2002)

Diese Definition eines Industrieroboters nach VDI-Richtlinie 2860 ist eine der am häufigsten verwendeten. Weitere sind durch die Japan Industrial Robot Association (JIRA) und europäischen Norm EN775 bestimmt. Während die JIRA den Industrieroboter umfangreicher definiert, decken sich die Definitionen der VDI-Richtlinie und der EN775 inhaltlich.

Im Rahmen der Arbeit wurden steuerbare und zugleich einarmige Industrieroboter verwendet. Diese können in diversen Aufgabenbereichen eingesetzt werden. Durch entsprechende Zubehörteile sind die Roboter in der Lage Gegenstände zu greifen und an einen anderen Platz zu legen. Ein Beispiel dafür wäre das Legen von Produkten von einem Fließband in einen Karton. Eine weitere Möglichkeit wäre, die Roboter innerhalb der Produktionskette einzusetzen um bspw. Oberflächen zu polieren, lackieren oder schweißen. Für die Implementierung der Steuerung wird ein grundlegendes Wissen über die Funktionsweise der Gelenke und die mathematischen Berechnungen dieser benötigt.

Der verwendete Industrieroboter wird von Universal Robots gebaut und trägt die Bezeichnung UR-5. Insgesamt verfügt dieser über sechs Gelenke, die sich jeweils um die eigene Achse drehen können (Abb. 2). Alle sechs Gelenke sind über statische Elemente miteinander verbunden, man spricht von einer Gelenkskette oder *Joint Chain*. Am Ende der Kette befindet sich ein Endeffektor. Dieser ist eine unspezifische Beschreibung für das Werkzeug des Roboters. Das Werkzeug kann je nach Arbeitsaufgabe des Industrieroboters und Einsatzgebiet bspw. ein Greifer, Schweiß- oder Poliergerät sein. Genau wie bei einem menschlichen Arm ist die Position des Endeffektors von der Stellung der Gelenke im Arm des Roboters abhängig. Der verwendete Roboter besitzt Gelenke mit jeweils einem Freiheitsgrad. Das bedeutet, dass sich die Gelenke jeweils um eine feste Achse drehen können. Folglich hat der gesamte Roboter sechs Gelenke mit insgesamt sechs Freiheitsgerade. Mit der erreichten Beweglichkeit kann der Roboter flexibel innerhalb seines Arbeitsraums eingesetzt werden.

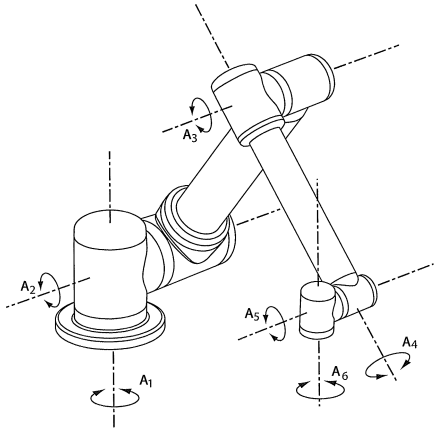

Fig. 27

Abbildung 2: Die sechs Gelenke des UR5

Die Größe des Arbeitsraumes ist durch Maße wie die Länge des Roboterarmes begrenzt. Der gesamte Arbeitsraum des Roboters setzt sich aus allen durch seinen Endeffektor erreichbaren Positionen zusammen. Die Position wird in Kartesischen Koordinaten angegeben, man spricht auch vom Kartesischen Raum oder *Cartesian space*. Sowohl Translation im Raum entlang der Achsen x, y und z, als auch die Rotationen *yaw*, *pitch* und *roll* (Abb. 3) werden entweder als zwei 3-dimensionale Vektoren oder ein 6-dimensionaler Vektor dargestellt.

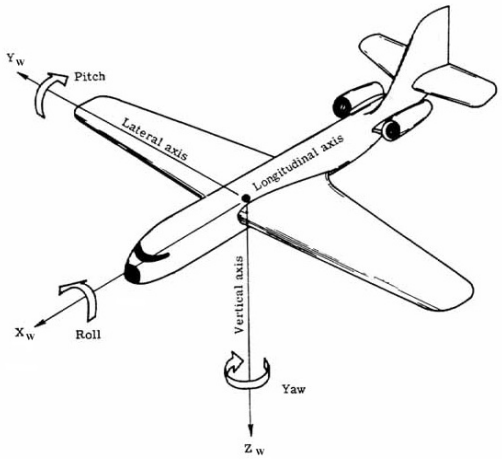

Abbildung 3: Die drei Achsen der Rotation

Die Position des Endeffektors bei A_6 (Abb. 2) ist von der Stellung aller Gelenke abhängig. Die aktuelle Position des Endeffektors wird mit Hilfe der einzelnen Gelenke A_1 bis A_6 und einer Vorwärtskinematik Matrix berechnet. Der Winkel eines Gelenkes A_i beträgt θ_i. Über alle Winkel und den jeweils dazugehörigen Transpositionsmatrizen wird das Produkt gebildet. Allgemein gilt: handelt es sich um einen Roboter mit n Gelenken, die sich jeweils

um θ_i drehen, ergibt sich für die Matrix ${}^0_nT$:

$$ {}^0_nT = \prod_{i=1}^{n} {}^{i-1}_iT(\theta_i) $$

Die Transpositionsmatrix ${}^{i-1}_iT$ vom $i-1$-ten zum i-ten Gelenk enthält sowohl einen Translations- als auch Rotationsanteil:

$$ {}^{i-1}_iT = R(z_{i-1}, \theta_i) \cdot T(z_{i-1}, d_i) \cdot T(x_{i-1}, a_i) \cdot R(x_i, \alpha_i) $$

Die Matrizen $T(x_{i-1}, a_i)$ und $R(x_i, \alpha_i)$ beschreiben die Translation und Rotation um die lokale x-Achse, die Matrizen $T(z_{i-1}, d_i)$ sowie $R(z_{i-1}, \theta_i)$ hingegen die Translation und Rotation um die z-Achse. Der Parameter θ_i ist bei dem UR-5 von der aktuellen Position des Gelenkes abhängig, die Parameter für d_i, a_i sowie α_i können aus der Tabelle 1entnommen werden. Mit der Matrix ${}^0_6T$ für den UR-5 kann mit Hilfe von gegebenen Gelenkswerten $\theta_1, ..., \theta_6$ die Position des Endeffektor berechnet und visualisiert werden. (Paul, 1981)

Gelenk	a	α	d	θ	Offset
1	0	$\frac{\pi}{2}$	0.089159	q_1	0
2	-0.425	0	0	q_2	$-\frac{\pi}{2}$
3	-0.39225	0	0	q_3	0
4	0	$\frac{\pi}{2}$	0.10915	q_4	$-\frac{\pi}{2}$
5	0	$-\frac{\pi}{2}$	0.09465	q_5	0
6	0	0	0.0823	q_6	0

Tabelle 1: Denavit-Hartenberg-Parameter des UR-5

Umgekehrt wird die Position der Gelenkswinkel für eine gegebene Position des Endeffektors mit Hilfe der inversen Kinematik berechnet. (Abb. 4) (Cubero, 2007) Dies wird benötigt, wenn die Endeffektorposition gegeben ist und eine mögliche Gelenkskonfiguration dazu gesucht wird.

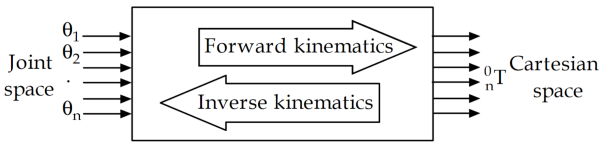

Abbildung 4: Umrechnung vom *Joint Space* in das Kartesische Koordinatensystem

Die Berechnung der inversen Kinematik ist analytisch komplex. Daher verwendet man einen algorithmischen Ansatz. Der verwendete Algorithmus für den UR-5 wird von Universal Robots zur Verfügung gestellt und ist unter „ur kinematics at indigo universal robots“ (2018) nachvollziehbar.

Mit Hilfe der Vorwärtskinematik und der inversen Kinematik lassen sich alle Größen wie

Gelenkswinkel berechnen um Steuerbefehle umzusetzen.

Eine weitere wichtige Art der Steuerung von Industrierobotern ist die Bahnplanung. Ein Roboter kann unterschiedliche Formen von Bahnen abfahren. Die einfachste Form ist die lineare Trajektorie ohne Kollisionserkennung. Der Roboter fährt eine Strecke zwischen zwei definierten Punkten A und B entlang. (Abb. 5)

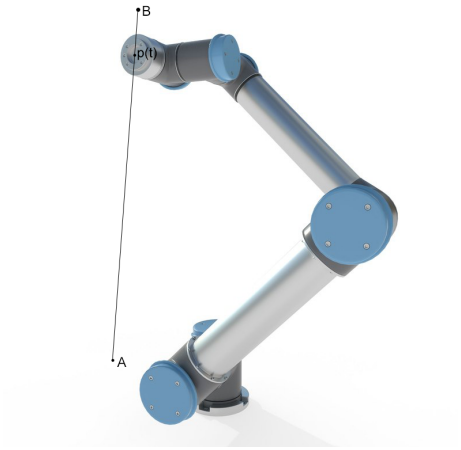

Abbildung 5: Lineare Trajektorie zwischen den Punkten A und B

Der aktuelle Punkt zum Zeitpunkt t auf der Bahn $p(t)$ wird berechnet durch:

$$p(t) = p_A + \frac{s(t)}{\|p_A p_B\|} \cdot (p_B - p_A)$$

p_A und p_B sind die Vektoren zu den Punkten A und B. $\|p_A p_B\|$ ist der Abstand zwischen A und B. Die Funktion $s(t)$ gibt an, wie viel Strecke zum Zeitpunkt t bereits zurückgelegt wurde. $s(t)$ ist von der maximalen Geschwindigkeit und Beschleunigung des verwendeten Roboters abhängig. Die zurückgelegte Strecke, die Geschwindigkeit und Beschleunigung in Abhängigkeit von der Zeit (Abb. 6) werden mit Hilfe der Bibliothek KDL berechnet. („Kinematic and Dynamic Solvers | The Orocos Project“, 2018)

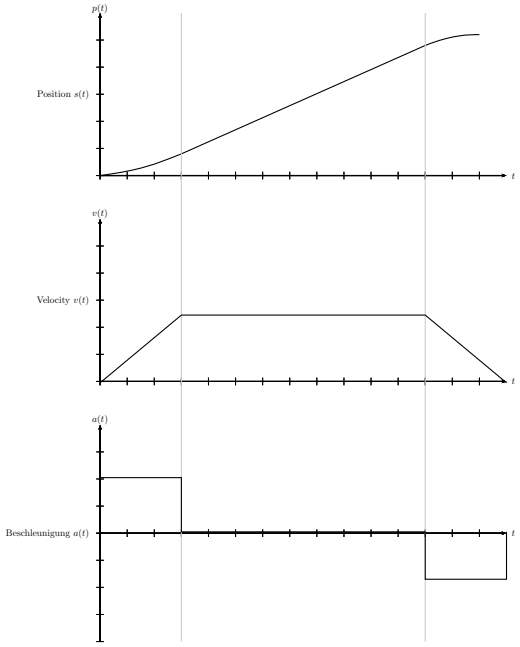

Abbildung 6: Strecke, Geschwindigkeit und Beschleunigung in Abhängigkeit von der Zeit

2.2. Visualisierung

Zur Visualisierung der Industrieroboter bieten sich unterschiedliche Geräte an. Ein Roboter kann als 3D Modell auf einem Computer oder Tablet angezeigt werden. Seit dem Jahr 2016 gibt es eine noch anschaulichere Möglichkeit diese darzustellen.

Sogenannte Head-Mounted-Displays, die ähnlich wie Brillen getragen werden, werden immer praxistauglicher. Man unterscheidet zwischen zwei Kategorien: *Virtual Reality* und *Augmented Reality*. Die *Virtual Reality* simuliert dem Nutzer sämtliche visuelle Eindrücke. Er sieht nicht mehr eine reelle Umgebung, sondern ausschließlich eine Virtuelle. Wände, Objekte und Personen werden vollständig als 3D Modelle dargestellt. Beispiele für *Virtual Reality* Brillen sind Oculus Rift oder die HTC Vive. („Oculus Rift | Oculus“, 2018; „VIVE | Discover Virtual Reality Beyond Imagination“, 2018)

Im Gegensatz dazu wird in der *Augmented Reality*, zu Deutsch erweiterte Realität, die Umgebung wie gewohnt dargestellt. Zusätzlich können mit Hilfe der Brille Hologramme im Blickfeld platziert werden. Bspw. kann auf einem Tisch im Raum das 3D Modell eines Industrieroboters angezeigt und von allen Seiten betrachtet werden. Die HoloLens von Microsoft ist auf dem Markt seit Oktober 2016 erhältlich. Sie erkennt Gesten, die mit den Händen ausgeführt werden. Auf diese Weise kann der Benutzer an Hologrammen ziehen, diese antippen und manipulieren.

Weiterhin ist die HoloLens mit zahlreichen Sensoren ausgestattet. Sie verfügt über eine Sensorleiste, die sich vorne oben an der Brille befindet. (Abb. 7) Darin enthalten sind vier Umgebungskameras, die sich paarweise seitlich an der Sensorleiste befinden. Sie dienen dazu die Kopfbewegung in Relation zum Raum zu ermitteln. Des Weiteren enthält die Leiste eine *Inertial Measurement Unit* (IMU), welche einen Beschleunigungsmesser, Rotationsmesser

und Magnetometer beinhaltet. Hinzu kommt eine Tiefenkamera, die sich vorne zentral befindet. Diese wird zur Hand- und Oberflächenerkennung, auch Spatial Mapping genannt, genutzt. Ein Umgebungslicht-Sensor, eine Kamera für Foto- und Videoaufnahmen sowie vier Mikrofone komplettieren die Sensorik der HoloLens.

Abbildung 7: Sensorleiste der HoloLens

Die HoloLens wird zur Visualisierung und Steuerung des verwendeten Industrieroboters genutzt. Bspw. wird der Endeffektor mit Hilfe von Gesten in eine bestimmte Position gezogen und gedreht. Der holografische Roboterarm stellt die Gelenkswinkel automatisch ein, sodass die gewünschte Position erreicht wird. Die Winkel des Armes müssen ebenfalls manuell eingestellt werden. Das Resultat, die Position des Endeffektors, muss berechnet werden. Hierfür werden die Vorwärtskinematik und die inverse Kinematik (Kapitel 2.1) des Roboters benötigt.

Wird der holografische Roboter in die korrekte Position gebracht, nimmt der reelle Roboter nach einer Bestätigung durch den Benutzer diese Position ein. Um dies ausführen zu können, muss die HoloLens über ein Netzwerk mit dem Roboter verbunden sein. Handelt es sich um einen einzigen Roboter, ist es möglich die HoloLens dermaßen zu programmieren, dass dieser direkt angezeigt und gesteuert werden kann. Handelt es sich um einen Industriepark aus mehreren Robotern, die möglicherweise unterschiedlicher Art sind, ist eine feste und direkte Verbindung zwischen Roboter und HoloLens eine unbefriedigende Lösung.

2.3. Netzwerke

Ein Netzwerk mit vielen unterschiedlichen Teilnehmern ist ein verteiltes Netzwerk. Die Teilnehmer können auf gleichen oder verschiedenen Hardware-Geräten laufen. Zusätzlich können die Teilnehmer unterschiedlicher Natur sein. Beispiele für Teilnehmer eines verteilten Netzwerkes sind Roboter, eine HoloLens oder Geräte wie Smartphone, Tablet oder Computer. Verschiedene Anzeigegeräte sollen mit diversen Industrierobotern kommunizieren und diese darstellen und steuern.

Die vollständige Netzwerkarchitektur kann vereinfacht über das *Open Systems Interconnection Model* (OSI-Modell) dargestellt werden. (ITU-T, 1994) Das Modell beschreibt

dabei sieben Schichten: Physical, Data Link, Network, Transport, Session, Presentation und Application. Die physikalische Topologie wird von den ersten beiden Layern beschrieben. Dabei sind bspw. Leitungen und Stecker Teil der physikalischen Schicht und der Switch in Kombination mit dem MAC-Protokoll Teil der Sicherungsschicht (Data Link Layer). Die Kommunikation auf diesen beiden Schichten wird vernachlässigt und als gegeben betrachtet.

Auf Schicht drei, der Netzwerkschicht, wird in dem Anwendungsszenario das Internet-Protokoll *IP* verwendet, auf Schicht vier, der Transportschicht, das Transmission Control Protocol *TCP*. Die Schichten fünf bis sieben werden als Anwendungsschicht zusammengefasst, in der zwei Protokolle eingesetzt werden. Der Großteil der Kommunikation erfolgt über das von Google entwickelte Protokoll *Protocol Buffers*. In wenigen Ausnahmen wird das bekannte Hypertext Transfer Protocol *HTTP* genutzt.

Auf Schicht drei und vier beruht die logische Topologie, das Overlay-Netzwerk. Hierbei gibt es zahlreiche verschiedene Formen bzw. Architekturen. Gängige Formen sind Peer-to-Peer oder Client-Server Architekturen. In der Peer-to-Peer Architektur sind alle Teilnehmer gleichberechtigt. Die Kenntnisse der einzelnen Teilnehmer übereinander muss dabei nicht vollständig sein. (Abb. 8) Sie können innerhalb des Netzwerkes Dienste bereitstellen und nutzen. Peer-to-Peer Netzwerke zeichnen sich durch ihre beachtliche Robustheit und Performance gegenüber anderen Architekturen aus. (Bawa et al., 2003; Schollmeier, 2001)

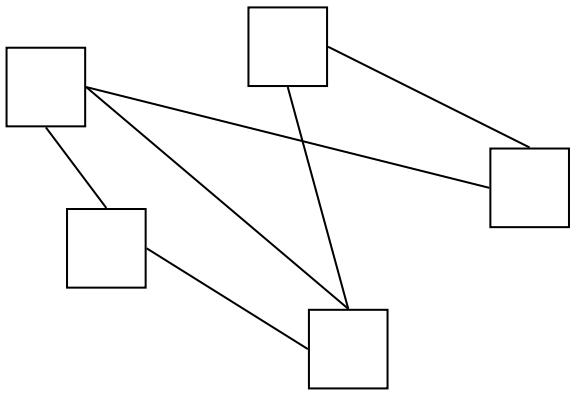

Abbildung 8: Peer-to-Peer Netzwerk

Da es keine feste Anlaufstelle innerhalb einer Peer-to-Peer Architektur gibt, müssen Funktionen für die Suche innerhalb eines Netzwerkes zur Verfügung gestellt werden. In diesem Bereich liegen die Schwächen der Peer-to-Peer Architektur. (Cooper and Garcia-Molina, 2004) Die Suche von Diensten im Netzwerk muss aufwendig implementiert werden, da nicht jedem Teilnehmer alle weiteren Teilnehmer des Netzwerkes bekannt sind. Benötigt bspw. eine HoloLens eine Liste von allen Teilnehmern mit bestimmten Eigenschaften, muss diese zunächst alle Teilnehmer des Netzes finden. Anschließend überprüft die HoloLens, ob die ermittelten Teilnehmer jeweils die Eigenschaften erfüllen.

Demgegenüber steht die Client/Server-Architektur. In dieser gibt es den Server als zentrale Anlaufstelle. Er vermittelt sämtliche Anfragen und Antworten zwischen den Teilneh-

mern. Benötigt ein Gerät eine Liste von bestimmten Teilnehmern, wird diese beim Server angefragt. Die Kommunikation innerhalb des Netzwerkes läuft ausnahmslos über den Server. (Abb. 9) Dies stellt einen erheblichen Nachteil des Netzwerkes dar.

Während in Peer-to-Peer Netzwerken die Kommunikation direkt erfolgt, ist in der Client/Server Architektur stets der Server dazwischen geschaltet. Sowohl Bandbreite als auch Ausfallsicherheit sind in der Client/Server-Architektur deutlich geringer. Bei einem Serverausfall bricht das gesamte Netzwerk zusammen. Verfügt der Server über eine geringe Bandbreite, ist das gesamte Netzwerk überlastet. Im Gegensatz dazu ist die einfache Durchsuchbarkeit des Netzwerkes von Vorteil, da der Server sämtliche Teilnehmer kennt.

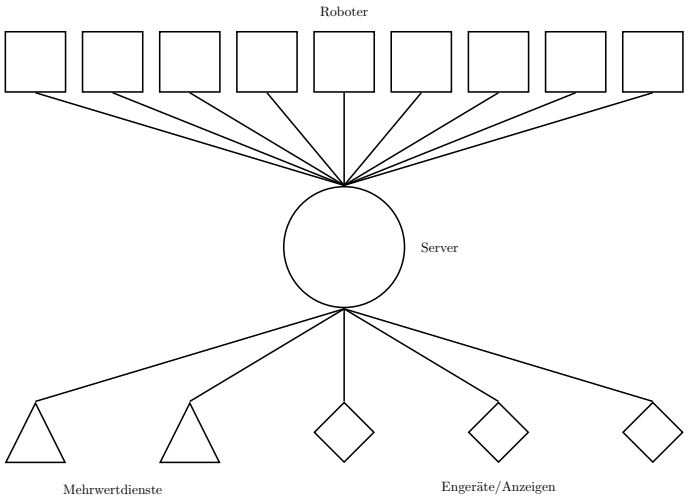

Abbildung 9: Client/Server-Architektur

Auf der Anwendungsebene wird *Googles Protocol Buffers* verwendet. Dies ist ein Format zur serialisierten Übertragung von Daten. Geläufige Formate wie XML oder JSON sind textbasiert. In Protocol Buffers werden die Daten im Binärformat übertragen. Protocol Buffers ist ein weit verbreitetes, robustes und performantes Protokoll. Die Daten werden in Textdateien durch eine eigene Syntax beschrieben und daraufhin mittels eines Compilers für die jeweilige Programmiersprache compiliert. (Alg. 1)

Algorithmus 1 Beispiel für eine uncompilierte Protocol Buffers Nachricht

```
message Person {
        required string name = 1;
        required int32 size = 2;
        optional string email = 3;
}
```

Insgesamt erfolgt die Netzwerkkommunikation mit IP, TCP, Protocol Buffers sowie HTTP. Die entwickelte und verwendete Netzwerkarchitektur wird in Kapitel 4.1 vorgestellt.

3. Stand der Technik

In Zeiten von Cloud basierten Systemen gibt es mehrere Ansätze der Dienst basierten Kommunikation zwischen Robotern. Ein Ansatz ist in „Robot control as a service — Towards cloud-based motion planning and control for industrial robots“ beschrieben. Hier geht es um die Entwicklung eines offenen und Service basiertem Framework zur flexiblen Bahnplanung und Steuerung von Industrierobotern. Im Rahmen der Arbeit wird ein Testsystem erfolgreich mit Hilfe des Frameworks implementiert. (Vick et al., 2015a) Die Abhandlung „Cloud robotics: Formation control of a multi robot system utilizing cloud Infrastruktur“ befasst sich mit dem Steuern von mehreren Robotern über das Internet. Das Konzept wurde erfolgreich umgesetzt. Der mögliche Einsatz von neuronalen Netzen wird in diesem System ebenfalls analysiert. (Turnbull and Samanta, 2013) Eine Umgebung von verteilten Robotern wird als Platform-as-a-Service in „A Cloud Computing Environment for Supporting Networked Robotics Applications“ vorgestellt. Diese Roboter teilen sich Rechnerresourcen und verbinden sich mit virtualisierten Diensten im Netzwerk. (Agostinho et al., 2011)

Die open-source Plattform Rapyuta befasst sich ebenfalls mit der Cloud basierten Robotersteuerung und kann gleichzeitig auf die Datenbank von RoboEarth zugreifen. Rapyuta ermöglicht es Robotern Aufgaben, wie bspw. komplizierte Berechnungen, an andere Netzwerkteilnehmer abzugeben. (Mohanarajah et al., 2015; van de Molengraft et al.)

In der Arbeit „Feasibility of connecting machinery and robots to industrial control Services in the cloud“ geht es um die Möglichkeiten Roboter ohne Netzwerkanbindung in Cloud-Systeme zu integrieren. Dazu wird eine einheitliche Netzwerkschnittstelle entwickelt, welche in drei unterschiedlichen Szenarien zur Anwendung kommt. (Horn and Krüger, 2016) Die Schnittstellen werden in „Control of robots and machine tools with an extended factory cloud“ weiter genutzt um eine private Cloud für Roboterkontroller sowie programmierbare Logikkontroller umzusetzen. Vor- sowie Nachteile des Ansatzes werden weiterhin diskutiert. (Vick et al., 2015b)

Vorteile, Herausforderungen und Probleme mit der Verwendung von Robotern in der Cloud werden in „Cloud robotics: architecture, challenges and Applications“, „Cloud robotics: Current trends and possible use as a service“ und „Robotic Services in Cloud Computing Paradigm“ erarbeitet und ausgewertet. (Hu et al., 2012; Lorencik and Sincak, 2013; Doriya et al., 2012) In „Cloud robotics: architecture, challenges and Applications“ wird die Sicherheit dieser Systeme genauer betrachtet. Die Arbeit „Cloud robotics: Current trends and possible use as a service“ analysiert die Vorteile der erhöhten Rechenleistung in der Cloud und der Möglichkeit eine zentrale Steuerungseinheit zu entwickeln. Die Möglichkeiten große Datenmengen in der Cloud zu verarbeiten um so Prozesse zu optimieren, wird in „Robotic Services in Cloud Computing Paradigm“ beurteilt.

Der Ansatz, die Cloud basierte Steuerung mit der Augmented Reality zu kombinieren, ist kein Bestandteil bereits bestehender Arbeiten. Es kommen lediglich geschlossene Systeme zur Anwendung. Zur Laufzeit können unbekannte Robotertypen nicht eingepflegt und in Betrieb genommen werden. Außerdem wird oftmals auf eine Vielzahl von Protokollen zu-

rückgegriffen, die alle miteinander kommunizieren. In dieser Ausarbeitung wird ein global verwendetes Protokoll verwendet um alle Dienste zu nutzen.

Die Idee zur Visualisierung und Steuerung mittels Augmented Reality existierte bereits zu Beginn dieser Arbeit. Innerhalb des Fachgebiets „Industrielle Automatisierungstechnik“ der Technischen Universität Berlin wurde vorher an einzelnen Komponenten gearbeitet. Diese waren ohne eine Netzwerkarchitektur direkt miteinander verbunden.

Zur Steuerung eines Industrieroboters kam die ur-bridge zum Einsatz. Diese Software ermöglicht es Steuerbefehle eines UR-5 Roboters mittels TCP/IP entgegenzunehmen und an den Roboter weiterzugeben. Sie wurde im Rahmen der Arbeit verwendet und an die neuen Bedingungen angepasst.

Um Roboter auf mobilen Endgeräten wie Tablets visualisieren zu können ist die Anwendung RoboViz entwickelt worden. Der Name der Software sowie das verwendete 3D Modell wurden beibehalten. Bestimmte Komponenten wie die Manipulation des Hologramms wurden weitestgehend umgeschrieben. Für die Kommunikation zwischen der Applikation RoboViz und der ur-bridge wurde ein neues Netzwerkkonzept entwickelt.

4. Konzept

Die Entwicklung des Konzepts ist in unterschiedliche Bereiche aufgeteilt. Zuerst wird ein Netzwerk erarbeitet, welches die Grundlage des Systems zur Steuerung von Industrierobotern bildet. Anschließend wird eine mögliche Form der Visualisierung entworfen. Zum Schluss wird auf die Steuerung der Roboter mittels augmented Reality eingegangen.

4.1. Netzwerk

Im Rahmen dieser Arbeit wurde eine Netzwerk- und Softwarearchitektur gesucht, die unabhängig von ihren Teilnehmern ist.

Des Weiteren gab es die Anforderung, dass das Netzwerk erweiterbar ist. Die Teilnehmer sollen dabei innerhalb des Netzwerkes nicht angepasst werden müssen. Als Beispiel soll ein neuer Roboter entwickelt und in die Umgebung und das Netzwerk integriert werden. Um mit dem Roboter interagieren zu können, sollen alle Endgeräte, wie HoloLens oder Tablets, nicht neu konfiguriert werden müssen. Der neue Roboter muss dazu in der Lage sein alle Geräte über seine Eigenschaften zu informieren. Weiterhin sendet er eine Anleitung wie er gesteuert wird.

Dies geschieht auf Basis von unterschiedlichen Diensten. Diese Dienste sind ebenfalls Mitglieder des Netzwerkes. Dienste können Teil eines Roboters oder komplett unabhängiger Komponenten sein. (Abb. 10)

Zusätzlich wurden einige Voraussetzungen gestellt um bereits bestehende Systeme in das neue Netzwerk integrieren zu können. Diese existierenden Systeme nutzen bspw. zur Kommunikation TCP und Google Protocol Buffers. Beide Technologien bilden die Grundlage für die neue Netzwerkarchitektur.

Im Folgenden wird fast ausschließlich über Dienste gesprochen. Dabei ist dieser Begriff generisch aufzufassen. Dienste können in Bezug auf ihre Aufgaben unterschiedlich sein. Die einzige Gemeinsamkeit aller Dienste ist, dass sie auf einem Teilnehmer im Netzwerk laufen und spezifische Aufgaben für einen oder mehrere Roboter im Netzwerk übernehmen. Vor der Konzepterarbeitung bringt folgende beispielhafte Anwendung von unterschiedlichen Diensten und Geräten das gewünschte Resultat näher. Innerhalb eines Netzwerkes befinden sich ein Roboter, eine HoloLens und ein Rechner mit dem Betriebssystem Linux. Der Roboter stellt zwei Dienste bereit. Ein Dienst ermöglicht die Achswinkel des Roboters auszulesen, der zweite diese zu ändern. Der Dienst, welcher inverse Kinematiken für Robotermodelle berechnet, wird auf dem Linux Computer ausgeführt. Die HoloLens stellt wiederum die Dienste „Anzeigen“ und „Steuern“ bereit.

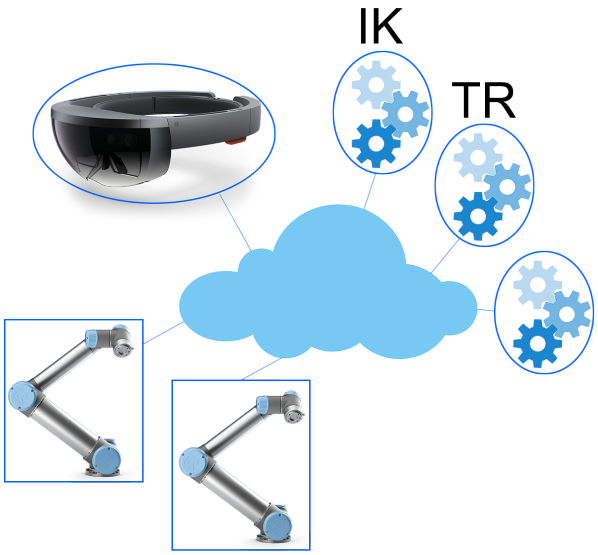

Abbildung 10: Teilnehmer und Dienste (Inverse Kinematik und Trajektorie) innerhalb eines Netzwerkes

Alle diese Dienste interagieren miteinander. Um den Roboter korrekt anzeigen zu können, benötigt die HoloLens die aktuelle Ausrichtung der Achsen des Roboters. Die HoloLens erfragt den entsprechenden Dienst auf dem Roboter. Anschließend kann sie den Roboter korrekt als Hologramm im Raum darstellen. Möchte der Benutzer der HoloLens die Position des Endeffektors verändern, kommt der Dienst „Steuern“ zur Anwendung. Mit Hilfe des Dienstes positioniert er einen virtuellen Platzhalter für den Endeffektor des Roboters an der gewünschten Position im Raum. Um den Endeffektor des Roboters an die Position des Platzhalters zu steuern, benötigt die HoloLens die entsprechenden Achswinkel des Roboters. Zur Berechnung der Achswinkel wird die inverse Kinematik aus Kapitel 2.1 benötigt. Es wird der Dienst „Inverse Kinematik“ vom Linux Rechner angefragt und die neuen Achswinkel werden berechnet. Die HoloLens erhält die neuen Achswinkel und kann das Hologramm des Roboters entsprechend anpassen. Das Hologramm ist damit in der neuen Position, der reelle Roboter hingegen nicht. Die HoloLens kontaktiert den Dienst „Achswinkel setzten“ auf dem Roboter und setzt die Winkel. Der reelle Roboter bewegt seinen Endeffektor auf die entsprechende Position.

Um diese Funktionalität im Netzwerk umzusetzen, wurde eine hybride Netzwerkarchitektur und -struktur entwickelt. Das Netzwerk ist einerseits ein Client-Server Netzwerk bei dem sich sämtliche Teilnehmer mit dem Server, genannt Broker, verbinden. Andererseits ist es ein Peer-to-Peer Netzwerk in dem die Teilnehmer direkt untereinander kommunizieren. (Abb. 11) Der Broker (Kreis) hat eine Verbindung zu allen Teilnehmern (Rechtecke). Diese bauen untereinander bei Bedarf untereinander eine Verbindung auf. Innerhalb des Client-Server Netzwerkes können Teilnehmer nach anderen Teilnehmern und Diensten suchen. Der Broker speichert Listen über alle verfügbaren Teilnehmer sowie ihre jeweiligen Dienste ab. Fragt ein Teilnehmer nach einem speziellem Dienst im Netzwerk, kann der Broker

direkt eine Auskunft erteilen, ob der Dienst verfügbar ist, oder nicht. Die Komplexität der Suche ist minimiert. Ist ein Dienst verfügbar, kontaktiert der Teilnehmer diesen direkt und ohne Umwege über den Broker. Der am Broker entstehende Traffic wird minimiert. Die im Kapitel 2.3 angesprochenen Probleme der beiden vorgestellten Netzwerkarchitekturen werden damit deutlich reduziert.

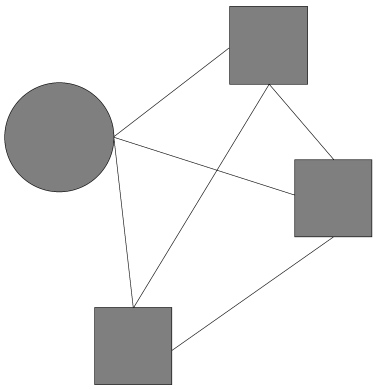

Abbildung 11: Hybrid-Netzwerk aus Client-Server und Peer-to-Peer

Sollte der Broker durch einen Ausfall nicht mehr verfügbar sein, kann ein Teilnehmer nicht nach Diensten suchen. Bereits laufende Dienste sind weiterhin verfügbar und nicht von dem Ausfall betroffen. Nimmt der Broker die Arbeit erneut auf, müssen sich sämtliche Teilnehmer erneut bei diesem melden. Der Broker aktualisiert schließlich seine internen Listen über vorhandene Geräte und Dienste.

Meldet sich ein neues Gerät im Netzwerk an, schickt es zunächst eine Benachrichtigung an den Broker. Dies geschieht im Format „Ich stelle für folgende(n) Roboter folgende Dienste zur Verfügung“. Im Anschluss kann das Gerät den Broker nach Informationen fragen. Diese werden im Format „Ich benötige eine Liste über alle Geräte folgenden Typs“ für die Anfrage einer Geräteliste oder „Ich benötige folgenden Dienst für folgenden Roboter(typ)“ für die Anfrage eines Dienstes geschickt.

4.2. Visualisierung

Die Visualisierung der Roboter mit Hilfe von mobilen Endgeräten wie der HoloLens oder einem Tablet wird mittels Hologrammen umgesetzt. Für jeden Roboter innerhalb des Industrieparks wird ein entsprechendes Hologramm erzeugt und im Raum dargestellt. Zu Beginn ist kein Roboter sichtbar. Vor sich im Raum sieht der Nutzer eine Liste mit verfügbaren Robotern. Wählt der Benutzer einen Roboter aus, schließt sich das Fenster der Liste. Der Roboter erscheint vor dem Nutzer im Raum und folgt seinem Blick. Die gewünschte Position des Roboters, bspw. ein Tisch im Raum, kann anvisiert werden. Wird der Roboter durch den Benutzer verankert, folgt er nicht weiterhin dem Blick, sondern verbleibt an entsprechender Stelle. Der Benutzer kann ab diesem Zeitpunkt den Roboter von allen Seiten

betrachten. An Stellen, an denen der Benutzer mit dem Roboter in Interaktion treten kann, werden Markierungen dargestellt. Ein Beispiel sind die einzelnen Gelenke des Roboters, die beim Betrachten aufleuchten und dem Benutzer anzeigen, dass sie veränderbar sind. (Abb. 12)

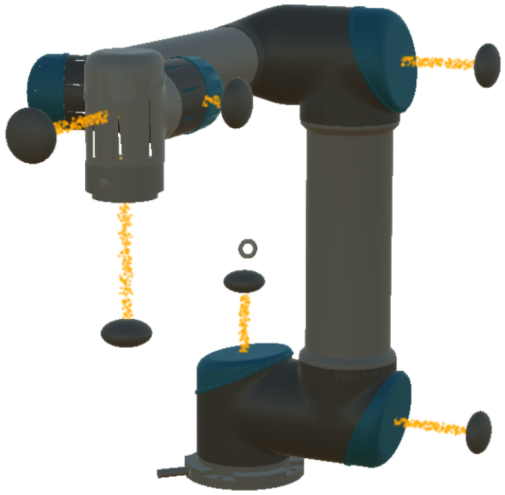

Abbildung 12: Hologramm eines UR-5 mit Steuerelementen für die Manipulation der Achsen

Abschließend gibt es die Möglichkeit Bewegungen zu simulieren. Das Hologramm führt, unabhängig vom reellen Roboter, eine geplante Bewegung aus. Nach einer Überprüfung der Simulation gibt es die Möglichkeit die Simulation an den Roboter zu übertragen. Er führt die Bewegung der Simulation entsprechend aus.

4.3. Steuerung

Die Roboter werden mit Hilfe der geladenen Dienste visualisiert. Zusätzlich ist es möglich Steuerungsbefehle an den Roboter zu senden. Im Rahmen dieser Arbeit werden drei verschiedene Steuerungsmöglichkeiten exemplarisch für die HoloLens implementiert. Diese stellen jeweils einen Dienst im Netzwerk dar.

Industrieroboter sollen häufig Gegenstände greifen und in einer bestimmten Position halten. Dazu muss es möglich sein diese Position mit Hilfe der Gelenkswinkel anzusteuern. Der Benutzer benötigt eine Eingabeoberfläche mit deren Hilfe er direkt jede einzelne Achse individuell einstellen kann. Um dies umzusetzen muss an jedem Gelenk ein hervorgehobenes Steuerelement sichtbar sein, welches der Benutzer auswählen und durch Handgesten manipulieren kann. Diese Gestenausführung hat eine Veränderung der Achswinkel zur Folge. Der Dienst zur direkten Steuerung der Winkel heißt „SET_AXIS_ANGLES“.

Der folgende Dienst „GET_INVERSE_KINEMATIC“ ist eine weitere Steuerungsoption. Er ermöglicht dem Benutzer, den Endeffektor direkt und nicht, wie bei „SET_AXIS_ANGLES“, indirekt zu steuern. Um den Endeffektor im Raum zu positionieren, müssen Punkte verschoben und rotiert werden können. Letztendlich schwebt ein klar

erkennbares Element im Raum, welches durch Gesten bewegt wird. (Abb. 13)

Abbildung 13: GUI zur Steuerung des Endeffektors - Translation (links) und Rotation (rechts)

Der Benutzer wählt eine Achse aus. Anschließend kann entlang dieser das Objekt versetzt werden. Um eine Rotation durchzuführen, wählt der Nutzer eine der Scheiben aus. Durch eine Drehung dieser rotiert das zentrale Objekt. Mit Hilfe der drei Achsen der Translation und Rotation können sämtliche Positionen im Raum erreicht werden. Die inverse Kinematik (s. Kapitel 2.1) bzw. der dazugehörige Dienst kann über das Netzwerk die Stellung des Roboters berechnen. Aufgrund dessen sind die Achswinkel bekannt und können gesetzt werden. Diese Funktion ist bereits durch den Dienst „SET_AXIS_ANGLES“ implementiert. Der Dienst „GET_INVERSE_KINEMATIC“ baut somit auf dem Dienst „SET_AXIS_ANGLES“ auf.

Die dritte Möglichkeit der Steuerung ist die Bahnplanung. Der Benutzer definiert im Umfeld des Roboters Punkte im Raum. Der Industrieroboter fährt diese, bspw. linear oder in Form eines Bezier-Splines, nacheinander ab. Der Dienst „INTERPOLATE_SERVICE“ errechnet die einzelnen Schritte der Bahnkurve. Nachfolgend wird für jeden Schritt die Achsstellung durch den Dienst „GET_INVERSE_KINEMATIC“ berechnet.
„SET_AXIS_ANGLE“ bewegt den Roboter in die Stellung. Durch berechnete Schritte zwischen den einzelnen Punkten resultiert eine gleichmäßige Bewegung des Roboters entlang der geplanten Strecke.

Diese drei Beispiele zeigen, dass der Benutzer den Roboter auf unterschiedliche Weise steuern kann. Die Umsetzung ist dienstweise aufeinander aufgebaut. Der Benutzer muss in der Lage sein mögliche Interaktionen zu erkennen. Jeder verfügbare Dienst, der über das Netzwerk geladen wird, muss über eine sinnvolle und einfache Bedienung verfügen und mit anderen Steuerungsdiensten kommunizieren können.

5. Umsetzung / Implementierung

Dieses Kapitel befasst sich mit dem Prozess vom Konzept zur fertigen Software. Hierfür mussten unter anderem sämtliche Rahmenbedingungen berücksichtigt und bereits vorhandene Bibliotheken analysiert werden. Damit einhergehend wurde festgelegt, dass sämtliche Kommunikation mittels Google Protocol Buffers erfolgt. Nur in wenigen Ausnahmen wurden andere Protokolle, wie HTTP, verwendet. Zur Kommunikation kommt auf der Transportschicht (ITU-T, 1994) TCP zum Einsatz.

Bevor die einzelnen Teilnehmer, wie Broker, Roboter oder HoloLens, programmiert werden, muss folgendes Problem der Netzwerkübertragung gelöst werden: Einzelne TCP/IP Pakete haben eine begrenzte Größe. Innerhalb eines Paketes können maximal 1.500 Bytes übertragen werden. Davon sind 20 Bytes für den IP Header reserviert und 20 Bytes für die Headerdaten von TCP. Für die Anwendung verbleiben maximal 1460 Bytes pro übertragenem Paket. Da in der Anwendung zum Teil größere Pakete, von mehreren hundert Interpolationsschritten innerhalb einer Nachricht, übertragen werden, muss die maximale Größe angehoben werden. Um innerhalb der Anwendung größere Datensätze als 1460 Bytes zu übertragen, wurde ein weiterer Bereich reserviert. 8 Bytes sind zusätzlich für die Angabe der Größe des Datensatzes reserviert (Abb. 14). Dieser Zusatz von 8 Byte stellt ein eigenes Protokoll dar und wurde aufgrund der geringen Komplexität *Microprotokol* genannt.

Abbildung 14: Größe der Paket-Headerdaten durch die Protokolle IP, TCP und Microprotokol

Es ist zu beachten, dass der Header von 8 Bytes immer zu Beginn eines Datensatzes geschickt wird und nicht zwangsweise innerhalb jedes TCP/IP Paketes steht. Als Beispiel wird die lineare Trajektorie des Endeffektors für einen Roboter berechnet. Dazu werden auf einer Gerade zwischen dem Start und dem Zielpunkt 1000 Schritte berechnet. Pro Schritt werden für den UR-5 Roboter sechs Achswinkel benötigt. Die Zahlen werden als *float* übertragen, nehmen somit jeweils 4 Bytes ein. Die Größe der Zahlendaten in Bytes z ist $z = 6 \cdot 4 \cdot 1000 = 24000$. Google Protocol Buffers benötigt für die Encodierung weitere $p = 4000$ Bytes. Für die Bahnkurve werden insgesamt $z + p = 28000$ Bytes übertragen. Für alle 1.000 Achsstellungen bzw. 28.000 Bytes werden $\lceil \frac{28000}{1460} \rceil = 20$ Pakete verschickt.

Um dieses Problem der Paketgröße zu lösen, wird für alle verwendeten Programmiersprachen zu Beginn jeweils die Klasse *Microprotokol* geschrieben. Die Klasse hat zwei Methoden. *read* ließt von einem TCP Socket und *write* schreibt beliebig große Datensätze in einen TCP Socket. (Alg. 2; Alg. 3) Insgesamt wurde die Klasse in drei Programmiersprachen, Python, C++ und C#, implementiert.

Algorithmus 2 Implementierung der read-Methode in Python (Microprotokol)

```
buf = []
data = reader(self.MAX_HEADER_BYTE_SIZE) # read header from stream, also
    blocks until data is received

# rCount is the total length of all data read from tcp stream
rCount = len(data)

# size is the total size of the data set; position is begining of the data
    set
(size, position) = decoder._DecodeVarint(data, 0)

# add first data package to buffer
buf.append(data)

# read packages as long rCount (current size) is smaller then total size +
    1
while rCount < size + 1:
    data = reader(size + 1 - rCount)
    rCount += len(data)
    buf.append(data)

# combine all packages
binary_message = b''.join(buf)

# received message as byte stream without length prefix
return binary_message[position:(position + size)]
```

Algorithmus 3 Implementierung der write-Methode in Python (Microprotokol)

```
# calculate message length
bytes = encoder._VarintBytes(message.ByteSize())

# write message length and message itself to stream writer
writer(bytes + message.SerializeToString())
```

Mit Hilfe der Klassen ist es möglich Protocol Buffers Nachrichten in jeder der oben genannten Programmiersprachen über TCP/IP zu schicken.

Somit konnten im Anschluss die Teilnehmer des verteilten Netzen umgesetzt werden. Im ersten Schritt wurden der Broker entwickelt. Ein Kriterium für diesen war die Portabilität, die Einsatzmöglichkeit auf unterschiedlichen Betriebssystemen. Als Plattform-unabhängige und geeignete Programmiersprache wurde Python in der Version 3 festgelegt.

5.1. Broker

Nachdem die Grundlagen für das Versenden von Nachrichten innerhalb des verteilten Netzwerkes durch TCP/IP und dem entwickelten *Microprotokol* gegeben sind, kann der Broker

unabhängig von den anderen Teilnehmern entwickelt werden. Der Broker ist vollständig in Python programmiert, wodurch er auf sämtlichen von Python unterstützten Plattformen ausgeführt werden kann. Der Broker wartet in einem Thread auf eingehende Verbindungen. Verbindet sich ein Client mit dem Broker, wird ein neuer Thread erstellt. Aufgrund dessen kann der Broker mit mehreren Clients asynchron kommunizieren. Innerhalb des erstellten Threads wartet der Broker mit Hilfe der Microprotokol-Klasse (Alg. 2) auf eingehende Protocol Buffers Nachrichten.

Im ersten Schritt wurde die Routine zum Registrieren von Geräten auf dem Broker implementiert. Jeder neue Teilnehmer im Netzwerk kann sich beim Broker melden und ihm eine *greeting* Nachricht schicken. Inhalt der Nachricht *greeting* sind zwei Felder. Das Gerät selbst sowie eine Liste von Diensten, die das Gerät anbietet:

```
message greeting {
    device device = 1;
    repeated service services = 2;
}
```

Die Angabe eines Feldes erfolgt durch die Typisierung gefolgt von einem Feldnamen. Die Zahlen sind Tag-Nummern und werden von Googles Protocol Buffers intern genutzt. Einziges Kriterium für diese ist die Einzigartigkeit innerhalb einer Nachricht. Das erste Feld der Nachricht *greeting* hat den Namen „device“ und ist vom Typ *device*. Das zweite Feld ist vom Typ *service* und heißt „services“. Durch den Zusatz *repeated* wird aus dem Feld eine Liste.

Eine *device* Nachricht enthält ebenfalls zwei Felder und sieht wie folgt aus:

```
message device {
    repeated string identifiers = 1;
    string ip = 2;
}
```

Der Inhalt des Feldes „ip“ ist die IP-Adresse des Gerätes selbst. Das Feld „identifiers“ ist eine Liste von Zeichenketten bzw. Strings. Die einzelnen Werte der Liste sind Bezeichner um den Roboter zu identifizieren. Die Reihenfolge ist relevant und wird von genau nach ungenau sortiert. Ein Beispiel für die identifiers eines UR-5 Roboters mit der Seriennummer „S4X8D986“ ist: `["UR-5-S4X8D986", "UR-5", "industrial-robot", "robot"]`. Ein Rechner, der den Dienst zum Berechnen einer inversen Kinematik für den UR-5 anbietet, hat als identifiers bspw. die Liste: `["inverse-kinematic", "UR-5", "service"]`.

Somit handelt es sich bei letzterem um einen Dienst für den UR-5 und nicht um einen Roboter.

Jedes Gerät, welches Dienste anbietet, schickt diese Dienste als Liste innerhalb der *greeting*-Nachricht. Eine Dienstnachricht *service* beinhaltet zwei Felder: den Namen des Diens-

tes und die Portnummer unter der der Dienst läuft. Durch die IP-Adresse des Gerätes in der *device*-Nachricht ist jeder Dienst eindeutig lokalisierbar.

```
message service {
    enum names {
        GET_AXIS_ANGLES = 0;
        SET_AXIS_ANGLES = 1;
        GET_FORWARD_KINEMATIC = 2;
        GET_INVERSE_KINEMATIC = 3;
        ...
    }

    names name = 1;
    int32 port = 2;
}
```

Das Feld „name“ vom Typ *names* hat einen Wert aus der Liste *enum names*.

Möchte sich ein Teilnehmer des Netzwerkes beim Broker registrieren, schickt er eine vollständige Nachricht mit IP-Adresse, Identifiers und einer Liste seiner Dienste an den Broker.

Es ist nicht festgelegt, dass das Gerät nach dem Verbinden mit dem Broker eine Begrüßung schickt. Einige Geräte stellen keine Dienste bereit und nutzten andere Dienste im Netzwerk. Diese werden keine *greeting* Nachricht schicken. Es gibt Rechner, die ggf. zur Laufzeit lokale Dienste starten oder beenden. Diese schicken eine neue Begrüßung an den Broker um diesen über die Änderung zu informieren.

Welcher Nachrichtentyp vom Rechner bzw. Client geschickt wird, ist nicht definiert. Daher wird eine logische Weiche benötigt. Der Client könnte bspw. eine *greeting*-Nachricht oder alternativ die Suchanfrage für einen Dienst schicken. Der Broker muss vorausschauen können, welche Nachricht als nächstes eintrifft. Dies kann durch einen endlichen Automaten umgesetzt werden. Der Automat entscheidet je nach seinem aktuellen Zustand, welcher Nachrichtentyp als nächstes eintrifft. (Wuttke/Henke, 2003) Für jeden Client, der mit dem Broker verbunden ist, müsste folglich ein Automat im Hintergrund laufen.

Es gibt eine weitere Möglichkeit dieses Problem mit Hilfe von Protocol Buffers zu lösen. Der Broker erwartet nur einen Nachrichtentyp. Teil der Sprache sind *oneof*-Felder. Innerhalb dieser können Unterfelder definiert werden. Eine Nachricht enthält genau einen der Unterfelder als Wert. Über eine API-Methode lässt sich ermitteln, welches Unterfeld einen Wert hat. Im Fall des Brokers erwartet dieser den Nachrichtentyp *to_ broker*. Diese Nachricht ist wie folgt definiert:

```
message to_broker {
    oneof request_or_greeting {
        request request = 1;
        greeting greeting = 2;
```

```
    }
}
```

Erhält der Broker von einem Client eine Nachricht, ist in dieser entweder ein *request* oder eine *greeting* enthalten. Je nachdem welches Feld gesetzt ist, kann der Broker entscheiden wie die Nachricht beantwortet wird.

In der Implementierung wurde ein Handler Interface umgesetzt. Für jeden Typ in dem *oneof*-Feld request_or_greeting gibt es eine Klasse, die das Interface implementiert. Es wird beim Eintreffen einer *to_ broker* Nachricht der entsprechende Handler im Broker geladen. (Alg. 4)

Algorithmus 4 Feld im oneof-Block - Entscheidung welcher Handler geladen wird

```
if _to_broker.WhichOneof("request_or_greeting") == "request":
    print("request␣detected")
    return self.request_handler
else _to_broker.WhichOneof("request_or_greeting") == "greeting":
    print("greeting␣detected")
    return self.greetings_handler
```

Der Handler führt die im Anschluss folgende Aktion aus. Empfängt der Broker ein *greeting*, wird der Teilnehmer, bzw. das Gerät in einer Liste gespeichert. Die Liste der Dienste wird ebenfalls mit einem Eintrag versehen. Dabei verweist innerhalb der Liste jeder Dienst auf das entsprechende Gerät, welches den Dienst anbietet. Empfängt der Broker eine *request* Nachricht, wird ermittelt welche Information der Teilnehmer angefragt hat.

Teilnehmer können den Broker nach unterschiedlichen Informationen fragen. Die Nachricht ist wie die *to_ broker* Nachricht aufgebaut und enthält ein *oneof* Feld. Die vollständige Nachricht *request* beinhaltet folgende Felder:

```
message request {
    bool use_or_for_query = 1;
    device receiver = 2;

    oneof requested_data {
        bool bag = 3;
        service service = 4;
    }
}
```

Erneut kann nur ein Feld innerhalb des Blocks „request_data“ einen Wert haben. Es wird entweder eine *bag* oder ein *service* angefragt. Ist das Feld „bag“ auf den Wert *true* gesetzt, durchsucht der Broker seine Liste an Geräten. Welche Geräte zurückgegeben werden, hängt wiederum vom Feld „receiver“ ab. Dieses Gerät muss kein bestimmtes Gerät im

Netz sein. Es können ein oder mehrere Identifier angegeben werden um in der Liste eine Auswahl an Geräten zu treffen. Wird nach dem Gerät „UR-5“ gesucht, werden alle Geräte, die in ihren Identifiers „UR-5“ enthalten, zurückgegeben. Somit sind in der Liste alle Roboter vom angegebenen Typ und Teilnehmer, die Dienste für diesen Robotertyp anbieten, enthalten. Zur Präzisierung der Suche, können weitere Begriffe, wie „robot“, angegeben werden. Umgekehrt ist es möglich durch das Setzten des Feldes use_or_for_query eine Oder-Suche, anstatt einer Und-Suche, durchzuführen. In diesem Fall müssen die Geräte mindestens einen der angegebenen Identifiers haben.

Erfragt der Client einen Service, wird der zweite Wert von „request_data“ gesetzt. Die *service* Nachricht ist identisch mit der obigen in der *greeting* Nachricht. Lediglich der Wert des Feldes „port“ wird nicht gesetzt. Als Antwort erhält der Client vom Broker eine Nachricht vom Typ *response*. Sie ist erneut mit mehreren Feldern ausgestattet, die je nach Fall einen Wert haben.

```
message response {
    oneof response_data {
        string error = 1;
        device_bag devices = 2;
        service_adress location = 3;
        string message = 4;
    }
}
```

Sind Fehler aufgetreten, weil bspw. ein angefragter Service nicht zur Verfügung steht, ist das Feld error mit einer entsprechenden Nachricht gesetzt. Andernfalls ist jeweils das Feld gesetzt, nach dem gefragt wurde.

Zur Verdeutlichung folgt ein beispielhafter Ablauf mit drei Teilnehmern und dem Broker. Die drei Teilnehmer sind eine HoloLens, ein Roboter und ein Rechner mit Diensten.

Schritt 1 Der Rechner verbindet sich mit dem Broker und grüßt ihn. Er schickt eine *to_broker* Nachricht mit einer *greeting* als Inhalt. Die Begrüßung beinhaltet seine Bezeichner „UR-5“ sowie „service“ und seine IP Adresse. Als Dienste schickt der Rechner eine Liste mit einem Eintrag, dem GET_INVERSE_KINEMATIC und der entsprechenden Portnummer.

Schritt 2 Die HoloLens verbindet sich mit dem Broker und fragt nach einer Liste mit Robotern. Sie schickt eine Nachricht *to_broker*. Diese enthält einen *request* als Inhalt. Für das Feld receiver wird eine *device* Nachricht mit dem Bezeichner „robot“ geschickt. Als request_data wird das Feld bag auf *true* gesetzt.

Schritt 3 Der Broker antwortet der HoloLens mit einer *response*. Es traten keine Fehler auf. Somit wird als „response_data“ das Feld devices gesetzt. Die *device_bag* enthält eine leere Liste, da noch kein Roboter den Broker gegrüßt hat.

Schritt 4 Der Roboter meldet sich beim Broker und begrüßt ihn. Der Inhalt der *to_ broker* Nachricht ist wieder ein *greeting*, mit den Bezeichnern „UR-5-Seriennummer“, „UR-5“, „robot“ und seiner IP Adresse. Die Dienste sind eine Liste mit zwei Eintragen, GET_AXIS_ANGLES und SET_AXIS_ANGLES, und den jeweiligen Portnummern.

Schritt 5 Alle 5 Sekunden erfragt die HoloLens, wie in Schritt 2 beschrieben, erneut eine Liste.

Schritt 6 Der Broker antwortet wie in Schritt 3. Die *device_ bag* beinhaltet nun einen Eintrag, den Roboter aus Schritt 4.

Schritt 7 Die HoloLens lädt den Roboter in die Szene und erfragt beim Broker, ob Dienste für den Roboter vorhanden sind. Pro Dienst wird eine *request* Nachricht, die den jeweiligen Namen des Dienstes im Feld service beinhaltet, geschickt.

Schritt 8 Der Broker antwortet für jeden Dienst mit einer *service_ adress*. Für die Dienste GET_AXIS_ANGLE und SET_AXIS_ANGLE liefert er die IP des Roboters mit den jeweiligen Portnummern. Für den Dienst GET_INVERSE_KINEMATIC wird die *service_ adress* zum Rechner aus Schritt 1 zurückgegeben.

Schritt 9 Die HoloLens kennt die verfügbaren Dienste für den Roboter, kann eine GUI laden und sich mit Diensten verbinden.

Da die HoloLens beim Platzieren Dienste für einen Roboter erfragt (Schritt 7), ist es nötig, dass sich Mehrwertdienste, wie die zur Berechnung der inversen Kinematik, vorher beim Broker anmelden. Nur dann erhält die HoloLens vom Broker einen entsprechenden Eintrag. Diese Voraussetzung verringert die Komplexität. Weiterhin bringt sie kaum bis keine Nachteile mit sich, da davon auszugehen ist, dass sich Mehrwertdienste einmalig beim Starten des Programm mit dem Broker verbinden. Lediglich Roboter werden in manchen Situationen ab- und wieder angeschaltet.

Nach der Beschreibung der Protocol Buffers Nachrichten, die mit dem Broker ausgetauscht werden, wird die Verarbeitung dieser Nachrichten implementiert. Innerhalb des Broker gibt es zwei Verzeichnisse, eines für Geräte bzw. Teilnehmer und eines für Dienste. Wird eine Nachricht aus Schritt 1 empfangen, verarbeitet der „greetings_handler“ (Alg. 4) die Nachricht. Dabei werden sowohl die Dienste des grüßenden Gerätes, als auch das Gerät selber registriert. (Alg. 5)

Algorithmus 5 Hinzufügen eines neuen Geräts in die Bestandsliste und Registrierung seiner Dienste

```
def registerDevice(self, client, greeting):
    self.known_devices.append(client)
    self.device_resolver.addDevice(greeting, client)
    self.service_resolver.addServices(greeting, client)
```

Jeder Dienst innerhalb der *greeting* wird im Dienstverzeichnis service_resolver mit der Methode addServices hinzugefügt. Innerhalb der Methode wird ein Kanal für jeden Dienst erzeugt, welcher sämtliche Informationen über den Dienst enthält.

Erhält der Broker eine *request* Nachricht, wird der „request_handler“ für die Verarbeitung der empfangenen Nachricht verwendet. Der Handler entscheidet, wie die Anfrage beantwortet wird. Es können zwei verschiedene Daten angefragt werden. Entweder wird eine *device_bag*, eine Liste von Geräten eines bestimmten Typs, oder ein *service* für einen speziellen Roboter angefordert. Wird der Broker nach einer Geräteliste gefragt, sucht er innerhalb des Geräteverzeichnisses device_resolver nach entsprechenden Geräten.

```
requested_identifiers = _to_broker.request.receiver.identifiers

devices = self.broker.device_resolver.findDevices(requested_identifiers)
```

Die gefundenen Geräte werden anschließend mit Hilfe einer *response* Nachricht an das anfragende Gerät zurückgesendet. Im Falle einer Anfrage eines Dienstes an den Broker wird nicht der device_resolver, sondern der service_resolver angefragt.

```
service_channel = self.broker.service_resolver.resolveFirstService(
    _to_broker.request.receiver,
    _to_broker.request.service,
    _to_broker.request.use_or_for_query
)
```

Für jeden auf dem Broker registrierten Dienst wird ein Kanal erzeugt. Die Klasse für den Kanal heißt *ServiceChannel* und enthält unter anderem die Methode isResponsableFor. Diese überprüft, ob der Kanal für einen Dienst zuständig ist.

Wird ein Dienst, wie in Schritt 7, angefragt, überprüft der Broker jeden *ServiceChannel*, ob er für den angefragten Dienst zuständig ist. Dafür werden drei Parameter benötigt: der angefragte Name des Dienstes, das Gerät, für welches der Dienst zuständig sein muss, und in welchem Modus gesucht werden soll. (s. use_or_for_query in *request*) Ist der Kanal für einen anderen Dienst zuständig, kann direkt mittels *False* verneint werden. Sind der Name vom angefragten Dienst und vom vorliegenden Kanal identisch, wird die Zuständigkeit überprüft. Ein Dienst, wie GET_INVERSE_KINEMATIC, ist nicht für jeden Roboter

zuständig. Somit muss überprüft werden, ob die Identifiers des Kanals mit denen des angefragten Roboters übereinstimmen. Hierbei ist der Wert der Flag use_or_for_query relevant. Ist der Wert *False*, müssen die Listen identisch sein. (Alg. 6)

Algorithmus 6 Algorithmus zum Überprüfen der Zuständigkeit eines Dienstes für ein Gerät

```
for identifier in device.identifiers:
    if identifier not in self.device.identifiers:
        return False

return True
```

Nur wenn beide Listen dieselben Identifiers beinhalten, liefert die Methode *True*. Für denn Fall, dass use_or_for_query den Wert *True* hat, muss ein Identifier des Gerätes in der Liste der Identifiers des Kanals enthalten sein. Dazu muss die Methode isResponsableFor bei der ersten Übereinstimmung eines Identifiers *True* zurückgeben. Der Kanal enthält sowohl die IP-Adresse, als auch den Port des Dienstes. Die Werte werden anschließend als *service_adress* an das anfragende Gerät zurückgeliefert.

Die HoloLens erfragt eine Liste aller Teilnehmer. Im Anschluß dazu lädt die HoloLens alle verfügbaren Dienste. Sie erfragt beim Broker die Adresse und verbindet sich mit dem jeweiligen Dienst. (Abb. 15)

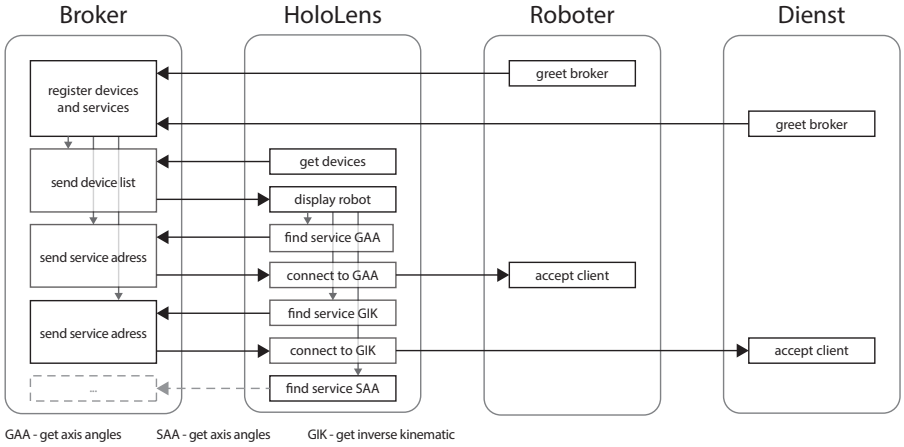

Abbildung 15: Beispielhafte Kommunikation zwischen Broker, HoloLens, Roboter und Dienst

Durch die Implementierung der Funktionen *greeting* und *requesting,* ist der Broker in der Lage Geräte zu listen und Dienste zwischen den Teilnehmern zu vermitteln.

5.2. HoloLens

Die HoloLens wird mit der Entwicklungsumgebung Unity programmiert. („Unity“, 2018) Unity ist eine Plattform, mit der Spiele und Anwendungen für Augmented Reality, erweiterte Realität, entwickelt werden. Für die Programmierung wird von Microsoft eine Paket für Unity bereitgestellt, in dem bereits Funktionen implementiert sind. („MixedRealityToolkit-Unity uses code from the base MixedRealityToolkit [...]“, 2018) In Unity werden einzelne Bereiche der Anwendung in Szenen unterteilt. Eine Szene, ähnlich wie im Theater, ist eine Zusammenstellung von Objekten (Requisiten) und dem Spieler (Schauspieler). Der Spieler kann mit den Objekten innerhalb der Szene interagieren. Objekte können zur Laufzeit der Szene hinzugefügt und wieder entfernt werden. Im Bereich der Augmented Reality unterscheidet sich die Szene maßgeblich von Spielen. Während in Spielen die Umgebung hinzugefügt werden muss, ist diese bei der HoloLens von Beginn an sichtbar. Die Objekte in der Szene sind Hologramme, die in die Umgebung eingeblendet werden.

Wie bereits erwähnt, handelt es sich bei Hologrammen um Objekte. Unity bezeichnet diese als GameObject. Ein GameObject wird mit Komponenten versehen. Eine wichtige Komponente ist das Mesh. Dieses ist die Form des Objektes. Ohne Mesh ist ein GameObject nicht sichtbar. Solche unsichtbaren Objekte werden auch leere, bzw. empty GameObjects genannt. Weitere Komponenten sind Kollisionsboxen für die Physik, Texturen für die Oberfläche und Skripte. Ein Skript ist eine C#-Klasse, welche die Unity-Klasse MonoBehaviour erweitert. Wird ein solches Skript an ein GameObject als Komponente gebunden, kann mit Hilfe des Skriptes das Objekt programmiert werden. Dabei gibt es vordefinierte Methoden innerhalb der Klasse, welche von Unity selbst ausgeführt werden. Beim Platzieren des Objektes in der Szene wird die Methode *start* aufgerufen. Soll sich ein in der Szene platzierter Roboter mit einem Dienst über das Netzwerk verbinden, geschieht dies in dieser Methode. Eine weitere wichtige Methode ist *update*. Diese wird innerhalb jedes einzelnen Frames von Unity aufgerufen. Schickt ein Roboter in regelmäßigen Abständen seine Achswinkel an die HoloLens, muss das Hologramm in der Szene entsprechend verändert werden. Innerhalb der *update* Methode können die regelmäßigen Änderungen stattfinden.

5.2.1. Sensorik

Um die Kommunikation mit der Sensorik zu vereinfachen, wurden im ersten Schritt Hilfsklassen programmiert. Die HoloLens ist in der Lage Gesten, wie ein Fingertippen oder eine Handbewegung, zu erkennen. Zur vereinfachten Handhabung dieser Gesten wurden Manager, welche jeweils eine Geste managen, implementiert. Der einfachste Manager ist der *TapEventManager*. Er bietet eine Schnittstelle zur Kommunikation zwischen der HoloLens Sensorik und dem eigentlichen Programm. Erkennt die HoloLens ein Tap (tippen) mit dem Finger, wird dieses Event vom Manager registriert und an Objekte in der Szene geleitet. Aus Sicht des Managers gibt es zwei unterschiedliche Formen von Objekten, globale und lokale. Globale Objekte erhalten Kenntnisse über sämtliche vom Benutzer ausgeführte Taps. Lokale Objekte erfahren hingegen nur von solchen, die direkt auf Sie gezielt wurden. Dazu

wird vom *TapEventManager* ein Raycast, ein Strahl von der Position der Kamera in Blickrichtung, durchgeführt. Das erste getroffene Objekt ist das lokale Objekt. Dieses empfängt zusammen mit den globalen Objekten die Benachrichtigung über einen Tap. (Alg. 7)

Nicht jedes Objekt soll über einen lokalen Tap informiert werden. Um Objekte ansprechen zu können, müssen diese bestimmte Schnittstellen, genannt *Interfaces*, implementieren. Derzeit existieren Interfaces für die Gesten Tippen *(ITapable)*, Navigieren *(INavigatable)* und Manipulieren *(IMovable)*. Um ein Interface zu implementieren, muss eine *MonoBehaviour*-Komponente die entsprechende Schnittstelle erweitern. Somit wird das lokale Objekt lediglich über den Tap informiert, wenn es über die Komponente ITapable verfügt.

Algorithmus 7 Benachrichtigung globaler und lokaler Objekte mittels OnTapped()

```
private void NavigationRecognizer_Tapped(TappedEventArgs e)
{
    // trigger all global tapables
    foreach (var globalTapable in _globalTapables)
    {
        globalTapable.OnTapped();
    }

    // perform raycast
    RaycastHit hit;

    if (Physics.Raycast(e.headPose.position, e.headPose.forward, out hit))
    {
        // store tapped object
        CurrentTappedObject = hit.transform.gameObject;

        // fetch ITapable component from GameObject
        var iTapable = CurrentTappedObject.GetComponent<ITapable>();

        // trigger method if iTabable is not null
        iTapable?.OnTapped();

        // ...
    }
}
```

Für die Interfaces *INavigatable* und *IMovable* wurden ebenfalls entsprechende Manager, *NavigateEventManager* und *ManipulateEventManager*, implementiert. Diese erkennen die jeweiligen Gesten, verarbeiten sie und rufen anschließend die dazugehörigen Methoden der Komponenten auf.

Um die Umgebung realistisch in der Anwendung abbilden zu können, erfolgt ein räumliches Kartografieren, genannt Spatial Mapping. Die HoloLens ermittelt durch den Tiefensensor (s. Kapitel 2.2) den Abstand zu Elementen im Raum und berechnet daraus ein Mesh, welches über den Raum gelegt wird. Wird ein Hologramm im Raum platziert, platziert die HoloLens es auf dem Mesh. Der Benutzer erhält den Eindruck, dass das Hologramm auf

einem Gegenstand, bspw. Tisch, steht.

Die Umgebungserkennung wird von vier seitlich angebrachten Kameras und einem Tiefensensor vorgenommen. Das HoloToolKit beinhaltet das GameObject *SpatialMapping*. Ist das Objekt in der Szene enthalten, wird ein Mesh zur Laufzeit berechnet und über die Raumoberfläche gelegt. (Abb. 16)

Abbildung 16: Visualisierung der Oberflächenerkennung durch ein Mesh

In dieser Hardwareversion der HoloLens erfolgt die Erkennung der Oberflächen unpräzise. Der linke Tischrand ist bspw. um 3cm nach links verschoben. Die dünnen Tischbeine werden kaum erkannt. Glatte Oberflächen, wie die Tischplatte oder Wände, werden hingegen äußerst genau wahrgenommen. Aufgrund dessen kann das Mesh zur Positionierung der Hologramme im Raum genutzt werden.

5.2.2. Netzwerkkommunikation

Zum Verschicken der Protocol Buffers Nachrichten mit Hilfe des TCP/IP wird die *Mircoprotokol* Klasse in C# verwendet. (s.Kapitel 5) Beim Starten der Unity Applikation RoboViz auf der HoloLens wird in der Startszene ein *empty GameObject* namens BrokerCommunicator erzeugt. Dieses unsichtbare Objekt enthält ein Skript, welches die Verbindung mit dem Broker steuert. Da *GameObjects* bzw. deren Komponenten miteinander kommunizieren können, greifen alle GameObjects in der Szene auf das Objekt BrokerCommunicator mit dem Verbindungsskript zu und kontaktieren darüber den Broker. Bspw. kann die Komponente zur Synchronisation der Roboterliste auf der HoloLens über diese Abstraktionsebene leicht Anfragen an den Broker senden. Die Methode RequestDevices abstrahiert das Senden einer *request* Nachricht mit einer *device_bag* und das Empfangen einer *response*. (s. Kapitel 5.1; Alg. 8)

Algorithmus 8 Synchronisation der Gerätelisten zwischen Broker und HoloLens

```
private async void LoadDevicesPeriodicaly()
{
    while (Client.IsConnected)
    {
        var devices = Broker.RequestDevices("robot");

        SyncDeviceLists(_devices, devices.Devices.ToList());

        await Task.Delay(TimeSpan.FromSeconds(UpdateRate));
    }
}
```

Während der Client mit dem Broker verbunden ist, wird die lokale Geräteliste mit der Liste des Brokers in vorgegebenen Abständen synchronisiert.

Diese Synchronisation läuft in einer Endlosschleife. Dadurch entsteht ein entscheidendes Problem. Die Methode LoadDevicesPeriodicaly kann nicht im Hauptthread des Programmes gestartet werden, da die Schleife nie zur Laufzeit verlassen wird. Dies führt wiederum zum Einfrieren der gesamten Applikation. Die Methode LoadDevicesPeriodicaly muss auf einem anderen Thread des Programmes laufen. Um dies zu erreichen wird die Methode durch einen *Task* parallelisiert. 9

Algorithmus 9 Ausführung der Methode LoadDevicesPeriodicaly innerhalb eines neues Tasks

```
new Task(LoadDevicesPeriodicaly).Start();
```

Durch dieses Parallelisierung wird der Hauptthread der Applikation während der Anfragen an dem Broker nicht blockiert. Es werden weiterhin Gesten den Benutzers verarbeitet und die Szene immer wieder neu aktualisiert und gezeichnet. Ist die Anfrage an den Broker erfolgreich beantwortet worden, müssen die empfangenen Daten in der Anwendung verarbeitet werden. Dazu muss die neue Geräteliste vom Broker mit der bestehenden Geräteliste auf der HoloLens verglichen werden. Neue Geräte im Netzwerk werden zur Szene hinzugefügt und entfernte Geräte aus der Szene gelöscht.

Ein durch die Methode SyncDeviceList ermitteltes Gerät wird in der Szene dem Inventar hinzugefügt. Zusätzlich erscheint ein kleiner Hinweis, dass ein neues Gerät im Inventar zu finden ist. (Abb. 17)

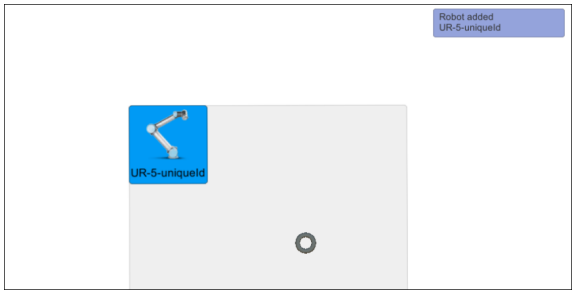

Abbildung 17: Ansicht des Inventars und einer Benachrichtigung in der HoloLens

Um ein Objekt der Szene hinzuzufügen, muss dieses instanziiert werden. Dies geschickt mittels *GameObject.Instanciate*. Erfolgt dieser Aufruf innerhalb der Methode *SyncDeviceList,* um bspw. ein Item im Inventar anzuzeigen oder den Hinweis in der Szene zu platzieren, wird dies mit der Fehlermeldung „Instanciate can only be called from the main thread." quittiert. Problematisch ist, dass der Aufruf von *GameObject.Instanciate* im Nebenthread und nicht im Hauptthread der Anwendung erfolgt. Unity ist eine Non-Thread-Safe Plattform. Sämtliche Methoden von Unity müssen im Hauptthread ausgeführt werden.

Das Anzeigen im Inventar inklusive Hinweismeldung muss im Hauptthread erfolgen. Um dieses Problem der Thread-Kommunikation zu lösen, wird das interne Parallelisierungssystem, *Coroutines,* von Unity genutzt. Die *Coroutines* ermöglichen es Aufgaben in den Hauptthread einzufügen. Unity verwendet dafür einen Slicing-Ansatz. Die Aufgaben werden im Hauptthread vorrangig und nicht parallel ausgeführt. Dennoch eignet sich diese Form der Umsetzung für kleine Aufgaben, wie das Anzeigen einer Nachricht am Bildschirmrand, da diese wenig Rechenzeit benötigen.

Um die Kommunikation zwischen Hauptthread und Nebenthread zu vereinfachen, wurde eine weitere Hilfsklasse implementiert. Innerhalb des Hauptthreads läuft eine FifoQueue, die *Coroutines* sammelt und bei Bedarf ausführt. (Alg. 10)

Algorithmus 10 First-In-First-Out Queue des Dispatchers zur Thread-Synchronisation

```
public class Dispatcher : Singleton<Dispatcher> {
    private static readonly Queue<Action> ExecutionQueue = new Queue<Action
        >();

    // ...
}
```

In jedem Frame werden vom Dispatcher alle in der ExecutionQueue enthaltenen Aktionen im Hauptthread eingebunden. Um eine Aktion innerhalb jedes einzelnen Frames auszuführen, stellt Unity die bereits beschriebene Methode *Update* bereit. Während der Verarbeitung der Queue darf kein weiterer Thread diese verändern. Aus diesem Grund wird die Queue für alle anderen Threads mittels *lock* gesperrt. (Alg. 11)

Algorithmus 11 Frameweise Abarbeitung der Queue mittels Update-Methode

```
public class Dispatcher : Singleton<Dispatcher> {
    // ...

    public void Update()
    {
        lock (ExecutionQueue)
        {
            while (ExecutionQueue.Count > 0)
            {
                ExecutionQueue.Dequeue().Invoke();
            }
        }
    }

    // ...
}
```

Möchte ein Nebenthread eine Aufgabe in die Warteschlange einfügen, genügt ein Aufruf der Methode Enqueue auf dem Dispatcher. Diese sperrt die Queue für den Hauptthread, sodass dieser nicht vorzeitig mit der Abarbeitung beginnen kann. Weiterhin fügt er eine anonyme Aktion inklusive *Coroutine* der Queue hinzu. (Alg. 12) Die Verwendung einer *Coroutine* innerhalb des Hauptthreads sorgt dafür, dass Unity die Aktion nicht sofort, sondern zum bestmöglichen Zeitpunkt ausführt. Idealerweise ist dies, wenn sich der Hauptthread nicht mit rechenintensiven Aufgaben, wie Spatial Mapping, beschäftigt.

Algorithmus 12 Hinzufügen einer neuen Aufgabe zum Dispatcher mittels Enqueue-Methode

```
public class Dispatcher : Singleton<Dispatcher> {
    // ...

    public void Enqueue(IEnumerator action)
    {
        lock (ExecutionQueue)
        {
            ExecutionQueue.Enqueue(() => {
                StartCoroutine(action);
            });
        }
    }

    // ...
}
```

Innerhalb der Methode SyncDeviceList erfolgt für jeden neuen Roboter ein Aufruf der Methode Enqueue des Dispatchers. (Alg. 13) Es wird eine anonyme Funktion, genannt

Closure, verwendet um die Aktion in einer *Coroutine* starten zu können. Der Inhalt der Closure, EventBus.Fire, wird zu einem späterem Zeitpunkt ausgeführt.

Algorithmus 13 Auslösung des Events auf dem Main-Thread

```
Dispatcher.Instance.Enqueue(() => { EventBus.Fire(new RobotAdded(device));
    });
```

Durch den *EventBus* werden die nötigen Aufgaben, wie das Anzeigen der Hinweisnachricht oder das Einfügen des Roboters in das Inventar, ausgeführt. Es ist möglich auf dem EventBus Listener zu registrieren. Ein Listener wartet auf einen definierten Eventtyp um infolgedessen eine Funktion auszuführen. Auf das Event RobotAdded warten zwei Listener, InventoryListener und DisplayNotificationListener. Sie sorgen für das Anzeigen des Roboters im Inventar sowie Benachrichtigen am Bildschirmrand.

Sind Roboter im Inventar vorhanden, kann dieses geöffnet werden. Der Benutzer erhält eine Übersicht über alle im System registrierten Industrieroboter. (Abb. 17) Wählt der Benutzer einen Roboter aus, wird dieser vor ihm platziert, jedoch nicht verankert. Bewegt der Benutzer seinen Kopf, bewegt sich der Roboter mit ihm. Befindet sich das Hologramm des Roboters in einer vom Benutzer erwünschten Position, kann dieser den Roboter durch einen Tap verankern. Der Roboter wird an dieser Stelle im Raum, bspw. auf einer Tischplatte, fixiert. (Abb. 18)

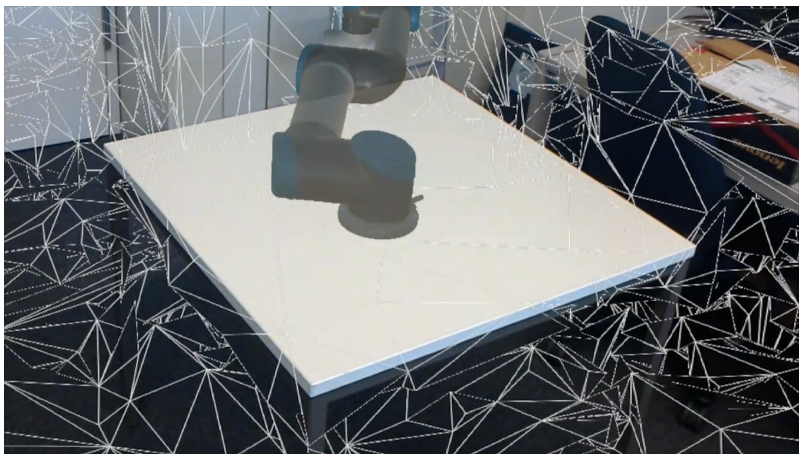

Abbildung 18: Positionierung des Hologramms auf dem Mesh, bspw. auf einer Tischplatte

Nach dem Fixieren wird vom Roboter ein Ladevorgang gestartet. Der Roboter ermittelt sämtliche für ihn verfügbaren Dienste im Netzwerk. Dazu wird erneut der Broker kontaktiert. Jeder Roboter besitzt eine Skript-Komponente, Identität bzw. *Identity*. Diese ermöglicht es dem Roboter den Broker nach Diensten zu fragen. (Alg. 14)

Algorithmus 14 Laden aller verfügbaren Dienste mit ihren Abhängigkeiten

```
public class Identity : MonoBehaviour
{
    // ...

    public void LoadAvailableServices()
    {
        var loader = ServiceLoader.GetInstance();

        loader.LoadService(service.Types.names.GetAxisAngles, this);
        loader.LoadService(service.Types.names.SetAxisAngles, this);

        loader.LoadService(service.Types.names.GetForwardKinematic, this,
            false);
        loader.LoadService(service.Types.names.InterpolateService, this,
            false,
            service.Types.names.SetAxisAngles, service.Types.names.
                GetForwardKinematic);
        loader.LoadService(service.Types.names.GetInverseKinematic, this,
            false,
            service.Types.names.SetAxisAngles, service.Types.names.
                GetForwardKinematic);
        // ...
    }

    // ...
}
```

Die Methode LoadService der Klasse ServiceLoader führt eine Anfrage beim Broker aus. Dazu wird eine entsprechende Protocol Buffers Nachricht an den Broker geschickt. (s. Kapitel 5.1 - Abb. 15; Alg. 15) Enthält die Antwort eine gültige *service_adress*, wird der Dienst gebaut. Das bedeutet, es werden Steuerelemente in der Szene platziert und die Dienste nehmen die Kommunikation mit den Endpunkten auf.

Algorithmus 15 Laden der Adresse des Endpunktes

```
public service_adress RequestService(service.Types.names serviceName,
    string[] identifiers, bool useOrForQuery)
{
    var message = new to_broker()
    {
        Request = new request()
        {
            UseOrForQuery = useOrForQuery,
            Receiver = new device()
            {
                Identifiers = { identifiers }
            },
            Bag = false,
            Service = new service()
            {
                Name = serviceName
            }
        }
    };

    Protocol.SendMessageWithHeader(message);

    var response = Protocol.ReadMessage<response>();

    if (response.ResponseDataCase != response.ResponseDataOneofCase.
        Location)
    {
        throw new ServiceNotFoundException(
            "Invalid response for service ...");
    }

    return response.Location;
}
```

Konnte die Adresse nicht ermittelt werden, wird der Dienst nicht gestartet. Weitere Dienste, die diesen benötigen, werden ebenfalls nicht gestartet.

Der vollständige Ablauf der Netzwerkkommunikation zwischen HoloLens und dem Broker wurde in Abbildung 15 in Kapitel 5.1 bereits veranschaulicht.

Ist die Verbindung zu den einzelnen Diensten aufgebaut, wird der Broker für die weitere Kommunikation nicht mehr benötigt. Die HoloLens schickt infolgedessen alle Anfragen direkt an die Dienste.

5.2.3. Interaktion

Um eine Interaktion mit einem reellen Roboter zu ermöglichen, benötigt die HoloLens für ihr Hologramm die aktuellen Achswinkel des Roboters. Der entsprechende Dienst heißt „GET_AXIS_ANGLES“. Wird der Dienst gestartet, verbindet er sich mit dem reellen Roboter. Um die Anwendung nicht zu blockieren, wird Multi-Threading benötigt. Zu Beginn wird ein neuer Thread gestartet um darin die Methode SyncAxisAngles auszuführen. Die HoloLens wartet in diesem Thread auf eine Nachricht mit neuen Achswinkeln des Robo-

ters. Beim Empfangen einer Nachricht werden die Winkel in einen gemeinsamen Speicher CurrentState von Haupt- und Nebenthread abgelegt. Zusätzlich wird mittels der Flag CurrentStateChanged der Hauptthread über eine Aktualisierung des Speichers informiert. Im nächsten Frame wird das Hologramm mit den neuen Winkeln aktualisiert. (Alg. 16)

Algorithmus 16 Multi-Threading zur Synchronisation im Dienst GET_AXIS_ANGLES

```
public class GetAxisAngleService : Service
{
    // ...
    protected get_axis_angles CurrentState = new get_axis_angles();
    protected bool CurrentStateChanged;

    public override void Start()
    {
        new Task(SyncAxisAngles).Start();
    }

    // Update is running in main thread each frame
    public override void Update()
    {
        lock (CurrentState)
        {
            // check for changes in main thread
            if (!CurrentStateChanged) return;

            // update hologram
            _identity.Interaction.SetAxisAngles(CurrentState.States);

            CurrentStateChanged = false;
        }
    }

    // SyncAxisAngles is running in sub thread
    private void SyncAxisAngles()
    {
        while (SyncClient.IsConnected)
        {
            var angleMessage = proto.ReadMessage<get_axis_angles>();

            lock (CurrentState)
            {
                CurrentState = angleMessage;
                CurrentStateChanged = true;
            }
        }
    }
}
```

Die Interaktion zwischen dem Benutzer, dem Hologramm und somit dem Roboter erfolgt über Gesten. Mittels Tap kann der Benutzer Steuerelemente an den einzelnen Achsen auswählen. Führt er anschließend eine Navigationsgeste aus, wird nach Beendigung dieser eine Nachricht an die Steuerzentrale des Roboters geschickt. Der Roboter führt die Bewegung

aus und schickt nach dem Erreichen der Endposition eine Bestätigung. (Alg. 17)

Algorithmus 17 Setzen von Winkeln am reellen Roboter durch SET_AXIS_ANGLES

```
public class SetAxisAngleService : Service
{
    // send joints asynchron to real robot, use new thread for sending
    public void SendJointStatesAsync()
    {
        // access gameobject in main thread
        var joints = _identity.Interaction.GetAxisAnglesRad();

        // send joints in non blocking thread
        new Task(() =>
        {
            SendJointStates(joints);
        }).Start();
    }

    // send given joints to ur-bridge and wait till completion
    public void SendJointStates(List<float> joints)
    {
        var message = new set_axis_angles
        {
            States = new joint_states { Axis = {joints} }
        };

        Proto.SendMessageWithHeader(message);
        Proto.ReadMessage<joint_states>();
    }
}
```

In Kapitel 4.3 wurden neben dem Dienst SET_AXIS_ANGLES zwei weitere Dienste, GET_INVERSE_KINEMATIC und INTERPOLATE_SERVICE, vorgestellt. Beide besitzen eine Abhängigkeit zu dem Dienst GET_FORWARD_KINEMATIC. (Alg. 14) GET_FORWARD_KINEMATIC berechnet die aktuelle Position des Endeffektors. (s. Kapitel 2.1) Nachdem die Netzwerkadresse des Dienstes für die Vorwärtskinematik ermittelt wurde, verschickt die HoloLens eine Nachricht *calculate_forward* an diesen. Sie beinhaltet die aktuellen Achswinkel. Als Antwort versendet der Dienst die Nachricht *calculate_forward_result*, welche eine Position beinhaltet. Diese Position besteht aus der Translation und der Rotation des Endeffektors.

Um die inverse Kinematik für den Dienst GET_INVERSE_KINEMATIC zu berechnen, wird an der aktuellen Position des Endeffektors mit dem Dienst GET_FORWARD_KINEMATIC ein Dragger-Objekt platziert. (Abb. 19)

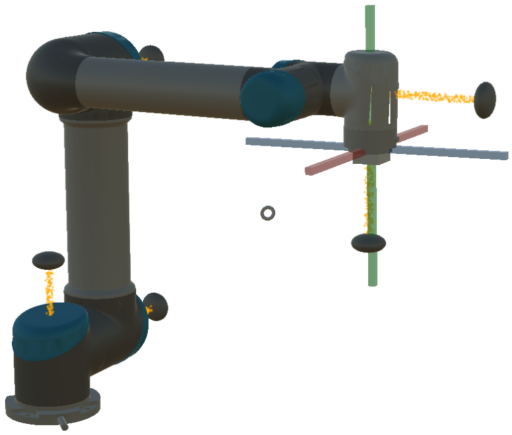

Abbildung 19: Roboter mit Dragger - Steuerung des Endeffektors durch Dragger

Der Benutzer wählt eine der Achsen den Draggers aus und verschiebt bzw. rotiert diese. Dazu wird die ausgewählte Achse beim *NavigateEventManager* registriert und kann über die Gestensteuerung bewegt werden. Im Hintergrund wird beim Bewegen des Draggers eine Protocol Buffers Nachricht *calculate_inverse* an den Dienst geschickt. Der Inhalt der Nachricht sind die aktuellen Gelenkswinkel und die Position des Draggers, welche der gewünschten Position des Endeffektors entspricht. Der Dienst berechnet für die neue Position, mit Hilfe der inversen Kinematik (s. Kapitel 2.1) und den aktuellen Gelenkswinkel, neue Werte für die Achsen. Die aktuellen Winkel werden für die Berechnung der Abweichung benötigt. Die Stellung mit der geringsten Abweichung gegenüber der gegenwärtigen wird verwendet, um die erforderliche Bewegung möglichst gering zu halten.

Der Dienst antwortet nach Berechnung mit der Nachricht *calculate_inverse_result*. Der Inhalt der Nachricht ist die neue Gelenkskonfiguration sowie ein bool'scher Wert. Der Boolean ist bei einem gültigem Ergebnis *true*, sonst *false*. Ungültige Ergebnisse werden vom Dienst geliefert, falls sich der Dragger außerhalb des Arbeitsraumes des Industrieroboters befinden. Ist das Ergebnis gültig, werden die Gelenkswinkel innerhalb der Nachricht an das Hologramm übertragen und die Winkel angepasst. Insgesamt wird die Berechnung maximal acht mal pro Sekunde ausgeführt. Daraus resultiert eine angemessen flüssige Bewegung des Hologramms und reduziert zugleich die Anzahl der versendeten Nachrichten.

Als dritte Methode zur Steuerung des Roboters wurde die Interpolation entlang einer linearen Trajektorie gewählt. Der Benutzer setzt freie Punkte im Raum, welche sich miteinander linear verbinden. (Abb. 20) Die Dragger-Komponente des GET_INVERSE_KINEMATIC Dienstes wurde erneut verwendet. Durch einen doppelten Tap platziert der Benutzer vor sich einen neuen Punkt im Raum. An der Stelle wird ein Dragger-Objekt instanziiert. Das Objekt wird automatisch mit dem vorherigem verbunden. Den Start bildet die Position des Endeffektors. An dieser Position wird mittels GET_FORWARD_KINEMATIC ein Dragger platziert. Der Benutzer kann jeden einzelnen Punkt mit Hilfe der Achsen anpassen. Die Verbindungslinien aktualisieren sich dementsprechend.

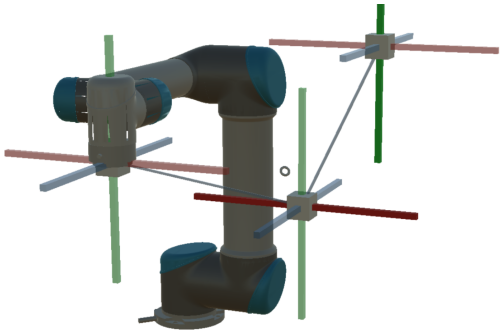

Abbildung 20: Drei definierte Punkte im Raum, die durch den Roboter abgefahren werden

Stellt der Benutzer einen linearer Pfad ein, wird anschließend eine Nachricht an den INTERPOLATE_SERVICE Dienst gesendet. Die Nachricht *interpolate* beinhaltet fünf Felder. Zwei Felder definieren die maximale Geschwindigkeit und Beschleunigung, welche der Roboter ausführen kann. Des Weiteren können viele Industrieroboter eine maximale Anzahl an Steuerbefehlen pro Sekunde entgegennehmen. Der UR-5 verarbeitet in jeder Sekunde maximal acht Befehle zum Setzen seiner Achsen. Diese update_rate wird ebenfalls in der Nachricht übergeben. Schließlich enthält die Nachricht die aktuellen Achswinkel im Feld initial_state sowie die Positionen aller definierten Punkte.

Die Nachricht wird an den Dienst im Netzwerk gesendet und verarbeitet. Das Ergebnis empfängt die HoloLens in Form einer *interpolate_result* Nachricht. Sie beinhaltet für jeden Schritt, den der Roboter ausführen muss, die jeweilige Position der Achsen. Die HoloLens sendet für jeden Schritt eine Nachricht an den SET_AXIS_ANGLES Dienst. Der reelle Roboter aktualisiert daraufhin seine Gelenkswinkel. Da der Dienst GET_AXIS_ANGLES die Position des echten Roboters mit dem des Hologramms synchronisiert, sieht der Benutzer das Resultat unmittelbar vor sich.

5.2.4. Problemlösung

Während der Implementierung der Software RoboViz traten Probleme mit dem Framework der HoloLens auf. Die HoloLens verwendet eine angepasste Windows Distribution, auf welcher *.Net Core* ausgeführt wird. Dieses Framework muss im Zusammenhang mit der Software verwendet werden. Dazu zählen Klassen zur Netzwerkkommunikation oder für das Multi-Threading. Die Programmierumgebung Unity basiert auf dem *.Net Framework*, welches ebenfalls auf Windows 10 installiert ist. Durch diese zwei unterschiedlichen Frameworks ist der Quellcode häufig nicht miteinander kompatibel. Für die Übertragung von Daten wird Googles Protocol Buffers verwendet. Google stellt eine Bibliothek zur Verfügung, die unter *.Net Framework* funktioniert. Auf der HoloLens funktioniert diese Bibliothek hingegen nicht. Der Code musste an mehreren Stellen angepasst werden, sodass abhängig von der Umgebung der richtige Code ausgeführt wird. Bspw. wurde die Methode „GetBuffer“ der Klasse System.IO.MemoryStream in *.Net Core* umbenannt in „TryGetBuffer“. Die Compiler-Flag NETFX_CORE erlaubt es, Code an den entsprechenden Stellen

auszutauschen. (Alg. 18)

Algorithmus 18 Verwendung der Compiler-Flag NETFX_CORE um .Net Core zu unterstützen

```
byte[] bytes;

#if NETFX_CORE
// Code in .Net Core
ArraySegment<byte> buffer;

memoryStream.TryGetBuffer(out buffer);

bytes = buffer.Array;
#else
// orginaler Code in .Net Framework
bytes = memoryStream.Length == memoryStream.Capacity ? memoryStream.
    GetBuffer() : memoryStream.ToArray();
#endif
```

Des Weiteren war das verwendete 3D Modell des UR-5 Roboters fehlerhaft. Die Achsen fünf und sechs wiesen falsche Drehrichtungen auf. Dies wurde korrigiert, indem zusätzliche empty GameObjects in das Robotermodell eingefügt wurden. Diese GameObjects wurden entlang einer orthogonalen Achse der Drehachse um 180° gedreht. Anschließend wurde die betroffene Achse in die korrekte Position rotiert. Aufgrund dieser Vorgehensweise wechselt die Drehrichtung der Drehachse das Vorzeichen.

5.3. Roboter

Die Software zur Robotersteuerung ist in C++ geschrieben. Es wurde die bereits vorhandene Software „ur-bridge“ (s. Kapitel 3) als Grundlage verwendet. Die Steuerungssoftware stellt drei Dienste bereit. Der Dienst GET_AXIS_ANGLES startet auf dem Port 9050 einen TCP/IP Server, welcher auf Netzwerkteilnehmer wartet. Verbindet sich ein Teilnehmer, bspw. eine HoloLens, schickt der Server in regelmäßigen Abständen von 50ms die aktuelle Konfiguration der Gelenkswinkel. (Alg. 19)

Algorithmus 19 Bereitstellen des Dienstes GET_AXIS_ANGLES durch den Server

```
robo_sim::TcpipProtobufServer<RoboSimulation::get_axis_angles,
    RoboSimulation::get_axis_angles> serv_axes_provider (
    port_serv_axes_provider /* 9050 */);
serv_axes_provider.setCallbackFunctionForMessageToBeSent( cbSendAxesVals );
serv_axes_provider.setSleepForMilliSecondsDuringSpin(50);
```

Die aktuellen Gelenkswinkel werden lediglich dann geschickt, wenn sich diese verglichen zur letzten Nachricht geändert haben. Die Methode cbSendAxesVals akzeptiert als Parameter eine Referenz auf die ausgehende Nachricht. Diese enthält keinen Inhalt und kann innerhalb der Funktion, durch die Referenz, verändert werden. Als Rückgabewert liefert die Funktion einen Boolean. Ist der Wert *true* wird die Nachricht verschickt, andernfalls nicht. (Alg. 20)

Algorithmus 20 Callback für das Senden der Achswinkel

```
bool cbSendAxesVals(RoboSimulation::get_axis_angles& msg)
{
        axisAnglesAct = get_actual_robot_angles();
        bool axisAreEqual = std::equal(
        axisAnglesAct.begin(),
        axisAnglesAct.end(),
        axisAnglesOld.begin(),
        [](double value1, double value2)
        {
            return std::fabs(value1 - value2) < epsilon;
        });

        axisAnglesOld = axisAnglesAct;

        if( !axisAreEqual ) {
                msg = build_protobuf_robot_message( axisAnglesAct ); //
                    store inside reference

                return true;
        }

        return false;
}
```

Als weiteren Dienst stellt die ur-bridge SET_AXIS_ANGLES zur Verfügung. Der Dienst ermöglicht bspw. der HoloLens die Gelenkswinkel des Roboters zu manipulieren. Über den Port 9051 schickt die HoloLens *set_ axis_ angles* Nachrichten an die ur-bridge. Die Nachrichten enthalten das Feld states mit *joint_ states*. Wird eine solche Nachricht empfangen, sendet die Software einen Steuerungsbefehl an den Industrieroboter. Der UR-5 wird mittels der Skriptsprache URScript programmiert. ("The URScript Programming Language", 2015)

Algorithmus 21 Funktion zum Setzten der Gelenkswinkel mittels URScript

```
def moving():
        movej([0.000000, -1.570000, 0.000000, -1.570000, 0.000000,
            0.000000],1.400000, 1.050000, 0.000000, 0.000000)
        movej([0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
            0.000000],10.000000, 10.000000, 0.000000, 0.000000)
end
```

Die Hauptfunktion *moving* wird automatisch beim Laden des Programms ausgeführt. Die Funktion movej ist Teil der Sprache und erwartet, unter anderem, sechs Gelenkswinkel. Der Roboter bewegt sich erst in die Achspositionen 0°, −90°, 0°, −90°, 0° und 0°. Im zweiten Schritt stellt der Roboter alle Achsen auf 0°. Innerhalb der ur-bridge werden diese Befehle als Zeichenkette erstellt und an die Roboterschnittstelle gesendet. (Alg. 22) Die Werte werden der Protocol Buffers Nachricht entnommen.

Algorithmus 22 Erstellen der Steuerzeichenkette in URScript und Senden an die Schnittstelle

```
std::string cmd = "";
cmd = "def␣moving():\n";
cmd += "\t";
cmd += urCreateMoveString( q_start_arr, "j", false);
cmd += "\n";
cmd += "\t";
cmd += urCreateMoveString( conf_arr, "j", false, MAX_ACC, MAX_VEL);
cmd += "\n";
cmd += "end\n";

driver_->rt_interface_->addCommandToQueue(cmd);
```

Nach dem Versenden des Befehls wird solange gewartet, bis der Roboter das Programm vollständig ausgeführt hat. Im Anschluss dazu wird eine Protocol Buffers Nachricht an die HoloLens zurückgeschickt. Diese wird informiert, sobald der Roboter mit seiner Bewegung fertig ist.

Der dritte Server auf der ur-bridge nimmt Befehle für einen Greifer an und läuft auf dem Port 9060. Der Greifer wird ebenfalls mittels URScript angesteuert. Empfängt der Server eine *gripper* Nachricht, enthält diese einen Positionswert zwischen 0 und 255. Der Wert 0 steht für komplett offen, der Wert 255 für komplett geschlossen. Der Server erstellt nach dem Empfangen der *gripper* Nachricht das Steuerungsskript und übermittelt es, wie beim SET_AXIS_ANGLES Dienst. Anschließend wartet der Server bis das Programm durchlaufen ist und antwortet mit einer *gripper_complete* Protocol Buffers Nachricht.

5.4. Mehrwertdienste

Die Steuerung des Industrieroboters mittels direkter Eingabe der Gelenkswinkel ist mit Hilfe der HoloLens (Kapitel 5.2.3) und der ur-bridge (Kapitel 5.3) möglich. Es werden weitere Dienste für die Steuerung mit inverser Kinematik oder das Abfahren einer definierten Strecke benötigt. Zwei der Dienste laufen auf einem Server. Die beiden Dienste GET_FORWARD_KINEMATIC und GET_INVERSE_KINEMATIC basieren auf ROS. ("ROS.org | Powering the world's robots", 2018) Es wurde bereits vorhandene Software, wie die ur-bridge, erweitert und an das Netzwerk angepasst. Der ursprüngliche GET_FORWARD_KINEMATIC Dienst lieferte falsche Rotationswerte und wurde korrigiert. Benötigt ein Netzwerkteilnehmer die Vorwärtskinematik, schickt er eine *calculate_forward* Nachricht an den Dienst. Die Nachricht beinhaltet die Gelenkswinkel des Industrieroboters. Als Antwort schickt der Server eine Nachricht vom Typ *calculate_forward_result* mit der Position als Inhalt. (Alg. 23)

Algorithmus 23 Berechnen der Vorwärtskinematik mit Hilfe von ROS

```
void callBackForIncomingForwardMessage(const RoboSimulation::
    calculate_forward &msg)
{
        // read joints from message
        std::vector<double> joints;
        for(signed int axis = 0; axis < msg.joint_states().axis_size();
            axis++)
        {
                joints.push_back(msg.joint_states().axis(axis));
        }

        // calculate forward kinematic with ROS and store in storage
        found_pose_ = kinematics_handler_->forwardKinematics( joints );
}

bool callBackForwardResult(RoboSimulation::calculate_forward_result &msg)
{
        tfScalar x, y, z, roll, pitch, yaw;

        // extract rotation
        found_pose_.getBasis().getRPY(roll, pitch, yaw);

        // extract translation
        x = found_pose_.getOrigin().getX();
        y = found_pose_.getOrigin().getY();
        z = found_pose_.getOrigin().getZ();

        // write result into msg (by reference)
        RoboSimulation::point* point = new RoboSimulation::point;

        msg.set_allocated_position(point);

        point->set_x(x);
        point->set_y(y);
        point->set_z(z);
        point->set_ar(roll);
        point->set_ap(pitch);
        point->set_ay(yaw);

        return true;
}
```

Der inverse Kinematik Dienst GET_INVERSE_KINEMATIC läuft identisch wie der GET_FORWARD_KINEMATIC Dienst. Er erwartet als Eingabe einen Punkt mit x, y und z, sowie *yaw*, *pitch* und *roll*. Optional kann die aktuelle Position des Industrieroboters ebenfalls angegeben werden. Anschließend werden durch ROS die Gelenkswinkel errechnet. Findet der Algorithmus keine gültige Lösung, wird als Gelenkswinkel ∞ zurückgegeben, in jedem anderen Fall die errechneten Winkel.

Um eine lineare Bahn zu berechnen, wird der Dienst INTERPOLATE_SERVICE implementiert. Der Dienst nutzt die Bibliothek KDL von Orocos. ("Kinematic and Dynamic Solvers | The Orocos Project", 2018) Er erwartet eine Protocol Buffers Nachricht vom Typ *interpolate* und sendet als Antwort die Nachricht *interpolate_result*. Als Parameter werden

innerhalb der Nachricht die maximale Geschwindigkeit des Roboters und seine maximale Beschleunigung übergeben. Zusätzlich enthält die Nachricht das Feld update_rate. Dieses enthält die Anzahl der Schritte pro Sekunde, die interpoliert werden sollen. Im letzten Feld werden die Punkte als Liste übergeben. Es wird von Punkt zu Punkt jeweils linear interpoliert.

Im ersten Schritt wird zwischen zwei Punkten eine Bahn, eine Trajektorie, erstellt. (Alg. 24)

Algorithmus 24 Erstellen einer Trajektorie zwischen den Punkten start und target

```
std::shared_ptr<KDL::Trajectory> traj;

KDL::Frame p0(KDL::Rotation::RPY(start[3], start[4], start[5]), KDL::Vector
    (start[0], start[1], start[2]));
KDL::Frame p1(KDL::Rotation::RPY(target[3], target[4], target[5]), KDL::
    Vector(target[0], target[1], target[2]));
KDL::RotationalInterpolation_SingleAxis* ri = new KDL::
    RotationalInterpolation_SingleAxis();

ri->SetStartEnd(p0.M, p1.M);

KDL::Path_Line* path = new KDL::Path_Line(p0, p1, ri, eqrad); // ri is
    deleted by destructor (path_line take ownership)

KDL::VelocityProfile_Trap* vp = new KDL::VelocityProfile_Trap(this->
    max_vel_, this->max_acc_);
vp->SetProfile(0, path->PathLength());

traj.reset<KDL::Trajectory_Segment>( new KDL::Trajectory_Segment(path, vp)
    ); // p & vp are deleted by destructor of Traj_Seg (takes ownership)
```

Im zweiten Schritt wird die Trajektorie schrittweise interpoliert. Die Anzahl der Schritte ist von der Dauer (in s) und der update_rate, der Anzahl der Schritte pro Sekunde, abhängig. Jeder Punkt pro Schritt wird in der Protocol Buffers Nachricht gespeichert (Alg. 25)

Algorithmus 25 Einfügen der Punkte in die Protocol Buffers Nachricht

```
double time_rate = 1 / update_rate;
double time = 0;
while(time < traj->Duration()) {
        KDL::Frame current_frame = traj->Pos(time);

        double roll, pitch, yaw;

        RoboSimulation::point* points = return_msg.add_points();

        points->set_x(current_frame.p.x());
        points->set_y(current_frame.p.y());
        points->set_z(current_frame.p.z());

        current_frame.M.GetRPY(roll, pitch, yaw);

        points->set_ar(roll);
        points->set_ap(pitch);
        points->set_ay(yaw);

        time += time_rate;
}
```

Im Letzten Schritt wird `time = traj->Duration();` gesetzt. Aufgrund dessen ist der letzte Eintrag identisch dem Endpunkt der Trajektorie. Die `return_msg` ist vom Typ *interpolate_result* und wird im Anschluss zurückgeschickt. Der anfragende Teilnehmer erhält eine Liste aller Punkte auf der Bahn.

Durch die Kombination der Dienste GET_FORWARD_KINEMATIC, GET_INVERSE_KINEMATIC, INTERPOLATE_SERVICE und SET_AXIS_ANGLES lässt sich auf der HoloLens eine vollständige Bewegung des Hologramms und reellen Roboters durchführen. Der folgende Algorithmus lässt den Roboter von der aktuellen Position auf einen vorher definierten Punkt (s. Kapitel 5.2) fahren.

Algorithmus 26 Endeffektor des Roboters fährt von der aktuellen Position an einen definierten Punkt

```
var points = new List<point>();

// start position via forward kinematic
points.Add(Identity.Helper.EndeffectorPositionRos());
points.Add(Target);

var response = InterpolateService.CalculateTrajectory(points);

foreach (var point in response.Points)
{
    var joints = InverseKinematicService.CalculateInverse(Identity.
        Interaction.GetAxisAnglesProto(), point);

    if (joints == null)
    {
        Debug.Log("invalid step... aborting");
        return;
    }

        // send joints to ur-bridge
    SetAxisAngleService.SendJointStates(joints);

    await Task.Delay(TimeSpan.FromMilliseconds(update_rate)); // update
        rate in ms
}

Debug.Log("finished trajectory");
```

Eine vollständige Liste der implementierten Dienste kann aus der Tabelle 2 entnommen werden.

Dienst	Anfrage	Antwort	Beschreibung	Port
GET_AXIS_ANGLES		*get_axis_angles*	aktuelle Gelenkswinkel auslesen	9050
SET_AXIS_ANGLES	*set_axis_angles*	*get_axis_angles*	Gelenkswinkel setzen	9051
SET_GRIPPER	gripper	gripper_complete	Greifer öffnen/schließen	9060
GET_PREFAB_FOR_UNITY_HTTP	HTTP (Unity)	HTTP (Unity)	Roboter 3D Modell laden	8080
GET_FORWARD_KINEMATIC	*calculate_forward*	*calculate_forward_result*	Vorwärtskinematik für Gelenkskonfiguration	9052
GET_INVERSE_KINEMATIC	*calculate_inverse*	*calculate_inverse_result*	inverse Kinematik für eine Position	9053
INTERPOLATE_SERVICE	*interpolate*	*interpolate_result*	Interpolation einer Trajektorie zwischen zwei Punkten	9054

Tabelle 2: Implementierte Netzwerkdienste und die verwendeten Protocol Buffers Nachrichten

6. Validierung

6.1. System

Das System dient der Visualisierung und Steuerung von Industrierobotern mit Hilfe von mobilen Endgeräten. Es wurde eine Software, die es ermöglicht mit Hilfe einer HoloLens und Augmented Reality einen Industrieroboter vom Typ UR-5 zu kontrollieren, entwickelt. Um die Funktionalität und Exaktheit der Software zu überprüfen, wurde ein Experiment durchgeführt. Mittels der HoloLens soll ein UR-5 Roboter programmiert werden, ein Element von einem Punkt auf einen anderen zu legen. Im Folgenden wird dokumentiert, wie zuverlässig der Roboter die geplante Bewegung ausführt. Zusätzlich wird gemessen, wie stark das holografische Bild von dem reellen Bild abweicht.

6.1.1. Versuchsaufbau

Der Roboter ist an einer festen Position auf einer Arbeitsplatte fixiert. Ausgerüstet ist er mit einem Greifer. Zum Greifen liegt ein $3,1$cm breiter, $6,2$cm langer und $9,6$cm hoher Gegenstand bereit. Zu Beginn des Experiments bzw. bei Start der Anwendung schaut der Benutzer frontal auf den Roboter.

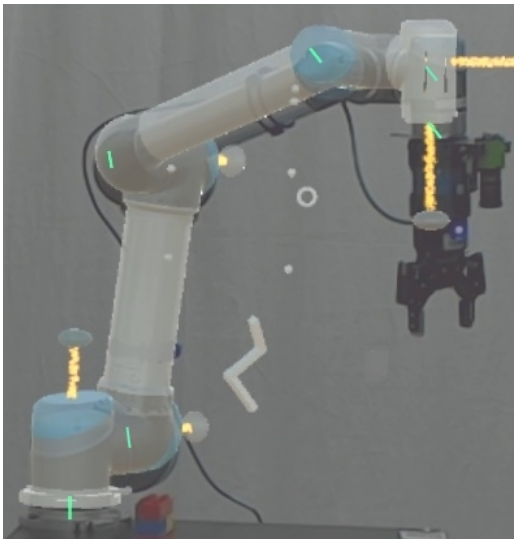

Abbildung 21: Abweichungen (grün) an den Achsen zwischen Hologramm und Roboter

Der Roboter sowie die HoloLens sind in einem gemeinsamen Netzwerk. Zusätzlich stehen alle Dienste aus Tabelle 2 im Netzwerk zur Verfügung.

6.1.2. Versuchsdurchführung

Nachdem der Roboter über das Netzwerk geladen wurde, wird das Hologramm vom Benutzer über den reellen Roboter gelegt. Das Mesh vom Spatial Mapping wird zur Positionierung verwendet. Nach Start der Anwendung und dem Platzieren des Roboters werden die Abweichungen zwischen den Achsen des Hologramms und des Roboters mit einem

Maßband gemessen. Anschließend wird das Programm zum Aufheben des Gegenstandes fünf mal in Folge ausgeführt. Die Anwendung RoboViz wird zwischen den Abläufen nicht neu gestartet, um die Zuverlässigkeit der Anwendung zu validieren. Nach dem fünfmaligem Ausführen wird die Abweichung zwischen dem reellen Roboter und dem Hologramm erneut gemessen. Danach wird die Anwendung RoboViz neu gestartet und der Roboter erneut auf der Arbeitsplatte positioniert. Die Anwendung wird insgesamt fünf mal gestartet, mit jeweils fünf Programmabläufen. Der Gegenstand wird 25 mal aufgehoben und versetzt. Es wird an allen sechs Achsen jeweils zehn mal gemessen.

6.1.3. Messprotokoll

Die verwendeten Variablen:

- i - Nummer des Anwendungsstarts von RoboViz
- j - Nummer der Achse
- s_{ij} - Abweichung (in mm) an Achse j nach dem i-ten Start von RoboViz und vor dem Start des Programms zum Aufheben des Gegenstandes
- t_{ij} - Abweichung (in mm) an Achse j nach dem i-ten von RoboViz und fünf Programmdurchläufen
- M_j - Median der Abweichung im j-ten Start von RoboViz

j	1		2		3		4		5		6				
i	s_{i1}	t_{i1}	s_{i2}	t_{i2}	s_{i3}	t_{i3}	s_{i4}	t_{i4}	s_{i5}	t_{i5}	s_{i6}	t_{i6}	$\sum$	$\frac{\sum}{12}$	M_j
1	3	5	4	15	10	10	25	22	30	27	37	33	221	18,4	15
2	19	37	31	35	33	28	29	29	14	19	21	21	316	26,3	28
3	8	11	9	13	9	14	12	21	19	23	20	24	183	15,3	13
4	18	16	18	15	25	25	25	26	20	23	29	28	268	22,3	23
5	21	21	19	21	25	24	29	30	29	25	27	25	296	24,7	25

Tabelle 4: Messergebnisse der Abweichungen (in mm) zwischen Hologramm und reellem Roboter

6.1.4. Versuchsauswertung

Von den 25 durchgeführten Versuchen wurden alle erfolgreich abgeschlossen. Der Greifer hob den Gegenstand in jedem Durchlauf von derselben Stelle auf und platzierte ihn immer an der vorher definierten Position. Für die Abweichung A zwischen dem Hologramm und dem reellen Roboter wird der Durchschnitt berechnet. Die Messung s_{ij} bezeichnet die Messung nach dem Start der Anwendung im i-ten Durchlauf an der j-ten Achse. Die Messung

t_{ij} bezeichnet analog dazu die Messung nach dem Durchlaufen der fünf Wiederholungen.

$$A = \sum_{i=1}^{5} \sum_{j=1}^{6} \frac{(s_{ij} + t_{ij})}{2}$$

Die gesamte Abweichung beträgt im Durchschnitt $A = 21,4$mm. Zusätzlich werden die Summen aller Messungen pro Anwendungsstart, der Durchschnitt der Messungen und der Median ermittelt.

$$a = \sum_{i=1}^{5} \frac{(s_{i6} + t_{i6})}{2}$$

An Position $j = 6$ weist der Endeffektor eine Abweichung a von durchschnittlich $26,5$mm auf. Aufgrund der perspektivischen Verzerrung zwischen Hologramm und Roboter beträgt der Messfehler 2mm.

6.2. Software

Der Broker ist das Zentrum des Netzwerkes. Fällt dieser aus, können keine neuen Verbindungen aufgenommen werden. Dennoch laufen bestehende Verbindungen zwischen Diensten, Robotern und Anzeigegeräten weiter. Daher ist eine fehlerfreie Funktionsweise des Brokers entscheidend, um die Funktionsfähigkeit des gesamten Netzwerkes zu gewährleisten. Der Broker basiert auf Komponenten. Diese verwalteten größere Bereiche der Software, wie bspw. das Geräteverzeichnis *device repository.* (Abb. 22) Eine Komponente besteht aus vielen kleineren Softwareteilen, sogenannten *Units.* Eine Software-Unit führt genau eine Aufgabe innerhalb einer Komponente aus. Sie sind die kleinste logische Einheit in der Software.

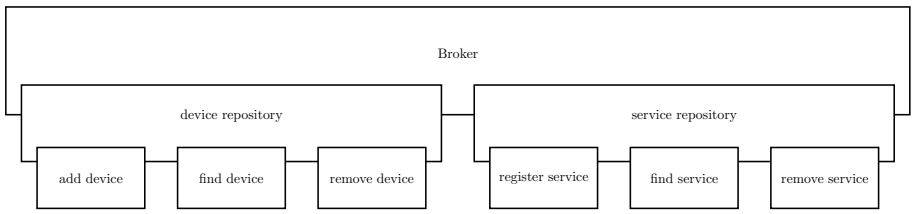

Abbildung 22: Broker Software-Architektur

Die gesamte Software funktioniert nur dann fehlerfrei, wenn sämtliche Komponenten der Software dies ebenfalls sind. Eine Komponente ist wiederum fehlerfrei, wenn auch sämtliche *Units* ohne Fehler sind. Da Units kleine Softwarebereiche sind und definierte Aufgaben übernehmen, lassen sich diese gut testen. Diese Tests heißen *Unit-Tests.*

Die Funktionsweise und das Testen sollen anhand des Beispiels „add device“ beschrieben werden. Die Anfangs- und Ausgangsbedingungen einer Unit sind definiert. Der Zustand ist die aktuelle Liste des Geräteverzeichnisses. Zu Beginn ist die Liste leer. Bei Eingabe eines Gerätes, wird eine Liste mit einen Gerät ausgegeben. Dieser Ablauf lässt sich testen.

(Alg. 27) Der Test endet mit dem Aufruf eines *assert* Befehls. Dieser überprüft, ob die erforderliche Ausgangssituation eintrifft. Andernfalls wird eine Fehlermeldung ausgegeben und das Testergebnis ist negativ.

Algorithmus 27 Unit-Test zum Hinzufügen eines Gerätes in ein leeres Repository

```
def test_a_device_can_be_added_to_the_repository(self):
    # creating of state is not required, repository is empty by default

    # create input
        greeting = self.buildGreeting("some-unique-id",
        ["identifier-1", "..."])

    self.resolver.addDevice(greeting, "channel")

    self.assertEqual(len(self.resolver.list_of_devices), 1,
        "one device should be in repository")
```

Andere *Unit*s, wie „find service“, werden ebenfalls getestet. Dazu muss zunächst für jeden Testdurchlauf derselbe Ausgangszustand hergestellt und eine Eingabe erzeugt werden. (Alg. 28) Am Ende wird die Ausgabe der Unit erneut mit einem *assert* Befehl getestet.

Algorithmus 28 Unit-Test zum Finden eines Services im Repository

```
def test_a_service_can_be_resolved(self):
    # register services (state)
    greeting = self.buildGreeting(
        "some-unique-id", ["robot", "arm"],
        [[service_msg.GET_AXIS_ANGLES, 1234],
        [service_msg.GET_INVERSE_KINEMATIC, 1234],]
    )

    self.resolver.addServices(greeting, "channel")

    # build service request (input)
    device = device_msg()
    device.identifiers.extend(["some-unique-id"])

    service = service_msg()
    service.name = service_msg.GET_AXIS_ANGLES

    # resolve channel service (result)
    channel = self.resolver.resolveFirstService(device, service)

    # test result
    self.assertEqual(channel.channel, "channel",
        "resolved channel did not match")
```

Jede Unit wird mit unterschiedlichen Zuständen und Eingaben getestet. Je nach Ergebnis wird eine Fehlermeldung oder ein „OK“ nach Ende des Testdurchlaufes ausgegeben.

(Abb. 23) Dabei sollten nicht nur gängige Eingaben getestet werden. Diese führt die Software meist korrekt aus. Wichtiger ist es, falsche Eingaben und die Reaktionen der Software darauf, zu testen. Wird bspw. die korrekte Fehlermeldung erzeugt, wenn ein Dienst nicht vorhanden ist? Was passiert, wenn ein Gerät versucht sich mehrmals anzumelden? Wie reagiert der Broker, wenn ein Gerät entfernt werden soll, welches nicht in der Liste auftaucht?

Wie Abbildung 23 zu entnehmen ist, wurden 15 Units getestet. Alle Tests wurden erfolgreich absolviert.

Abbildung 23: Ergebnisse eines Unit-Tests vor und nach Behebung eines Softwarefehlers

Durch die unterschiedlichen Tests der einzelnen Units, ist die korrekte Funktionsweise jeder größeren Komponente gewährleistet. Dies wiederum sorgt für eine zuverlässige Software. Der vollständige Code des Brokers kann nicht durch Unit-Tests überprüft werden, da nicht jeder Code zu einer Unit gehört. Bspw. der Code, der einzelne Units miteinander verbindet, muss separat geprüft werden. Insgesamt konnte der Broker umfangreich getestet werden. Folglich ist fehlerfreie Funktionsweise garantiert.

6.3. Diskussion

Die Durchführung eines Versuches und das Testen mittels Unit-Tests haben gezeigt, dass das Netzwerk mit allen Teilnehmern funktioniert. Der UR-5 Roboter lässt sich mit der HoloLens programmieren und kann wiederholt eine Aufgabe ausführen. Auch äußere Einwirkungen, wie das manuelle Bewegen des Roboters zwischen Programmabläufen, irritierten das System nicht. Die Änderungen werden automatisch über das Netzwerk synchronisiert. Der Roboter lässt sich durch drei Eingabemethoden steuern. Die erste Variante ist die Veränderung der Achsen des Hologramms durch Gesten. Der reelle Roboter bewegt sich je nach Gestensteuerung in die neue Position. In der zweiten Form der Steuerung lässt sich die Position des Endeffektors mit Hilfe der GUI verändern. Hierzu zeigt das Hologramm eine Vorschau der neuen Roboterposition. Nach Bestätigung bewegt sich der Industrieroboter in die Position. Die dritte Eingabemethode ist die lineare Punkt-zu-Punkt Bewegung durch Interpolation. Sie verwendet intern die beiden anderen Eingabemethoden und erweitert diese.

Um den Roboter interaktiv zu steuern, wurde das Hologramm über den Industrieroboter

gelegt. Die Sensoren der HoloLens sind in der vorliegenden Version nicht präzise genug, um das Hologramm exakt an der Umgebung auszurichten. Dies lässt sich ebenfalls den Messwerten entnehmen. Das Hologramm weicht im Durchschnitt um $21,4$mm vom reellen Roboter ab. Diese Abweichung resultiert aus der Positionierung des Hologramms auf dem Mesh. Der Benutzer muss die Abweichung durch Einberechnung dieser kompensieren. Für präzise Aufgaben, wie Schweißarbeiten, kann die HoloLens nicht verwendet werden. Hingegen für andere Aufgaben, wie das Greifen und Platzieren von Gegenständen, ist die HoloLens problemlos einsetzbar.

7. Zusammenfassung und Ausblick

Ziel der Arbeit war es ein Netzwerk zu schaffen, welches einen dynamischen Aufbau ermöglicht. Mobile Endgeräte müssen keine genauen Funktionen über Netzwerkteilnehmer besitzen, um diese zu steuern. Der Fokus lag auf der Visualisierung und Steuerung eines beliebigen Industrieroboters. Ein Netzwerk, welches als Topologie eine Hybridstruktur zwischen Peer-to-Peer und Client/Server-Netzwerk aufweist, wurde erfolgreich implementiert. Im Zentrum des Netzwerks steht der Broker, welcher die einzelnen Teilnehmer untereinander vermittelt. Nach der Vermittlung läuft die Kommunikation direkt zwischen den Teilnehmern. Das Endgerät, bzw. die HoloLens ermittelt zur Laufzeit die Roboter, sowie die benötigten Dienste und lädt diese über das Netzwerk. Anschließend wird der Roboter über ein Hologramm visualisiert und durch Gesten gesteuert. 24

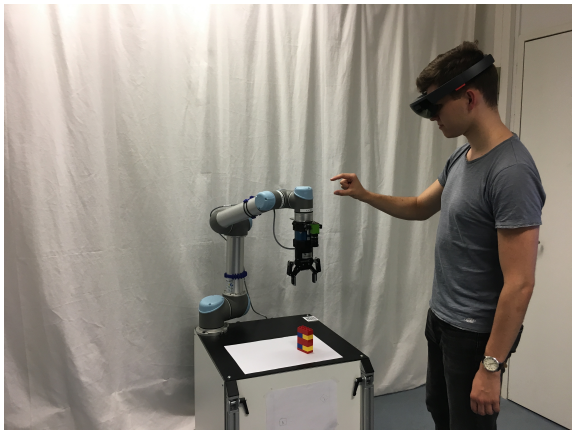

Abbildung 24: Steuerung des Industrieroboters mittels Gesten

Durch einen Versuch (s. Kapitel 6.1) hat sich herausgestellt, dass die Sensorik der HoloLens ungenau ist. Durch das verwendete Spatial Mapping zur Positionierung des Hologramms weicht dieses um durchschnittlich 21,4mm vom reellen Roboter ab. Zusätzlich bereitete die Plattform der HoloLens während der Implementierung zahlreiche Probleme. Die HoloLens verwendet das proprietäre Framework *.Net Core*. Dieses ist mit zahlreichen bestehenden Bibliotheken, wie Googles Protocol Buffers, inkompatibel. Die Bibliotheken wurden an das Framework angepasst und funktionieren sowohl unter *.Net Framework* als auch *.Net Core*.

Es ist möglich den Roboter mit Hilfe der HoloLens zu programmieren. Einige Funktionen wurden nicht implementiert. Bspw. existiert für den Greifer kein 3D Modell, welches durch Gesten manipuliert werden kann. Der Greifer kann jedoch über einen Dienst in Programmen eingebunden werden.

Eine Möglichkeit um die ermittelte Abweichung zwischen Hologramm und Industrieroboter zu verringern, ist die Implementierung eines Marker-Trackings. Das Hologramm wird folglich anhand eines exakt positionierten Markers ausgerichtet. Bestehende Tracking-Systeme sind allerdings nicht mit der HoloLens kompatibel. Des Weiteren können durch

die HoloLens zusätzliche Sensoren visualisiert werden. Es ist möglich die Punktwolke einer Kinect mit Diensten aufzuarbeiten und anschließend in der HoloLens anzuzeigen. ("Kinect – Entwicklung von Windows-Apps", 2018) Die Gestenerkennung kann mittel externer Hardware, wie Leap Motion, optimiert werden. ("Leap Motion", 2018)

Nicht nur die Visualisierung und Steuerung, auch das Netzwerk lässt sich um zahlreiche Funktionen, wie bspw. das Zusammenstellen von Robotergruppen, erweitern. Zusätzlich ist ein Authentifizierungssystem für den produktiven Einsatz notwendig. Auf diese Weise ist gewährleistet, dass lediglich Nutzer mit einer Berechtigung Roboter steuern und Dienste nutzen können. Die Steuerung weiterer Werkzeuge kann ebenfalls implementiert werden.

Das Vorhaben, Industrieroboter mittels Augmented Reality zu steuern, wurde erfolgreich umgesetzt. Dennoch weist das Thema „Entwurf und Entwicklung eines Systems zur Visualisierung und Steuerung von Industrierobotern auf mobilen Endgeräten“ ein großes Potenzial zur Weiterentwicklung auf.

Abbildungsverzeichnis

Literatur

AP, "Deutsche Unternehmen flüchten zunehmend ins Ausland," *Frankfurter Allgemeine Zeitung*, vol. Mai 2003, 2003. [Online]. Available: http://www.faz.net/aktuell/wirtschaft/standortwettbewerb-deutsche-unternehmen-fluechten-zunehmend-ins-ausland-1103961.html

D. Creutzburg, "Deutsche Unternehmen verlagern Investitionen ins Ausland," *Frankfurter Allgemeine Zeitung*, vol. April 2015, 2015. [Online]. Available: http://www.faz.net/aktuell/wirtschaft/unternehmen/deutsche-unternehmen-draengen-wieder-ins-ausland-13534175.html

J. Gemma, S. Wyatt, and G. Litzenberger. How robots conquer industry worldwide. Abgerufen: 2018-05-07. [Online]. Available: https://ifr.org/downloads/press/Presentation_PC_27_Sept_2017.pdf

W. Weber, *Industrieroboter. Methoden der Steuerung und Regelung.* Fachbuchverlag Leipzig, 2002.

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control (Artificial Intelligence).* The MIT Press, 1981.

S. Cubero, Ed., *Industrial Robotics: Theory, Modelling and Control.* Pro Literatur Verlag, Germany / ARS, Austria, 2007, ch. 4, pp. 117–125.

ur kinematics at indigo universal robots. Abgerufen: 2018-05-09. [Online]. Available: https://github.com/ros-industrial/universal_robot/tree/indigo/ur_kinematics

Kinematic and dynamic solvers | the orocos project. Abgerufen: 2018-03-16. [Online]. Available: http://www.orocos.org/kdl/UserManual/kinematic_solvers

Oculus rift | oculus. Abgerufen: 2018-05-09. [Online]. Available: https://www.oculus.com/rift/#oui-csl-rift-games=robo-recall

Vive | discover virtual reality beyond imagination. Abgerufen: 2018-05-09. [Online]. Available: https://www.vive.com/de/

ITU-T, "Information technology – open systems interconnection – basic reference model: The basic model," International Telecommunication Union, Recommendation X.200, Apr. 1994.

M. Bawa, B. F. Cooper, A. Crespo, N. Daswani, P. Ganesan, H. Garcia-Molina, S. Kamvar, S. Marti, M. Schlosser, Q. Sun, P. Vinograd, and B. Yang, "Peer-to-peer research at stanford," *SIGMOD Rec.*, vol. 32, no. 3, pp. 23–28, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/945721.945728

R. Schollmeier, "[16] a definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proceedings of the First International Conference on Peer-to-Peer Computing*, ser. P2P '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 101–.

B. F. Cooper and H. Garcia-Molina, "Sil: Modeling and measuring scalable peer-to-peer search networks," in *Databases, Information Systems, and Peer-to-Peer Computing*, K. Aberer, M. Koubarakis, and V. Kalogeraki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 2–16.

A. Vick, V. Vonásek, R. Pěnička, and J. Krüger, "Robot control as a service — towards cloud-based motion planning and control for industrial robots," in *2015 10th International Workshop on Robot Motion and Control (RoMoCo)*, July 2015, pp. 33–39.

L. Turnbull and B. Samanta, "Cloud robotics: Formation control of a multi robot system utilizing cloud infrastructure," in *2013 Proceedings of IEEE Southeastcon*, April 2013, pp. 1–4.

L. Agostinho, L. Olivi, G. Feliciano, F. Paolieri, D. Rodrigues, E. Cardozo, and E. Guimaraes, "A cloud computing environment for supporting networked robotics applications," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, Dec 2011, pp. 1110–1116.

G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel, "Rapyuta: A cloud robotics platform," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 481–493, April 2015.

M. van de Molengraft, M. Steinbuch, O. Zweigle, and P. Levi. Roboearth | a world wide web for robots. Abgerufen: 2018-05-10. [Online]. Available: http://roboearth.ethz.ch/

C. Horn and J. Krüger, "Feasibility of connecting machinery and robots to industrial control services in the cloud," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2016, pp. 1–4.

A. Vick, C. Horn, M. Rudorfer, and J. Krüger, "Control of robots and machine tools with an extended factory cloud," in *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, May 2015, pp. 1–4.

G. Hu, W. P. Tay, and Y. Wen, "Cloud robotics: architecture, challenges and applications," *IEEE Network*, vol. 26, no. 3, pp. 21–28, May 2012.

D. Lorencik and P. Sincak, "Cloud robotics: Current trends and possible use as a service," in *2013 IEEE 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Jan 2013, pp. 85–88.

R. Doriya, P. Chakraborty, and G. C. Nandi, "Robotic services in cloud computing paradigm," in *2012 International Symposium on Cloud and Services Computing*, Dec 2012, pp. 80–83.

Wuttke/Henke, *Schaltsysteme - eine automatenorientierte Einführung.* Pearson Studium, 2003.

Unity. Abgerufen: 2018-05-16. [Online]. Available: https://unity3d.com/de/

Install the tools. Abgerufen: 2018-03-12. [Online]. Available: https://developer.microsoft.com/en-us/windows/mixed-reality/install_the_tools

Urscript api reference - scriptmanual_en_3.1.pdf. Abgerufen: 2018-05-27. [Online]. Available: http://www.sysaxes.com/manuels/scriptmanual_en_3.1.pdf

Ros.org | powering the world's robots. Abgerufen: 2018-05-27. [Online]. Available: http://www.ros.org/

Kinect – entwicklung von windows-apps. Abgerufen: 2018-05-26. [Online]. Available: https://developer.microsoft.com/de-de/windows/kinect

Leap motion. Abgerufen: 2018-04-25. [Online]. Available: https://www.leapmotion.com/

A. Installationsanleitung

A.1. Voraussetzungen

Für die Installation des Systems werden folgende Komponenten vorausgesetzt. Python wird für die Ausführung des Brokers benötigt. Visual Studio, Unity, .Net Framework und das Windows Universal SDK werden für die Anwendung RoboViz der HoloLens verwendet. Für die Mehrwertdienste werden ROS, Catkin und Cmake benutzt.

A.2. Broker

Der Broker verwendet Python, eine Skriptsprache. Diese müssen nicht compiliert werden. Der Broker wird mit dem Befehl `python broker.py` gestartet.

A.3. HoloLens

Die Anwendung der HoloLens muss erstellt werden. Im ersten Schritt wird die Anwendung RoboViz mit Unity geöffnet. Anschließend wird unter „File →Build Serttings...“ die Anwendung für die Universal Windows Plattform gebaut. Nach erfolgreichem Bauen öffnet sich automatisch der Windows Explorer mit dem Build-Verzeichnis. Nach dem Öffnen der .sln-Datei mit Visual Studio kann die Anwendung auf der HoloLens installiert werden. Dazu muss als Systemtyp „x86“ ausgewählt werden. Im Anschluss daran wird in der Liste der verfügbaren Geräte die HoloLens ausgewählt. Diese muss per USB-Kabel mit dem Computer verbunden sein. Nach der Installation wird die Anwendung automatisch gestartet.

A.4. Roboter und Mehrwertdienste

Sowohl die Software der Robotersteuerung ur-bridge, als auch die Mehrwertdienste werden unter Linux compiliert. Im Falle des „interpolator“ und der „ur-bridge“ muss im Terminal mittels `cd path-to-project/build` in das Unterverzeichnis „build“ gewechselt werden. Je nach System muss ggf. Cmake konfiguriert werden. Dazu wird im build-Order `ccmake ..` ausgeführt. Der Pfad zur Protocol Buffers Bibliothek muss entsprechend eingetragen werden. Anschließend wird mit dem Befehl `make` die Software compiliert. Abhängig vom Projekt wird die Anwendung mittels `./ur-bridge` oder `./interpolator` gestartet.

Der Mehrwertdienst für die Vorwärtskinematik und die inverse Kinematik muss im Catkin Workspace liegen. Nach dem Ausführen von `catkin_make` wird der Dienst mit `roslaunch inverse_kinematic_service launch_without_robot.launch` gestartet.

B. Danksagung

Im Anschluss an diese Arbeit möchte ich zuerst meinen Betreuern, Prof. Dr. Daniel Göhring und Prof. Dr. Jörg Krüger, danken, die mir stets unterstützend zur Seite standen. Ebenso möchte ich Jan Guhls intensive Unterstützung und Ratschläge an schwierigen Stellen anerkennen.

Zusätzlich möchte ich bei meiner Familie, insbesondere Juliane Herm, und meiner Freundin Nikola Tomkow bedanken, die mich stets motiviert haben.

Abschließend möchte ich meine Kommilitonen Son Tung Nguyen, David Bohn, Luca Keidel und Fabian Reimeier erwähnen, die immer eine hilfreiche Meinung zu meiner Arbeit hatten.