

Bachelorarbeit 2020/04

Entwicklung eines Kommunikationssystems zur Aufnahme, Übertragung und Darstellung der Fahrdaten von Elektrofahrzeugen

Marcello Chiaramonte

08.03.2021

Bachelorarbeit 2020/04

Thema

Entwicklung eines Kommunikationssystems zur Aufnahme, Übertragung und Darstellung der Fahrdaten von Elektrofahrzeugen

Aufgabenstellung

Mit der zunehmenden Verbreitung der Elektromobilität entstehen neue Herausforderungen und gleichzeitig neue Chancen im Bereich der Netzintegration und der Mobilitätsplanung. Informationen über Elektrofahrzeuge, wie beispielsweise die aktuelle Position oder der State Of Charge, können den Verteilnetzbetreibern dazu dienen, das Energiemanagement zu optimieren und Ladevorgänge zu steuern. Mit einer geeigneten Verarbeitung und Darstellung der Informationen kann die Mobilitätsplanung der Nutzer vereinfacht werden. Ziel der Arbeit ist die Entwicklung eines Kommunikationssystems bestehend aus einem Elektronikmodul, eines Cloudbasierten Datenmanagementsystems und einer Benutzerschnittstelle. Das Elektronikmodul soll in Fahrzeuge integriert werden und relevante Daten des Elektrofahrzeugs erfassen, aufbereiten und drahtlos an ein übergeordnetes Datenmanagementsystem übermitteln. Die Fahrdaten sollen in einer Cloud gespeichert werden und über eine Benutzerschnittstelle in aufbereiteter Form angezeigt werden. Schließlich soll das Kommunikationssystem getestet und die geforderten Funktionen validiert werden.

Bearbeiter:	Marcello Chiaramonte	
Matrikelnummer:	108016259886	
Beginn der Arbeit:	27.10.2020	Ende der Arbeit: 08.03.2021
Betreuer:	M.Sc. Daniel Breuer	
Gutachter:	Prof. Dr.-Ing. Constantinos Sourkounis	
Zweitgutachter:	Prof. Dr.-Ing. Volker Staudt	

Institut für Energiesystemtechnik und Leistungsmechatronik
Prof. Dr.-Ing. Constantinos Sourkounis, Prof. Dr.-Ing. Volker Staudt
Fakultät für Elektrotechnik und Informationstechnik, Ruhr-Universität Bochum
Gebäude ID 1/221, 44780 Bochum, Telefon: 0234/32-23956, Telefax: 0234/32-14597

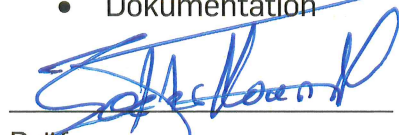
Bachelorarbeit für Herrn Marcello Chiaramonte

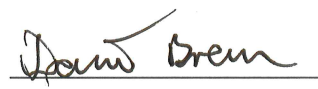
Thema: Entwicklung eines Kommunikationssystems zur Aufnahme, Übertragung und Darstellung der Fahrdaten von Elektrofahrzeugen

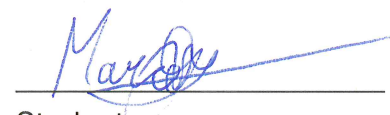
Mit der zunehmenden Verbreitung der Elektromobilität entstehen neue Herausforderungen und gleichzeitig neue Chancen im Bereich der Netzintegration und der Mobilitätsplanung. Informationen über Elektrofahrzeuge, wie beispielsweise die aktuelle Position oder der State Of Charge, können den Verteilnetzbetreibern dazu dienen, das Energiemanagement zu optimieren und Ladevorgänge zu steuern. Mit einer geeigneten Verarbeitung und Darstellung der Informationen kann die Mobilitätsplanung der Nutzer vereinfacht werden. Ziel der Arbeit ist die Entwicklung eines Kommunikationssystems bestehend aus einem Elektronikmodul, eines Cloubasierten Datenmanagementsystems und einer Benutzerschnittstelle. Das Elektronikmodul soll in Fahrzeuge integriert werden und relevante Daten des Elektrofahrzeugs erfassen, aufbereiten und drahtlos an ein übergeordnetes Datenmanagementsystem übermitteln. Die Fahrdaten sollen in einer Cloud gespeichert werden und über eine Benutzerschnittstelle in aufbereiteter Form angezeigt werden. Schließlich soll das Kommunikationssystem getestet und die geforderten Funktionen validiert werden.

Im Einzelnen ergeben sich folgende Aufgaben:

- Literaturrecherche
- Recherche zur Verfügbarkeit von Fahrzeug- und Ladedaten in Elektrofahrzeugen
- Recherche zu standardisierten Übertragungsverfahren und Kommunikationsprotokollen zum Informationsaustausch von Elektrofahrzeugen mit externen Systemen
- Aufstellung eines Anforderungskatalogs für das Elektronikmodul und Entwurf eines Konzepts zur Aufnahme, Übertragung, Speicherung und Darstellung der Daten
- Entwicklung des Elektronikmoduls
 - Auswahl geeigneter Bauelemente
 - Entwurf eines Schaltplans und eines Platinenlayouts, Fertigung der Platinen
 - Aufbau und Test eines Prototyps des Elektronikmoduls
- Einbau des Moduls in Fahrzeuge, Validierung der entwickelten Komponenten
- Präsentation der Ergebnisse im Rahmen eines Vortrags
- Dokumentation


 Prüfer
 Prof. Dr.-Ing. C. Sourkounis


 Betreuer
 Daniel Breuer, M.Sc.


 Student
 Marcello Chiaramonte

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Bochum, den 08.03.2021



Marcello Chiaramonte

Inhaltsverzeichnis

Inhaltsverzeichnis.....	I
Abkürzungsverzeichnis	IV
Abbildungsverzeichnis.....	V
Tabellenverzeichnis.....	VII
Listingverzeichnis	VIII
1 Einleitung.....	1
2 Problemstellung und Motivation.....	2
2.1 Elektrischen Fahrzeugen und Laden Möglichkeiten.....	2
2.2 Vernetzen Fahrzeugen.....	3
3 Stand der Technik.....	4
3.1 Verbindung eines Fahrzeugs mit dem Internet.....	4
3.1.1 Open Vehicle Monitoring System.....	4
3.1.2 Tesla API	5
3.2 Fahrdaten in Elektrofahrzeugen	5
3.2.1 Standardisierte CAN-basierte Protokolle	5
3.2.2 CAN-Bus.....	6
3.3 Open Systems Interconnection model.....	8
3.4 Technologien zur mobilen Datenübertragung	8
3.5 Internet-Kommunikationsprotokolle	10
3.5.1 MQTT	10
3.5.2 HTTP	12
3.5.3 JSON	13
3.6 Sichere Übertragung von Daten.....	14
3.6.1 Symmetric-Key-Verschlüsselung	14
3.6.2 Asymmetric-Key-Verschlüsselung	15
3.7 Microservices	15
3.7.1 Docker	16
3.7.2 Orchestrierung von Diensten	16
3.8 Microcontroller.....	17
4 Entwicklung eines neuen Systems	19
4.1.1 Anforderungskatalogs für das Elektronikmodul	19
4.1.2 Systemübersicht	20
5 Design und Aufbau der Hardware des Moduls	22
5.1 Gesamt-Hardware-Schaltplan	22

	II
5.2	Spannungsversorgung 22
5.3	CAN-Transceiver 23
5.4	USB-UART-Schnittstelle 23
5.5	Pycom GPy Microcontroller 23
5.6	Externe Komponenten und Verkabelung..... 24
5.7	Tasten für Reset, Bootloader und Safeboot 24
6	Design und Aufbau der Software des Moduls 25
6.1	Flashen der Firmware 25
6.2	Verwendete Bibliotheken..... 25
6.2.1	Uasyncio..... 25
6.2.2	CAN 26
6.2.3	UART 26
6.2.4	LTE und WLAN..... 26
6.2.5	SSL..... 27
6.2.6	MQTT 27
6.2.7	GPS 27
6.3	Program structure..... 27
6.3.1	Boot- und Main-Dateien 28
6.3.2	Vehicle Class 28
6.3.3	MQTT Client Class 28
6.3.4	WLAN and LTE Connectivity Class 30
6.3.5	GPS-Reader Class 30
6.3.6	CAN Controller Class..... 30
6.3.7	Fahrzeugspezifischen Klassen 31
6.3.8	Status des Moduls 32
7	Design und Aufbau der Cloud-Infrastruktur..... 33
7.1	Virtuelle Linux Maschine bei Amazon Web Services 33
7.2	Docker und Docker-compose 33
7.2.1	Dockerfiles 33
7.2.2	Docker-Compose 34
7.2.3	Docker Netzwerk 35
7.3	MQTT Broker 35
7.4	Python Connector 36
7.5	Java Spring Boot Backend REST-API..... 37
7.5.1	REST HTTP Controller 37
7.5.2	Class Diagramm 38

	III
7.5.3 Jakarta Persistence API	39
7.5.4 Verbindung mit der Datenbank	41
7.6 Angular Frontend Web App	41
7.7 Reverse Proxy	43
7.8 Deployment der Anwendung	43
8 Tests und Ergebnissen	45
8.1 CAN-Bus Frame-Erkennung und -Dekodierung	45
8.2 Fahrzeugortung über GPS	45
8.3 Datenverarbeitung in der Cloud	46
8.4 Frontend-Webanwendung	46
8.5 Weiterentwicklung	46
9 Zusammenfassung	47
10 Literaturverzeichnis	48
11 Anhang	49
Schaltplan der Leiterplatte	49

Abkürzungsverzeichnis

API	Application Programming Interface
ca.	<i>circa</i>
CAN-Bus	Controller Area Network
ECU.....	<i>Engine Control Unit</i>
EDGE	<i>Enhanced Data Rates for GSM Evolution</i>
Gbit/s	<i>Gigabits pro Sekunde</i>
GPS.....	Global Positioning System
HSPA+	High Speed Packet Access +
HTTP	Hypertext Transfer Protocol
IoT	<i>Internet of Things</i>
JSON.....	JavaScript Object Notation
Kbit/s	<i>Kilobits pro Sekunde</i>
LKW	Lastkraftwagen
LoRaWAN	<i>Long Range Wide Area Network</i>
LTE Cat M1	<i>Long-Term Evolution for Machines</i>
LTE Cat NB1	<i>Long-Term Evolution Narrow Band</i>
LTE-A	<i>Long-Term-Evolution-Advanced</i>
Mbit/s.....	Megabits pro Sekunde
MCU	Microcontroller Unit
MQTT	Message Queuing Telemetry Transport
NMEA.....	National Marine Electronics Association
OBD2	<i>On Board Diagnostics 2</i>
OSI.....	Open Systems Interconnection
OVMS.....	<i>Open Vehicle Monitoring System</i>
PCB.....	<i>Printed Circuit Board</i>
PID	<i>Parameter Identifier</i>
REPL.....	Read–Eval–Print Loop
REST.....	Representational State Transfer
RTOS	Real-Time Operating System, See
SMS	<i>Short Message Service</i>
SPI	Serial Peripheral Interface
SQL	Structured Query Language
SSL	Secure Sockets Layer
Tbit/s	<i>Terabits pro Sekunde</i>
TLS.....	Transport Layer Security
UART	Universal Asynchronous Receiver Transmitter
URI	Uniform Resource Identifier
USB.....	Universal Serial Bus
UTF-8	8-Bit Universal Coded Character Set Transformation Format
WLAN.....	Wireless Local Area Network
YAML	YAML Ain't Markup Language
z. B.	<i>zum Beispiel</i>

Abbildungsverzeichnis

Abbildung 3.1 Standard CAN Data Frame	6
Abbildung 3.2 Zusammenhang zwischen logischen Zuständen und CAN-Bus-Adern Spannungen	7
Abbildung 3.3 Übersicht der Client-Broker Architektur der MQTT Protokoll	10
Abbildung 3.4 Beispiel MQTT-Nachrichten des Standorts und des Ladezustands des Fahrzeugs.....	11
Abbildung 3.5 Darstellung der Request-Response-Architektur von HTTP	12
Abbildung 3.6 Beispiel HTTP-Anfrage und -Antwort.....	13
Abbildung 3.7 Beispiel für ein JSON-Array von Fahrzeugeigenschaftsobjekten ..	13
Abbildung 3.8 Verschlüsselung mit symmetrischem Schlüssel	14
Abbildung 3.9 Verschlüsselung mit asymmetrischem Schlüssel	15
Abbildung 4.1 Allgemeine modularisierte Systemübersicht.....	20
Abbildung 5.1 Vereinfachte Übersicht über das Hardware-Design.....	22
Abbildung 6.1 Struktur des Moduls Software.....	28
Abbildung 6.2 Beschreibung der Subroutine, die die Verbindung mit dem MQTT- Broker hält.....	29
Abbildung 7.1 Klassendiagramm des Projekts	38
Abbildung 7.2 Drei Tabellen mit relationalen Beispieldaten	40
Abbildung 7.3 Liste der vom System erfassten Fahrzeuge	42
Abbildung 7.4 Grafik des Batteriezustands eines Fahrzeugs im Zeitverlauf.....	42
Abbildung 7.5 Karte, die die vom Fahrzeug zurückgelegten Wege basierend auf GPS-Daten anzeigt	42
Abbildung 8.1 CAN-Bus-Datenframes, die vom seriellen Monitor des Mikrocontrollers angezeigt werden.....	45
Abbildung 8.2 Informationen über den Standort und die Zeit, die von einem GPS- Satelliten empfangen wurden	45
Abbildung 8.3 Links: MQTT-Nachricht mit Informationen über Fahrzeugkenndaten und die aktuelle Position. Rechts: Ergebnis der SQL-Abfrage auf eine Tabelle, die die Fahrzeugkennzahlen enthält	46

Tabellenverzeichnis

Tabelle 3.1 CAN-Bus-Felder und ihre Bedeutung	7
Tabelle 3.2 Schichten des OSI-Modells	8
Tabelle 3.3 Vergleich der Netzwerktechnologien für Mobilfunkkommunikation	9
Tabelle 3.4 Einige HTTP-Methoden und ihre Beschreibung	13
Tabelle 4.1 Anforderungen des neuen Systems.....	19
Tabelle 5.1 Umverdrahtungsschema des OBD2 - DB9 Kabels	24
Tabelle 6.1 Beschreibung der Mitsubishi Fahrzeug Kennzahlen.....	32
Tabelle 7.1 Wichtigste Dockerfile-Schlüsselwörter und ihre Bedeutung.....	34
Tabelle 7.2 Hauptschlüsselwörter der Docker-Compose-Dateien.....	34
Tabelle 7.3 Beschreibung einiger Schlüsselwörter der Datei mosquito.conf	36
Tabelle 7.5 Nomenklatur der Nginx-Webserver-Zertifikate.....	43

Listingverzeichnis

Listing 3.1 Beispiel für die Datei docker-compose.yml, die zwei Dienste definiert	17
Listing 6.1 Beispiel für zwei endlose MicroPython-Routinen, die asynchron ausgeführt werden.....	26
Listing 7.1 Inter-Container-Referenz innerhalb der Datei nginx.conf	35
Listing 7.2 Beispiel für eine MQTT Datachange-Nachricht und resultierende HTTP-Anfragen.....	37
Listing 7.3 Vereinfachtes Beispiel für einen HTTP-GET-Request-Handler.....	38
Listing 7.6 JSON-Payload mit Fahrzeugmetrikwerten	39
Listing 7.5 Java-Pseudocode einer Funktion, die Fahrzeugereignisse filtert.....	40
Listing 7.7 Inhalt der Datei application.properties.....	41
Listing 7.8 Kommandozeilenbefehle zum Erstellen und Ausführen der Cloud-Anwendung	44

1 Einleitung

Um die Herausforderungen der zunehmenden Elektromobilität zu beherrschen, wie z. B. die Planung neuer städtischer Infrastruktur, ist es notwendig, relevante Daten über die Nutzung von Elektroautos in Städten und auf Straßen zu sammeln. Um dieses Ziel zu erreichen, muss ein System zur Datenerfassung entworfen und entwickelt werden.

Fahrzeuge, die mit dem Internet verbunden sind, bringen nicht nur Vorteile für den Nutzer des Autos, sondern auch für die Umwelt und die Gesellschaft, da Fahrten innerhalb eines Stadtgebiets besser geplant werden können und gleichzeitig ein besseres Nutzererlebnis auf dem Weg von Punkt A zu Punkt B geboten wird.

Die Kernpunkte bei der Entwicklung eines solchen Systems sind die Erforschung der verfügbaren Daten über Elektrofahrzeuge und die Kommunikationsmethoden, um diese Daten zuverlässig online zu übertragen. Sobald die Daten in der Cloud verfügbar sind, müssen die Benutzer in der Lage sein, die wichtigsten Kennzahlen der Fahrzeuge auf einfache Weise auszulesen.

Die übertragenen Live-Daten über die Position und den Ladezustand des Fahrzeugs können auch für andere Projekte nützlich sein. Zum Beispiel wird die Planung von multimodalen Fahrten, bei denen auch Elektroautos eines der Transportmittel sein können, erleichtert, wenn der Standort und die Verfügbarkeit des Fahrzeugs bekannt sind.

Diese Arbeit wird die Schritte für den Entwurf, die Entwicklung und den Einsatz eines Systems vorstellen, das die vorgestellten Anforderungen erfüllt.

2 Problemstellung und Motivation

Dieses Kapitel gibt einen Überblick über die aktuelle Situation des Ladens von Elektrofahrzeugen und einige der Schwierigkeiten beim Aufbau einer Ladeinfrastruktur. Die Bedeutung der Erfassung von Fahrzeugdaten für eine bessere Planung dieser Infrastruktur wird ebenfalls erläutert.

2.1 Elektrischen Fahrzeugen und Laden Möglichkeiten

Da die Anzahl der verkauften Elektrofahrzeuge jedes Jahr steigt und immer mehr Modelle auf den Markt kommen, muss die Ladeinfrastruktur diesem Wachstum entsprechend folgen.

Basierend auf der Norm IEC 61851-1 gibt es vier Modi für das Laden von Elektrofahrzeugen. Jeder Lademodus erfordert eine andere Infrastruktur und bietet unterschiedliche Aufladungskapazitäten in Abhängigkeit von der örtlichen elektrischen Infrastruktur. Langsamere Ladestationen werden in der Regel am Zielort, wie z. B. am Arbeitsplatz oder in einem Parkhaus, oder zu Hause platziert und können bis zu mehreren Stunden benötigen, um einige wenige 100 Kilometer Reichweite zu liefern. Die Infrastruktur für langsamere Ladestationen ist in der Regel kostengünstiger als die der schnelleren Ladestationen, daher sind sie stärker verbreitet. Der Nachteil ist, dass das Aufladen über einen längeren Zeitraum erfolgen muss, was die Entscheidung des Kunden, ein Elektroauto zu kaufen, beeinflussen kann.

Um ein ähnliches Erlebnis wie beim Tanken eines Benzinautos zu bieten und dem Auto in wenigen Minuten eine große Reichweite zu verschaffen, müssen mehr Schnellladestationen in den Städten und auf den Straßen zwischen den Städten installiert werden. Beim Schnellladen werden in der Regel hohe Gleichspannungen und -ströme verwendet, die das Auto je nach Situation in ca. 20 Minuten fast volltanken können. Da die meisten Stromnetze Wechselspannungen liefern, sind die Kosten für die Installation und Herstellung von Schnellladestationen aufgrund der für die Umwandlung erforderlichen Technologie hoch. Eine optimierte Platzierung dieser Schnellladestationen ist daher notwendig, damit die Kosten für die Bereitstellung dieser Ladeinfrastruktur geringer sind und die Stationen nicht untergenutzt werden.

Um die optimalen Standorte für die Bereitstellung von Ladeinfrastruktur zu bestimmen, ist es notwendig, Daten über den Ladezustand und den Standort der Fahrzeuge zu sammeln. Mit einer repräsentativen Datenmenge ist es möglich, das Energienutzungsprofil der Fahrzeuge auf den Straßen zu analysieren und die Infrastruktur besser zu planen.

2.2 Vernetzen Fahrzeugen

Seitdem ersten Anwendungen von Live-Telemetrie in die Pisten der Formel 1 in den 80er und die Vernetzung von kommerziellen Autos für Verfügbarkeit von Hilfe in Notsituationen im Jahr 1996, werden die Anwendungen von an Netzwerken verbundenen Fahrzeugen ständig größer.

Damit ergibt sich das Problem, dass jeder Automobilhersteller die Fahrzeugdaten anders strukturiert. Dies hat auch den Nachteil, dass dem Benutzer des Fahrzeugs nicht klar oder bekannt gemacht wird, welche Daten tatsächlich vom Fahrzeug übertragen werden und wie die Daten gesichert und verarbeitet werden. Dies wirft Bedenken hinsichtlich des Datenschutzes persönlicher Daten auf.

Darüber hinaus ist die Vernetzung von Fahrzeugen in Netzwerken aufgrund der hohen Kosten, die mit der Einrichtung und Wartung solcher Systeme verbunden sind, nicht weit verbreitet. Von der Perspektive der Forschung, manche Daten die Interessant für die vollständigere Nutzung der Energie eines Fahrzeugs sind entweder nicht übertragen oder nicht zu dem Benutzer des Autos verfügbar gemacht, wie z.B. die Spannung und Temperatur Einzel Batteriezellen oder die Spannung und Strom der Batterie und eines Ladevorgangs. In solchen Fällen ist daher eine Nachrüstung eines Fahrzeugs erforderlich.

3 Stand der Technik

In diesem Kapitel wird ein Überblick über die notwendigen Technologien und Konzepte gegeben, die notwendig sind, um ein Elektrofahrzeug in das Internet zu integrieren und Daten dieses Fahrzeugs zu erfassen.

3.1 Verbindung eins Fahrzeugs mit dem Internet

3.1.1 Open Vehicle Monitoring System

Das OVMS ist ein Open-Source-Projekt, das eine Live-Überwachung von Fahrzeugen ermöglicht¹. Das System selbst besteht aus einem Board, das Daten-Frames aus dem CAN-Bus von Autos liest und drahtlos mit dem Internet kommuniziert, um Daten an einen Cloud-Server zu senden.

Dieses Projekt bietet Vorteile wie die Menge der abgedeckten Fahrzeugmodelle und die Community-Foren zur Fehlerbehebung. Es gibt jedoch auch einige Nachteile wie die Verwendung eines 3G-Modems für die mobile Kommunikation. Das 3G-Netz in Deutschland wird ab dem 30.06.2021 abgeschaltet², so dass ein Ersatz für das Mobilfunkmodul gefunden werden muss. Hier zeigt sich ein weiterer Nachteil des OVMS-Systems. Die Programmstruktur ist sehr groß und komplex und sie ist größtenteils in C geschrieben. Obwohl eine zuverlässige Programmiersprache, macht sie die schnelle Integration neuer Hardware sehr arbeitsintensiv. Bei einem Projekt mit einer kurzen Frist wurde eine Lösung mit einer schnelleren Entwicklungszeit als notwendig erachtet. Außerdem sind die PCB-Schaltpläne nicht Open Source, so dass es keine Garantie dafür gibt, dass die Hardware-Software-Integration von z.B. CAN-Controllern in einer selbst entworfenen Leiterplatte korrekt durchgeführt wird, wenn das Design nur auf diesem System basiert.

Da die Codebasis jedoch öffentlich ist, ist es möglich, die Funktionen, die jeden CAN-Daten-Frame dekodieren, für jedes von diesem System abgedeckte Fahrzeug zu nutzen. Aus diesem Grund ist dieses Projekt von großer Bedeutung für die Entwicklung einer neuen Lösung, obwohl die Hard- und Software nicht direkt genutzt werden.

¹ Open Vehicles.

² Telekom.

3.1.2 Tesla API

Tesla API ist ein Open-Source-Projekt mit dem Ziel, die Tesla-API zurückzuentwickeln, so dass Fahrzeug- und Ladedaten für den Besitzer des Fahrzeugs zugänglich sind³. Obwohl die Datenerfassung und -Übertragung in proprietärer Weise von Tesla durchgeführt wird, ist dieses Projekt als Inspiration erwähnenswert.

3.2 Fahrdaten in Elektrofahrzeugen

Seit 1996 müssen bei allen Leichtbaufahrzeugen die Emissionsdaten über einen OBD2-Stecker zur Verfügung gestellt werden. Elektrofahrzeuge produzieren keine direkten Emissionen, jedoch sind der OBD2-Stecker und der CAN-Bus weiterhin verfügbar, um Daten zu übertragen und auszulesen.

Um die Überwachung von Emissionen und die Diagnose von Fehlercodes von Fahrzeugen zu erleichtern, wurde eine Reihe von Standards für die Nummerierung der IDs der elektronischen Steuergeräte definiert. einige davon werden in Kapitel 3.2.1 vorgestellt.

Obwohl einige Fahrzeugdaten über den OBD2-Standard verfügbar sind, haben viele Hersteller von Elektrofahrzeugen eine freie ID-Nummer verwendet, um weitere elektronische Steuergeräte in ihren Fahrzeugen zu identifizieren. Es gibt jedoch keinen öffentlich verfügbaren Standard und jeder Hersteller hat unterschiedliche IDs und Kodierungen des Datenfeldes des Data Frames. Um lesbare Daten zu erhalten, ist es daher notwendig, die Kodierung der Frames zurück zu entwickeln.

3.2.1 Standardisierte CAN-basierte Protokolle

Der SAE J1939 Standard wird vor allem in schweren Nutzfahrzeugen, LKWs und landwirtschaftlichen Fahrzeugen eingesetzt. Die ID-Codes in diesem Standard haben einen 29 Bit langen Code⁴. Die Baudrate beträgt typischerweise 250 Kbit/s und es gibt eine komplexere Organisation der Parameternummern der Steuergeräte-IDs. Dieser Standard wird jedoch bei dieser Arbeit nicht berücksichtigt.

Der OBD2-Standard ist für Personenkraftwagen erforderlich und verwendet den Standard Data Frame mit einem 11 Bit langen Parameter Identifier (PID). Er wird in

³ Blau et al. 2021.

⁴ Prasad et al.

erster Linie für die On-Board-Diagnose von Verbrennungsemissionen sowie für Fehlercodes verwendet. Es ermöglicht auch den Zugriff auf das Motorsteuergerät (ECU). Da Elektrofahrzeuge keine Emissionen erzeugen, werden die meisten IDs dieser Standard nicht genutzt.

CANOpen ist ein Protokoll, das viele Netzwerkarchitekturen unterstützt, z. B. Master-Slave, Producer-Consumer, usw. Dieser Standard wird in vielen industriellen Anwendungen eingesetzt, z. B. bei der Steuerung von Roboterarmen und Maschinen, und ist auch in einigen Transportfahrzeugen zu finden. Es stützt sich auf den CAN-Bus als Transportprotokoll für die Nachrichten, es können aber auch andere Protokolle verwendet werden.

3.2.2 CAN-Bus

Die in Kapitel 3.2.1 beschriebenen Protokolle und Standards basieren auf dem CAN-Bus-Standard. Dieses Kapitel enthält eine Übersicht über dieses serielle Bussystem.

Das CAN-Bus-Protokoll belegt im OSI-Referenzmodell die Schichten 1 und 2. Das heißt, es dient sowohl der physikalischen Übertragung von Bits als auch der Redundanzprüfung, um die Konsistenz der Daten zu gewährleisten.

Die Datenübertragung mit dem CAN-Bus erfolgt durch Senden von Datenpaketen von einem Knoten in den Bus. Alle anderen Knoten am Bus empfangen dann die Datenpakete und entscheiden, ob die Nachricht von ihnen verarbeitet oder verworfen werden soll. Diese Pakete werden als Frames bezeichnet und bestehen aus verschiedenen Feldern von Bits. Abbildung 3.1 zeigt einen Standard-Daten-Frame. Einige Felder der Daten-Frames sind in Tabelle 3.1 beschrieben.

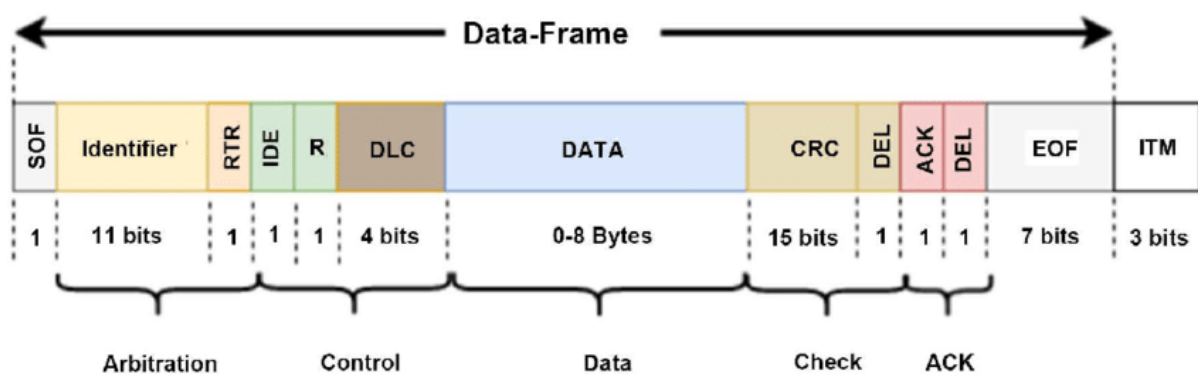


Abbildung 3.1 Standard CAN Data Frame

Feldbezeichnung	Bits	Beschreibung
Start-of-Frame (SOF)	1	Zeigt anderen Knoten an, dass eine Nachricht folgen wird
Identifier (ID)	11	Enthält die Knotenadresse im Bus. In diesem Feld wird die Priorität der Nachricht angegeben. Nachrichten mit niedrigeren ID-Nummern haben eine höhere Priorität
Remote Transmission Request (RTR)	1	Ermöglicht einem Knoten, Daten von anderen Knoten anzufordern
Identifier Extension Bit (IDE)	1	Gibt an, ob ein 29 Bit langer Identifier verwendet wird
Data Length Code (DLC)	4	Definiert die Anzahl der Bytes im Datenfeld (0-8)
Data Field (DATA)	0-64	Sequenz von Bytes, die die Nutzdaten enthalten
Cyclic Redundance Check (CRC)	16	Sorgt für die Datenintegrität der gesendeten Nachricht
End-of-Frame (EOF)	7	Signalisiert das Ende der CAN-Nachricht

Tabelle 3.1 CAN-Bus-Felder und ihre Bedeutung

Die physikalische Übertragung erfolgt über einen 2-Draht-Bus, die als CAN Hi und CAN Lo bezeichnet werden. Die Spannung zwischen den beiden Drähten wird als $V_{\text{diff}} = V_{\text{CAN HI}} - V_{\text{CAN LO}}$ bezeichnet. Wenn $V_{\text{diff}} = 2\text{V}$ beträgt, befindet sich der Bus in einem dominanten Zustand, was für einen CAN-Controller eine logische 0 bedeutet. Wenn die Spannung $V_{\text{diff}} \leq 0$ ist, befindet sich der Bus in einem rezessiven Zustand, was eine logische 1 bedeutet. Ein Zusammenhang zwischen den logischen Zuständen und den Busdrahtspannungen ist in Abbildung 3.2 dargestellt.

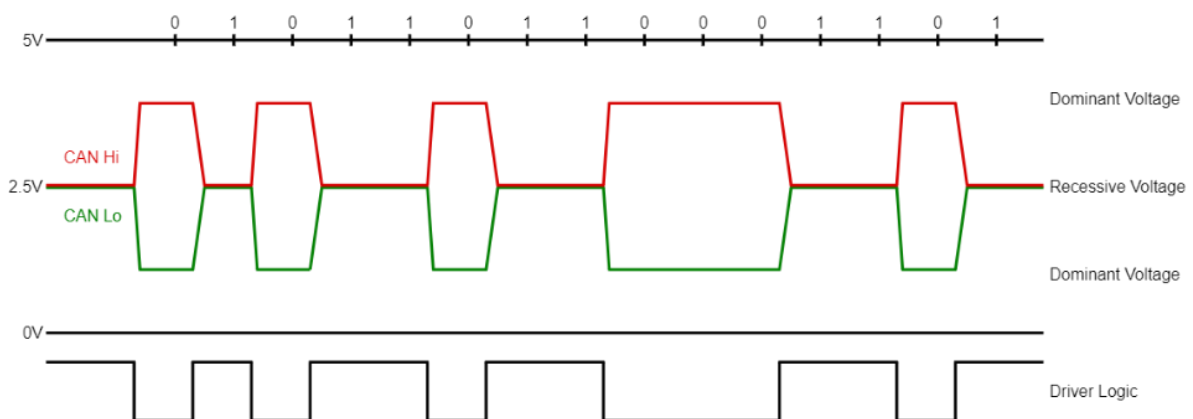


Abbildung 3.2 Zusammenhang zwischen logischen Zuständen und CAN-Bus-Adern Spannungen

3.3 Open Systems Interconnection model

Das Open Systems Interconnection-Modell (OSI) ist ein Konzept zur Charakterisierung und Vereinheitlichung von Kommunikationsfunktionen. Die Hauptaufgabe des Modells ist die Aufteilung des Datenflusses in sieben Abstraktionsschichten, um Netzwerkprotokolle zu vereinheitlichen und das Netzwerkmanagement zu vereinfachen. Dies umfasst die physikalische Umsetzung der Bitübertragung bis hin zur Nutzung der Daten in einer Anwendung. Eine Beschreibung der einzelnen Schichten ist in der Tabelle 3.2 angegeben⁵.

Schicht	Protokolle	Beschreibung
7 Application	HTTP MQTT	High-Level-APIs, einschließlich Ressourcenfreigabe, Remote-Dateizugriff
6 Presentation	DNS LDAP	Übersetzung von Daten zwischen einem Netzwerkdienst und einer Anwendung
5 Session	SMTP DHCP	Verwalten von Kommunikationssitzungen
4 Transport	TCP UDP	Zuverlässige Übertragung von Datensegmenten zwischen Punkten in einem Netzwerk
3 Network	IP IPsec	Strukturierung und Verwaltung eines Netzwerks mit mehreren Knoten
2 Data link	Ethernet WLAN CAN Bus	Zuverlässige Übertragung von Datenframes zwischen zwei über eine physikalische Schicht verbundenen Knoten
1 Physical	Token Ring CAN Bus	Übertragung und Empfang von Rohbitströmen über ein physikalisches Medium

Tabelle 3.2 Schichten des OSI-Modells

3.4 Technologien zur mobilen Datenübertragung

Im Vergleich zum Anfang des Jahrhunderts gibt es heute viel mehr Möglichkeiten, sich über Mobilkommunikation mit dem Internet zu verbinden. In den letzten Jahren hat sich das Internet der Dinge von einem Modewort zu einem Antrieb für viele Industrien entwickelt, und der Telekommunikationsmarkt hat darauf mit immer mehr Optionen für die Verbindung von Geräten geantwortet. Einen kurzen Überblick über die kommerziellen Mobilfunkprotokolle zeigt Tabelle 3.3.

⁵ ISO 7498-4.

Drahtlose Technologie	Inbetriebnahme	Maximale Datenrate	Kommentare
	Auslaufphase		
1G	1979	-	Analoge Datenübertragung
2G EDGE	1991	384 KBit/s	Fallback-Netzwerk für Voice-Kommunikation
	2017-2022		
3G HSPA+	2001	56 Mbit/s	Netzwerk in Deutschland wird am 30.06.2021 ausgeschaltet
	2019-2025		
4G LTE-A	2009	1 Gbit/s	Aktueller Standard der Mobilfunktechnologien
	-		
5G	2019	10 Gbit/s	Großes Breitband, aber begrenzte Abdeckung
	-		
6G	2030	~1 Tbit/s ⁶	Wird zurzeit geforscht
	-		
LTE Cat M1	2016	1 Mbit/s	Full-Duplex Datenübermittlung Nahtlose Übergabe der Mobilfunkzelle ⁷ ca.100% Abdeckung in Deutschland ⁸
	-		
LTE Cat NB1	2016	250 KBit/s	Half-Duplex Datenübermittlung Härtere Übergabe der Mobilfunkzelle ⁹
	-		
LoRaWAN	2015	50 KBit/s	Half-Duplex Datenübermittlung Gut geeignet für stationäre Sensoren
	-		

Tabelle 3.3 Vergleich der Netzwerktechnologien für Mobilfunkkommunikation

Es ist zu erkennen, dass ca. alle 10 Jahre ein Generationswechsel in der Mobilfunktechnologie stattfindet, der bei der Konzeption eines Systems, das drahtlos kommunizieren soll, berücksichtigt werden muss. Die Auswahl der Hardware muss so erfolgen, dass sie auch in Zukunft mit der lokalen Übertragungstechnik kompatibel sein muss.

⁶ Yang et al. 2019.

⁷ Telekom 2019.

⁸ Telekom 2021.

⁹ Telekom 2019.

Die letzten drei Zeilen der Tabelle 3.3 sind Netzwerktechnologien, die extra für IoT-Anwendungsfälle entwickelt wurden. Diese Technologien haben spezifische Anwendungsfälle, für die sie besser geeignet sind.

LTE Cat NB1 ist gut geeignet, um Daten von Sensoren zu übertragen, die sich physisch nicht bewegen, um Antennenübergaben und Neuverbindungen zu vermeiden.

LTE Cat M1 ist eine besser geeignete Technologie für Systeme mit häufigem Standortwechsel und mit geringem bis mittlerem Datenübertragungsvolumen. Die Vollduplex-Datenübertragung sorgt zudem für eine schnellere Kommunikation, da Pakete gleichzeitig empfangen und gesendet werden können.

LoRaWAN ist ein Low-Power-Wide-Area-Protokoll, das einen geringeren Energieverbrauch als LTE Cat M1 hat, jedoch geringere Datenübertragungsraten. Es ist besser für Anwendungen mit sehr begrenzten Energieressourcen geeignet.

3.5 Internet-Kommunikationsprotokolle

3.5.1 MQTT

MQTT ist ein leichtgewichtiges und effizientes Kommunikationsprotokoll, das nach der Publish-Subscribe-Architektur arbeitet¹⁰. Diese Architektur ist in Abbildung 3.3 dargestellt.

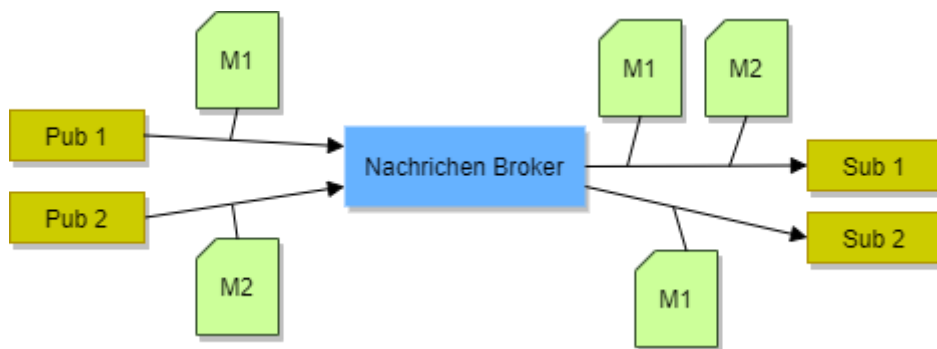


Abbildung 3.3 Übersicht der Client-Broker Architektur der MQTT Protokoll

Die Publisher, die im Abbildung 3.3 als Pub 1 und Pub 2 bezeichnet sind, senden Nachrichten an den MQTT-Broker. Die Nachrichten, im Abbildung 3.3 als M1 und M2 dargestellt, enthalten sowohl ein Topic als auch eine Payload.

¹⁰ OASIS 2014.

Das Topic ist eine Zeichenkette, die die Hierarchie der Nachricht im Broker definiert, zum Beispiel Fahrzeug/1234/Batterie. Die Payload enthält die Daten, die auf der beim Topic definierten Hierarchie platziert werden sollen.

Es gibt keine Einschränkungen für das Format der Payload. Es kann definiert werden, ob es sich um ein nicht spezifiziertes Array von Bytes oder eine UTF-8 kodierte Nachricht handelt. Ihr Inhalt könnte also eine Zeichenfolge, eine Zahl, ein JSON-Objekt oder ein einzelnes Byte sein.

Die Subscriber, im Abbildung 3.3 als Sub 1 und Sub 2 dargestellt, erstellen ein Abonnement für ein Topic auf dem Broker. Ein Abonnement bedeutet, dass die Subscriber die aktuelle und alle folgenden Nachrichten dieses Topics erhalten werden. Zur Veranschaulichung werden in Abbildung 3.4 zwei Nachrichten betrachten, die Daten über die Positionen und den aktuellen Ladezustand eines Elektrofahrzeugs enthalten.

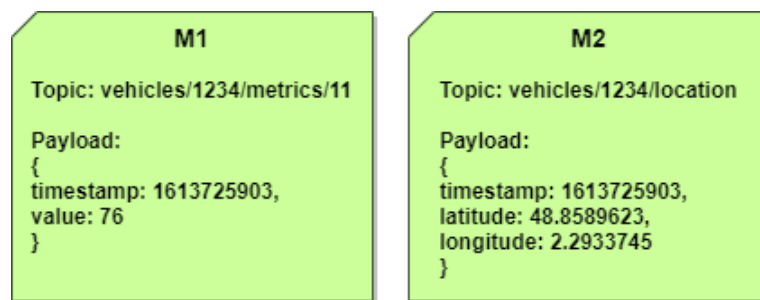


Abbildung 3.4 Beispiel MQTT-Nachrichten des Standorts und des Ladezustands des Fahrzeugs

Das Topic der Nachricht M1 in Abbildung 3.4 definiert, dass sich diese Nachricht auf das Fahrzeug mit der ID 1234 bezieht und die Metrik die ID 11 hat. Analog dazu bezieht sich die Nachricht M2 auf den aktuellen Standort des Fahrzeugs mit der ID 1234. In beiden Nachrichten ist die Payload ein JSON-Objekt, das einen Zeitstempel und den Wert der Metrik enthält.

Die Verwendung von MQTT als IoT-Protokoll für Fahrzeuge wurde in ähnlichen Anwendungsfällen vorgeschlagen¹¹. Es wurde auch zur Erfassung von Telemetriedaten und zur Steuerung eines kleinen Roboterautos verwendet¹². Außerdem ist dieses einfache Protokoll für viele Sprachen und Frameworks leicht

¹¹ Dhall und Solanki 2017.

¹² Panchi et al. 2018.

zu implementieren und bietet sich als gutes komponentenübergreifendes Integrationsprotokoll an.

3.5.2 HTTP

HTTP funktioniert als Anfrage-Antwort-Protokoll im Client-Server-Computermode. Es ist das gängige Protokoll für die Bereitstellung von Webinhalten wie z. B. Websites. und wird auch in vielen IoT-Anwendungen verwendet.

Eine Übersicht über die Architektur zeigt Abbildung 3.5. Ein Client startet die Verbindung, indem er eine Anfrage an den Server sendet. Die Anfrage, dargestellt als Req, enthält eine fest definierte Struktur. Diese Anfrage wird vom Server verarbeitet und die Antwort, dargestellt als Res, wird an den Client zurückgeschickt.

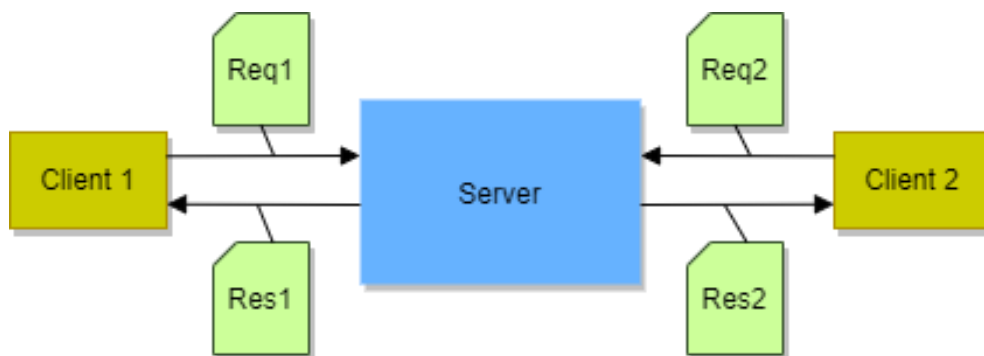


Abbildung 3.5 Darstellung der Request-Response-Architektur von HTTP

Einen tieferen Einblick in den Aufbau der Anfragen und Antworten liefert Abbildung 3.6. Die Anfrage wird durch eine Anfragezeile, ein Header und einen optionalen Body definiert. Die Anfragezeile enthält die HTTP-Methode und den URI. Die Header enthalten Informationen über die Verbindung, Caching, Authentifizierung, etc. Der optionale Body der Anfrage ist für POST-Anfragen relevant und könnte z.B. ein JSON-Objekt sein. Die wichtigsten HTTP-Methoden sind in Tabelle 3.4 aufgeführt.

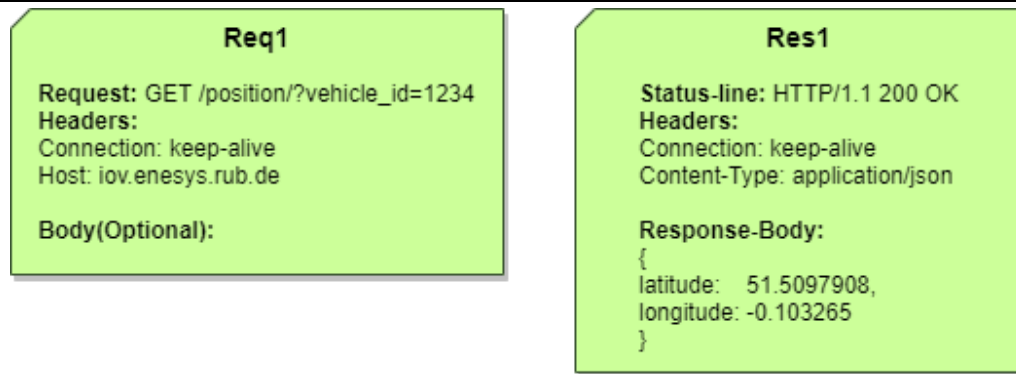


Abbildung 3.6 Beispiel HTTP-Anfrage und -Antwort

HTTP-Methode	Beschreibung
GET	Wird verwendet, um Daten vom Server abzurufen
POST	Wird verwendet, um Daten an den Server zu senden, z. B. den Inhalt eines ausgefüllten Formulars oder ein JSON-Objekt
DELETE	Wird verwendet, um Daten vom Server zu löschen

Tabelle 3.4 Einige HTTP-Methoden und ihre Beschreibung

3.5.3 JSON

JSON ist ein offenes Standard-Dateiformat¹³, das häufig zur Übertragung von Daten zwischen Anwendungen verwendet wird. Es ist nicht von einer Programmiersprache abhängig, so dass es eine weit verbreitete Unterstützung des Formats gibt.

Ein JSON-Objekt besteht aus Schlüsselpaarwerten, die durch Kommas getrennt sind. Ein Objekt wird durch { } abgegrenzt. Ein JSON-Array besteht aus einer Folge von Objekten oder Werten, die durch Kommas getrennt sind. Ein Array wird schließlich durch [] abgegrenzt. Ein Beispiel für ein JSON-Array ist in Abbildung 3.7 dargestellt.

```

1  [
2    {
3      "id": "1236", "model": "Ampera", "make": "Opel", "iconColor": "white"
4    },
5    {
6      "id": "1234", "model": "i-MiEV", "make": "Mitsubishi", "iconColor": "red"
7    },
8    {
9      "id": "1235", "model": "iOn", "make": "Peugeot", "iconColor": "blue"
10   }
11 ]

```

Abbildung 3.7 Beispiel für ein JSON-Array von Fahrzeugeigenschaftsobjekten

¹³ Ecma International 2017.

3.6 Sichere Übertragung von Daten

Das Internet ist ein intrinsisch unsicheres Medium, über das Daten von Knoten zu Knoten bzw. von Computer zu Computer übertragen werden. Um Daten in einem offenen Netzwerk sicher zu übertragen, ist eine Verschlüsselung notwendig.

Das am weitesten verbreiteten Protokoll ist das Transport Layer Security (TLS) oder sein Vorgänger Secure Sockets Layer (SSL). Die Sicherheit oder Vertraulichkeit der Verbindung wird durch die Verwendung eines symmetrischen Schlüsselalgorithmus zur Verschlüsselung der Daten erzwungen. Die Authentizität von Client und Server kann durch die Verwendung von SSL-Zertifikaten nachgewiesen werden.

3.6.1 Symmetric-Key-Verschlüsselung

Bei der Symmetric-Key-Verschlüsselung haben beide Teilnehmer den gleichen Schlüssel, mit dem eine Nachricht entweder verschlüsselt oder entschlüsselt werden kann. Dieser Vorgang ist in Abbildung 3.8 dargestellt. Der Client möchte eine Nachricht M1 an den Server senden, also verschlüsselt er sie mit einem Schlüssel, den sowohl er als auch der Server haben. Der Server kann dann die Nachricht mit demselben Schlüssel entschlüsseln. Der mathematische Aufwand zum Entschlüsseln der Nachricht ist so groß, dass es praktisch unmöglich ist, Nachrichten zu entschlüsseln, ohne den Schlüssel zu haben. Damit ist sichergestellt, dass die Daten sicher durch das Internet übertragen werden.

Das Problem ist, wie man diesen Schlüssel sicher mit beiden Teilnehmern teilen kann. Eine Möglichkeit wäre die physische Übergabe des Schlüssels durch eine vertrauenswürdige Person, aber das wird unpraktisch, wenn es viele Teilnehmern oder Server gibt oder der Schlüssel geändert werden muss. Eine Lösung für dieses Problem ist die Verschlüsselung mit asymmetrischen Schlüsseln, oder Public-Key-Verschlüsselung, die in Kapitel 3.6.2 beschrieben wird.

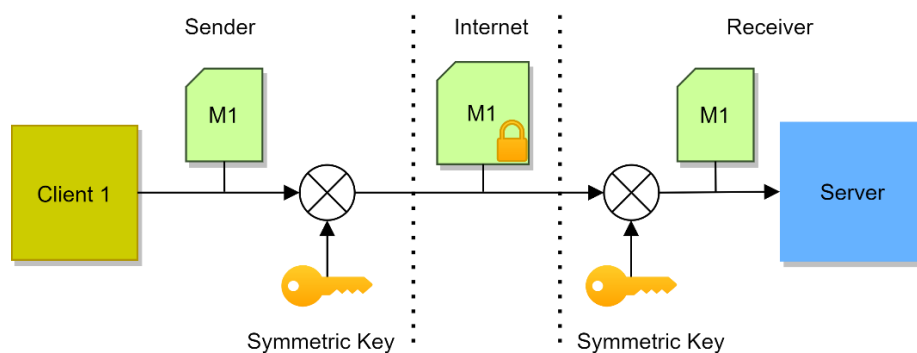


Abbildung 3.8 Verschlüsselung mit symmetrischem Schlüssel

3.6.2 Asymmetric-Key-Verschlüsselung

Bei der Asymmetric-Key-Verschlüsselung hat jeder Teilnehmer ein Schlüsselpaar bestehend aus einem öffentlichen und einem privaten Schlüssel. Beide Schlüssel sind mathematisch so aufeinander bezogen, dass die Entschlüsselung von Daten, die mit dem öffentlichen Schlüssel verschlüsselt wurden, nur mit dem privaten Schlüssel möglich ist.

In Abbildung 3.9 ist eine sichere Kommunikation vom Client zum Server dargestellt. Analog zur symmetrischen Verschlüsselung möchte der Client eine Nachricht M1 an den Server senden. Dazu verwendet er den öffentlich verfügbaren Schlüssel des Servers, den Public-Key, um die Nachricht zu verschlüsseln.

Da der Server der einzige andere Computer ist, der den privaten Schlüssel in Bezug auf den verwendeten Public Key besitzt, können die Nachrichten nur im Server entschlüsselt werden. Da der Inhalt von M1 für Mithörer im Internet geheim ist, könnte der Symmetric Key durch diese Nachricht gesendet werden. Dies ist wichtig, da die symmetrische Kryptographie weniger rechenintensiv ist als die asymmetrische.

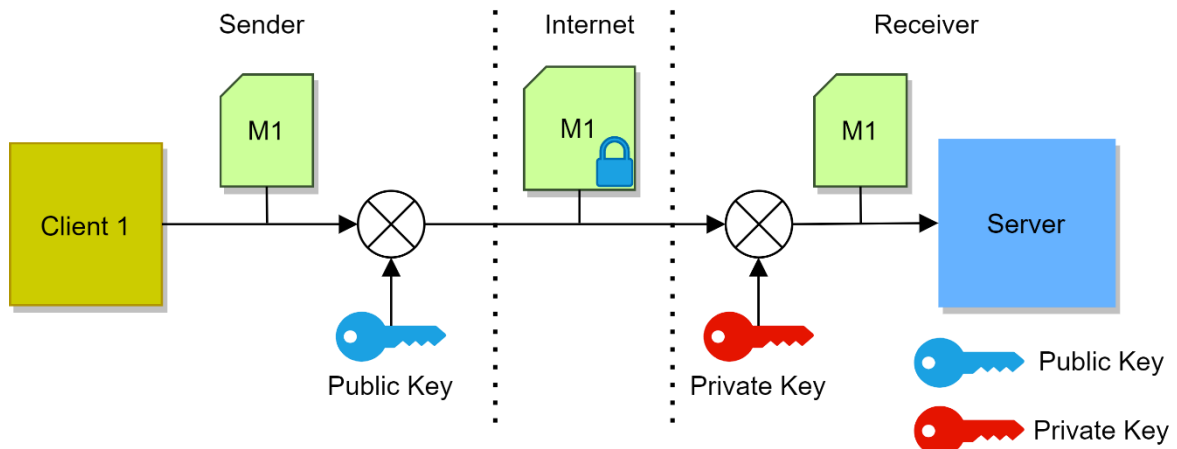


Abbildung 3.9 Verschlüsselung mit asymmetrischem Schlüssel

3.7 Microservices

Mit der Entwicklung einer digitalen Anwendung wird die Komplexität der verschiedenen Komponenten und die Integration zwischen ihnen immer umfangreicher. Um diese Komplexität zu reduzieren, werden die Funktionalitäten der Anwendung aufgeteilt und von unabhängigen, kleineren Diensten ausgeführt. Dies wird als Microservices-Architekturansatz bezeichnet.

Ein Beispiel: Eine Backend-Anwendung verarbeitet HTTP-Anfragen und liest oder schreibt Daten aus einer Datenbank. Währenddessen stellt der Frontend-Dienst einer Web-Anwendung die Visualisierung der Daten zur Verfügung. Beide Dienste sind in unterschiedlichen Programmiersprachen geschrieben und haben unterschiedliche Aufgaben. Aus diesem Grund werden sie in unterschiedlichen Umgebungen ausgeführt und kommunizieren miteinander über High-Level-APIs, wie HTTP. Ein weiterer Vorteil dieser Architektur ist, dass Änderungen an einem oder mehreren Diensten vorgenommen werden können, ohne dass die gesamte Anwendung neu implementiert oder neu gestartet werden muss.

3.7.1 Docker

Docker ist eine Anwendung zur Virtualisierung und Verwaltung von Software in Paketen, die Container genannt werden. Container sind voneinander getrennt und sind eigenständige Dienste, die alle notwendigen Bibliotheken und Konfigurationen in sich tragen. Dies bedeutet, dass keine externen Bibliotheken oder Abhängigkeiten auf dem Host-Rechner installiert werden müssen, damit Container ausgeführt werden können. Dies hat den Vorteil, dass Container unabhängig vom Host-Betriebssystem sind und dass Aktualisierungen und Änderungen, die am Host vorgenommen werden, die laufenden Dienste nicht beeinträchtigen.

Damit ein Container läuft, ist ein Image notwendig. Ein Image ist eine schreibgeschützte Vorlage, die zum Erstellen von Containern verwendet wird. Images können entweder lokal mittels einer Dockerfile erstellt oder von öffentlichen oder privaten Repositorien, den Registries, heruntergeladen werden. Eine genauere Beschreibung und Definition eines Dockerfiles wird in Kapitel 7.2.1 gegeben

3.7.2 Orchestrierung von Diensten

Für Anwendungen mit vielen Diensten muss eine standardisierte Verwaltungsstruktur verwendet werden. In vielen Fällen gibt es Abhängigkeiten zwischen den Diensten. So muss z. B. eine Datenbank laufen, bevor eine Backend-Anwendung SQL-Abfragen an sie stellen kann. Die Backend-Anwendung muss warten, bis die Datenbank hochgefahren ist und läuft, bevor sie versucht, eine Verbindung herzustellen. Es ist auch eine gute Praxis, ein definiertes Docker-Netzwerk zu haben, in dem Container mit sich selbst kommunizieren und von anderen Netzwerken isoliert sind.

Um diese Verwaltung zu erleichtern, wird ein Tool namens Docker Compose verwendet. Damit es funktioniert, sollten alle Dienste in einem YAML-Dateiformat definiert sein. Eine Beispiel-YAML-Datei mit der Definition von zwei Diensten und einem Netzwerk ist in Listing 3.1 dargestellt. Außerdem gibt es Schlüsselwörter, die den Containern zusätzliche Eigenschaften zuweisen. Zum Beispiel definiert das Schlüsselwort *restart*, wann der Container sich initialisieren soll, falls der Host-Computer neu startet. Eine genauere Übersicht über die in dieser Arbeit verwendeten Schlüsselwörter ist in Kapitel 7.2.2 aufgeführt.

```
# Datei: docker-compose.yml
version: '3.7'
services:
  iov-backend:
    restart: always
    image: iov-backend:1.0.0
    ports:
      - 8080:8080

  iov-frontend:
    restart: always
    depends_on:
      - iov-backend
    image: iov-frontend:1.0.0
```

Listing 3.1 Beispiel für die Datei docker-compose.yml, die zwei Dienste definiert

3.8 Microcontroller

Um die Anforderungen an ein Elektronikmodul mit geringem Energieverbrauch für den Einsatz in einem Elektroauto zu erfüllen, sind Mikrocontroller eine gute Möglichkeit.

Das OVMS-Modul hat gezeigt, dass eine Lösung mit einer ESP32-MCU genügend Rechenleistung bietet, um alle Systemanforderungen zu erfüllen. Allerdings ist das Programmier-Framework des OVMS-Systems sehr umfangreich. Für eine schnellere und optimierte Entwicklung und Bereitstellung des Systems wurde ein neues Framework für die Mikrocontroller-Entwicklung gewählt.

MicroPython ist eine leichtgewichtige und effiziente Python 3-Implementierung, die für die Ausführung auf Mikrocontrollern und anderen hardwarebeschränkten Geräten optimiert ist. Das Framework unterstützt derzeit viele Mikrocontroller, darunter auch den ESP32. Eine Lösung mit integriertem Mobilfunkmodem wird von

der Firma Pycom angeboten. Das GPy-Modul bietet die Standard-WLAN- und Bluetooth-Konnektivität des ESP32 und ein integriertes LTE-Modem zu einem günstigen Preis.

Obwohl Python eine Sprache für schnelles Prototypenbau und Entwicklung ist, bietet es einige Nachteile für den Aufbau eines robusten Elektronikmoduls. Da Python eine interpretierte Sprache ist, gibt es keine vorherige Kompilierung, um auf Fehler zu prüfen, so dass beim Aufruf einer nicht definierten Funktion zur Laufzeit eine Exception geworfen wird und das Programm abbricht.

Einige Lösungen für dieses Problem sind die Verwendung von Unit-Testing, um sicherzustellen, dass alle Klassen und Funktionen korrekt aufgerufen werden und die Ergebnisse wie erwartet sind. Eine andere Lösung wäre ein robusteres System zum Lesen von Daten aus dem Can-Bus und ein weiteres System, dann mit Micropython, zur Kommunikation und zum Senden von Daten.

4 Entwicklung eines neuen Systems

Inspiziert durch das in Unterkapitel 3.1.1 beschriebene OVMS-Projekt und durch die Notwendigkeit der Einfachheit in seiner Entwicklung, wurde ein neues System als Lösung vorgeschlagen. Dieses Kapitel listet die Anforderungen an ein neues Elektronikmodul auf und gibt einen Überblick über das Gesamtsystem.

4.1.1 Anforderungskatalogs für das Elektronikmodul

Um den Entwurf und die Entwicklung eines neuen Moduls zu strukturieren, werden die Anforderungen an das System in Tabelle 4.1 definiert.

	Anforderung	Beschreibung
1	Energie-Effizienz	Geringe Stromaufnahme während des Systembetriebs
2	Mobilfunkkommunikation	Kommunikation mit dem Internet über Mobilfunk
3	Robuste Übertragung von Daten	Minimierter Datenverlust bei der Übertragung
4	Effizienz der Datenübertragung	Geringe Nutzung von mobilen Daten
5	Sicherheit bei der Datenübertragung	Daten sollten nicht von nicht autorisierten Personen gelesen werden können
6	Erfassung von Standortdaten	Aktueller Standort des Fahrzeugs muss ermittelt werden
7	Sammeln von Fahrzeugdaten über CAN-Bus	Modul muss CAN-Bus-fähig sein

Tabelle 4.1 Anforderungen des neuen Systems

Um die gesammelten, gefilterten und bearbeiteten Daten in der Cloud übertragen zu können, es müssen kommunikationsschnittstellen angelegt werden. Der Pycom Modul GPy würde aus diesem gründen gewählt, da es schon ein LTE Modem integriert. Außerdem enthält das Modul einen integrierten CAN-Bus-Controller und UART-Kommunikationsports, so dass die Kommunikation mit dem Fahrzeug und dem Standort gewährleistet ist. Mit der Auswahl dieses Low-Energy-Mikrocontroller-Moduls werden die Anforderungen 1, 2, 6 und 7 erfüllt.

Die erste Generation vernetzter Autos hatte Kommunikationslösungen, die auf SMS und HTTP basierten. Beide Protokolle sind sehr bandbreitenintensiv und wurden nicht für das kontinuierliche Streaming von Daten konzipiert.

Ein besser geeignetes Protokoll für das Streaming von Telemetriedaten ist MQTT. Dieses Protokoll wurde 1999 für die Überwachung von Ölpipelines über Satellitendatenverbindungen entwickelt und ist für Umgebungen mit sehr begrenzter Bandbreite und Hardwarebeschränkungen ausgelegt.

Der Fall des vernetzten Autos stellt eine ähnliche technische Herausforderung dar, da die Internetverbindung nicht immer verfügbar ist, während sich das Auto bewegt, und die Nutzung der mobilen Daten auf ein Minimum beschränkt werden muss.

Ein weiterer positiver Aspekt der Verwendung von MQTT ist die Reduzierung der Netzwerknutzung. Ein Report von Google Cloud hat gezeigt, dass nach dem Aufbau der Verbindung mit dem Server die Größe der gesendeten Nachricht im Vergleich zu HTTP um fast das Zehnfache reduziert wird¹⁴. Die Verschlüsselung von Daten ist auch durch die Einrichtung von Broker und Client mit SSL-Zertifikaten möglich. Durch die Wahl von verschlüsseltem MQTT als Kommunikationsprotokoll werden die Anforderungen 3, 4 und 5 erfüllt.

4.1.2 Systemübersicht

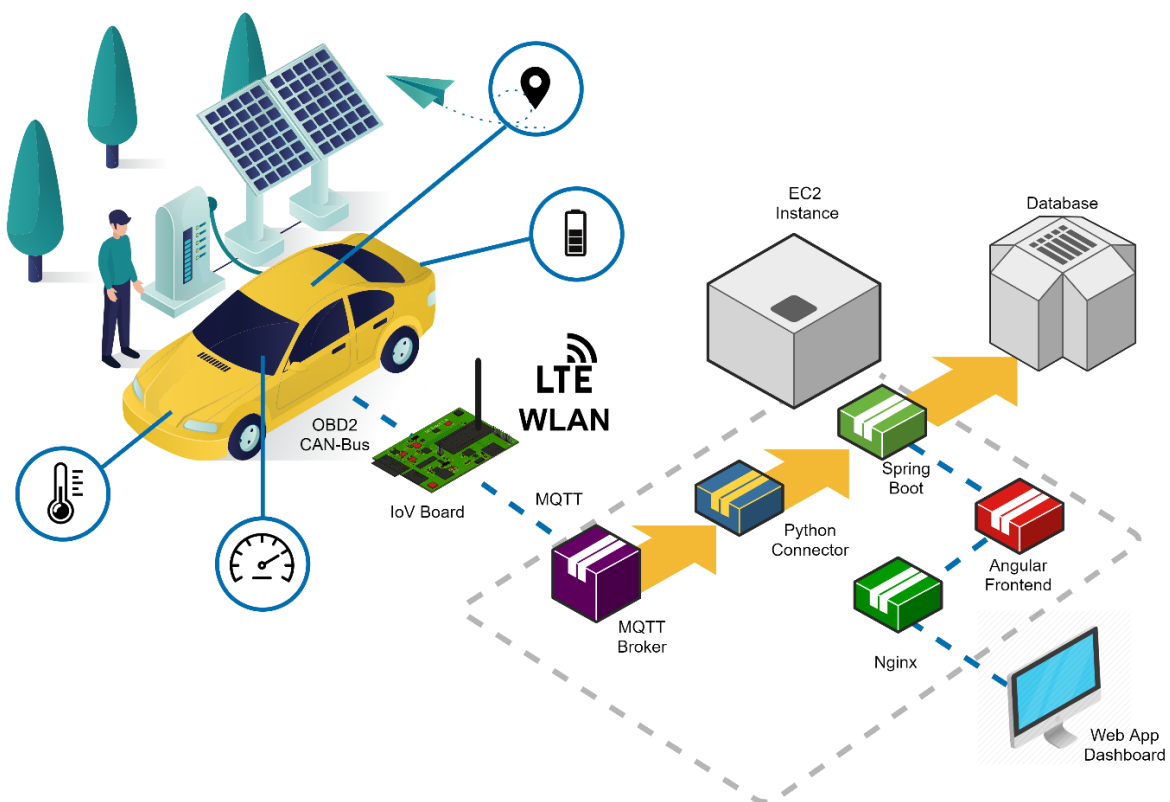


Abbildung 4.1 Allgemeine modularisierte Systemübersicht

¹⁴ Wang 2018.

Abbildung 4.1 zeigt das gesamte System in einer modularisierten Form. Der Datenpfad der Fahrzeugsdaten beginnt mit dem Auslesen der CAN-Bus-Frames durch das neu entwickelte Elektronikmodul namens IoV (Internet of Vehicles).

Der nächste Schritt ist die Datenübertragung über das Internet zu einer Cloud-Anwendung. Der Eingangspunkt für die Fahrzeugdaten ist ein MQTT-Broker. Nachrichten werden verschlüsselt empfangen und für andere Teilnehmer zur Verfügung gestellt.

Eine in Python geschriebene Connector-Anwendung übersetzt die eingehenden MQTT-Nachrichten in HTTP-POST-Anfragen und leitet sie an die Backend REST-Anwendung weiter. Die HTTP-Anfrage wird dann verarbeitet und die Daten werden in einer Datenbank gespeichert.

Benutzer können dann über eine Web-Anwendung auf die gespeicherten Daten zugreifen. Die gesamte Kommunikation zwischen dem Webbrowser und der Cloud ist durch SSL-Zertifikate gesichert.

5 Design und Aufbau der Hardware des Moduls

Um alle in Unterkapitel 4.1.1 beschriebenen Anforderungen an das System zu erfüllen, wurde die Entwicklung einer neuen Hardware für notwendig gehalten. In diesem Kapitel wird das gesamte Hardwaresystem beschrieben.

5.1 Gesamt-Hardware-Schaltplan

Das Schaltungsdesign basierte auf dem OVMS-Design, allerdings musste die Leiterplatte von Grund auf neu entwickelt werden, da von den Entwicklern von OVMS kein Schaltplan zur Verfügung gestellt wurde. Einige Teile der Schaltung, wie das LTE-Modem, sind bereits auf dem Pycom-GPy-Modul eingebettet, so dass die Schaltung noch weiter vereinfacht wird.

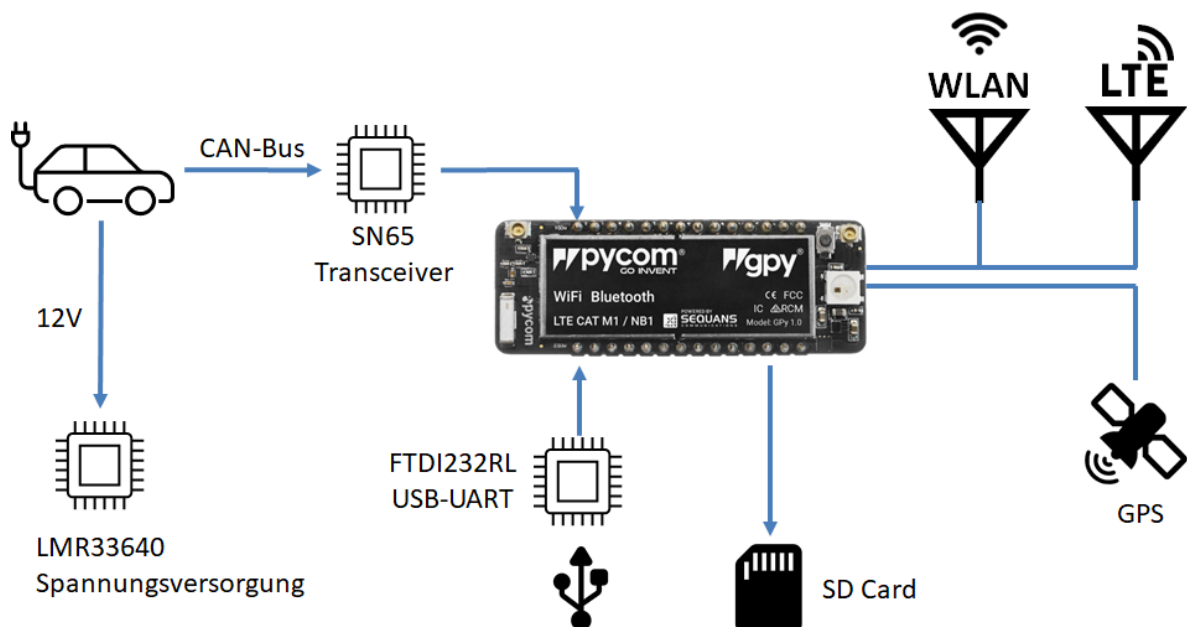


Abbildung 5.1 Vereinfachte Übersicht über das Hardware-Design

Der in Abbildung 5.1 dargestellte vereinfachte Schaltplan stellt die Komponenten der entworfenen und gefertigten Leiterplatte dar. Der EAGLE-Schaltplan liegt in digitaler Form vor und befindet sich im Ordner *Daten/1_Hardware_PCB/Eagle_Schematic* der Dokumentation. Die einzelnen Schaltungskomponenten sind ebenfalls im Anhang dieser Arbeit dargestellt.

5.2 Spannungsversorgung

Diese Schaltungsteil wird im Schaltplan als "1 - POWER SUPPLY" bezeichnet. Die Schaltung versorgt das Modul mit 5 V und bis zu 4 A Strom durch den integrierten

Konverter LMR33640. Die Schaltung folgte den Designrichtlinien aus dem Datenblatt. Die Frequenz des Konverters beträgt 1000kHz.

Die Eingangsspannung kann im Bereich von 4.5 V bis zu 36 V betragen. Es ist möglich, eine 12V-Stromversorgung und eine 5V-USB-Versorgung gleichzeitig anzuschließen. Eine Diodenschaltung verhindert, dass das 12-V-Potenzial die 5-V-USB-Stromquelle beschädigt.

5.3 CAN-Transceiver

Dieser Schaltungsteil wird im Schaltplan als "2 - CANBUS" bezeichnet. Es ermöglicht den gleichzeitigen Anschluss von bis zu 3 Can-Bussen an den Mikrocontroller.

Das Pycom Microcontroller-Modul hat einen integrierten CAN-Controller. Einer der SN65 CAN-Transceiver wird direkt mit diesem Controller verbunden. Die anderen beiden Transceiver sind mit CAN-SPI-Schnittstellen verbunden, die wiederum über einen SPI-Bus mit dem Mikrocontroller verbunden sind. Obwohl die Schnittstellen derzeit nicht verwendet werden, sind sie bereits mit dem Microcontroller verbunden, so dass sie in zukünftigen Projekten verwendet werden können.

5.4 USB-UART-Schnittstelle

Dieser Schaltungsteil wird im Schaltplan als "3 – USB-UART" bezeichnet. Um eine serielle Kommunikation zwischen dem Mikrocontroller und einem Computer zu ermöglichen, wird die integrierte Schaltung FT232RL verwendet. Die Schaltung folgte den Designrichtlinien aus dem Datenblatt.

Zum Schutz sowohl des Systems als auch des angeschlossenen Computers vor Überspannungen und elektrostatischen Entladungen wurde eine SP05-Diodenzeile hinzugefügt.

5.5 Pycom GPy Microcontroller

Das GPy-Modul präsentierte sich als interessantes Mikrocontrollermodul. Darüber hinaus gibt es mehrere Bibliotheken für die Internetkommunikation, serielle Kommunikationsschnittstellen und die meisten Python-Standardbibliotheken. Allerdings gibt es Nachteile in Bezug auf Geschwindigkeit und Zuverlässigkeit im Vergleich zu Firmwares, die in C oder anderen hardwarenahen Programmiersprachen kompiliert wurden. Dieser Schaltungsteil wird im Schaltplan als "4 - PYCOM GPy " bezeichnet.

5.6 Externe Komponenten und Verkabelung

Die Schaltungsteile, die externe Hardware unterstützen, sind im Schaltplan entsprechend gekennzeichnet. Darunter ist der SD-Card-Anschluss, der UART-Anschluss für das externe GPS-Modul und der DB9-Anschluss, der das System mit dem Auto verbindet, aufgelistet.

Das kommerziell erhältliche OBD2-DB9-Kabel musste neu verdrahtet werden. In der Tabelle 5.1 ist definiert, wie das Kabel anhand der Farben der Adern umverdrahtet wird.

Der linke Teil der Tabelle beschreibt, welcher OBD2-Steckerpin mit welchen Adern verbunden ist. Der rechte Teil der Tabelle macht das Gleiche, aber mit einem DB9-Stecker. Die mittlere Spalte gibt an, wie die Kabel wieder anzuschließen sind.

OBD2 (Car)				DB9 (Module)	
Pin	Name	Connection		Pin	Name
4	GND			3	GND
6	CAN +			7	CAN 0_H
14	CAN -			2	CAN0_L
16	Battery+			9	12V +

Tabelle 5.1 Umverdrahtungsschema des OBD2 - DB9 Kabels

5.7 Tasten für Reset, Bootloader und Safeboot

Das Pycom-Modul hat einen integrierten Reset-Taster, der mit dem Reset-Pin des ESP32 verbunden ist. Bei Verwendung in Kombination mit dem Bootloader-Taster kann neue Firmware hochgeladen werden. Die Reset-Taste kann auch gleichzeitig mit der Safeboot-Taste gedrückt werden, wodurch der Mikrocontroller in einer Python REPL-Umgebung gebootet wird. Dies ist in Fällen nützlich, in denen das Programm in eine Schleife gerät und nicht über das Terminal gestoppt werden kann.

6 Design und Aufbau der Software des Moduls

Dieses Kapitel erklärt den Aufbau des Programms und die Routinen, die im Mikrocontroller ablaufen. Einige der verwendeten Bibliotheken werden ebenfalls erläutert.

6.1 Flashen der Firmware

Der erste Schritt ist, den MicroPython-Interpreter in den ESP32 zu flashen. Im Falle des Pycom-Moduls ist die Firmware bereits standardmäßig installiert. Es kann jedoch notwendig sein, die Firmware zu aktualisieren. Dies ist mit einem von der Firma Pycom zur Verfügung gestellten Programm leicht möglich.

6.2 Verwendete Bibliotheken

Über die Standardbibliotheken hinaus mussten einige anwendungsspezifische Bibliotheken auf den Mikrocontroller geladen werden, die im Unterkapitel erläutert werden.

6.2.1 Uasyncio

Die uasyncio-Bibliothek, obwohl eine Standardbibliothek von Micropython, ist nicht auf der Pycom-Firmware vorhanden, daher muss sie manuell als Datei hinzugefügt werden. Diese Bibliothek hat eine ähnliche Rolle wie ein RTOS in einer C-basierten Firmware. Sie ermöglicht eine nicht blockierende, multitaskingfähige Architektur.

Die Hauptkomponenten der Architektur sind die Event-Loop und Tasks. Task ist eine Kapselung einer Funktion, die, wenn sie der Ereignisschleife hinzugefügt wird, ausgeführt wird, sobald die Event-Loop frei ist. Wenn die Funktion beendet, wird die Task aus der Event Loop entfernt.

Ein Implementierungsbeispiel ist in Listing 6.1 dargestellt. Voraussetzung ist, dass die Funktionen mit einem `async`-Schlüsselwort definiert sind, da sie sonst nicht von der Schleife ausgeführt werden können. Sobald das Event-Loop-Objekt definiert ist, kann die Methode `create_task()` aufgerufen werden, wobei das Argument die Funktion ist, die in der Schleife ausgeführt werden soll. Ähnlich wie bei anderen RTOS-Frameworks kann eine Endlosschleife erzeugt werden, indem der Funktionsinhalt innerhalb einer `while True:` Schleife platziert wird.

Die Verzögerung der Ausführung kann mit der `sleep`-Methode von `uasyncio` eingestellt werden. Sobald die Funktion ausgeführt wurde und sich im Wartezustand

befindet, können andere Funktionen von anderen Teilen des Programms aufgerufen und ausgeführt werden, so dass die Verzögerung nicht blockierend ist.

```
async def poll_gps_location():
    while True:
        location = GPS.read()
        await uasyncio.sleep(1)

async def scan_wlan_networks():
    while True:
        networks = WLAN.scan()
        await uasyncio.sleep(3)

loop = uasyncio.get_event_loop()
loop.create_task(poll_gps_location())
loop.create_task(scan_wlan_networks())
```

Listing 6.1 Beispiel für zwei endlose MicroPython-Routinen, die asynchron ausgeführt werden

6.2.2 CAN

Die CAN-Bibliothek ist in der Pycom-Firmware eingebaut. Sobald eine CAN-Controller-Instanz mit Baudrate und Pins definiert ist, kann eine Callback-Funktion ausgewählt werden, die bei jeder eingehenden Can-Bus-Nachricht aufgerufen wird. Eine Can-ID-Filter-Implementierung ist ebenfalls möglich.

6.2.3 UART

Ähnlich wie bei der CAN-Bibliothek muss auch hier eine Instanz der Klasse definiert werden. Allerdings müssen die Daten periodisch aus dem Eingangspuffer gelesen werden. In dieser Arbeit wurde diese Klasse verwendet, um die eingehenden GPS-Standortdaten zu lesen.

6.2.4 LTE und WLAN

Um die mobile Kommunikation über das LTE-M-Netz zu ermöglichen, enthält die LTE Class API-Methoden zum Ein- und Ausschalten der Station und zur Verbindung mit einem bestimmten Band und Zugangspunktnamen. Die WLAN-Klasse bietet ebenfalls die Kontrolle über die Station und hat ähnliche Methoden wie die LTE-

Klasse. Beide Bibliotheken bieten auch Methoden, um den aktuellen Verbindungs- und Stationszustand zu prüfen.

6.2.5 SSL

Die SSL-Bibliothek bietet eine Implementierung des Transport Layer Security (TLS)-Protokolls. Sie versieht die Kommunikation zwischen dem Mikrocontroller mit einer Sicherheitsschicht, indem sie die übertragenen Nachrichten verschlüsselt. Es wird auch von anderen Bibliotheken, wie z. B. MQTT, verwendet, um den Client mit dem Server durch digitale Zertifikate zu autorisieren.

6.2.6 MQTT

Der MQTT-Client ist in der umqtt-Bibliothek implementiert. Er stellt die notwendigen Methoden zum Verbinden, Veröffentlichen und Abonnieren von MQTT-Topics zur Verfügung. Es gibt auch Unterstützung für eine verschlüsselte Verbindung mit dem Broker über die SSL-Bibliothek.

6.2.7 GPS

Um die vom UART-GPS-Modul empfangenen NMEA-GPS-Sätze zu dekodieren, wird die Bibliothek micropyGPS verwendet. Eine Instanz der Klasse dieser Bibliothek enthält Eigenschaften über die aktuelle Position, Geschwindigkeit und Höhe.

6.3 Program structure

Die Dateien des Modulprogramms befinden sich im Verzeichnis *Daten/2_Software_Module/iov-pycom*. Das Programm ist in 5 Hauptordnern organisiert.

Im cert-Ordner sind die Zertifikate für die SSL-Kommunikation abgelegt. Der config-Ordner enthält Konfigurationsdateien, auf die in anderen Teilen des Programms zugegriffen wird, wie z. B. die bekannten WLAN-Access-Points, der aktuell verwendete Fahrzeugtyp, die Aktualisierungsraten für das GPS und die Kommunikationsprotokolle, die IP-Adresse des Cloud-Servers und weitere Parameter für die MQTT-Kommunikation.

Die extern hinzugefügten Bibliotheken befinden sich im Ordner libraries. Dies sind Bibliotheken, die sich auf die asynchrone Ausführung des Programms, das MQTT-Kommunikationsprotokoll und die GPS-Dekodierung beziehen. Der Rest des Programms ist im Ordner project organisiert. Auch hier gibt es Unterordner, die die

Dateien nach ihrer Aufgabe trennen. Dieses Kapitel wird sich auf die Funktionalitäten der im Verzeichnis `project` vorhandenen Klassen konzentrieren. Der Zusammenhang zwischen den Softwarekomponenten ist in Abbildung 6.1 dargestellt.

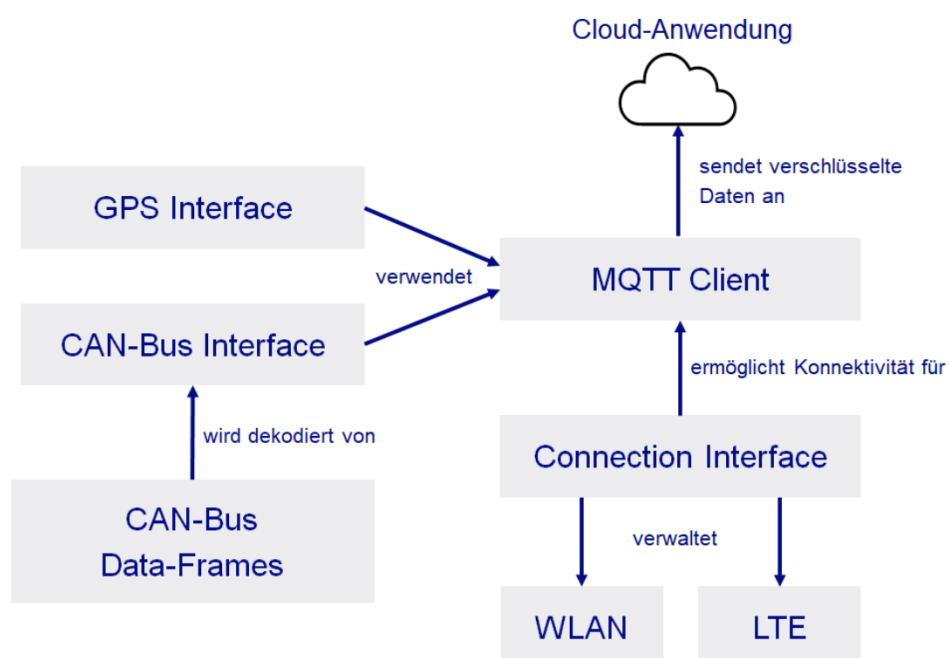


Abbildung 6.1 Struktur des Moduls Software

6.3.1 Boot- und Main-Dateien

Nach dem Booten sucht MicroPython nach einer `boot.py` und einer `main.py` Datei und führt sie in dieser Reihenfolge aus. Die für das Projekt relevantere Datei ist die Datei `main.py`, in der die Fahrzeuginstanz erstellt und die Ereignisschleife gestartet wird.

6.3.2 Vehicle Class

Die Fahrzeuginstanz ist eine Instanz der `BaseVehicle`-Klasse, die sich unter `project/vehicles/base_vehicle.py` befindet. Bei der Initialisierung wird eine Instanz der Klassen `MQTTClient` und `GPSReader` erzeugt. Danach wird eine Subroutine gestartet, die periodisch prüft, ob es übersetzte CAN-Nachrichten oder neue GPS-Messwerte gibt, die an den MQTT-Broker gesendet werden können.

6.3.3 MQTT Client Class

Der MQTT-Client ist eine Instanz der Klasse `IoVMQTTClient` und wird in der Datei `project/mqtt/mqtt_connection.py` definiert. Bei der Instanz Erstellung wird das `ConnectionInterface` gestartet, um die Konnektivität mit dem Internet herzustellen und der MQTT-Client verbindet sich mit dem Broker. Nach der ersten Verbindung

wird eine Connection-Keepalive-Subroutine erstellt, um in regelmäßigen Abständen den Server anzupingen und auf Antworten zu prüfen. Wenn die Zeit seit dem letzten empfangenen Paket vom Server einen Schwellenwert überschreitet, verbindet sich der Client erneut mit dem Server, da dies ein Anzeichen für den Verlust der Konnektivität ist. Nachrichten werden nur veröffentlicht, wenn der Client mit dem MQTT-Broker verbunden ist. Ein Flussdiagramm, das die Connection-Keepalive-Subroutine beschreibt, ist in Abbildung 6.2 dargestellt.

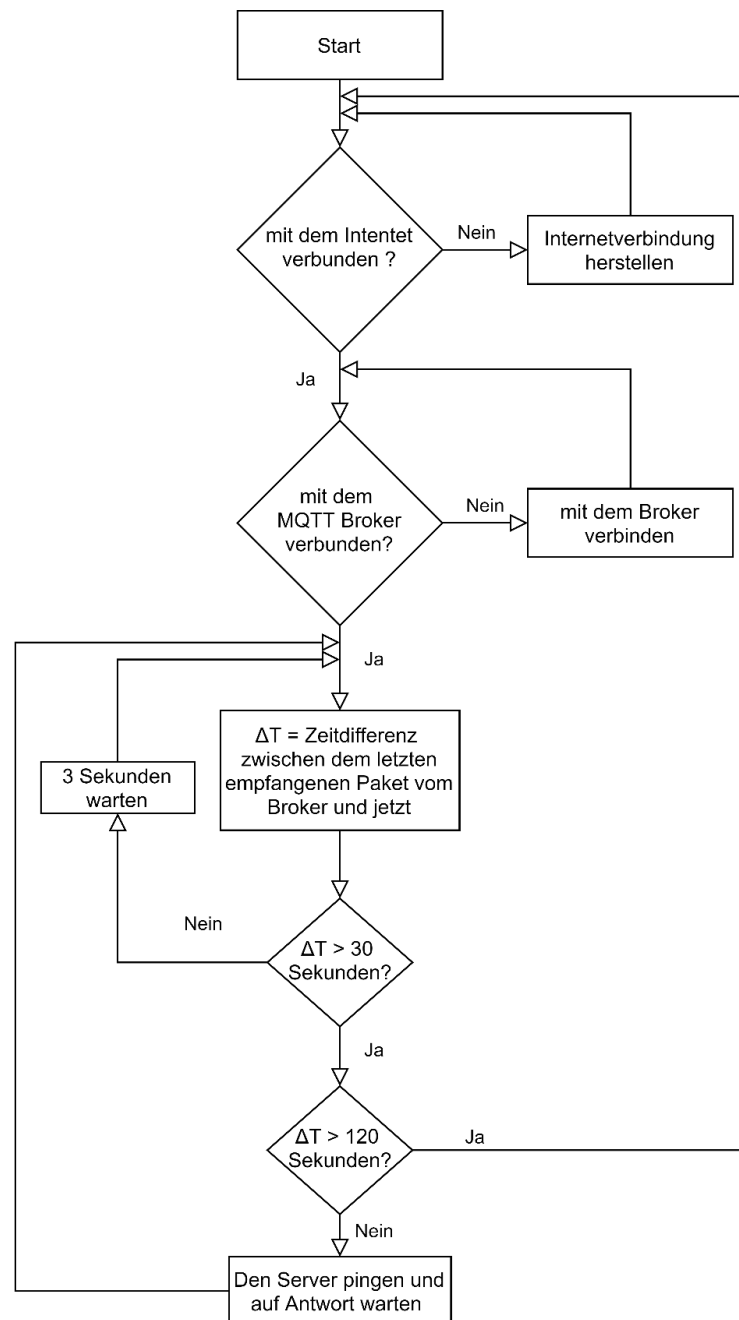


Abbildung 6.2 Beschreibung der Subroutine, die die Verbindung mit dem MQTT-Broker hält

6.3.4 WLAN and LTE Connectivity Class

Wenn die MQTT-Client-Instanz erstellt wird, wird auch eine Instanz der Klasse `ConnectionInterface` erstellt, die unter `project/connection_interface.py` definiert ist. Diese Klasse verwaltet die Verbindung mit den beiden LTE- und WLAN-Schnittstellen des Pycom-Moduls. Die Hauptmethode dieser Klasse heißt `connect_to_some_network()` und wird regelmäßig von einer anderen Unterroutine der Klasse aufgerufen. Ihr Hauptziel ist es, sicherzustellen, dass das Modul mit dem Internet verbunden ist, mit Priorität auf WLAN-Netzwerke.

Zunächst prüft die Methode, ob das Modul mit einem WLAN-Netzwerk verbunden ist, und wenn nicht, dann scannt sie die verfügbaren Netzwerke und versucht, eine Verbindung zu einem von ihnen herzustellen. Bei erfolgreicher Verbindung wird das LTE-Modul deaktiviert, falls es vorher eingeschaltet war, um Energie zu sparen. Wenn kein WLAN-Netzwerk verfügbar ist, prüft die Methode, ob die LTE-Verbindung aktiv ist, und verbindet sich mit dem LTE-Netzwerk.

6.3.5 GPS-Reader Class

Definiert in `project/gps/gps.py`, behandelt diese Klasse die empfangenen UART-GPS-Daten und verarbeitet sie mit Hilfe der `micropyGPS`-Bibliothek. Der aktuelle Standort wird in regelmäßigen Abständen abgefragt und der aktuelle Wert wird in der Eigenschaft `current_location` gehalten. Da die Genauigkeit von GPS für zivile Zwecke einige Meter beträgt, muss die Differenz zweier empfangener Positionen einen Schwellenwert überschreiten, um als neue Position zu gelten, was ebenfalls in dieser Klasse behandelt wird.

6.3.6 CAN Controller Class

Diese Klasse befindet sich unter `project/can/can_controller.py` und ihre Hauptaufgabe ist es, die Kommunikation mit dem CAN-Controller des ESP32 abzuwickeln. Beim Anlegen einer Instanz wird ein CAN-Objekt erzeugt. Dieses Objekt wird von der `BaseVehicle`-Klasse verwendet, die über Methoden zum Aktivieren und Deaktivieren eines Callbacks auf empfangene CAN-Frames verfügt.

Die Callback-Funktion wird bei einem eingehenden CAN-Frame aufgerufen, dessen Argument ein Python-Tupel in der Form `(id,data)` ist, wobei `id` und `data` die gleichnamigen Felder eines CAN-Frames sind.

6.3.7 Fahrzeugspezifischen Klassen

Derzeit ist die CAN-Frame-Übersetzung für die Fahrzeuge Mitsubishi i-Miev, Citroen C-Zero und Peugeot iOn implementiert. Die Fahrzeuge sind baugleich, daher kann die Klasse Mitsubishi, die sich unter *project/vehicles/mitsubishi.py* befindet, für alle drei oben genannten Fahrzeuge verwendet werden. Die fahrzeugspezifischen Klassen haben die Aufgabe, die Can Frames zu übersetzen. Wenn die empfangenen Daten vom aktuellen Stand der Metriken abweichen, werden die aktuellen Metriken aktualisiert und in einen Puffer geschrieben, der später in eine Datei geschrieben werden kann.

Die Tabelle 6.1 zeigt die Metriken, die von der Mitsubishi-Klasse unterstützt und übersetzt werden. Einige der Metriken werden ignoriert, da das Tempo der erzeugten Daten sehr hoch ist, wie z. B. die Motordrehzahl oder die Pedalpositionen. Einige andere Metriken werden unterstützt, aber die Übersetzung muss noch verifiziert werden, daher sind sie derzeit ebenfalls inaktiv.

Aktiv	CAN ID Metrik ID	Kennzahl	Datentyp	Beschreibung
Ja	0x101 101	car_on	boolean	true: on false: off
Ja	0x208 102	brake_pedal_position	float	0-100% des gedrückten Pedals
Ja	0x210 103	accelerator_pedal_position	float	0-100% des gedrückten Pedals
Ja	0x231 104	brake_pedal_switch	boolean	true: gedrückt false: nicht gedrückt
Nein	0x286 105	charger_temperature	integer	in Grad Celcius
Ja	0x298 107	motor_temperature	integer	in Grad Celcius
Ja	0x298 108	motor_rpm	integer	in Umdrehungen pro Minute
Nein	0x346 -	estimated_range	integer	Geschätzte Reichweite in km
Ja	0x346 109	handbrake	boolean	true: angezogen false: gelöst
Nein	0x697 114	quick_charge	boolean	true: aktiv false: inaktiv
Nein	0x286 106	charger_detection	boolean	true: aktiv false: inaktiv
Nein	0x373 -	bat_current	float	Batteriestrom in Ampere

Nein	0x373 -	bat_voltage	float	Batteriespannung in Volt
Ja	0x374 110	battery_soc	float	Ladezustand der Batterie in %
Nein	0x389 -	charge_voltage	float	Eingangsspannung des Ladegerätes
Nein	0x389 -	charge_current	float	Vom Ladegerät aufgenommener Strom
Ja	0x412 111	speed	integer	Geschwindigkeit in km/h
Ja	0x412 112	odometer	integer	Kilometerzählerwert in km
Ja	0x418 113	transmission	integer	-2: P -1: R 0: N 1: D 2: B 3: C
Nein	0x697 115	charge_current_limit	integer	Ladestrom-Grenzwert in Ampere

Tabelle 6.1 Beschreibung der Mitsubishi Fahrzeug Kennzahlen

6.3.8 Status des Moduls

Es ist möglich, den Status des Moduls zu ermitteln, indem das MQTT-Topic `v/{carId}/status` abonniert wird. Das Modul veröffentlicht den aktuellen Zeitstempel des Systems alle 2 Minuten. Wenn also die Zeitdifferenz zwischen der letzten empfangenen Nachricht und der aktuellen Zeit größer als 2 Minuten ist, ist es möglich, dass das Modul die Verbindung verloren hat.

7 Design und Aufbau der Cloud-Infrastruktur

In diesem Kapitel werden die Schritte zum Aufbau einer Cloud-Umgebung mit mehreren laufenden Diensten beschrieben. Das Ziel dieser Dienste ist es, die notwendige Konnektivität vom Modul zur Datenbank bereitzustellen und eine Webanwendung zur Anzeige der Daten bereitzustellen.

7.1 Virtuelle Linux Maschine bei Amazon Web Services

Für den Umfang dieser Arbeit wurde eine virtuelle Maschine mit einer laufenden Ubuntu Server 20.04.2 LTS-Instanz als Einsatzumgebung gewählt. Diese Hosting-Lösung von Amazon Web Services vereinfachte die Routing- und Netzwerkschritte einer selbst gehosteten Lösung erheblich. Das Projekt kann jedoch in jedem Betriebssystem, das Docker unterstützt, bereitgestellt werden.

7.2 Docker und Docker-compose

Docker ist eine Lösung zur Bereitstellung von Multi-Service-Anwendungen, bei denen die Anwendungen voneinander abhängen. Es bietet eine laufende Umgebung für jeden Dienst mit nur den notwendigen Abhängigkeiten.

Jeder einzelne Dienst hat im Verzeichnis *Daten/3_Software_Cloud/iov-full-stack-app* ein eigenes Unterverzeichnis, das die notwendigen Dateien zum Erstellen der Images enthält. Diese Verzeichnisse enthalten in der Regel ein Dockerfile im Stammverzeichnis und anwendungsabhängige Dateien.

7.2.1 Dockerfiles

Dockerfiles sind eine vereinfachte Möglichkeit, die Umgebung eines Containers zu definieren. Die wichtigsten Schlüsselwörter für dieses Projekt sind in Tabelle 7.1 beschrieben.

Schlüsselwort	Beschreibung
FROM image:tag	Legt das Docker-Basisabbild für den Container fest. Wenn es auf dem lokalen System nicht verfügbar ist, wird das Image von einer öffentlichen Registry heruntergeladen.
COPY source dest	Kopiert Dateien oder Verzeichnisse vom Host-Rechner in die Image-Umgebung
RUN command	Führt einen Befehl aus, der im Image bestehen bleibt. Dies kann z. B. verwendet werden, um Abhängigkeiten zu installieren.

CMD ["executable"]	Legt fest, welche ausführbare Datei oder welches Skript des Images beim Starten des Containers ausgeführt werden soll
--------------------	---

Tabelle 7.1 Wichtigste Dockerfile-Schlüsselwörter und ihre Bedeutung

7.2.2 Docker-Compose

Docker-Compose ist ein Tool zum Definieren und Ausführen von Multicontainer-Docker-Anwendungen. Die Dienste werden in einer YAML-Datei namens *docker-compose.yml* definiert. Mit einem einzigen Befehl ist es möglich, alle Dienste aus der Konfiguration zu erstellen und zu starten. Die wichtigsten Schlüsselwörter der Compose-Datei sind in Tabelle 7.2 beschrieben.

Schlüsselwort	Beschreibung
version:	Minimale Version von Docker-compose, die unterstützt ist.
services:	Definition eines Clusters von Containern, die gemeinsam als Teile einer Anwendung laufen sollen.
image: name	Name des Images, das von einem Container verwendet werden soll. Wenn nicht lokal vorhanden, wird es aus dem Internet heruntergeladen oder lokal erstellt, wenn das Schlüsselwort <i>build</i> vorhanden ist.
build: context: dockerfile:	Wenn vorhanden, wird ein Image mit dem beim Schlüsselwort <i>image</i> angegebenen Namen erstellt. Der Erstellungskontext wird mit dem Schlüsselwort <i>context</i> definiert und die zu verwendende Dockerfile mit dem Schlüsselwort <i>dockerfile</i> definiert.
container_name:	Definiert den Containernamen. Der Container kann von anderen Containern im gleichen Docker-Netzwerk z. B. über diesen Namen anstelle der internen IP-Adresse erreicht werden.
restart	Legt fest, ob der Container starten muss, wenn der Host-Rechner oder der Docker-Daemon neu gestartet wird.
depends_on:	Ein Dienst mit diesem Schlüsselwort wird erst dann erstellt, wenn die anderen unter diesem Schlüsselwort aufgeführten Dienste erstellt sind.
networks	Innerhalb der Definition eines Dienstes, listet die internen Docker-Netzwerke auf, die mit diesem Container verbunden sind. Wenn im globalen Bereich der Datei, deklariert sie die Docker-Netzwerke.
ports	Definiert die Port Zuordnung im Format: <i>host-port : container-port</i> , z.B. 443:8080

Tabelle 7.2 Hauptschlüsselwörter der Docker-Compose-Dateien

7.2.3 Docker Netzwerk

Um die Kommunikation zwischen Containern zu ermöglichen, kann das Schlüsselwort `network` zur Servicedefinition hinzugefügt werden, so dass Container mit anderen Containern über deren Namen innerhalb des Netzwerks kommunizieren können. Die Verwendung eines definierten Netzwerks für eine Anwendung vermeidet auch die Öffnung von Ports für andere Anwendungen, die eventuell auf demselben Docker-Daemon gehostet werden.

Ein Nginx-Reverse-Proxy-Container kann z. B. bestimmte Anfragen an den Frontend- und Backend-Container weiterleiten. Im Beispiel in Listing 7.1 werden HTTP-Anfragen, die mit dem Pfad `/api` beginnen, an den Backend-Container mit dem Namen `iov-backend` geroutet.

```
server {  
    listen 443 ssl;  
    server_name iov.enesys.rub.de;  
    location / {  
        proxy_pass http://iov-frontend;  
    }  
    location /api {  
        proxy_pass http://iov-backend:8080;  
    }  
}
```

Listing 7.1 Inter-Container-Referenz innerhalb der Datei `nginx.conf`

7.3 MQTT Broker

Die gewählte Implementierung eines MQTT-Brokers ist das Open-Source-Projekt Eclipse Mosquitto. Die Einrichtung des Docker-Images für die Verwendung mit TLS wird in diesem Unterkapitel beschrieben.

Der Eclipse-Mosquitto-Broker erwartet eine `mosquitto.conf`-Datei und die Broker-Zertifikate am Speicherort `/mosquitto/config/` innerhalb des Containers. Wenn das Image mit einer Dockerfile erstellt wird, werden die `.conf`-Datei und die Zertifikate vom Host kopiert. Wenn es erforderlich ist, die Zertifikate zu ändern, wird das Zertifikat im Host-Ordner durch das neue Zertifikat mit demselben Namen ersetzt

und das Image neu gebaut. Einige der bei der Konfiguration für dieses Projekt verwendeten Schlüsselwörter sind in Tabelle 7.3 aufgeführt und beschrieben.

Schlüsselwort	Wert	Beschreibung
listener	8883	Port, an dem der Broker zuhört
require_certificate	true	Client muss ein TLS-Zertifikat bereitstellen, um sich mit dem Broker zu verbinden
use_identity_as_username	true	Verwendet den im Client-Zertifikat definierten CommonName als Benutzernamen
allow_anonymous	false	Erlaubt anonyme Verbindungen zum Broker

Tabelle 7.3 Beschreibung einiger Schlüsselwörter der Datei mosquito.conf

7.4 Python Connector

Die Verbindung zwischen der Springboot-REST-Anwendung und dem MQTT-Broker durch die Verwendung von SSL-Zertifikaten zur Authentifizierung des Clients erwies sich als arbeitsintensiv. Eine Lösung für dieses Problem war die Erstellung eines dedizierten Microservice, der auf einem Container läuft, in dem ein Python-Skript eine Instanz eines MQTT-Paho-Clients erstellt und sich auf die Topics auf dem Broker abonniert.

Bei empfangenen Nachrichten wandelt der Konnektor die empfangenen MQTT Daten in HTTP Post-Requests um und sendet diese Requests an die Spring Boot Rest API. Listing 7.2 zeigt ein Beispiel für eine empfangene MQTT-Nachricht und die daraus resultierenden, vom Connector erstellten HTTP-Anfragen. Die Daten der CAN-Bus-Frames und des Fahrzeugstandorts werden vom Modul entsprechend in den Schlüsseln "m" und "p" des JSON-Objekts gesendet.

MQTT Datachange-Nachricht:**Topic:** v/1234/live**JSON-Payload:**

```
{
  "m": [
    {"t": 1612738798,"id": 110, "v": "78"},
    {"t": 1612738950,"id": 110, "v": "77"}
  ],
  "p": [
    {"t": 1612738798, "lat": 51.44384, "lon": 7.26179},
    {"t": 1612738810, "lat": 51.44393, "lon": 7.26185}
  ]
}
```

Resultierende POST-Anfragen:`http://iov-backend:8080/api/event/new/multiple?carId=1234`

HTTP-Request Data:

```
[
  {"t": 1612738798,"id": 110, "v": "78"},
  {"t": 1612738950,"id": 110, "v": "77"}
]
```

`http://iov-backend:8080/api/location/new/multiple?carId=1234`

HTTP-Request Data:

```
[
  {"t": 1612738798, "lat": 51.44384, "lon": 7.26179},
  {"t": 1612738810, "lat": 51.44393, "lon": 7.26185}
]
```

Listing 7.2 Beispiel für eine MQTT Datachange-Nachricht und resultierende HTTP-Anfragen

7.5 Java Spring Boot Backend REST-API

Spring Boot ist ein Java-Framework, mit dem es möglich ist, einen REST Web Service zu entwickeln und aufzubauen. Einige der Merkmale, die die Wahl dieses Frameworks motivierten, waren die Menge an Online-Dokumentation und Beispielen sowie die Robustheit der Anwendung.

7.5.1 REST HTTP Controller

Representational State Transfer ist ein De-facto-Standard, der in Webservices verwendet wird. Einige der Eigenschaften der Architektur umfassen die Performance in Netzwerken, Skalierbarkeit und Simplizität. Bei der Implementierung eines REST-Dienstes in diesem Projekt verwaltet ein REST-Controller die HTTP-Anfragen der Clients. Die Definition der URIs und die

Funktionalität, die mit jeder Anfrage verbunden ist, wird in den Klassen definiert, die im Controller-Verzeichnis der Spring Boot Java-Anwendung vorhanden sind. Listing 7.3 demonstriert die Definition des REST-Endpunkts durch Java-Dekoratoren.

```
@GetMapping("api/vehicles/find")
Vehicle findVehicle(@RequestParam String id){
    return vehicleService.findOneById(id);
}
```

Listing 7.3 Vereinfachtes Beispiel für einen HTTP-GET-Request-Handler

7.5.2 Class Diagramm

Die Beziehungen zwischen den im Directory Model definierten Entitäten sind in Abbildung 7.1 dargestellt. Zu einem Fahrzeug können viele Location- und VehicleEvents-Einträge zugeordnet sein. Jedes VehicleEvent bezieht sich auf eine der vordefinierten Metriken in der Tabelle Metric. Bei der Initialisierung der SpringBoot-Anwendung wird geprüft, ob die zu diesen Klassen gehörenden Tabellen existieren, und wenn nicht, werden die Tabellen erstellt oder aktualisiert.

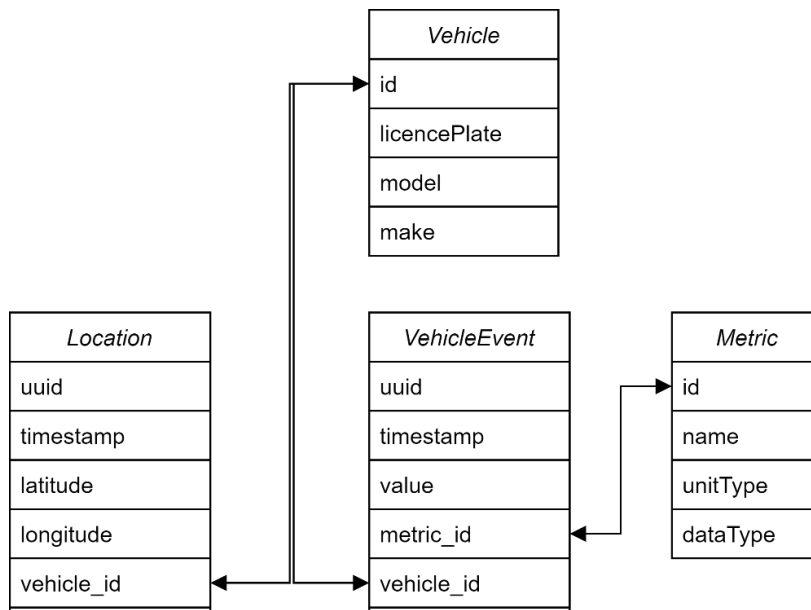


Abbildung 7.1 Klassendiagramm des Projekts

7.5.3 Jakarta Persistence API

Die Jakarta Persistence API ist eine Spezifikation von Java, die es erlaubt, Daten zwischen einem Java-Objekt und Einträgen in einer relationalen Datenbank zu persistieren. Diese Spezifikation wird in Spring Boot durch das Tool Hibernate implementiert.

Um die Verwendung dieser Persistenz-API zu veranschaulichen, zeigt

Listing 7.5 eine Anfrage nach VehicleEvent-Einträgen der Metrik 110 für das Fahrzeug mit der ID 1234, die einen Unix-Zeitstempel zwischen 1612738000 und 1612740000 haben. Der Java-Pseudocode, der diesen Filtervorgang realisiert, ist ebenfalls in Listing 7.5 angegeben.

Zeile 6 definiert, dass der eingebaute Webserver von SpringBoot an der URI *api/events/filter* hört. In den Zeilen 8-11 werden die erwarteten HTTP-Request-Parameter definiert. In Zeile 13 wird eine Methode des VehicleRepository Interface aufgerufen und JPA startet einen SQL Request an die Datenbank. Wenn ein Fahrzeug gefunden wird, macht JPA eine Abfrage an die VehicleEvent-Tabelle, um nach Einträgen mit der carId 1234 zu suchen.

Dieser Vorgang ist durch das JPA-Framework vordefiniert, so dass keine SQL-Abfragen codiert werden müssen. Anschließend können die Daten mit Standard-Java-Methoden verarbeitet werden, z. B. Streams zum Filtern und Sortieren der Daten. Diese Funktion würde auch auf anderen Datenbanken funktionieren, wobei JPA die Übersetzung zwischen den SQL-Dialekten vornimmt.

Mit dem in Abbildung 7.2 angegebenen Zustand der Datenbanktabellen wird die HTTP-Antwort des Servers in Listing 7.6 dargestellt.

```
[
  {"timestamp":1612738798,"value":"42"},
  {"timestamp":1612738854,"value":"41.5"},
  {"timestamp":1612739108,"value":"40"}
]
```

Listing 7.4 JSON-Payload mit Fahrzeugmetrikwerten

```

1  HTTP Anfrage:
2  http://iov-backend:8080/api/events/filter?carId=1234&metricId=110
3                                     &start=1612738000&end=1612740000
4
5  REST Controller URI Listener:
6  @GetMapping("api/events/filter")
7  List<VehicleEvents> findEvents(
8                                     @RequestParam String carId,
9                                     @RequestParam Integer metricId,
10                                    @RequestParam Long start,
11                                    @RequestParam Long end){
12      eventList = new ArrayList();
13      vehicleRepository.findOneById(carId).ifPresent(vehicle ->
14          eventList = vehicle.getVehicleEventList()
15          .stream()
16          .filter(event ->
17              event.getTimestamp() >= start &&
18              event.getTimestamp() <= end &&
19              event.getMetric().getId().equals(metricId)
20          );
21      return eventList;
22  };

```

Listing 7.5 Java-Pseudocode einer Funktion, die Fahrzeugereignisse filtert

Tabelle VehicleEvent			
timestamp	value	metric_id	vehicle_id
1612738798	42	110	1234
1612738854	41.5	110	1234
1612739108	40	110	1234

Tabelle Metric	
id	name
110	battery_soc
111	speed
112	odometer

Tabelle Vehicle		
id	model	make
1234	i-MiEV	Mitsubishi
5678	Ampera	Opel

Abbildung 7.2 Drei Tabellen mit relationalen Beispieldaten

7.5.4 Verbindung mit der Datenbank

Um die Verbindung der Java-Anwendung mit einer Datenbank zu erleichtern, werden die Verbindungseigenschaften in der Datei *resources/application.properties* definiert. Der Inhalt der Datei wird in Listing 7.6 angezeigt.

Um eine Verbindung zu einer anderen Datenbank herzustellen, müssen die Felder in den Zeilen 5-7 auf die richtige Adresse der Datenbank, den Benutzernamen und das Passwort geändert werden. Sobald dies geändert ist, muss das Docker-Image erneut erstellt und bereitgestellt werden.

Damit die automatische Tabellenerzeugung funktioniert, muss der SQL-Benutzer das Recht haben, Tabellen zu **erstellen** und zu **ändern** sowie Zeilen in die Tabellen **einzufragen** und zu **aktualisieren**.

```
1 spring.datasource.driverClassName=org.mariadb.jdbc.Driver
2 spring.jpa.database-platform=org.hibernate.dialect.MariaDBDialect
3 spring.datasource.initialization-mode=always
4 spring.jpa.hibernate.ddl-auto=update
5 spring.datasource.url=jdbc:mariadb://database-ip-address:3306/db-name
6 spring.datasource.username=user
7 spring.datasource.password=password
```

Listing 7.6 Inhalt der Datei *application.properties*

7.6 Angular Frontend Web App

Die Datenvisualisierung wird durch den Einsatz einer Web-Applikation erreicht. Das gewählte Framework zur Erstellung der App ist Angular, das von Google entwickelt wurde und unterstützt wird. Die Wahl dieses Frameworks basierte auf der Einfachheit der Erstellung und Integration neuer Funktionen in die App. Außerdem unterstützt es die Integration von externen Bibliotheken für Diagramme und Karten, wie z. B. Apexcharts.js und OpenLayer. Die Kommunikation mit dem Backend-Service erfolgt über HTTP-Requests und die Daten werden in einem Browser angezeigt.

Die drei Komponenten der aktuellen Version der App sind in den folgenden Abbildungen dargestellt. Abbildung 7.3 zeigt die Liste der Fahrzeuge mit einigen dargestellten Metriken. Abbildung 7.4 zeigt den Batterieladezustand eines Fahrzeugs in einem Zeitfenster, das vom Benutzer gewählt werden kann. Abbildung 7.5 zeigt eine Karte mit Markierungen, die den vom Fahrzeug gefahrenen Weg

darstellen. Das Fahrzeugsymbol stellt den aktuellen Standort des Fahrzeugs dar, der in regelmäßigen Abständen aktualisiert wird.





EneSys Vehicles Map							
ID	Model	Make	Car Status	Soc	Odometer	Charging State	Color
1236	Ampere	Opel	On	52%	36,381 km	0	
1234	i-MiEV	Mitsubishi	On	64%	47,678 km	2	
1235	iOn	Peugeot	On	7%	42,161 km	1	
54567	Stromos	German E-Cars	Off	87%	54,620 km	0	

Abbildung 7.3 Liste der vom System erfassten Fahrzeuge

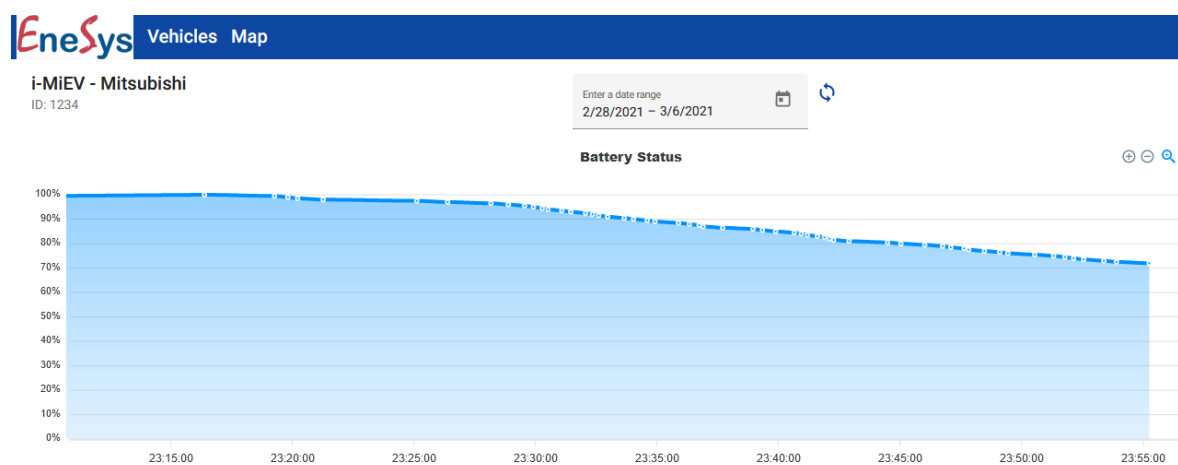


Abbildung 7.4 Grafik des Batteriezustands eines Fahrzeugs im Zeitverlauf

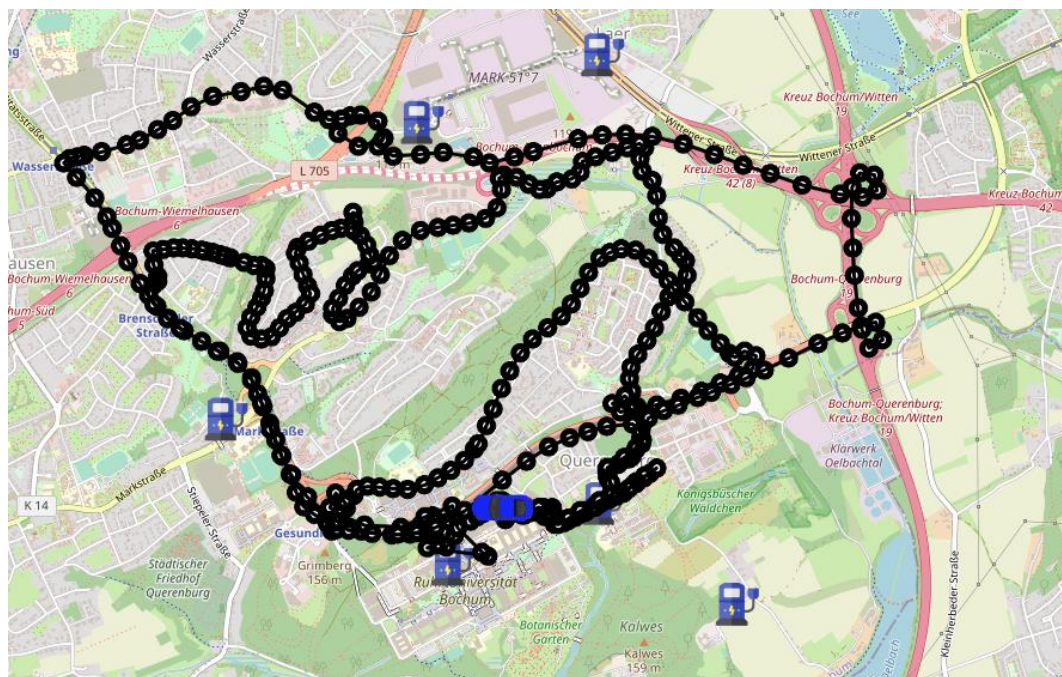


Abbildung 7.5 Karte, die die vom Fahrzeug zurückgelegten Wege basierend auf GPS-Daten anzeigt

7.7 Reverse Proxy

Ein Reverse-Proxy ist der Eintrittspunkt für die Web Anwendung, und seine Aufgabe ist es, die HTTP-Anfragen an die Frontend- und Backend-Container weiterzuleiten. Außerdem stellt er eine verschlüsselte und authentifizierte Verbindung zwischen dem Client am Internetbrowser und dem Server her.

Ähnlich wie bei der MQTT-Broker-Client-Verbindung wird der Zugriff auf die Webanwendung durch SSL-Zertifikate eingeschränkt. Eine Anleitung zur Installation dieser Zertifikate ist im Verzeichnis *tutorials/browser-certificate* zu finden.

Die Webserver-Zertifikate werden im Verzeichnis *nginx-orchestrator/certs* abgelegt und müssen der in der Tabelle 7.4 angegebenen Nomenklatur folgen. Das Client-Zertifikat muss von der gleichen Certificate Authority generiert werden.

ca.crt	Zertifikat der Certificate Authority
nginx_server.crt	Zertifikat des Webserver, in diesem Fall des nginx-Reverse-Proxys
nginx_server.key	Privater Schlüssel zum Server-Zertifikat

Tabelle 7.4 Nomenklatur der Nginx-Webserver-Zertifikate

Im Fall dieser Anwendung wurde ein selbstsigniertes Zertifizierungsstellen-Zertifikat generiert und verwendet, um die weiteren Zertifikate des Servers und der Clients zu signieren. Die Implementierung mit einer öffentlichen Zertifizierungsstelle ist möglich, aber die App muss mit Benutzerrollen und Login und Passwort gesichert werden.

7.8 Deployment der Anwendung

Unter dem Basisordner *iov-full-stack-app* gibt es eine Datei namens *docker-compose.yml*, die das Deployment der vollständigen Anwendung koordiniert. Die in der Datei *docker-compose* verwendeten Variablen werden in der Datei *.env* definiert, die sich im gleichen Stammverzeichnis befindet.

Um die Anwendung zum ersten Mal bereitzustellen oder wenn Sie Änderungen an den Diensten vornehmen, müssen die in Listing 7.7 definierten *docker-compose*-Befehle auf dem Verzeichnis *iov-full-stack-app* ausgeführt werden.

docker-compose build

Alle Images der Container erstellen

docker-compose up

Alle definierten Container ausführen

docker-compose up -d

das Flag -d bedeutet, dass die Dienste im Hintergrund laufen, d.h. wenn das Terminal geschlossen wird, laufen die Dienste weiter

docker-compose down

Schaltet alle Anwendungen aus. **Achtung!** Dieser Befehl löscht auch die Container und die darin enthaltenen Daten, falls es sich um eine Datenbank handelt. Es wird empfohlen, Docker-Volumes zu verwenden, um die Daten zwischen Containern zu persistieren.

Listing 7.7 Kommandozeilenbefehle zum Erstellen und Ausführen der Cloud-Anwendung

8 Tests und Ergebnissen

8.1 CAN-Bus Frame-Erkennung und -Dekodierung

Der erste Schritt zur Verarbeitung von Fahrzeugdaten ist eine Plattform, die CAN-Bus-Datenframes lesen und übersetzen kann. Abbildung 8.1 zeigt die rohen CAN-Datenframes, die von der CAN-Bibliothek von Pycom verarbeitet werden. Das Lesen und Filtern von CAN-Datenframes waren erfolgreich.

```
CAN Rx: (id=768, data=b'\x00\x1b\x1f\xff\x07\xd0\xff\xff', rtr=False, extended=False)
CAN Rx: (id=776, data=b'\x00\x00\x00\x00\x10\x00\x00\x80', rtr=False, extended=False)
CAN Rx: (id=1048, data=b'P\x00\x00\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=1746, data=b'\\U\x15\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=645, data=b'\x07\xd0\x05\x00r\x00h\x13', rtr=False, extended=False)
CAN Rx: (id=528, data=b'\x00\x00\x00\x00\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=648, data=b"\x07\xd0'\x10\x01\x00\x11\x00", rtr=False, extended=False)
CAN Rx: (id=646, data=b'\x00\x00\x00+\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=561, data=b'\x00\x00\x00\x00\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=1380, data=b'\x00@T\x00\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=1750, data=b'\x00\x00\x00\x00\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=645, data=b'\x07\xd0\x05\x00r\x00h\x13', rtr=False, extended=False)
CAN Rx: (id=776, data=b'\x00\x00\x00\x00\x10\x00\x00\x80', rtr=False, extended=False)
CAN Rx: (id=1048, data=b'P\x00\x00\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=901, data=b'C\x00\x00\x00\x00\x00\x00\x00', rtr=False, extended=False)
CAN Rx: (id=645, data=b'\x07\xd0\x05\x00r\x00h\x13', rtr=False, extended=False)
```

Abbildung 8.1 CAN-Bus-Datenframes, die vom seriellen Monitor des Mikrocontrollers angezeigt werden

8.2 Fahrzeugortung über GPS

Das Elektronikmodul empfängt serielle Daten von einem externen GPS-Modul und die Fahrzeugposition sowie die Uhrzeit können ermittelt werden. Abbildung 4 zeigt Daten aus dem Modul, das den aktuellen Standort enthält.

```
unix time: 1615043539
YYYY.MM.DD HH:MM:SS
2021.3.6 15:12:19
latitude: 51.44753
longitude: 7.266654

unix time: 1615043547
YYYY.MM.DD HH:MM:SS
2021.3.6 15:12:27
latitude: 51.44753
longitude: 7.266651

unix time: 1615043554
YYYY.MM.DD HH:MM:SS
2021.3.6 15:12:34
latitude: 51.44753
longitude: 7.266646
```

Abbildung 8.2 Informationen über den Standort und die Zeit, die von einem GPS-Satelliten empfangen wurden

8.3 Datenverarbeitung in der Cloud

Die vom Modul gesendeten Daten werden vom MQTT Broker empfangen und es ist möglich, das Topic `v/{carId}/live` zu abonnieren und Daten in Echtzeit zu empfangen. Abbildung 8.3 Links zeigt eine empfangene MQTT-Nachricht mit Informationen über Fahrzeug-Kennzahlen und die aktuelle Position. Die erfassten Fahrzeug-Kennzahlen und der Standort werden dann in der Datenbank gespeichert und können wie in Abbildung 8.3 Rechts abgerufen werden.

v/123/live

v/123/#

06-03-2021 16:21:06.58866630

```
{  "m" : [ {    "t" : 1615044062,    "id" : 110,    "v" : 34.5  }, {    "t" : 1615044062,    "id" : 111,    "v" : 79  }, {    "t" : 1615044062,    "id" : 112,    "v" : 263695  } ],  "p" : [ {    "t" : 1615044051,    "lat" : 51.44761,    "lon" : 7.266868  } ]}
```

SELECT TIMESTAMP ,VALUE ,METRIC_ID ,VEHICLE_ID FROM VEHICLE_EVENT;

TIMESTAMP	VALUE	METRIC_ID	VEHICLE_ID
1614265926	82	110	1234
1614266106	79	110	1234
1614266286	78	110	1234
1614266526	77	110	1234
1614266706	74	110	1234
1614266946	73	110	1234
1614267186	72	110	1234
1614267300	70	110	1234
1614267546	68	110	1234
1614267786	67	110	1234
1614267966	65	110	1234

(11 rows, 5 ms)

Abbildung 8.3 Links: MQTT-Nachricht mit Informationen über Fahrzeugkennzahlen und die aktuelle Position. Rechts: Ergebnis der SQL-Abfrage auf eine Tabelle, die die Fahrzeugkennzahlen enthält

8.4 Frontend-Webanwendung

Neben der Abbildung 7.3, Abbildung 7.4 und Abbildung 7.5 ist die Webanwendung online unter der Adresse <https://iov.chiaramonte.me:8080> zu sehen.

8.5 Weiterentwicklung

Einige Punkte der weiteren Entwicklung umfassen ein robusteres Datenerfassungsmodul, das in einer hardwarenäheren Sprache programmiert ist, beispielsweise in C. Die Frame-Übersetzungsmethode des Moduls muss erweitert werden, um mehr Fahrzeugkennzahlen abzudecken. Die Web-App zur Datenvisualisierung kann ebenfalls mit mehr Funktionen sowie echter Live-Datenvisualisierung, z. B. mit MQTT über Web-Sockets, verbessert werden.

9 Zusammenfassung

Die für diese Arbeit definierten Ziele betrachteten die vielen Schritte bei der Entwicklung und dem Aufbau eines elektronischen Moduls und einer Cloud-Infrastruktur, um eine zuverlässige Übertragung der von Elektrofahrzeugen gesammelten Echtzeitdaten zu ermöglichen.

Nach einer Konzeptionsphase, in der die verfügbaren Daten und Methoden zur sicheren Datenübertragung recherchiert wurden, wurde ein elektronisches Modul erfolgreich entworfen und aufgebaut.

Die Test- und Validierungsphase demonstrierte die Zuverlässigkeit des aufgebauten Systems und der zugehörigen Infrastruktur, wobei Live-Fahrzeugdaten wie Geschwindigkeit, Batteriestatus und der Standort der Fahrzeuge in Echtzeit an eine Datenbank übertragen und in einer Webanwendung angezeigt wurden.

Der Umfang der gesammelten Daten könnte nicht nur den Projekten im Zusammenhang mit dieser Arbeit nützlich sein, sondern eröffnet auch Möglichkeiten für neue Forschungen über die Nutzung von Elektrofahrzeugen als Ganzes, z. B. wie das Verhältnis von Fahrgewohnheiten die Batterielebensdauer von Fahrzeugen beeinflussen kann und wie die Infrastruktur von Städten auf der Basis von historischen Daten aus diesem System besser geplant werden kann.

10 Literaturverzeichnis

Blau, Joe; Wagland, Paul; Harrison, Kevin (2021): Tesla API. Online verfügbar unter <https://www.teslaapi.io/>, zuletzt geprüft am 01.03.2021.

Dhall, Rohit; Solanki, Vijender Kumar (2017): An IoT Based Predictive Connected Car Maintenance Approach. In: *IJIMAI* 4 (3), S. 16. DOI: 10.9781/ijimai.2017.433.

Ecma International (2017): ECMA-404. The JSON Data Interchange Syntax, zuletzt geprüft am 21.01.2021.

OASIS (2014): MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta, zuletzt aktualisiert am 29.10.2014, zuletzt geprüft am 24.01.2021.

ISO 7498-4, 1989: Open Systems Interconnection. Online verfügbar unter [https://standards.iso.org/ittf/PubliclyAvailableStandards/s014258_ISO_IEC_7498-4_1989\(E\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/s014258_ISO_IEC_7498-4_1989(E).zip), zuletzt geprüft am 02.02.2020.

Open Vehicles: Open Vehicles Monitoring System. Online verfügbar unter <https://docs.openvehicles.com/>, zuletzt geprüft am 23.01.2021.

Panchi, Freddy; Hernandez, Karla; Chavez, Danilo (2018): MQTT Protocol of IoT for Real Time Bilateral Teleoperation Applied to Car-Like Mobile Robot. In: 2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM). 2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM). Cuenca, Ecuador, 15.10.2018 - 19.10.2018: IEEE, S. 1–6.

Prasad, B.V.P.; Tang, Jing-Jou; Luo, Sheng-Jhu: Design and Implementation of SAE J1939 Vehicle Diagnostics System. In: 2019 IEEE International Conference 11.2019, S. 71–74.

Telekom: 3G geht, LTE für alle kommt: Die zehn wichtigsten Fragen. Online verfügbar unter <https://www.telekom.com/de/blog/netz/artikel/3g-abschaltung-608274>, zuletzt geprüft am 01.03.2021.

Telekom (2019): Mobile IoT Guide. Online verfügbar unter <https://iot.telekom.com/resource/blob/data/176340/02ccc79436c73ed5a6632ffc04a438d6/narrowband-mobile-iot-guide-2019.pdf>, zuletzt geprüft am 18.02.2021.

Telekom (2021): Mobile-IoT Verfügbarkeit. Online verfügbar unter <https://t-map.telekom.de/tmap2/mobileiot/>, zuletzt geprüft am 18.02.2021.

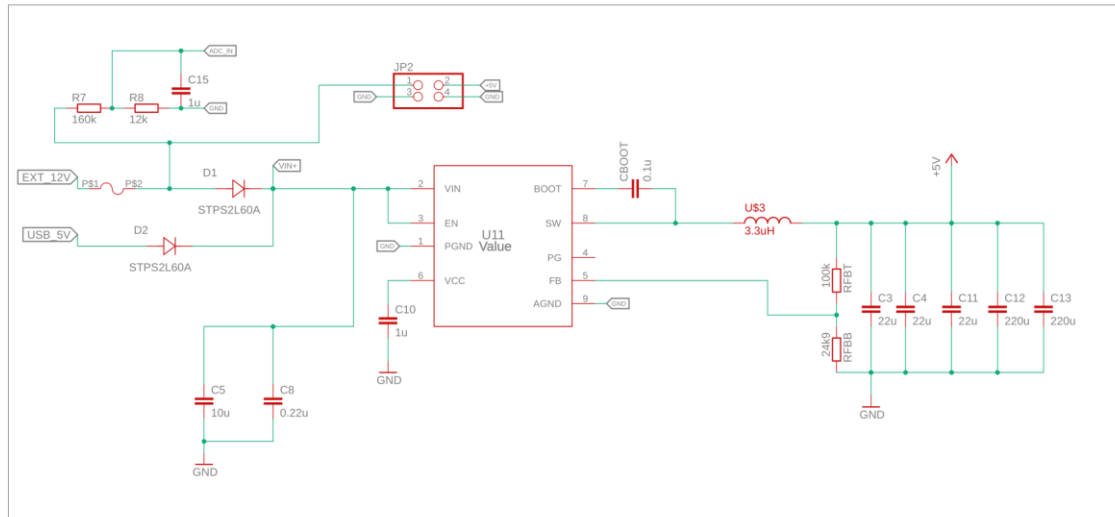
Wang, Charlie (2018): HTTP vs. MQTT: A tale of two IoT protocols. Google. Online verfügbar unter <https://cloud.google.com/blog/products/iot-devices/http-vs-mqtt-a-tale-of-two-iot-protocols>, zuletzt geprüft am 10.02.2021.

Yang, Ping; Xiao, Yue; Xiao, Ming; Li, Shaoqian (2019): 6G Wireless Communications: Vision and Potential Techniques. In: *IEEE Network* 33 (4), S. 70–75. DOI: 10.1109/MNET.2019.1800418.

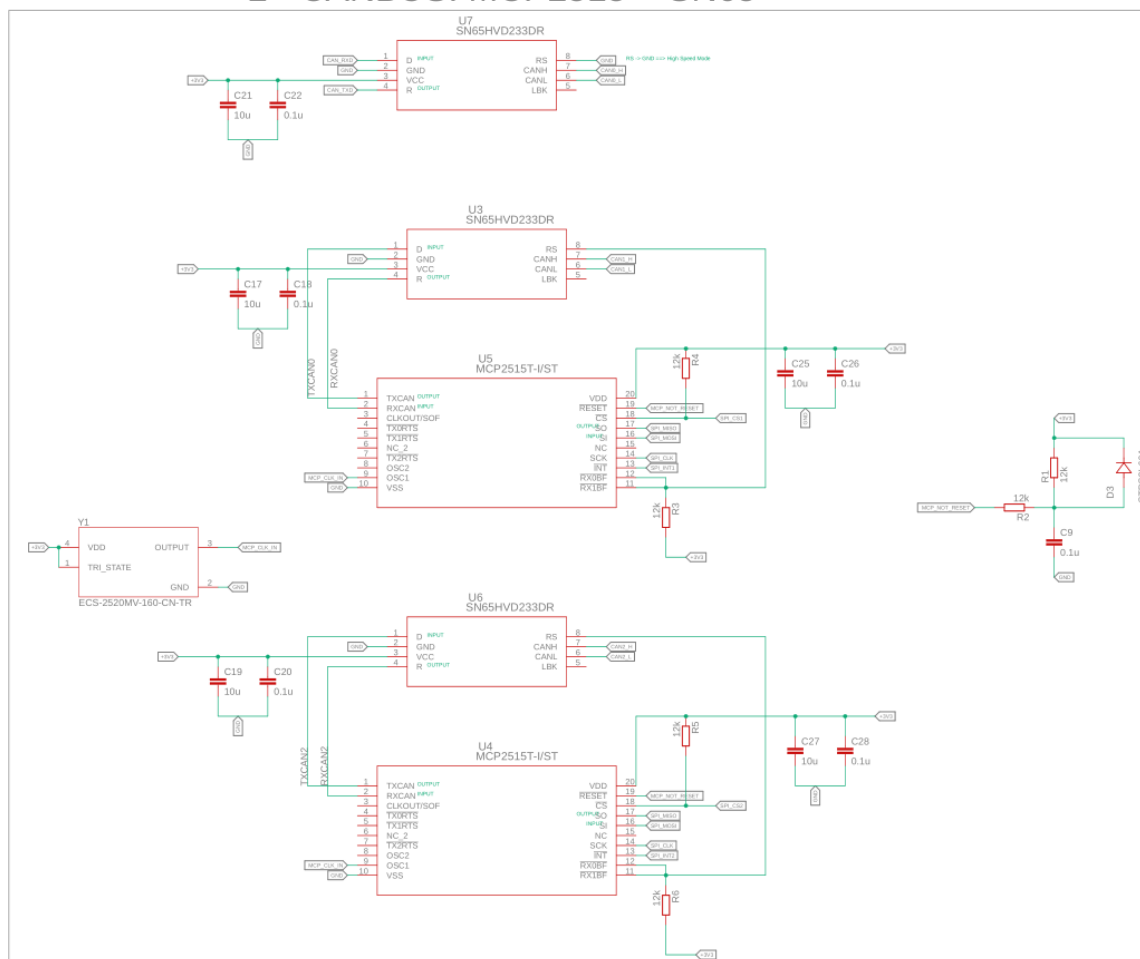
11 Anhang

Schaltplan der Leiterplatte

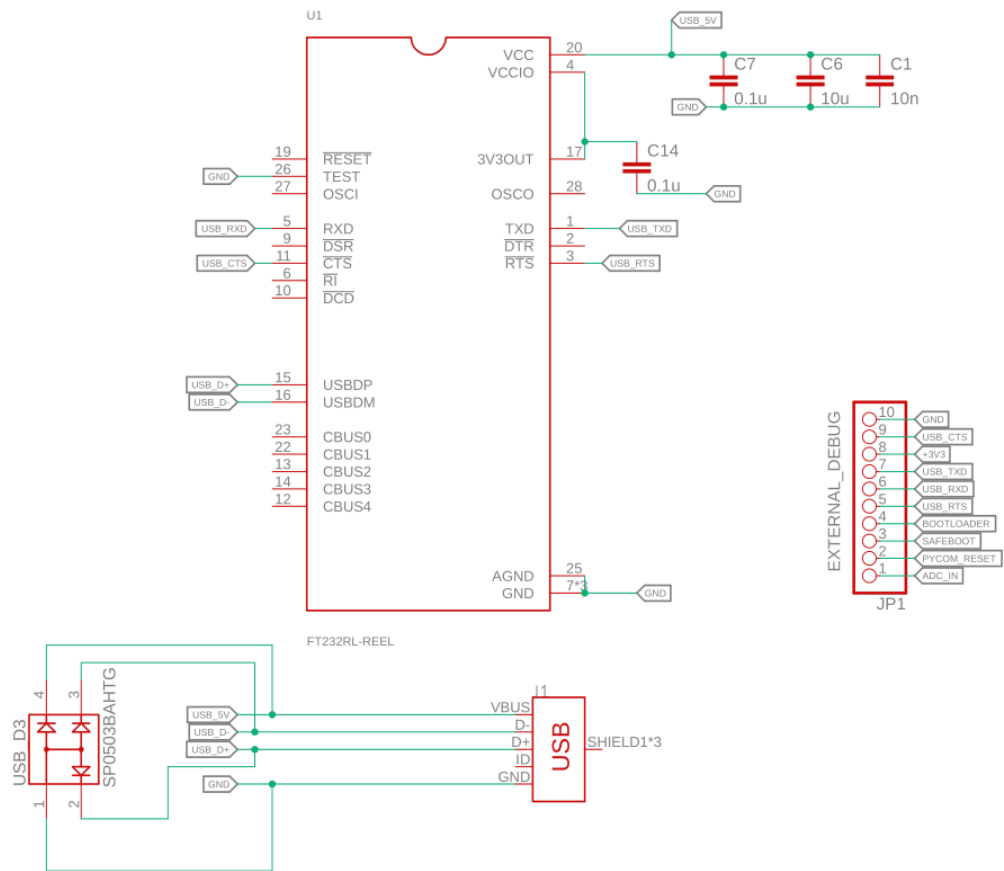
1 - POWER SUPPLY +12V +5V --> +5V



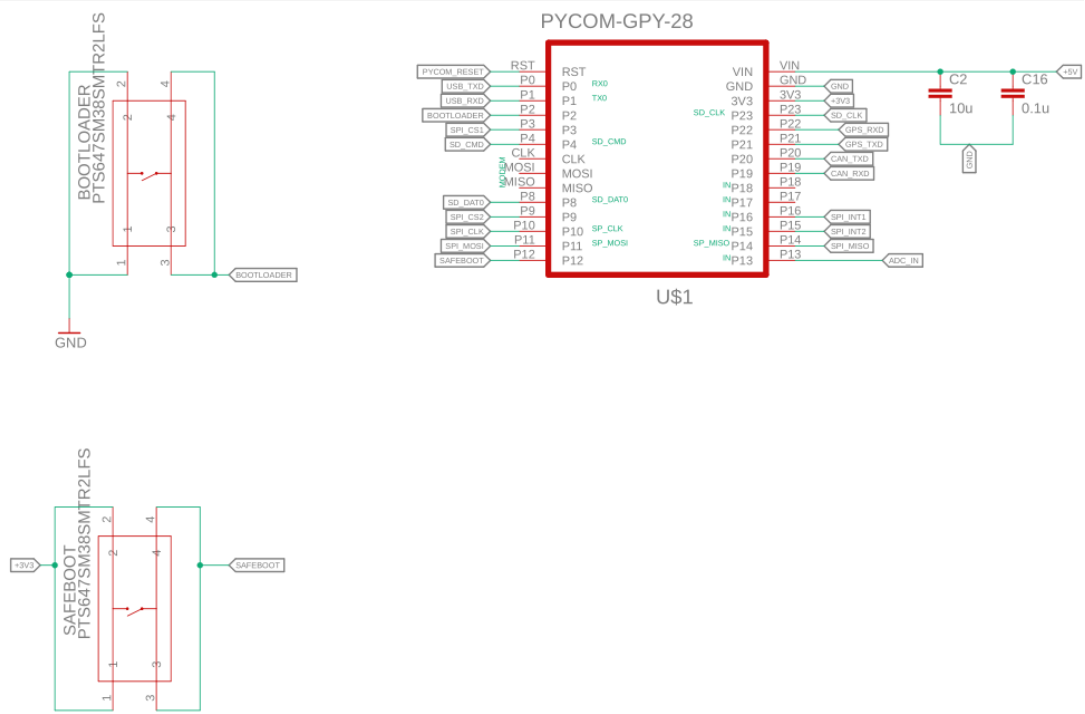
2 - CANBUS: MCP2515 + SN65



3 - USB --> UART



4 - PYCOM GPy



5 - SDCARD

