

# Entwicklung eines Ad-Hoc-Kommunikationsansatzes für autonome Modellautos

**Developing an ad-hoc-communication for autonomous modelcars**

Bachelor-Thesis eingereicht von

Marcel Mann

am 28. April 2017



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



Fachgebiet Echtzeitsysteme

Elektrotechnik und  
Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr  
Merckstraße 25  
64283 Darmstadt

[www.es.tu-darmstadt.de](http://www.es.tu-darmstadt.de)

Gutachter: Prof. Dr. rer. nat. A. Schürr  
Betreuer: M.Sc. Géza Kulcsár

ES-B-0126



---

# **Erklärung zur Bachelor-Thesis**

Hiermit versichere ich, die vorliegende Bachelor-Thesis selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die ich aus fremden Quellen direkt oder indirekt übernommen habe, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Ich erkläre mich damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 28. April 2017

---

(Marcel Mann)

---



---

---

## **Zusammenfassung**

---

Die vorliegende Bachelorthesis beschäftigt sich mit der Entwicklung und Implementierung eines Ad-hoc-Kommunikationsansatzes für die autonomen Modellautos des Fachgebiets Echtzeitsysteme an der TU Darmstadt. Ziel ist es, den Autos die Möglichkeit zu geben, selbstständig ein drahtloses Kommunikationsnetz zu erstellen und zu verwalten, sodass Studenten im Rahmen des Projektseminars Echtzeitsysteme in der Lage sind, kooperative Aufgabenstellungen des autonomen Fahrens oder der autonomen Robotik zu lösen. Der entwickelte Kommunikationsansatz basiert auf dem ROS-Paket *adhoc\_communication* und erweitert dieses um eine vereinfachte und flexiblere Schnittstelle, um beliebige Daten zu versenden. Teil dieser Arbeit ist auch das Evaluieren des Ansatzes, wozu eine eigene Testsoftware entworfen wurde. Es wird gezeigt welche Möglichkeiten sich unter Verwendung der neuen Software erschließen, aber auch, welche Probleme und Schwierigkeiten dabei auftreten.



---

---

## Inhaltsverzeichnis

---

<b>1 Einführung</b>	<b>1</b>
1.1 Problemstellung . . . . .	2
1.2 Ziele . . . . .	3
1.3 Aufbau der Arbeit . . . . .	3
<b>2 Grundlagen</b>	<b>5</b>
2.1 Kommunikation . . . . .	5
2.1.1 Bluetooth . . . . .	5
2.1.2 Wireless-LAN . . . . .	6
2.2 Routing . . . . .	7
2.2.1 AODV . . . . .	9
2.2.2 DSR . . . . .	11
2.2.3 Beispiel . . . . .	11
2.3 ROS . . . . .	13
2.3.1 Messages . . . . .	14
2.3.2 Publish/Subscribe . . . . .	14
2.3.3 Services . . . . .	15
2.3.4 Parameter-Server . . . . .	16
2.4 Serialisierung . . . . .	17
<b>3 Umsetzung</b>	<b>19</b>
3.1 Modellautos . . . . .	19
3.1.1 Rechnerhardware . . . . .	19
3.1.2 Chassis, Fahrmechanik, Mikrocontroller und Sensoren . . . . .	21
3.1.3 Software . . . . .	21
3.2 adhoc_communication-Paket . . . . .	21
3.2.1 Inbetriebnahme . . . . .	22
3.2.2 Datenfluss . . . . .	22
3.3 Zusätzliche Serialisierung . . . . .	24
3.4 Wiederverwendung . . . . .	25
<b>4 Evaluation</b>	<b>27</b>
4.1 Forschungsfragen . . . . .	27
4.2 Evaluations-Setup und Testumgebung . . . . .	27
4.2.1 Software . . . . .	27
4.2.2 Testumgebung . . . . .	30
4.3 Allgemeine Tests . . . . .	32
4.3.1 Reichweite . . . . .	32
4.3.2 Korrektheit . . . . .	32
4.3.3 Overhead . . . . .	33

---

---

4.4	Leistung Direktverbindungen . . . . .	34
4.4.1	Senderate . . . . .	34
4.4.2	Latenz . . . . .	34
4.4.3	Paketumlaufzeit . . . . .	35
4.4.4	Daten . . . . .	37
4.5	Leistung indirekte Verbindung . . . . .	37
4.5.1	Latenz . . . . .	37
4.5.2	Daten . . . . .	38
4.6	Diskussion . . . . .	38
4.6.1	Reichweite . . . . .	38
4.6.2	Latenz . . . . .	38
4.6.3	Übertragungsrate . . . . .	39
4.7	Schlussfolgerungen . . . . .	39
<b>5</b>	<b>Fazit und Ausblick</b>	<b>41</b>
5.1	Verwandte Arbeiten . . . . .	41
5.2	Fazit . . . . .	41
5.3	Ausblick . . . . .	42

---

---

## Abbildungsverzeichnis

---

1.1	Sensornutzung in einem modernen Auto [Jun16] . . . . .	1
2.1	Netzwerktopologien . . . . .	8
2.2	Verbindungsarten . . . . .	8
2.3	Routenfindung bei AODV [CBR04] . . . . .	10
2.4	Beispielnetzwerk . . . . .	11
2.5	Illustrierung des ROS-Graph [Rob15] . . . . .	13
3.1	Modellauto . . . . .	19
3.2	Rechenhardware im Detail . . . . .	20
3.3	Veranschaulichung der Abhangigkeiten . . . . .	26
4.1	Grundriss der Testumgebung mit Messpositionen . . . . .	31
4.2	Abdeckung des Funksignals des Senders . . . . .	32
4.3	Ubertragungsdauer in Abhangigkeit von Netto-Datenmenge . . . . .	33
4.4	Abhangigkeit der Latenz von Entfernung . . . . .	35

---

## Tabellenverzeichnis

---

2.1	Brutto- vs. Nettodatenrate nach WLAN-Standard [avm16] . . . . .	7
4.1	Distanzen der Messstellen . . . . .	31
4.2	Messwerte Ping auf geringe Distanz [ms] . . . . .	34
4.3	Pingwerte . . . . .	35
4.4	ICMP-Echo Latenzen . . . . .	36
4.5	Ubertragungsdauer von 100 kB . . . . .	37

---

## Quellcodeverzeichnis

---

2.1	Beispielhafter Aufbau der Message-Datei <i>Student.msg</i> . . . . .	14
2.3	Ein einfacher Subscriber fur String-Messages [ROS16b] . . . . .	15
2.2	Ein einfacher Publisher fur String-Messages [ROS16b] . . . . .	15
2.4	Servicedatei <i>AddTwoInts.srv</i> [ROS16a] . . . . .	16
2.5	Callbackfunktion und Service-Server [ROS15] . . . . .	16
2.6	Serviceaufruf [ROS15] . . . . .	17
2.7	Launch-Datei zum Starten des Senders (Abschnitt 4) . . . . .	17
2.8	Serialisierte Student-Message in Hexadezimaler Schreibweise . . . . .	18
3.1	Skriptdatei zum Konfigurieren der Schnittstelle . . . . .	22
3.2	Callbackfunktion des <i>sendStudent</i> -Service . . . . .	23
3.3	Ausschnitt aus <i>publishPacket</i> . . . . .	23
3.4	Definition der <i>sendMessage</i> -Methode . . . . .	25
4.1	Pseudocode des Senders . . . . .	29
4.2	Receive-Time-Message . . . . .	29
4.3	Pseudocode All-in-One . . . . .	30
4.4	Kommandozeilenauftrag <i>Ping6</i> . . . . .	36



## 1 Einführung

Das Automobil hat eine lange Entwicklungsgeschichte hinter sich. Eines der ersten Probleme war die Entwicklung mobiler Motoren, die klein und handlich genug waren um auf einer Kutsche Platz zu finden, und die bis dahin vorherrschenden Pferde als Leistungsbringer abzulösen. Die bis weit ins 19. Jahrhundert genutzten Dampfmaschinen waren zu groß, zu schwer und ihr Energielieferant, die Kohle, benötigte zu viel Platz in der Lagerung, sodass Dampfmaschinen hauptsächlich stationär oder auf Schiffen und Zügen zum Einsatz kamen. Erst die Entwicklung von Verbrennungsmotoren ebnete den Weg für den Siegeszug des modernen Automobils.

Während der Antrieb immer weiter verbessert wurde, geriet die Sicherheit bei der Entwicklung neuer Fahrzeuge immer weiter in den Vordergrund. Fahrgastzelle, Sicherheitsgurte und Airbags sind heute längst standardmäßige Sicherheitsfeatures. Diese Systeme sind alle passiv, sie schützen erst wenn es bereits zu einem Unfall gekommen ist. Aktive Sicherheitssysteme hingegen versuchen Unfälle von vornherein zu verhindern.



**Abbildung 1.1: Sensornutzung in einem modernen Auto [Jun16]**

Um aktiv in das Geschehen eingreifen zu können, muss ein Auto einige Fähigkeiten eines Menschen beherrschen, zum Beispiel das Wahrnehmen der direkten Umgebung. Mithilfe vielfältiger Sensoren (vgl. Abbildung 1.1) und intelligenten Steuerungen können moderne Autos ein großes Plus an Sicherheit bieten.

Kameras erkennen selbstständig Hindernisse und können Notbremsungen einleiten. Spurassistenten können die Fahrspur halten und so vor versehentlichen Spurübertretungen schützen. Wird die Spur gehalten und das Tempo dem vorrausfahrenden Auto angepasst, nennt man das Automatic-Cruise-Control. Mit einer solchen kann ein Auto auf der Autobahn sich fast von alleine steuern.

---

Es ist zu erkennen, dass die Technik in bestimmten, überschaubaren Situationen den Autofahrer ergänzen und teilweise sogar schon ersetzen kann. Der nächste logische Schritt der Automobilentwicklung geht folglich dahin, den Menschen komplett zu ersetzen und Autos autonom, ohne Eingreifen des Menschen, fahren zu lassen.

Durch die Nutzung von lokalen, in das Auto eingebaute Sensorsysteme lässt sich jedoch nur die direkte Umgebung auswerten. Um einen zusätzlichen Informationsgewinn zu erhalten, muss das Auto jedoch über den durch die Sensoren erfassbaren Horizont hinaus schauen können. Eine Möglichkeit besteht darin, die Umgebung nicht nur auszuwerten, sondern auch mit ihr zu kommunizieren. Erfährt das Auto von der Notbremsung des übernächsten Autos nicht nur durch das verzögerte Bremsen des direkt vorausfahrenden Verkehrsteilnehmers, sondern durch eine elektronische Benachrichtigung, lässt sich die so gewonnene Zeit zu einer sicheren Bremsung nutzen.

Zum Autofahren ist nicht nur das Beobachten der Umgebung, sondern auch die Kommunikation mit ihr entscheidend. Kommen vier Fahrzeuge gleichzeitig an eine Kreuzung, an der die Rechts-vor-Links-Regel gilt, so muss die Vorfahrt unter den Teilnehmern ausgehandelt werden. Zur Umgebung auf der Straße zählen jedoch nicht nur andere Verkehrsteilnehmer. Auch intelligente Teile der Infrastruktur wie z.B. Ampeln und Schilder, auch Road Side Units (RSU) genannt, sind mögliche Kommunikationspartner. Ein intelligentes Straßenschild könnte das geltende Geschwindigkeitslimit nicht nur visuell darstellen, sondern es auch elektronisch übermitteln. Dies spart aufwändige Bildbearbeitungsprozesse im Auto. Ebenso könnte eine kommunikative Ampel den Verkehrsteilnehmern ihren Status über Funktechniken mitteilen. Probleme bei der Erkennung des Lichtsignals, die z.B. durch starkes Gegenlicht auftreten, könnten so elegant gelöst werden.

Sicherheitsrelevante Innovationen können im Automobilbereich logischerweise nicht sofort im öffentlichen Straßenverkehr getestet werden. Auch Prototypen kommen aufgrund hoher Herstellungskosten nicht sofort zum Einsatz. Daher bietet es sich bei einigen Techniken an, diese zuerst in Modellen zu testen.

---

## 1.1 Problemstellung

---

Versuche an Modellen werden häufig an Universitäten durchgeführt. Solche Versuche vermitteln Studenten grundlegende Arbeitsweisen bei der Entwicklung neuer Technologien. Am Fachgebiet Echtzeitsysteme der TU Darmstadt sind bereits seit einigen Jahren autonome Modellautos im Rahmen eines Projektseminars im Einsatz. An ihnen lernen Studenten aus geeigneten Fachrichtungen vielfältige Probleme des autonomen Fahrens kennen. Die angesprochenen Modellautos besitzen in ihrer aktuellen Revision durch ihre WLAN-Adapter zwar zeitgemäße und leistungsfähige Kommunikationshardware, welche ihr volles Potential aufgrund mangelnder Softwareunterstützung bisher nicht entfalten kann. Um mit diesen Autos auch aktuelle Aufgabenstellungen des autonomen Fahrens lösen zu können, muss eine stabile Softwarebasis zur Kommunikation zwischen den Autos entwickelt werden.

Frühere Revisionen der Autos verfügten über andere Kommunikationstechniken wie z.B. Bluetooth oder 868 MHz-Funkmodule der Firma Amber Wireless. Deren Verwendung brachte jedoch einige Probleme mit sich. Die Funkmodule waren oftmals nicht

---

---

sehr leistungsfähig. Sobald man begann mit mehreren Autos Nachrichten zu senden, kam es häufig zu Datenverlusten. Bei der Kommunikation via Bluetooth waren der häufig notwendige manuelle Verbindungsaufbau sowie die Beschränkung auf Unicast-Verbindungen Schwachstellen. Vorteil der Bluetooth-Lösung war die auf geringe Distanz sichere Verbindung. Vorteil der Funkmodul-Lösung war die Möglichkeit der direkten Nutzung von Multicast und Broadcast, was das Versenden von Nachrichten an mehrere Ziele ermöglicht.

Einige der aufgetretenen Probleme waren sicherlich auf eine nicht vollends ausgereifte Implementierung zurückzuführen.

---

## 1.2 Ziele

Ziel dieser Arbeit ist es, einen Kommunikationsansatz für die Modellautos des Projektseminars zu entwickeln. Aus den Erfahrungen mit den alten Ansätzen, lassen sich einige wünschenswerte Eigenschaften des neuen Kommunikationsansatzes ableiten:

1. Einfacher, automatischer Verbindungsaufbau zwischen den Klienten
2. Ausreichende schnelle Übertragungen
3. Unterstützung von Unicast, Multicast und Broadcast
4. Verwendung der bereits vorhandenen WLAN-Hardware.

---

## 1.3 Aufbau der Arbeit

In Abschnitt 2 sollen Grundlagen zum Verständnis der Thematik geschildert werden. Grundlagen zu drahtlosen Kommunikationstechniken sollen ebenso erläutert werden, wie die Problematiken, die sich bei der Vernetzung von fahrenden Autos ergeben, welche bei stationären Rechnern nicht vorkommen. Insbesondere soll dabei auf Routingprotokolle eingegangen werden, welche sich für Ad-hoc-Netze eignen. Schlussendlich wird das auf den Modellautos genutzte Meta-Betriebssystem ROS vorgestellt werden.

Abschnitt 3 beschäftigt sich mit dem implementierten Lösungsansatz. Zuerst werden die Hardwarevoraussetzungen der Modellautos geschildert. Anschließend wird das Paket *adhoc\_communication* vorgestellt, welches die Basis dieser Arbeit bildet. Insbesondere soll gezeigt werden, wie die Daten durch das Paket verarbeitet werden. Anhand dieser Erkenntnisse wird aufgezeigt, an welchen Punkten angesetzt werden muss, um das Paket einfacher nutzbar zu machen. Schlussendlich wird die konkrete Implementierung vorgestellt, und deren Nutzen anschaulich aufgezeigt.

In der Evaluation (Abschnitt 4) wird der Kommunikationsansatz getestet. Es soll herausgefunden werden, wie leistungsfähig das Gesamtsystem ist und ob alle Anforderungen zur Nutzung im Rahmen des Projektseminars erfüllt werden. Es sollen jedoch auch Beschränkungen des entwickelten Ansatzes herausgefunden werden.

Im abschließenden Kapitel (Abschnitt 5) werden zunächst zwei verwandte Arbeiten vorgestellt, die sich mit ähnlichen Themen beschäftigen. Nach einem Fazit zu der vorliegenden Arbeit, sollen mögliche Anwendungszenarien im Projektseminar Echtzeit-systeme und mögliche Fortsetzungen der angestoßenen Entwicklung gezeigt werden.



---

## 2 Grundlagen

---

In diesem Kapitel soll auf einige Grundlagen der Netzwerkkommunikation eingegangen werden. Dabei stellen sich folgende Fragen:

- Welche Techniken eignen sich zur Kommunikation?
- Welche Netzwerktopologien sind zu verwenden?
- Welche Routing-Strategien sind bei sich bewegenden Modellautos sinnvoll?

Ebenso soll das Meta-Betriebssystem ROS vorgestellt und seine Grundmechaniken erläutert werden.

---

### 2.1 Kommunikation

---

Eine Herausforderung bei der Verbindung zwischen Autos ist die Tatsache, dass Autos nicht nur in der Lage sind sich zu bewegen, sondern dass Bewegung sogar ihre primäre Aufgabe ist. Die räumliche Mobilität unterscheidet sie dabei grundlegend von anderen klassischen Netzwerkknoten wie Desktop-Computern oder Smartphones. Die erste zu klärende Frage ist die nach der physikalischen Verbindungsart, wobei klar ist, dass drahtlose Kommunikation das Mittel der Wahl ist. Die zwei gängigsten drahtlosen Kommunikationstechniken im Endkunden-Bereich sind Wireless-LAN (WLAN) und Bluetooth. Beide Techniken haben einige Vor- und Nachteile auf die nun eingegangen werden soll.

---

#### 2.1.1 Bluetooth

---

Bluetooth ist eine drahtlose Kommunikationstechnik. Bluetoothgeräte funken im freien 2,4 GHz-Frequenzband. Das ursprüngliche Ziel von Bluetooth ist, Kabelverbindungen zwischen einzelnen Geräten zu ersetzen. Oftmals wird dabei ein kleines Zubehörgerät mit einem Hauptgerät verbunden, z.B. ein Eingabegerät mit einem stationären Rechner. Wichtig für solche kleinen Geräte ist ein möglichst geringer Energieverbrauch, da sie meist mit Akkus oder Batterien betrieben werden. Ein geringer Energieverbrauch äußert sich jedoch immer in geringeren Reichweiten und Übertragungsraten, weswegen diese bei der Entwicklung des Bluetooth Standards mit niedrigerer Priorität behandelt wurden.

Auch die neuesten Entwicklungen der Bluetooth-Spezifikationen liefern weitere Möglichkeiten zur Energieeinsparung. So wurde mit Bluetooth 4 ein Low-Energy-Modus eingeführt. Die Bluethooth-Version 4.2 erhielt den Beinamen SSmart und verbesserte den Low-Energy-Modus weiter. In der erst kürzlich verabschiedeten Bluetooth-Version 5 wurden, den Ankündigungen nach, die Reichweite auf bis zu 40 Meter und die Datenrate auf 2 MBit/s erhöht. Diese Entwicklung wurde in Hinblick auf das Internet-of-Things (IoT) getätigt. So soll es möglich sein kleinste Sensoren und Aktoren, die ihre Energie aus möglichst kleinen Akkus oder auch durch Energy-Harvesting beziehen, z.B. im Smart-Home-Bereich miteinander per Bluetooth zu vernetzen. Es ist ein erklärtes Ziel, dass bis zum Jahr 2020 ein Drittel der IoT-Geräte eine Bluetooth-Verbindung zur Vernetzung nutzen.

Bluetooth-Geräte formen bei der Verbindung immer ein sogenanntes Piconetz. In einem solchen gibt es immer genau einen Master. Alle anderen Geräte werden Slaves genannt. Jeder Slave hält eine Punkt-zu-Punkt-Verbindung mit seinem jeweiligen Master (Abbildung 2.1a). Es besteht jedoch auch die Möglichkeit mehrere Piconetze zu einem sogenannten Scatternetz zu verbinden. Dies wird dadurch ermöglicht, dass ein Gerät sowohl Slave in einem Netz, als auch Master eines neuen Netzes sein kann. Zu Scatternetzen gibt es jedoch bisher keine einheitlichen Spezifikationen, was die Implementierung eines solchen schwierig gestaltet.

### 2.1.2 Wireless-LAN

Wird von einem Wireless-LAN gesprochen, wird meist die Verwendung einer drahtlosen Übertragungstechnik nach einer Norm aus der Normengruppe IEEE 802.11 gemeint. Je nach Spezifikation wird wie bei Bluetooth das 2,4 GHz-Band oder in neueren Varianten auch das 5,0 GHz-Band genutzt. WLAN hat sich vor allem bei stationären oder komplexeren Geräten oder Geräten mit ausreichend großem Akku durchgesetzt, sodass der im Vergleich zu Bluetooth in der Regel höhere Energiebedarf nicht gravierend ist. Auch wenn hohe Übertragungsraten erforderlich sind, ist eine WLAN-Verbindungen der Verwendung von Bluetooth vorzuziehen. Die maximal mögliche Datenrate schwankt je nach Spezifikation sehr stark. Entscheidende Parameter sind hierbei die genutzte Kanalbreite, das Frequenzband und die Anzahl der genutzten Datenströme. So ist es theoretisch möglich bei Nutzung von zwei Datenströmen die doppelte Datenrate im Vergleich zur Nutzung eines einzelnen Datenstroms zu erzielen. Die ursprüngliche 802.11-Spezifikation bot eine Bruttodatenrate von 2 MBit/s. Das aktuell theoretisch mögliche Maximum bietet die 802.11ac Spezifikation. Dazu nötig sind drei je 160 MHz breite Datenströme im 5,0 GHz-Band.

Tabelle 2.1 gibt einen Überblick über die Datenraten eines WLANs nach IEEE 802.11n/ac bei verschiedenen Kanalbreiten und Datenströmen. Es ist deutlich zu erkennen, dass die Netto-Datenrate in den meisten Fällen weniger als 50% der Bruttodatenrate erreicht.

Eine WLAN-Schnittstelle kann in zwei verschiedenen Modi betrieben werden:

Der **Infrastruktur-Modus** ist die am weitesten verbreitete Art ein WLAN zu betreiben. So aufgebaute Netze sind fast überall zu finden. Dabei verbinden sich die einzelnen Klienten des Netzwerks drahtlos mit einem sogenannten Access Point (AP), der nur als Zugangspunkt für die Endgeräte dient. So entsteht eine sternförmige Netzwerktopologie (Abbildung 2.1a). Die Kommunikation zwischen den einzelnen Klienten läuft dann indirekt über diesen AP. Vorteilhaft ist hier die einfache Implementierung für die Klienten, da sämtliche Routingaufgaben vom AP übernommen werden. Nachteilig ist jedoch, dass bei Ausfall der Verbindung zwischen einem Klienten und dem AP, der Klient nicht mehr erreichbar ist. Fällt der AP aus, bricht folglich das ganze Netz zusammen. Außerdem ist die maximale Flächenabdeckung des Netzes allein von der Reichweite des APs abhängig. Der hier entwickelte Kommunikationsansatz soll ohne vorhandene Infrastruktur funktionieren. Um unter dieser Voraussetzung ein solches Netzwerk zu implementieren, wäre es nötig ein Auto als Access Point zu nutzen.

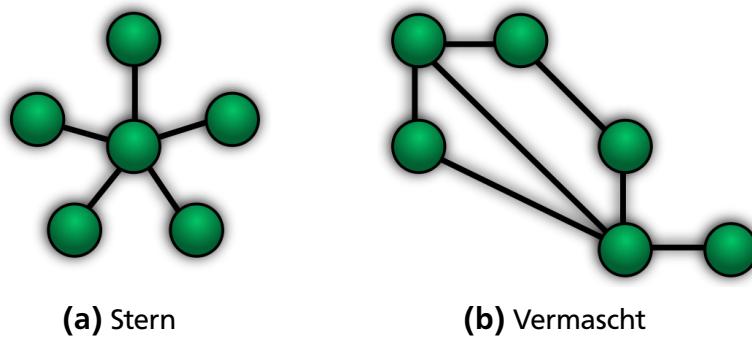
**Tabelle 2.1:** Brutto- vs. Nettodatenrate nach WLAN-Standard [avm16]

Standard	Datenströme	Kanalbandbreite	max. Netto-Datenrate	max. Bruttodatenrate
802.11ac	3	80 MHz	1300 Mbit/s	650 Mbit/s
	2		866 Mbit/s	300 Mbit/s
	1		433 Mbit/s	200 Mbit/s
	3	40 MHz	600 Mbit/s	220 Mbit/s
	2		400 Mbit/s	180 Mbit/s
	1		200 Mbit/s	90 Mbit/s
	3	20 MHz	260 Mbit/s	120 Mbit/s
	2		173 Mbit/s	80 Mbit/s
	1		86 Mbit/s	40 Mbit/s
802.11n	3	40 MHz	450 Mbit/s	200 Mbit/s
	2		300 Mbit/s	150 Mbit/s
	1		150 Mbit/s	75 Mbit/s
	3	20 MHz	195 Mbit/s	90 Mbit/s
	2		130 Mbit/s	60 Mbit/s
	1		65 Mbit/s	30 Mbit/s

Der **Ad-hoc-Modus** ist die zweite Möglichkeit der WLAN-Vernetzung. Das Wort Ad-hoc kommt aus dem lateinischen und bedeutet sinngemäß "für diesen Augenblick gemacht". Das beschreibt schon eine wichtige Eigenschaft eines solchen Netzes: Es ist sehr flexibel und bildet sich erst in dem Moment, in dem es gebraucht wird. Damit ein solches Netz sich jederzeit bilden kann, basiert es nicht auf vorhandener Infrastruktur. In einem Ad-hoc-Netzwerk bilden die einzelnen Knoten ein vermaschtes Netz, wobei jeder Knoten mit einem oder mehreren anderen Knoten verbunden ist. Zwischen welchen Klienten Verbindungen gebildet werden, hängt alleine von der Qualität der physischen Verbindung ab. So kann ein vermaschtes Netz sehr vielfältige Formen annehmen (Abbildung 2.1b). Fällt in einem solchen vermaschten Netz eine Verbindung aus (z.B. wenn Klienten sich voneinander entfernen), kann es immer noch möglich sein, die Verbindung über andere Knoten herzustellen. Durch diese flexible Netzwerktopologie, ist ein Ad-hoc-Netz optimal zum Verbinden sich bewegender Knoten.

## 2.2 Routing

Als Routing wird das Finden und Festlegen von Wegen für Datenpakete bezeichnet. In einem sternförmigen Netzwerk ist das Routing durch seine Einfachheit ein Vorteil, wird in einem Ad-hoc-Netz jedoch zu einem Nachteil. Da im Infrastrukturmodus alle Klienten direkt mit dem AP verbunden sind, muss dieser die ankommenden Daten nur direkt zu



**Abbildung 2.1:** Netzwerktopologien

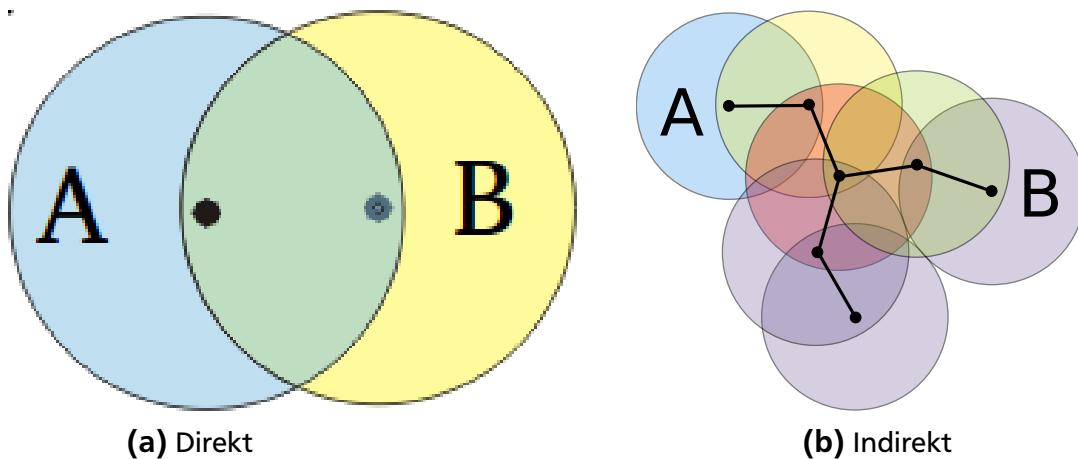
dem oder den Empfänger(n) weiterleiten. Noch einfacher ist es für die Klienten: Diese müssen ausgehende Pakete nur an den AP leiten.

Umso komplizierter kann Routing im Ad-hoc-Modus sein. Dabei ist grundsätzlich zwischen zwei Verbindungsarten zwischen Klienten zu unterscheiden: direkte und indirekte Verbindungen.

**Direkte Verbindungen** bestehen, wenn Start- und Zielknoten direkt benachbart sind. Die Kenntnis aller Nachbarn, ist eine Grundvoraussetzung für sämtliche Routingaufgaben in allen Protokollen. In einem solchen Fall wird das Routing trivial: Ein Paket muss nur an den richtigen Nachbarn geleitet werden. Das Finden längerer Pfade und Zwischenknoten entfällt logischerweise [JM96].

**Indirekte Verbindungen** liegen vor, wenn zwei Klienten zwar nicht benachbart sind, jedoch über andere Klienten eine Verbindung herstellen können. Dies wird dabei durch Weiterleitung von Datenpaketen über Zwischenknoten erreicht. So können auch größere Distanzen überbrückt werden [JM96].

In Abbildung 2.2a sind die Knoten A und B direkt miteinander verbunden, während sie in Abbildung 2.2b nur über einige Zwischenknoten miteinander kommunizieren können.



**Abbildung 2.2:** Verbindungsarten

---

---

Um der Flexibilität von Ad-hoc-Netzen gerecht zu werden, und stabiles Routing zu ermöglichen, kommen auf Ad-Hoc-Netze spezialisierte Routing-Algorithmen zum Einsatz. Diese werden in *reakтив* und *proaktiv* unterteilt.

**Reaktiv** werden Algorithmen genannt die nur aktiv werden, wenn ihnen zu einem Ziel-Knoten keine Route bekannt ist. Sie reagieren auf dieses fehlende Wissen und suchen eine Route.

**Proaktiv** werden Algorithmen genannt, die auch ohne konkrete Routing-Anfrage versuchen, eine ständig aktuelle Routingtabelle zu behalten.

Im Folgenden sollen zwei weit verbreitete Algorithmen vorgestellt werden: *Dynamic Source Routing* (DSR) und *Ad-hoc On-Demand Distance Vector* (AODV) Routing. Beide Algorithmen sind reaktiv, wie der Ön-DemandTeil (dt. *auf Anfrage*) des Namens vermuten lässt [Jyo11].

In beiden Algorithmen werden Routen in Tabellen, den sogenannten Routing-Tabellen, gespeichert. Diese verknüpfen einen Zielknoten mit den dazu gehörenden Routing-Informationen.

Zum leichteren Verständnis sei  $S$  der Startknoten in einem Netzwerk, welcher Daten an  $Z$ , den Zielknoten senden möchte.

---

### 2.2.1 AODV

---

Zum Auffinden von Nachbarknoten können Hello-Messages verwendet werden. [CBR04]. So weiß jeder Knoten, mit welchen anderen Knoten er direkt kommunizieren kann.

Das AODV Protokoll führt die drei **Nachrichtentypen** Route-Request (RREQ, dt. *Routen-Anfrage*), Route-Response (RREP, dt. *Routen-Antwort*. und *Route Error* (RERR, dt. *Routen-Fehler*) ein.

Ein RREQ-Paket ist 24 Byte groß und aus folgenden Feldern aufgebaut:

- 1 Byte: Message-Typ
- je 4 Byte: Start- und Zielknoten (IP-Adresse)
- je 4 Byte: Sequenznummern von Start- und Zielknoten
- 4 Byte: RREQ-ID
- 1 Byte: Hop-Zähler
- 5 bit: diverse Statusbits
- 11 bit: reserviert (= 0)

Das Typ-Feld identifiziert das Paket als RREQ und dient der Unterscheidung zu anderen Pakettypen. Der Hop-Zähler gibt an, wieviele Netzwerkknoten das Paket bereits passiert hat. Die Sequenznummern dienen dazu, gespeicherte Routen aktuell zu halten

und keine veralteten Informationen zu übernehmen [DPBR03]. Zusätzlich sind 11 bit ungenutzt, sie dienen nur dazu das Paket exakt 24 Byte groß werden zu lassen.

Einem RREP-Paket fehlt die RREQ-ID. Während einige Statusbits eines RREQ-Pakets fehlen, kommen andere dazu. Durch die fehlende 4-Byte-ID, ist eine RREP nur 20 Byte groß.

RERRs sollen den Ausfall einer Verbindung anzeigen, sodass Netzwerkknoten eventuelle Routingeinträge aktualisieren können.

Da AODV ein reaktives Protokoll ist, wird es erst aktiv, wenn  $S$  keine gültige Route zu  $Z$  kennt. Tritt dieser Fall ein, wird die zweiteilige **Routenfindung** angestoßen. Das Prinzip bestehend aus Route-Request und Route-Response sieht vor, dass der Startknoten, welcher eine Route sucht, eine RREQ erstellt und das Netzwerk diese per Broadcasts zum Zielknoten leitet. Der Zielknoten erstellt bei Empfang der RREQ eine RREP, und sendet sie im Unicast an den Startknoten zurück. Alternativ wird die RREP von einem Zwischenknoten erstellt, falls dieser eine gültige Route zu  $Z$  kennt. Da die Zwischenknoten beim Weiterleiten der RREQ sich die Route zum Startknoten gemerkt haben, können sie nun die RREP wieder effizient an diesen weiterleiten, und sich dabei wiederum die Route zum Zielknoten merken.

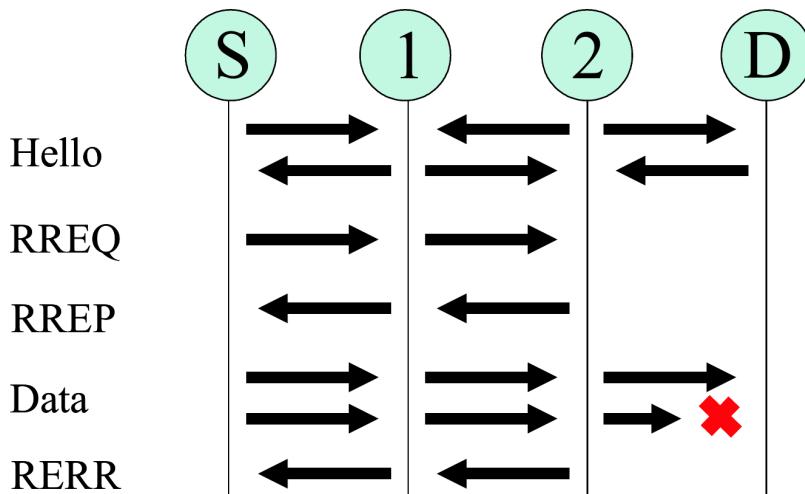


Abbildung 2.3: Routenfindung bei AODV [CBR04]

Abbildung 2.3 verdeutlicht den Ablauf der Routenfindung. Dabei stellt  $S$  den Startknoten,  $D$  den Zielknoten (engl. *Destination*) und 1 und 2 Zwischenknoten dar. Zuerst werden Hello-Messages versendet, um Nachbarn zu identifizieren.  $S$  startet nun die Routenfindung durch Senden einer RREQ. Gelangt diese zu Knoten 2, antwortet dieser mit einer RREP (Knoten 2 kennt als Nachbar von  $D$  bereits eine Route zu  $D$ ). Nun kann Knoten  $S$  solange Daten senden, bis eine Verbindung zu  $D$  abreißt. geschieht dies (verdeutlicht durch das rote Kreuz), sendet Knoten 2 eine RERR aus um Knoten 2 mitzuteilen, dass die aktuelle Route nicht mehr genutzt werden kann.

Die **Routingtabellen** enthalten bei AODV keine vollständigen Routen. In der Routingtabelle eines Knotens wird nur die Information gespeichert, an welchen Nachbar ein Paket geleitet werden muss, um einen Zielknoten zu erreichen. So muss jeder Knoten

---

der an der Weiterleitung eines Paketes beteiligt ist, eine aktive Route zum Zielknoten besitzen. Dies wird dadurch sichergestellt, dass Zwischenknoten bei der Weiterleitung von RREQs und RREPs die jeweiligen Tabelleneinträge aktualisieren.

---

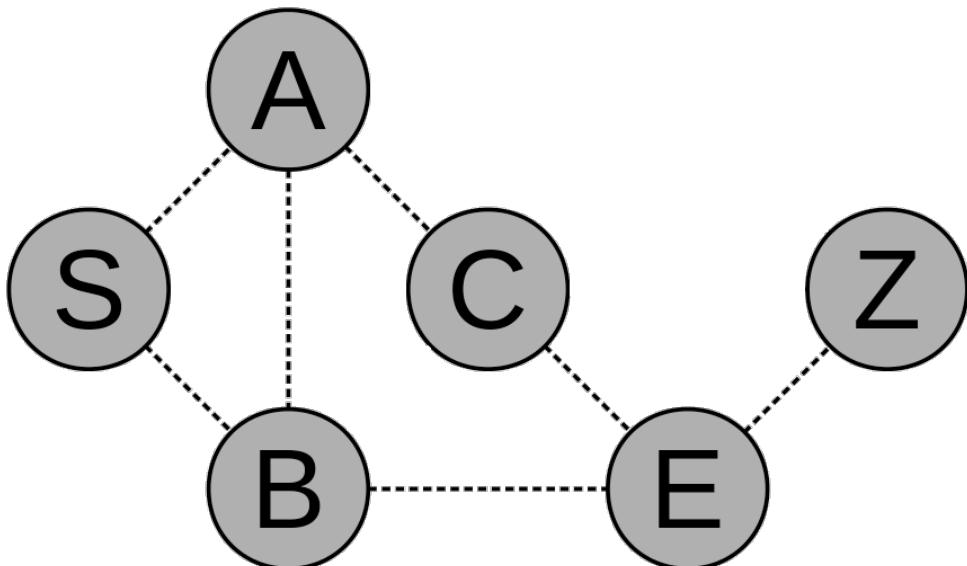
### 2.2.2 DSR

DSR hat viele Gemeinsamkeiten mit AODV. Der wesentliche Unterschied besteht in der Speicherung der Routen in den Routingtabellen der einzelnen Knoten. Während bei AODV nur der jeweils nächste Knoten, an den ein Paket weitergeleitet werden muss um sein Ziel Z zu erreichen, gespeichert wird, wird bei DSR die komplette Route inklusive aller Zwischenknoten gespeichert. Um dies zu ermöglichen wird in den RREQs und RREPs jeder Zwischenknoten gespeichert. Dadurch sind deren Pakete von dynamischer Größe, je nach Anzahl der Zwischenknoten. Ein weiterer Unterschied zu AODV ist, dass DSR nicht vom Senden von Hello-Messages oder ähnlichem abhängig ist [JHM07].

### 2.2.3 Beispiel

Um die Funktionsweise der Routingprotokolle verständlicher erklären zu können, soll die Routenfindung an einem vereinfachten Beispiel erklärt werden. Dieses Beispiel verwendet AODV. Ein RREQ-Paket ist aus den Feldern Start-Knoten, Ziel-Knoten, ID und Hop-Count aufgebaut und wird in eckigen Klammern repräsentiert: **[Start, Ziel, ID, Hop-Count]**. Dasselbe gilt für ein RREP-Paket. Alle anderen Komponenten werden zur besseren Übersicht weggelassen, da sie zum grundsätzlichen Verständnis nicht zwingend notwendig sind. Einträge der Routingtabellen werden in Runden Klammern nach dem Schema **(Ziel|nächster Knoten)** dargestellt.

Man betrachte das Netzwerk in Abbildung 2.4.



**Abbildung 2.4:** Beispielnetzwerk

Es sei  $S$  der Startknoten,  $Z$  der Zielknoten. Die Knoten  $A, B, C$  und  $E$  seien Zwischenknoten. Im Initialzustand kennt jeder Knoten die Verbindungen zu seinen Nachbarn

durch das Senden von Hello-Messages. Jeder Knoten besitzt eine leere Routingtabelle.  $S$  hat nun die Aufgabe Daten an  $Z$  zu schicken. Da es keine Route kennt, stößt es den Routenfindungsprozess an. Die Routenfindung besteht aus zwei Teilen: Route-Request, und Route-Response.

- **Route Request**

$S$  erstellt ein RREQ-Paket  $[S, Z, 1, 0]$ . Dieses sendet es nun an all seine Nachbarn ( $A, B$ ).

$A$  und  $B$  erhalten  $[S, Z, 1, 0]$

Die empfangenden Knoten haben nun drei Aufgaben:

- Das RREQ-Paket merken, bzw. mit gemerkten Paketen vergleichen.
- Erstellen der Routingtabelleneintrags zu  $S$  ( $S|S$ )
- Prüfen, ob in seiner Routingtabelle bereits eine Route zu  $Z$  vorhanden ist.

Da keiner der Knoten eine Route zu  $Z$  kennt, wird der Hop-Count inkrementiert und das RREQ-Paket  $[S, Z, 1, 1]$  wieder an alle Nachbarn gesendet.

$S, A, C$  und  $E$  erhalten  $[S, Z, 1, 1]$

$S$  kann das Paket logischerweise verwerfen, da es das Paket selbst erstellt hat.  $A$  kann das Paket verwerfen, da es sich eine frühere Version des Pakets mit geringererem Hop-Count gemerkt hat. Das Paket ist dabei durch die Angabe des Startknotens, und der von  $S$  vergebenen ID eindeutig identifizierbar.

$C$  kann nun einen Eintrag zur Routingtabelle hinzufügen ( $S|A$ ).

$E$  kann nun einen Eintrag zur Routingtabelle hinzufügen ( $S|B$ ).

$C$  und  $E$  inkrementieren den Hop-Count und broadcasten das Paket an alle Nachbarn.

Das Verwerfen von Paketen wird nun nicht mehr erwähnt.

Im finalen Schritt erhält  $Z$  das RREQ-Paket von  $E$   $[S, Z, 1, 2]$ .  $Z$  kann den Routeneintrag ( $S|E$ ) anlegen.

- **Route Reply**

Da  $Z$  der im Paket gesuchte Zielknoten ist, kann er nun ein RREP-Paket  $[S, Z, 1, 0]$  erstellen. Gemäß des eben angelegten Eintrages in der Routing-Tabelle, leitet  $Z$  das Paket an  $E$  weiter.

$E$  kann einen Eintrag ( $Z|Z$ ) erstellen, und leitet das Paket wiederum an  $B$  weiter.

$B$  kann einen Eintrag ( $Z|E$ ) erstellen, und leitet das Paket wiederum an  $S$  weiter.

$S$  kann einen Eintrag ( $Z|B$ ) erstellen. Da  $S$  erkennt, dass diese Antwort für ihn selbst erstellt wurde, erfolgt keine Weiterleitung. Die Route zu  $Z$  ist gefunden.

Hätte einer der Zwischenknoten bereits eine Route zu  $Z$  gekannt, so hätte dieser die RREP erstellt. Nach Abschluss der Routenfindung kennt jeder Knoten, der auf der Route zwischen  $S$  und  $Z$  liegt, auch eine Route zu  $Z$  bzw.  $S$ . Dieses Prinzip mit Route Requests und Responses gilt auch bei DSR.

## 2.3 ROS

ROS steht für Robot Operating System und ist ein Metabetriebssystem für Roboter [ROS14]. ROS-Prozesse, die sogenannten Nodes, bilden ein loses Peer-to-Peer Netzwerk mit dem ROS-Master. Diese Verbindung von ROS-Master und -Nodes wird als ROS-Graph bezeichnet. Ein Master kann dabei beliebig viele Nodes anbinden, ein Node jedoch nur mit einem Master verbunden sein. Auf einem Roboter können dabei mehrere ROS-Nodes laufen. Um vollständig zu funktionieren, muss ein ROS-Core gestartet werden. Dieser besteht aus einem ROS-Master, einem ROS-Parameterserver (vgl. Abschnitt 2.3.4) und einer Instanz von rosout, welche Nachrichten protokolliert.

Der ROS-Graph ermöglicht nicht nur die Interprozesskommunikation von lokalen, auf einem Rechner laufenden Nodes, sondern auch über ein Netzwerk. So kann ein Node unter Angabe von IP-Adresse und Port des entfernten Masters auch mit diesem kommunizieren.

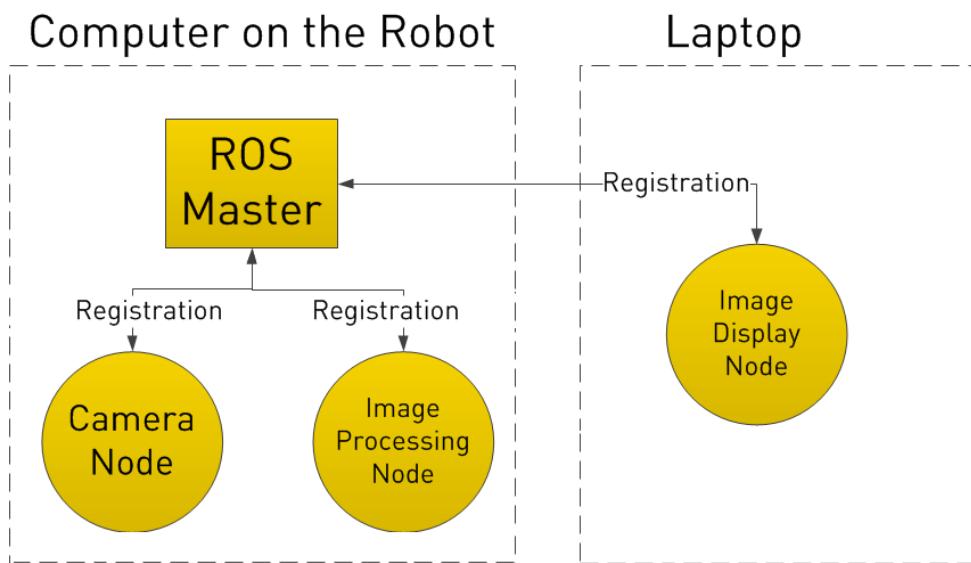


Abbildung 2.5: Illustrierung des ROS-Graph [Rob15]

Abbildung 2.5 zeigt eine Illustration eines ROS-Graphs. Dabei läuft der ROS-Master mit einem Kamera-Node und einem Bildverarbeitungs-Node auf dem Roboter, während ein anderer Knoten zur Bildanzeige auf einem Laptop läuft.

Einzelne Software-Module für ROS werden in Paketen organisiert. Dabei ist keine Mindestmenge an Funktionalität vorgeschrieben, sodass Pakete sehr unterschiedliche Größen annehmen können. Ein Paket soll lediglich Funktionalität in sinnvoll abgegrenzte und leicht wiederverwendbare Module kapseln. Dabei sollte man immer versuchen, die richtige Balance zwischen viel Funktionalität und feinstufiger Granulierung zu finden. Die einzigen notwendigen Bedingungen zur Identifikation eines Paketes als solches sind, dass eine Paketdeskriptor-Datei (`package.xml`) mit Informationen über das Paket im Stammverzeichnis des Pakets liegt, und das Stammverzeichnis ein Unterordner des ROS-Paket-Pfades ist, welcher über eine Umgebungsvariable gesetzt wird (`ROS_PACKAGE_PATH`).

---

### **Quellcode 2.1: Beispielhafter Aufbau der Message-Datei *Student.msg***

```
string Name
string Vorname
uint32 Matrikelnummer
bool immatrikuliert
```

---

#### **2.3.1 Messages**

Eine ROS-Message ist vergleichbar mit einer einfachen Klasse in einer objektorientierten Programmiersprache (z.B. Java), welche nur zum Speichern von Daten dient, sprich keine Methoden implementiert.

Um das Implementieren einer Message zu vereinfachen, und um sowohl C++ als auch Python zu unterstützen, werden ROS-Messages nicht direkt in einer dieser Sprachen geschrieben, sondern aus einfach zu implementierenden Message-Dateien (\*.msg) in die gewünschte Programmiersprache übersetzt.

ROS-Message-Dateien werden durch Listen mit zeilengetrennten Datenfeld- und Konstantendefinitionen dargestellt. Unterstützt werden dabei die primitiven Datentypen Wahrheitswert (engl. Boolean), Ganzzahlen (Vorzeichenlos/-behaftet 8/16/32/64-Bit), Gleitkommazahlen (32/64-Bit), Zeichenkette (engl. String), die ROS-spezifischen Datentypen Zeitpunkt (engl. time) und Zeitdauer (engl. duration), sowie Arrays der primitiven DatenTypen mit fixer oder variabler Länge. Konstante Werte werden dabei per Gleichzeichen gesetzt. Zusätzlich kann jeder beliebige andere Messagetypr der in ROS implementiert wurde verwendet werden, was verschachtelte Messages ermöglicht. Ein spezieller Messagetypr sind die Header, welche für jede erstellte Message einzigartige IDs und Zeitstempel enthalten.

Ein einfaches Beispiel wäre eine Student-Datenstruktur, welche die Datenfelder **Name**, **Vorname**, **Matrikelnummer** und **immatrikuliert** besitzt (Quellcode 2.1). **Name** und **Vorname** werden durch Zeichenketten implementiert, die **Matrikelnummer** ist ganzzahlig (engl. Integer) und die Eigenschaft **immatrikuliert** ist ein Wahrheitswert.

Der daraus generierte Code ist um ein Vielfaches umfangreicher und komplexer und ermöglicht weitere Verarbeitungsweisen von Messages. Zusätzlich zu selbstdefinierten Nachrichtentypen gibt es das ROS-Paket der Standard-Messages (`std_msgs`) mit einigen vordefinierten Standard-Messages, die z.b. jeweils einen primitiven Datentyp in einer Message verkapseln.

---

#### **2.3.2 Publish/Subscribe**

Um die Kommunikation zwischen einzelnen Nodes zu ermöglichen, nutzt ROS das Publish/Subscribe-Verfahren. Dabei stellt der ROS-Master namentlich definierte Schnittstellen zum Austauschen von Messages bereit. Die Namen dieser Schnittstellen nennt man Topics. Möchte also ein Node die ihm zur Verfügung stehenden Informationen anderen Nodes zugänglich machen, so sendet er diese unter einem selbstgewählten Topic

---

### Quellcode 2.3: Ein einfacher Subscriber für String-Messages [ROS16b]

---

```
1 void callback(const std_msgs::String::ConstPtr& msg){  
2     ROS_INFO("I heard: [%s]", msg->data.c_str());  
3 }  
4  
5 int main(int argc, char **argv){  
6     ros::init(argc, argv, "listener");  
7     ros::NodeHandle n;  
8     ros::Subscriber sub = n.subscribe("topic", 1000,  
9                                     callback); }
```

---

an den Master. Diesen Prozess nennt man veröffentlichen (engl. publish). Ein anderer Node, der diese Informationen empfangen möchte, muss dies dem Master mitteilen und ihm eine Callbackfunktion bekanntmachen, die beim Eintreffen neuer Nachrichten ausgelöst werden soll. Dies nennt man abonnieren (engl. subscribe). Wird von einem Publisher eine Nachricht veröffentlicht, löst der Master bei allen Subscribers die entsprechenden Callbackfunktionen aus, welche die Nachricht dann weiter verarbeiten können.

---

### Quellcode 2.2: Ein einfacher Publisher für String-Messages [ROS16b]

---

```
1 ros::NodeHandle n;  
2 ros::Publisher pub = n.advertise<std_msgs::String>("topic", 1000);  
3 std_msgs::String msg;  
4 msg.data = "Hello_World!";  
5 pub.publish(msg);
```

---

In Quellcode 2.2 sieht man einen einfachen Publisher. In Zeile 2 wird der Publisher dem Master bekannt gemacht. Dabei werden das Topic auf "topic" und der darüber ausgetauschte Message-Typ als String-Message festgelegt. Das data-Feld der Message wird befüllt und mit einem einfachen Aufruf der *publish*-Funktion auf dem Publisher-Objekt wird die Nachricht veröffentlicht.

In Quellcode 2.3 wird zuerst die Callbackfunktion definiert, die in diesem Beispiel keinerlei Funktionalität aufweist. In Zeile 5 wird auf das Topic "topic" abonniert und die Callback-Funktion registriert. Beide Beispiele sind unvollständig und sollen nur als Skizze dem besseren Verständnis dienen.

---

#### 2.3.3 Services

---

Auch wenn beim Publish/Subscribe-Prozess letztlich eine Funktion in einem anderen Node aufgerufen wird, ist es trotzdem nicht angemessen diesen explizit zum Aufrufen einer Funktion zu nutzen. Dafür besser geeignet ist die von ROS bereitgestellte Service-Funktionalität. Ein Service besteht dabei nicht nur aus der gesendeten Anfrage, sondern

---

auch aus einer Antwort (Request und Response). Diese beiden Elemente sind durch ROS-Messages realisiert.

---

#### **Quellcode 2.4: Servicedatei AddTwoInts.srv [ROS16a]**

```
1 int8 a
2 int8 b
3 ---
4 int16 sum
```

---

Auch die Servicedefinition ähnelt einer Messagedefinition sehr stark. Services werden in \*.srv Dateien implementiert (Quellcode 2.4), aus welchen dann, ähnlich wie bei Messages, der Programmcode generiert wird. Bei der Servicedefinition werden Request und Response durch drei Bindestriche (---) voneinander separiert. Für Servicedefinitionen gelten die selben Regeln bezüglich der erlaubten Datentypen wie für Messages.

---

#### **Quellcode 2.5: Callbackfunktion und Service-Server [ROS15]**

```
1 bool add(Add2Ints::Request &req, Add2Ints::Response &res){
2   res.sum = req.a + req.b;
3   return true;
4 }
5 int main(int argc, char **argv){
6   ros::init(argc, argv, "addIntsServer");
7   ros::NodeHandle n;
8   ros::ServiceServer service=n.advertiseService("addInts", add);
9 }
```

---

Damit ein Service aufgerufen werden kann, muss ein Service-Server instanziert werden, und die Callbackfunktion diesem bekannt gemacht werden (advertise). In Quellcode 2.5 Zeile 1 wird die Callbackfunktion implementiert. Das Bekanntmachen erfolgt in Zeile 8.

In Quellcode 2.6 wird der Service aufgerufen. Dazu müssen ein Service-Client (Zeile 2) und ein Service (Zeile 3) instanziert werden. Nachdem die Datenfelder des Service-Requests mit Daten gefüllt werden, wird in Zeile 5 der Service mit dem Serviceclient aufgerufen.

---

#### **2.3.4 Parameter-Server**

Eine weitere Funktionalität des ROS-Cores ist der Parameter-Server. Der Parameter-Server speichert Werte als Parameter, auf welche die Nodes zugreifen können. So ist es möglich, die Parameter innerhalb eines Nodes abzufragen, und verschiedene Funktionalitäten in Abhängigkeit dieser Parameter zu verwirklichen. Dies erspart häufiges neucompilieren des Nodes bei nur geringen Änderungen.

---

### Quellcode 2.6: Serviceaufruf [ROS15]

---

```
1 ros::NodeHandle n;
2 ros::ServiceClient client=n.serviceClient<Add2Ints>("addInts");
3
4 beginner_tutorials::AddTwoInts srv;
5 srv.request.a = 5;  srv.request.b = 4;
6 if (client.call(srv)){
7     ROS_INFO("Sum: %ld", (long int)srv.response.sum);
8 } else{
9     ROS_ERROR("Failed to call service");
}
```

---

Parameterwerte können aus einem Node heraus (*nodeHandle.setParam( <Parameter>, <Wert>)*), über die Kommandozeile (*rosparam set <parameter> <wert>*) oder über das Ausführen einer sogenannten Launch-Datei gesetzt werden. Eine solche ist im XML-Format aufgebaut und ermöglicht neben dem Setzen von Parametern unter anderem das Starten von mehreren Nodes und das Zusammenfassen von Nodes in Gruppen (bei Bedarf mit eigenem Namespace). Ist noch kein ROS-Core gestartet, wenn eine Launch-Datei ausgeführt wird, wird auch dieser automatisch gestartet.

---

### Quellcode 2.7: Launch-Datei zum Starten des Senders (Abschnitt 4)

---

```
<launch>
  <group ns="sender">
    <node pkg="adhoc_tests" name="sender" type="sender">
      <param name="loop" value="1" />
      <param name="strLen" value="1" />
      <param name="dst_robot" value="pses-car7" />
      <param name="mode" value="0" />
    </node>
  </group>
</launch>
```

---

Quellcode 2.7 zeigt, wie der Sender-Node gestartet werden kann, sodass einige Parameter automatisch gesetzt werden.

---

## 2.4 Serialisierung

---

Um Datenobjekte über das Netzwerk austauschen zu können, müssen diese in einer sequenziellen Form vorliegen. Da Objekte in der Regel in Form eines Graphen vorliegen, müssen diese zuerst in eine sequenzielle Form umgewandelt werden. Diese Umwandlung nennt man **Serialisierung**. Dabei werden sowohl alle Datenfelder des Objekts, als auch alle referenzierten Objekte serialisiert.

Nach der Netzwerkübertragung muss die sequenzielle Darstellung wieder deserialisiert werden. Dieser Prozess wandelt die über das Netzwerk empfangenen Daten wieder in eine objektorientierte Datenstruktur um. Dazu muss der deserialisierenden Funktion jedoch der ursprüngliche Objekttyp bekannt sein. Die Deserialisierung soll an einem Beispiel verdeutlicht werden.

**Quellcode 2.8: Serialisierte Student-Message in Hexadezimaler Schreibweise**

03	00	00	00		52	61	75		03	00	00	00		4b	61	69		7b	00	00	00		01	
length					Name					length					Vorname					matNr				Immatr.

In Quellcode 2.8 zeigt Zeile 1 die Daten, zur besseren Übersicht in Abschnitte unterteilt, und Zeile 2 die jeweiligen Bedeutungen der Abschnitte. Unter Kenntnis der Struktur einer Student-Message, kann die oben gezeigte Nachricht deserialisiert werden. Der deserialisierenden Funktion ist also bekannt, dass eine Student-Message aus den zwei Strings “Name” und “Vorname”, dem Integerwert “Matrikelnummer” sowie dem Wahrheitswert “immatrikuliert” besteht. So ist bekannt, dass das erste Feld die Länge des Name-Strings enthält. Die Längenangabe ist 4 Byte lang. Aus [0x03,0x00,0x00,0x00] wird so die Länge 3 Byte bestimmt. Unter Kenntnis dieser Länge, werden die darauffolgenden 3 Byte als “Name” interpretiert: [0x52,0x61,0x75] entsprechen “Rau” in ASCII-Kodierung. Ebenso wird für das Vorname-Feld verfahren: Die 3 Byte [0x4b,0x61,0x69] entsprechen “Kai” in ASCII-Kodierung. Das nächste Feld ist die Matrikelnummer, repräsentiert durch einen 32-Bit Integer: 0x7b entspricht 123 in dezimaler Schreibweise. Das darauf folgende Byte wird als der boolesche Wert immatrikuliert interpretiert: [0x01] entspricht dabei *true*.

So kann aus der seriellen Kette von Bytes die komplette Struktur inklusive der Daten der ursprünglichen Student-Message wiederhergestellt werden (Quellcode 2.1).

---

## 3 Umsetzung

---

In diesem Kapitel soll auf die Implementierung des Kommunikationsansatzes eingegangen werden. Es sollen die Modellautos vorgestellt werden, auf denen dieser Ansatz zum Einsatz kommen soll. Die Software, die zur Umsetzung verwendet wurde, soll mit ihren Stärken und Schwächen analysiert werden, um herauszufinden an welchen Punkten man sie durch Verbesserungen leichter zugänglich machen kann. Schlussendlich sollen diese Verbesserungen gezeigt und ihr Nutzen klar dargestellt werden.

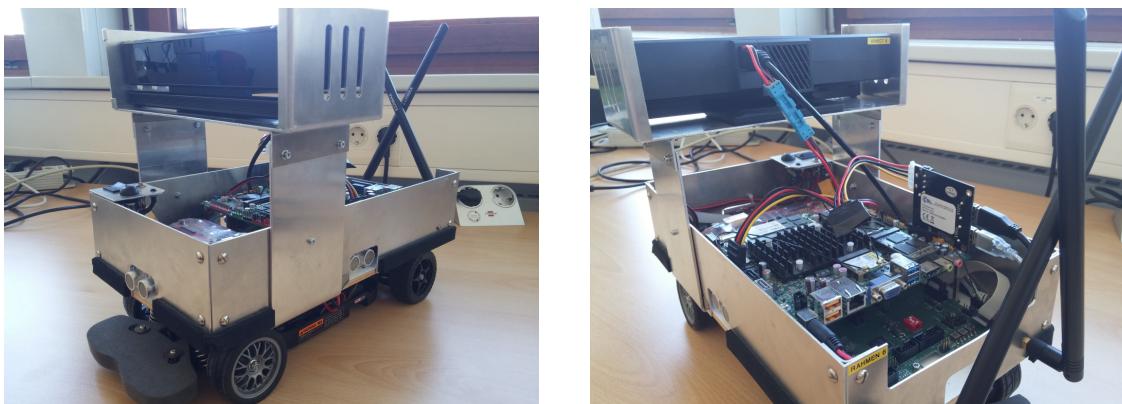
---

### 3.1 Modellautos

---

Die Modellautos die zur Umsetzung dieser Arbeit genutzt werden, sind Eigenentwicklungen des Fachgebiets Echtzeitsysteme für das Projektseminar Echtzeitsysteme an der TU Darmstadt. An ihnen sollen Studenten Prinzipien und Arbeitsweisen von autonomen Robotern lernen. Die Hardware der Autos ist in drei Bestandteile unterteilt: Die Fahrmechanik, den Hauptrechner und den Mikrocontroller. Die Fahrmechanik und der Mikrocontroller sind in dieser Arbeit von eher geringer Bedeutung, da das Fahren der Autos keine Rolle spielt, weshalb ich hier nicht weiter darauf eingehen werde. Weite Teile der Kommunikation verhalten sich unabhängig zu der Tatsache, dass die Implementierung nicht auf stationären Computern stattfindet, sondern auf sich bewegenden Autos.

Abbildung 3.1 zeigt ein Exemplar der Modellautos von vorne und von hinten. Auf den ersten Blick zu erkennen sind der große Aufbau mit der Kinect-Kamera und die zwei WLAN-Antennen.



**Abbildung 3.1:** Modellauto

Betrachtet man die Autos im Detail (Abbildung 3.2) erkennt man die WLAN-Erweiterungskarte (gelb), die M.2-SSD als Hauptspeicher (rot) und den LiFe-Akku für den Rechner (grün).

---

#### 3.1.1 Rechnerhardware

---

Der Computer ist aus handelsüblichen Endkunden-Komponenten zusammengestellt. Herzstück ist das auf die Grundplatte des Autos verschraubte Mini-ITX Board Mitac

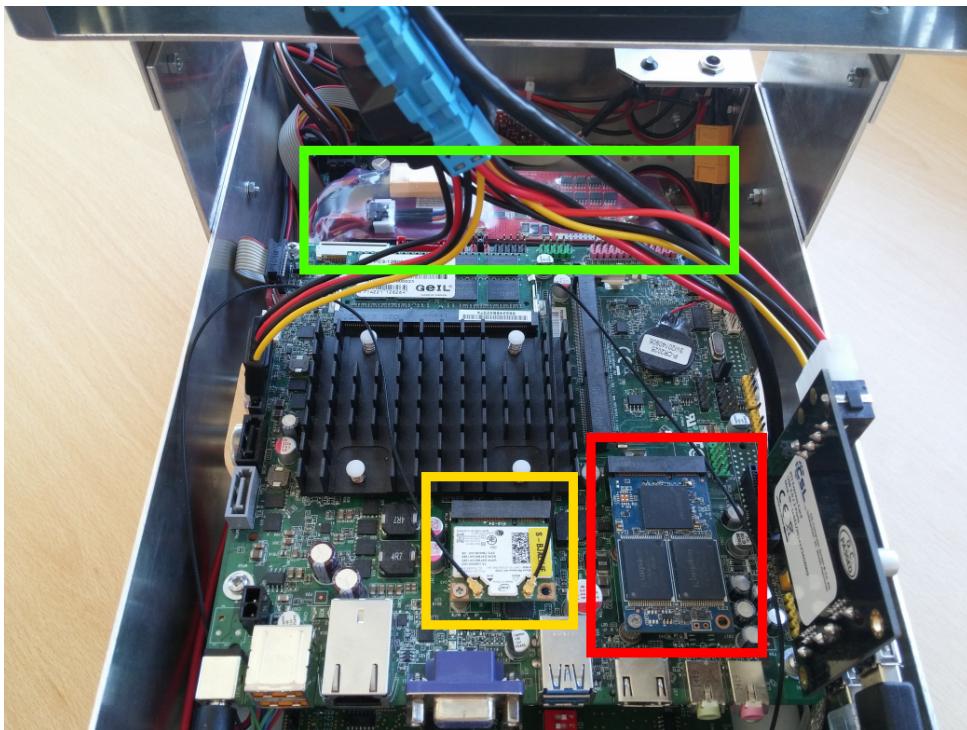


Abbildung 3.2: Rechenhardware im Detail

PD10BI MT Think Mini-ITX DN2800. Das Board ist bestückt mit einem Intel Celeron Prozessor mit 4 Kernen, welcher passiv gekühlt wird und eine Grafikeinheit beinhaltet. Die passive Kühlung ist für die bisherigen Aufgabengebiete ausreichend. Bei hoher Belastung des Prozessors kann es jedoch zu einem Senken der Taktfrequenz der Hauptrecheneinheit kommen, da die entstehende Wärme nicht mehr ausreichend abgeleitet werden kann. Dazu wäre ein größerer Kühler oder ein Ventilator nötig, was jedoch Gewicht bzw. Strombedarf des Autos erhöhen würde. Als Hauptspeicher dient ein einzelnes DDR3-1600 Modul mit 8GB Kapazität. Als Massenspeicher kommt eine 60GB SSD zum Einsatz, welche Platzsparend im M.2 Format per PCI an das Mainboard angebunden ist. Um Kabellose Netzwerkverbindungen zu ermöglichen, wurde die Hauptplatine um eine Intel Dual-Band Wireless AC-7260 Netzwerkkarte erweitert. Diese beherrscht sowohl WLAN nach Standards 802.11 ac/a/b/g/n als auch Bluetooth [Int13]. Zwei externe, an das Chassis des Modellautos geschraubte Stabantennen verbessern den Empfang im Vergleich zum Einsatz von Drahtantennen. VGA und HDMI-Anschluss sowie USB-Anschlüsse ermöglichen die einfache Verbindung mit Monitor und Eingabegeräten. Damit das Auto auch mobil ist und auch auf der Fahrt vollständig funktioniert, wurde zusätzlich zur Anschlussbuchse für ein externen Netzteil ein 3200 mAh fassender Li-Fe Akku verbaut. Um komplexe Bilderkennungsaufgaben lösen zu können, wurden die Autos mit einer Microsoft Kinect-Kamera in Version 2 ausgestattet. Da diese Probleme mit den direkt an den Prozessor angebundenen USB-Ports des Mainboards hatte, wurde das Mainboard um eine USB 3.0 PCIe Karte erweitert.

---

### **3.1.2 Chassis, Fahrmechanik, Mikrocontroller und Sensoren**

---

Das Chassis des Autos besteht aus der oben schon genannten hölzernen Grundplatte, sowie seitlich angeschraubten Aluminiumwänden. Durch Aussparungen an den beiden seitlichen und der vorderen Aluminiumwand, messen Ultraschallsensoren den Abstand zu Hindernissen. Unter die Grundplatte ist das zweiachsige Fahrgestell, welches aus einem Modellbausatz der Firma Tamiya entnommen wurde, montiert. Die Vorderachse wird zum Lenken verwendet, die hinten gelegene Antriebsachse ist starr. Das Fahrgestell bestehend aus Antriebsmotor und Lenkungsmotor wird von einem separaten Akku angetrieben. Um einen eventuellen Frontalaufprall abzufedern, sind alle Autos mit breiten Schaumstoffstoßdämpfern ausgestattet. Des Weiteren sind ein Hall-Sensor und ein Inkrementaldrehgeber verbaut, um die zurückgelegte Wegstrecke bestimmen zu können. Um die Orientierung des Autos bestimmen zu können, ist ein Gyrosensor verbaut. Die Schnittstelle zwischen der Fahrmechanik mit den Sensoren und dem Mainboard ist ein 16-Bit Fujitsu Mikrocontroller der Serie MB96300. Über ein USB-Kabel, ist dieser seriell via UART mit dem Mainboard verbunden. Über diese serielle Schnittstelle werden Messdaten der Sensoren vom Mikrocontroller an das Mainboard, sowie Steuerungsbefehle in umgekehrter Richtung übertragen.

---

### **3.1.3 Software**

---

Als Betriebssystem kommt Lubuntu zum Einsatz, welches eine leichtgewichtige Version von Ubuntu 14.04 ist. Diese Version ist eine LTS (Long-Term-Support) Version von Ubuntu, die bis 2019 unterstützt wird. Auf den Autos zum Einsatz kommt die ROS-Distribution Indigo Igloo. Auch wenn seit Release von ROS Indigo schon zwei neue Distributionen veröffentlicht wurden, wird weiterhin Indigo genutzt. Indigo Igloo wird noch bis 2019 unterstützt, und viele schon existierende ROS-Packages sind mit Indigo Igloo entwickelt worden und darauf abgestimmt. Da ROS-Indigo auf die Ubuntu 14.04 Version aufbaut, ist auch das Betriebssystem noch auf diesem Stand.

---

## **3.2 adhoc\_communication-Paket**

---

Recherchen zu Ad-hoc-Netzen unter ROS führten zu dem Paket *adhoc\_communication*. Dieses nutzt die WLAN-Schnittstelle um ein Ad-hoc-Netz zu bilden und Pakete zwischen mehreren ROS-Cores auszutauschen. Über das verwendete Routingprotokoll finden sich unterschiedliche Angaben. Ein Kommentar im Quellcode weist auf DSR hin, während die Publikation auf AODV hinweist [ANB14]. Vermutlich liegt die Wahrheit irgendwo dazwischen. Die Konsolenausgabe im Betrieb deutet darauf hin, dass komplett Routen gespeichert werden, was für DSR spricht. Das Versenden von Hello-Messages und das damit verbundene Auffinden von Nachbarn spricht jedoch für AODV. Da die Unterschiede zwischen den beiden Protokollen jedoch gering sind, und die Routenfindung große Ähnlichkeit hat, ist die ungenaue Angabe zwar ärgerlich, jedoch nicht von großer Bedeutung (siehe Abschnitt 2.2.1 und Abschnitt 2.2.2).

Auch unterstützt das Paket sowohl Unicast-Verbindungen von einem Roboter zu einem anderen, als auch Multicast- und Broadcast-Verbindungen von einem zu mehreren bzw. allen Robotern. Wird der Empfang eines gesendeten Datenpaketes nicht innerhalb einer

---

bestimmten Zeit bestätigt, wird das Paket erneut gesendet, sodass die Verbindung für Anwendungen zuverlässig ist. Große Datenmengen werden, wenn nötig in, kleinere Pakete aufgeteilt.

Das Paket ist Teil eines drei Pakete umfassenden Projekts zur kooperativen autonomen Erkundung unbekannter Gebiete durch Roboter. Dies ist an vielen Stellen des Quellcodes deutlich zu erkennen. So werden z.B. Nachrichtentypen und Services implementiert, die Ihnen Nutzen in diesem Bereich haben.

Zum Senden ist der einfache Aufruf eines Services nötig. Das Paket kümmert sich um die komplette Kommunikation. Trifft die Nachricht auf dem Zielroboter ein, so wird sie auf einem vorher spezifizierte Topic gepublished. Das Paket stellt schon einige Services bereit, um selbstdefinierte Messages aus dem Bereich des autonomen Erkundens zu versenden.

---

### 3.2.1 Inbetriebnahme

Aufgrund des Fehlens einer konkreten Anleitung, gestaltete sich die Inbetriebnahme des Paketes zunächst schwieriger als gedacht. Die Lösung war jedoch relativ einfach: Um das Paket nutzen zu können, muss die drahtlose Netzwerkschnittstelle konfiguriert werden. Die Konfiguration geschieht am besten über die Kommandozeile. Ein selbst geschriebenes Bash-Skript setzt alle Parameter (siehe Quellcode 3.1).

---

#### Quellcode 3.1: Skriptdatei zum Konfigurieren der Schnittstelle

```
#!/bin/bash
# sets the parameters on wlan0
sudo ifconfig wlan0 down
sudo iwconfig wlan0 mode ad-hoc
sudo iwconfig wlan0 essid cars
sudo iwconfig wlan0 ap fe:ed:de:ad:be:ef
sudo ifconfig wlan0 up
```

---

Bevor die Einstellungen vorgenommen werden können, ist es nötig die Schnittstelle abzuschalten. Die Schnittstelle muss anschließend in den Ad-Hoc-Modus versetzt werden. Damit alle Autos sich auch im selben Ad-Hoc-Netzwerk befinden, müssen sie dieselbe *essid*- und *Access-Point*-Einstellung besitzen. Die Werte können dabei beliebig gesetzt werden. Auch wenn die Einstellung Access-Point heißt, sind die Autos nicht mit einem solchen verbunden. Da die Angabe des Ziel-Autos einer Nachricht in Form seines Hostnamen geschieht, ist es notwendig, dass alle Autos unterschiedliche Hostnamen besitzen. Da das Paket sogenannte Raw-Sockets verwendet, ist es notwendig den Besitzer der Programmdatei zu *root* zu ändern und das UID-Bit zu setzen. Dazu müssen die Befehle *sudo chown root* und *sudo chmod +s* auf die Programmdatei angewendet werden.

---

### 3.2.2 Datenfluss

Nicht implementiert ist die Funktion, jeden beliebigen Messagetyp zu senden. Um eigene Datentypen versenden zu können, ist es laut dem Wiki zum Paket [Cwi15] vorgesehen den Quellcode an diversen Stellen zu verändern. Dies steht aber im Widerspruch zu

dem Ziel der leichten Wiederverwendbarkeit. Daran soll diese Arbeit ansetzen. Um möglichst flexibel einsetzbar zu sein, soll das vorhandene Paket so verändert und ergänzt werden, dass es möglichst einfach ist eigene Nachrichtentypen zu versenden. Zunächst muss der Informationsfluss innerhalb des Paketes untersucht werden. Dies soll am Beispiel einer Student-Message (Quellcode 2.1) verdeutlicht werden. Dazu wurde gemäß dem Wiki der *sendStudent*-Service implementiert. Dieser enthält im Request-Teil eine Student-Message sowie den Hostnamen des Zielautos und das Topic, auf welchem später veröffentlicht werden soll.

#### Quellcode 3.2: Callbackfunktion des sendStudent-Service

```
string s_msg = getSerializedMessage(req.student);
res.status = sendPacket(req.dst_robot, s_msg,
    FRAME_DATA_TYPE_STUDENT, req.topic);
return res.status;
```

In Quellcode 3.2 ist ersichtlich, dass das Student-Objekt, welches dem Service im Request übergeben wurde zuerst mit dem Aufruf der *getSerializedMessage*-Methode serialisiert wird. Die so serialisierte Nachricht wird nun mit den anderen Parametern der *sendPacket*-Methode übergeben. Diese benötigt zusätzlich zu den zu sendenden Daten, Ziel-Host und Topic auch noch einen Integer-Wert (*FRAME\_DATA\_TYPE*), welcher durch eine symbolische Konstante repräsentiert wird. Innerhalb der *sendPacket*-Methode (*sendPackage* wäre vermutlich die korrektere Bezeichnung, der Name ist wörtlich aus dem vorhandenen Quellcode übernommen) werden die Datenpakete in Netzwerkframes aufgeteilt und letztlich an die *socketSend*-Methode übergeben. Diese leitet die Daten an die Netzwerkschnittstelle weiter.

Auf der Empfangsseite werden eingehende Frames wieder zu Paketen zusammengefügt und der *publishPacket*-Methode übergeben. Ein solches Paket enthält alle Daten, welche aus Sender-Seite an die *sendPacket*-Methode übergeben wurden. Nun wird der *FRAME\_DATA\_TYPE* an eine verschachteltes *if-else*-Statement übergeben. So wird sicher gestellt, dass je nach Datentyp die nächsten Schritte richtig gewählt werden.

#### Quellcode 3.3: Ausschnitt aus publishPacket

```
1 else if (p->data_type_ == FRAME_DATA_TYPE_STUDENT){
2     adhoc_communication::Student stud;
3     deserializeObject((unsigned char*) payload.data(),
4         payload.length(), &stud);
5     // adhoc_communication::RecvStudent recvStud;
6     // recvStud.Student = stud;
7     // recvRect.srcCar = hostname_source_;
8     // publishMessage(recvStud, p->topic_);
9     publishMessage(stud, p->topic_);
10 }
```

In Quellcode 3.3 werden in den Zeilen 1 bis 3 und 8 die immer notwendigen Schritte gezeigt: erstellen eines Message-Objekts, deserialisieren und auf dem Topic veröffentlichen. Häufig ist es notwendig, den Hostnamen des Source-Autos mit zu veröffentlichen.

Dazu muss ein eigener Messagetyp eingerichtet werden, welcher neben dem eigentlichen Datentypen noch ein Feld für die Zeichenkette des Quellautos bereithält. In diesem können dann die ursprüngliche Nachricht und der Hostname verpackt werden. Die dazu nötigen Schritte sind in Quellcode 3.3 auskommentiert, Zeile 8 würde logischerweise entfallen.

Um einen eigenen Nachrichtentypen zu versenden, müssen also folgende Schritte vollzogen werden:

- *defines.h*  
symbolische Konstante FRAME\_DATA\_TYPE definieren
- neue \*.srv-Datei  
Servicedefinition erstellen
- neue \*.msg-Datei  
Messagedefinition implementieren
- *CMakeList.txt*  
Service-Datei einbinden Message-Datei einbinden
- *header.h*  
Service-Header einbinden Message-Header einbinden
- *adhoc\_communication.cpp*  
Service-Callback implementieren  
Advertise Service  
neuen *else-if*-Block in *publishPacket* implementieren

Die Datei *CMakeList.txt* dient zum Konfigurieren des Build-Werkzeuges *CMake* und muss daher immer bearbeitet werden. Dies bedeutet, es müssen eine Datei erstellt und drei Dateien an insgesamt sechs verschiedenen Stellen bearbeitet werden. Darunter befindet sich auch die *adhoc\_communication.cpp*, welche knapp 3500 Zeilen lang ist, und daher sehr unübersichtlich. Um das Paket im eigenen Quellcode zu integrieren, müssen Message-Header und Service-Header eingebunden werden. Dieser komplexe Prozess soll erleichtert werden. Im Optimalfall wird er auf das Erstellen der Message-Definition (was sowieso geschehen muss) und das Einbinden eines einzigen Headers reduziert. Dieser Header und die CMakeList sollen die einzigen Dateien sein, in denen noch Veränderungen vorgenommen werden müssen.

### 3.3 Zusätzliche Serialisierung

Damit jeder Messagetyp gleich behandelt werden kann, muss die Nachricht schon vor der Übergabe an einen Serviceaufruf serialisiert werden. Die erste Idee war, einen gesonderten Service zu Implementieren. Da die Request-Felder eines Services jedoch die explizite Angabe eines Datentypes benötigen und daher nicht mit Templates realisiert werden können, kommt diese Lösung nicht in Frage.

---

Stattdessen wurde die *sendMessage*-Methode im Paket *adhoc\_communication* implementiert. Sie nimmt die Nachricht selbst, das Topic, das Zielauto und den *FRAME\_DATA\_TYPE* entgegen, serialisiert die Nachricht und versendet sie. Dazu wird der *sendAnything*-Service genutzt. Dieser nimmt die Nachricht schon als Zeichenkette entgegen, und verschickt sie unverändert. Damit die *sendMessage*-Methode jeden Nachrichtentyp verarbeiten kann, wurde sie mit Templates implementiert.

---

#### Quellcode 3.4: Definition der *sendMessage*-Methode

---

```
template <class t>
bool sendMessage(t message, uint8_t msg_type,
                 std::string dst_robot, std::string topic)}
```

---

Der Aufruf der *getSerializedMessage*-Methode innerhalb von *sendMessage* ist unproblematisch, da diese schon Templates nutzt. Die neu implementierte Methode ermöglicht nun das Senden beliebiger Nachrichten. Da die *sendMessage*-Methode komplett unabhängig von den verschickten Message-Typen arbeitet, ist es nicht nötig sie später zu ergänzen oder zu modifizieren. Daher kann sie als Teil des Communication-Pakets in der *functions.h* implementiert werden.

Das Deserialisieren und Veröffentlichen der Message muss jedoch für jeden Messagetyp entsprechend erweitert werden. Um die spätere Modifizierung des Quellcodes möglichst einfach zu gestalten, wurde dieser Teil in eine neuen Paket mit Namen *adhoc\_customize* ausgelagert. In diesem Paket befindet sich die Header-Datei *include.h*. Alle nötigen Quellcode-Veränderungen können in ihr vorgenommen werden:

- Definition des *FRAME\_DATA\_TYPE*
- include des Message-Headers
- *else-if*-Block in *publishPacket*

Damit diese Änderungen auch vom Communication-Paket aus erreichbar sind, muss dieses in Abhängigkeit des Customize-Pakets kompiliert werden. Ein selbst erstelltes Paket muss in Abhängigkeit beider Pakete (Customize und Communication) kompiliert werden.

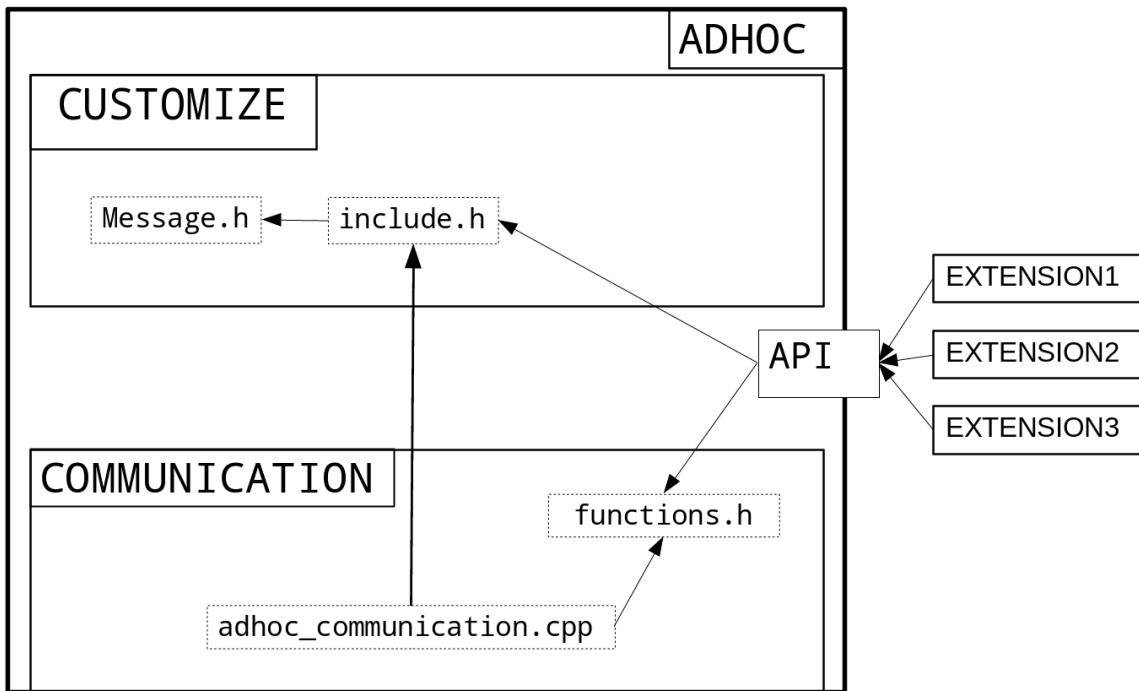
Abbildung 3.3 zeigt, in welchen Beziehungen die unterschiedlichen Pakete nun zueinander stehen. *Adhoc* ist dabei kein echtes ROS-Paket, sondern nur die Vereinigung der Pakete *Communication* und *Customize*. Mögliche eigene Erweiterungen zum Senden oder Empfangen von Nachrichten über das Ad-hoc-Netzwerk knüpfen über die Header-Dateien *include.h* und *functions.h* an.

---

### 3.4 Wiederverwendung

---

Durch die Einführung des zusätzlichen Paketes *Customize* und die Implementierung der *sendMessage*-Funktion konnte der zum Versenden eigener Nachrichten nötige Aufwand erheblich verringert werden. Möchte man nun aus einem selbst geschriebenen Node heraus eine Nachricht (z.B. *Student*) versenden, sind nur noch folgende Schritte notwendig:



**Abbildung 3.3:** Veranschaulichung der Abhängigkeiten

- Customize-Paket:
  - Student.msg erstellen
  - Student.msg in der CMakeList.txt hinzufügen
  - include.h:
    - \* FRAME\_DATA\_TYPE\_STUDENT definieren
    - \* #include "adhoc\_customize/Student.h"
    - \* else-if-Teil hinzufügen
- Eigenes Paket:
  - Quellcode:
    - \* #include "adhoc\_customize/include.h"
    - \* #include "adhoc\_communication/functions.h"
  - CMakeList.txt:
    - \* Abhängigkeit zu adhoc\_customize hinzufügen
    - \* Abhängigkeit zu adhoc\_communication hinzufügen

Die in der *include.h* vorzunehmenden Veränderungen werden auch im Quellcode selbst verdeutlicht, um die Modifizierung so einfach und intuitiv wie möglich zu gestalten. Nun ist der Aufruf der *sendMessage*-Methode mit der selbstdefinierten Message möglich. Man sieht, dass zum Verwenden der Ad-hoc-Kommunikation lediglich das Einbinden von zwei Header-Dateien nötig ist. Veränderungen an vorhandenem Quellcode beschränken sich auf drei Stellen in einer einzigen, übersichtlich gestalteten Datei.

---

## 4 Evaluation

---

In diesem Kapitel soll der implementierte Kommunikationsansatz evaluiert werden. Durch den Vergleich von Messwerten in verschiedenen Szenarien, soll die Performanz des gesamten Kommunikationssystems bewertet werden, und der Einfluss verschiedener Faktoren auf das System bestimmt werden. Anschließend sollen die gemessenen Werte interpretiert und mit den Erwartungen verglichen werden.

---

### 4.1 Forschungsfragen

---

Um die Performanz bewerten zu können, soll die Leistungsfähigkeit an einigen Größen bemessen werden. Zu jeder dieser Größen stellen sich einige Fragen, die geklärt werden sollen.

- Distanz
  - Welche Distanzen können direkt zwischen zwei Autos überbrückt werden?
  - Welche Distanzen können indirekt, also mit Zwischenknoten überbrückt werden?
- Datenrate:
  - Welche Datenübertragungsraten sind in dem gebildeten Ad-hoc-Netz möglich?
  - Welche Auswirkungen haben die Distanz und die Anzahl an Zwischenknoten zwischen den Autos?
- Latenz
  - Welche Zeit benötigt eine Datenübertragung mindestens?
- Korrektheit
  - Wird bei Abreißen einer Direktverbindung automatisch auf eine indirekte Verbindung umgeschaltet?
  - Werden neu entstandene Verbindungen automatisch in das Netz einbezogen?

All diese Größen müssen natürlich unter verschiedenen Voraussetzungen bestimmt und anschließend verglichen werden, wie z.B. direkte gegen indirekte Verbindung.

---

### 4.2 Evaluations-Setup und Testumgebung

---

Zuerst muss geklärt werden, wie und wo getestet wird. Als Testsoftware werden drei selbst implementierte ROS-Nodes genutzt: Sender, Mirror und Receiver. Als Testumgebung bietet sich die spätere Einsatzumgebung in den Räumen der TU Darmstadt.

---

#### 4.2.1 Software

---

Der **Sender-Node** lässt sich über den Parameter-Server des ROS-Masters den jeweils genutzten Anforderungen anpassen, ohne dass ein erneutes Kompilieren des Quellcodes nötig ist. Die Parameter sind alle im Namespace “/sender”, was bedeutet, dass der komplette Parametername “/sender/<name>” angegeben werden muss um den Wert zu setzen. Des weiteren generiert der Sender-Node eine Kurzbeschreibung der aktuellen Parameter. Diese wird dem Receiver-Node über einen Service mitgeteilt und dann als Dateiname zur Ausgabe verwendet. Die Parameter sind :

- loop  
Der Loop-Wert bestimmt, wie oft die Sendefunktion wiederholt wird, und damit wieviele Messwerte aufgenommen werden.
- strLen  
Um das Senden großer Daten zu simulieren, wird über den strLen Parameter die Länge einer bedeutungslosen "dummy"-Zeichenkette bestimmt. Diese wird mit 10 Byte initialisiert, und anschließend so oft an sich selbst gehängt wie der strLen Parameter vorgibt. Die so erzeugte Zeichenkette entspricht der zu sendenden Netto-Datenmenge.
- mode  
Über den Mode-Parameter wird der Modus, in dem der Sender sendet bestimmt. Die Modi sind wie folgt:
  - 1  $\hat{=}$  serialize String  
Modus 1 sendet einen String mit der sendMessage-Methode. Das heißt er durchläuft eine Serialisierung.
  - 2  $\hat{=}$  String Service  
Modus 2 sendet einen String direkt über den sendString-Service. So ist es möglich den Effekt der Serialisierung auf die Übertragung einzuschätzen.
  - 3  $\hat{=}$  Rectangle  
Modus 3 sendet eine Rectangle-MESSAGE. Dies diente anfangs zum Testen des Sendens eines beliebigen Nachrichtentyps unter Verwendung der Serialisierung. Für die Evaluation hat dieser Modus keine Bedeutung.
  - 4  $\hat{=}$  Student  
Dieser Modus wurde nur implementiert um die Serialisierung einer Student-MESSAGE zu testen
  - 5  $\hat{=}$  Ping  
In Mode 5 wird eine Nachricht mit Zeitstempel erstellt, ohne weiteren Inhalt. Diese Nachricht wird mit der sendMessage-Methode versendet.
- rate  
Über den Rate-Parameter wird die Zeit bestimmt, die bei jedem Aufruf von `rate.sleep()` gewartet wird, siehe Quellcode 4.1 Zeile 7. Dabei gilt:  $\text{sleepTime} = \text{rate}^{-1}$ .
- dst\_car  
Der Parameter dst\_car bestimmt das Zielauto, welches die Daten empfangen soll. Darüber können auch Multicast und Broadcast gesteuert werden.
- pos  
Dieser Parameter hat für die Ausführung des Programms keine Bedeutung, sondern wird nur in die Ausgabe übernommen, um zwischen Messungen auf kurzer bzw. langer Distanz zu unterscheiden.

---

#### Quellcode 4.1: Pseudocode des Senders

---

```
1 init();
2 data = strLen * dummy;
3 for(int i; i<loop; i++){
4     time = now();
5     message = merge(data, time)
6     send(message);
7     if (sleep) rate.sleep();
8 }
```

---

Quellcode 4.1 zeigt abstrakt die Funktion des Senders. Die Variable *data* ist die bedeutungslose Zeichenkette, welche eine Nutzlast simulieren soll. Bei jedem Durchlauf der *for*-Schleife wird ein Datenpaket gesendet. Die Senderate wird durch den *rate.sleep()*-Aufruf gesteuert.

Der **Mirror-Node** dient dazu, den Empfang von Datenpaketen auf dem Zielauto zu simulieren. Bei Empfang der Daten können diese ausgegeben werden, oder bei langen Zeichenketten deren Länge. Zusätzliche Funktionalitäten hat der Node bei den Laufzeitmessungen. Die dazu implementierten Callback-Funktionen müssen den empfangenen Zeitstempel wieder an das Ausgangsauto weiterleiten. Damit dies möglich ist, muss vor dem Veröffentlichen des Paketes der Zeitstempel in eine RecvTime-Message (Quellcode 4.2) verpackt werden. So ist es möglich, in der Callbackfunktion auf den Absender des Paketes zuzugreifen, und ihm den Zeitstempel wieder zurück zu senden.

---

#### Quellcode 4.2: Receive-Time-Message

---

```
time time
string src_car
```

---

Außerdem hat der Mirror-Node die Fähigkeit eine kurze Audiodatei über einen Systembefehl wiederzugeben. Dies dient bei Reichweitenmessungen dazu, festzustellen ob das Auto, auf welchem der Mirror läuft, sich noch in der Reichweite des Senders befindet. Die vergleichsweise komplexe Audiowiedergabe über die Soundkarte ist nötig, da die Mainboards nicht mit Pins für einen eigenen Piezo-Speaker ausgestattet sind.

Der **Receiver-Node** dient dazu die Laufzeiten zu berechnen. Er läuft auf demselben Auto wie der Sender-Node, und empfängt die Zeitstempel, die dieser vorher versendet hat. Er berechnet nun die Laufzeiten der Pakete und gibt diese auf der Konsole aus und speichert sie in eine Datei. Um die Messwerte später gut verarbeiten zu können, werden die einzelnen Werte durch ein Semikolon abgeschlossen und im Comma-Separated-Value-Format (\*.csv) gespeichert. So sind sie später leicht in ein Programm zur Tabellenkalkulation zu importieren und können weiterverwertet werden.

Um die Übertragungszeiten unabhängig von Sleep- und Rate-Parametern bestimmen zu können, wurde eine zweite Version der Test-Software implementiert. Die Funktionen des Senders und Receivers wurden zusammengefasst in einen **All-in-One-Node** (AiO). Unverändert weiter genutzt wird die Implementierung des Mirror-Nodes.

Dar AiO-Node wurde so implementiert, dass er erst ein neues Datenpaket sendet, wenn entweder eine Antwort auf das zuvor gesendete Paket erhalten wurde, oder ein

Timeout-Wert seit der Sendung überschritten wurde. Diese Funktion wurde über einen einfachen Zustandsautomaten realisiert, siehe Quellcode 4.3.

#### Quellcode 4.3: Pseudocode All-in-One

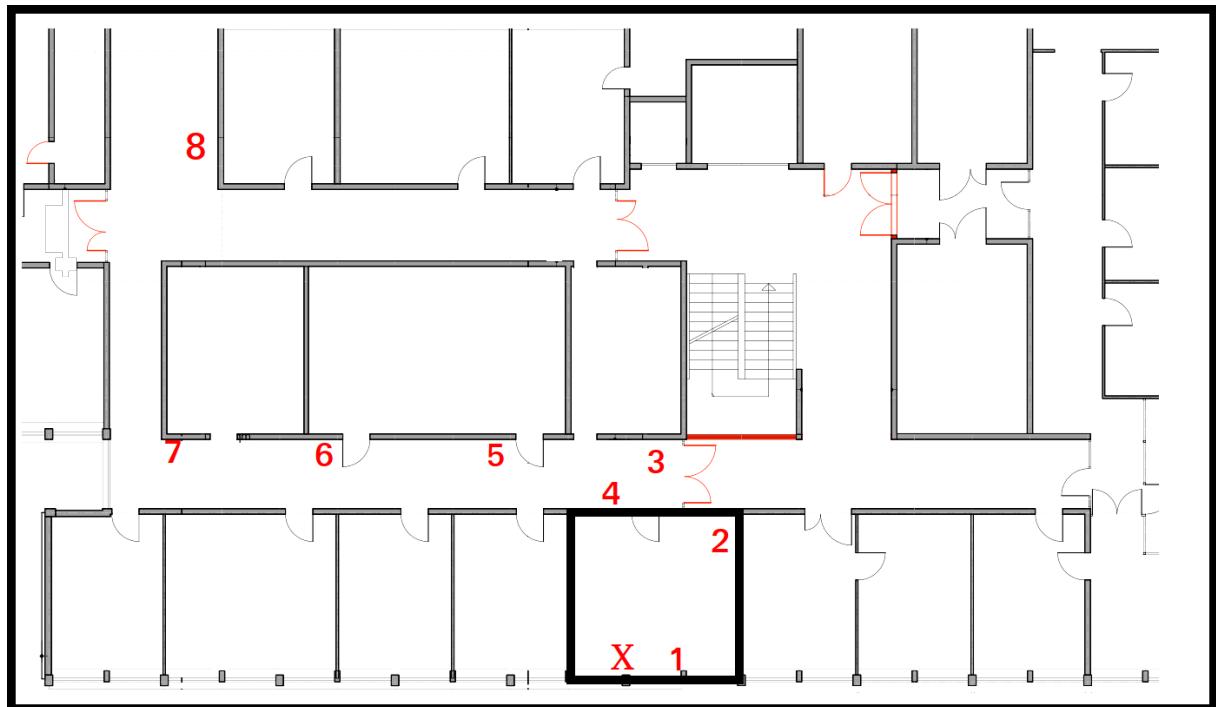
```
void answerCallback(RecvTime recvTime){  
    ros::Time afterSend = ros::Time::now();  
    float sendTimeSec = (afterSend - recvTime->time);  
    valArray[readingCounter] = sendTimeSec;  
    readingCounter++;  
    received=true;  
    if (readingCounter == loop) finish = true;  
}  
  
int main (int argc, char **argv){  
    initNode();  
    state = SEND;  
    while(ros::ok()){  
        switch (state){  
            case SEND:  
                sendMessage(ros::Time::now());  
                nextState = WAIT_FOR_ANSWER;  
            case WAIT_FOR_ANSWER:  
                if (timeout || received)  
                    nextState = SEND;  
                else  
                    nextState = state;  
            case FINISH:  
                makeOutput();  
        }  
    }  
}
```

Ein weiterer Unterschied ist die Ausgabe der Messdaten. Der AiO-Node speichert alle Messwerte in ein Array. So ist es möglich, statt der rohen Messdaten statistische Größen auszugeben. Durch Sortierung des Arrays ist es leicht, Lagemaße wie Quartile und Quantile auszugeben. Des weiteren werden Minimal- und Maximalwert ausgegeben, sowie die Anzahl der *Timeouts*, und der Prozentsatz der Werte, welche den eingestellten *Threshold* überschreiten. Diese Art der Ausgabe macht es einfacher, die Daten in einem Tabellenkalkulationsprogramm weiter zu verarbeiten.

#### 4.2.2 Testumgebung

Sinn dieser Evaluation soll es sein, möglichst realistische Ergebnisse zu erzielen. Zum Erzielen optimaler Ergebnisse wäre sicherlich ein Freiluftgelände, in dem es keine störenden Wände oder andere Funknetze gibt, am besten geeignet. Da die Modellautos im Rahmen des Projektseminars in aller Regel nur im Gebäude genutzt werden, soll diese Umgebung auch als Testumgebung dienen. Die Flure und Räume des Hans-Busch-Instituts der TU-Darmstadt sind der vorgesehene Einsatzbereich der Autos, und in Be-

zug auf die oben genannten Störfaktoren sicherlich nicht ideal. Im Gebäude gibt es viele Wände und andere Funknetze, welche die Autos stören können.



**Abbildung 4.1:** Grundriss der Testumgebung mit Messpositionen

Abbildung 4.1 zeigt einen Grundriss der Testumgebung. Zur besseren Illustration ist nur ein Teil des Stockwerks abgebildet. Schwarz eingerahmt ist der Arbeitsraum, von dem aus die Messungen stattfanden. Mit Zahlen markiert sind die Positionen des Mirror-Autos, mit einem roten X markiert ist die Position des Senders. Eine Übersicht über die ungefähren Distanzen zwischen den Autos zeigt Tabelle 4.1. Die Messdistanzen wurden absichtlich so gewählt, dass auch Wände zwischen den Autos sind. Messungen auf dem Flur bei Sichtkontakt, wären einfacher umzusetzen, jedoch wären die Messergebnisse nicht so realistisch.

Position	Distanz [m]
1	2,00
2	6,00
3	7,00
4	5,50
5	8,00
6	12,00
7	16,00

**Tabelle 4.1:** Distanzen der Messstellen

## 4.3 Allgemeine Tests

Zuerst sollen allgemeine Größen wie die Reichweite, Korrektheit und Einfluss der Paketgröße bestimmt werden. Später wird dann die Leistung bei direkten bzw. indirekten Verbindungen bestimmt.

### 4.3.1 Reichweite

Die maximale Reichweite der Kommunikation zwischen zwei Autos wurde wie folgt ermittelt: Der Sender-Node wird im Ping-Modus auf eine Senderate von 1 Hz gestellt. Das sendende Auto wird stationär betrieben (rotes X in Abbildung 4.2), während das empfangende Auto bewegt wird. Über die Audioausgabe des Mirrors, lässt sich feststellen, ob eine Verbindung zustande kommt oder nicht. Die Distanz zwischen beiden Autos kann nun als Reichweite angesehen werden. Die maximale Reichweite konnte entlang des Flurs gemessen werden (rote Linie in Abbildung 4.2). Sie beträgt etwa 30 m. Dabei haben beide Autos Sichtkontakt, das einzige Hindernis ist eine Feuerschutztür. In Abbildung 4.2 grün markiert ist der Bereich, in dem das Mirror-Auto die Pakete des Senders noch empfangen konnte.

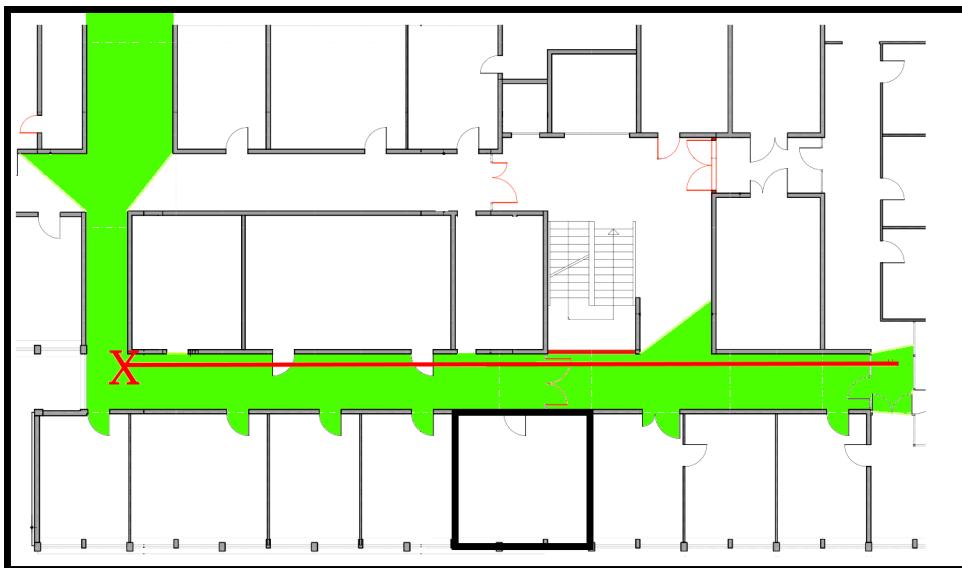


Abbildung 4.2: Abdeckung des Funksignals des Senders

### 4.3.2 Korrektheit

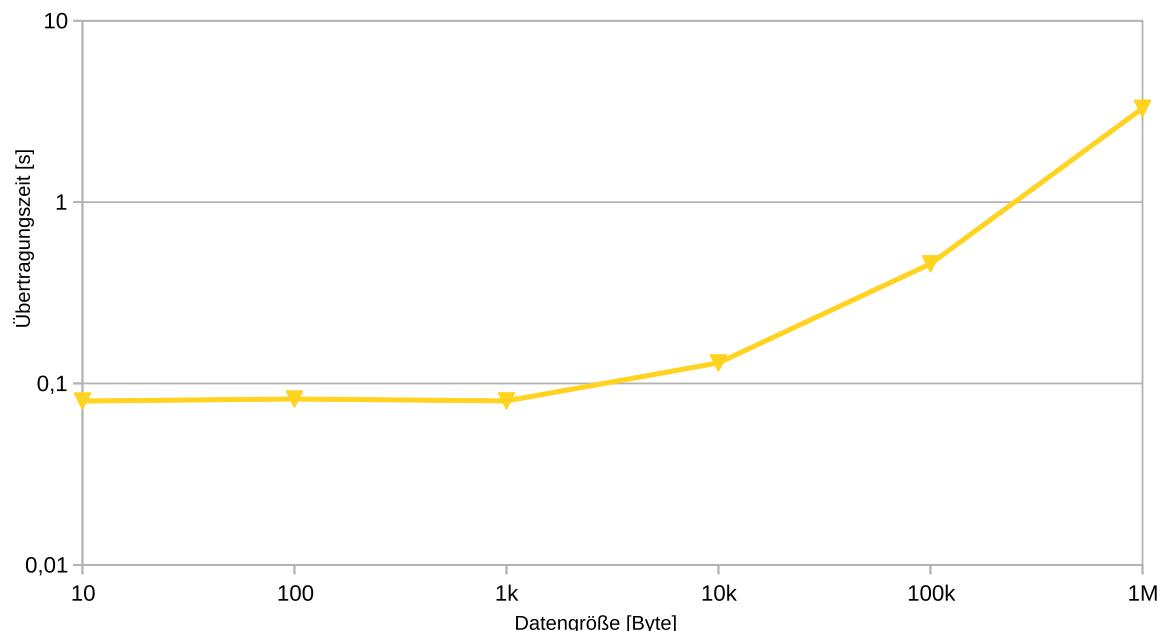
Nun soll festgestellt werden, ob der Routing-Algorithmus korrekt arbeitet. Dass Unicasts bei Direktverbindungen zwischen zwei Autos fehlerfrei funktionieren, wurde durch die Reichweitenmessung bereits implizit bewiesen. Nun gilt es noch zu testen, ob auch Multicasts und indirekte Verbindungen möglich sind. Dazu sind drei Autos nötig, Auto A, B und C. Diese sollen so angeordnet werden, dass sich Auto B etwa mittig zwischen A und C befindet. Konkret auf die Messpositionen bezogen steht dann Auto A bei Position 1, Auto B bei Position 7 und Auto C bei Position 8.

Durch einfaches Senden von Ping-Nachrichten und Empfangen von Antworten, ist das Funktionieren der Weiterleitung von Paketen durch Auto B bewiesen. Auch das Umschalten von Direktverbindung zu indirekter Verbindung bei Verbindungsabbruch aufgrund der Fortbewegung eines Autos konnte bei diesen Tests beobachtet werden. Zu Problemen kommt es jedoch immer wieder, wenn die Autos sich knapp an der Grenze der Reichweite eines anderen Autos bewegen. Dann waren instabile Verbindungen zu beobachten, was heißen soll, dass z.B. Auto A zwar Auto B als seinen Nachbarn erkennt, jedoch nicht andersherum.

Auch das Funktionieren von Multicasts und Broadcasts konnte beobachtet werden. dazu wurden die Autos alle in einem Raum betrieben. Broadcasts funktionieren anstandslos, auch wenn Tests gezeigt haben, dass sie deutlich mehr Netzwerkauslastung erzeugen als Uniscasts. Multicasts sind mit mehr Programmieraufwand verbunden und funktionieren aufgrund des komplexen Mechanismus zum Erstellen eines Multicast-Baumes nur eingeschränkt. Es kommt immer wieder zu vermeintlichen Verbindungsabbrüchen, welche die Übertragungen verlangsamen.

#### 4.3.3 Overhead

Um den Einfluss der Datenmenge pro Sendevorgang auf die Übertragungszeiten zu bestimmen, wurden beide Autos auf dem Schreibtisch betrieben (Position 1, Abbildung 4.1). Die Datenmenge wurde über den *strLen*-Parameters variiert.



**Abbildung 4.3:** Übertragungsdauer in Abhängigkeit von Netto-Datenmenge

Abbildung 4.3 zeigt, dass ein annähernd linearer Zusammenhang zwischen Netto-Datenmenge pro Paket und durchschnittlicher Übertragungszeit erst ab Netto-Datenmengen von mehr als 10 kB gegeben ist. Daher wurden die Messungen mit einem *strLen*-Parameter von 10000, was einer Netto-Datenmenge von 100 kB entspricht, durchgeführt.









---

84,3 Sekunden gemessen. Dementsprechend wurden 44,6% der Programmlaufzeit zum Verbindungsaufbau benötigt.

---

#### 4.5.2 Daten

Die Messungen zur Datenübertragungsrate bei indirekter Verbindung wurden nach dem selben Schema durchgeführt, wie bei direkter Verbindung. Aus acht Messreihen wurden die beiden extremsten gestrichen und der Mittelwert aus den übrigen sechs gebildet. Dabei ergibt sich eine durchschnittliche Übertragungszeit von 1 kB Nutzdaten von 59,8 ms. Die Reduzierung auf Nutzdatengrößen von 1 kB war nötig, da größere Datenmengen nur selten übertragen werden konnten. Und auch bei der verringerten Nutzdatenmenge kamen immer noch 2,6% Timeouts vor.

Wie auch bei der Latenzbestimmung wurde auch hier die Gesamlaufzeit des Programmes bestimmt. Aus den gemessenen Werten ergibt sich ein Anteil von 78,9% des Verbindungsaufbaus an der Gesamlaufzeit.

---

### 4.6 Diskussion

In diesem Abschnitt sollen die gemessenen Werte diskutiert werden. In Bezug zu den Einsatzbereichen soll die Performanz des Lösungsansatzes bewertet werden. Es war zu erwarten, dass die Dauer zur Datenübertragung negativ von den Faktoren Distanz, Zwischenknoten und anderer Netzwerkauslastung beeinflusst wird. Des Weiteren lag die Vermutung nahe, dass eine möglichst große Datenmenge am Stück schneller übertragen wird, als dieselbe Datenmenge in stärkerer Fragmentierung. Schuld daran wäre ein relativ zu den Nutzdaten größerer Overhead, der die gesamt zu Übertragende Datenmenge erhöht.

---

#### 4.6.1 Reichweite

Es ist zu vermuten, dass die maximal gemessene Reichweite von etwa 30 m nur selten tatsächlich voll ausgeschöpft wird. In einer idealen Umgebung scheint es jedoch durchaus möglich, noch höhere Reichweiten zu erreichen. Für den Einsatz in den Fluren des Hans-Busch-Instituts ist diese Reichweite jedoch als vollkommen ausreichend zu betrachten. Die gemessene Strecke war zugleich die Länge des längsten Flures. Eine größere Strecke wird im Projektseminar nicht direkt überbrückt werden müssen. Des Weiteren hat sich gezeigt, dass massive Wände den Empfang stark verschlechtern. Dieses Problem war jedoch zu erwarten und ist nicht zu vermeiden.

---

#### 4.6.2 Latenz

Die Latenz zeigt, dass bei Positionen eins bis sechs 90% aller gemessenen Werte unter 72 ms liegen. Die naheliegende Bewertung dieses Wertes wäre seine Bedeutung in einem sicherheitskritischer Moment. Angenommen ein Auto muss einem anderen eine sicherheitskritische Nachricht übermitteln. Dies würde bedeuten, dass die Übermittlung der Information von Auto A zu Auto B in 90% aller Fälle maximal 72 ms in Anspruch nimmt. Aufgrund der geringen Geschwindigkeit der Autos, scheint dieser Wert sehr gut.

---

---

Die Kommunikation würde die Reaktionszeit von Auto B vermutlich nicht gravierend beeinflussen.

---

#### 4.6.3 Übertragungsrate

Auch wenn die Übertragung von großen Datenmengen nicht zu den klassischen Problemstellungen des autonomen Fahrens gehört, zeigen die Messwerte, dass es zumindest bei Direktverbindungen möglich ist. Indirekte Verbindungen hingegen sind auf kleine Nachrichten beschränkt.

---

### 4.7 Schlussfolgerungen

Die Messwerte bei direkten Verbindungen sind alle absolut zufriedenstellend. Bei dem ausschließlichen Einsatz von direkten Verbindungen ist es unwahrscheinlich, dass die gemessenen Limits zu einer Einschränkung der Einsatzgebiete im Projektseminar werden. Die Reichweite ist ausreichend hoch, um Aufgaben auf den Fluren des Hans-Busch-Instituts zu lösen. Die Latenzen sind gering genug, dass auch sicherheitskritische Nachrichten schnell ihr Ziel finden. Auch die Übertragung von größeren Datenmengen ist zügig möglich. Bei den indirekten Verbindungen hingegen sind klare Probleme erkennbar. Aufgrund der massiven Schwierigkeiten im Verbindungsaufbau kann man von indirekten Verbindungen nur abraten. Sie ermöglichen keine verlässlichen Übertragungen. Es ist zu empfehlen, die Aufgabenstellungen im Projektseminar so zu konzipieren, dass auf indirekte Verbindungen verzichtet werden kann. Die Verwendung von Broadcasts ist möglich, jedoch sollte auf große Datenmengen verzichtet werden. Multicasts sind nicht zu empfehlen. Es erscheint sinnvoller, kleine Nachrichten per Broadcast zu verteilen, und diese auf Autos, welche nicht als Ziel gedacht waren, einfach zu verwerfen.



---

## 5 Fazit und Ausblick

---

Im letzten Kapitel dieser Arbeit soll zuerst auf zwei Arbeiten eingegangen werden, welche sich ebenfalls mit der drahtlosen Kommunikation von autonomen Fahrzeugen beschäftigen. Im Fazit soll die vorliegende Arbeit rekapituliert und ihre Ergebnisse zusammengefasst werden. Abschließend soll ein Ausblick über mögliche Anwendungsszenarien der Ad-hoc-Kommunikation gegeben werden.

---

### 5.1 Verwandte Arbeiten

---

In diesem Abschnitt sollen einige Arbeiten vorgestellt werden, welche sich mit ähnlichen Themen beschäftigen wie die vorliegende.

Zuallererst muss hierbei natürlich die Arbeit "Coordinated Multi-Robot Exploration: Out of the Box Packages for ROS" [ANB14] von Andre et al. an der Alpen-Adria-Universität (AAU) Klagenfurt erwähnt werden. Im Rahmen dieser Arbeit wurde das hier verwendete ROS-Paket `adhoc_communication` implementiert. Wie in dieser Arbeit, bilden auch die Roboter des Teams der AAU ein drahtloses Ad-hoc-Netz zur Kommunikation. Hauptziel der Arbeit war jedoch nicht die Ad-hoc-Kommunikation an sich, sondern das autonome Erkunden unbekannter Gebiete ohne vorhandene Infrastruktur durch Roboter. Es wird keine Verbindung zum Straßenverkehr oder Modellautos gezogen. Daher wird logischerweise die Möglichkeit zur Kommunikation mit RSUs nicht in Betracht gezogen.

Die Kommunikation mit RSUs spielt in der Master-Thesis "Car2X-Kommunikation für autonom fahrende Modellautos" [Jun16] von Severin Junker eine größere Rolle. Hierbei werden die Anwendungsfälle einer Ampel und eines Verkehrszeichens betrachtet. Es wird dabei schon sehr konkret auf mögliche Nachrichtentypen und andere wichtige Details, wie z.B. die Zuordnung eines Verkehrsschildes zu einer Fahrrichtung, eingegangen. Eine weitere Gemeinsamkeit ist die Verwendung von WLAN-Hardware im 2,4 GHz-Bereich. Ein elementarer Unterschied ist jedoch die verwendete Netzwerkarchitektur. Auch wenn Junker das AODV-Protokoll vorstellt, wird in seiner Arbeit ein WLAN im Infrastrukturmodus betrieben. Folglich muss zu jeder Zeit ein Access Point verfügbar sein. In der hier vorliegenden Arbeit wurde jedoch auf sämtliche vorhandene Infrastruktur verzichtet, die Modellautos erstellen ihr eigenes Ad-hoc-Netzwerk auf.

---

### 5.2 Fazit

---

Im Rahmen dieser Arbeit wurde ein Ad-hoc-Kommunikationsansatz für die autonomen Modellautos des Fachgebiets Echtzeitsysteme der TU Darmstadt entwickelt. Dazu mussten zuerst die Anforderungen genau abgeklärt werden. Um möglichst wenig zusätzliche Infrastruktur einsetzen zu müssen, wurde sich für einen Ad-hoc-Ansatz entschlossen. Als nächstes musste geklärt werden, welche Funktechnik am besten geeignet ist. Dazu wurden die weit verbreiteten Techniken Bluetooth und WLAN miteinander verglichen. Da Bluetooth in erster Linie auf Energiesparsamkeit ausgelegt ist, bietet WLAN eine höhere Reichweite und Datenübertragungsrate. Außerdem existiert für Bluetooth keine standardisierte Implementierung von Ad-hoc-Netzen. Daher fiel die Wahl auf WLAN. Nun galt es zu klären, auf welche Art sich Datenpakete in einem Ad-hoc-Netz routen lassen.

Es ist klar, dass gängige Routingalgorithmen den speziellen Anforderungen in einem Ad-hoc-Netz nicht gewachsen sind. daher wurden die beiden reaktiven Protokolle *Dynamic Source Routing* (DSR) und *Ad-hoc On-Demand Distance Vector Routing* (AODV) vorgestellt und ihre Funktionsweise anhand eines Beispiels verdeutlicht. Im nächsten Abschnitt dieser Arbeit wurde das Meta-Betriebssystem ROS mit seinen Grundmechaniken vorgestellt. Zum Abschluss des Grundlagenkapitels wurde die zur Netzwerkkommunikation wichtige Serialisierung von Datenobjekten erklärt und anhand eines einfachen Beispiels verdeutlicht.

Zur Umsetzung der geforderten Ziele wurde das bereits existierende ROS-Paket *ad-hoc\_communication* genutzt. Um es noch flexibler zu gestalten und leichter einsetzbar zu machen, wurde eine zusätzliche Schnittstelle hinzugefügt, welche das Senden von Nachrichten für den späteren Anwender auf einen einzigen Funktionsaufruf reduziert. Im Kapitel der Umsetzung wurden außerdem noch die Modellautos mit ihrer Soft- und Hardware vorgestellt.

Getestet und evaluiert wurde der fertige Kommunikationsansatz schließlich auf den Modellautos des Fachgebiets, auf welchen er auch später im Rahmen des Projektseminars zum Einsatz kommen soll. Dazu wurde eigens ein ROS-Paket erschaffen, welches verschiedene Funktionen zum Testen von Reichweite, Übertragungsrate und Latenz implementiert. Die Tests zeigten, dass der hier vorgestellte Ansatz sehr gute Ergebnisse bei direkten Verbindungen liefert. Es ist jedoch auch aufgefallen, dass gerade bei indirekten Verbindungen und beim Betrieb am Rande der Reichweite große Probleme beim Verbindungsaufbau auftreten. Daher ist von indirekten Verbindungen nur abzuraten.

Er ermöglicht die Kommunikation zwischen den Autos, ohne dabei auf vorhandene Infrastruktur aufzubauen. So ist maximale Flexibilität gegeben. Die Autos sind nun in der Lage, selbstständig ihr Netzwerk aufzubauen, und Daten darin zu verteilen.

Die in Abschnitt ?? formulierten Anforderungen an den Kommunikationsansatz werden vollständig erfüllt.

### 5.3 Ausblick

Da diese Arbeit das klare Ziel hatte einen Kommunikationsansatz zu entwickeln, welcher auch im Rahmen des Projektseminars Echtzeitsysteme genutzt werden kann, sollen nun einige Überlegungen zur konkreten Einbindung in mögliche Aufgabenstellungen angestellt werden. Die Auswertung der Messdaten hat uns gezeigt, dass zu großen Distanzen und Datenmengen sowie indirekte Verbindungen nur schwierig zu meistern sind. Daher sollen die Aufgaben diesen Randbedingungen angepasst werden.

Eine naheliegende Aufgabenstellung wäre natürlich das kooperative Erkunden eines unbekannten Gebietes, wie es auch in [ANB14] geschehen ist. So könnte man versuchen, die Erkundung auf zwei Autos aufzuteilen, und im Idealfall doppelt so schnell zu einer Lösung kommen, wie bei der Erkundung mit nur einem Auto.

In Abschnitt 1 wurde bereits die Idee von intelligenten Straßenschildern oder Ampeln (RSUs) vorgestellt. Diese Teile der Verkehrsinfrastruktur müssten für die Modellversuche natürlich erst einmal entwickelt werden. Eine kostengünstige Art dies zu realisieren, wäre beispielsweise die Nutzung eines Einplatinencomputers wie dem Raspberry-Pi. In Revision drei ist dieser bereits mit WLAN-Hardware ausgestattet. Sein geringer Strom-

---

---

verbrauch ermöglicht die Nutzung eines Akkus, bei gleichzeitig ausreichender Rechenleistung. Im ROS-Wiki existieren Einträge zur ROS-Installation auf einem Raspberry-Pi, sodass auch dies kein Problem darstellen sollte. Als intelligentes Straßenschild könnte der Raspberry dann in periodischen Abständen Befehle aussenden, die auf Autos in Reichweite beispielsweise die zulässige Maximalgeschwindigkeit ändern. Auch die Modellierung einer Ampel in Software sollte gut realisierbar sein. Um den menschlichen Beobachter den Zustand der Ampel mitzuteilen, könnte man die General Purpose Input Output-Pins (*GPIO*) dazu verwenden, einige LEDs als Ampel-Leuchtmittel anzusteuern. Die Aufgabe der Studenten könnte es nun sein, alle Autos unter Verwendung der gefunkten Ampelinformationen sicher die Kreuzung passieren zu lassen. Zu den Herausforderungen dabei gehören Dinge wie das Finden der richtigen Warteposition vor der Ampel oder das voreinander links abbiegen oder auch die Gewährung der Vorfahrt, wenn man links abbiegen möchte.

Bereits beim Linksabbiegen an einer Ampelkreuzung ist es notwendig Informationen über den Gegenverkehr zu sammeln, da dieser Vorfahrt hat. Dies kann unter hohem Rechenaufwand visuell geschehen, oder durch Kommunikation von Auto zu Auto. Von diesem Gedanken ist es nun auch nicht mehr weit hin zu einer Kreuzung, bei der ausschließlich die Rechts-vor-Links-Regel gilt. Hierbei müssen sämtliche Informationen, die zum korrekten Fahren benötigt werden, durch Kommunikation mit den andern Autos gesammelt werden.

Ein weiteres Szenario wäre ein Rettungswagen oder ein ähnliches Einsatzfahrzeug. Bei Einschalten des Martinshorns wird eine Nachricht ausgesendet, die alle Autos dazu veranlasst, zügig an den Straßenrand zu fahren und das Einsatzfahrzeug passieren zu lassen.

Auf lange Sicht ist es das Ziel, die hier vorgestellten Ideen im realen Straßenverkehr zu etablieren, und so einen Beitrag zur Sicherheit der Autonomisierung unserer Fortbewegung zu leisten.



---

## Literatur

---

- [ANB14] ANDRE, Torsten ; NEUHOLD, Daniel ; BETTSTETTER, Christian: Coordinated multi-robot exploration: Out of the box packages for ROS. In: *Globecom Workshops IEEE*, 2014, S. 1457–1462
- [avm16] AVM.DE: *Übertragungsraten nach WLAN-Standard*. [https://avm.de/service/fritzbox/fritzbox-7490/wissensdatenbank/publication/show/514\\_WLAN-Verbindungen-langsam-geringe-Datenrate/](https://avm.de/service/fritzbox/fritzbox-7490/wissensdatenbank/publication/show/514_WLAN-Verbindungen-langsam-geringe-Datenrate/). Version: 2016, Abruf: 2017-03-22
- [CBR04] CHAKERES, Ian D. ; BELDING-ROYER, Elizabeth M.: AODV routing protocol implementation design. In: *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on IEEE*, 2004, S. 698–703
- [Cwi15] CWIORO, Guenter: *ROS-Wiki adhoc\_communication*. [http://wiki.ros.org/adhoc\\_communication](http://wiki.ros.org/adhoc_communication). Version: 2015, Abruf: 2017-04-24
- [DPBR03] DAS, Samir R. ; PERKINS, Charles E. ; BELDING-ROYER, Elizabeth M.: *Ad hoc On-Demand Distance Vector (AODV) Routing*. RFC 3561. <http://dx.doi.org/10.17487/rfc3561>. Version: Juli 2003 (Request for Comments)
- [Int13] INTEL: *Produktspezifikationen Intel Dual Band Wireless-AC 7260*. <https://ark.intel.com/de/products/75439/Intel-Dual-Band-Wireless-AC-7260>. Version: 2013, Abruf: 2017-03-12
- [JHM07] JOHNSON, David ; Hu, Y ; MALTZ, D: RFC 4728: The dynamic source routing protocol (DSR) for mobile ad hoc networks for IPv4. In: *The IETF Trust* (2007)
- [JM96] JOHNSON, David B. ; MALTZ, David A.: Dynamic source routing in ad hoc wireless networks. In: *Mobile computing*. Springer, 1996, S. 153–181
- [Jun16] JUNKER, Severin: *Car2X-Kommunikation für autonom fahrende Modellautos*, FU Berlin, Masterarbeit, Januar 2016
- [Jyo11] JYOTI, Nitasha S.: Comparative Study of Adhoc Routing Protocol AODV, DSR and DSDV in Mobile Adhoc NETwork. In: *International Journal of Applied Engineering Research* 7 (2011), Nr. 11, S. 2012
- [Rob15] ROBOTICS, Clearpath: *Intro to ROS*. [https://www.clearpathrobotics.com/assets/guides/ros/\\_images/ros101two.png](https://www.clearpathrobotics.com/assets/guides/ros/_images/ros101two.png). Version: 2015, Abruf: 2017-04-10
- [ROS14] ROS.ORG: *What is ROS?* <http://wiki.ros.org/ROS/Introduction>. Version: 2014, Abruf: 2017-04-05
- [ROS15] ROS.ORG: *Writing a Simple Service and Client (C++)*. <http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29>. Version: 2015, Abruf: 2017-04-05

---

---

[ROS16a] ROS.ORG: *Creating a ROS msg and srv.* <http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>. Version: 2016, Abruf: 2017-03-22

[ROS16b] ROS.ORG: *Writing a Simple Publisher and Subscriber (C++)*. <http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29>. Version: 2016, Abruf: 2017-03-22