HAW HAMBURG

Bachelorarbeit

Jannes Volkens

Eine Gateway-Funktion zwischen CAN und Ethernet oder SOME/IP für RIOT

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Jannes Volkens

Eine Gateway-Funktion zwischen CAN und Ethernet oder SOME/IP für RIOT

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

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 20. Juli 2020

Jannes Volkens

Thema der Arbeit

Eine Gateway-Funktion zwischen CAN und Ethernet oder SOME/IP für RIOT

Stichworte

Gateway, CAN, Ethernet, SOME/IP, RIOT

Kurzzusammenfassung

In dieser Arbeit wird die Realisierung eines Gateways vorgestellt, welches ein erster Entwicklungsschritt hin zu einer leistungsstärkeren Kommunikationsverbindung mit hoher Bandbreite in Automobilen ist. Das Gateway arbeitet als Schnittstelle zwischen CAN- und Ethernet- oder IP-basierten Netzwerken, wobei die IP-Kommunikation das neue Kommunikationsprotokoll SOME/IP verwendet. Damit das Gateway ein kommunikationsfähiges System darstellt, das auch mit den geringen Ressourcen eines Steuergerätes im Automobil auskommt, wird die Implementierung mithilfe des IoT-Betriebssystems RIOT realisiert. Auf dieser Grundlage wird das Zeitverhalten sowie der Paketverlust untersucht und einige Interoperabilitätstests durchgeführt. Mithilfe dieser Ergebnisse werden Unterschiede zwischen CAN und Ethernet oder IP aufgezeigt und gleichzeitig die Leistungen dieser Kommunikationstechnologien verglichen.

Jannes Volkens

Title of Thesis

A gateway-function between CAN and Ethernet or SOME/IP for RIOT

Keywords

Gateway, CAN, Ethernet, SOME/IP, RIOT

Abstract

In this paper the realization of a gateway is introduced, which represents a first developmental step towards a more powerful communication link with high bandwidth in automobiles. The gateway works as an interface between CAN and Ethernet or IP based networks, wherein the IP communication uses the new communication protocol SOME/IP. To ensure that the gateway provides a system that remains capable of communication using the low resources of an automotive ECU, the implementation is realized using the IoT operating system RIOT. On this basis, the timing behavior and packet loss are analyzed and some interoperability tests are performed. With the help of these results, differences between CAN and Ethernet or IP are shown and the performance of these communication technologies is compared.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Menge an Daten innerhalb eines Fahrzeuges nimmt durch neue Anforderungen aus dem Bereich des autonomen Fahrens kontinuierlich zu und bringt das klassische Fahrzeugnetzwerk an seine Grenzen. Neue Anwendungen der Umgebungserkennung, die moderne Sensoren, Kameras und Steuergeräte nutzen, benötigen leistungsstärkere Kommunikationsverbindungen, um diese Fülle an Daten zuversichtlich transportieren zu können. Automotive Ethernet stellt eine mögliche Lösung dar, die Engpässe herkömmlicher CAN-Busse aufzulösen. Ein erster Schritt hin zu der Etablierung von Ethernet im Automobil ist das Ersetzen von Kommunikationskomponenten durch eine Ethernet-Infrastruktur. Dabei werden alte CAN-basierte Steuergeräte weiterhin genutzt, wodurch sogenannte Gateways benötigt werden, die zwischen den verschiedenen Kommunikationstechnologien übersetzen.
In dieser Arbeit wird ein Gateway vorgestellt, welches als Schnittstelle zwischen den verschiedenen Kommunikationstechnologien eines Automobils agieren kann. Es ist in der Lage, Informationen transparent zwischen CAN- und Ethernet- oder IP-basierten Netzwerken umzusetzen, wobei für die IP-Kommunikation das neue Automotive-Kommunikationsprotokoll SOME/IP benutzt wird. Zusätzlich wird das Gateway mit den geringen Ressourcen eines Steuergerätes im Automobil, einem minimalen Stromverbrauch sowie einer geringen Komplexität umgesetzt. Damit das Gateway ein kommunikationsfähiges System darstellt, welches die verschiedenen Standards beherrscht und auf angemessener Hardware ausführbar ist, wird das IoT-Betriebssystem RIOT verwendet.
Diese Arbeit ist in sechs verschiedene Abschnitte unterteilt, welche die Erarbeitung des Gateways vorstellen. Kapitel 2 schafft zunächst einen Überblick über die theoretischen Grundlagen sowie den Stand der Technik und beinhaltet eine Beschreibung des Automotive Ethernets, die Vorstellung des Kommunikationsprotokolls SOME/IP, eine Erläuterung des Betriebssystems RIOT und die Charakterisierung des Controller Area Networks. Die funktionalen und nichtfunktionalen Anforderungen an das Gateway werden in Kapitel 3 aufgezeigt. Anschließend veranschaulicht Kapitel 4 die Implementierung des Gateways, woraufhin der genutzte Versuchsaufbau in Kapitel 5 beschrieben wird. Die

Evaluierung dieser Arbeit findet in Kapitel 6 statt, auf welche in Kapitel 7 ein Fazit sowie ein kurzer Ausblick folgt.

2 Stand der Technik

Im Folgenden wird ein Überblick über den Stand der Technik und die theoretischen Grundlagen gegeben, welche für die vorliegende Arbeit notwendig sind.

2.1 Automotive Ethernet

Heutige Kraftfahrzeuge setzen diverse Fahrerassistenzsysteme wie beispielsweise ASR (Antriebsschlupfregelung) oder ESP (Elektronisches Stabilitätsprogramm) ein, um den Fahrer zu unterstützen. Aktuell ist das Fahrzeugnetzwerk ein komplexes System, das aus verschiedenen Netzwerktechnologien, wie beispielsweise dem Controller Area Network (CAN), besteht, welche über ein zentrales Gateway miteinander kommunizieren. Sensoren verarbeiten Informationen typischerweise direkt und übertragen die Ergebnisse über Feldbusse an die jeweiligen Steuergeräte (ECUs) [1, 2]. Da CAN durch eine geringe Bandbreite und Nachrichtengröße limitiert ist, steigt mit den neuen Anforderungen des autonomen Fahrens der Bedarf an leistungsstärkeren Kommunikationsverbindungen [1, 3]. Assistenzsysteme der Zukunft greifen auf Kamera-, Radar- oder LIDAR-Sensoren (LIght, Detection And Ranging) zurück, die eine sehr hohe Auflösung nutzen und in mehreren Dimensionen arbeiten, weswegen sie größere Datenmengen als bisherige Sensoren produzieren. Daher benötigen sie eine hohe Bandbreite, um Informationen an verarbeitende Steuergeräte übertragen zu können. Um noch akkuratere Ergebnisse der Umfelderkennung zu bewirken, sollen zukünftig Sensorfusionssysteme, die Sensorrohdaten mehrerer Sensoren vereinen, verwendet werden [1].
Der Begriff Automotive Ethernet beschreibt die derzeitigen Bestrebungen für die Etablierung von Ethernet im Auto, um leistungsfähigere Fahrzeugnetzwerkarchitekturen zu ermöglichen. Bisher handelt es sich hierbei um eine Auswahl von Technologiezweigen und Ideen für diese Umsetzung [1].
Neue Netzwerkarchitekturen verwenden ein zentralisiertes Backbone-Netzwerk, welches

verteilte Steuereinheiten miteinander verbindet, um den künftigen Anforderungen gerecht zu werden. Geschaltetes Ethernet als Kommunikationsinfrastruktur ist eine Möglichkeit, einen Kommunikations-Backbone im Auto zu realisieren, da Ethernet als ein sehr flexibles, skalierbares und kosteneffizientes Protokoll gilt, jedoch keine erforderlichen Eigenschaften der Echtzeitkommunikation aufweist [1, 4]. Echtzeit-Ethernet-Protokolle, wie beispielsweise die Echtzeit-Ethernet-Erweiterung Time-Triggered Ethernet (TTEthernet), bieten Ansätze, um Echtzeitverhalten zu erreichen. TTEthernet stellt die Möglichkeit bereit, Nachrichten mit harten Echtzeitanforderungen zu übertragen, indem alle Komponenten ihre eigene global synchronisierte lokale Zeit und einen eigenen Übertragungszeitplan besitzen [4]. Auf dieser Grundlage ist Automotive Ethernet eine Möglichkeit der Bewältigung von zukünftigen Herausforderungen eines Boardnetzwerkes [1].
Die heutige Topologie eines Automobils beinhaltet Feldbusse für die verschiedenen Anwendungsdomänen, welche domänenübergreifend über ein zentrales Gateway miteinander kommunzieren. Das Ziel wäre ein Ethernet-basiertes Boardnetz mit einer flachen Topologie, was bedeutet, dass alle Steuergeräte, die vorher über Feldbusse verbunden waren, nun über Ethernet-Links miteinander kommunizeren können. Ein erster Entwicklungsschritt wäre die Etablierung einer Domänen-Gateway-Topologie, in welcher das zentrale Gateway durch mehrere Domänen-Gateways ersetzt wird. In dieser Topologie wird erstmalig ein zentraler Ethernet-Switch verwendet, über den die Anwendungsdomänen miteinander kommunizieren [1].
Serviceorientierte Architekturen (SOA) sind eine effiziente und flexible Möglichkeit, Systeme miteinander zu verbinden, um bestimmte Aufgaben zu erfüllen. Der Einsatz von Ethernet in der Automobilindustrie ermöglicht das Verwenden von SOAs in den eingebetteten Systemen der Kraftfahrzeuge. Da Steuereinheiten im Automobil limitierte Ressourcen besitzen, können die auf abstrakter Ebene entwickelten Computer und Server SOA-Protokolle nicht in eingebetteten Fahrzeugsystemen verwendet werden. Es bedarf neue Protokolle und Middleware, die SOA-Fähigkeiten für eingebettete Systeme im Automobil umsetzen [5]. AUTOSAR (AUTomotive Open System ARchitecture) hat sich als ein globales Standardisierungskonsortium für Software-Architekturen in der Automobilindustrie der Aufgabe angenommen und ein Protokoll names Scalable service-Oriented MiddlewarE over IP (SOME/IP) kreiert[5, 6].

2.2 SOME/IP

2.2.1 Protokollspezifikation

Scalable service-Oriented MiddlewarE over IP (SOME/IP) ist ein neues Kommunikationsprotokoll aus dem Bereich Automotive Ethernet, welches diverse Middleware-Eigenschaften wie das Aufrufen von entfernten Funktionen (RPC), Ereignisbenachrichtungen und die Serialisierung in und aus dem On-Wire-Format unterstüzt [7, 5]. Das Protokoll kann sowohl auf verschiedenen Betriebssystemen als auch auf eingebetteten Geräten ohne Betriebssystem implementiert werden [7, 8]. SOME/IP wird über ein IP-basiertes Netzwerk im Automobil als Nutzlast von UDP (User Datagram Protocol) oder TCP (Transmission Control Protocol) transportiert, wobei die Auswahl des Transportprotokolls die Nachricht selber nicht beeinflusst [7, 9].
SOME/IP erlaubt ein Publish/Subscribe-Konzept, wohingegen klassische Bussysteme wie beispielsweise CAN nach einem Multi-Master-Prinzip arbeiten. CAN sieht vor, dass Sender, wenn sie die Notwendigkeit dazu sehen, Daten übertragen und zwar unabhängig davon, ob die Empfänger die Daten benötigen. Durch das Publish/Subscribe-Konzept werden Daten eines Senders im Netzwerk nur dann übertragen, wenn mindestens ein Abonnent diese benötigt. So wird das Netzwerk entlastet, da die angeschlossen Knoten keine unnötigen Daten erhalten. Mithilfe der SOME/IP-Service-Discovery können Klienten Inhalte eines Dienstes beim Server abonnieren und erhalten bei „Events“ die aktualisierten Daten des Servers [10, 11]. Ein Dienst kann von mehreren Klienten gleichzeitig abonniert werden und besteht aus einer Kombination aus keinen oder mehreren Events, Methoden oder Feldern [7, 11].
Anbieter senden Daten mithilfe von Events entweder zyklisch oder bei Änderungen an die Abonnenten des jeweiligen Dienstes. Methoden geben Abonnenten die Möglichkeit, entfernte Methoden auf Anbieterseite aufzurufen und auszuführen. Felder bestehen aus einer Kombination von einem oder mehreren „Notifiern“, „Gettern“ oder „Settern“. Bei Änderungen von Daten sendet der Notifier diese mit einer Nachricht an die Abonnenten. Mithilfe von Gettern und Settern erhalten Abonnenten lesenden oder schreibenden Zugriff auf Felder auf Seiten des Anbieters. Der Hauptunterschied zwischen den Notifiern von Feldern und Events liegt darin, dass Events ausschließlich bei Änderungen von Daten gesendet werden und Notifier die Daten zusätzlich beim Abonnieren eines Dienstes an die Abonnenten senden [7].

2.2.2 SOME/IP Paketaufbau

Eine SOME/IP-Nachricht besteht aus insgesamt zwei Teilen: Dem SOME/IP-Paketkopf und der Nutzlast der jeweiligen Nachricht. Der SOME/IP-Paketkopf besteht aus insge-

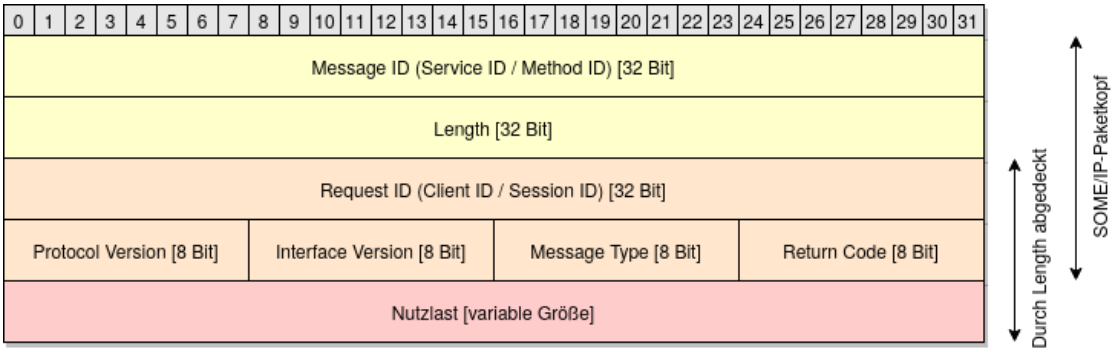

Abbildung 2.1: SOME/IP Paketaufbau
Quelle: In Anlehnung an [7]

samt sieben Feldern (siehe Abbildung 2.1). Die Message ID (Nachrichten-ID) lässt sich weiter in Service ID (Dienst-ID) und Method ID (Methoden-ID) unterteilen, welche Identifikatoren des jeweiligen Dienstes und der genutzten Methoden sind. Das Feld Length (Längenfeld) enthält die Länge des Paketkopfes inklusive der Nutzlast und exklusive der Message ID und der Length in Byte. Die Request ID (Anfrage-ID) ist auch in Client ID (Klienten-ID) und Session ID (Sitzungs-ID) unterteilt. Die Client ID ist ein einzigartiger Identifikator des aufrufenden Klienten und die Session ID erlaubt die Unterscheidung von sequentiellen Nachrichten und Anfragen des gleichen Absenders und wird bei jedem Aufruf inkrementiert. Die Protocol Version (Protokollversion) und Interface Version (Schnittstellenversion) geben das Paketkopfformat (ohne Nutzlastformat) und die Hauptversion der Schnittstelle an. Anhand des Message Types (Nachrichtentyps) werden die verschiedenen Nachrichtentypen unterschieden und der Return Code (Rückgabecode) gibt an, ob eine Anfrage erfolgreich verarbeitet wurde [7].
Nach dem SOME/IP-Paketkopf folgt die Nutzlast der jeweiligen Nachricht, welche die zu transportierenden Daten mit einer variablen Größe enthält. Die Größe der Nutzlast ist dabei abhängig von dem genutzten Transportprotokoll. In einem UDP Paket können mehrere SOME/IP-Nachrichten enthalten sein, da durch das Längenfeld das Ende der einzelnen Nachricht ermittelt werden kann. TCP wird verwendet, sobald Nachrichten die Größe eines UDP Pakets übersteigen, da TCP die Segmentierung der Nutzlast erlaubt und somit automatisch größere Nachrichten ermöglicht [7, 9].

2.2.3 Kommunikationsmuster und Fehlerbehandlung

SOME/IP unterstützt verschiedene Kommunikationsmuster, wie zum Beispiel die Request/Response-Kommunikation, in welcher Anfragen eines Klienten von dem Server beantwortet werden. Daneben unterstützt SOME/IP auch Anfragen ohne Antwortnachrichten [7, 12]. Außerdem bietet SOME/IP Benachrichtigungen, welche ein Publish/Subscribe-Konzept beschreiben. Hier veröffentlichen Server einen Dienst, welcher von Klienten abonniert werden kann. In bestimmten Fällen sendet der Server nun Ereignisse an die jeweiligen Abonnenten, die entweder geänderte Werte oder Ereignisse, die aufgetreten sind, enthalten können [7].
Fehler werden bei SOME/IP entweder von der Anwendungs- oder der Kommunikationsschicht darunter behandelt. Deshalb unterstüzt SOME/IP zwei verschiedene Mechanismen zur Fehlerbehandlung: zum Einen das Senden von bestimmten Return Codes in den Antwortnachrichten von Methoden, die angeben ob eine Anfrage korrekt verarbeitet wurde oder ob ein Fehler und welcher Fehler aufgetreten ist. Zum Anderen können explizite Fehlernachrichten, die Fehlermeldungen in der Nutzlast enthalten, gesendet werden [7].

2.3 RIOT

2.3.1 Das Betriebssystem

Aufgrund der zunehmenden Verbreitung des Internets ist es möglich, verschiedenste Geräte mit unterschiedlichen Ressourcen und Fähigkeiten über das Internet der Dinge (IoT) miteinander zu verbinden. Vermehrt werden sogenannte eingeschränkte Geräte (engl. constrained devices) eingesetzt, die auf stark eingeschränkte Leistungs-, Speicher- und Verarbeitungsressourcen zurückgreifen [13]. RIOT ist ein Betriebsystem für diese eingebetteten Low-End-Geräte im IoT, welches von einer weltweiten Entwicklergemeinschaft entwickelt wird. Es zeichnet sich durch eine modulare Architektur aus, die um einen Mikrokernel gebaut ist, und kann auf Geräten ohne MMU (Memory Management Unit) und MPU (Memory Protection Unit) ausgeführt werden. Aktuell wird RIOT unter der Lizenz LGPLv2.1[1] als freie Software verbreitet. Somit kann jeder den von der RIOT Entwicklergemeinschaft entwickelten und verwalteten Open-Source-Code weitergeben, benutzen und verändern, wobei die Lizenz bestehen bleiben muss [14].

[1] https://github.com/RIOT-OS/RIOT/blob/master/LICENSE

RIOT nutzt offene und standardisierte Netzwerkprotokollspezifikationen (zum Beispiel IETF-Protokolle) und stützt sich auf die Einhaltung von relevanten Standards, um auf eine große Menge an Software von Drittanbietern zugreifen zu können [14, 15]. Zusätzlich gewährleistet RIOT Hardware-Abstraktion sowie eine Einheitlichkeit der Programmierschnittstellen (APIs) über die gesamte untersützte Hardware hinweg, um Code-Portabilität zu ermöglichen und auch die Code-Duplizierung zu minimieren. Außerdem verwendet RIOT die Programmiersprache C, was für einen geringen Ressourcenbedarf und eine einfache Programmierbarkeit sorgt. Darüber hinaus ist RIOT modular, da in sich geschlossene Bausteine definiert sind, die frei kombiniert werden können und gleichzeitig den Speicherbeschränkungen entsprechen. Des Weiteren wird ein statischer Speicher genutzt, was sowohl Zuverlässigkeit als auch Echtzeitanforderungen erfüllt. Hinzu kommt, dass Herstellerbibliotheken in der Regel vermieden werden, was Unabhängigkeit schafft und Codeduplizierung minimiert [14].

2.3.2 Die Softwarestruktur

RIOT ist in Softwaremodule aufgeteilt, welche zur Kompilierungszeit aggregiert werden. So können komplette Systeme, je nach Anwendungsfall, ausschließlich mit den benötigten Modulen gebaut werden. Daraus resultiert eine Reduktion des Speicherverbrauchs und der Gesamtkomplexität des Systems. Der Code in RIOT lässt sich in verschiedene Gruppen wie beispielsweise den Kern, Prozessoren und Boards unterteilen, wobei die Funktionalitäten innerhalb der Gruppen in Module aufgeteilt sind (siehe Abbildung 2.2). Der Begriff Board bezieht sich in RIOT auf ein IoT-Gerät als Ganzes und wird als Synonym für eine Hardware-Plattform verwendet. Eine minimale Konfiguration besteht dabei nur aus dem Kern-Modul und einem Prozessor- und Board-Modul. Alle weiteren Module sind optional [14].

2.3.3 Hardwareabstraktion

RIOTs Hardwareabstraktion teilt hardwareabhängigen Code in drei Kategorien auf: Boards, Prozessoren und Treiber. Alle Aspekte des verwendeten Mikrocontrollers wie beispielsweise die Interrupt-Behandlung, Systemtaktverwaltung und Treiber für Peripheriegeräte werden von der Prozessor-Abstraktion implementiert und konfiguriert. Prozessor-externe Komponenten wie Sensoren, Aktoren, Speicher oder Netzwerk-Transceiver, werden in RIOT als Softwaremodule von Treibern gesteuert. Jede Anwendung muss genau eine Instanz

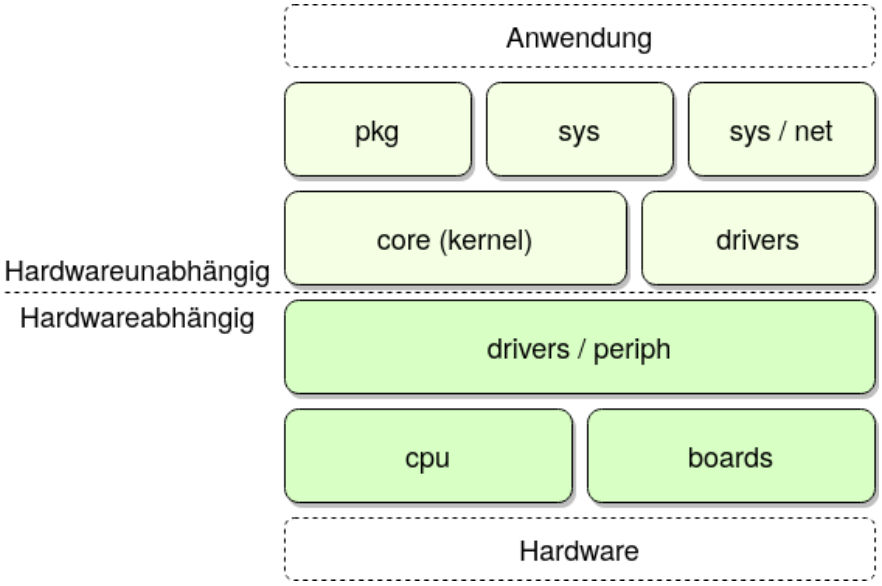

Abbildung 2.2: RIOT Architekturübersicht
Quelle: In Anlehnung an [14]

eines Boards und einer CPU implementieren. Darüber hinaus können keine oder mehrere Treiber enthalten sein [14].

2.3.4 Der Betriebssystemkern

Der Betriebssystemkern in RIOT bietet allgemeine Funktionalitäten des Multithreadings, wie zum Beispiel Kontextwechseln, Scheduling, Interprozesskommunikation (IPC) und Prozesssynchronisation (mithilfe von Mutexen, Semaphore und Nachrichten) [15]. Alle weiteren Komponenten, wie Gerätetreiber oder Anwendungslogik, werden vom Kern getrennt behandelt. Durch die minimalisitische API des Kerns ist es möglich mit diesen Komponenten zu interagieren. Die Threads in RIOT sind denen in Linux sehr ähnlich. Durch die Kern-API in RIOT kann jede Komponente, sei es ein Treiber oder anwendungsspezifische Logik, in einem seperaten Thread mit zugewiesener Priorität gestartet werden. So können Funktionalitäten logisch voneinander getrennt und Aufgaben einfach priorisiert werden. Zusätzlich erlaubt sie es, den Code leichter zu importieren. Durch statisches Allokieren von Speicher bei Datenstrukturen oder Vermeiden von rekursiven Funktionen wird die Speichernutzung während der Laufzeit minimiert. Daher ist es in RIOT möglich, Threads mit einer simplen Logik auszuführen, welche insgesamt nur wenig RAM-Speicher

(Random-Access Memory) verbrauchen. Mutexe, Semaphoren und Nachrichten sind Untermodule des Kerns und dienen der Interprozesssynchronisation. Sie sind daher optional und können je nach Anwendungsfall kompiliert werden, was den Speicherverbrauch gering hält. Falls ein sehr geringer Speicherverbrauch gefordert ist, kann ebenfalls komplett auf Multithreading verzichtet und die Benutzeranwendung als einziger Thread ausgeführt werden [14]. RIOTs Kern ermöglicht weiche Echtzeitfähigkeiten, indem die Zeit zum Unterbrechen und Wechseln auf andere Threads eine kleine Obergrenze nicht überschreitet, da es sich hier um deterministische Operationen mit konstanter Laufzeit handelt. Durch eine klassenbasierte Ausführung bis zum Abschluss werden Threads mit der höchsten Priorität zuerst ausgeführt und nur von Interrup-Service-Routinen (ISR) unterbrochen. Dabei können Threads mit niedriger Priorität von Threads mit einer höheren Priorität unterbrochen werden. So können Aufgaben priorisiert behandelt und den Aufgaben mit niedriger Priorität vorgezogen werden [14, 15].

2.3.5 Netzwerk-Subsystem

Das Netzwerk-Subsystem in RIOT bietet zwei externe Schnittstellen an: die Anwendungsprogrammierschnittstelle *sock* und die Gerätetreiber-API *netdev*. Diese Schnittstellen sind beide generisch, da der Netzwerkstack oder das Netzwerkgerät ausgetauscht werden können, ohne die Anwendung oder das Netzwerkuntersystem zu ändern [14]. Zusätzlich gibt es die einheitliche Schnittstelle *netapi*, welche das Interagieren der Protokollschichten innerhalb des Netzwerkstacks erlaubt (siehe Abbildung 2.3) [16].
Sock ist eine Sammlung bestehend aus High-Level-Netwerkzugriffs-APIs. Die Schnittstelle wurde anhand von Einschränkungen, wie zum Beispiel der ausschließlichen Nutzung von statischem Speicher, hoher Portabilität durch common types und Definitionen aus libc oder POSIX und Benutzerfreundlichkeit, entwickelt. *Sock* unterstützt derzeit rohen IP-Verkehr sowie TCP- und UDP-Verkehr, wobei jede API gesondert genutzt werden kann [14, 16].
RIOT abstrahiert einzelne Geräte über *netdev*, sodass Netzwerkstacks Zugriff auf Netzwerkschnittstellen erhalten. Gleichzeitig bleibt *netdev* dabei neutral in Bezug auf die Verbindungsschicht und der Netzwerktechnologie. *Netdev* übermittelt vollständige Datenrahmen einschließlich der Sicherungsschicht-Paketköpfe in einem Puffer, der vom aufrufenden Netzwerkstack bereitgestellt wird. Die Schnittstelle lässt sich in die Handhabung von Netzwerkdaten, Konfiguration und Initialisierung von Netzwerkgeräten und die Ereignisbehandlung zerlegen. Durch die Kombination dieser drei Aspekte wird *netdev* in

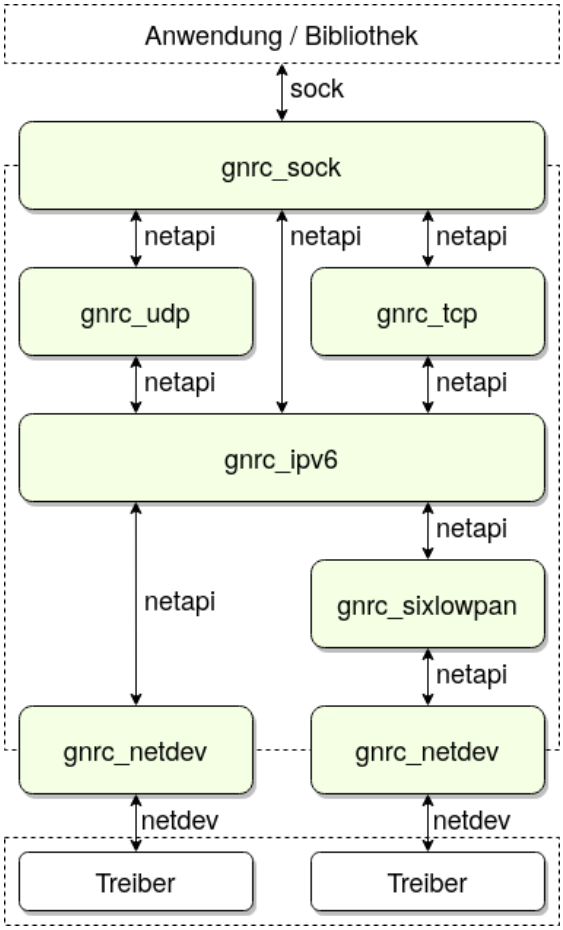

Abbildung 2.3: GNRC-Netzwerksubsystem
Quelle: In Anlehnung an [16]

Bezug auf die Funktionalität der Netzwerkgeräte abgerundet und ermöglicht gleichzeitig die vollständige Steuerung dieser Geräte [16].
Mithilfe der einheitlichen Schnittstelle *netapi* können die Protokollschichten innerhalb des Stacks über Nachrichten miteinander interagieren. Hierfür sind in *netapi* sowohl asynchrone Nachrichtentypen, die für die Kommunikation von Paketdaten genutzt werden, als auch synchrone Nachrichten, die eine Antwort erwarten, definiert [16].

2.3.6 GNRC

Der Generic Network Stack (GNRC) (siehe Abbildung 2.3) ist der Standard-Netzwerkstack in RIOT. Er kapselt jedes Protokoll in einem eigenen Thread und ermöglicht so

das Hot-Plugging von weiteren Netzwerkmodulen. GNRC nutzt RIOTs threadorientierte Interprozesskommunikation durch Nachrichtenwarteschlangen in jedem Thread und die in *netapi* definierten Nachrichtenformate [16]. Neben GNRC bietet RIOT außerdem einen vollständigen CAN-Netzwerkstack[2] an, welcher Anwendungs- und CAN-Geräte-Schnittstellen implementiert.
Das Register *netreg* leitet die Verarbeitungskette eines Paketes zwischen den verschiedenen Modulen [16, 15]. Bei *netreg* handelt es sich um eine Nachschlagetabelle, welche die Typ-ID eines Protokolls auf eine Liste von Prozess-IDs abbildet. Module, die an einem gewissen Typen interessiert sind, können sich für diesen mit einem Demultiplexing-Kontext, wie einem Port in UDP, registrieren. Sollte ein Thread eine Nachricht an einen bestimmten UDP-Port senden wollen, schlägt er den Protokolltypen im *netreg* nach und sucht nach dem Demultiplexing-Kontext, was in diesem Fall der angefordete UDP-Port ist. Jede Prozess-ID des Registers, auf die beide Parameter zutreffen, ist somit ein Ziel für die Nachricht [16].
GNRC nutzt einen flexiblen und zentralen Paketpuffer names *pktbuf*, um Paketkomponenten während der Stackdurchquerung zu speichern. *Pktbuf* handhabt Blöcke mit variabler Länge, um sogenannte *Snips* zu bearbeiten, welche Bruchteile von Paketen sind. Normalerweise werden diese *Snips* genutzt, um die verschiedenen Headern und Nutzlasten zu repräsentieren [16, 15]. Da feste Paketgrößen vermieden und die Länge der Pakete dynmisch zugwiesen werden, wird Speicher gespart, der ansonsten möglicherweise nicht in seiner voller Länge hätte verwendet werden können. Duplikate können vermieden werden, da *Snips* wiederverwendbar sind und jede Protokollimplementierung in der Lage ist, auf die *Snips* zuzugreifen, anstatt ihren eigenen Puffer zu verwalten. Diese flexible Art der Speicherung verzichtet somit auf das interne Kopieren von Daten. Zusätzlich bietet es einen effizienten und strukturierten Zugriff auf Protokollinformationen und hilft bei komplexen Protokollfunktionen, wie der Fragmentierung oder erneuter Übertragung von Daten. Außerdem ermöglicht *pktbuf* mehrere Pakete als eines zu behandeln, da die Anzahl der Pakete nicht direkt mit der Größe des Puffers verknüpft ist [16].

[2] http://doc.riot-os.org/group__sys__can.html

2.4 Controller Area Network

2.4.1 CAN-Spezifikation

Das Controller Area Network (CAN) ist ein serielles Kommunikationsprotokoll, das zu den Feldbussen gehört und eine verteilte Echtzeitsteuerung mit hoher Sicherheit bietet [17, 18, 19]. Feldbusse reduzieren die Komplexität der zugehörigen Verdrahtungen durch beispielsweise einfache Installation der Kabel sowie einfaches Hinzufügen weiterer Steuereinheiten. Seit 1983 wurde CAN von der Robert Bosch GmbH als Lösung für den steigenden Bedarf an Kommunikation und Reduzierung der Komplexität der Kabelbäume im Kraftfahrzeug entwickelt. CAN wurde erstmals im Februar 1986 auf der SAE-Konferenz (Society of Automotive Engineers) in Detroit vorgestellt und erschien als Version CAN 1.1, welche zu CAN 2.0A, sowie CAN 2.0B weiterentwickelt wurde [18, 20]. Der Anwendungsbereich reicht von Hochgeschwindigkeitsnetzwerken bis hin zu kostengünstigen Multiplex-Verkabelungen. Komponenten wie Motorsteuergeräte, Sensoren und Antiblockiersysteme können im Automobil über CAN mit Bitraten von bis zu 1 Mbit/s miteinander verbunden werden [17].

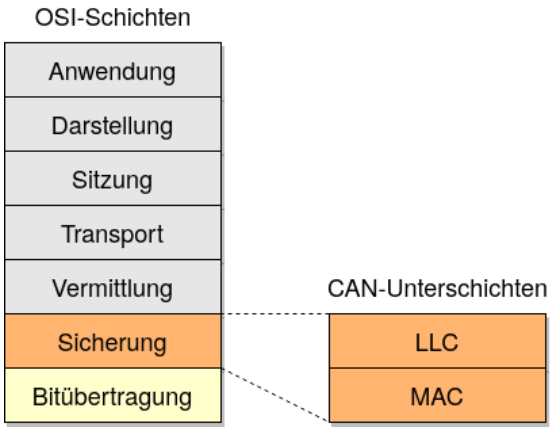

Abbildung 2.4: CAN-Unterschichten im OSI-Modell
Quelle: Eigene Darstellung

Um Gestaltungstransparenz und Implementierungsflexibilität zu schaffen, wurde CAN in die Sicherungs- und Bitübertragungsschicht des ISO/OSI-Referenzmodells unterteilt (siehe Abbildung 2.4) [19]. Zusätzlich wird die Sicherungsschicht in die Verbindungssteuerungsunterschicht (LLC) und die Medienzugriffssteuerungsschicht (MAC) unterteilt [17]. Die LLC-Unterschicht stellt sowohl einen Dienst für die Datenübertragung als auch für

entfernte Datenabfragen bereit. Daneben ist sie für die Nachrichtenfilterung zuständig und entscheidet, welche der empfangenen Nachrichten akzeptiert werden. Des Weiteren bietet sie Mittel zum Wiederherstellen verloren gegangener Daten an [17].
Die MAC-Unterschicht repräsentiert den Kern des CAN-Protokolls. Hier werden empfangene Nachrichten der LLC-Unterschicht präsentiert und Nachrichten akzeptiert, welche an die LLC-Unterschicht übertragen werden sollen. Innerhalb dieser Schicht wird entschieden, ob der Bus für den Beginn einer neuen Übertragung frei oder besetzt ist [17, 20]. Daneben ist sie für die Arbitrierung sowie die Fehlererkennung und Fehlersignalisierung zuständig. Eine Verwaltungseinheit namens Fehlereingrenzung (engl. Fault Confinement), ein selbstprüfender Mechanismus zur Unterscheidung von kurzen Störungen und dauerhaften Ausfällen, überwacht die MAC-Unterschicht und ist in der Lage, defekte Knoten auszuschalten [17].
Die Bitübertragungsschicht beinhaltet die tatsächliche Übertragung der Bits zwischen den verschiedenen Knoten eines Netzwerks. Sie definiert, wie Signale übertragen werden und beinhaltet daher die Beschreibung der Bit-Zeiten, der Bit-Kodierung und der Synchronisation eines Netzwerks. Die Bitrate kann zwischen verschiedenen Systemen variieren, aber innerhalb eines Systems muss sie uniform und fix sein. Neben der Bitrate ist hier auch der Abtastpunkt (engl. Sample Point) definiert. Dies ist der Zeitpunkt, an dem der Buspegel gelesen und als Wert des jeweiligen Bits interpretiert wird [17].
In CAN-Systemen nutzen anliegende Knoten keinerlei Informationen der Systemkonfiguration (wie beispielsweise Adressen). Dadurch können CAN-Netzwerke um Knoten erweitert werden, ohne dass Änderungen in der Soft- oder Hardware der Knoten oder der Anwendungsschicht vorgenommen werden müssen. Der Identifikator einer Nachricht sagt nichts über das Ziel der Nachricht aus, sondern über die Bedeutung und die Priorität der Nachricht. So können alle anliegenden Knoten durch Nachrichtenfilter entscheiden, ob auf empfangene Daten reagiert wird. Die Nachrichtenfilter erlauben außerdem, dass eine beliebige Anzahl an Knoten eine Nachricht gleichzeitig empfangen und auf diese reagieren können. Zusätzlich ist es in einem CAN-Netzwerk garantiert, dass eine Nachricht von entweder allen oder keinen Knoten akzeptiert wird, was Datenkonsistenz schafft. Um den Stromverbrauch des Systems zu reduzieren, können CAN-Geräte ohne interne Aktivitäten und mit getrennten Bustreibern in den Schlafmodus versetzt werden. Durch Busaktivitäten oder interne Bedingungen des Systems wird der Schlafmodus beendet und das Gerät wacht wieder auf [17].
CAN arbeitet nach einem Multimaster-Prinzip, sodass jeder Knoten eines Netzwerkes mit der Übertragung einer Nachricht beginnen kann, sobald der Bus frei ist und sich im Leerlauf befindet. Wenn mehr als ein Knoten gleichzeitig eine Übertragung startet,

wird der Buszugriffskonflikt durch die bitweise-Arbitrierung des Identifikators gelöst und der Knoten mit der höchsten Priorität gewinnt den Zugriff auf den Bus. Der Gewinner der Arbitrierung muss seine Übertragung nicht erneut von vorne beginnen, was garantiert, dass keine Daten oder Zeit verloren gehen. Während der Arbitrierung senden die konkurrierenden Knoten ihren Identifikator bitweise auf den Bus und versuchen so, den entsprechenden Buspegel ihres aktuellen Bitwerts auf den Bus zu legen [17, 19]. Dominante Pegel (logische „0“) überschreiben dabei rezessive Buspegel (logische „1“) [20]. Die sendenden Knoten vergleichen, ob der aktuelle Buspegel mit dem eigenen gesendeten Bit übereinstimmt. Knoten haben die Arbitrierung verloren und müssen die Übertragung unterbrechen, wenn dies nicht der Fall ist. Nun müssen sie warten, bis der Bus wieder frei ist und die nächste Arbitrierungsphase beginnt, um erneut senden zu können [17, 18].

2.4.2 Nachrichtenformat

Informationen auf dem Bus werden über einen Kanal und in einem fest definierten Nachrichtenformat mit variabler, aber begrenzter Länge gesendet. CAN bietet zwei verschiedene Nachrichtenformate, die sich in der Länge des Identifikators unterscheiden. Nachrichten mit einem 11 Bit langen Identifikator (CAN 2.0A) werden Standard-Frames und Nachrichten mit einem erweiterten 29 Bit langen Identifikator (CAN 2.0B) werden Erweiterte-Frames genannt. Darüber hinaus wird die Nachrichtenübertragung durch vier

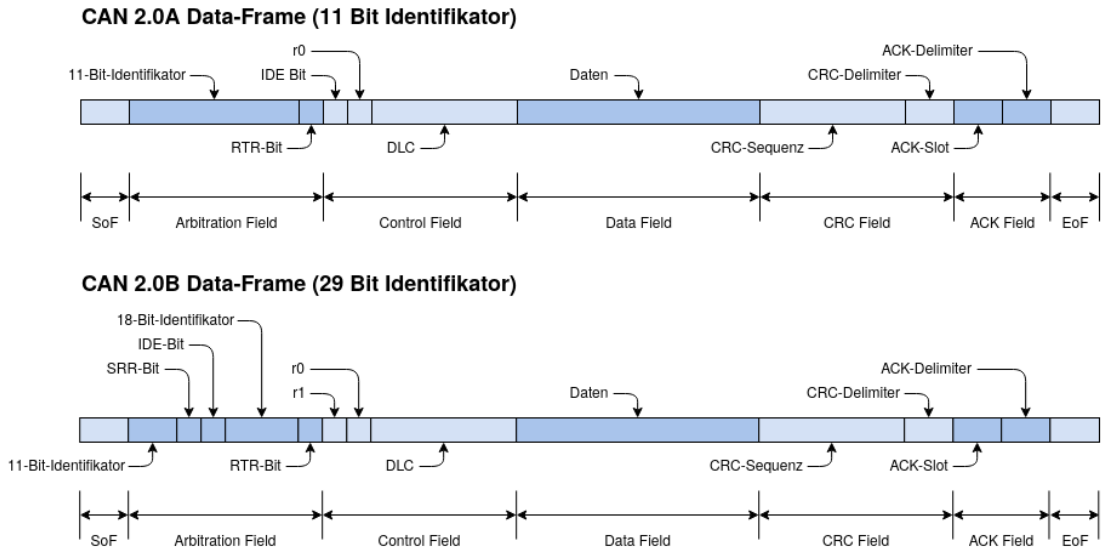

Abbildung 2.5: Aufbau der CAN Data Frames
Quelle: Eigene Darstellung

verschiedene Nachrichtentypen gesteuert [17, 19]:

- **Data Frames** transportieren Nutzdaten über den Bus an die Knoten, welche nach dem Identifikator filtern (siehe Abbildung 2.5). Sie können im Standard- und auch im erweiterten Frameformat genutzt werden.
- Knoten fordern mit **Remote Frames** andere Knoten dazu auf, die Übertragung eines Data Frames mit gleichem Identifikator zu starten. Sollte ein Knoten kurzzeitig inaktiv gewesen sein, kann er somit die aktuellen Datenwerte eines anderen Knotens anfordern und ist nicht darauf angewiesen, auf die nächste Nachricht dieses Knotens zu warten. Zusätzlich wird dadurch die Datenkonsistenz im Netzwerk gewährleistet, da alle Knoten, die an der Information interessiert sind, die Antwort des Senders erhalten. Remote-Frames können im Standard- und auch im erweiterten Frameformat genutzt werden [17, 18].
- **Error Frames** werden von Knoten beim Erkennen von Fehlern gesendet [17].
- **Overload Frames** liefern eine zusätzliche Verzögerung zwischen zwei aufeinanderfolgenden Data oder Remote Frames [17].

Data Frames bestehen in CAN aus insgesamt sieben verschiedenen Bitfeldern [17]:

- Das Feld **Start of Frame (SoF)** markiert den Beginn eines Data oder Remote Frames und beinhaltet ein einziges dominantes Bit. Alle anderen Knoten müssen sich auf den anführenden Pegel des SoF synchronisieren, welcher durch den sendenden Knoten verursacht wird [17].
- Im **Arbitration Field** ist der Identifikator, also die Priorität und die logische Adresse der Information enthalten. Im Standard-Frameformat besteht das Feld aus einem 11 Bit langen Identifikator und dem Remote-Transmission-Request-Bit (RTR-Bit) [17]. Das RTR-Bit gibt an, ob es sich bei der Nachricht um einen Data oder Remote Frame handelt [20]. Es wird im Data Frame dominant gesendet. Data Frames im erweiterten Frameformat beinhalten einen 29 Bit langen Identifikator, das Substitute-Remote-Request-Bit (SRR-Bit), das Identifier-Extension-Bit (IDE-Bit) und das RTR-Bit. Das SRR-Bit steht im Erweiterten-Frameformat an Stelle des RTR-Bits und ersetzt dieses. Das IDE-Bit gibt an, ob es sich bei der Nachricht um das Standard- oder erweiterte Frameformat handelt [17].

- Das **Control Field** besteht insgesamt aus sechs Bits. Es enthält reservierte Bits, den Data Length Code (DLC) und je nach verwendetem Frameformat das IDE-Bit. Der DLC besteht aus vier Bits und gibt die Anzahl an Bytes an, welche übertragen werden sollen. Die zulässige Anzahl an Datenbytes liegt dabei zwischen 0 und 8 inklusive. Alle anderen Werte sind nicht erlaubt [17].
- Das **Data Field** enthält die zu sendenden Daten eines Data Frames. Das Feld kann zwischen 0 und 8 Bytes an Daten enthalten [17].
- Das **CRC Field** enthält die Cyclic-Redundancy-Code-Sequenz (CRC-Sequenz), welche eine Prüfsumme zur Fehlererkennung ist, gefolgt von einem Cyclic-Redundancy-Code-Delimiter (CRC-Delimiter), der das Feld beendet. Die CRC-Sequenz enthält das Ergebnis der CRC-Berechnung des Senders. Empfänger berechnen die CRC-Sequenz auf die gleiche Weise. [17].
- Das **ACK Field** ist zwei Bits lang und beinhaltet den Acknowledge-Slot (ACK-Slot) und den Acknowledge-Delimiter (ACK-Delimiter). Während des ACK-Slots legt der Sender einen rezessiven Bitpegel auf den Bus und erwartet, dass die Busteilnehmer diesen mit einem dominanten Bitpegel überschreiben und somit den Erhalt der Nachricht bestätigen. Das Quittieren der Nachricht gibt dem Sender aber nur Auskunft darüber, dass mindestens ein zugehöriger Knoten die Nachricht korrekt empfangen hat. Der ACK-Delimiter beendet das Feld [17, 19].
- Das Feld **End of Frame (EoF)** markiert das Ende jedes Data oder Remote Frames durch sieben aufeinanderfolgende rezessive Bits [17].

Da die Bit-Kodierung keine Informationen über den Bittakt enthält, werden die Felder von SoF bis CRC nach dem Prinzip des Bit-Stuffings kodiert, um Synchronisation zwischen den verschiedenen Knoten zu schaffen. Aus diesem Grund wird, sobald ein Sender fünf aufeinanderfolgende gleiche Bits sendet, ein weiteres komplementäres Bit in den Bitstrom eingefügt. Dieser Flankenwechsel dient der Nachsynchronisation der anliegenden Knoten während einer Datenübertragung. Alle übrigen Felder behalten dabei ihre definierte Form. Die zusätzlich anfallenden Bits müssen auf Empfängerseite daraufhin wieder herausgefiltert und entfernt werden, da sich sonst der Inhalt der Nachricht ändern würde [17, 18].
Innerhalb des Aufbaus ähneln sich Remote Frames und Data Frames, jedoch verzichten Remote Frames auf das Data Field. Daher kann der DLC einen beliebigen Wert zwischen

0 und 8 annehmen. Im direkten Vergleich zu Data Frames wird das RTR-Bit hier rezessiv gesendet [17, 19]. Zwischen Anfrage und Antwort können unter Umständen mehrere höherpriorisierte Nachrichten auf dem Bus liegen [18].
Error Frames verletzen die Formatierungsregeln von CAN-Nachrichten und bestehen aus insgesamt zwei verschiedenen Feldern [19, 20]. Das erste Feld besteht aus der Überlagerung von gesendeten Fehlerflanken verschiedener Stationen. Dabei wird zwischen aktiven und passiven Fehlerflanken unterschieden. Aktive Fehlerflanken bestehen aus sechs aufeinanderfolgenden dominanten Bits. Passive Fehlerflanken enthalten dagegen sechs aufeinanderfolgende rezessive Bits, außer, sie werden von einem dominanten Bit eines anderen Knotens überschrieben. Das zweite Feld schließt den Error Frame ab [17].
Overload Frames bestehen analog zu Error Frames aus zwei Bitfeldern, die Overload-Flanken enhalten und das Frame abschließen [20]. Es gibt insgesamt zwei Bedingungen, die zu der Generierung einer Overload-Flanke führen [17]:

- Die internen Bedinungen eines Empfängers, die eine Verzögerung des nächsten Data oder Remote Frames erfordert [20].
- Erkennung eines dominanten Bits während des ersten oder zweiten Bits einer Übertragungspause des eigenen Sendevorgangs [17].

Die Overload-Flanken entsprechen in ihrer Form der den aktiven Fehlerflanken und bestehen aus sechs dominanten Bits [17].

2.4.3 Fehlerbehandlung

Um größtmögliche Sicherheit bei der Datenübertragung zu erreichen, sind in jedem CAN-Knoten Maßnahmen zur Fehlererkennung, Fehlersignalisierung und Selbstüberprüfung implementiert [17]. Fehlerhafte Nachrichten werden von Knoten, die einen Fehler erkennen, mit Flanken gekennzeichnet. Dies sorgt dafür, dass solche Nachrichten abgebrochen und automatisch erneut gesendet werden, sobald der Bus wieder frei ist [19]. Es gibt dabei fünf verschiedene Fehlertypen, welche sich gegenseitig nicht ausschließen [17]:

- Sendende Knoten überwachen auch gleichzeitig den Bus, auf den sie Bits senden. Ein **Bit-Error** liegt vor, wenn sich der überwachte Bitpegel von dem gesendeten Bitpegel unterscheidet. Ausgenommen davon ist die Arbitrierung und der ACK-Slot [17, 19].

- Wenn mehr als fünf aufeinanderfolgende gleiche Bits erkannt werden, liegt ein **Stuff-Error** vor, da so das Prinzip des Bit-Stuffings verletzt wird [17, 18].
- Ein **CRC-Error** liegt vor, wenn der Wert des berechneten CRCs nicht mit dem empfangenen übereinstimmt [17]. Dabei dient die CRC-Prüfsumme lediglich der Fehlererkennung und nicht der Fehlerkorrektur [18].
- Ein **Form-Error** wird erkannt, wenn eine Verletzung des vordefinierten Datenformats vorliegt [17, 18].
- Wenn ein Sender während des ACK-Slots kein dominantes Bit erkennt, liegt ein **Acknowledgment-Error** vor [17, 18].

3 Anforderungen

Damit das Gateway seine Aufgabe als Schnittstelle zwischen CAN- und Ethernet- oder IP-basierten Netzwerken erfüllt, muss es bestimmten Anforderungen gerecht werden. Dieses Kapitel zeigt die für diese Arbeit benötigten funktionalen und nichtfunktionalen Anforderungen auf, welche für die Umsetzung des Gateways und die Einhaltung der Rahmenbedingungen benötigt werden.

3.1 Funktionale Anforderungen

Das Gateway muss die Anforderungen von CAN 2.0A und CAN 2.0B erfüllen, um in der Lage zu sein, mit CAN-basierten Netzwerken zu kommunizieren. Dies beinhaltet die Umsetzung der Übertragungsverfahren und des Synchronisationsverhaltens sowie der Bit-Kodierung, des Zeitverhaltens und der Fehlerbehandlung. Daneben muss das Gateway Nachrichten sowohl mit einem Standard- (CAN 2.0A) als auch mit einem erweiterten Identifikator (CAN 2.0B) verstehen können. Außerdem ist es notwendig, die Bitrate und den Sample Point modular als Konfigurationsparameter zu gestalten, um in verschiedenen CAN-Netzwerkkonfigurationen interagieren zu können.
Für die Kommunikation mit Ethernet- oder IP-basierten Netzwerken muss das Gateway auf einer Netzwerkimplementierung aufsetzen, welche die erforderlichen Netzwerkschichten implementiert. Darüber hinaus muss das Gateway für die Kommunikation in IP-Netzwerken das SOME/IP-Format nutzen, wobei der SOME/IP-Paketkopf in der aktuellen Protokoll-Version (Version 1) verwendet werden soll. Die Kommunikation soll hier mit den Internetprotokollen IPv4 und IPv6 möglich sein.
Weiterleitungsinformationen, wie Zielports und Ziel-IP-Adressen, werden benötigt, damit das Gateway empfangene Nachrichten zuverlässig an die jeweiligen Zielnetzwerke weiterleiten kann. Dabei sollen dem Gateway diese Informationen während der Laufzeit übergeben werden.

Des Weiteren soll das Gateway mit den limitierten Leistungs-, Speicher-, und Verarbeitungs-Ressourcen eines Steuergerätes im Automobil ausführbar sein, wie beispielsweise die des in dieser Arbeit verwendete Evaluierungsboards Nucleo-F207ZG[1], welches einen Arm® 32-Bit Cortex®-M3 Kernprozessor verwendet und 1 Megabyte an Flashspeicher besitzt. Dieses Evaluierungsboard ist damit laut der Terminologie für eingeschränkte Netzwerkknoten ein Gerät der Leistungsklasse 2 [13].

3.2 Nichtfunktionale Anforderungen

Da es sich bei Automobilen um verteilte Echtzeitsysteme handelt, in denen beispielsweise Daten von Sensoren für die Umgebungserkennung genutzt werden, muss das Gateway weiche Echtzeitanforderungen erfüllen. Diese umfassen eine maximale Latenz über dem Gateway, sowie ein minimaler Jitter. Zusätzlich soll die Paketverlustrate und somit der Informationsverlust so gering wie möglich gehalten werden. Daneben soll das Gateway leicht erweiterbar sein, wenn zusätzliche Funktionalität in der Zukunft benötigt wird. Die einfache Verständlichkeit des Gateways muss darüber hinaus gegeben sein, sodass Benutzer die Funktionen leicht erlernen und das Gateway somit bedienen können. Des Weiteren muss das Gateway interoperabel sein, um Informationen mit anderen Komponenten oder Systemen austauschen und nutzen zu können.

[1] https://www.st.com/en/evaluation-tools/nucleo-f207zg.html

4 Implementierung

Ziel des vierten Kapitels ist es, die Implementierung des Gateways vorzustellen. Dies erfolgt anhand der Erläuterung der Konfiguration des Netzwerkstacks und der Ausführung der Initialisierung und Laufzeitkonfiguration. Abschließend wird der Nachrichtenempfang und der Nachrichtenversand des Gateways beschrieben.

4.1 Konfiguration des Netzwerkstacks

Das Gateway muss während der Kompilierzeit für seinen Einsatz und seine Bereitstellung konfiguriert werden. Aus diesem Grund können verschiedene Betriebsmodi für das Gateway gewählt werden, sodass das Gateway entweder als Schnittstelle zwischen CAN und Ethernet- oder IPv6-basierten Netzwerken oder als Schnittstelle zwischen CAN und IPv4-basierten Netzwerken, genutzt werden kann (siehe Abbildung 4.1). In jedem Anwendungsfall wird der CAN-Netzwerkstack von RIOT verwendet, wobei sich die Anzahl der vorhandenen CAN-Schnittstellen je nach genutzter Hardware unterscheiden kann. Dagegen besitzt das Gateway aufgrund der genutzten Hardware (Nucleo-F207ZG) immer nur eine Netzwerkschnittstelle, über die Ethernet- oder IP-Kommunikation möglich ist.
Mit der Auswahl des Betriebsmodus wird gleichzeitig der dazugehörige Netzwerkstack über den Parameter GATEWAY_OPERATING_MODE ausgewählt. Zur Auswahl stehen hierbei der RIOT-Netzwerkstack GNRC und der Open-Source-TCP/IP-Netzwerkstack LwIP[1] (Lightweight IP stack). Nutzt das Gateway GNRC als Netzwerkstack, ist es in der Lage, über die Anwendungsschnittstelle *sock* IP-Kommunikation zu betreiben. Dabei ist das zugrunde liegende Internetprotokoll IPv6 und das genutzte Transportprotokoll UDP. Gleichzeitig ermöglicht GNRCs Schnittstelle *netapi* die Interaktion mit der Netzwerkschnittstelle, womit reine Ethernet-Kommunikation möglich ist. Somit ist in diesem Fall die Kommunikation mit der Netzwerkschnitstelle über Ethernet und IPv6 gleichzeitig

[1] https://www.nongnu.org/lwip/2_1_x/index.html

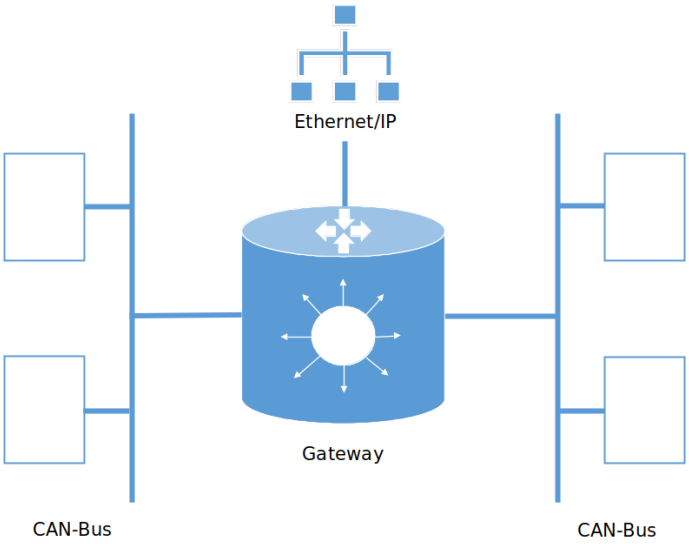

Abbildung 4.1: Position des Gateways als Schnittstelle zwischen den verschiedenen Netzwerktypen
Quelle: Eigene Darstellung

möglich. Durch die Wahl von LwIP als Netzwerkstack ist das Gateway ebenfalls in der Lage, über die Anwendungsschnittstelle *sock* IP-Kommunikation zu tätigen, wobei hier das Internetprotokoll IPv4 mit dem Transportprotokoll UDP genutzt wird. Die reine Ethernet-Kommunikation ist im Falle von LwIP nicht möglich, da dieser Netzwerkstack auf die IP-Kommunikation ausgelegt ist. In beiden Betriebsmodi wird die Anwendungsschnittstelle *sock* verwendet, welche von GNRC oder dem Netzwerkstack LwIP abstrahiert. UDP wird als Transportprotokoll gewählt, da innerhalb eines Fahrzeuges viele Anwendungen eine sehr kurze Zeitspanne erfodern, um schnell reagieren zu können. Diese Anforderungen werden von UDP besser abgedeckt als von TCP. Außerdem ist es bei zyklisch gesendeten Daten ein besserer Ansatz, im Falle einer fehlerhaften Übertragung auf die nächste Übertragung zu warten, als die beschädigten Daten zu reparieren.
Zusätzlich benötigt jede CAN-Schnittstelle einen CAN-Transceiver, der die Busankopplung zwischen dem genutzten Evaluierungsboard und den physikalischen Drähten der CAN-Busleitung realisiert. Daher werden die Pins der CAN-Transceiver während der Konfiguration definiert, welche später benötigt werden, um die CAN-Transceiver zu initialisieren. Die CAN-Schnittstellen können dabei unterschiedliche CAN-Transceivermodelle nutzen. Des Weiteren werden während der Konfiguration die Bitrate und der Sample

Point der CAN-Schnittstellen und der Port des Sockets für die IP-Kommunikation definiert.

4.2 Initialisierung und Laufzeitkonfiguration

Mit der Initialisierung werden die genutzten CAN-Transceiver initialisiert und CAN-Bitraten eingestellt sowie Empfangsthreads gestartet und CAN-Identifikatoren mit Zielschnittstellen-Informationen einer Identifikatorliste hinzugefügt, wodurch das Gateway bereit ist, mit anliegenden Schnittstellen interagieren zu können. Die Initialisierung des Gateways findet in der Funktion init_gateway() statt.

4.2.1 CAN-Transceiver-Initialisierung

Am Anfang der Initialsierung werden die genutzten CAN-Transceiver initialisiert und in den gewünschten Empfangsmodus versetzt. Dazu werden die in der Konfiguration definierten Pins aller CAN-Transceiver, welche in dem Array *devs[] definiert sind, mithilfe von can_trx_init() als Ausgabe-Pins konfiguriert. Anschließend werden die CAN-Transceiver per can_trx_set_mode() in den in TRX_MODE benutzerdefinierten Empfangsmodus gebracht. Wenn alle Schritte erfolgreich ausgeführt wurden, sind die CAN-Transceiver nun vollständig initialisiert, wobei Fehler zum Abbruch der Initialisierung des Gateways führen.

4.2.2 CAN-Bitzeiteinstellungen

Als Nächstes werden die nötigen Bitzeiteinstellungen aller vorhandenen CAN-Schnittstellen mithilfe von _set_can_bitrate() vorgenommen. Dazu bekommt die Funktion die jeweilige CAN-Schnittstelle, die in der Konfiguration definierte Bitrate CAN_BITRATE sowie den in der Konfiguration definierten Sample Point CAN_SAMPLE_POINT als Parameter übergeben. Über raw_can_set_bitrate() werden hier die Bitzeiteinstellungen für die jeweilige CAN-Schnittstelle vorgenommen. Fehler beim Setzen der Parameter führen zum Abbruch der Initialisierung des Gateways.

4.2.3 Starten der Empfangsthreads

Nachdem die Bitzeiteinstellungen erfolgreich vorgenommen wurden, werden die Empfangsthreads für jede benötigte Schnittstelle mit thread_create() gestartet. Dabei besitzt jeder Thread eine Stackgröße von 1536 Bytes sowie eine Priorität von THREAD_PRIORITY_MAIN - 1. Außerdem verwaltet jeder Thread eine eigene Nachrichtenwarteschlange für die Interprozesskommunikation mit insgesamt acht Plätzen. Eine Warteschlange kann also bis zu acht empfangene Nachrichten gleichzeitig halten, bis empfangene Nachrichten verworfen werden.
Zunächst werden die CAN-Empfangsthreads (*_can_receive_thread()) für alle vorhandenen CAN-Schnittstellen gestartet. Gleichzeitig speichert das Array can_receive_pid[] die jeweiligen Prozess-Identifikatoren der Threads für die spätere Benutzung ab. Sobald die CAN-Empfangsthreads gestartet wurden, warten sie auf den Empfang einer Nachricht (IPC) des Typs CAN_START_RECV, sodass sie beginnen CAN-Nachrichten entgegenzunehmen.
Im Anschluss an das Starten der CAN-Empfangsthreads wird der IP-Empfangsthread (*_ip_receive_thread) gestartet. Der Thread erstellt hier ein UDP-Socket auf dem in SERVER_PORT definierten Port über sock_udp_create(), welches je nach Konfiguration entweder IPv4 (LwIP) oder IPv6 (GNRC) als Internetprotokoll verwendet. Nach dem Erstellen des Sockets erhält die Boolean-Variable server_running den Wert „true“. Außerderm wird beim Nutzen des Netzwerkstacks von LwIP eine statische IPv4-Adresse mit IP4_ADDR() vergeben, da es sich bei dem Automobil um ein statisches Netzwerk handelt. Bei IPv6 ist eine statische IP-Adressvergabe nicht nötigt, weil hier direkt die Link-Local-Adresse als IPv6-Adresse genutzt wird. Nach dem Starten des Sockets ist es möglich, Nachrichten über den Thread zu empfangen und versenden.
Abhängig von der Konfiguration des Gateways wird nun noch der Ethernet-Empfangsthread (*_eth_receive_thread) gestartet. Der daraus enstandende Prozess-Identifikator wird daraufhin genutzt um den Thread für den benutzerdefinierten Ethertype zu registrieren, was das Empfangen von Paketen dieses Ethertypes ermöglicht. Zusätzlich erlangt das Gateway per netif_iter() Zugriff auf die Netzwerk-Schnittstelle, durch welche Nachrichten über Ethernet versendet werden.

4.2.4 Konfigurationswerkzeug

Das Gateway bietet die Möglichkeit, bestimmte Konfigurationen während der Laufzeit über den Shell-Kommando-Handler can_gateway_handler() vorzunehmen. Dazu bekommt das Gateway die verschiedenen Kommandos über eine serielle Schnittstelle von einem Konfigurationswerkzeug übergeben, die dann auf dem Gateway ausgeführt werden. Die serielle Schnittstelle wird bei dem Übergeben eines Kommandos über die Programmiersprache Python mit der Funktion Serial() aufgebaut. Als Parameter werden hier der jeweilige Zielport des Gateways und die Baudrate der aktuellen Übertragung übergeben. Nachdem die Verbindung erfolgreich aufgebaut wurde, können die verschiedenen Kommandos an das Gateway übertragen werden. Wenn beispielsweise die Bitrate und der Sample Point einer CAN-Schnittstelle nachträglich geändert werden müssen, kann dies über das Konfigurationswerkzeug erreicht werden, indem die Funktion _set_bitrate() mit den gewünschten Parametern auf dem Gateway aufgerufen wird. In jedem Fall wird die Verbindung zum Gateway aufgelöst, sobald das jeweilige Kommando übertragen wurde.

4.2.5 Konfiguration der Identifikatorliste

Damit das Gateway empfangene Nachrichten zuverlässig an die jeweiligen Zielschnittstellen weiterleiten kann, braucht es eine Identifikatorliste, welche die erforderlichen Zielschnittstelleninformationen der jeweiligen CAN-Identifikatoren enthält. Über das Konfigurationswerkzeug wird eine JSON-Datei (engl. JavaScript Object Notation), welche eine solche Liste enthält, eingelesen und die Einträge werden einzeln an das Gateway gesendet.

```
1  struct interfaces {
2      uint16_t dst_port;
3      #ifndef GATEWAY_OPERATING_MODE
4      ipv6_addr_t ipv6_addr;
5      bool eth_send;
6      #else
7      ipv4_addr_t ipv4_addr;
8      #endif
9      int8_t dst_ifnum;
10     uint8_t src_ifnum;
11 };
12
13 struct id {
```

```
14      uint32_t can_id;
15      struct interfaces interface_information;
16      struct id *next;
17  };
```

Listing 4.1: Aufbau eines Listeneintrages

Diese Einträge werden nun von dem Gateway als Parameter der Funktion _add_can_-id() dieser Identifikatorliste hinzugefügt. Die Identifikatorliste ist ein Array, welches die in ID_LIST_COUNT definierte Anzahl an Plätzen besitzt. Sollte diese Identifikatorliste voll sein, wird eine Fehlermeldung generiert und der aktuelle Eintrag wird verworfen. Wenn nicht, wird der Eintrag am Ende der Identifikatorliste eingefügt. Zusätzlich wird ein Zähler erhöht, der angibt, wie viele Einträge sich in der Identifikatorliste befinden. Identifikatorlisteneinträge besitzen dabei die in Listing 4.1 gezeigte Datenstruktur und enthalten neben den CAN-Identifikatoren zusätzlich Informationen darüber, ob und wohin empfangene Nachrichten des jeweiligen CAN-Identifikator gesendet werden sollen. Dazu enthalten Identifikatorlisteneinträge den Zielport und die Ziel-IP-Adresse (je nach Internetprotokoll IPv4 oder IPv6) sowie die Ziel-CAN-Busnummer und einen Wahrheitswert, ob die Nachricht per Ethernet versendet werden soll. Falls Nachrichten eines jeweiligen CAN-Identifikators nicht an alle Schnittstellen weitergeleitet werden sollen, können die entsprechenden Felder den Wert „0“(für Zielport, Ziel-IP-Adresse und Ethernet-Wahrheitswert) oder „-1“ (Ziel-CAN-Busnummer) erhalten. Ansonsten enthalten die Einträge noch die Quell-CAN-Busnummer, von welcher Nachrichten mit dem jeweiligen CAN-Identifikator erwartet werden.

4.2.6 Konfiguration der CAN-Schnittstellen

Um Nachrichten der CAN-Schnittstellen empfangen zu können, müssen die CAN-Identifikatoren aus der Identifikatorliste als Nachrichtenfilter gesetzt werden. Zusätzlich benötigen die CAN-Empfangsthreads ein Startsignal, damit sie beginnen, CAN-Nachrichten zu empfangen, welches über das Konfigurationswerkzeug mithilfe der Funktion _start_-can_receive() erreicht wird. Dazu wird als Erstes überprüft, ob der jeweilige CAN-Empfangsthread beschäftigt ist. Wenn dies nicht der Fall ist, werden für diesen CAN-Empfangsthread alle CAN-Identifikatoren aus der Identifikatorliste als Nachrichtenfilter verwendet, deren Quell-CAN-Busnummer mit der CAN-Schnittstelle des CAN-Empfangsthreads übereinstimmt. Die Anzahl an Nachrichtenfiltern pro CAN-Schnittstelle kann

dabei den vordefinierten Wert von FILTER_NUM nicht überschreiten. Sollten mehr CAN-Identifikatoren als zulässig in der Liste enthalten sein, werden die überschüssigen ignoriert und nicht als Nachrichtenfilter verwendet. Außerdem erzeugt dies eine Fehlermeldung für den Anwender. Anschließend wird mit conn_can_raw_create() ein CAN-Verbindungssocket aufgebaut, mit dem die Nachrichtenfilter an die aktuelle CAN-Schnittstelle gebunden werden. Zusätzlich wird der aktuelle CAN-Empfangsthread in dem Array can_thread_busy[] als beschäftigt markiert. Der CAN-Netzwerkstack von RIOT lässt das Definieren eines Zeitlimits zu, nach dessen Überschreitung der jeweilige CAN-Empfangsthread stoppen würde CAN-Nachrichten entgegenzunehmen. Daher ist das Zeitlimit in CAN_RECV_TIMEOUT immer mit einer „0“ definiert, da dieser Effekt so ausbleibt. Schließlich wird eine Nachricht an den jeweiligen CAN-Empfangsthread vom Typ CAN_START_RECV und mit dem vorher in CAN_RECV_TIMEOUT definierten Zeitlimit über msg_send() gesendet. Damit wird dem aktuellen CAN-Empfangsthread signalisiert, CAN-Nachrichten zu empfangen, welche CAN-Identifikatoren besitzen, die den Nachrichtenfiltern entsprechen. Wenn für eine der vorhandenen CAN-Schnittstelle keine CAN-Identifikatoren als Filter in der Identifikatorliste existieren, passiert nichts.

4.2.7 Änderungen an der Identifikatorliste

Soll der Inhalt der Identifikatorliste geändert werden, wird dies mithilfe des Konfigurationswerkzeugs erreicht. Dazu müssen zu Beginn die vorhandenen CAN-Empfangsthreads stoppen CAN-Nachrichten zu empfangen, was in der Funktion _close_can_connection() passiert. Erreicht wird dies mit der Funktion conn_can_raw_close(), welche alle vorhandenen CAN-Verbindungssockets schließt. Zusätzlich werden die jeweiligen CAN-Empfangsthreads als nicht beschäftigt markiert. Diese beiden Schritte sind für Änderungen an der Identifikatorliste notwendig, da ansonsten alte Nachrichtenfilter bestehen bleiben würden und den CAN-Empfangsthreads keine neuen Nachrichtenfilter übergeben werden könnten. Nun wird über die Funktion _delete_id_list() die vorhandene Identifikatorliste gelöscht, indem der Zähler, der angibt wie viele Einträge in der Identifikatorliste enthalten sind, den Wert „0“ erhält und das Element an dieser Position gelöscht wird. In der Zeit ohne Identifikatorliste werden keine CAN-Nachrichten des Gateways empfangen, da die CAN-Empfangsthreads keine Nachrichtenfilter besitzen. Über das Konfigurationswerkgzeug kann dem Gateway nun eine neue Identifikatorliste übergeben werden. Außerdem werden mit _start_can_receive() die neuen Nachrichtenfilter

der CAN-Schnittstelle gesetzt und den CAN-Empfangsthreads signalisiert, Nachrichten entgegenzunehmen.

4.3 Nachrichtenempfang

Nach der Initialisierung des Gateways können Nachrichten aus den entsprechenden Netzwerken über die Empfangsthreads empfangen werden.

4.3.1 CAN

Das Empfangen von CAN-Nachrichten findet über conn_can_raw_recv() in den CAN-Empfangsthreads *_can_receive_thread() statt (siehe Abbildung 4.2). Dabei werden

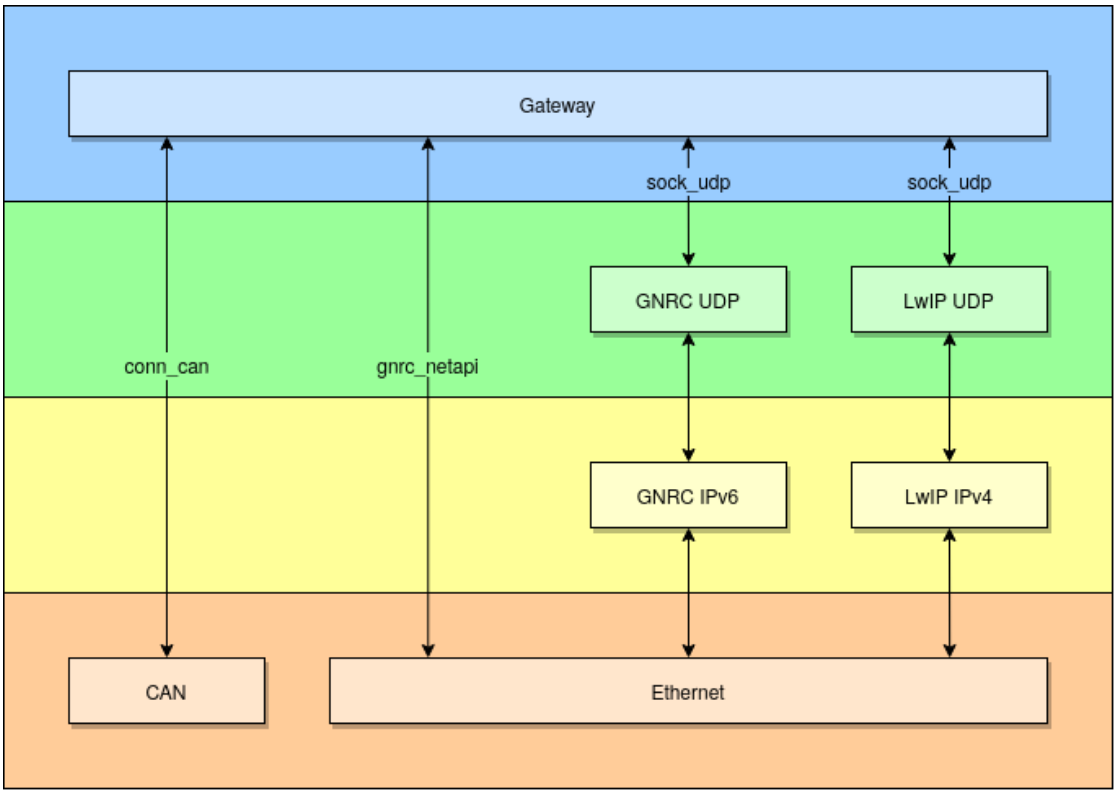

Abbildung 4.2: Übersicht der genutzten APIs
Quelle: Eigene Darstellung

ausschließlich Nachrichten empfangen, deren Identifikator den gesetzten Nachrichtenfiltern der jeweiligen Schnittstelle entsprechen. Alle übrigen Nachrichten werden ignoriert. Nach dem erfolgreichen Empfangen einer Nachricht wird mit _get_can_id() der zugehörige Identifikatorlisteneintrag gesucht. Sollte es keinen Eintrag zu dem empfangenen Identifikator geben, wird die Nachricht verworfen und es kann eine neue Nachricht empfangen werden. Wenn die Identifikatorliste aber den Identifikator enthält, wird anhand der Felder des Eintrags entschieden, über welches Netzwerk und wohin die Nachricht weitergeleitet werden soll. Wenn eine Ziel-CAN-Busnummer enthalten ist, wird die Nachricht mithilfe von _can_send() an die angegebene Schnittstelle weitergeleitet. Falls das Ethernet-Wahrheitswertfeld eine „1“ aufweist, wird die Nachricht mit _ethernet_send() in das Ethernetformat übersetzt und als reine Ethernet-Nachrichtversendet. Sollte sowohl ein Port als auch eine Ziel-IP-Adresse vorhanden sein, wird die Nachricht per _ip_send() in das SOME/IP-Format übersetzt über IP an die angegebenen Zielinformationen gesendet.

4.3.2 Ethernet

Ethernet-Nachrichten der Netzwerkschnittstelle werden von dem Ethernet-Empfangsthread *_eth_receive_thread() entgegengenommen. Dazu erhält der Thread eine Nachricht über die Schnittstelle *netapi* vom Typ GNRC_NETAPI_MSG_TYPE_RCV, sobald das Gateway eine Ethernet-Nachricht mit dem benutzerdefinierten Ethertype empfangen hat. Als Erstes muss diese Nachricht nun in das CAN-Format übersetzt werden, um sie

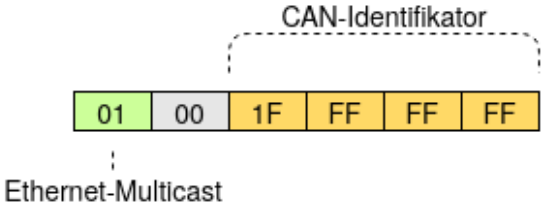

Abbildung 4.3: Aufbau einer MAC-Adresse mit dem größtmöglichen CAN-Identifikator einkodiert
Quelle: Eigene Darstellung

über eine CAN-Schnittstelle versenden zu können. Aus diesem Grund werden die Inhalte der empfangenen *Snips*, angefangen mit der Nutzlast der Nachricht, mit _conv_eth_-frame_to_can() in die Struktur can_frame eingepflegt. In _fill_can_data() wird der

DLC (Data Length Code) zu Beginn der Struktur hinzugefügt, welcher an erster Stelle der empfangenen Nachricht steht. Sollte der DLC eine ungültige Größe haben, wird dies hier erkannt und die Nachricht verworfen. Danach überträgt die Funktion mithilfe des DLCs den Inhalt der empfangenen Nachricht in die CAN-Struktur. Anschließend wird der CAN-Identifikator durch die Funktion gnrc_netif_hdr_get_dst_addr und per Bitschieben aus der Ziel-Adresse der empfangenen Nachricht gewonnen und in die CAN-Struktur eingefügt. Der CAN-Identifikator ist in den unteren vier Bytes der Zieladresse kodiert, da ein CAN-Identifikator maximal 29 Bit groß sein kann (siehe Abbildung 4.3). Nun sucht _get_can_id() den Identifikatorlisteneintrag des empfangenen Identifikators. Im Falle, dass der Identifiaktor größer als 11 Bit sein sollte, wird dieser anschließend mit CAN_EFF_FLAG als ein erweiterter Identifiaktor gekennzeichnet. Daraufhin wird die Nachricht mit _can_send() an die angegebene CAN-Schnittstelle gesendet. Falls der Identifikator nicht in der Identifikatorliste enthalten ist wird die Nachricht verworfen.

4.3.3 SOME/IP

Über die Funktion sock_udp_recv() der Schnittstelle *sock* werden Pakete über IP empfangen, die an den Port und die IP-Adresse des Gateways addressiert sind. Je nach Konfiguration nimmt der IP-Empfangsthread *_ip_receive_thread() diese Pakete über IPv4 oder IPv6 entgegen. Die Nutzlast dieser Pakete repräsentieret dabei jeweils eine SOME/IP-Nachricht, bestehend aus dem Paketkopf und der Nutzlast.

```
1 /**
2  * @brief       Structure of the Message ID
3  */
4 typedef struct {
5     uint16_t service_id; /**< Service ID */
6     uint16_t method_id;  /**< Method ID */
7 } someip_message_id_t;
8
9 /**
10  * @brief       Structure of the Request ID
11  */
12 typedef struct {
13     uint16_t client_id;  /**< Client ID */
14     uint16_t session_id; /**< Session ID */
15 } someip_request_id_t;
16
17 /**
```

```
18  * @brief       SOME/IP header
19  */
20 typedef struct __attribute__((packed)) {
21     someip_message_id_t message_id;   /**< Message ID */
22     uint32_t length;                  /**< Length */
23     someip_request_id_t request_id;   /**< Request ID */
24     uint8_t protocol_version;         /**< Protocol Version */
25     uint8_t interface_version;        /**< Interface Version*/
26     uint8_t msg_type;                 /**< Message Type*/
27     uint8_t return_code;              /**< Return Code*/
28 } someip_hdr_t;
```

Listing 4.2: SOME/IP-Paketkopfstruktur

Zunächst wird die Nutzlast empfangener Pakete durch Datentypumwandlung in die Typdefinition someip_hdr_t (siehe Listing 4.2) übersetzt. Daraufhin werden der SOME/IP Paketkopf und die Nutzlast der SOME/IP-Nachricht der Funktion _conv_someip_to_can() übergeben, welche die empfangene Nachricht in das CAN-Format übersetzt. Dazu wird zum Einen die Protokollversion des Pakets mit der des Gateways verglichen. Sollten sich die Versionen unterscheiden, wird das jeweilige Paket verworfen. Zum Anderen wird das Längenfeld des empfangenen Pakets auf seine Validität kontrolliert. Wenn diese kleiner als der SOME/IP-Paketkopf oder größer als die Summe aus dem SOME/IP-Paketkopf addiert mit der maximalen CAN-Nutzlastgröße CAN_MAX_DLEN ist, wird das Paket aufgrund einer ungültigen Länge ebenfalls verworfen. Sobald das Paket diese beiden Kontrollen passiert hat, wird die jeweilige Nachrichten ID als CAN-Identifikator in eine can_frame-Struktur eingefügt. Anschließend sucht _get_can_id() den zugehörigen Identifiaktorlisteneintrag des empfangenen Identifikators, wobei die Nachricht verworfen wird, wenn dieser nicht vorhanden ist. Sollte es sich um einen erweiterten Identikator handeln, wird dieser nun mit CAN_EFF_FLAG als solches gekennzeichnet. Daraufhin wird geprüft, ob der DLC, welcher aus der Nutzlastlänge der SOME/IP-Nachricht resultiert, im Rahmen der erlaubten acht Bytes einer CAN-Nachricht liegt. Wenn dies zutrifft wird der DLC der CAN-Struktur hinzugefügt, ansonsten wird die Nachricht fallen gelassen. Danach wird die Nutzlast der SOME/IP-Nachricht mithilfe des DLCs in das Datenfeld der CAN-Struktur eingefügt, sodass eine vollständige CAN-Nachricht vorliegt. Diese kann nun mit _can_send() an die Schnittstelle gesendet werden, welche im jeweiligen Identifikatorlisteneintrag angegeben ist.

4.4 Nachrichtenversand

Das Gateway sendet Nachrichten als Reaktion auf das Empfangen von Nachrichten über die verschiedenen Schnittstellen. Dabei geben die empfangenen CAN-Identifikatoren Aufschluss darüber, ob und an welche Schnittstelle die Nachricht gesendet werden soll.

4.4.1 CAN

Empfangene Nachrichten werden in der Funktion _can_send() an die geforderte CAN-Busnummer gesendet. Dazu bekommt die Funktion die zu sendende CAN-Nachricht im CAN-Format und die Ziel-CAN-Busnummer als Parameter übergeben. Zuerst wird hier überprüft, ob es sich bei der übergebenen Schnittstelle um eine registrierte Schnittstelle der genutzten Hardware handelt. Dazu wird die Schnittstelle mit der Anzahl der registrierten CAN-Schnittstellen in CAN_DLL_NUMOF verglichen, wobei die Nachricht verworfen wird, wenn die Schnittstelle einen Wert außerhalb des Rahmens besitzt. Anschließend wird per conn_can_raw_create() ein CAN-Verbindungssocket zu der übergebenen Schnittstelle aufgebaut, über die mit conn_can_raw_send() die Nachricht an die Zielschnittstelle gesendet wird. Sobald der Sendevorgang erfolgreich abgeschlossen ist, gibt die Funktion eine „0“ als Rückgabewert zurück.

4.4.2 Ethernet

Ethernet-Nachrichten werden innerhalb der Funktion _ethernet_send() über die Netzwerkschnittstelle gesendet. Hierbei bekommt die Funktion die zu sendende CAN-Nachricht im CAN-Format übergeben. Der CAN-Identifikator wird zunächst in die unteren vier Bytes der Ziel-MAC-Adressse kodiert. Zusätzlich wird durch das Setzen einer „1“ im LSB des ersten Bytes der Ziel-MAC-Adresse angegeben, dass es sich um Multicast-Nachrichten handelt, sodass sie an alle Knoten des Netzwerkes gesendet werden, die Teil der Ethernet-Multicast-Gruppe sind. Als Nächstes befüllt die Funktion die Nutzlast der Ethernet-Nachricht mit dem DLC und den Daten der übergebenen CAN-Nachricht. Da ein benutzerdefinierter Ethertype verwendet wird, der einen größeren Wert als 1536 hat, besitzen die Ethernet-Frames kein Längenfeld [21]. Aus diesem Grund ist der DLC in dem ersten Byte der Nutzlast kodiert, damit ein Empfänger weiß, wie viele Daten in dem jeweiligen

Ethernet-Frame enthalten sind. Auf den DLC folgen die eigentlichen Daten der übergebenen CAN-Nachricht. Nun werden die nötigen *Snips* über die Funktionen gnrc_pktbuf_add() und gnrc_netif_hdr_build() erstellt und mit dem benutzerdefinierten Ethertype GNRC_CUSTOM_ETH versehen, dass eine vollständige Ethernet-Nachricht vorliegt, welche über die Schnittstelle *netapi* mit gnrc_netapi_send() an die Netzwerkschnittstelle gesendet wird. Ein erfolgreicher Sendevorgang wird mit dem Rückgabewert „0“ quittiert.

4.4.3 SOME/IP

Nachrichten werden in der Funktion _ip_send() über IP versendet. Als Parameter bekommt die Funktion dabei die zu sendende CAN-Nachricht im CAN-Format sowie den Zielport und die Ziel-IP-Adresse übergeben. Je nach Konfiguration handelt es sich dabei um eine IPv4- oder IPv6-Adresse. Zunächst muss die CAN-Nachricht in das SOME/IP-

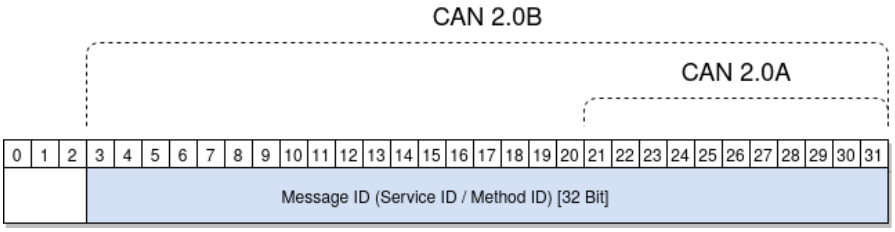

Abbildung 4.4: Aufbau einer SOME/IP Message ID bei einem kodierten CAN-Identifikator (CAN 2.0A und CAN 2.0B)
Quelle: Eigene Darstellung

Format someip_hdr (siehe Listing 4.2) übersetzt werden, was mit _conv_can_to_someip erreicht wird. Dazu wird der empfangene CAN-Identifikator in die Nachrichten-ID des Paketkopfs kodiert (siehe Abbildung 4.4). Anschließend wird das Längenfeld des SOME/IP-Paketkopfs mit der Summe aus dem CAN-DLC und der Länge des SOME/IP-Paketkopfs, aber ohne der Länge des Nachrichtenidentifikators und des Längenfeldes belegt. Nun werden alle weiteren Felder des SOME/IP-Paketkopfes mit den vorher festgelegten Einstellungen befüllt. Dies beinhaltet die Anfrage-ID, sowie die Protokoll- und Schnittstellenversion, Nachrichtentyp und den Rückgabecode. Als Nachrichtentyp ist hier der Typ „Notification“ gewählt, da diese Nachrichten keine Antwort erwarten, ähnlich zu der CAN-Kommunikation mit Data Frames. Zum Schluss werden die CAN-Daten der übergebenen CAN-Nachricht in die Nutzlast des Paketes übertragen, sodass ein vollständiges SOME/IP-Paket vorliegt.

Nachdem die Nachricht erfolgreich übersetzt wurde, wird nun der geforderte Zielpunkt definiert. Aus diesem Grund werden der Konfiguration des Gateways entsprechend der UNIX-Adressfamilien IPv4 (AF_INET) oder IPv6 (AF_INET6) gewählt und die übergebene IPv4- oder IPv6-Adresse als Ziel-IP-Adresse definiert. Außerdem wird der übergebene Port als Zielport gesetzt und die Schnittstelle *sock* über den Netzwerkstack-spezifischen-Identifikator (SOCK_ADDR_ANY_NETIF) gewählt. Im nächsten Schritt wird das Socket durch server_running auf Inaktivität geprüft. Falls dies zutreffen sollte, wird die Nachricht fallen gelassen. Das Paket wird nun im letzten Schritt per sock_udp_send() an die Anwendungsschnittstelle *sock* übergeben, welche an das ausgewählte Ziel sendet. Falls das Senden fehlschlagen sollte, wird der Server geschlossen und das Paket verworfen. Ein erfolgreiches Senden wird mit dem Rückgabewert „0“ bestätigt.

5 Versuchsaufbau

Das Gateway wurde in dieser Arbeit mit dem Evaluierungsboard Nucleo-F207ZG realisiert. Dieses Evaluierungsboard besitzt zwei CAN-Schnittstellen für die Kommunikation mit anliegenden CAN-Netzwerken, welche sowohl CAN 2.0A als auch CAN 2.0B mit einer Übertragungsgeschwindigkeit von bis zu 1 Mbit/s unterstützen. Dabei sind 14 ver-

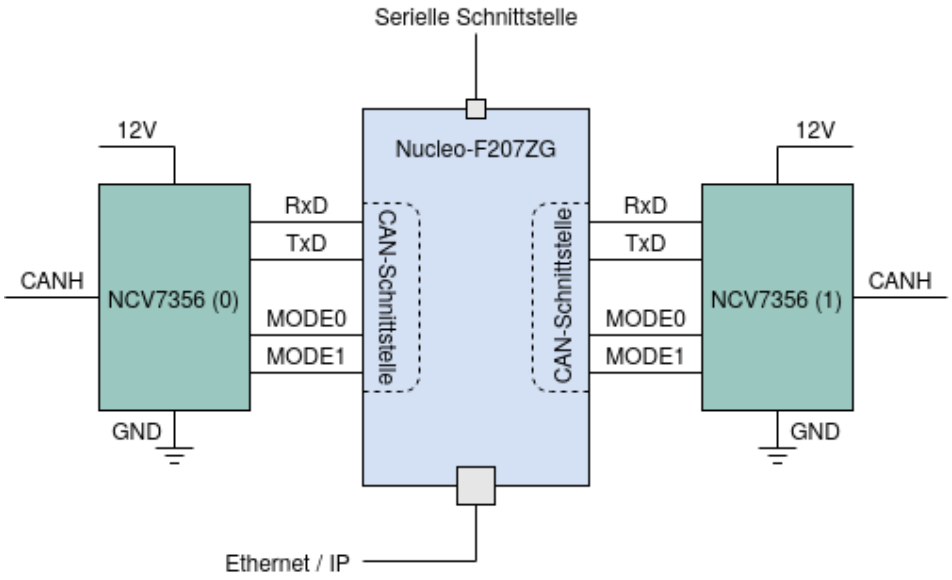

Abbildung 5.1: Schematischer Versuchsaufbau
Quelle: Eigene Darstellung

schiedene Nachrichtenfilter pro CAN-Schnittstelle möglich, sodass Nachrichten von bis zu 28 verschiedenen CAN-Identifikatoren empfangen werden können. Neben den CAN-Schnittstellen besitzt das Evaluierungsboard eine 10/100-Netzwerkschnittstelle, über welche die Ethernet- und IP-Kommunikation möglich ist. Außerdem verfügt das Evaluierungsboard über eine serielle Schnittstelle, über die es programmiert und konfiguriert werden kann.
Für die Kommunikation zwischen den CAN-Schnittstellen des Evaluierungsboards und

den anliegenden CAN-Netzwerken wurden in dieser Arbeit zwei NCV7356[1] CAN-Transceiver gewählt. Dieser CAN-Transceiver kommuniziert mit dem jeweiligen CAN-Bus über eine einzelne Leitung (CANH) mit einer maximalen Übertragungsgeschwindigkeit von 100 kbit/s, wobei die laut Datenblatt empfohlene Bitrate von 83 kbit/s in dem Versuchsaufbau genutzt wurde. Mithilfe der Pins TxD und RxD leitet der CAN-Transceiver CAN-Nachrichten des Evaluierungsboards an den CAN-Bus oder CAN-Nachrichten des CAN-Busses an das Evaluierungsboard weiter (siehe Abbildung 5.1). Außerdem besitzt der CAN-Transceiver die zwei Pins MODE0 und MODE1, mit denen der gewünschte Betriebsmodus eingestellt werden kann.

Nucleo-F207ZG	**NCV7356 (0)**	**NCV7356(1)**
PD_0	RxD	-
PD_1	TxD	-
PB_5	-	RxD
PB_6	-	TxD
PD_14	MODE0	MODE0
PD_15	MODE1	MODE1

Tabelle 5.1: Pinbelegung des Versuchsaufbaus

Damit die CAN-Transceiver korrekt funktionieren können, sind die Pins TxD, RxD, MODE0 und MODE1 direkt mit dem Evaluierungsboard verbunden (siehe Tabelle 5.1). Beide CAN-Transceiver werden in diesem Versuchsaufbau im Hochgeschwindigkeitsbetriebsmodus (TRX_HIGH_SPEED_MODE) verwendet, weswegen die beiden MODE0-Pins und MODE1-Pins in diesem Versuchsaufbau mit den gleichen Pins des Evaluierungsboards verbunden sind. Der Betriebsmodus wird erreicht, indem die Pegel der MODE0-Pins auf logisch „high“ und die MODE1-Pins auf logisch „low“ gesetzt werden. In dieser Arbeit werden beide CAN-Transceiver mit einer Spannung von 12 Volt betrieben. Darüber hinaus wurden die laut Datenblatt nötigen Widerstände in den Versuchsaufbau eingebaut. Die Netzwerkschnittstelle dieses Evaluierungsboards ist in dieser Arbeit mit einem Laptop verbunden, der das Paketmanipulationswerkzeug Scapy[2] ausführt. Dieses Werkzeug erlaubt es, Netzwerkpakete zu erstellen, zu senden und den Netzwerkverkehr überwachen. Auf der anderen Seite sind die CAN-Schnittstellen über einen CAN-Bus mit einem

[1] https://www.onsemi.com/pub/Collateral/NCV7356-D.PDF
[2] https://scapy.net/

Laptop verbunden, welcher die CAN-Entwicklungssoftware CANoe[3] ausführt. Diese bietet die Möglichkeit, CAN-Busverkehr zu erzeugen, indem individuelle CAN-Nachrichten generiert und versendet werden sowie CAN-Nachrichten von externen Steuergeräten zu empfangen. Außerdem wurde ein externes Evaluierungsboard gewählt, mit welchem die Zeiten der Ereignisse des Gateways festgehalten werden können.

[3] https://www.vector.com/de/de/produkte/produkte-a-z/software/canoe/

6 Test und Evaluierung

Die Funktionalität des Gateways wurde durch verschiedene Szenarien getestet. Zum Einen sind die Verarbeitungszeiten und zum Anderen der Paketverlust des Gateways getestet worden. Diese Tests sind mit standard und erweiterten Identifikatoren und allen zulässigen Nutzlastgrößen (0 - 8 Byte) durchgeführt worden. Darüber hinaus wurde das Gateway auf seine Interoperabilität getestet.

6.1 Zeitmessung

Als Erstes werden die Zeiten empfangener Nachrichten über dem Gateway evaluiert. Dies beinhaltet das Übersetzen in das korrekte Format sowie das abgeschlossene Versenden an die jeweilige Schnittstelle.
In Abbildung 6.1 sind die Zeiten ab dem Empfangen von CAN-Nachrichten bis hin zum erfolgreichen Weiterleiten an die geforderten Schnittstellen zu sehen. Es is zu erkennen, dass die Zeiten für das Versenden von Nachrichten über Ethernet, IPv4 und IPv6 trotz verschiedener Identifikatoren und Nuzlastgrößen konstant sind. Im Schnitt benötigt das Gateway 165 μs bis eine Nachricht über IPv6 weitergeleitet wurde. Über IPv4 dauert dies im Vergleich 57 μs und über Ethernet lediglich 45 μs. Die Unterschiede zwischen IPv4 und IPv6 lassen sich hier mit den unterschiedlichen Netzwerkstackimplementierungen von LwIP (IPv4) und GNRC (IPv6) begründen, da bei GNRC zwischen den laufenden Prozessen mehrere Nachrichten (IPC) ausgetauscht werden und der Kontext auf jeder Netzwerkschicht gewechselt wird. Dies kostet Zeit und sorgt für die Differenz in der Messung. Nachrichten über Ethernet schneiden hier am besten ab, da sie durch das Fehlen des IP- und UDP-Paketkopf kleiner ausfallen. Dadurch besteht weniger Rechenaufwand für das Gateway, sodass sie schneller übertragen werden können.

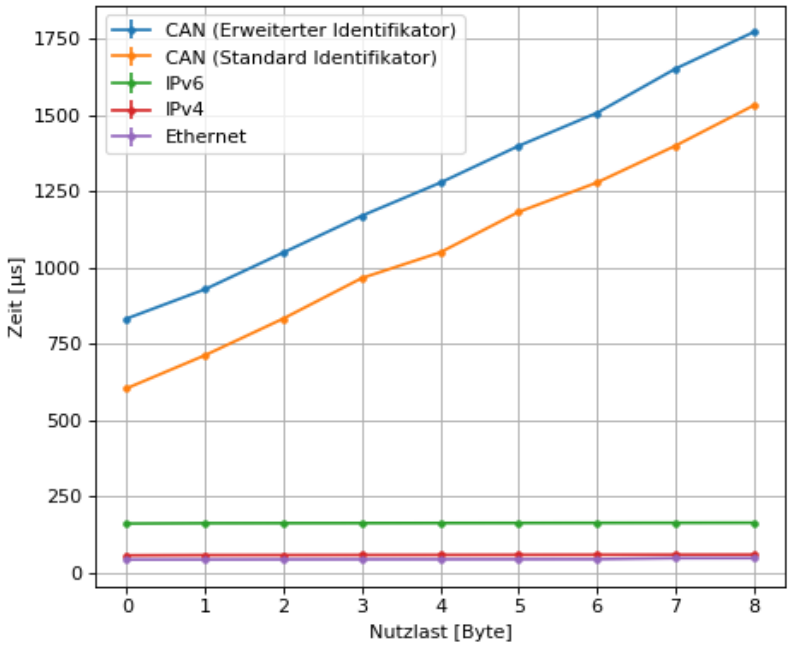

Abbildung 6.1: Verarbeitungszeiten ab dem Empfangen einer CAN-Nachricht bis zum erfolgreichen Versenden

Feldname	**CAN 2.0A**	**CAN 2.0B**
SoF	1	1
Arbitration Field	12	32
Control Field	6	6
Data Field	0 - 64	0 - 64
CRC Field	16	16
ACK Field	2	2
EoF	7	7

Tabelle 6.1: CAN-Feldlängen in Bit

Im Gegensatz dazu steigt die Zeit zum Versenden von Nachrichten über CAN, je größer die Nutzlast der Nachricht ist. Außerdem macht es hier einen Unterschied, ob die Nachricht einen Standard- oder erweiterten Identifikator besitzt. Die Erklärung dafür ist, dass sich die Größe der CAN-Nachrichten je nach Nutzlast und Identifikator unterscheiden

kann (siehe Tabelle 6.1). Es werden also bei unterschiedlichen Nutzlastgrößen verschieden große Nachrichten über das Netzwerk versendet, wobei die Nutzlast nicht größer als 8 Byte sein kann. Darüber hinaus transportieren Nachrichten mit einem erweiterten Identifikator (CAN 2.0B) 20 Bits mehr als Nachrichten mit einem Standard-Identifikator (CAN 2.0A), was für die Differenz in der Zeit zwischen den beiden Identifikatortypen sorgt. In Kombination mit der relativ geringen Übertragungsrate von 83 kbit/s benötigt das Gateway somit mehr Zeit, die Nachrichten an die CAN-Schnittstelle zu senden, als über Ethernet und IPv6 oder IPv4, da hier eine größere Bandbreite zu Verfügung steht. Außerdem muss berücksichtigt werden, dass das Prinzip des Bit-Stuffing zusätzliche Bits erzeugen kann, welche die Größe der Nachricht weiter beeinflussen. Im besten Fall bedeutet dies eine Verarbeitungszeit einer Nachricht von 0.6 ms und im ungünstigsten Fall eine Verarbeitungszeit von 1.77 ms, um diese Nachricht zu übertragen (siehe Abbildung 6.1).

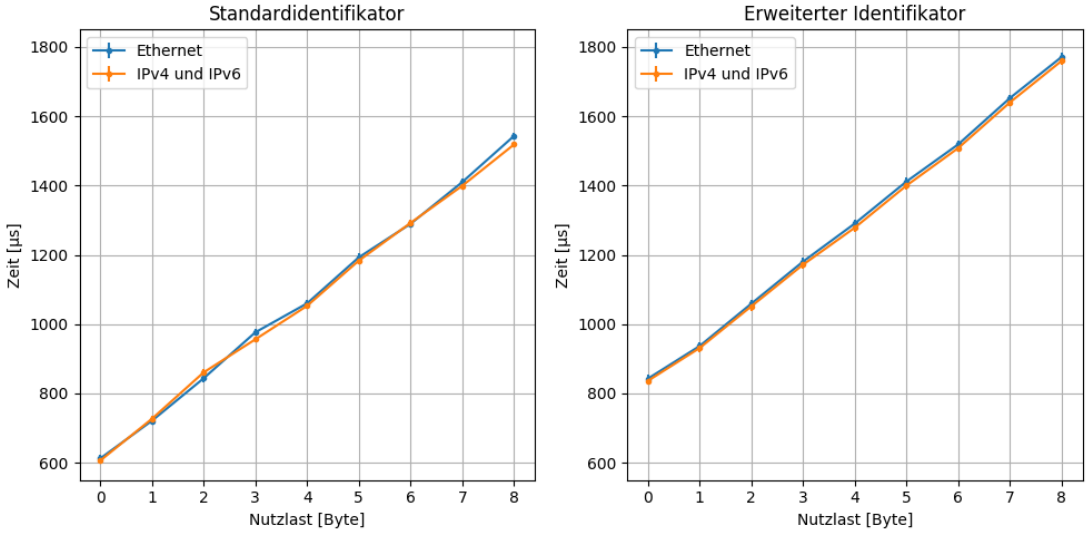

Abbildung 6.2: Verarbeitungszeiten ab dem Empfangen einer Nachricht über Ethernet, IPv4 und IPv6 bis zum erfolgreichen Versenden

Abbildung 6.2 zeigt die Verarbeitungszeiten, die das Gateway benötigt, um Nachrichten, die über Ethernet, IPv4 oder IPv6 empfangen wurden, in das CAN-Format zu übersetzen und per CAN weiterzuleiten. Hier ist zu sehen, dass Nachrichten, die über Ethernet, IPv4 oder IPv6 empfangen wurden, eine sehr ähnliche Zeit benötigen, um über CAN weitergeleitet zu werden. Diese Zeiten stimmen mit den Zeiten für das Weiterleiten von empfangenen CAN-Nachrichten über CAN aus der vorherigen Messung überein. Die oben be-

schriebene Begründung, dass die geringe CAN-Bitrate gepaart mit den unterschiedlichen CAN-Nachrichtengrößen für Zeitdifferenzen bei der Übertragung von CAN-Nachrichten sorgt, trifft auch hier zu.
Da die Fülle an Informationen bei modernen Steuergeräten steigt, ist es unwahrscheinlich, dass CAN-Nachrichten mit weniger als der maximalen Nutzlastgröße an Informationen genutzt werden. Dies bedeutet aber auch gleichzeitig, dass CAN-Nachrichten mit der maximalen Nutzlastgröße die größte Latenz besitzen, bis sie ihr Ziel erfolgreich erreichen. Zudem muss beachtet werden, dass die CAN-Identifikatoren verschiedene Prioritäten repräsentieren, sodass Nachrichten mit niedrigen Prioritäten denen mit einer hohen Priorität nachgestellt sind, was zu weiteren Zeitverzögerungen führen kann.
Die Messergebnisse zeigen, dass die Übertragung von Informationen über Ethernet, IPv4 oder IPv6 der Übertragung von Informationen über CAN überlegen sind. Selbst die maximal mögliche CAN-Bitrate von 1 Mbit/s liegt weit unter den möglichen Übertragungsraten von Ethernet. Daneben sind CAN-Nachrichten auf einem Informationsgehalt von 8 Byte beschränkt, was bei der steigenden Fülle an Informationen durch moderne Steuergeräte nicht ausreichend ist. Außerdem gibt es keine Möglichkeit, das Ziel von CAN-Nachrichten zu definieren, da der CAN-Identifikator für die Priorität und den Inhalt der Nachricht steht. Im Vergleich zur Übertragung von Informationen über Ethernet ermöglicht die Übertragung über SOME/IP des Weiteren das Senden von Daten auf Anfrage der jeweiligen Abonnenten durch das Umsetzen des Publish/Subscribe-Konzepts, was das Netzwerk zusätzlich entlastet.

6.2 Verarbeitungszeit

In diesem Abschnitt werden die reinen Verarbeitungszeiten innerhalb des Gateways evaluiert. Dies beinhaltet das Übersetzen empfangener Nachrichten sowie das Setzen der zugehörigen Weiterleitungsinformationen.
Die Abbildung 6.3 zeigt die Verarbeitungszeiten für empfangene CAN-Nachrichten, die benötigt werden, um diese in entsprechende Formate (Ethernet, IPv4 oder IPv6) zu übersetzen, damit sie über die anliegenden Netzwerke versendet werden können. In allen Fällen steigen die Kurven parallel zueinander mit zunehmender Nutzlast an. Dies lässt sich damit begründen, dass mit wachsender Nutzlast auch mehr Byte an Daten in die jeweiligen Formate übersetzt werden müssen. Dabei ist zu beachten, dass in diesen Tests die Einträge, die zum Übersetzen und Weiterleiten benötigt werden, an erster Stelle der Identifikatorliste standen. Es kann minimale Unterschiede geben, sollten sich

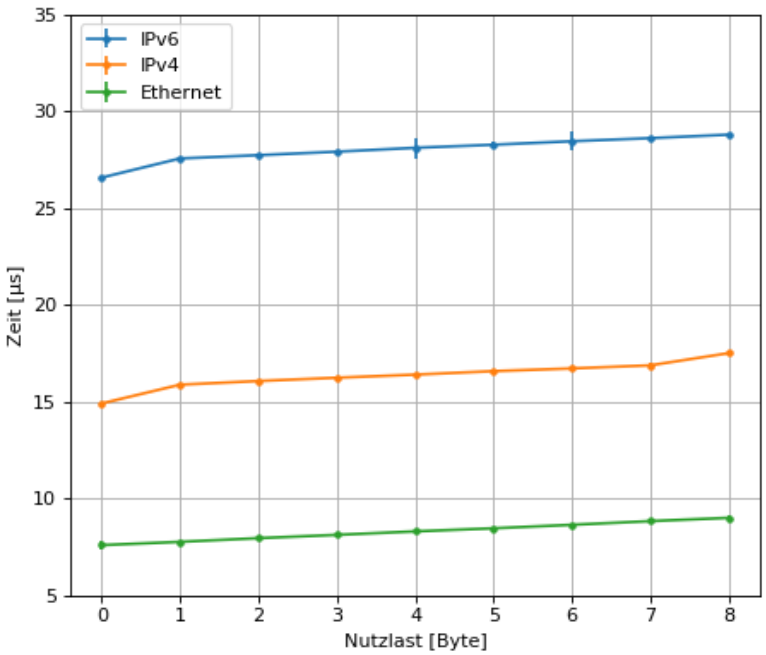

Abbildung 6.3: Zeit ab dem Empfangen einer CAN-Nachricht bis zum fertigen Übersetzen in das Ethernet-, IPv4- oder IPv6-Format

die Einträge an anderen Positionen der Identifikatorliste befinden. Auffällig ist, dass die Verarbeitungszeiten von dem CAN- ins Ethernet-, IPv4- oder IPv6- variieren, wobei das Verarbeiten nach Ethernet am besten abschneidet. Da Nachrichten im Ethernetformat keine IP- und UDP-Paketköpfe besitzen, benötigt das Gateway weniger Rechenaufwand für das Übersetzen dieser Nachrichten. Bei dem Verarbeiten von CAN-Nachrichten nach IPv4 und IPv6 sind diese Paketköpfe enthalten und zusätzlich wird die Nachricht in das SOME/IP-Format übersetzt, was in der Umsetzung mehr Zeit für das Bauen der Nachrichten benötigt. Hier ist zu beobachten, dass die Verarbeitung nach IPv4 weniger Zeit benötigt, als die Verarbeitung nach IPv6. Dieser Unterschied lässt sich hier mit den unterschiedlichen Netzwerkstackimplementierungen von LwIP (IPv4) und GNRC (IPv6) begründen, da GNRC Interprozesskommunikation zwischen den laufenden Prozessen betreibt und den Kontext auf jeder Netzwerkschicht wechselt. Daneben sind IPv6-Adressen (16 Byte) länger als IPv4-Adressen (4 Byte), wodurch das Übergeben der Ziel-IP-Adresse an die Schnittstelle *sock* im Falle von IPv6 länger dauert als bei IPv4. Ob empfangene CAN-Nachrichten einen Standard- oder erweiterten Identifikator besitzen hat in diesen

Messungen keinen Unterschied ergeben.

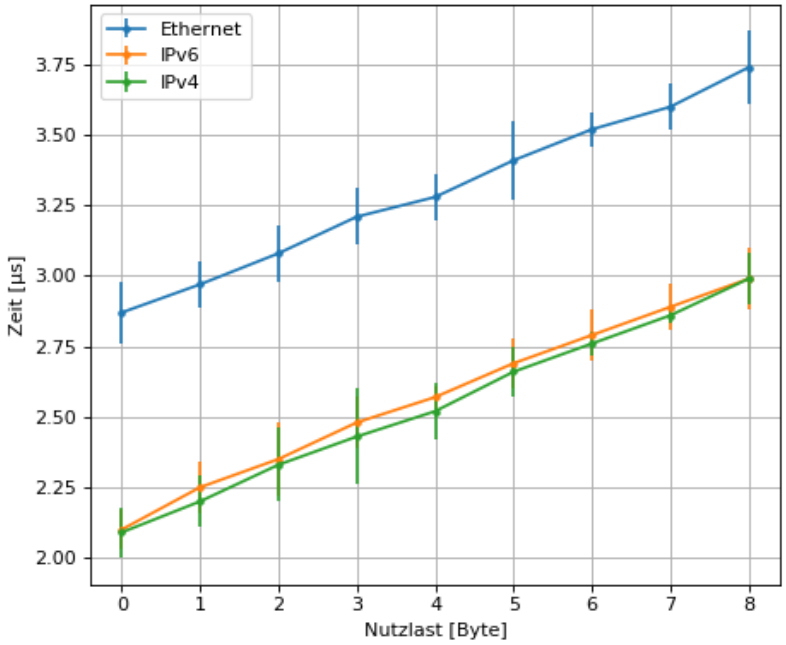

Abbildung 6.4: Zeit ab dem Empfangen einer Nachricht über Ethernet, IPv4 oder IPv6 bis zum fertigen Übersetzen in das CAN-Format

In der Abbildung 6.4 sind die Verarbeitungszeiten ab dem Empfangen von Nachrichten über Ethernet, IPv4 oder IPv6 zu sehen, die das Gateway benötigt, um diese Nachrichten in das CAN-Format zu übersetzen. Für alle empfangenen Nachrichten gilt: Je größer die Nutzlast ist, desto länger dauert das Übersetzen in das CAN-Format. Dabei ist die Dauer für das Verarbeiten von Nachrichten mit einem Standard- und erweiterten Identifikator gleich, daher sind sie in einem Diagramm dargestellt. Ethernet benötigt in diesen Messungen mehr Zeit als IPv4 oder IPv6, weil in diesem Fall die nötigen Informationen aus den einzelnen *Snips* der empfangenen Ethernet-Nachricht gewonnen werden müssen, wohingehen die SOME/IP-Nachrichten in den IPv4 und IPv6 Nutzlasten durch einfache Datentypumwandlung erhalten werden. Daneben wird zuerst die gesamte sechs Byte lange Zieladresse aus den empfangenen Ethernet-Nachrichten kopiert, woraufhin der vier Byte lange CAN-Identifikator gewonnen wird. Es werden also zwei zusätzliche Bytes aus der Zieladresse heraus kopiert, was mehr Zeit in der Umsetzung kostet. Die Zeit kann sich hier auch minimal je nach Position der Identifikatorlisteneinträge unterscheiden.

6.3 Paketverlust

Dieser Absatz behandelt die gemessene Paketverlustrate des Gateways. Zu diesem Zwecke wurden dem Gateway bis zu 10.000 Nachrichten in verschiedenen Sendefrequenzen gesendet.

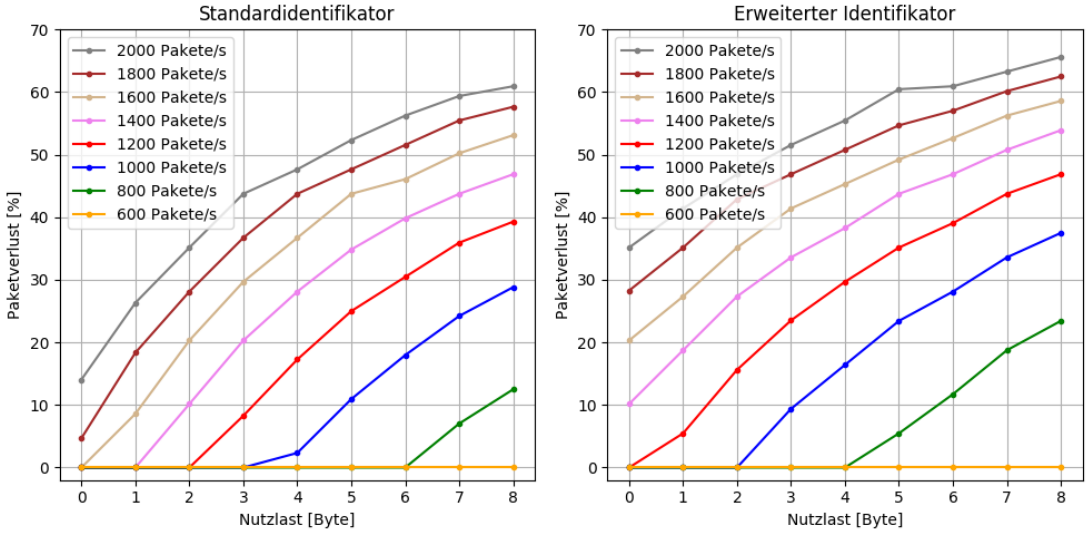

Abbildung 6.5: Paketverlust über Ethernet

Zuerst wird der Paketverlust für Nachrichten ausgewert, die über Ethernet empfangen werden. Die Abbildung 6.5 zeigt, dass die Paketsendefrequenz, mit welcher die Pakete pro Sekunde an das Gateway gesendet werden, einen großen Einfluss auf die Verlustrate hat. Außerdem hängt die Verlustrate von der Nutzlastgröße und dem enthaltenen CAN-Identifikator der jeweiligen Nachrichten ab. Es ist zu erkennen, dass bei einer Paketsendefrequenz von 600 Paketen je Sekunde kein Paketverlust zu verzeichnen ist. Wie die Tabelle 6.1 zeigt, kann eine CAN-Nachricht bis zu 128 Bit groß sein. Bei einer CAN-Bitrate von 83 kbit/s bedeutet dies, dass das Gateway ca. 600 Pakete pro Sekunde entgegen nehmen und die darin enthaltenen CAN-Nachrichten versenden kann, ohne einen Paketverlust zu verzeichnen. Sobald die Paketsendefrequenz 600 Pakete die Sekunde überschreitet, finden die ersten Paketverluste statt, da nun mehr Pakete das Gateway erreichen, als es CAN-Nachrichten versenden kann. Alle überschüssigen Pakete werden daher sofort verworfen. Paketverlust tritt also dann auf, wenn die genutzte CAN-Bitrate ausgeschöpft ist, was bei CAN-Nachrichten mit einer größeren Nutzlast oder erweiterten Identifikator schneller passiert, als bei Nachrichten mit kleiner Nutzlast und Standard-

Identifikator. Je höher die Paketsendefrequenz ist, desto größer wird der Paketverlust.

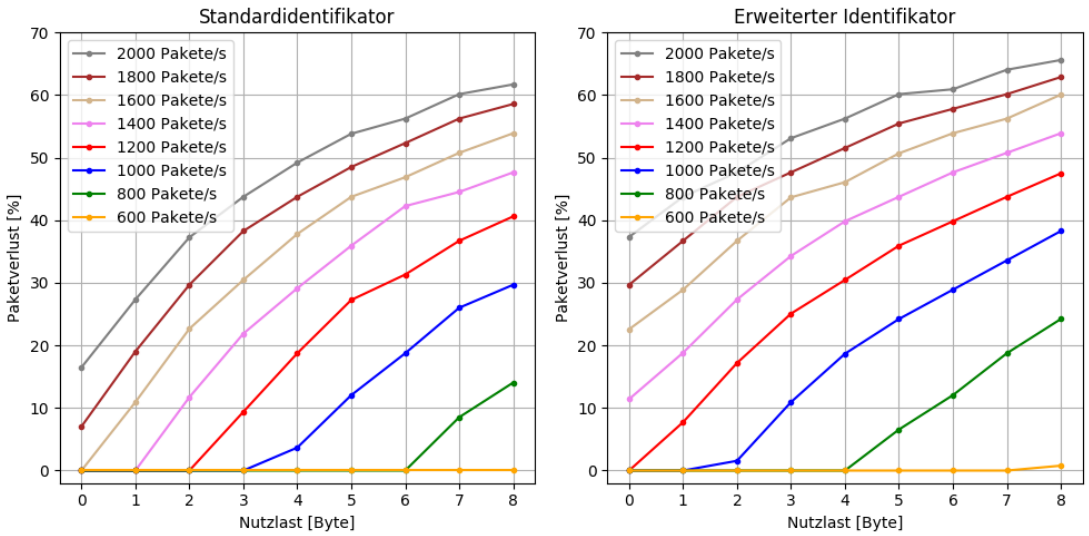

Abbildung 6.6: Paketverlust über IPv4 oder IPv6

In Abbildung 6.6 ist der Paketverlust des Gateways für empfangene Nachrichten über IPv4 und IPv6 zu sehen. Der Paketverlust ist für die beiden Internetprotokolle identisch, daher werden sie in einem Diagramm dargestellt. Hier ist das gleiche Ergebnis wie bei dem Empfangen von Nachrichten über Ethernet zu beobachten. Wenn das Gateway mehr als ca. 600 Pakete pro Sekunde empfängt, kann es nicht alle rechtzeitig verarbeiten und verwirft daher die überschüssigen Pakete, was den Paketverlust erklärt. Verantwortlich dafür ist auch hier die niedrige CAN-Bitrate. Die Größe der CAN-Nachrichten spielt hier ebenfalls eine Rolle, sodass der Paketverlust bei Nachrichten mit großer Nutzlast und einem erweiterten Identifikator größer ausfällt als bei solchen mit geringer Nutzlast und Standard-Identifikator.
Auf der anderen Seite wurde bei dem Empfangen von CAN-Nachrichten und dem Weiterleiten dieser über Ethernet, IPv4 oder IPv6 kein Paketverlust gemessen. Dies lässt sich mit der Übertragungsrate des Evaluierungsboards erklären, weil diese mit 10 beziehungsweise 100 Mbits/s weit über der CAN-Bitrate von 83 kbit/s liegt. So ist keine Situation entstanden, in der mehr CAN-Nachrichten empfangen wurden als das Gateway verarbeiten kann. Mit diesem Hintergrund sollte selbst die maximal mögliche CAN-Bitrate von 1 Mbit/s keinen Paketverlust verursachen, wobei zu beachten ist, dass der Versuchsaufbau kein gesamtes Fahrzeugnetzwerk repräseniert und das Vorkommen von Paketverlust daher nicht ausgeschlossen werden kann.

Um den Paket- und somit den Informationsverlust in den gezeigten Bereichen so gering wie möglich zu halten, gibt es zwei Ansätze. Zum Einem wäre es sinnvoll, die Paketsendefrequenz der CAN-Bitrate anzupassen, um so für jede Nutzlastgröße den Paketverlust zu vermeiden. Zum Anderen könnte ein leistungsstärkerer CAN-Transceiver herangezogen werden, der größere CAN-Bitraten verarbeiten kann, sodass das Gateway mehr Nachrichten pro Sekunde empfangen könnte.
Das Gateway ist momentan so konstruiert, dass jede Nachricht, die über Ethernet, IPv4 oder IPv6 versendet wird, maximal eine CAN-Nachricht enthält. Wenn man das Beispiel einer Ethernet-Nachricht betrachtet, muss diese nun künstlich auf die Mindestgröße von 64 Byte aufgefüllt werden. Ein Ansatz der Verbesserung des Gateways wäre einen Puffer zu implementieren, der ankommende CAN-Nachrichten abspeichert, wodurch das Gateway mehrere CAN-Nachrichten auf einmal über Ethernet, IPv4 oder IPv6 übertragen könnte, um eine größere Bandbreite von Ethernet nutzen zu können. Um die dabei anfallende Zeitverzögerung so gering wie möglich zu halten, könnte ein Timer implementiert werden, nach dessen Ablauf der Inhalt des Puffers unabhängig vom Füllstand versendet wird. Daneben ist das Gateway durch eine begrenzte Anzahl an CAN-Nachrichtenfiltern in der Hinsicht eingeschränkt, dass nicht jede empfangene CAN-Nachricht weitergeleitet werden kann, sondern nur vorher definierte. Diese Einschränkungen gibt es jedoch nicht bei Nachrichten die über Ethernet- oder IP-basierten Netzwerke empfangen werden.

6.4 Interoperabilitätstest

In diesem Abschnitt werden die Interoperabilitätstests betrachtet, welche das Gateway bestanden hat. Die einzelnen Interoperabilitätstests sind in Tabelle 6.2 aufgeführt. Mit diesen Tests wurde gezeigt, dass das Gateway in der Lage ist, mit anderen Systemen zu interagieren.
Um die Ethernet- und IP-Kommunikation zwischen dem Gateway und anderen Systemen zu testen, wurde das Paketmanipulationswerkzeug Scapy genutzt. Das Gateway war hier in der Lage, von Scapy generierte Ethernet-Nachrichten sowie Pakete über IPv4 und IPv6, welche das SOME/IP-Format besaßen, korrekt zu empfangen. Die empfangenen Nachrichten wurden anhand ihrer Identifikatoren erfolgreich in das CAN-Format übersetzt und anschließend über die jeweilige CAN-Schnittstelle zum Ziel weitergeleitet. Andersherum wurden Nachrichten des Gateways über Ethernet, IPv4 oder IPv6 korrekt von Scapy im Netzwerk erkannt und verstanden. Nachrichten, die in diesem Falle per IP versendet wurden, haben ebenfalls das SOME/IP-Format umgesetzt.

Szenario	Erwartung	Erfüllt
Von Scapy generierte Ethernet-Nachrichten werden an das Gateway gesendet	Nachrichten werden über Ethernet empfangen und anhand des CAN-Identifikators korrekt weitergeleitet. Standard- und erweitertes Frameformat sind möglich	✓
Von Scapy generierte Nachrichten im SOME/IP-Format werden über UDP an das Gateway gesendet	Nachrichten werden über *sock* (IPv4 oder IPv6) empfangen und anhand des CAN-Identifikators korrekt weitergeleitet. Standard- und erweitertes Frameformat sind möglich	✓
Scapy empfängt Nachrichten des Gateways über Ethernet	Nachrichten werden über Ethernet empfangen und mit dem Inhalt korrekt dargestellt	✓
Scapy empfängt Nachrichten des Gateways im SOME/IP-Format über UDP	Nachrichten werden über IPv4 oder IPv6 empfangen und mit dem Inhalt korrekt dargestellt	✓
Von CANoe generierte CAN Nachrichten werden an das Gateway gesendet	Nachrichten werden von dem Gateway empfangen und anhand der ID per Eth/IP/CAN weitergeleitet. Standard- und erweitertes Frameformat sind möglich	✓
CANoe empfängt CAN Nachrichten des Gateways	CANoe empfängt von dem Gateway gesendeten CAN Nachrichten. Standard- und erweitertes Frameformat sind möglich	✓

Tabelle 6.2: Interoperabilitätstests

Um die CAN-Seite zu testen, wurde die CAN-Entwicklungssoftware CANoe genutzt. Von CANoe generierte CAN-Nachrichten wurden hier von dem Gateway erfolgreich empfangen und anhand ihrer Identifikatoren korrekt in die nötigen Formate übersetzt und an die Zielschnittstellen gesendet. Auf der anderen Seite konnte CANoe die von dem Gateway gesendeten CAN-Nachrichten erfolgreich empfangen und darstellen.

7 Fazit und Ausblick

Diese Arbeit zeigt die Umsetzung eines Gateways, welches ein erster Entwicklungsschritt hin zu einer leistungsstärkeren Kommunikationsverbindung mit hoher Bandbreite in Automobilen darstellt. Für die Umsetzung des Gateways wurde das Betriebssystem RIOT, mit dessen CAN-Implementierung und dem Netzwerkstack GNRC gewählt. Neben GNRC wurde zusätzlich der Netzwerkstack LwIP genutzt. Das Gateway fungiert als Schnittstelle zwischen CAN- und Ethernet- oder IP-basierten Netzwerken, indem ankommende Nachrichten anhand ihres CAN-Identifikators weitergeleitet werden. Dazu kann das Gateway je nach Anwendungsfall in verschiedenen Betriebsmodi genutzt werden, welche das Austauschen von Nachrichten zwischen CAN und Ethernet, IPv4 oder IPv6 erlauben. Dabei werden die benötigten Weiterleitungsinformationen dem Gateway während der Laufzeit übergeben und in einer Liste verwaltet. Die in den Listeneinträgen enthaltenen Informationen geben Auskunft darüber, wie und ob empfangene Nachrichten der anliegenden Netzwerke übersetzt und an welches Ziel sie gesendet werden sollen. Zusätzlich nutzt das Gateway für die Kommunikation mit IP-basierten Netzwerken das SOME/IP-Format, welches ein neues Kommunikationsprotokoll aus dem Bereich Automotive Ethernet darstellt und das Publish/Subscribe-Konzept im Automobil ermöglicht.
In dieser Arbeit wird dargestellt, dass ein solches Gateway mit den limitierten Ressourcen eines Steuergerätes im Automobil umgesetzt werden kann, wobei Einschränkungen, wie beispielsweise eine begrenzte Anzahl an CAN-Nachrichtenfiltern, zu beachten sind. Zusätzlich stellt die Auswertung dieser Arbeit dar, dass der Transport von Informationen über CAN aufgrund der geringen Nutzlasten und Bitraten dem über Ethernet und IP unterliegt.
Mithilfe dieses Gateways kann die Umsetzung einer Domänen-Gateway-Topologie in Betracht gezogen werden, in welcher die Kommunikation zwischen den Anwenungsdomämen über Domänen-Gateways stattfindet. So könnte der Engpass des zentralen Gateways innerhalb des Automobils aufgehoben und gleichzeitig ein erster Ethernet-Backbone etabliert werden. Dies würde ermöglichen neue hochauflösende Sensoren direkt in das Fahr-

zeugnetzwerk einzufügen, sodass klassische und zukünftige Steuergeräte in einer Architektur kombiniert werden könnten.

Literaturverzeichnis

[1] T. Steinbach, *Ethernet-basierte Fahrzeugnetzwerkarchitekturen für zukünftige Echtzeitsysteme im Automobil.* Wiesbaden: Springer Vieweg, October 2018. [Online]. Available: http://dx.doi.org/10.1007/978-3-658-23500-0

[2] T. Steinbach, H.-T. Lim, F. Korf, T. C. Schmidt, D. Herrscher, and A. Wolisz, "Beware of the Hidden! How Cross-traffic Affects Quality Assurances of Competing Real-time Ethernet Standards for In-Car Communication," in *2015 IEEE Conference on Local Computer Networks (LCN).* Piscataway, NJ, USA: IEEE Press, October 2015, pp. 1–9, lCN Best Paper Award.

[3] T.-C. Nichiţelea and M.-G. Unguritu, "Automotive Ethernet Applications Using Scalable Service-Oriented Middleware over IP: Service Discovery," in *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR).* IEEE, 2019, pp. 576–581.

[4] T. Steinbach, F. Korf, and T. C. Schmidt, "Real-time Ethernet for Automotive Applications: A Solution for Future In-Car Networks," in *1st IEEE International Conference on Consumer Electronics - Berlin (ICCE-Berlin 2011).* Piscataway, NJ, USA: IEEE Press, Sep. 2011, pp. 216–220. [Online]. Available: http://dx.doi.org/10.1109/ICCE-Berlin.2011.6031843

[5] G. L. Gopu, K. V. Kavitha, and J. Joy, "Service Oriented Architecture based connectivity of automotive ECUs," in *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT).* IEEE, 2016, pp. 1–4.

[6] S. Fürst and M. Bechter, "AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W).* IEEE, 2016, pp. 215–217.

[7] AUTOSAR, "SOME/IP Protocol Specification," AUTOSAR, Tech. Rep. R19-11, 2019.

[8] L. Völker, "Scalable service-Oriented MiddlewarE over IP (SOME/IP)," http://some-ip.com/index.shtml, 2020, Zugriffsdatum: 20.07.2020.

[9] GENIVI vSOMEIP, "vsomeip in 10 minutes," https://github.com/GENIVI/vsomeip/wiki/vsomeip-in-10-minutes, 2018, Zugriffsdatum: 20.07.2020.

[10] Vector Informatik GmbH, "SOME/IP," https://elearning.vector.com/mod/page/view.php?id=32, 2018, Zugriffsdatum: 20.07.2020.

[11] ATMES GmbH, "SOME/IP und automobiles Ethernet," https://www.atmes.de/presse/fachartikel/someip-und-automobiles-ethernet.html, 2020, Zugriffsdatum: 20.07.2020.

[12] DATACOM Buchverlag GmbH, "SOME/IP (scalable service-oriented middleware over IP)," https://www.itwissen.info/SOME-IP-scalable-service-oriented-middleware-over-IP.html, March 2020, Zugriffsdatum: 20.07.2020.

[13] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks," IETF, RFC 7228, May 2014.

[14] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2018.2815038

[15] P. Kietzmann, T. C. Schmidt, and M. Wählisch, "RIOT - das freundliche Echtzeitbetriebssystem für das IoT," in *Internet der Dinge*. Berlin: Springer Vieweg, Nov. 2016, pp. 43–52.

[16] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündogan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," Open Archive: arXiv.org, Technical Report arXiv:1801.02833, January 2018. [Online]. Available: https://arxiv.org/abs/1801.02833

[17] Robert Bosch GmbH, "CAN Specification Version 2.0," Bosch, Tech. Rep. ISO 11898, 1991.

[18] W. Lawrenz and N. Obermöller, *CAN: Controller Area Network: Grundlagen, Design, Anwendungen, Testtechnik*, 5th ed. VDE Verlag, 2011.

[19] H. Chen and J. Tian, "Research on the Controller Area Network," in *2009 International Conference on Networking and Digital Society*, vol. 2. IEEE, 2009, pp. 251–254.

[20] H. F. Othman, Y. R. Aji, F. T. Fakhreddin, and A. R. Al-Ali, "Controller area networks: Evolution and applications," in *2006 2nd International Conference on Information Communication Technologies*, vol. 2. IEEE, 2006, pp. 3088–3093.

[21] "IEEE Standard for Ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, p. 121, Aug 2018.

A Anhang

Der Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Eine Gateway-Funktion zwischen CAN und Ethernet oder SOME/IP für RIOT

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

_____	_____	_____
Ort	Datum	Unterschrift im Original