HAW HAMBURG

Bachelorarbeit

Mehmet Cakir

Evaluation dienstorientierter Kommunikation in automobilen Zonalarchitekturen

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Mehmet Cakir

Evaluation dienstorientierter Kommunikation in automobilen Zonalarchitekturen

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. Franz Korf
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 17.01.2020

Mehmet Cakir

Thema der Arbeit
Evaluation dienstorientierter Kommunikation in automobilen Zonalarchitekturen

Stichworte
Kommunikationsarchitektur, dienstorientiert, Zonenarchitektur, SOA, Middleware

Kurzzusammenfassung
In Automobilnetzwerken steigt mit zunehmender Anzahl von Netzwerkkomponenten der Bandbreitenbedarf, wodurch Ethernet Bussysteme verdrängt. Dienstorientierte Architekturen verringern die Komplexität und können mit Dienstgüteverhandlungen heterogene Anforderungen erfüllen. Diese Arbeit evaluiert mithilfe einer praxisnahen Simulationsumgebung eine dienstorientierte Middleware mit dynamischer Dienstgüteverhandlung. Die Middleware und eine Netzwerkbeschreibungssprache werden erweitert, sowie das Zeitverhalten der Middleware untersucht. Zeitliche Anforderungen in heterogenen Autonetzwerken werden eingehalten und die Setup-Time liegt deutlich unter den Anforderungen.

Mehmet Cakir

Title of Thesis
Evaluation Of Service Oriented Communication In Automobile Zone Architectures

Keywords
Communication Architecture, Service Oriented, Zone Architecture, SOA, Middleware

Abstract

In automotive networks the bandwidth requirements rise because of the increasing count of network components. Therefore ethernet replaces bus systems. Service-Oriented Architectures reduce the complexity. With Quality-of-Service (QoS) negotiations they are able to fulfill heterogeneous requirements. This work evaluates a service-oriented middleware with a dynamic QoS negotiation procedure using a realistic simulation environment. The middleware and a network description language will be extended. The timings of the middleware will be analysed. The middleware fulfills requirements in heterogenous network communication. The setup time also complies clearly with the automobile requirements.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

ANDL Abstract Network Description Language.

ARP Adress Resolution Protocol.

AVB Audio Video Bridging.

BAG Bandwidth Allocation Gap.

BE Best-Effort.

CAN Controller Area Network.

CBS Credit Based Shaper.

DDS Data Distribution Service.

ECU Electronic Control Unit.

ECUs Electronic Control Units.

ESP Elektronische Stabilitätsprogramm.

HTTP Hypertext Transfer Protocol.

ICT Information and Communication Technology.

IPS IP-based Services.

LSM Local Service Manager.

LSR Local Service Registry.

MAC-Adresse Media-Access-Control-Adresse.

NED Network Description Language.

OMG Object Management Group.

OSI Open Systems Interconnection model.

QoS Quality-of-Service.

QoSM Quality-of-Service Manager.

QoSNP QoS Negotiation Protocol.

ReST Representational state transfer.

RTS Real-Time Services.

SAE Society of Automotive Engineers.

SOA Service-Oriented Architecture.

SOAP Simple Object Access Protocol.

SOME/IP Scalable service-Oriented MiddlewarE over IP.

SOQoSMW Service Oriented Quality-of-Service MiddleWare.

SR Stream Reservation.

SRP Stream Reservation Protocol.

SRTS Static-Real-Time Services.

TCP Transmission Control Protocol.

TDMA Time Division Multiple Access.

TSN Time-Sensitive Networking.

TTE Time-Triggered Ethernet.

UDP User Datagram Protocol.

WS Web-based Services.

1 Einleitung

Automobile werden mit immer intelligenteren Komponenten ausgestattet, welche den Zustand des Fahrzeugs überwachen oder zur Unterhaltung dienen. Alle Komponenten bilden in Kombination mit entsprechender Software zusammen ein Netzwerk. Bei steigender Komplexität des Netzwerks und der Anzahl an Hardware, steigt auch die Komplexität in der Software. Hardware umfasst konkret für die Information and Communication Technology (ICT) Komponenten wie Sensoren, Aktoren und Steuergeräte, sogenannte Electronic Control Units (ECUs). Was dies für die Softwarekomplexität bedeutet und wieviel die Software vom Gesamtwert des Automobils ausmacht, zeigt die folgende Abbildung 1.1.

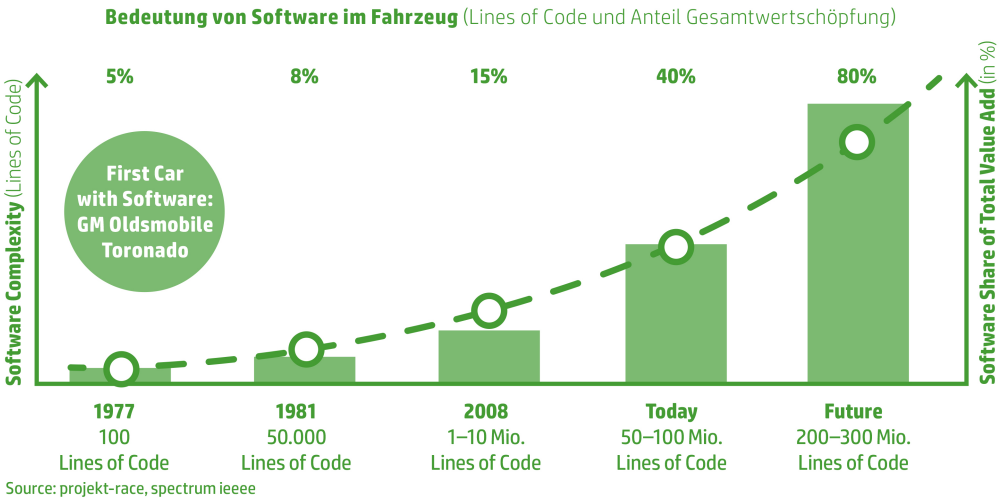

Abbildung 1.1: Bedeutung von Software im Fahrzeug bezüglich des Codeaufwands. (Quelle: [22])

Auch Broy [6] benennt die Wichtigkeit von hochwertigen Softwarekomponenten für die Automobilbranche. Für das Auto bedeuten mehr ECUs höhere Anforderungen an die Kommunikationsarchitektur hinsichtlich ihrer Platzierung und Anbindung im Netzwerk. Daneben steigt mit komplexer Vernetzung auch der Bedarf an Bandbreite und die Anforderung an das Netz. Die derzeitig gängigen ICT Architekturen moderner Fahrzeuge

laufen mit steigender Komplexität Gefahr immer mehr an komfortabler Erweiterbarkeit und Wartbarkeit abzunehmen, wodurch die Weiterentwicklung moderner Fahrzeugnetzwerke laut Buckl u.a. [7] behindert wird. Weiter ist laut einer Studie der Fortiss GmbH [12] eine neue ICT Architektur notwendig, um aktuellen Problemen zu begegnen. Mit Scalable service-Oriented MiddlewarE over IP (SOME/IP) hat AUTOSAR [5] bereits eine dienstorientierte Middleware für den Automobilbereich entwickelt, um Übersichtlichkeit und Erweiterbarkeit trotz steigender Anzahl von Komponenten zu gewährleisten. Jedoch fehlt in SOME/IP die Unterstützung für Dienstgüte-Kommunikation. Desweiteren wird SOME/IP statisch konfiguriert. Das bedeutet, dass bei jeder Änderung im Autonetzwerk eine neue Offlinekonfiguration durchgeführt werden muss. Mit anderen Worten werden Änderungen nicht dynamisch zur Laufzeit übernommen, was hohen Aufwand bei sich ändernden Kommunikationsbeziehungen mit sich bringt. Häckel [13] hat im Rahmen einer Masterarbeit ein Konzept entwickelt. Es vereint die Konzepte von dienstorienter Architektur und Dienstgüte-Kommunikation für sich dynamisch ändernde Kommunikationsbeziehungen. Es umfasst eine dienstorientierte Middleware, welche Diensten erlaubt ihre Dienstgüte dynamisch zur Laufzeit auszuhandeln. Daneben bietet die Middleware einen variablen Protokollstack an, um heterogene Anforderungen an die Netzwerkkommunikation zu erfüllen. In dieser Arbeit wird die Middleware in einer Simulationsumgebung evaluiert. Die Middleware wird zunächst in kleineren Netzwerken darauf geprüft, wie viel Zeit das Aushandeln von Diensten in Anspruch nimmt. Dadurch wird überprüft, ob die Bereitschaft der Dienste schnell genug erfolgt. Außerdem wird in einem realistischen Autonetzwerk eines modernen Autos, welche auf einer realen Kommunikationsmatrix basiert, ebenfalls die Dauer der Bereitschaft von Diensten ausgewertet. Dabei werden Dienstgüteklassen mit verschiedenen Zeitanforderungen eingesetzt. Die Arbeit ist wie folgt gegliedert: In Kapitel 2 werden Grundlagen und Begrifflichkeiten zum Verständnis dieser Arbeit vermittelt. Anschließend wird in Kapitel 3 näher auf das Autonetzwerk und die Middleware eingegangen. Entwicklungsdetails zu einem zusätzlichen Transportprotokoll und einer Netzwerkbeschreibungssprache werden in Kapitel 4 beschrieben. In Kapitel 5 werden die zur Qualitätssicherung unternommenen Schritte gezeigt. Das Hauptaugenmerk dieser Arbeit ist die Evaluation, welche in Kapitel 6 beschrieben wird. Abschließend fasst Kapitel 7 die Resultate der Evaluation zusammen und gibt einen Ausblick für den Einsatz der Middleware in einem Fahrzeugnetzwerk in einer Zonalarchitektur. Im Rahmen dieser Bachelorarbeit wurde eine wissenschaftliche Veröffentlichung von Cakir u.a. [8] verfasst und auf der 2019 IEEE Vehicular Networking Conference (VNC) [14] vorgestellt.

2 Grundlagen

In diesem Kapitel werden verwendete Technologien genannt, auf welche die Middleware basiert. Zunächst werden die Kernprinzipien einer Service-Oriented Architecture (SOA) genannt. Anschließend wird beschrieben, wie ein Controller Area Network (CAN) funktioniert. Darauf folgen die für die Middleware relevanten Echtzeiterweiterungen für Ethernet. Abschließend wird die eingesetzte Simulationsumgebung in Verbindung der verwendeten Erweiterungen beschrieben.

2.1 Service-Oriented Architecture

Die folgende Erklärung zu einer Service-Oriented Architecture (SOA) stützt sich auf das Memento *Service Orientierte Architekturen Übersicht und Einordnung* von Rausch [26]. Bei einer SOA wird nicht von einer einheitlichen Architektur gesprochen. Vielmehr ist hierbei ein Architekturmuster gemeint, in dem Softwarekomponenten in Dienste gekapselt werden und zu weiteren Diensten zusammengefasst werden können. Das Ziel ist einen Dienst in sich abgeschlossen zu konzipieren und eigenständig nutzen zu können, wodurch Abhängigkeiten minimiert werden. Weiter können Dienste auch auf unterschiedlichen Systemen betrieben werden und sind bspw. nicht an Programmiersprachen oder Betriebssysteme gebunden. Oft werden plattformübergreifende Middleware-Lösungen für die Kompatibilität unterschiedlicher Systeme untereinander eingesetzt. Für den Informationsaustausch werden Standards zur Kodierung von Nachrichten und Daten festgelegt, wie z.B. XML. Dienste, welche in sich abgeschlossen sind und auf unterschiedlichen Systemen betrieben werden können, bieten eine hohe Wiederverwendbarkeit, wodurch weitere Programmierung entfällt. Damit Dienste genutzt werden können, müssen diese auffindbar sein. Häufig dient hier ein Namensdienst, mit dem registrierte bzw. im Netzwerk bekannt gemachte Dienste gesucht und gefunden werden können. Dienste registrieren sich also bei einem Namensdienst und machen sich somit im Netzwerk bekannt. Ein Konsument

fordert einen Dienst beim Namensdienst an, woraufhin der Namensdienst ggf. den geforderten Dienst anbietet oder die Forderung zurückweist, falls kein passender Dienst gefunden wurde.

2.2 CAN-Bus (Controller Area Network)

Die folgende Erklärung zu Controller Area Network (CAN) stützt sich auf die Doktorarbeit von Steinbach [25]. Die Firma Bosch hat ab Mitte der achtiziger Jahre den CAN-Bus entwickelt und ist im Jahr 1991 das erste Bussystem, das in einem Serienfahrzeug eingesetzt wurde [27]. Ein CAN-Bus ist ein serielles Bussystem. Angeschlossene Teilnehmer an einem CAN-Bus werden im Allgemeinen auch ECU genannt. Standard-CAN ist ein Multi-Master-System, wo jeder Teilnehmer gleichberechtigt ist und keiner die Kommunikation steuert. Teilnehmer adressieren keine anderen Teilnehmer. Sie adressieren stattdessen die Nachricht mithilfe einer eindeutigen Nachrichten-ID, welche statisch vor der Laufzeit festgelegt werden muss. Anhand dieser entscheiden andere Teilnehmer, ob diese Nachricht für sie bestimmt ist. Mithilfe der Nachrichten-ID wird eine Nachricht priorisiert. Eine Nachricht mit einer niedrigeren Nachrichten-ID wird gegenüber einer Nachricht mit einer höheren bevorzugt. Gleichzeitig werden so Kollisionen von gleichzeitig gesendeten Nachrichten aufgelöst. Es darf in solch einem Fall stets der Sender zuerst Senden, dessen Nachricht eine niedrigere Nachrichten-ID aufweist. CAN arbeitet als Low-Speed-CAN mit Geschwindigkeiten von 5 kbit/s bis 125 kbit/s. High-Speed-CAN erreicht Geschwindigkeiten von 125 kbit/s bis 1 Mbit/s.

2.3 Echtzeiterweiterungen

Aufgrund der steigenden Anforderungen hinsichtlich der Bandbreite in Automobilnetzwerken, verdrängt Ethernet immer mehr Bussysteme. Ethernet ermöglicht höhere Bandbreiten und somit eine hohe Zukunftssicherheit. Gewöhnlicher Ethernet-Verkehr mit der Nutzung von Protokollen wie TCP und UDP interagieren nach dem Best-Effort (BE)-Prinzip. Da Ethernet nach dem IEEE-Standard 802.3 [1] keine Zeitgarantien bzw. keine Quality-of-Service (QoS) zusichert, wird innerhalb der IEEE 802.1 TSN Task Group [15] an Ethernet-Standards gearbeitet, um diese Lücke zu füllen. Unter der Verwendung von Ethernet ist die Kerndisziplin das Aufstauen von Nachrichten zu vermeiden, um Echtzeitfähigkeit zu erreichen. Dazu werden in diesem Kapitel die Kernmerkmale der relevanten

Ethernet Erweiterungen für diese Arbeit beschrieben. Für detailliertere Beschreibungen wird auf die Standards verwiesen.

2.3.1 Time-triggered Ethernet

Die folgende Erklärung zu Time-Triggered Ethernet (TTE) stützt sich auf die Doktorarbeit von Steinbach [25]. TTE ist eine time-triggered Ethernet-Lösung für den Einsatz in Fahrzeugen. Eine dieser Lösungen ist TTEthernet, welche seit Ende 2011 von der Society of Automotive Engineers (SAE) als AS6802 [24] standardisiert wurde. Von der IEEE 802.1 TSN Task Group [15] wird mit dem IEEE-Standard 802.1Qbv [4] eine eigene Echtzeit-Ethernet-Lösung herausgearbeitet. TTEthernet definiert neben time-triggered Datenverkehr auch rate-constrained und Best-Effort (BE)-Datenverkehr. Im Folgenden werden diese Verkehrsklassen erklärt.
TT-Datenverkehr arbeitet mithilfe einer global synchronisierten Zeit in einem Netzwerk. Dies wird erreicht, indem jeder Teilnehmer im Netzwerk seine lokale Uhr auf Basis eines Synchronisierungsprotokolls mit den der anderen synchronisiert. Um nun das Aufstauen zu vermeiden und den Nachrichtenfluss deterministisch zu halten, wird für jeden Teilnehmer ein Zeitschlitz festgelegt, in dem dieser seine Nachricht senden darf. Mit einem Zeitschlitz ist ein Zeitraum gemeint, in dem ein Netzwerkteilnehmer senden darf. Außerhalb des Zeitschlitzes sendet der Teilnehmer nicht. Auch die Dauer der Nachrichtenübertragung ist festgelegt. Als Resultat erhält man sich kaum verändernde Ende-zu-Ende-Latenzen sowie geringe Jitter. Dieses Verfahren wird auch als koordiniertes Time Division Multiple Access (TDMA) bezeichnet.
RC-Datenverkehr unterstützt im Gegensatz zum TT-Datenverkehr Echtzeitgarantien mithilfe der Limitierung der Bandbreite eines Senders. Für jeden Sender wird mit einer festgelegten Bandwidth Allocation Gap (BAG) der Abstand von Nachrichten definiert, wodurch die Anzahl von Nachrichten nicht überschritten werden kann. Dafür prüft jeder auf dem Weg liegende Switch die Einhaltung der BAG.
BE-Datenverkehr ist der am niedrigsten priorisierte Nachrichtenverkehr. Für Nachrichten dieser Verkehrsklasse werden lediglich die aktuell verfügbaren Ressourcen genutzt. Jede höher priorisierte Nachricht wird bevorzugt, wodurch BE-Nachrichten immer hinten angestellt werden. Wenn eine BE-Nachricht eine Echtzeitnachricht gefährdet und keine Ressourcen zum Buffern etc. zur Verfügung stehen, wird sie verworfen. In Ethernet-Netzwerken wird der Datenverkehr nach dem IEEE-Standard 802.3 [1] standardmäßig als BE-Datenverkehr behandelt.

2.3.2 Audio Video Bridging

Die folgende Erklärung zu Audio Video Bridging (AVB) stützt sich auf die Doktorarbeit von Steinbach [25]. Mit der Echtzeit-Ethernet-Erweiterung AVB wird der Ansatz verfolgt, Nachrichten möglichst gleichmäßig über die Zeit zu verteilen. Dafür wird ein sogenannter Credit Based Shaper (CBS) in Kombination mit Prioritätsklassen eingesetzt. AVB wird in IEEE 802.1Qav [3] standardisiert und ist eine Erweiterung des IEEE-Standard 802.1Q [2]. Der IEEE-Standard 802.1Q unterstützt die Zuweisung von Prioritäten zu Nachrichten. Die Prioritäten reichen von 0 bis 7. Diese spielen im CBS insofern eine Rolle, dass Prioritätsklassen auf diese Prioritäten abbildet. In AVB gibt es die beiden SR-Klassen (Stream Reservation) A und B sowie BE-Datenverkehr. Die Klasse A hat dabei die höchste Priorität, B die zweithöchste und der BE-Datenverkehr teilt sich in weitere Prioritäten auf. Weiter sind Zeitintervalle für die beiden SR-Klassen festgelegt. Die SR-Klasse A, darf alle 125 µs eine Nachricht senden, während bei der SR-Klasse B alle 250 µs gesendet werden darf. Dadurch bestimmt sich auch die maximale Nachrichtengröße eines Senders. Wird eine Nachricht der SR-Klasse A über eine 100 Mbit/s Leitung gesendet, welche die Leitung zu 50 % belastet, darf diese auch nur 50 % von 125 µs einnehmen. In konkreten Zahlen betrachtet man ein Datenvolumen von 50 Mbit pro Sekunde, was umgerechnet 6,25 MB entspricht. Entsprechend sind das 6,25 B pro Mikrosekunde. Multipliziert man 6,25 B/µs mit 125 µs, erhält man eine maximale Nachrichtengröße von etwa 780 B für diesen Fall. Zur Einhaltung der Sendefrequenz, wird der bereits erwähnte CBS eingesetzt. Mit dem CBS wird ein kreditbasiertes Verfahren realisiert. Ein Sender darf seine Nachricht erst Senden, wenn sein Kredit größer oder gleich null ist. Der Kredit steigt bis auf null, wenn keine Nachricht gesendet wird. Wird die Nachricht des Senders durch eine andere Nachricht blockiert, kann der Kredit auch über null ansteigen. Der Kredit eines Senders kann auch unter null absinken. Er kann wieder senden, sobald sein Kredit auf null oder größer angestiegen ist. Das folgende Beispiel zum CBS stützt sich auf die Masterarbeit von Meyer [16]. Abbildung 2.1 zeigt dazu ein Beispiel. In der Ausgangssituation ist der CBS-Kredit und der AVB-Puffer auf null. Jetzt wird eine BE-Nachricht auf den Ausgangsport gelegt und soll versendet werden. Zum Zeitpunkt t_1 soll eine AVB-Nachricht versendet werden, welche aber zunächst im AVB-Puffer gehalten wird, da noch die BE-Nachricht den Ausgangsport blockiert. Für die Dauer, welche die AVB-Nachricht im AVB-Puffer verweilt, steigt der CBS-Kredit. Zum Zeitpunkt t_2 ist die BE-Nachricht abgeschickt und die AVB-Nachricht liegt nun auf dem Ausgangsport. Während des Sendevorgangs der AVB-Nachricht sinkt der CBS-Kredit. Die AVB-Nachricht ist zum Zeitpunkt t_3 abgeschickt. Gleichzeitig ist der CBS-Kredit negativ. Bis Zeitpunkt t_4

wird gewartet bzw. keine AVB-Nachricht gesendet, wodurch der CBS-Kredit wieder auf null steigt. Jetzt kann wieder eine AVB-Nachricht gesendet werden. Es dürfen bei einem CBS-Kredit von null oder größer aufeinanderfolgende AVB-Nachrichten gesendet werden, bis der CBS-Kredit wieder kleiner null ist. So wird verlorene Bandbreite wieder aufgeholt, wenn bspw. eine BE-Nachricht den Ausgangsport blockiert hatte. Mit dem Stream Reservation Protocol (SRP) aus dem IEEE-Standard 802.1Qat wird in AVB dynamisch zur Laufzeit Bandbreite auf der Leitung für einen Datenstrom reserviert. Damit wird für den Echtzeitdatenverkehr ausreichend Bandbreite auf dem Pfad zwischen Sender und Empfänger sichergestellt. Stehen in einem Automobil die notwendigen Verkehrsflüsse zur Designzeit fest, ist auch die Konfiguration von statischen Strömen ohne SRP möglich.

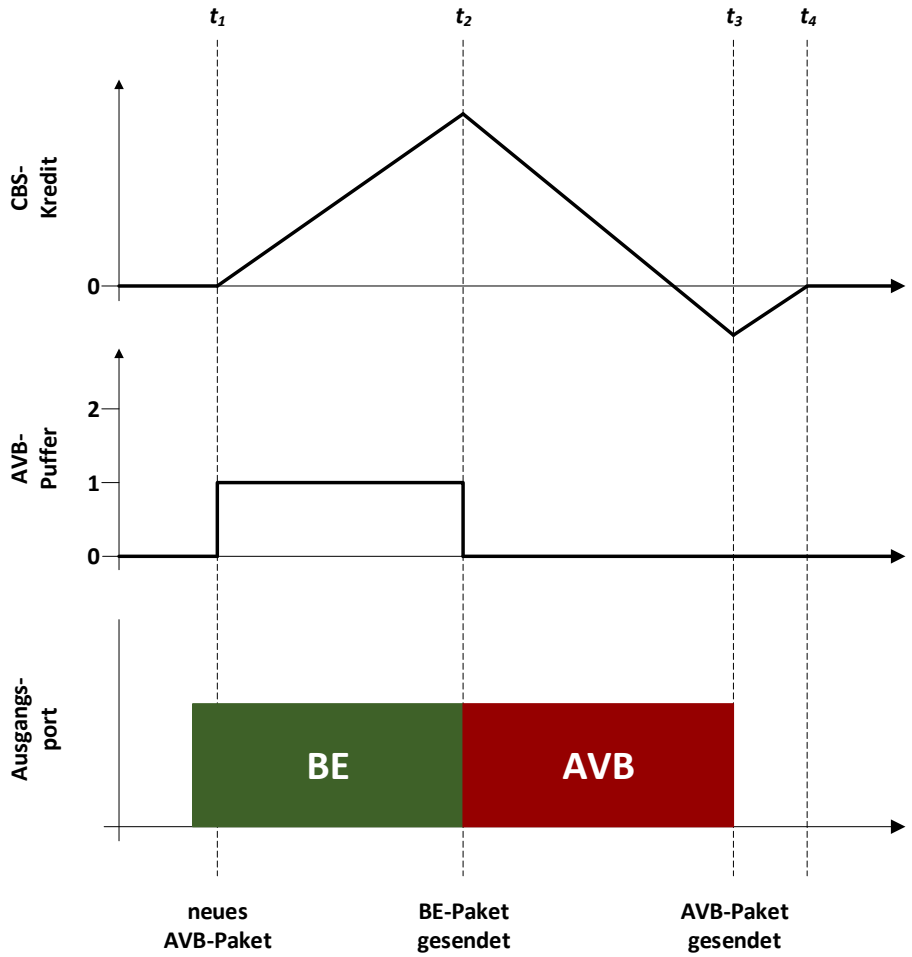

Abbildung 2.1: Credit Based Shaper Beispiel (Quelle: [16]).

2.4 Time-Sensitive Networking

Das folgende Erklärung zu Time-Sensitive Networking (TSN) stützt sich auf die Masterarbeit von Meyer [16]. TSN ist ein Echtzeit-Ethernet-Protokoll. Es umfasst eine Menge von Standards speziell für den Einsatz in Automobilnetzwerken und industriellen Kontrolleinrichtungen. Diese sind in der IEEE 802.1 Time-Sensitive Networking Task Group standardisiert [15]. Die Paketauswahlmethoden AVB und TDMA-Verkehr werden in TSN kombiniert, was eine vergleichbare Lösung wie TTE bietet. TTE wird in Kapitel 2.3.1 erklärt. Somit können Zeitschlitze in Kombination mit AVB genutzt werden. Daneben erweitert TSN den IEEE-Standard [2]. Der IEEE-Standard 802.1Q unterstützt die Zuweisung von Prioritäten zu Nachrichten. Die Prioritäten reichen von 0 bis 7. Abbildung 2.2 zeigt die Umsetzung der Paketauswahl. Jeder Priorität ist ein Pfad zugeordnet. Auf jedem Pfad gibt es einen Puffer, einen *Transmission Selection Algorithm* und ein *Transmission Gate*. Alle diese Komponenten münden in die *Transmission Selection*. Die *Transmission Selection* ist für das Entnehmen eines Pakets aus einem *Transmission Gate* zur weiteren Übertragung zuständig. Dabei wird das Paket mit der höchsten Priorität bevorzugt. Im *Transmission Selection Algorithm* stehen verschiedene Algorithmen zur Paketauswahl bereit. Ein Beispiel dafür ist der CBS-Algorithmus aus AVB. Sobald der *Transmission Selection Algorithm* ein Paket ausgewählt hat und es sendet, entscheidet als Nächstes das *Transmission Gate*. Das *Transmission Gate* hat die zwei Zustände *OPEN* und *CLOSED*. Das Paket wird erst gesendet, wenn der Zustand auf *OPEN* ist. Wann der Zustand auf *OPEN* und *CLOSED* wechselt, kann zeitgesteuert auf Basis einer Offlinekonfiguration vorgenommen werden. Beispielsweise kann hier das TDMA-Verfahren aus Kapitel 2.3.1 umgesetzt werden. Dabei wird also analog ein Zeitschlitz festgelegt, in dem ein *Transmission Gate* seinen Zustand auf *OPEN* wechselt. Außerhalb eines Zeitschlitzes ist der Zustand auf *CLOSED*.

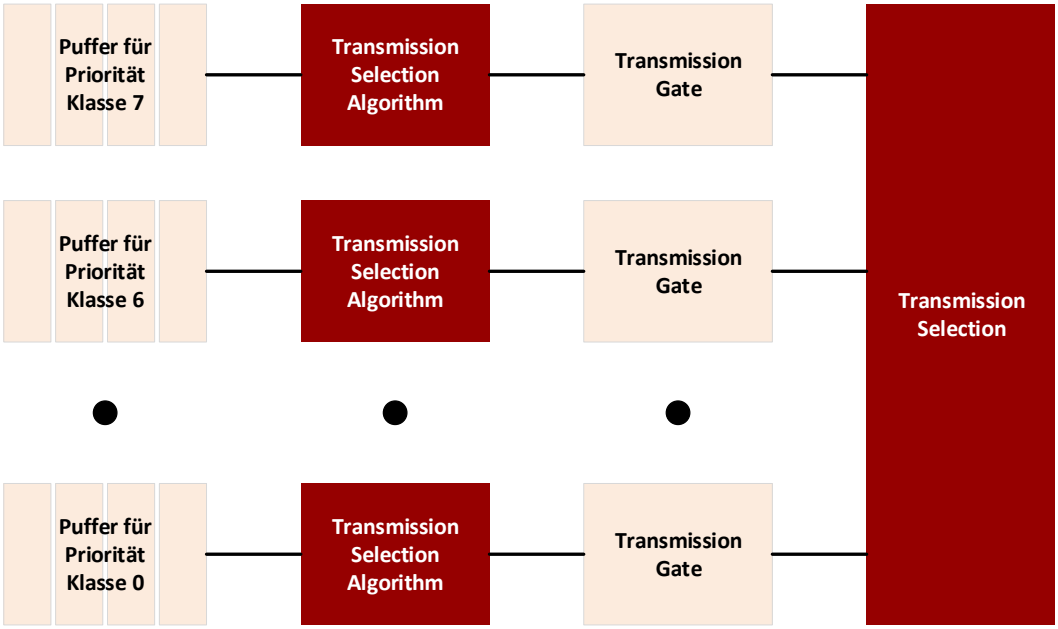

Abbildung 2.2: IEEE 802.1Qbv [4] Paketauswahl (Quelle: [16]).

2.5 Simulationsumgebung

Die Middleware wird als Machbarkeitsstudie in einer Simulationsumgebung untersucht. Als Umgebung dient OMNeT++ in Kombination mit dem weit verbreiteten INET-Framework sowie selbst entwickelten Frameworks der CoRE-Arbeitsgruppe [10]. OMNeT++ ist ein Framework von OpenSim Ltd. [21], welches eine hohe Erweiterbarkeit erlaubt um Netzwerksimulationen durchzuführen. Dabei werden einzelne Module und Topologien mit einer sogenannten Network Description Language (NED) beschrieben. Das Verhalten wird in C++ programmiert und event-basiert ausgelöst. Mit dem INET-Framework, ebenfalls von OpenSim Ltd. [20], kann OMNeT++ um zahlreiche Netzwerkkomponenten und Netzwerkprotokolle erweitert werden. Mit den Frameworks CoRE4INET, FiCo4OMNeT und SignalsAndGateways der CoRE-Arbeitsgruppe [10] stehen weitere Technologien und Komponeneten zur Verfügung. CoRE4INET bietet die Möglichkeit TSN-Technologien wie AVB und TTE zu simulieren. Mit FiCo4OMNeT wurde Feldbus-Kommunikation für CAN realisiert. Außerdem bietet SignalsAndGateways CAN-to-Ethernet Gateways an, um heterogene Netzwerke mit diesen zu simulieren. So können

ECUs an CAN-Bussen über solche Gateways mit einem Ethernet Backbone verbunden werden. Abbildung 2.3 zeigt die Zusammensetzung der Frameworks. Die Middleware ist in der Abbildung mit der Bezeichnung SOQoSMW gekennzeichnet. SOQoSMW steht für Service Oriented Quality-of-Service MiddleWare. Diese Bezeichnung wird in dieser Arbeit jedoch nicht weiter genutzt.

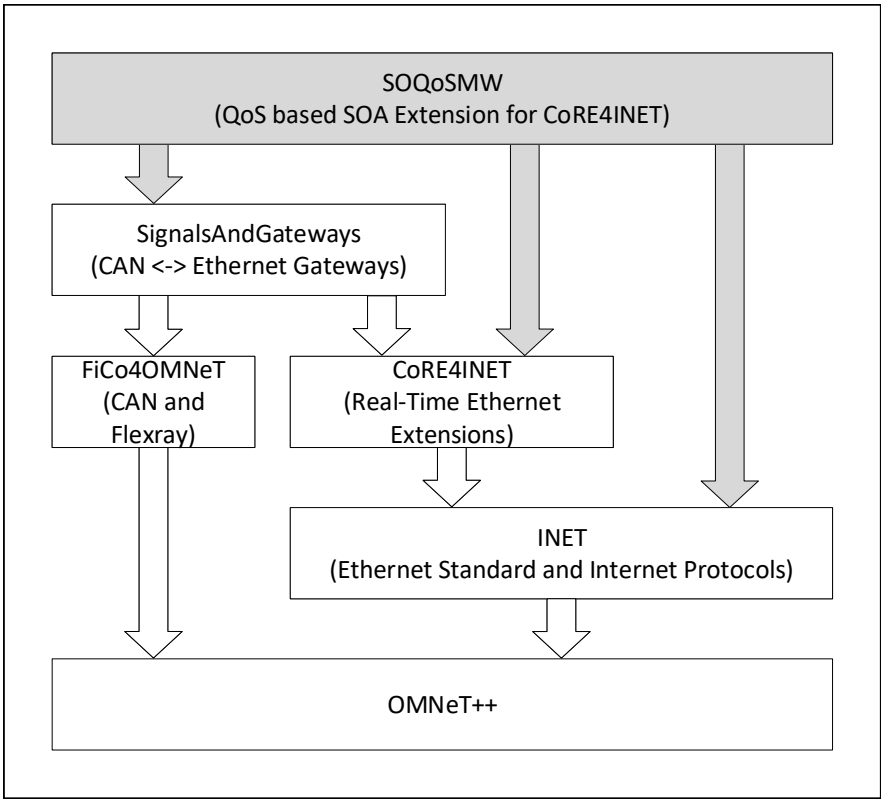

Abbildung 2.3: Zusammensetzung der Frameworks von OpenSim Ltd. [21] und der CoRE-Arbeitsgruppe [10] (Quelle: [13]).

3 Analyse

In den folgenden Kapiteln wird die Architektur der Middleware erklärt. Danach werden automobile Dienste in Dienstgüteklassen eingeordnet. Im Weiteren wird der Protokollstack genannt, der für die Dienstgüteklassen eingesetzt wird. Zudem wird das realistische Autonetzwerk erläutert, in welcher die Middleware evaluiert wird. Zusätzlich wird auf die Umwandlung des ursprünglichen Autonetzwerks von einer Domainenarchitektur in eine Zonalarchitektur eingegangen. Daraufhin wird die Abstract Network Description Language (ANDL) vorgestellt, welche die Konfiguration und Erzeugung von Netzwerken für das Framework OMNeT++ ermöglicht. Darauffolgend wird das QoS Negotiation Protocol (QoSNP) vorgestellt, welches das Protokoll zur Dienstgüteverhandlung realisiert und die Dienstgüteverhandlung dynamisch zur Laufzeit durchführt.

3.1 Architektur der Middleware

Im Nachfolgenden wird auf die Architektur der Middleware von Häckel [13] eingegangen. Abbildung 3.1 zeigt die Komponenten. Sowohl auf der Seite des Publishers, als auch auf der Seite des Subscribers sind die gleichen Komponenten vertreten. Ein Publisher ist die Komponente, die einen Dienst anbietet, wohingegen ein Subscriber den Dienst eines Publishers abonniert. Im Wesentlichen unterscheiden sie sich in ihren Applikationen. Beide Seiten besitzen statische Komponeneten wie den Local Service Manager (LSM), die Local Service Registry (LSR) und den Quality-of-Service Manager (QoSM). Diese statischen Komponenten sind jeweils einmal pro Knoten vorhanden. Die dynamischen Komponenten Connector und Endpoint sind pro Dienst enthalten. Die LSR realisiert eine Liste, wo alle registrierten Services gehalten werden. Der LSM dient als Eintrittspunkt zur Middleware. Mithilfe des LSM lassen sich Dienste in der LSR suchen und registrieren. Die LSR und der LSM bilden zusammen im Kontext einer SOA den Namensdienst. Für die Dienstgüteverhandlung dient der QoSM. Nach einer abgeschlossenen Dienstgüteverhandlung werden Connector und Endpoint erstellt. Der Connector fungiert als Schnittstelle

zwischen Applikation und Endpoint. Während der Publisher dem Connector Nachrichten übergibt, entnimmt der Subscriber Nachrichten vom Connector. Mit dem Endpoint wird ein Verbindungsendpunkt realisiert, der die tatsächliche Netzwerkkommunikation übernimmt. Hierdurch wird der Dienst von Protokollen entkoppelt. Eine Anwendung auf der Seite des Publishers kann somit über protokollspezifische Endpoints einen Dienst über verschiedene Netzwerkprotokolle übertragen. Wird ein Dienst über unterschiedliche Protokolle abonniert, werden jeweils entsprechende Endpoints genutzt. Abonnieren mehrere Anwendungen auf der Seite des Subscribers den selben Dienst und derselben Dienstgüte, nutzen diese gemeinsam einen entsprechenden Endpoint. Somit können weitere Kommunikationsmuster beziehungsweise Protokolle zur Netzwerkkommunikation umgesetzt werden, ohne Änderungen an der Anwendung vorzunehmen.

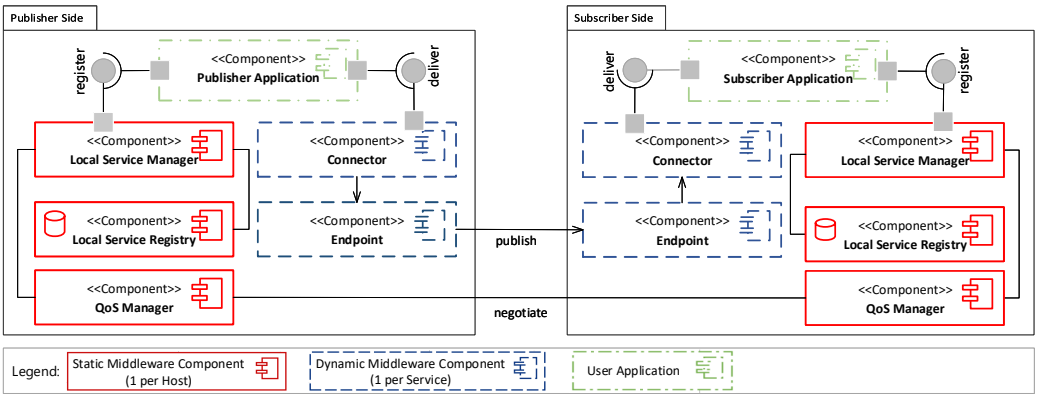

Abbildung 3.1: Komponentendiagramm der Middleware (Quelle: [8])

3.2 Klassifizierung von automobilen Diensten

Im Folgenden wird die Klassifizierung von automobilen Diensten aus der Masterarbeit von Häckel [13] erläutert. Daneben werden die Dienstgüteklassen definiert. In einem Automobilnetzwerk herrschen unterschiedliche Zeitanforderungen an die internen Systeme. Während für sicherheitskritische Funktionalitäten wie Bremsen die Bereitschaft in jedem Fall garantiert werden muss, ist es nicht genauso tragisch, wenn das Radio mal ausfällt. Abbildung 3.2 zeigt die vier Dienstgüteklassen, welche für die Middleware bestimmt wurden. Diese vier Dienstgüteklassen decken unterschiedliche Anforderungen heterogener Kommunikation ab.

Web-based Services (WS) fassen Dienste zusammen, die auf globale Angebote setzen.

Hierunter fallen Dienste wie Infotainment oder Smart City, wo eine globale Netzwerkkommunikation stattfindet. Hierbei könnten zentrale Informationsstandpunkte angesprochen werden, worüber Fahrzeuge Informationen über den Verkehr erhalten oder für den Verkehr relevante Informationen diesen übergeben. Beispielsweise könnten Fahrzeuge mit Ampeln kommunizieren, wodurch die Ampelsteuerung dynamisch für einen höheren Verkehrsdurchsatz sorgt. Weiter können Fahrzeuge durch kooperative Kommunikation sich über Staus oder Sperrungen informieren. Als Plattform sind aufgrund der hohen Datenmengen Cloud-Infrastrukturen und PCs vorgesehen.
IP-based Services (IPS) fassen Dienste zusammen, die keine zeitkritsichen Anforderunen haben. Dies wären Dienste, die weder globalen Zugriff benötigen noch Echtzeitanforderungen unterliegen. Beispiele hierzu wären Fensterheber oder Temperaturregler. Zur Kommunikation sind hier standardisierte Internetprotokolle wie TCP und UDP vorgesehen. Bezüglich der Kompatibilität sollten hier angesiedelte Dienste auf möglichst vielen Geräten funktionieren.
Real-Time Services (RTS) fassen zeitktritische Dienste zusammen. Hierunter würde Sicherheitselektronik wie das Elektronische Stabilitätsprogramm (ESP) oder die Heckkamera fallen. Diese unterliegen Echtzeitanforderungen. Das bedeutet, sie müssen höher priorisiert werden und Übertragungszeiten mit harten Deadlines erfüllen. Für RTS-Dienste eignen sich Protokolle wie AVB mit SRP oder TSN.
Static-Real-Time Services (SRTS) fassen zeitkritische Dienste zusammen, wo eine dynamische Konfiguration zur Laufzeit aufgrund der Anforderungen nicht möglich ist. Beispiele hierfür sind Airbags und Bremsen, für welche die Bereitschaft in jedem Fall garantiert werden muss. Hier sind Prozesse wie eine Dienstgüteverhandlung dynamisch zur Laufzeit nicht möglich. Eine Dienstgüteverhandlung birgt das Risiko, dass diese aufgrund von einem Nachrichtenverlust fehlschlagen kann. Auch eine erneute Übertragung der Nachricht garantiert nicht, dass diese beim Publisher oder Subscriber ankommt, da anderer Nachrichtenverkehr diese Nachricht immer wieder blockieren kann. Theoretisch kann also eine Dienstgüteverhandlung unendlich lange dauern, was bei sicherheitskritischen Diensten fatal ist. Deshalb ist es empfohlen bei sicherheitskritischen Diensten wie Bremsen und Airbags statische Konfigurationen zu wählen. Als Möglichkeit bietet sich hier TSN bzw. die Kombination aus AVB und TTE an, um SRTS-Dienste als TDMA-Verkehr zu konfigurieren. Die Klasse SRTS ist kein Bestandteil der Middleware.

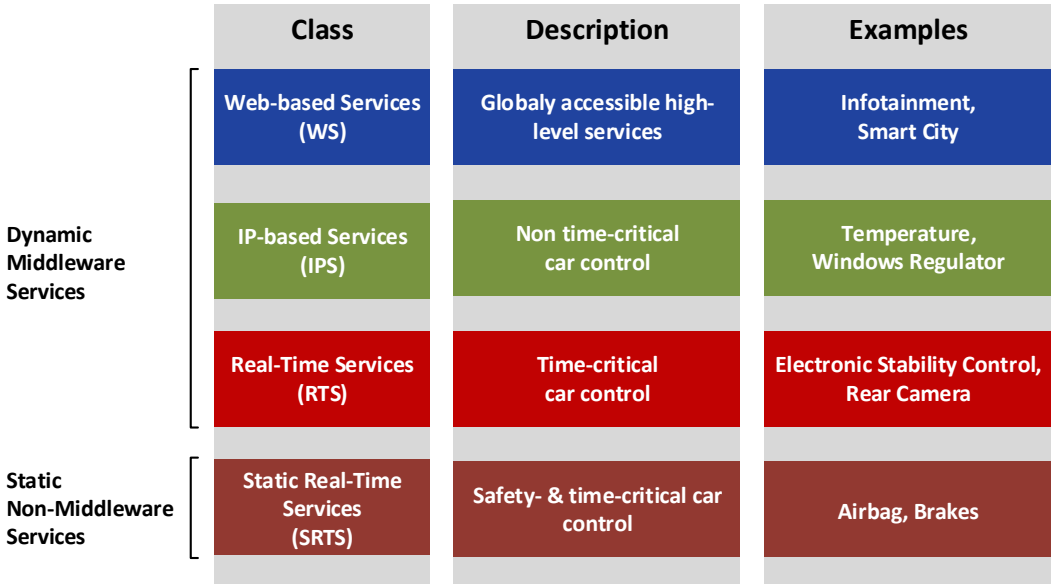

Abbildung 3.2: Klassifikation von automobilen Diensten in vier Dienstklassen

3.3 Protokoll zur Dienstgüteverhandlung

Häckel [13] hat ein QoS Negotiation Protocol (QoSNP) bzw. Protokoll zur Dienstgüteverhandlung entwickelt, welche eine dynamische Aushandlung zwischen einem Publisher und einem Subscriber realisiert. Ein Publisher ist die Komponente, die einen Dienst anbietet, wohingegen ein Subscriber den Dienst eines Publishers abonniert. Hierbei wird geprüft, ob der Dienst, den der Subscriber abonnieren will, beim entsprechenden Publisher in der angeforderten Dienstgüte zur Verfügung steht. Für die Dienstgüteverhandlung wird UDP genutzt. Im Folgenden wird das Protokoll schrittweise vorgestellt. Dabei wird in den folgenden Abbildungen anhand eines Sequenz- und Komponentendiagramms gezeigt, wie die Dienstgüteverhandlung abläuft und wie die Komponenten währenddessen miteinander interagieren. Die Middleware-Komponenten LSM, LSR, QoSM und der Connector, welche in Kapitel 3.1 vorgestellt wurden, werden zur Vereinfachung der Erklärung der Dienstgüteverhandlung nicht betrachtet. Erwähnenswert ist an dieser Stelle, dass die Dienstgüteverhandlung jeweils vom Quality-of-Service Manager (QoSM) der Middleware-Komponenten ausgeführt wird. Die Dienstgüteverhandlung durchläuft zwei Phasen. Phase 1 beschreibt das *Handshake*, wo geprüft wird ob der Dienst in der geforderten Dienstgüte verfügbar ist. Phase 2 beschreibt die *Connection*, in der die Verbin-

dung zur Netzwerkkommunikation aufgebaut wird. Neben dem Sequenzdiagramm sind die Komponenten der Publisher- und Subscriber-Seite abgebildet. Auf beiden Seiten gibt es die Middleware und eine entsprechende Anwendung. In Abbildung 3.3 ist die Ausgangssituation zu sehen, in der die Anwendung auf der Publisher-Seite bereits bei der Middleware registiert ist und einen Dienst bereitstellt. Auf der Subscriber-Seite ist die Anwendung noch nicht bei der Middleware registriert.

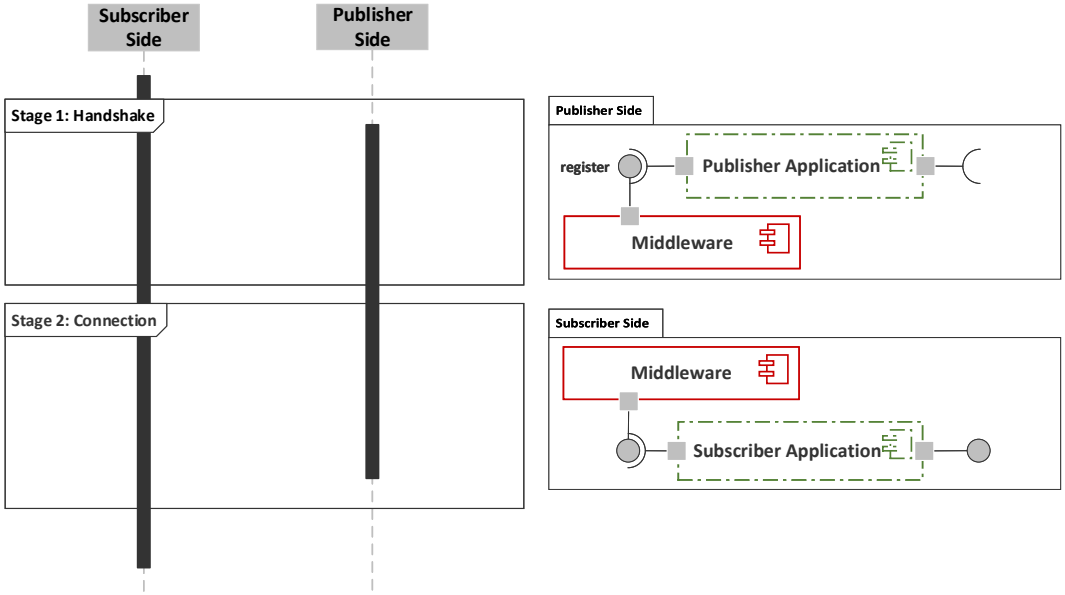

Abbildung 3.3: Ausgangssituation der Dienstgüteverhandlung: Der Publisher ist bei der Middleware registiert und bietet einen Dienst an. Der Subscriber ist noch nicht bei der Middleware registiert.

Abbildung 3.4 zeigt, dass sich die Anwendung auf der Subscriber-Seite bei der Middleware registriert, um den Dienst des Publishers zu abonnieren. Hierdurch wird die Dienstgüteverhandlung initiiert. Phase 1 wird betreten. Die Middleware auf der Subscriber-Seite sendet der Middleware auf der Publisher-Seite eine *QoSRequest*-Nachricht, in dem die geforderte Dienstgüte drin steht. Wenn der Dienst in der geforderten Dienstgüte angeboten wird, wird von der Publisher-Seite eine positive *QoSRespone*-Nachricht an die Subscriber-Seite geschickt. Phase 1 ist somit erfolgreich abgeschlossen. Bei einer negativen Rückmeldung wird die Dienstgüteverhandlung abgebrochen.

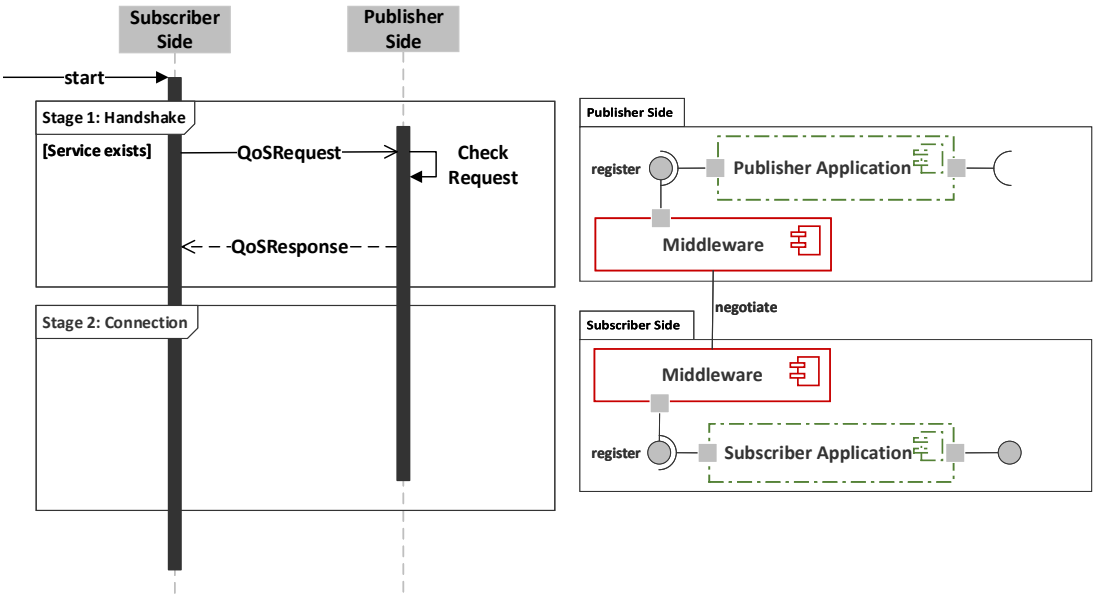

Abbildung 3.4: In Phase 1 wird ein Handshake durchgeführt.

Nachdem Phase 1 erfolgreich abgeschlossen wurde, geht die Dienstgüteverhandlung in Phase 2 über, welche nun in Abbildung 3.5 weiterverfolgt wird. Die Subscriber-Seite sendet hier eine *ConnectionRequest*-Nachricht, worin die verhandelten Eigenschaften mitgesendet werden und fordert somit den Verbindungsaufbau. Als Resultat wird auf der Publisher-Seite ein entsprechender Endpoint erstellt, der das entsprechende Netzwerkprotokoll zur geforderten Dienstgüte für die Netzwerkkommunikation nutzt. Würde der Dienst als IPS-Dienst gefordert werden, würde der Endpoint beispielsweise TCP als Netzwerkprotokoll entsprechend der zuvor verhandelten Eigenschaften nutzen. Für alle anderen Dienstgüteklassen wird auf die Abbildung 3.6 in Kapitel 3.4 verwiesen. Wenn die Erstellung des Endpoints auf der Publisher-Seite erfolgreich gewesen ist, sendet die Publisher-Seite eine positive *ConnectionResponse*-Nachricht, in welcher die Verbindungsinformationen wie IP-Adresse und Port mitgesendet werden. Bei einer negativen Rückmeldung wird die Dienstgüteverhandlung abgebrochen. Nach einer positiven Rückmeldung wird auf der Subscriber-Seite ebenfalls ein Endpoint erstellt. Anschließend wird entsprechend des Netzwerkprotokolls der gewählten Dienstgüte die Verbindung aufgebaut. Darauffolgend werden bei einem erfolgreichen Verbindungsaufbau aktiv von der Anwendung der Publisher-Seite dienstrelevante Nachrichten an die Anwendung der Subscriber-Seite gesendet. In der Middleware stand im Protokollstack für die Dienstgüteklasse IPS nur TCP zur Verfügung. Die notwendige Erweiterung sowie Änderung für UDP wird in

Kapitel 4.1 behandelt.

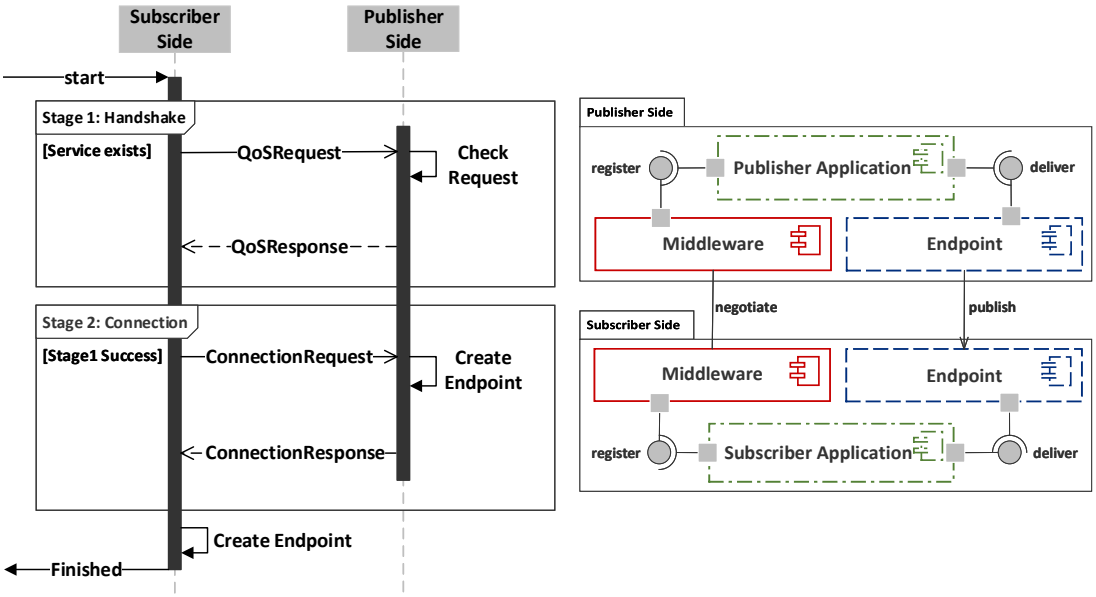

Abbildung 3.5: Phase 2 ist erfolgreich und die Verbindung wurde aufgebaut. Nachrichten werden nun übertragen.

3.4 Eingesetzte Protokolle

Zur Einhaltung von Echtzeitanforderungen und Realisierung globaler Dienste, ist die entsprechende Wahl von Netzwerkprotokollen entscheidend. Häckel [13] hat dabei einen Multi-Protokollstack eingesetzt. Abbildung 3.6 zeigt den Protokollstack der Middleware. Die Middleware wählt während der Dienstgüteverhandlung das entsprechende Protokoll entsprechend der Dienstgütanforderung. Anhand der Anforderungen eines Subscribers wird das entsprechende Protokoll zur Kommunikation mit dem Publisher bei der Dienstgüteverhandlung ausgewählt. Verläuft die Dienstgüteverhandlung erfolgreich, werden entsprechende Endpoints erstellt, wodurch die Netzwerkkommunikation von der Applikation der Publisher und Subscriber entkoppelt ist. Real-Time Services (RTS) benutzen Time-Sensitive Networking (TSN)-Prioritäten. Dadurch wird Bandbreite reserviert und das Einhalten von harten Echtzeitanforderungen garantiert. RTS werden direkt an die zweite OSI-Schicht übertragen. In Zukunft könnten vergleichbare Protokolle wie UDP oder SOME/IP in darüber liegenden Schichten implementiert und zur Übertragung genutzt werden. Verbindungsorientierte Protokolle wie TCP sind ungeeignet, weil sie aufgrund

von Mechnismen wie retransmitting und reihenfolgegesicherter Übertragung für hohe Verzögerungszeiten sorgen.
IP-based Services (IPS) nutzen den reinen IP-basierten Protokollstack zur Übertragung zur zweiten OSI-Schicht, wobei Best-Effort (BE)-Garantien genutzt werden. Entsprechend der Dienstgüteanforderung, wird TCP oder UDP auf der Transportebene des OSI-Modells genutzt.
Web-based Services (WS) nutzen das Hypertext Transfer Protocol (HTTP). Damit können alle modernen Implementierungen von Internetdiensten wie Simple Object Access Protocol (SOAP) oder Representational state transfer (ReST) realisiert werden. Da die Evaluation auf einem echten Automobilnetzwerk basiert, wird die WS-Klasse nicht evaluiert, da in diesem Automobilnetzwerk keine WS vorhanden sind.

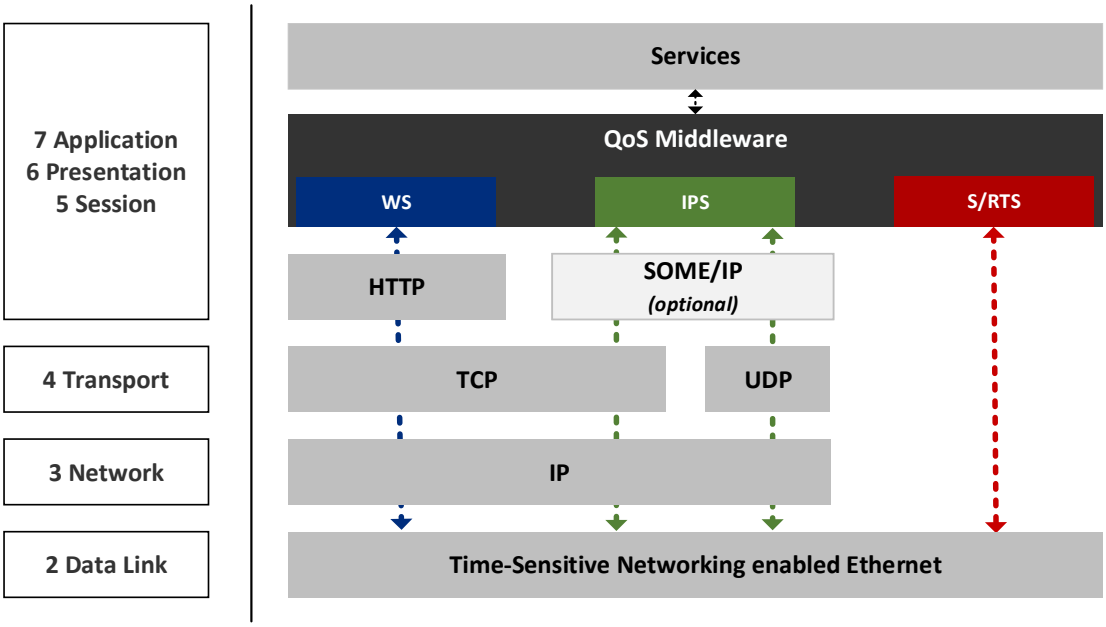

Abbildung 3.6: Multi-Protokollstack mit entsprechender Zuordnung zu den OSI-Schichten und Aufteilung in Dienstklassen (Quelle: [8])

3.5 ANDL

In OMNeT++ wird die Definition und Konfiguration von Netzwerken mithilfe von NED- und INI-Dateien vorgenommen. Dies kann bei komplexen Netzwerken sehr zeitaufwändig werden, worunter auch die Übersichtlichkeit leidet. Damit der Netzwerkplaner sich

auf den Entwurf des Netzwerks fokussieren kann, hat die CoRE-Arbeitsgruppe [10] die Abstract Network Description Language (ANDL) entwickelt. Die ANDL ist eine domänenspezifische Sprache zur Generierung von Netzwerken für die Simulation mit OMNeT++ [17]. Diese wurde mithilfe des Xtext Plugins [11] unter Eclipse entwickelt. Mit der ANDL ist es möglich mithilfe einer kompakten Syntax ein Netzwerk zu beschreiben. Im Folgenden wird zu einem Netzwerk die zugehörige ANDL gezeigt, mit der das Netzwerk erzeugt wurde. In Abbildung 3.7 ist ein Netzwerk mit zwei ECUs *Scheinwerfer* und *Lichtregulierung* abgebildet. Die beiden ECUs sind entsprechend über ihre CAN-Busse *Licht* und *Steuerung* an die CAN-to-Ethernet Gateways *GatewayLicht* und *GatewaySteuerung* angeschlossen, wodurch die beiden ECUs ans Ethernet Backbone angebunden werden. Durch den Ethernet-Switch *ethswitch* werden die beiden Gateways miteinander gekoppelt. In diesem Netzwerk ist der Nachrichtenfluss wie folgt. Die ECU *Scheinwerfer* sendet zyklisch jede Millisekunde eine CAN-Nachricht mit einer Größe von 8 B an *Lichtregulierung*. Die Nachricht wird mit der Dienstgüte BE übertragen.

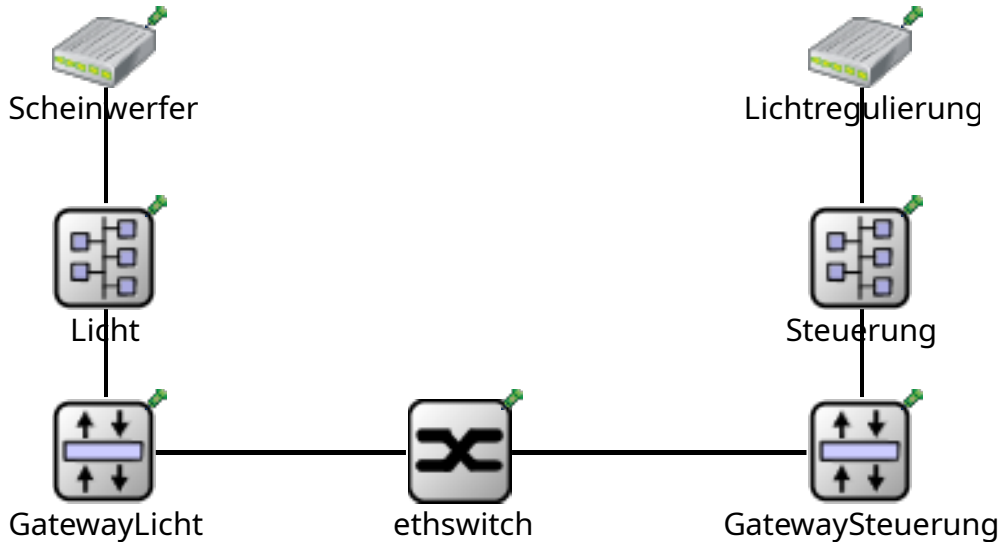

Abbildung 3.7: Netzwerk mit zwei ECUs Scheinwerfer und Lichtregulierung

Der ANDL-Code zum Netzwerk in Abbildung 3.7 ist in Listing 3.1 zu sehen. Die ANDL ist in Blöcke unterteilt, in denen die entsprechenden Merkmale zu einem Netzwerk angegeben werden. Im Folgenden werden die Blöcke näher erläutert. In dem Block *types* werden Typen definiert, die wiederverwendet werden können. Diese wären Netzwerkelemente und Nachrichten. Der Block *network* umfasst das zu erzeugende Netzwerk. Ab hier wird mit Unterblöcken das Netzwerk zusammengesetzt. Mit dem Block *devices* werden

Netzwerkknoten, Gateways, CAN-Busse und Switche angegeben. Der Typ eines Netzwerkknotens wird erst durch das Anschließen über ein Verbindungselement festgelegt. Wird der Netzwerkknoten über ein Ethernetkabel angebunden, wird es zu einem Ethernetknoten. Andererseits wäre die Anbindung des Netzwerkknotens über einen CAN-Bus möglich, wodurch der Netzwerkknoten zu einem CAN-Knoten bzw. einer ECU wird. Innerhalb des Blocks *connections* wird angegeben, wie die Netzwerkkomponenten miteinander verbunden sind. Zunächst muss jedoch innerhalb des Blocks *connections* der Block *segment* angegeben werden. Hierbei gilt es zu beachten, gegebenenfalls verschiedene Segmente für das Ethernet Backbone und CAN-Busse anzugeben, da dies wichtig beim *mapping* von Nachrichten wird. Werden Ethernetkomponenten miteinander verbunden, sind diese in einem eigenen Segment anzugeben. Verbindungen von CAN-Knoten mit CAN-Bussen und CAN-to-Ethernet Gateways mit CAN-Bussen müssen gemeinsam in einem anderen Segment angegeben werden. Der Nachrichtenfluss wird abschließend mit dem Block *communication* festgelegt. Innerhalb dieses Blocks kann mit dem Block *message* eine Nachricht angegeben werden. Für eine Nachricht müssen mit dem Schlüsselwort *sender* ein Sender, mit *receivers* ein oder mehrere Empfänger und mit *payload* die Nachrichtengröße angegeben werden. Weiter kann mit *period* angegeben werden, in welchem Zeitabstand die Nachricht zyklisch gesendet werden soll, sonst wird eine Zeitangabe per Gleichverteilung zufällig erzeugt. Schließlich müssen in dem Block *message* für eine Nachricht mit dem Block *mapping*, welcher bereits angedeutet wurde, mit *canbus* die Zuordnung von CAN-ID, mit *backbone* die Dienstgüte sowie die auf dem Kommunikationspfad involvierten CAN-to-Ethernet Gateways angegeben werden. Mithilfe der Informationen im Block *mapping* wird das Routing in den Gateways entsprechend gesetzt.

```
1  types std {  // Typen können definiert und wiederverwendet werden
2      ethernetLink ETH_100MBIT { // Definition eines Ethernetkabels
3          bandwidth 100Mb/s; // Die Verbindung hat eine Bandbreite von 100Mbit/s
4      }
5  }
6
7  network minimal_net { // Der Netzwerkname ist minimal_net
8      devices { // Definition aller devices im Netzwerk
9          node Scheinwerfer; // CAN-Knoten Scheinwerfer
10         node Lichtregulierung; // CAN-Knoten Lichtregulierung
11
12         canLink Licht; // CAN-Bus Licht
13         canLink Steuerung; // CAN-Bus Steuerung
14
15         gateway GatewayLicht; // Gateway GatewayLicht
16         gateway GatewaySteuerung; // Gateway GatewaySteuerung
```

```
17
18          switch ethswitch; // Ethernet Switch
19      }
20
21      connections { // Physikalische Verbindungen (segment = Gruppe)
22          segment backbone { // Ethernet Backbone Segment
23              // Ab hier folgen Ethernetverbindungen in dieser Gruppe
24              GatewayLicht <--> { new std.ETH_100MBIT } <--> ethswitch;
25              GatewaySteuerung <--> { new std.ETH_100MBIT } <--> ethswitch;
26          }
27
28          segment canbus { // CAN-Bus Segment
29              // Ab hier folgen CAN-Verbindungen in dieser Gruppe
30              GatewayLicht <--> Licht;
31              GatewaySteuerung <--> Steuerung;
32              Scheinwerfer <--> Licht;
33              Lichtregulierung <--> Steuerung;
34          }
35      }
36
37      communication { // Kommunikation im Netzwerk
38          message scheinwerfermsg { // Nachrichtendefinition für scheinwerfermsg
39              sender Scheinwerfer; // Scheinwerfer ist der Sender
40              period 1ms; // 1ms zyklisches Senden
41              payload 8B; // 8Byte Nachrichtengröße
42              receivers Lichtregulierung; // Lichtregulierung ist Empfänger
43              mapping { // Zuordnung der Dienstgüte, CAN-ID und eingebundenen Gateways
44                        // auf dem Übertragungsweg
45                  canbus : can { id 3; }; // CAN-ID ist 3
46                  GatewayLicht; // Nachricht wird über GatewayLicht übertragen
47                  backbone : be; // Dienstgüte der Nachricht ist Best-Effort
48                  GatewaySteuerung; // Nachricht wird über GatewaySteuerung übertragen
49              }
50          }
51      }
52  }
```

Listing 3.1: ANDL-Code zum Netzwerk in Abbildung 3.7

Das realistische Autonetzwerk aus Abbildung 3.9 wird bereits mit der ANDL erzeugt. Jedoch gibt es noch keine Möglichkeit mithilfe der ANDL dienstbasierte Netzwerke entsprechend der Middleware zu erzeugen. Da die Middleware auch im realistischen Autonetzwerk evaluiert werden soll, ist hier eine Erweiterung der ANDL zur effizienten Umstellung auf Services notwendig gewesen. Weitere Details dazu werden in Kapitel 4.2 behandelt.

3.6 Realistisches Autonetzwerk

Für die Evaluation der Middleware ist es vorgesehen, dessen Performance in einem realistischen Autonetzwerk zu ermitteln. Konkret wird es dabei darum gehen, wieviel Zeit benötigt wird bis Dienste in vollem Umfang zur Verfügung stehen. Der Datenfluss des Autonetzwerks basiert auf einer realen Kommunikationsmatrix. Tabelle 3.1 zeigt beispielhaft den Aufbau dieser Kommunikationsmatrix. Die Matrix enthält tatsächlich viel mehr Informationen bzw. Spalten darüber, wie zum Beispiel die Nachrichtengröße und in welchen Zeitintervallen oder zu welchen Konditionen sie gesendet wird. Jedoch sind diese Informationen für diese Arbeit bzw. die Evaluation nicht nennenswert, da sie für die spätere Umwandlung in ein dienstbasiertes Autonetzwerk mithilfe der ANDL nicht in Betracht gezogen werden. Im Autonetzwerk werden zahlreiche Nachrichten übertragen, die in der Matrix mit jeweils einer Identifikationsnummer bezeichnet werden, welche in der Spalte *Nachrichten-ID* aufgeführt wird. ECUs, die eine Nachricht senden, sind in der Spalte *Sender* mit ihrer Identifikationsnummer aufgeführt. Die ECUs, die eine Nachricht empfangen, sind in der Spalte *Empfänger* mit ihrer Identifikationsnummer angegeben. Für jede Nachricht ist in den meisten Fällen eine *Kritikalität* festgelegt, welche in der gleichnamigen Spalte angegeben ist. Diese gibt an, wie kritisch diese Nachricht ist bzw. mit welcher Priorität sie übertragen und behandelt wird. Für die Kritikalitäten gibt es die Stufen *1 für sehr kritisch*, *2 für kritisch* und *3 für nicht kritisch*. 1 ist somit die höchste Kritikalitätsstufe und 3 die niedrigste. Als *sehr kritisch* gilt Sicherheitselektronik wie der Airbag und der Bremskraftverstärker. *Kritisch* sind Funktionen wie die Einparkhilfe und das Heckradar. *Nicht kritisch* wäre das Radio und die Standheizung. Andererseits gibt es auch Nachrichten, für die keine Kritikalität festgelegt wurde. Für diese Nachricht gilt dann die Kritikalität der ECU, welche unter den Empfängern die höchste Kritikalitätsstufe hat. Mit der Kritikalität einer ECU ist gemeint, wie dringend sie Nachrichten benötigt. Bei den ECUs gilt das gleiche Kritikalitätssystem, wie bei den Nachrichten. Wenn eine Nachricht ohne Kritikalität von mehreren ECUs empfangen werden soll, die aber an unterschiedlichen Gateways angeschlossen sind, dann müssen jeweils die entsprechenden ECUs an den Gateways betrachtet werden. Im folgenden Beispiel sind ECU1, ECU2, ECU3 und ECU4 Empfänger einer Nachricht ohne Kritikalität, wovon aber ECU1 und ECU2 an einem anderen Gateway angeschlossen sind als ECU3 und ECU4. Dann gibt es die gleiche Nachricht mehrmals mit unterschiedlichen Kritikalitäten. Nämlich einmal auf dem Weg zu ECU1 und ECU2 sowie auf dem anderen Weg zu ECU3 und ECU4. So würde also für die eine Version der Nachricht die Kritikalität von ECU1 oder ECU2 gelten, für die andere Version die von ECU3 und ECU4. Es gilt wie bereits erwähnt die

Kritikalitätsstufe derjenigen ECU mit der höchsten Kritikalitätsstufe.

Nachrichten-ID	**Sender**	**Empfänger**	**Kritikalität**
0	45,80	90, 8, 33	1
1	54	67, 34, 43	2
2	99	77, 66	3
3	86	56	?
.	.	.	.
.	.	.	.
.	.	.	.

Tabelle 3.1: Aufbau der Kommunikationsmatrix des realistischen Autonetzwerkes

Das Autonetzwerk ist in einer Domänenarchitektur angeordnet und rein CAN-basiert. Abbildung 3.8 zeigt eine Abstraktion des Netzwerks. Man spricht hier von einer Domänenarchitektur, weil alle ECUs derselben Domäne an einem gemeinsamen CAN-Bus angeschlossen sind. Beispielsweise könnten alle ECUs am *CAN0-Bus* für die Sicherheitselektronik zuständig sein, während alle anderen am *CAN2-Bus* für Multimediaanwendungen zuständig sind. Außerdem ist in diesem Netzwerk der Ansatz mit einem zentralen Gateway gewählt worden, der alle CAN-Busse miteinander verbindet. Auch wenn Domänenarchitekturen heutzutage noch vermehrt eingesetzt werden, bringen diese Nachteile in der Verkabelung, Erweiterbarkeit sowie Wartbarkeit. Autos werden zunehmend intelligenter und das Repertoire an Funktionalitäten nimmt ebenfalls stetig zu. Dadurch steigt auch die Anzahl an ECUs. Wenn nun hunderte ECUs in einem Autonetzwerk verteilt platziert werden und das Konzept der Domänenarchitektur verfolgt wird, sind komplexe und lange Verkabelungen dieser Geräte miteinander notwendig. Als Maßnahme hat die CoRE-Arbeitsgruppe [9] für das domänenbasierte Netzwerk aus Abbildung 3.8 das Konzept der Zonenarchitektur verfolgt und es somit in ein zonenbasiertes Netzwerk transformiert, wovon eine Abstraktion in Abbildung 3.9 zu sehen ist. Hierbei wurden neun Zonen definiert, in denen jeweils ein Zonenkontroller platziert ist. Diese Zonenkontroller interagieren als CAN-to-Ethernet Gateways. Sie verbinden ECUs, die über ihren CAN-Bus an ihren entsprechenden Zonenkontroller angebunden sind, mit einem Ethernet Backbone. Aus Abbildung 3.8 sind beispielsweise alle sechs ECUs am *CAN0-Bus* in Abbildung 3.9 entsprechend ihrer Zone mit dem zuständigen Zonenkontroller verbunden. Somit erspart man sich einen langen CAN-Bus durch das ganze Netzwerk. Kommt eine neue ECU hinzu, braucht diese lediglich mit den ihr am naheliegendsten Zonenkontroller verbunden zu werden. Die Zonenkontroller sind über drei Switche im Ethernet Backbone miteinander verbunden. In diesem Netzwerk wird die Middleware in Kapitel 6.3

evaluiert.

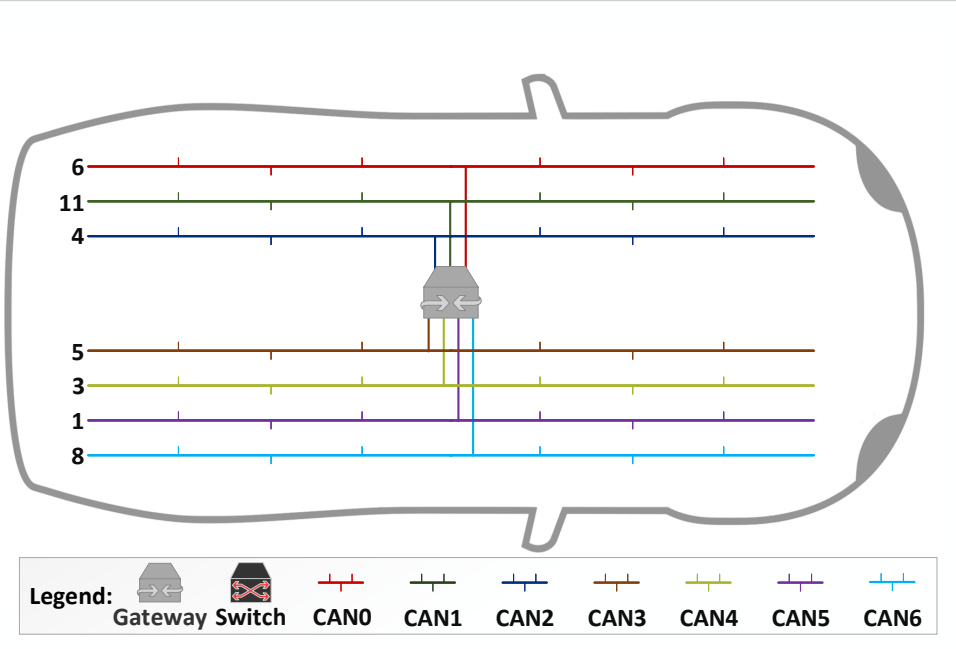

Abbildung 3.8: Realistisches Automobilnetzwerk in einer Domänenarchitektur.

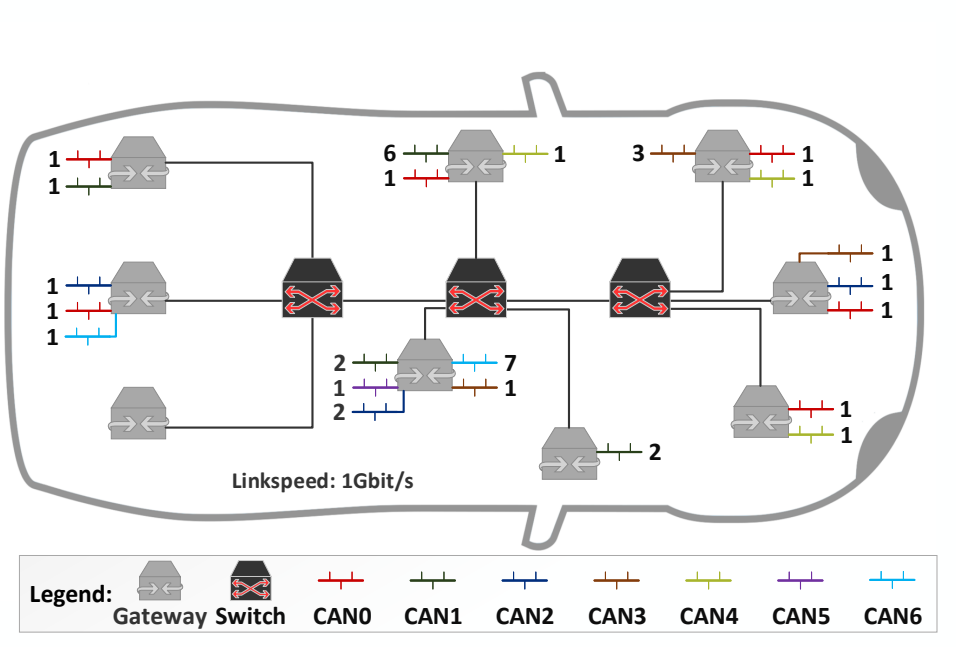

Abbildung 3.9: Realistisches Automobilnetzwerk in einer Zonalarchitektur.

4 Implementierung

Vor der Evaluation der Middleware, sind Erweiterungen notwendig gewesen. Die Middleware unterstützte keine UDP-Endpoints, weshalb IPS-Dienste nur über TCP kommunizieren konnten. Mit der ANDL bestand nicht die Möglichkeit, dienstbasierte Netzwerke zu beschreiben sowie zu erzeugen. In den folgenden Kapiteln werden zunächst die Erweiterungen an der Middleware zu den UDP-Endpoints vorgestellt. Daraufhin wird der Block in der ANDL erläutert, der für die Beschreibung von dienstbasierten Netzwerken eingeführt wurde. Anschließend wird auf eigene Skripte eingegangen, mit denen basierend auf der realen Kommunikationsmatrix des Autonetzwerks aus Kapitel 3.6 und dem bestehenden ANDL-Code dienstbasierter ANDL-Code erzeugt wurde. Daneben werden weitere Skripte genannt, mit denen Evaluationsergebnisse ausgewertet und zusammengeführt wurden.

4.1 UDP-Endpoints

Für die Netzwerkkommunikation verwendet die Middleware entsprechende Endpoints, je nachdem welche Dienstgüte gewählt wurde. Zuvor sind nur TCP für IPS-Dienste und AVB für RTS-Dienste hinsichtlich der Netzwerkkommunikation von der Middleware unterstützt worden. Jedoch stand UDP für IPS-Dienste nicht zur Verfügung. Für die Implementierung musste somit ein entsprechender Endpoint implementiert werden. Außerdem musste aufgrund der Eigenschaft, dass UDP ein verbindungsloses Protokoll ist, ebenfalls eine Anpassung im QoSNP vorgenommen werden. Das QoSNP wird in Kapitel 3.3 erklärt, ist aber in dieser Form nur für verbindungsorientierte Protokolle wie TCP und AVB geeignet und für verbindungslose Protokolle wie UDP ungeeignet. Der Grund dafür liegt darin, dass bei verbindungsorientierten Protokollen ein Verbindungsaufbau von einem Knoten zu einem anderen Bestandteil des Protokolls ist. Während eines Verbindungsaufbaus teilt ein Endpoint einem anderen Endpoint seine IP-Adresse und seinen Port mit. Dadurch weiß der andere Endpoint unter welcher IP-Adresse und welchem Port er einen anderen

Endpoint kontaktieren kann. Im Kontext der Middleware bedeutet das, dass wenn ein Subscriber-Endpoint über TCP oder AVB einen Dienst abonnieren will, der Publisher-Endpoint zwangsläufig aufgrund des Verbindungsaufbaus die IP-Adresse und den Port des Subscriber-Endpoints kennenlernt. So weiß der Publisher-Endpoint bereits zu diesem Zeitpunkt, wie der Subscriber-Endpoint erreicht werden kann. Bei verbindungslosen Protokollen wie UDP ist der Verbindungsaufbau nicht Bestandteil des Protokolls. Abbildung 3.5 zeigt, dass der Publisher-Endpoint erstellt wird, nachdem von der Subscriber-Seite eine *ConnectionRequest*-Nachricht gesendet wird, worin die verhandelten Eigenschaften mitgesendet werden. Da an dieser Stelle von der Subscriber-Seite zum letzten Mal eine Nachricht an die Publisher-Seite gesendet wird, ergibt sich gleichzeitig die letzte Gelegenheit der Publisher-Seite Informationen mitzuteilen. Da wie bereits erwähnt der Verbindungsaufbau unter UDP nicht stattfindet, wurde hier entschieden Verbindungsinformationen der Publisher-Seite zu senden. Gleichzeitig impliziert dies den Endpoint auf der Subscriber-Seite bereits hier erstellen zu müssen. Abbildung 4.1 zeigt das entsprechende Sequenzdiagramm. Phase 2 inkludiert somit für UDP zusätzlich die Erstellung des Endpoints auf der Subscriber-Seite, bevor die *ConnectionRequest*-Nachricht gesendet wird. In der *ConnectionRequest*-Nachricht werden wie zuvor die verhandelten Eigenschaften der Dienstgüte und zusätzlich Verbindungsinformationen wie IP-Adresse und Port des Endpoints mitgeteilt. Auf der Publisher-Seite wird wie gewohnt der Endpoint erstellt, wonach bereits hier Nachrichten von der Publisher-Seite zur Subscriber-Seite gesendet werden können. Nach dem Eintreffen der *ConnectionResponse*-Nachricht entfällt die Erstellung des Endpoints auf der Subscriber-Seite und der Verbindungsaufbau. Danach wird die Dienstgüteverhandlung lediglich abgeschlossen, da der Verbindungsaufbau bereits nach dem Eintreffen der *ConnectionRequest*-Nachricht auf der Publisher-Seite hergestellt wurde.

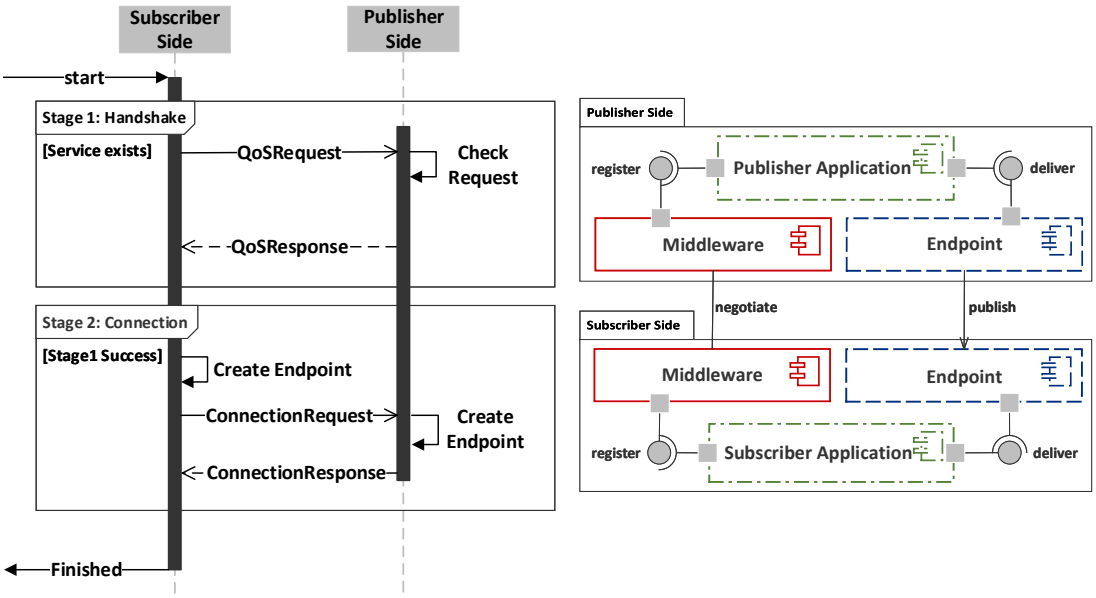

Abbildung 4.1: Sequenzdiagramm zum Protokoll für die Dienstgüteverhandlung bei UDP

4.2 Änderungen der ANDL

Die Abstract Network Description Language (ANDL) wird in Kapitel 3.5 vorgestellt und beschrieben. Ursprünglich hatte die ANDL keine Unterstützung zur Beschreibung von dienstbasierten Netzwerken geboten. Deshalb wurde nach einer Idee gesucht, Ethernetknoten und CAN-to-Ethernet Gateways künftig auch mithilfe der ANDL dienstbasiert zu beschreiben. Die erste Idee war innerhalb der Knoten einen Block *service* einzuführen und darunter die nötigen Parameter zu bestimmen, welche Listing 4.1 zeigt. Die Knoten werden bei dieser Idee mit den Schlüsselwörtern *publisher* und *subscriber* jeweils als Publisher oder Subscriber identifiziert. Bei einem Publisher steht innerhalb des Blocks *service* der Name des Dienstes, welcher hinter dem Schlüsselwort *servicename* angegeben wird. Für einen Subscriber wird im Block *service* der Name des zu abonnierenden Dienstes angegeben, welcher hinter dem Schlüsselwort *servicename* steht. Weiter wird mit dem vorangehenden Schlüsselwort *qos* die Dienstgüte angegeben, mit der ein Dienst vom Publisher zum Subscriber übertragen wird. Das gleiche Schema gilt auch für die CAN-to-Ethernet Gateways. Darüber hinaus wird für die Gateways, die als Subscriber angegeben wurden, mit dem Schlüsselwort *canIds* die CAN-IDs angegeben, wodurch entsprechende CAN-Nachrichten von ECUs im Kontext eines Dienstes übertragen werden sollen. Das Konzept aus Listing 4.1 wurde jedoch verworfen, da ein anderer Ansatz gefunden wurde,

der sich bezüglich seiner Modularität besser eignet.

```
1  node publisher Radio { // Publisher-Knoten mit dem Namen Radio
2      service { // Service-Block für Radio
3          serviceName "RadioPublisher"; // Der angebotene Service ist RadioPublisher
4      }
5  }
6
7  node subscriber Speaker { // Subscriber-Knoten mit dem Namen Speaker
8      service { // Service-Block für Speaker
9          serviceName "RadioPublisher"; // Der abonnierte Service ist RadioPublisher
10         qos udp // Der Service wird als IPS-Dienst (UDP) abonniert
11     }
12 }
13
14 gateway publisher GatewayLicht { // Publisher-Gateway mit dem Namen GatewayLicht
15     service { // Service-Block für GatewayLicht
16         serviceName "ScheinwerferPublisher"; // Der angebotene Service ist
               ScheinwerferPublisher
17     }
18 }
19
20
21 gateway subscriber GatewaySteuerung { // Subscriber-Gateway mit dem Namen
       GatewaySteuerung
22     service { // Service-Block für GatewaySteuerung
23         serviceName "ScheinwerferPublisher"; // Der abonnierte Services ist
               ScheinwerferPublisher
24         qos rt; // Der Service wird als RT-Dienst abonniert
25         canIds 5; // Nachrichten mit der CAN-ID 5 werden im Service-Kontext übertragen
26     }
27 }
```

Listing 4.1: Erstes Konzept zu ANDL-Code mit Unterstützung von Diensten

Statt des ersten Ansatzes aus Listing 4.1 wurde das Konzept aus Listing 4.2 umgesetzt. Der ANDL-Code aus Listing 4.2 erzeugt das dienstbasierte Netzwerk aus Abbildung 4.2. Bis auf den dienstbasierten Ethernet-Knoten *InfotainmentSystem*, bildet es topologisch das gleiche Netzwerk wie aus Abbildung 3.7 in Kapitel 3.5. Eine entsprechende Erklärung bis auf den dienstbasierten Teil findet sich dort. Die Zeilen 54 bis 60 aus Listing 4.2 zeigen den ANDL-Code, der für den dienstbasierten Teil verantwortlich ist. Bei diesem Ansatz wurde ein Block *service* innerhalb des Blocks *communication* eingeführt. Zunächst wird hinter dem Schlüsselwort *service* der Name des Dienstes angegeben. Danach werden weitere Parameter innerhalb des Blocks *service* für diesen Dienst angegeben. Hinter dem Schlüsselwort *publisher* wird der Name eines CAN-to-Ethernet Gateways oder Ethernet-Knotens angegeben, der den Dienst anbietet. Innerhalb des Blocks *subscriber* wird die

Dienstgüte mit *TCP*, *UDP* oder *RT* angegeben, gefolgt von einem oder mehrerer Namen von CAN-to-Ethernet Gateways oder Ethernet-Knoten, die dann diesen Dienst mit der entsprechenden Dienstgüte abonnieren. Mit dem Schlüsselwort *canIds* werden die CAN-IDs der CAN-Nachrichten angegeben, die im Kontext eines Dienstes übertragen werden. Mit Blick auf das Netzwerk in Abbildung 4.2 folgt nun eine konkrete Erläuterung.

```
1  types std {  // Typen können definiert und wiederverwendet werden
2      ethernetLink ETH_100MBIT { // Definition eines Ethernetkabels
3          bandwidth 100Mb/s; // Die Verbindung hat eine Bandbreite von 100Mbit/s
4      }
5  }
6
7  network minimal_net { // Der Netzwerkname ist minimal_net
8      devices { // Definition aller devices im Netzwerk
9          node Scheinwerfer; // CAN-Knoten Scheinwerfer
10         node Lichtregulierung; // CAN-Knoten Lichtregulierung
11         node InfotainmentSystem; // Ethernet-Knoten InfotainmentSystem
12
13         canLink Licht; // CAN-Bus Licht
14         canLink Steuerung; // CAN-Bus Steuerung
15
16         gateway GatewayLicht; // Gateway GatewayLicht
17         gateway GatewaySteuerung; // Gateway GatewaySteuerung
18
19         switch ethswitch; // Ethernet Switch
20     }
21
22     connections { // Physikalische Verbindungen (segment = Gruppe)
23         segment backbone { // Ethernet Backbone Segment
24             // Ab hier folgen Ethernetverbindungen in dieser Gruppe
25             GatewayLicht <--> { new std.ETH_100MBIT } <--> ethswitch;
26             GatewaySteuerung <--> { new std.ETH_100MBIT } <--> ethswitch;
27             InfotainmentSystem <--> { new std.ETH_100MBIT } <--> ethswitch;
28         }
29
30         segment canbus { // CAN-Bus Segment
31             // Ab hier folgen CAN-Verbindungen in dieser Gruppe
32             GatewayLicht <--> Licht;
33             GatewaySteuerung <--> Steuerung;
34             Scheinwerfer <--> Licht;
35             Lichtregulierung <--> Steuerung;
36         }
37     }
38
39     communication { // Kommunikation im Netzwerk
40         message scheinwerfermsg { // Nachrichtendefinition für scheinwerfermsg
41             sender Scheinwerfer; // Scheinwerfer ist der Sender
42             period 1ms; // 1ms zyklisches Senden
43             payload 8B; // 8Byte Nachrichtengröße
44             receivers Lichtregulierung,InfotainmentSystem; // Lichtregulierung und
```

```
                InfotainmentSystem sind Empfänger
45              mapping { // Zuordnung der Dienstgüte, CAN-ID und eingebundenen Gateways
46                        // auf dem Übertragungsweg
47                  canbus : can { id 3; }; // CAN-ID ist 3
48                  GatewayLicht; // Nachricht wird über GatewayLicht übertragen
49                  backbone : be; // Dienstgüte der Nachricht ist Best-Effort
50                  GatewaySteuerung; // Nachricht wird über GatewaySteuerung übertragen
51              }
52          }
53
54      service scheinwerferservice { // Der Servicename ist scheinwerferservice
55        publisher GatewayLicht; // Der Publisher ist GatewayLicht
56        subscriber { // Hierunter werden die Subscriber angegeben
57          RT GatewaySteuerung; // Subscriber GatewaySteuerung abonniert mit RT
58          UDP InfotainmentSystem; // Subscriber InfotainmentSystem abonniert mit UDP
59        }
60        canIds 3; // Nachricht scheinwerfermsg mit der CAN-ID 3 wird zum Service
61      }
62    }
63  }
```

Listing 4.2: ANDL-Code mit Service-Beschreibung zum Netzwerk in Abbildung 4.2

Das Netzwerk aus Abbildung 4.2 besteht aus den zwei ECUs *Scheinwerfer* und *Lichtregulierung* sowie dem Ethernet-Knoten *InfotainmentSystem*. Die Nachricht *scheinwerfermsg* ab Zeile 40 aus dem Listing 4.2 hat als Sender die ECU *Scheinwerfer* und als Empfänger *Lichtregulierung* und *InfotainmentSystem*. Durch den Block *mapping* wird das Routing entsprechend erzeugt und die CAN-ID der Nachricht mit 3 gesetzt. Der Block *service* sorgt dafür, dass die ECU *Scheinwerfer* im Gateway *GatewayLicht* als Publisher abstrahiert wird. Dies wird erzielt, indem es die CAN-Nachricht mit der CAN-ID 3, was hinter dem Schlüsselwort *canIds* angegeben wurde, im Kontext des entsprechenden Dienstes sendet. In diesem Fall wird die Nachricht einmal als RTS-Dienst mit AVB über das Ethernet Backbone an *GatewaySteuerung* übertragen. Durch die Angabe von *Lichtregulierung* als Empfänger hinter dem Schlüsselwort *receivers* im Block *communication* wird die Nachricht dann als CAN-Nachricht weiter an die ECU *Lichtregulierung* geleitet. Dann wird die Nachricht noch als IPS-Dienst mit UDP über das Ethernet Backbone an den Ethernetknoten *InfotainmentSystem* gesendet. Das Konzept aus dem Listing 4.2 wurde vorgezogen, da es die Blöcke der bisherigen ANDL syntaktisch kaum beeinflusst. Der Block *service* kann nach Belieben hinzugefügt werden, wenn eine CAN-Nachricht dienstbasiert übertragen werden soll.

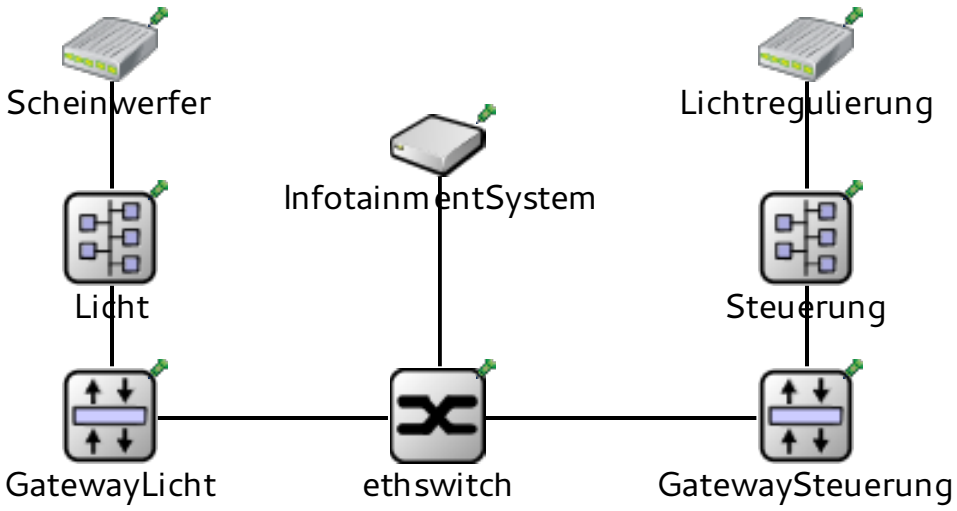

Abbildung 4.2: Netzwerk mit zwei ECUs Scheinwerfer und Lichtregulierung sowie einem Ethernet-Knoten InfotainmentSystem

4.3 Erzeugen von neuem ANDL-Code

Mit der Erweiterung der ANDL, welche in Kapitel 4.2 vorgestellt wird, besteht die Grundlage dienstbasierte Netzwerke komfortabel zu erzeugen. Für die Evaluation der Middleware ist es notwendig das realistische Autonetzwerk aus Abbildung 3.9 in Kapitel 3.6 auf ein dienstbasiertes Netzwerk umzustellen. Der Nachrichtenfluss im Autonetzwerk basiert auf eine reale Kommunikationsmatrix, in der für jede Nachricht Kritikalitäten festgelegt sind. Die reale Kommunikationsmatrix wird in Kapitel 3.6 in einem größeren Umfang erklärt. Nachrichten, denen keine Kritikalität zugeordnet ist, erhalten die Kritikalität des Empfängers, dessen Kritikalität am höchsten ist. Für die Kritikalitäten gibt es die Stufen *1 für sehr kritisch*, *2 für kritisch* und *3 für nicht kritisch*. 1 ist somit die höchste Kritikalitätsstufe und 3 die niedrigste. Nachrichten, die unter die Kritikalitätsstufe 1 fallen, bleiben unberührt und werden nicht als dienstbasierte Nachricht übertragen. Für dienstbasierte Nachrichten, die unter die Kritikalitätsstufe 2 oder 3 fallen, muss eine Konfiguration erfolgen, um sie als dienstbasierte Nachricht zu übertragen. Im Netzwerk werden 208 CAN-Nachrichten verschickt, von der jede darauf überprüft werden muss, unter welche Kritikalität sie fällt, um sie ggf. auf eine dienstbasierte Nachricht umzustellen. Dieses Vorgehen wäre per Hand sehr mühsam. Daher wurde in der Programmiersprache Python ein Programm geschrieben, welches die Überprüfung der Nachrichten auf ihre

Kritikalität überprüft und für jede Nachricht ggf. den entsprechenden ANDL-Code bzw. *service*-Block erzeugt. In einer älteren Version wurde aufgrund eines Missverständnisses die Kritikalität einer Nachricht, welcher keine Kritikalität zugeordnet gewesen ist, falsch bestimmt. Es wurde die Kritikalität der ECU unter den Empfängern gewählt, welcher die niedrigste Kritikalität zugeordnet ist. Jedoch sind nicht alle ECUs unter den Empfängern am gleichen CAN-to-Ethernet Gateway angeschlossen, weshalb dies differenziert betrachtet werden muss. Mit dem folgenden Beispiel wird dies verdeutlicht. Listing 4.3 zeigt einen *message*-Block einer CAN-Nachricht. Der Sender ECU105 ist am Gateway *gw_RearLeft* angeschlossen.

```
1  ...
2
3  message msg22 {
4    sender ecu105;
5    receivers ecu106,ecu107,ecu111,ecu113,ecu114,ecu115,ecu116,ecu121,ecu124,ecu125;
6    payload 8B;
7    period 200ms;
8    mapping {
9      canbus : can{id 1246;};
10     backbone : be;
11     gw_CenterLeft;
12     gw_Center;
13     gw_CenterRight;
14     gw_RearLeft;
15   }
16 }
17
18 ...
```

Listing 4.3: ANDL-Code zu einer CAN-Nachricht

Tabelle 4.1 zeigt die Zuordnung der Empfänger-ECUs und den CAN-to-Ethernet Gateways, an denen sie angeschlossen sind. Außerdem zeigt die Tabelle die Kritikalität der ECUs, mit der sie die Nachricht erwarten. Die ECUs bzw. CAN-Knoten empfangen Nachrichten über die CAN-to-Ethernet Gateways aus dem Ethernet Backbone. Die Middleware interagiert in den CAN-to-Ethernet Gateways. Somit bestimmt man anhand der Kritikalität der Nachricht oder in diesem Fall anhand der Kritikalität der ECU mit der niedrigsten Kritikalität die Dienstgüte, mit der das Gateway die Nachricht empfangen muss. Mit Blick auf die Tabelle 4.1 ergibt sich also für das Gateway *gw_CenterRight* die Kritikalität 2, da die ECU124 mit ihrer Kritikalität 2 kleiner ist als die von ECU115. Für das Gateway *gw_CenterLeft* ergibt sich ebenfalls die Kritikalität 2. Gateway *gw_Center* erhält die Kritikalität 3. Für Kritikalität 2 wurde die Dienstgüte RTS mit AVB sowie die SR-Klasse A festgelegt, für Kritikalität 3 die Dienstgüte IPS mit UDP.

ECU-ID	Kritikalität	Gateway
115	3	gw_CenterRight
124	2	gw_CenterRight
106	3	gw_CenterLeft
107	2	gw_CenterLeft
111	2	gw_CenterLeft
114	2	gw_CenterLeft
121	3	gw_CenterLeft
125	3	gw_CenterLeft
113	3	gw_Center
116	3	gw_Center

Tabelle 4.1: Zuordnung von ECU am angeschlossenen Gateway und Kritikalität der ECU

Somit resultiert der ANDL-Code aus Listing 4.4. Wie bereits erwähnt ist der Sender ECU105 am Gateway *gw_RearLeft* angeschlossen, weshalb *gw_RearLeft* der Publisher ist. Die Gateways *gw_CenterLeft* und *gw_CenterRight* abonnieren anhand ihrer Kritikalität den Dienst mit der Dienstgüte RTS, welches mit dem vorangehenden Schlüsselwort *RT* angegeben wird. Das Gateway abonniert anhand seiner Kritikalität den Dienst mit der Dienstgüte IPS über UDP, was mit dem vorangehenden Schlüsselwort *UDP* angegeben wird.

```
1  ...
2
3  service service1503 {
4      publisher gw_RearLeft;
5      subscriber {
6          RT gw_CenterLeft,gw_CenterRight;
7          UDP gw_Center;
8      }
9      canIds 1246;
10 }
11
12 ...
```

Listing 4.4: Dienstbasierter ANDL-Code zur CAN-Nachricht in Listing 4.3

In einer älteren Version wurde bereits wie erwähnt die Kritikalität nicht differenziert bzw. unabhängig davon an welchen Gateways die ECUs angeschlossen sind betrachtet. Mit Blick zurück auf die Tabelle 4.1 wurde unter allen Empfängern die niedrigste Kritikalität betrachtet, welche 2 ist und somit die Dienstgüte RTS abbildet. Mit diesem Ansatz resultierte der ANDL-Code aus Listing 4.5, wo alle Gateways den Dienst mit der Dienstgüte RTS mit AVB abonnierten, was falsch ist.

```
1  ...
2
3  service service1503 {
4      publisher gw_RearLeft;
5      subscriber {
6          RT gw_CenterLeft,gw_CenterRight,gw_Center;
7      }
8      canIds 1246;
9  }
10
11 ...
```

Listing 4.5: Falscher dienstbasierter ANDL-Code zur CAN-Nachricht in Listing 4.3

Der Ablauf in dem aktuellen Programm verläuft folgendermaßen. Zunächst werden aus dem bestehenden ANDL-Code zum realistischen Autonetzwerk die Zuordnungen von ECU zu Gateway, an dem die ECU angeschlossen ist, bestimmt und in einem Dictionary gespeichert. Der Datentyp Dictionary speichert Schlüssel-Wert Paare und in diesem Fall ECU als Schlüssel und Gateway als Wert. Danach werden die CAN-Nachrichten extrahiert und in dem Datentyp *Message* gespeichert. In Abbildung 4.3 ist ein entsprechendes Klassendiagramm zu sehen. Im nächsten Schritt werden aus der realen Kommunikationsmatrix die Nachrichten-IDs für die entsprechende CAN-Nachricht mithilfe der CAN-ID extrahiert. Die Kommunikationsmatrix enthält für jede CAN-Nachricht eine Nachrichten-ID, welche nicht im ANDL-Code steht. Mithilfe der Nachrichten-ID werden nun die Kritikalitäten aus der Kommunikationsmatrix für eine Nachricht extrahiert. Ist einer Nachricht keine Kritikalität zugeordnet, werden die Empfänger-ECUs, die am gleichen Gateway angeschlossen sind, zusammengefasst und in einer Liste gespeichert. Diese Liste wird im Datentyp *SummarizedNodes* gespeichert, wozu in Abbildung 4.4 ein entsprechendes Klassendiagramm zu sehen ist. In *SummarizedNodes* wird die höchste Kritikalität gesetzt, welche unter den ECUs mithilfe der Kommunikationsmatrix bestimmt wird. Anschließend werden in einem Dictionary die Gateways als Schlüssel und die entsprechenden *SummarizedNodes* als Wert gespeichert. Der Datentyp *Message* bekommt dann in dem Fall, wenn ihr keine Kritikalität zugeordnet ist, dieses Dictionary mit Gateways und den entsprechenden zusammengefassten Knoten in *SummarizedNodes* samt höchster Kritikalität. Im letzten Schritt wird mithilfe der Informationen aus *Message* für jede Nachricht dessen Kritikalität höher 1 ist, der Datentyp *Service* angeleget und entsprechend befüllt. Abbildung 4.5 zeigt ein entsprechendes Klassendiagramm. Aus diesem *Service*-Datentyp wird der ANDL-Code generiert. In Summe wurden 144 Dienste erzeugt.

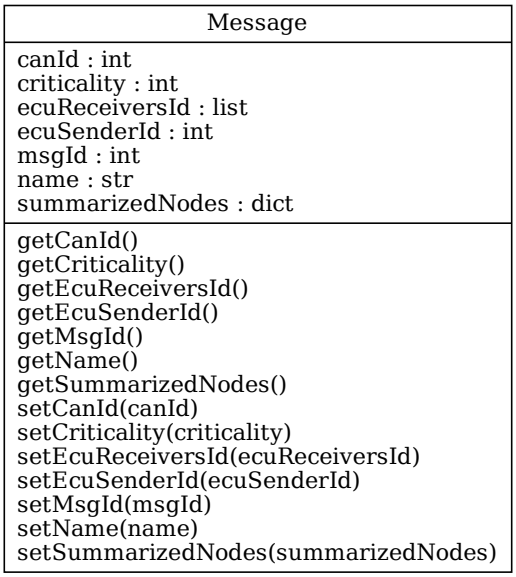

Abbildung 4.3: Im Datentyp Message werden die Eigenschaften einer CAN-Nachricht aus der ANDL zusammengefasst.

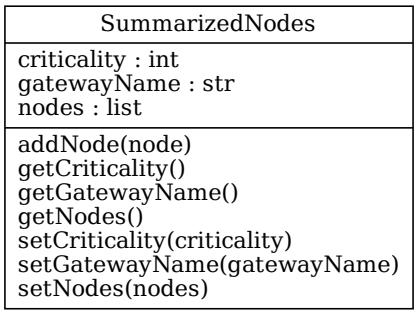

Abbildung 4.4: Im Datentyp SummarizedNodes werden alle Knoten zusammengefasst, die an einem gemeinsamen Gateway angeschlossen sind. Unter den Knoten wird die höchste Kritikalität bestimmt.

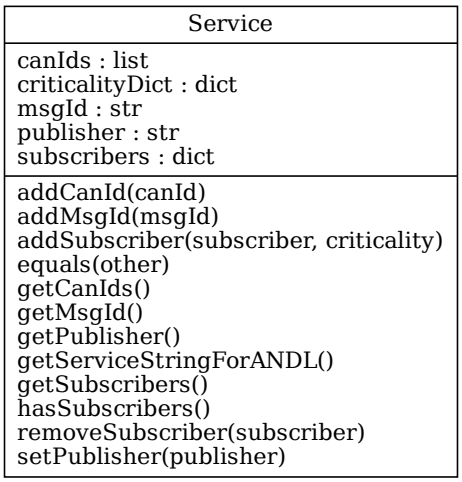

Abbildung 4.5: Mit dem Datentyp Service wird aus den Informationen aus dem Datentyp Message ein Dienst zusammengesetzt und der *service*-Block für den ANDL-Code generiert.

4.4 Ergebnisanalyse und Zusammenführung

Während der Qualitätssicherung der Middleware im realistischen Autonetzwerk ließ sich feststellen, dass Nachrichten verloren gehen. Die Ursache dafür wird im Kapitel 5.1 erläutert. Mit einem Tool von der CoRE-Arbeitsgruppe [9] lassen sich zu CAN-Nachrichten aus den Simulationsergebnissen, die von OMNeT++ nach einer Simulation erstellt werden, Informationen extrahieren. Diese Informationen werden in der Form wie in Listing 4.6 dargestellt. Am Anfang werden die Nachrichtenverluste zu CAN-Nachrichten aufgelistet. In diesem Ausschnitt ist zu sehen, dass bei der CAN-Nachricht mit der CAN-ID 22 ein Nachrichtenverlust von -1 aufgetreten ist. Nach der Kurzzusammenfassung aller Nachrichtenverluste zu den jeweiligen CAN-IDs folgen mehr Details. Zu der CAN-Nachricht mit der CAN-ID 22 sind Informationen zu dem Sender und den Empfängern der Nachricht. Unter *Source* ist die ECU aufgelistet, welche die Nachricht sendet. Die ECUs, welche die Nachricht empfangen, sind unter *Sinks* aufgelistet. Zu jeder ECU ist das Gateway angegeben, an der sie angeschlossen ist. Die Zahlen ganz rechts geben bei einer Sender-ECU an, wie oft die Nachricht verschickt wurde. Bei Empfänger-ECUs gibt sie an, wie oft die Nachricht angekommen ist. In diesem Beispiel hat ECU80 die Nachricht 1000 mal gesendet, während ECU76 sie 999 mal empfangen hat. Der ANDL-Code zum *message*-Block zur CAN-Nachricht mit der CAN-ID 22 findet sich in Listing 4.7. Für diese Nachricht ergab sich bei der Erzeugung von ANDL-Code ein Dienst, dessen *ser-*

vice-Block in Listing 4.8 zu sehen ist. An dieser Stelle wird nochmal angemerkt, dass die Zuordnung zwischen *message*- und *service*-Block über die CAN-ID stattfindet.

```
...

ID   22                      -1

...

ID   22:
Source:
  ecu80 :   gw_FrontRight   1000
Sinks:
  ecu76 :   gw_Rear          999
  ecu85 :   gw_FrontLeft    1000
  ecu90 :   gw_Front        1000
```

Listing 4.6: Ergebnisse zur Sende- und Empfangsanzahl der CAN-Nachricht mit der CAN-ID 22

```
message msg83 {
  sender ecu80;
  receivers ecu76,ecu85,ecu90;
  payload 8B;
  period 10ms;
  mapping {
    canbus : can{id 22;};
    backbone : be;
    gw_FrontLeft;
    gw_Front;
    gw_FrontRight;
    gw_Rear;
  }
}
```

Listing 4.7: ANDL-Code zur CAN-Nachricht mit der CAN-ID 22

```
service service1097 {
    publisher gw_FrontRight;
    subscriber {
        RT gw_Rear,gw_Front;
    }
    canIds 22;
}
```

Listing 4.8: ANDL-Code des *service*-Blocks zur CAN-Nachricht mit der CAN-ID 22

Eine weitere notwendige Information ist der Zeitpunkt, an dem ein Endpoint erstellt wurde. Der Endpoint übernimmt in der Middleware die Netzwerkkommunikation. Eine Nachricht wird verworfen, sofern kein Endpoint bereitsteht. Daher werden mithilfe

von OMNeT++ in jedem Connector, welcher die Schnittstelle zwischen Anwendung und Endpoint bildet, folgende Details im JSON-Format wie in Listing 4.9 protokolliert. Dazu gehört der Name des Gateways hinter dem Schlüsselwort *gatewayName* über den ein Dienst seine Nachrichten verschickt bzw. empfängt. Weiter wird der Name der Anwendungen bzw. der Dienste hinter dem Schlüsselwort *applications* aufgelistet. Hinter dem Schlüsselwort *endpoints* werden Endpoints mit ihrem jeweiligen Erstellzeitpunkt in Sekunden aufgezählt. Die entsprechenden Details zum Dienst aus dem Listing 4.8 zeigt Listing 4.9.

```
1 {
2   "gatewayName":"gw_FrontRight",
3   "applications":["service1097Publisher"],
4   "endpoints":[{"name":"publisherEndpoints[0]","created":"0.00022405"}]
5 }
```

Listing 4.9: Information im JSON-Format über den Erstellzeitpunkt in Sekunden von einem Endpoint des Dienstes *service1097Publisher* auf Gateway *gw_FrontRight*

Es sind insgesamt 18 CAN-Nachrichten von Verlusten betroffen gewesen. Daher wurde hier ein Programm in der Programmiersprache Python geschrieben, da dies per Hand fehleranfälliger wäre. Zu Beginn werden wie in Kapitel 4.3 alle Dienste erzeugt. Diese liegen somit jeweils in dem Datentyp *Service* wie im Klassendiagramm in Abbildung 4.5 vor. Um später schnelleren Zugriff über die CAN-ID auf einen *Service* zu haben, werden die CAN-ID als Schlüssel und der *Service* als Wert in einem Dictionary gehalten. Danach werden die Ergebnisse zu den Nachrichtenverlusten extrahiert. Die Informationen zu den Nachrichtenverlusten einer CAN-Nachricht werden im Datentyp *MessageCheck* gehalten. Abbildung 4.6 zeigt das dazugehörige Klassendiagramm. Der Eintrag für einen Sender wird in der Membervariable *source* gehalten. Die Einträge für die Empfänger werden in der Liste *sinks* gehalten. Für die jeweiligen Einträge der Sender und Empfänger wird der Datentyp *MessageCheckEntry* verwendet, dessen Klassendiagramm in Abbildung 4.7 gezeigt wird. Für schnellen sowie komfortablen Zugriff wird die CAN-ID als Schlüssel und die entsprechende *MessageCheck*-Instanz als Wert auch hier in einem Dictionary gehalten. Nun wird mithilfe der zuvor erstellten Dictionary, wo CAN-ID als Schlüssel und die entsprechende *MessageCheck*-Instanz als Wert gehalten wird, eine Zuordnung zwischen *Service*- und *MessageCheck*-Objekten hergestellt. Diese wird in dem Datentyp *MessageCheckService* gehalten, dessen Klassendiagramm in Abbildung 4.8 zu sehen ist. Alle *MessageCheckService*-Objekte werden zur Weiterverarbeitung in einer Liste gehalten.

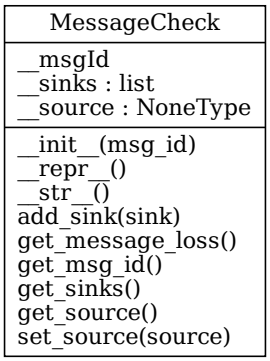

Abbildung 4.6: In dem Datentyp MessageCheck werden die extrahierten Informationen zu Nachrichtenverlusten einer CAN-Nachricht gehalten.

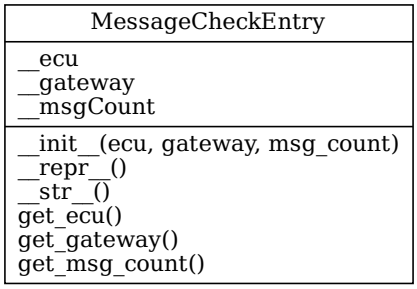

Abbildung 4.7: In dem Datentyp MessageCheckEntry wird aus den Ergebnissen der Nachrichtenverluste ein Eintrag einer ECU gehalten.

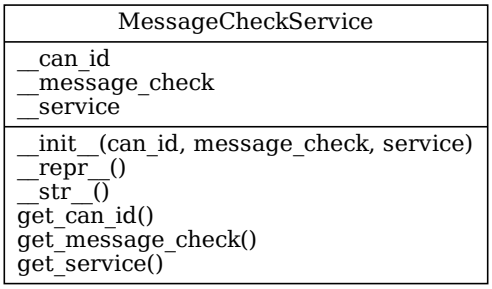

Abbildung 4.8: In dem Datentyp MessageCheckService wird die Zuordnung vom Datentyp *Service* und *MessageCheck* gehalten.

Zur weiteren Verarbeitung wurde der Datentyp *ConnectorMapping* aus Abbildung 4.9 entworfen, um die Informationen aus Listing 4.9 nach der Extraktion aus der JSON-Notation darin zu halten. Für die Endpoints wird der Datentyp *Endpoint* aus Abbildung 4.10 verwendet, worüber *ConnectorMapping* eine Liste hält. Zur Weiterverarbeitung werden *ConnectorMapping*-Objekte in einer Liste gehalten.

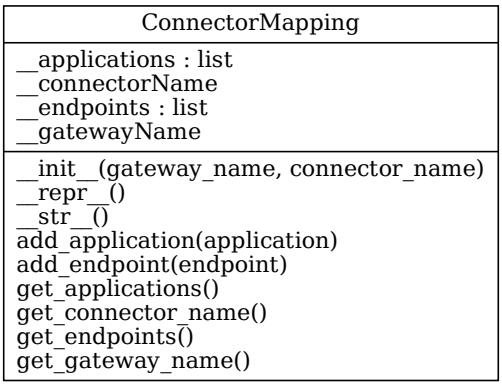

Abbildung 4.9: In dem Datentyp ConnectorMapping werden die Informationen aus Listing 4.9 gehalten.

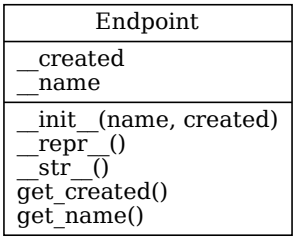

Abbildung 4.10: Im Datentyp Endpoint wird jeweils ein Endpoint gehalten.

Jetzt wird mithilfe der Listen über *MessageCheckService*- und *ConnectorMapping*-Objekte die Zuordnung von *Service* und *ConnectorMapping* über die Dienst-ID hergestellt. Mit Blick auf die Listings 4.8 und 4.9 wäre das hier die 1097. Die Zuordnung von *Service* und *MessageCheck* besteht bereits in *MessageCheckService*. Die eigentlichen Einträge werden in *TableEntry*, wozu ein Klassendiagramm in Abbildung 4.11 zu sehen ist. Zusammengehörende Einträge werden in dem Datentyp *TableSection* gehalten. Dazu ist ein Klassendiagramm in Abbildung 4.12 zu sehen. *TableSection* bildet eine Struktur für zusammengehörende *TableEntry*-Objekte. Für das hier behandelte Beispiel resultiert die Tabelle A.2 aus Kapitel A. Auf diese Tabellen wird nicht näher drauf eingegangen. Sie bieten die Basis zur Erklärung von Nachrichtenverlusten. Die Ursachen für Nachrichtenverluste werden in Kapitel 5.1 erklärt. Dort ist ebenfalls auch in den beiden Tabellen 5.1

und 5.2 eine kompaktere Darstellung als die der in Kapitel A.

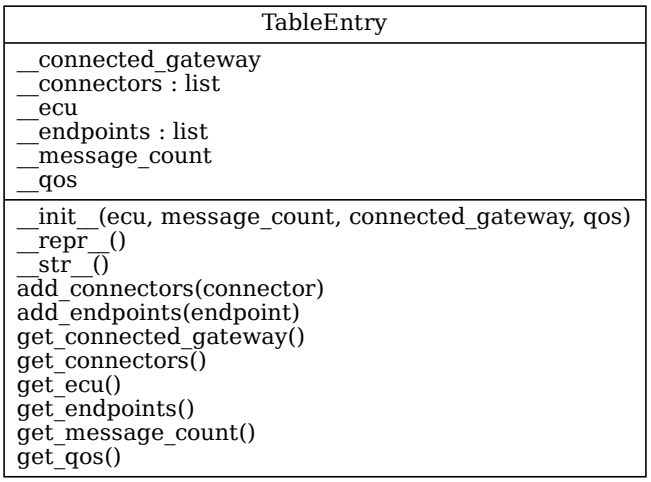

Abbildung 4.11: Der Datentyp TableEntry enthält die eigentliche Zusammenführung der Informationen aus dem Datentyp *MessageCheckService* und denen aus Listing 4.9

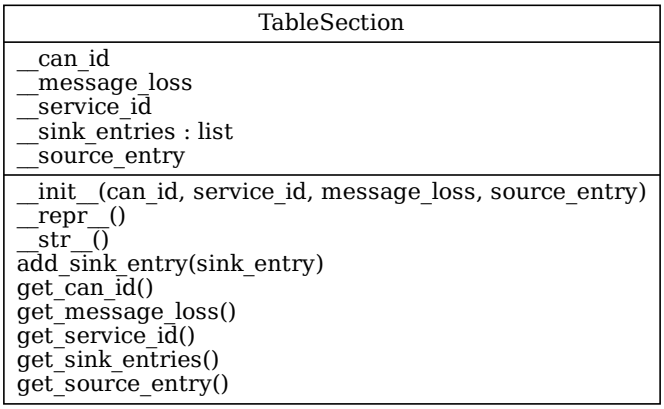

Abbildung 4.12: Der Datentyp TableSection dient als Struktur, in der entsprechende *TableEntry*-Objekte gehalten werden.

5 Qualitätssicherung

Mit der Qualitätssicherung werden Grundfunktionen der Middleware überprüft. Dabei wurden verschiedene Netzwerkkonfigurationen erstellt, welche durch automatisierte Tests auf ihre Stabilität und ihr Verhalten getestet werden. OMNeT++ bietet hierfür im Rahmen des INET-Frameworks zwei entsprechende Skripte an. Mit Smoketests werden Netzwerke für eine bestimmte Zeit simuliert und dabei auf Abstürze und Laufzeitfehler geprüft. So wird lediglich sichergestellt, dass das Netzwerk nicht abstürzt. Es finden keine semantischen Überprüfungen über das Verhalten des Netzwerks statt. Fingerprinttests berechnen am Ende einer Simulation eines Netzwerks einen Hashwert. Mit diesem Hashwert kann bei erneuter Ausführung festgestellt werden, ob sich das Verhalten des Netzwerks beispielsweise nach einer Änderung im Code geändert hat. Smoketests und Fingerprinttests geben keine Auskunft darüber, ob das Netzwerk sich sinnvoll verhält. Um auch auf ein sinnvolles Verhalten zu prüfen, wurde die Middleware in einem realistischen Automobilnetzwerk simuliert. Hierfür wurde in Kapitel 5.1 geprüft, ob durch den Einsatz der Middleware gegenüber der ursprünglichen Konfiguration Nachrichtenverluste entstehen.

5.1 Nachrichtenverluste

Zur Überprüfung hinsichtlich der Nachrichtenverluste, wurden die Simulationsergebnisse des Automobilnetzwerks ohne und mit Middleware miteinander verglichen. Mit einem selbstentwickelten Tool der CoRE-Arbeitsgruppe [9] werden Informationen über die Anzahl gesendeter sowie empfangener Nachrichten aus der Protokolldatei der Simulationsergebnisse extrahiert. Zur Sicherstellung, dass zum Zeitpunkt der Simulation keine CAN-Nachrichten mehr unterwegs sind, hören die ECUs nach zehn Sekunden auf zu senden. Es wurde insgesamt zwanzig Sekunden lang simuliert, sodass noch genügend Zeit zum Auslaufen für Nachrichten bleibt, die noch im Netzwerk unterwegs sind. Im ursprünglichen Automobilnetzwerk ohne Middleware kommen alle gesendeten Nachrichten in einer

Simulation von zwanzig Sekunden Laufzeit an. Mit der Middleware treten minimale Nachrichtenverluste bei gleicher Laufzeit auf. Als Ursache für die Nachrichtenverluste wurde die Dienstgüteverhandlung angenommen. Deshalb wurde die Dauer der Dienstgüteverhandlung plus die Dauer des Verbindungsaufbaus gemessen, dessen Summe im weiteren Verlauf als Setup-Time bezeichnet wird. Erst nach der Setup-Time ist ein Dienst vollständig bereit. Die Dauer des Verbindungsaufbaus spielt nur bei verbindungsorientierten Netzwerkprotokollen wie TCP und AVB eine Rolle. Bei UDP entfällt der Verbindungsaufbau, weil UDP ein verbindungsloses Netzwerkprotokoll ist. Genauer gesagt sind nach der Setup-Time die Endpoints auf Publisher- und Subscriber-Seite existent und bereit für die Netzwerkkommunikation. Die ECUs beginnen jedoch unabhängig von der Dienstgüteverhandlung Nachrichten zu senden bzw. senden unabhängig von der Setup-Time. Detaillierte Ergebnisse dazu finden sich im Anhang in Kapitel A.1, auf welche in dieser Arbeit jedoch nicht eingegangen wird. Stattdessen werden in diesem Kapitel diese Ergebnisse in einer kompakten Darstellung erläutert. Auffallend ist dabei, dass immer nur eine CAN-Nachricht verloren geht. Desweiteren tritt jeder Nachrichtenverlust unter 1 ms auf. Das bedeutet gleichzeitig, dass alle Dienste spätestens nach 1 ms bereit sind. In Kapitel 6 wird benannt, dass somit die Voraussetzungen für die Dienstbereitschaft in Autonetzwerken eingehalten wird. Während der Setup-Time treten drei unterschiedliche Fälle auf, in denen Nachrichtenverluste auftreten. (1) Nachrichten werden verworfen, wenn der Endpoint für den Publisher für eine entsprechende CAN-Nachricht zum Sendezeitpunkt noch nicht besteht. (2) Nachrichten können dem Empfänger nicht zugestellt werden, wenn der Endpoint für den Subscriber für die entsprechende Nachricht beim Eintreffen nicht besteht. (3) Nachrichten können dem Empfänger nicht zugestellt werden, wenn der Subscriber dem Publisher noch nicht bekannt ist bzw. der Verbindungsaufbau nicht abgeschlossen ist. Im Folgenden werden die Nachrichtenverluste aus der Perspektive des Komponentendiagramms der Middleware und der des Sequenzdiagramms der Dienstgüteverhandlung begründet. In Abbildung 5.1 ist das Komponentendiagramm der Middleware zu sehen. Dort sind die zuvor drei genannten Fälle mit (1),(2) und (3) an den entsprechenden Übergängen der Komponenten für potenzielle Nachrichtenverluste gekennzeichnet. Die Anwendung auf der Seite des Publishers übergibt die Nachricht ohne Kenntnis über die Existenz seines Endpoints an seinen Connector. Der Fall (1) tritt ein, wenn der Connector keinen Endpoint findet, an den er die Nachricht übergeben kann. Die Nachricht wird somit im Connector verworfen. Andererseits wird die Nachricht bei Existenz eines Endpoints an ihn übergeben. Jedoch kann bei Existenz eines Endpoints der Fall (2) eintreten, wenn die Nachricht vom Endpoint des Publishers aus gesendet wird, aber kein Endpoint eines Subscribers bekannt ist. Diese Nachricht wird somit im

Endpoint verworfen. Im Fall (3) bestehen Endpoints auf der Seite vom Publisher und Subscriber. Jedoch geht hier die Nachricht auf der Protokollebene verloren, wenn der Verbindungsaufbau noch nicht abgeschlossen ist. Nun werden die Fälle mit Blick auf das Sequenzdiagramm der Dienstgüteverhandlung erklärt. In Abbildung 5.2 ist das Sequenzdiagramm der Dienstgüteverhandlung abgebildet. Hier sind die drei Fälle (1),(2) und (3) eingezeichnet, in denen Nachrichten gesendet werden und verloren gehen. Im ersten Fall (1) sendet die ECU die Nachricht vor dem *establish*. Vor dem *establish* ist der Endpoint des Publishers nicht existent, wodurch die Nachricht verworfen wird. Der zweite Fall (2) tritt dann ein, wenn die ECU die Nachricht vor dem *finalize* sendet. Der Endpoint des Subscribers wird jedoch erst nach dem *finalize* erstellt, weshalb die Nachricht nicht weitergegeben werden kann und damit verworfen wird. Wie bereits erwähnt liegt im dritten Fall (3) der Nachrichtenverlust auf der Protokollebene vor. Die Nachricht wird vor dem *connection establish* beziehungsweise während eines noch nicht abgeschlossenen Verbindungsaufbaus losgeschickt. In den Tabellen 5.1 und 5.2 sind alle Dienste erwähnt, bei denen Nachrichtenverluste aufgekommen sind. In der Spalte *Service ID* ist die ID eines Dienstes zu sehen. Mit dieser ID wird ein Dienst innerhalb der Middleware identifiziert. Die Spalte *Sender* zeigt die ECU, welche Nachrichten für diesen Dienst produziert. Neben der Bezeichnung der ECU steht die Anzahl an Nachrichten, die sie gesendet hat. Die ECUs, welche die entsprechende Nachricht empfangen, sind in der Spalte *Empfänger* zu sehen. Neben der Bezeichnung der ECU steht die Anzahl an Nachrichten, die sie empfangen hat. Die Begründung für den Nachrichtenverlust wird in der Spalte *Grund* erwähnt. Diese enthält eine kurze Erläuterung und die entsprechende Fallnummer.

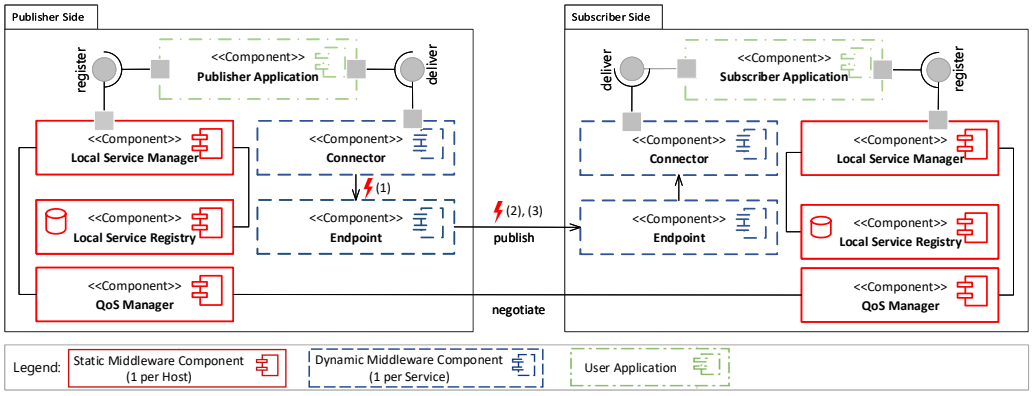

Abbildung 5.1: Komponentendiagramm der Middleware [8]

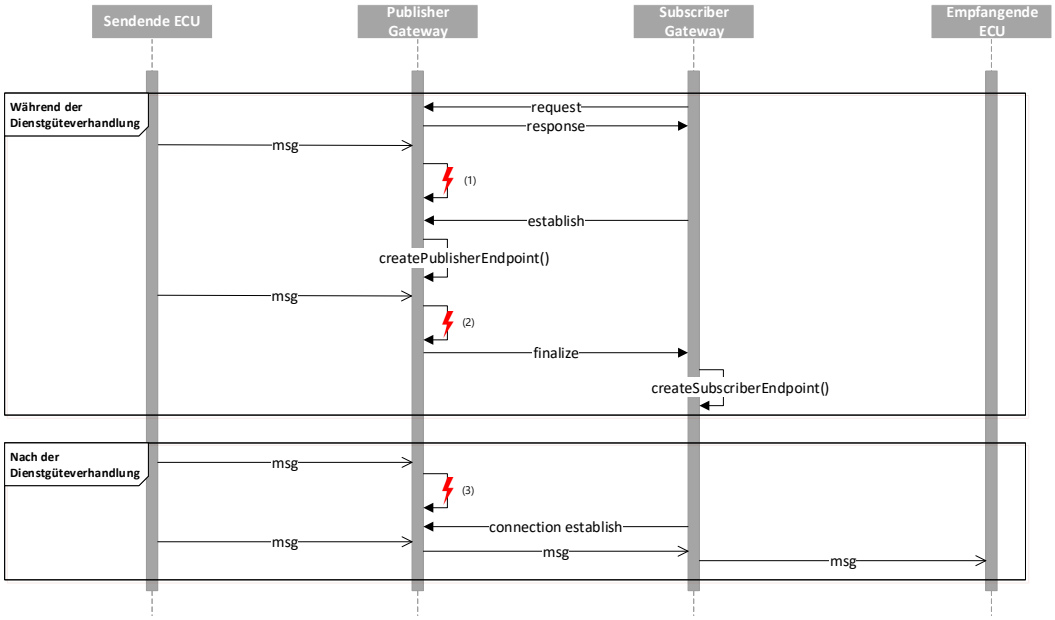

Abbildung 5.2: Sequenzdiagramm der Dienstgüteverhandlung mit potenziellen Stellen für Nachrichtenverluste.

Service ID	**Sender** <ecu:Nachrichtenanzahl>	**Empfänger** <ecu:Nachrichtenanzahl>	**Grund**
480	ecu85:1000	ecu34:1000 ecu39:1000 ecu48:1000 ecu64:1000 ecu67:1000 ecu70:999 ecu76:999 ecu80:1000 ecu84:999 ecu88:999 ecu90:1000	(1) Publisher Endpoint noch nicht erstellt
1097	ecu80:1000	ecu76:999 ecu85:1000 ecu90:1000	(2) Subscriber Endpoint noch nicht erstellt
926	ecu85:1000	ecu67:1000 ecu80:1000 ecu88:999	(2) Subscriber Endpoint noch nicht erstellt
89	ecu3:1000	ecu1:999 ecu2:999	(1) Publisher Endpoint noch nicht erstellt
8	ecu3:1000	ecu1:999 ecu2:999	(3) Verbindungs-aufbau noch nicht abgeschlossen
90	ecu1:1000	ecu3:999	(1) Publisher Endpoint noch nicht erstellt
91	ecu2:1000	ecu3:999	(1) Publisher Endpoint noch nicht erstellt
9	ecu1:1000	ecu3:999	(1) Publisher Endpoint noch nicht erstellt
10	ecu2:1000	ecu3:999	(1) Publisher Endpoint noch nicht erstellt

Tabelle 5.1: Nachrichtenverluste der jeweiligen Service IDs mit Grund (Teil 1)

Service ID	**Sender** <ecu:Nachrichtenanzahl>	**Empfänger** <ecu:Nachrichtenanzahl>	**Grund**
956	ecu70:200	ecu80:200 ecu88:199	(1) Publisher Endpoint noch nicht erstellt
959	ecu70:200	ecu114:199 ecu124:200 ecu125:199 ecu67:199	(3) Verbindungs-aufbau noch nicht abgeschlossen
991	ecu84:100	ecu114:99	(3) Verbindungs-aufbau noch nicht abgeschlossen
88	ecu2:100	ecu3:99	(1) Publisher Endpoint noch nicht erstellt
1	ecu2:100	ecu3:99	(2) Subscriber Endpoint noch nicht erstellt
462	ecu113:100	ecu115:100 ecu48:99	(3) Verbindungs-aufbau noch nicht abgeschlossen
543	ecu97:67	ecu116:67 ecu48:66 ecu65:67 ecu67:67	(3) Verbindungs-aufbau noch nicht abgeschlossen
296	ecu105:100	ecu106:99 ecu114:99 ecu115:100 ecu48:99	(2) Subscriber Endpoint noch nicht erstellt
292	ecu50:10	ecu114:9 ecu96:10 ecu99:10	(1) Publisher Endpoint noch nicht erstellt

Tabelle 5.2: Nachrichtenverluste der jeweiligen Service IDs mit Grund (Teil 2)

6 Evaluation

Dieses Kapitel evaluiert die Middleware. Fast alle ECUs von einem parkenden Auto sind vor dem Gebrauch inaktiv. Sobald der Fahrer das Auto beispielsweise per keyless entry kontaktiert, werden alle ECUs und Dienste Schritt für Schritt aktiviert. Als Erstes müssen die ECUs der Türen oder die der Innenbeleuchtung in kürzester Zeit verfügbar sein. ECUs für die Motorsteuerung müssen nach kürzester Zeit verfügbar sein, nachdem der Fahrer Platz im Auto genommen hat. In aktuellen Autos müssen die ECUs nach etwa 150 ms bis 200 ms bereit sein. Da Seyler u.a. [23] bereits den Einfluss der Servicediscovery evaluiert haben, wird in dieser Arbeit der Einfluss der Dienstgütevereinbarung auf die Setup-Time untersucht. Hierbei steht die Echtzeitfähigkeit im Fokus, welches den kritischsten Aspekt im Automobilbereich bildet. Für die Performance wird somit die Setup-Time gemessen, um die zeitliche Bereitschaft von Diensten festzustellen. Im Wesentlichen besteht die Setup-Time eines Dienstes aus der Zeit für die Dienstgüteverhandlung plus der Zeit für den Verbindungsaufbau zwischen Publisher und Subscriber. Weiter wird gezeigt, wie der Latenzbereich für verschiedene Dienstgüteklassen aussieht. Im Folgenden wird zunächst auf die Metrik eingegangen, die für die Evaluation relevant ist. Danach folgen Evaluationen hinsichtlich des Latenzverhaltens und der Setup-Time in simplen Netzwerken. Abschließend wird die Middleware im realistischen Autonetzwerk in einer Zonalarchitektur, welche bereits in Kapitel 3.6 vorgestellt wurde, evaluiert.

6.1 Metrik

OMNeT++ erstellt für jede abgeschlossene Simulation eine Protokolldatei mit Simulationsergebnissen. In der Protokolldatei werden mithilfe von selbst angelegten Signalen im Simulationscode Statistiken erfasst. Die Evaluation fokussiert (1) das Latenzverhalten für verschiedene Dienstgüteklassen und (2) den Einfluss der Dienstgüteverhandlung auf die Setup-Time. In beiden Punkten wurde die Latenz als Metrik gewählt. (1) Bei

dem Latenzverhalten für verschiedene Dienstgüteklassen wird die zur Verfügung stehende Bandbreite des Netzwerks bis zu 95 % mit Hintergrundverkehr belastet. Hier wird die Ende-zu-Ende-Latenz von dem IPS- und RTS-Subscriber betrachtet. Insbesondere wird das Latenzlimit vom RTS-Subscriber beobachtet, welche das AVB-Protokoll nutzt. Dies soll Aufschluss darüber geben, ob die Middleware mithilfe des AVB-Protokolls zeitliche Garantien zusichern kann. (2) Der Einfluss der Dienstgüteverhandlung wird dabei in zwei verschiedenen Netzwerken betrachtet. Dabei wird das Latenzverhalten der Setup-Time bei steigender Anzahl von Subscriber-Knoten betrachtet. Das zweite Netzwerk ist das realistische Autonetzwerk, welches in Kapitel 3.6 vorgestellt wurde. Das Autonetzwerk wird mit ansteigendem Hintergrundverkehr belastet. Auch hier wird das Latenzverhalten der Setup-Time unter Einfluss von Hintergrundverkehr betrachtet. Bei beiden Netzwerken soll deutlich werden, ob die Setup-Time dabei zu einem linearen oder exponentiellen Anstieg tendiert.

6.2 Evaluation in simplen Netzwerken

Zunächst wird die Dauer der Dienstgüteverhandlung für die einzelnen Dienstgüteklassen gemessen. Dafür wird das Netzwerk aus Abbildung 6.1 genutzt, wo der Hintergrundverkehr ausgeschaltet ist. Desweiteren ist für jede einzelne Dienstgüteklasse nur der entsprechende Knoten aktiv eingesetzt. Alle anderen sind inaktiv. Beim IPS-Subscriber ist einmal mit TCP und UDP jeweils separat gemessen worden. Mit dem RTS-Subscriber wird mit AVB gemessen. Tabelle 6.1 zeigt die reine Verhandlungsdauer und den Zeitpunkt, wann der Verbindungsaufbau abgeschlossen wird. Für alle Netzwerkprotokolle liegt die reine Verhandlungsdauer bei etwa 76 µs. Die reine Verhandlungsdauer wird aus der Differenz von Anfangs- und Endzeitpunkt der Dienstgüteverhandlung bestimmt, da nicht vorausgesagt werden kann, ob sie bei Sekunde null beginnt. Es wird mit statischen MAC-Adressen gearbeitet, sodass kein ARP beim Starten des Netzwerks stattgefunden hat. Der Abschluss des Verbindungsaufbaus wird absolut mit Beginn der Simulation erfasst, ohne dass die Dauer von ARP reingerechnet wird. Für IPS-Dienste ist der Verbindungsaufbau mit TCP nach 130 µs abgeschlossen, während mit UDP der Verbindungsaufbau nach 60 µs abgeschlossen ist. Die Ursache warum der Verbindungsaufbau bei UDP früher als die Dienstgüteverhandlung abschließt liegt darin, dass die Endpoints auf Publisher- und Subscriber-Seite sich kennen, bevor die Dienstgüteverhandlung abschließt. Die Erklärung dafür liefert Kapitel 4.1. Für den RTS-Dienst mit AVB besteht

die Verbindung nach 100 µs. Die verschiedenen Zeiten für TCP und AVB ergeben sich aus dem unterschiedlichen Overhead zum Verbindungsaufbau.

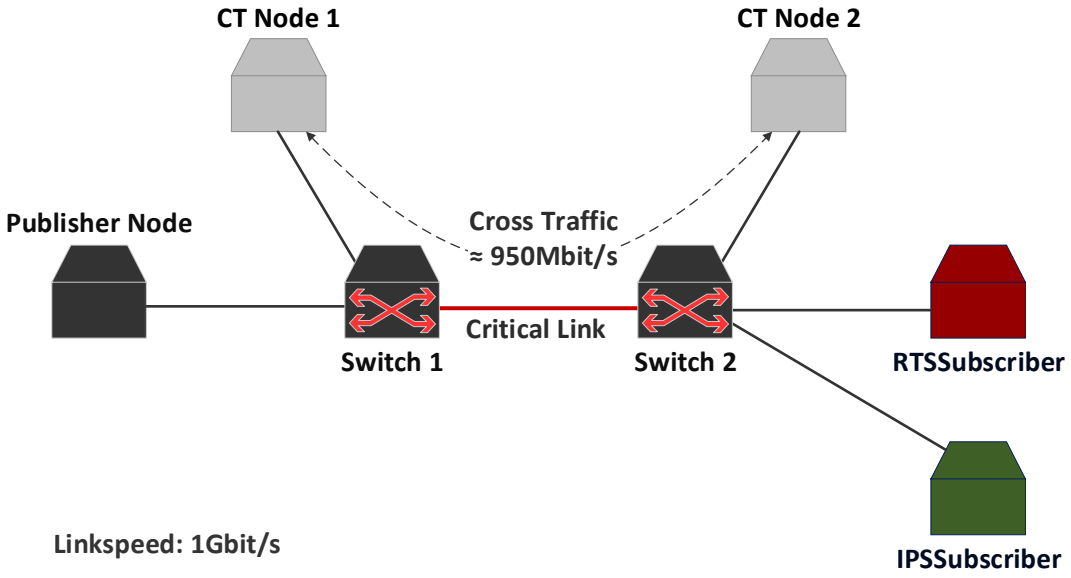

Abbildung 6.1: Simples Netzwerk mit Publisher, zwei Subscribern und Hintergrundverkehr

	Reine Verhandlungsdauer	**Zeitpunkt des Verbindungsabschlusses**
UDP	75,824 µs	56,92 µs
TCP	75,824 µs	129,128 µs
AVB	76,496 µs	94,324 µs

Tabelle 6.1: Reine Verhandlungsdauer und endgültige Setup-Time verschiedener Dienstgüteklassen

Abbildung 6.2 zeigt das Verhalten der Latenz vom RTS- und IPS-Subscriber mit Hintergrundverkehr. Die X-Achse zeigt die Simulationszeit in Sekunden, während die Y-Achse die Ende-zu-Ende Latenz zwischen dem Publisher und dem jeweiligen Subscriber in Mikrosekunden zeigt. Hierbei wurde das Netzwerk in Abbildung 6.1 eingesetzt. Bei der Latenz wurde die Ende-zu-Ende Latenz zwischen Publisher und jeweiligem Subscriber gemessen. $L_{AVB_{max}}$ ist das Latenzlimit, welche AVB für die Topologie des simplen Netzwerks benötigen darf. Das Netzwerk hat eine Bandbreite von 1 Gbit/s. Die Knoten für den Hintergrundverkehr senden sich gegenseitig Daten der Größe 1500 B in einer Normalverteilung zwischen $\approx$ 2041 ns und 23 408 ns zu. Der komplette Ethernetframe beträgt 1530 B. Dies sorgt laut den Simulationsergebnissen auf dem kritischen Link für

eine benötigte Bandbreite von $\approx 950\,\text{Mbit/s}$. Die Hardwareverzögerung der Switche beträgt $t_{Switchdelay} = 8\,\mu\text{s}$ und die Verarbeitungszeit der Publisher sowie Subscriber beträgt $t_{ProcessingOfNode} = 20\,\text{ns}$. Da nur ein Link von dem Hintergrundverkehr betroffen ist, darf die maximale Latenz für RTS-Dienste mit AVB nicht $L_{AVB_{max}} = 30{,}2\,\mu\text{s}$ überschreiten. Für die maximale Latenz $L_{AVB_{max}}$ wird die folgende Berechnung herangezogen.

$$L_{AVB_{max}} = t_{MTU} + 3 \cdot t_{AVBFrame} + 2 \cdot t_{Switchdelay} + IPG + 2 \cdot t_{ProcessingOfNode}$$

Die Übertragungsdauer lässt sich mit $t = \frac{Datenvolumen}{Bandbreite}$ berechnen. Daraus berechnen sich folgende Übertragungsdauern

$$t_{MTU} = \frac{1530\,\text{B}}{\frac{10^9}{8}\,\text{B/s}} = 12{,}240\,\mu\text{s}$$

$$t_{AVBFrame} = \frac{76\,\text{B}}{\frac{10^9}{8}\,\text{B/s}} = 0{,}608\,\mu\text{s}$$

Setzt man die Werte ein, folgt

$$L_{AVB_{max}} = 12{,}240\,\mu\text{s} + 3 \cdot 0{,}608\,\mu\text{s} + 2 \cdot 8\,\mu\text{s} + 0{,}096\,\mu\text{s} + 2 \cdot 0{,}020\,\mu\text{s} = 30{,}2\,\mu\text{s}$$

In einer Simulation mit einer Laufzeit von 20 s wurde eine maximale Latenz von 93 µs für den IPS-Dienst und 30 µs für den RTS-Dienst erfasst. Dies zeigt, dass beim parallelen Verwenden von unterschiedlichen Dienstgüteklassen das Latenzlimit $L_{AVB_{max}}$ für RTS-Dienste nicht überschritten wird.

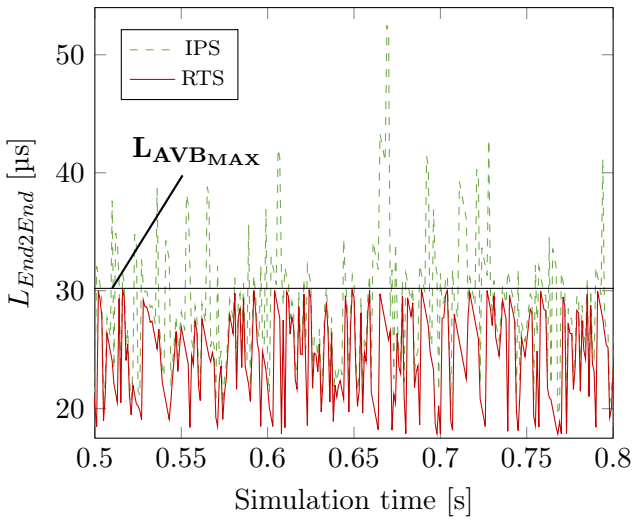

Abbildung 6.2: Ende-zu-Ende Latenzanalyse mit verschiedenen Dienstgüteklassen

Abbildung 6.4 vergleicht die Setup-Time von verschiedenen Mengen an Simulationen. Die X-Achse zeigt die Anzahl der Subscriber-Knoten, während die Y-Achse die Setup-Time in Mikrosekunden zeigt. Insgesamt werden zehn Mengen an Simulationen ausgeführt. In jeder Menge werden zehn Simulationen ausgeführt. Bei der ersten Menge wird mit einem Dienst auf dem Publisher-Knoten begonnen. Die Anzahl der Dienste wird mit jeder Menge in Einerschritten erhöht. Bei der letzten Menge wird somit mit zehn Diensten auf einem Publisher-Knoten simuliert. Für jede erste Simulation in einer Menge wird mit einem Subscriber-Knoten begonnen. Mit den nachfolgenden Simulationen steigt die Anzahl an Subscriber-Knoten in Einerschritten, sodass in jeder letzten Simulation mit zehn Subscriber-Knoten simuliert wird. Die unterste Linie in Abbildung 6.4 repräsentiert die erste Menge. Bei sechs Subscriber-Knoten und einem Dienst erfolgen somit sechs Dienstgüteverhandlungen. Die oberste Linie repräsentiert die letzte Menge. Hier sind es bei sechs Subscriber-Knoten und zehn Diensten 60 Dienstgüteverhandlungen. Alle Subscriber-Knoten beginnen zur selben Zeit mit ihrer Dienstgüteverhandlung. Für jede Simulation wird die letzte abgeschlossene Dienstgüteverhandlung erfasst. Die Abbildung zeigt, dass die Setup-Time mit der Anzahl von Dienstgüteverhandlungen im Netzwerk (über-)proportional ansteigt. Die horizontale Linie bei der 100 µs Marke der Setup-Time weist auf eine Änderung im linearen Verhalten bei nahezu 40 Dienstgüteverhandlungen hin. Das Netzwerk wird an diesem Punkt überlastet und der Stau im Datenverkehr verzögert die Verhandlungen. Diese Situation wird in realen Autonetzwerken in solch einem Ausmaß nicht stattfinden, da die Dienste dort schrittweise gestartet werden.

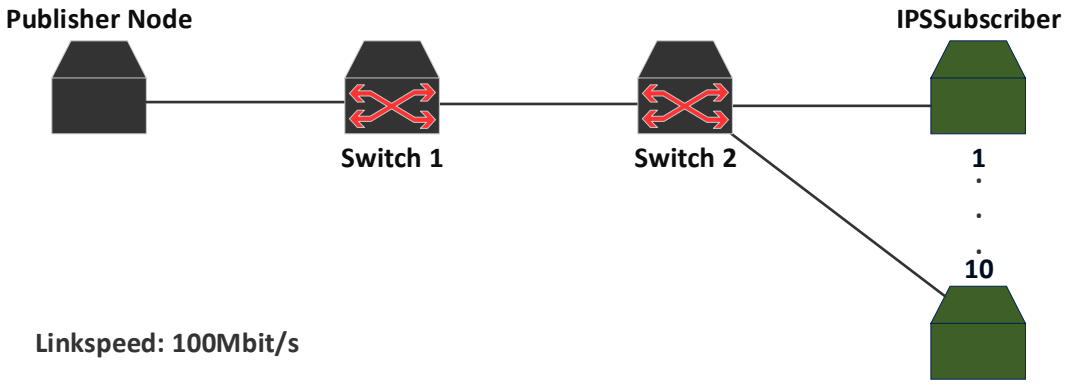

Abbildung 6.3: Simples Netzwerk mit Publisher und variierenden Subscribern

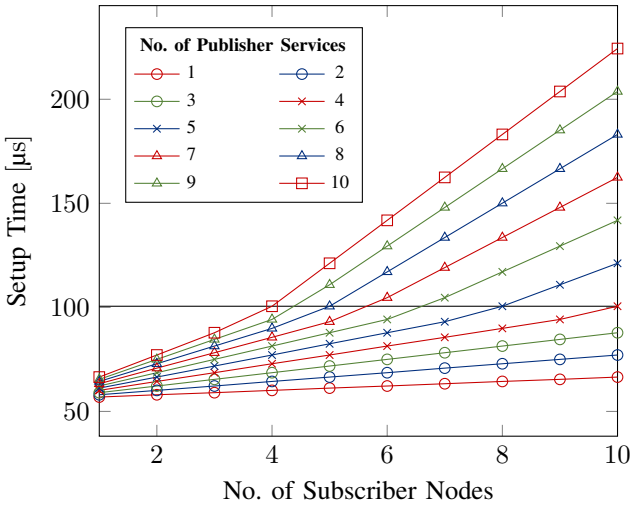

Abbildung 6.4: Setup-Time mit steigender Anzahl von Subscriber-Knoten sowie Diensten.

6.3 Evaluation im realistischen Automobilnetzwerk

Für realitätsnahe Ergebisse wurde die Middleware in einem realistischen Automobilnetzwerk simuliert, welches in Abbildung 6.5 zu sehen ist. Dieses Netzwerk ist eine Variation eines domänenbasierten Netzwerks. Die Variante bei dieser Evaluation ist zonenbasiert. Der Unterschied zwischen der domänen- und zonenbasierten Variante wird in Kapitel 3.6 detaillierter erläutert. Die ECUs generieren CAN-Nachrichten, welche über das entsprechende Gateway in der Zone in das Ethernet Backbone ausgesendet werden. Diese ECUs werden im Gateway als Publisher abstrahiert und stellen für jede CAN-Nachricht einen Dienst zur Verfügung. ECUs, welche solch eine dienstbasierte CAN-Nachricht empfangen, werden an ihrem angeschlossenen Gateway als Subscriber abstrahiert. Jedes Gateway abonniert somit den entsprechenden Dienst, welcher an einem anderen Gateway angeboten wird. Abbildung 6.6 bildet das Verhalten der Setup-Time mit ansteigendem Hintergrundverkehr ab. An der Y-Achse ist die Setup-Time in ms angegeben. Jede Leitung im Netzwerk wird mit einem Datenvolumen von 0 Mbit bis 1000 Mbit in beide Richtungen belastet. Entlang der X-Achse ist der Hintergrundverkehr in Mbit/s angegeben. Für jede Simulation wurde das Minimum, der Durchschnitt und das Maximum der Setup-Time erfasst. Das Minimum ist die Setup-Time aus der ersten abgeschlossenen Dienstgüteverhandlung plus dem Verbindungsaufbau. Der Durchschnitt ist die Setup-Time aller abgeschlossenen Dienstgüteverhandlungen plus dem Verbindungsaufbau im

Mittel. Das Maximum ist die Setup-Time der zuletzt abgeschlossenen Dienstgüteverhandlung plus dem Verbindungsaufbau. Bei einem Hintergrundverkehr von 300 Mbit/s beträgt die Setup-Time um die 1 ms. Im Normalfall gibt es in Autonetzwerken keinen Hintergrundverkehr, wodurch 1 ms für die Setup-Time ein praxisnahes Ergebnis darstellt. Ab einem Hintergrundverkehr von 700 Mbit/s steigt Setup-Time exponentiell an. Somit staut sich der Datenverkehr, wodurch Dienstgüteverhandlungen nicht mehr rechtzeitig abgeschlossen werden könnten. Hintergrundverkehr findet in einem Autonetzwerk in der Regel nicht statt. Der Hintergrundverkehr in dieser Simulation wird künstlich erzeugt. Jedoch erschließt sich daraus, dass hohe Verbindungslasten ab einem bestimmten Schwellenwert zu Nachrichtenverlusten führen können, wodurch Dienstgüteverhandlungen nicht abgeschlossen würden. Das bedeutet für sicherheitskritische Automobildienste, wo eine Verzögerung in jedem Fall zu vermeiden ist, diese statisch zu konfigurieren und als TDMA-Verkehr zu betreiben. Trotzdem halten die Ergebnisse in jedem Fall die ≈ 150 ms bis 200 ms deutlich ein, welche am Anfang von Kapitel 6 diskutiert wurden.

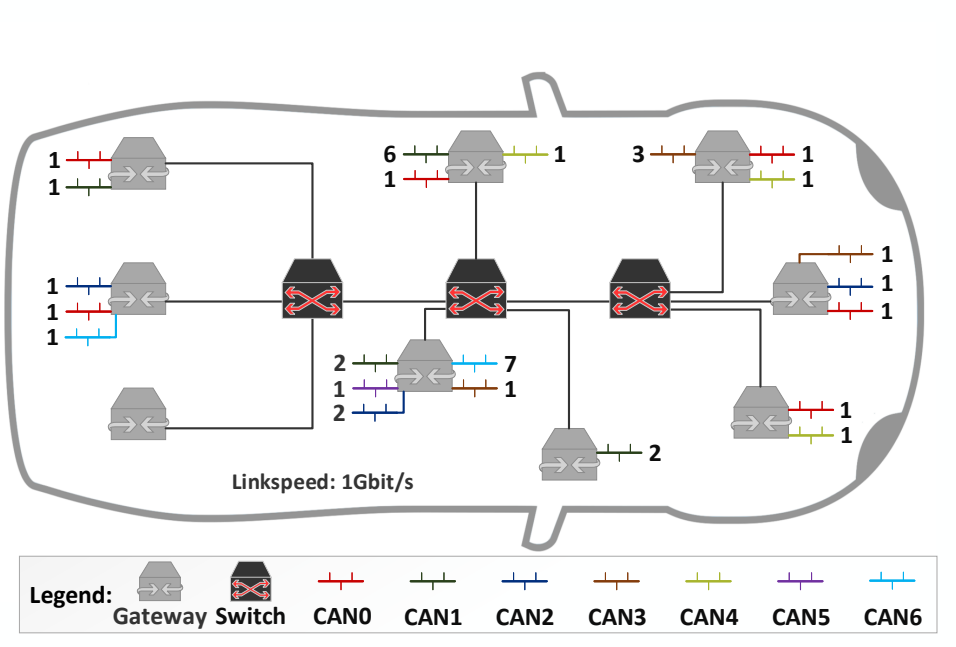

Abbildung 6.5: Realistisches Automobilnetzwerk mit realer Fahrzeugkommunikation

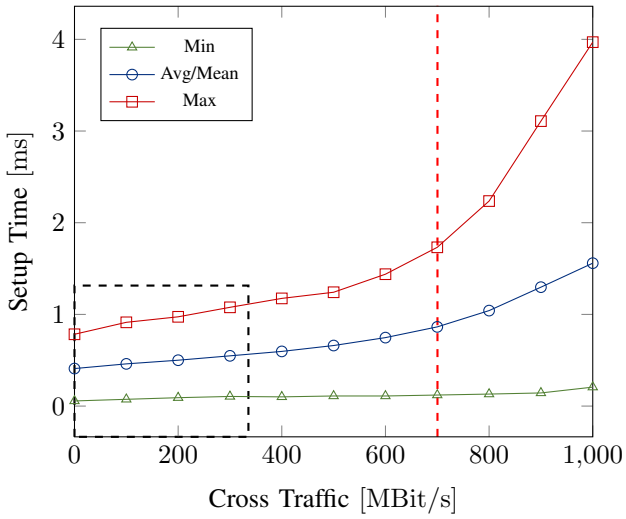

Abbildung 6.6: Minimum, Maximum und Durchschnitt der Setup-Time im realistischen Autonetzwerk

7 Fazit und Ausblick

In dieser Arbeit wurde eine dienstbasierte Middleware für Autonetzwerke in der Simulationsumgebung OMNeT++ evaluiert, welche Häckel [13] im Rahmen einer Masterarbeit konzipiert hat. Mit einer dienstorientierten Architektur unterstützt die Middleware lose Kopplung und Abgeschlossenheit in ihren Komponenten. Dies bringt hohe Erweiterbarkeit und Übersichtlichkeit. Mit dem Protokoll zur Dienstgüteverhandlung wurde gezeigt, wie die Dienstgüte dynamisch zur Laufzeit ausgehandelt werden kann. Mithilfe der Dienstgüteverhandlung und der dienstbasierten Architektur werden Dienste über die Middleware selbständig gefunden und der Verbindungsaufbau übernommen. Für Dienste, wo eine statische Konfiguration entfallen darf, minimiert sich dadurch der Konfigurationsaufwand. Anhand der Klassifizierung von automobilen Diensten, wurden Dienstgüteklassen definiert, um heterogene Netzwerkkommunikation in Autonetzwerken abzudecken. Die Netzwerkkommunikation wird mit einem Multi-Protokollstack realisiert, mit dem Best-Effort- und Echtzeitdatenverkehr ausgewählt wird. Für die Evaluation wurde ein realistisches Autonetzwerk eingesetzt. Es basiert auf einem Autonetzwerk, welches eine domänenbasierte Architektur aufweist und CAN-basiert ist. Dieses wurde von der CoRE-Arbeitsgruppe [9] in eine zonenbasierte Architektur mit einem Ethernet Backbone umgewandelt. Der Vorteil einer Zonalarchitektur ist seine Modularität sowie gute Erweiterbarkeit und der Geschwindigkeitszuwachs durch das Ethernet Backbone. Die UDP-Kommunikation ist im Multi-Protkollstack vor dieser Arbeit nicht verfügbar gewesen und wurde nachträglich implementiert. Als Folge musste das Protokoll zur Dienstgüteverhandlung erweitert werden, um auch mit verbindungslosen Protokollen umgehen zu können. Mithilfe einer sogenannten Abstract Network Description Language (ANDL) ist es möglich mit einer kompakten Syntax Netzwerke zu beschreiben und für OMNeT++ zu generieren. Vor dieser Arbeit ist die Beschreibung von dienstbasierten Netzwerken nicht möglich gewesen. Die Unterstützung zum Beschreiben von dienstbasierten Netzwerken wurde nachträglich implementiert. Eine Qualitätssicherung hat gezeigt, dass die Nachrichtenverluste durch die Dienstgütervehandlung unter 1 ms geblieben sind. Da die vorausgesetzte Bereitschaft für Dienste in Autonetzwerken zwischen 150 ms und 200 ms

liegt, sind die Nachrichtenverluste deutlich unter diesem Zeitfenster. Außerdem beläuft sich der Nachrichtenverlust auf maximal eine Nachricht pro Dienst. Die Evaluation der Middleware zeigt, dass bei paralleler Nutzung von einem IPS- und RTS-Dienst mit hohem Hintergrundverkehr Echtzeitgarantien eingehalten werden. Somit werden Anforderungen in heterogener Netzwerkkommunikation erfüllt. Messungen zur Setup-Time zeigen in einem Netzwerk, wo die Anzahl der Dienste und Subscriber variieren, einen linearen Verlauf. Im realistischen Autonetzwerk tendiert die Setup-Time zu einem exponentiellen Anstieg mit gleichzeitig ansteigendem Hintergrundverkehr. In beiden Fällen werden auch hier die Zeiten von 150 ms bis 200 ms eingehalten, in denen ein Automobildienst spätestens zur Verfügung stehen muss. Es wurden somit akzeptable Setup-Times in einem realistischen Autonetzwerk erzielt, womit die Middleware sich bisher in einer Simulationsumgebung als vielversprechend erweist. Als weitere Arbeit steht die Erfassung von realitätsnahen Ergebnissen offen. Die Implementierung von SOME/IP im Protokollstack der Middleware ist eine Möglichkeit die Praxistauglichkeit zu erhöhen. So kann die Middleware auf realer Hardware in derzeitigen Autonetzwerken, wo bereits SOME/IP läuft, mit geringem Aufwand integriert werden. Anschließend ist mit der Erfassung von realen Laufzeiten die Eignung der Middleware für den Einsatz auf dem Markt bewertbar. Außerdem ist die Implementierung der Middleware als Data Distribution Service (DDS) [18] von der Object Management Group (OMG) [19] denkbar. Ein Vergleich zwischen der Middleware mit SOME/IP und der Middleware als DDS würde Aufschluss über die Stärken und Schwächen der beiden Varianten untereinander geben.

Literaturverzeichnis

[1] IEEE Standard for Ethernet. In: *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)* (2016), March, S. 1–4017

[2] 802.1Q, IEEE: IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks–Corrigendum 1: Technical and editorial corrections. In: *IEEE Std 802.1Q-2014/Cor 1-2015 (Corrigendum to IEEE Std 802.1Q-2014)* (2016), Jan, S. 1–122

[3] 802.1Qav, IEEE: IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. In: *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)* (2010), Jan, S. C1–72

[4] 802.1Qbv, IEEE: IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic. In: *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)* (2016), March, S. 1–57

[5] AUTOSAR Development Cooperation: *AUTomotive Open System ARchitecture*. – URL http://www.autosar.org

[6] Broy, Manfred: Challenges in Automotive Software Engineering. In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA : ACM, 2006 (ICSE '06), S. 33–42. – ISBN 1-59593-375-1

[7] Buckl, C. ; Camek, A. ; Kainz, G. ; Simon, C. ; Mercep, L. ; Stähle, H. ; Knoll, A.: The Software Car: Building ICT Architectures for Future Electric Vehicles. In: *2012 IEEE International Electric Vehicle Conference*, März 2012, S. 1–8

[8] Cakir, Mehmet ; Häckel, Timo ; Reider, Sandra ; Meyer, Philipp ; Korf, Franz ; Schmidt, Thomas C.: A QoS Aware Approach to Service-Oriented Communication in Future Automotive Networks. In: *2019 IEEE Vehicular Networking Conference (VNC)*, URL https://arxiv.org/abs/1911.01805, Dezember 2019

[9] CoRE-Arbeitsgruppe: *Communication over Real-time Ethernet.* – URL https://core-researchgroup.de/. – Zugriffsdatum: 2018-06-29

[10] CoRE-Arbeitsgruppe: *CoRE Simulation Models for Real-time Networks.* – URL http://sim.core-rg.de. – Zugriffsdatum: 2018-06-29

[11] Eclipse Foundation Inc: *Xtend.* – URL https://www.eclipse.org/Xtext/index.html. – Zugriffsdatum: 2019-12-20

[12] fortiss GmbH: The Software Car: Information and Communication Technology (ICT) as an Engine for the Electromobility of the Future / fortiss GmbH. März 2011. – Forschungsbericht. summary of results of the ëCar ICT System Architecture for Electromobility"research project sponsored by the Federal Ministry of Economics and Technology

[13] Häckel, Timo: *Automobile Kommunikationsarchitekturen zur Unterstützung von Dienstgütevereinbarungen.* Hamburg, Hochschule für Angewandte Wissenschaften Hamburg, mastersthesis, Juni 2018

[14] IEEE: *2019 IEEE Vehicular Networking Conference (VNC).* – URL http://www.ieee-vnc.org/. – Zugriffsdatum: 2019-12-20

[15] IEEE 802.1 TSN Task Group: *IEEE 802.1 Time-Sensitive Networking Task Group.* – URL https://1.ieee802.org/tsn/. – Zugriffsdatum: 2020-01-12

[16] Meyer, Philipp: *Informationssicherheit für Echtzeit-Ethernet-Fahrzeugnetzwerke.* Hamburg, Hochschule für Angewandte Wissenschaften Hamburg, mastersthesis, Juni 2018

[17] Meyer, Philipp ; Korf, Franz ; Steinbach, Till ; Schmidt, Thomas C.: Simulation of Mixed Critical In-vehicular Networks. In: *Recent Advances in Network Simulation.* Springer, 2019, S. 317–345. – URL https://link.springer.com/chapter/10.1007/978-3-030-12842-5_10

[18] Object Management Group: *Data Distribution Service.* – URL https://www.dds-foundation.org/. – Zugriffsdatum: 2020-01-16

[19] Object Management Group: *Object Management Group*. – URL http://www.omg.org/. – Zugriffsdatum: 2020-01-16

[20] OpenSim Ltd.: *INET Framework*. – URL https://inet.omnetpp.org/

[21] OpenSim Ltd.: *OMNeT++ Discrete Event Simulator*. – URL https://omnetpp.org/

[22] PlattformAuto2018: *Automobil - Wie verändern digitale Plattformen die Automobilwirtschaft?*. – URL http://plattform-maerkte.de/auto/. – Zugriffsdatum: 2018-04-10

[23] Seyler, Jan R. ; Streichert, Thilo ; Glass, Michael ; Navet, Nicolas ; Teich, Jürgen: Formal Analysis of the Startup Delay of SOME/IP Service Discovery. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. San Jose, CA, USA : EDA Consortium, 2015 (DATE '15), S. 49–54. – URL http://dl.acm.org/citation.cfm?id=2755753.2755765. – ISBN 978-3-9815370-4-8

[24] Society of Automotive Engineers - AS-2D Time Triggered Systems and Architecture Committee: *Time-Triggered Ethernet AS6802*. SAE Aerospace. November 2011. – URL http://standards.sae.org/as6802/

[25] Steinbach, Till: *Ethernet-basierte Fahrzeugnetzwerkarchitekturen für zukünftige Echtzeitsysteme im Automobil*. Wiesbaden : Springer Vieweg, Oktober 2018. – ISBN 978-3-658-23499-7

[26] Till Rausch: *Service Orientierte Architektur Übersicht und Einordnung*. – URL https://web.archive.org/web/20081010033719/http://www.till-rausch.de/assets/baxml/soa_akt.pdf. – Zugriffsdatum: 2020-01-15

[27] Zimmermann, Werner ; Schmidgall, Ralf: *Bussysteme in der Fahrzeugtechnik*. 5. Wiesbaden : Vieweg, 2014 (ATZ/MTZ-Fachbuch). – ISBN 978-3-658-02418-5

A Anhang

A.1 Nachrichtenverluste

Die nachfolgenden Tabellen zeigen die Nachrichtenverluste während der Setup-Time:

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu85	1000	FrontLeft	/	[29]:422 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu34	1000	FrontLeft	/	/	/	/
ecu39	1000	FrontLeft	/	/	/	/
ecu48	1000	FrontLeft	/	/	/	/
ecu64	1000	Front	Static	/	/	/
ecu67	1000	Center	Static	/	/	/
ecu70	999	Rear	RT	[14]:760 µs	/	/
ecu76	999	Rear	RT	[14]:760 µs	/	/
ecu80	1000	FrontRight	Static	/	/	/
ecu84	999	RearLeft	RT	[24]:774 µs	/	/
ecu88	999	CenterLeft	RT	[45]:678 µs	/	/
ecu90	1000	Front	Static	/	/	/

Tabelle A.1: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 21 bzw. Service-ID 480

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu80	1000	FrontRight	/	[0]:50 µs	/	224 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu76	999	Rear	RT	[21]:770 µs	nicht da	/
ecu85	1000	FrontLeft	Static	/	/	/
ecu90	1000	Front	RT	[0]:61 µs	nicht da	/

Tabelle A.2: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 22 bzw. Service-ID 1097

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu85	1000	FrontLeft	/	[30] : 424 µs	/	446 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu67	1000	Center	Static	/	/	/
ecu80	1000	FrontRight	Static	/	/	/
ecu88	999	CenterLeft	RT	[46]:679 µs	nicht da	/

Tabelle A.3: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 30 bzw. Service-ID 926

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	1000	CenterLeft	/	[60]:619 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu1	999	FrontLeft	RT	[11]:654 µs	/	/
ecu2	999	FrontRight	RT	[2]:655 µs	/	/

Tabelle A.4: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 109 bzw. Service-ID 89

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	1000	CenterLeft	/	[61]:634 µs	/	668 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu1	999	FrontLeft	RT	[11]:654 µs	Remote at:774 µs	/
ecu2	999	FrontRight	RT	[2]:655 µs	Remote at:775 µs	/

Tabelle A.5: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 111 bzw. Service-ID 8

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu1	1000	FrontLeft	/	[17]:398 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	999	CenterLeft	UDP	[26]:314 µs	/	/

Tabelle A.6: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 115 bzw. Service-ID 90

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu2	1000	FrontRight	/	[7]:511 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	999	CenterLeft	UDP	[38]:492 µs	/	/

Tabelle A.7: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 117 bzw. Service-ID 91

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu1	1000	FrontLeft	/	[18]:399 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	999	CenterLeft	UDP	[27]:315 µs	/	/

Tabelle A.8: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 119 bzw. Service-ID 9

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu2	1000	FrontRight	/	[8]:513 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	999	CenterLeft	UDP	[39]:494 µs	/	/

Tabelle A.9: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 121 bzw. Service-ID 10

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu70	200	Rear	/	[2]:340 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu80	200	FrontRight	Static	/	/	/
ecu88	199	CenterLeft	RT	[36]:381 µs	/	/

Tabelle A.10: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 225 bzw. Service-ID 956

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu70	200	Rear	/	[0]:149 µs, [3]:341 µs	/	446 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu114	199	CenterLeft	RT	[37]:383 µs	Remote at: 526 µs	/
ecu124	200	CenterRight	UDP	[5]:126 µs	/	/
ecu125	199	CenterLeft	RT	[37]:383 µs	Remote at: 526 µs	/
ecu67	200	Center	Static	/	/	/

Tabelle A.11: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 285 bzw. Service-ID 959

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu84	100	RearLeft	/	[3]:339 µs	/	446 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu114	99	CenterLeft	RT	[35]:380 µs	Remote at: 525 µs	/

Tabelle A.12: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 331 bzw. Service-ID 991

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu2	100	FrontRight	/	[9]:515 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	99	CenterLeft	UDP	[40]:496 µs	/	/

Tabelle A.13: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 334 bzw. Service-ID 88

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu2	100	FrontRight	/	[10]:517 µs	/	650 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu3	99	CenterLeft	RT	[70]:738 µs	nicht da	/

Tabelle A.14: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 336 bzw. Service-ID 1

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu113	100	Center	/	[0]:111 µs	/	224 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu115	100	CenterRight	UDP	[0]:101 µs	/	/
ecu48	99	FrontLeft	UDP	[3]:172 µs	Remote at: 558 µs	/

Tabelle A.15: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 349 bzw. Service-ID 462

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu97	67	Center	/	[24]:572 µs	/	668 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu116	67	Center	/	/	/	/
ecu48	66	FrontLeft	RT	[6]:591 µs	Remote at: 750 µs	/
ecu65	67	Center	/	/	/	/
ecu67	67	Center	/	/	/	/

Tabelle A.16: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 377 bzw. Service-ID 543

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu105	100	RearLeft	/	[0]:145 µs	/	224 µs
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu106	99	CenterLeft	RT	[33]:367 µs	nicht da	/
ecu114	99	CenterLeft	RT	[33]:367 µs	nicht da	/
ecu115	100	CenterRight	RT	[12]:172 µs	/	/
ecu48	99	FrontLeft	RT	[17]:715 µs	nicht da	/

Tabelle A.17: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 490 bzw. Service-ID 296

Sender	**Gesendet**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu50	10	Center	/	[23]:257 µs	Dropped	/
Empfänger	**Empfangen**	**Gateway**	**QoS**	$\mathbf{t_{Endpunkt}}$	**Grund**	$\mathbf{t_{Ankunft}}$
ecu114	9	CenterLeft	RT	[31]:333 µs	/	/
ecu96	10	Center	/	/	/	/
ecu99	10	Center	/	/	/	/

Tabelle A.18: Sender- und Empfängerinformationen für die CAN-Nachricht mit der ID 501 bzw. Service-ID 292

Glossar

keyless entry (deutsch: schlüsselloser Eintritt) beschreibt ein System, um ein Fahrzeug ohne aktive Benutzung eines Autoschlüssels zu entriegeln.

QoS Negotiation Protocol QoS Negotiation Protocol (QoSNP) (deutsch: Protokoll zur Dienstgütevereinbarung) bezeichnet das Protokoll mit welchem die Middleware anhand der geforderten Dienstgüte die Netzwerkverbindung mit dem entsprechenden Netzwerkprotokoll zur Laufzeit aushandelt und herstellt.

Quality-of-Service Quality-of-Service (QoS) (deutsch: Dienstgüte) bezeichnet die Güte eines Kommunikationsdienstes anhand gewisser Eigenschaften (wie z.B. maximale Latenz) aus Sicht der Anwender.

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:

Evaluation dienstorientierter Kommunikation in automobilen Zonalarchitekturen

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