HAW HAMBURG

BACHELOR THESIS
Jan Schmitt-Solbrig

Unterstützung der SOME/IP Service Discovery durch Software-Defined Networking

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jan Schmitt-Solbrig

Unterstützung der SOME/IP Service Discovery durch Software-Defined Networking

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

Betreuender Prüfer: Prof. Dr. Franz Korf
Zweitgutachter: Prof. Dr. Martin Hübner

Eingereicht am: 07. Dezember 2023

Jan Schmitt-Solbrig

Thema der Arbeit

Unterstützung der SOME/IP Service Discovery durch Software-Defined Networking

Stichworte

SDN, SOME/IP-SD, OpenFlow, Auto, Fahrzeug, Netzwerk

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Einführung einer Service Discovery(SD) als Anwendung des SDN-Controllers, welche SD-Nachrichten durch cachen beantwortet. Diese soll SOME/IP-SD vollständig unterstützen und dabei transparent für bestehende SOME/IP-Implementierungen bleiben. Als Ist-Zustand wird die SOME/IP-SD-Spezifikation untersucht und im Konzept die Anforderungen geklärt. Die Implementierung in der Simulationsumgebung OMNeT++ ermöglicht die Evaluation.

Jan Schmitt-Solbrig

Title of Thesis

Support of the SOME/IP Service Discovery with Software-Defined Networking

Keywords

SDN, SOME/IP-SD, OpenFlow, automotive, vehicle, network

Abstract

This work focuses on the implementation of a Service Discovery (SD) as an application of the SDN controller which responds to SD messages through caching. The goal is to fully support SOME/IP-SD while maintaining transparency to existing SOME/IP implementations. As state of the art the SOME/IP-SD specification is studied and in the concept the requirements are clarified. The implementation in the OMNeT++ simulation environment enables the evaluation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Gas geben, Bremsen, Fahrerassistenzsysteme oder das Infotainmentsystem, alles in einem Auto funktioniert über Daten- oder Informationsaustausch. Mit der Einführung der Scalable Serviceoriented Middleware over IP (SOME/IP) wurde der Ethernet-Standard in Fahrzeugen eingeführt. Steuereinheiten (Electronical Control Units), Sensoren und Aktoren werden um eine Service Discovery (SD) erweitert. Die SD übernimmt die Kommunikation in Form von Dienstangeboten oder -anfragen.

Die Revolution in der Kommunikation von Mikrokontrollern, bzw. electronic control units (ECUs), im Auto: 1986 stellen Bosch und Intel den CAN-Bus(Controller Area Network-Bus) vor, eine Netzwerkkommunikation für ECUs auf der Grundlage zweipoliger Kabel um die Länge verbauter Kabel zu reduzieren. Später erreicht dieses Bussystem Geschwindigkeiten von bis zu 1 MBit/s. [27] Die Idee dahinter: Viele ECUs kommunizieren über ein Kabel, durch Spannungsänderungen werden die einzelnen Bits der Nachrichten von den ECUs an alle anderen an diesem Kabel gesendet. Bei mehreren gleichzeitigen Nachrichten existiert ein Priorisierungssystem. Die Nachricht mit der höchsten Priorität wird garantiert gesendet und es müssen keine Wiederholungen stattfinden. So werden Datenkollisionen im Netzwerk verhindert und die zeitkritische Voraussetzungen der Kommunikation werden erfüllt.

Heute sind über 80 ECUs im Fahrzeug verbaut und manche Systeme (z.B. zur Übertragung von unkomprimierten Videodaten) benötigen einen höheren und sichereren Datendurchsatz. Die in normalen Netzwerken gebräuchliche Technologie Ethernet wurde für den Einsatz auf 2-polige Kabel erweitert. CAN und Ethernet werden parallel eingesetzt, da die CAN-Komponenten für viele Aufgaben noch ausreichend und gleichzeitig kostengünstiger sind. Eine weitere Idee aus der Informatik wurde übernommen um die Koordination der Empfänger und Sender zu vereinfachen: Das publish-subscribe-pattern. Auf der einen Seite gibt es Sensoren die Daten erfassen und veröffentlichen - die Publisher oder Veröffentlicher und auf der anderen Seite Bauteile die diese Daten benötigen - die Subscriber oder Beobachter. Jede Netzwerkkomponente meldet sich bei Systemstart und

gibt an, welche Informationen es bereitstellt und welche Informationen es benötigt. In den Grundlagen wird näher auf das umgesetzte Muster eingegangen.

2005 entwickelte die Stanford University erste Konzepte eines software-defined networking(SDN). [18] Die Idee dahinter ist das Wissen der Topologie des Netzwerkes in einer Komponente zu zentralisieren. Mithilfe dieser Komponente, des SDN-Controllers, wird die Steuerungsebene von der Datenübertragungsebene entkoppelt. Der SDN-Controller definiert die Wege der Daten im Netzwerk.

So können Informationen des CAN-Busses an ein Ethernetgateway gesendet werden, welches die Übersetzung für den anderen Standard übernimmt. Mit anderen Worten: Anstatt jede Nachricht an alle zu senden und nur derjenige, den es interessiert, akzeptiert die Nachricht, werden die Nachrichten gezielt an diejenigen ECUs gesendet, die diese Informationen benötigen und das auf einem bestimmten Weg im Netzwerk. In Kombination mit dem SDN-Controller verfolgt diese Arbeit die Idee, dass alle angebotenen und angefragten Services in einer Service Discovery gespeichert und verarbeitet werden. So kann der Controller direkt auf Anfragen antworten und die Angebote und Anfragen müssen nicht im gesamten Netzwerk verteilt werden.

2 Grundlagen

In diesem Kapitel werden Einblicke in die grundlegenden Technologien aufgeführt, welche in dieser Arbeit kombiniert werden. Zuerst wird SOME/IP als Standard-Middleware zum Informationsaustausch aller Komponenten in heutigen Fahrzeugen vorgestellt. Es folgt die Erklärung was ein Software definiertes Netzwerken (Software-Defined Networking) ist und wie das Bedürfnis nach dieser Technologie aus der sich erhöhenden Komplexität bestehender Netzwerke gewachsen ist. Zuletzt wird das Protokoll OpenFlow als Werkzeug zur Umsetzung der Architektur des Software-Defined Networking präsentiert. OpenFlow wird als bemerkenswertestes Beispiel einer API für den SDN-Controller betitelt. [16]

2.1 Scalable serviceoriented middleware over internet protocol (SOME/IP)

Scalable serviceoriented middleware over internet protocol(SOME/IP) oder wie es in der Spezifikation steht "Yet another RPC-Mechanism". Vorweg werden die namensgebenden Inhalte des Akronyms erklärt:

Over internet protocol: Die Weiterentwicklung des Ethernetstandards auf single twisted pair [10] - ein einzelnes verdrilltes Adernpaar, ermöglichte die Integration von Ethernet in Fahrzeugen. Allerdings führten höhere Anforderungen der Automobilindustrie [2] [6] zu der Notwendigkeit eine neue Spezifikation zu entwickeln. Die Verfasser dieser neuen Spezifikation haben sich daran orientiert, welche Anwendungsfälle im Fahrzeugkontext auftreten und wie diese mit vorhandenen Technologien abgedeckt werden können.

Serviceoriented: Sie identifizierten die Kommunikation der ECUs als Services und entschieden sich für zwei serviceorientierte Technologien. Erstens haben sie den Remote Procedure Call(RPC) gewählt, durch welchen eine Client-ECU benötigte Informationen abruft, indem sie Methoden/Services einer Server-ECU aufruft. Dieser Mechanismus ermöglicht eine gezielte Nutzung oder Aufrufe von serverseitig angebotenen Services.

Zweitens haben die Verfasser das publish-subscribe pattern für Informationen gewählt, welche entweder zyklisch versendet werden oder wenn sich ein Zustand innerhalb einer ECU ändert. Dieser Mechanismus ist für die Echtzeitkommunikation erforderlich.

Scalable: Die Spezifikation ist für so viele Anwendungsfälle und so viele Kommunikationspartner wie möglich ausgelegt, sie soll skalierbar sein von Kleinst- bis Großplattform, auf allen Betriebssystemen der Automobilbranche und auf eingebetteten System ohne Betriebssystem implementiert werden können. [2]

Middleware: Der Programmstack jeder ECU wird um das SOME/IP-Protokoll erweitert. Der PDU-Router arbeitet nun mit dem SOME/IP-Transportprotokoll und die SOME/IP-Funktionalität setzt auf der Transportschicht auf. Deswegen sollen auch TCP und UDP unterstützt werden. [3]

In der SOME/IP Spezifikation werden die Serialisierung der Daten und der Einsatz des Transport Protokolls aufgeschlüsselt. Wichtig für das Verständnis dieser Arbeit sind zum einen welche Arten von Services existieren und zum anderen die Vorgabe des Headers für jede SOME/IP Kommunikation.

Services sind die Bereitstellung jeglicher Daten, bzw eine funktionale Einheit, welche eine Schnittstelle zur Verfügung stellt. [4] Sie werden durch eine Liste ihrer Funktionalitäten definiert, welche der Service anbietet. Die Nachrichten basieren darauf, dass sich ein Abonnent(Subscriber) bereits bei einem Anbieter(Publisher) angemeldet hat und so können Services aus einer Kombination aus Events, Methoden und Feldern bestehen(jeweils null oder mehrere).

Events: Der Anbieter sendet dem Abonnenten zyklisch oder bei Zustandsänderungen Daten.

Methoden: Der Abonnent darf Methodenaufrufe beim Anbieter starten.

Felder: Eine Kombination aus einer oder mehreren notifier-, getter- oder setter-Nachrichten. Notifier: Der aktuelle Änderungswert des Anbieters wird beim Abonnieren gesendet, dieser dient als Grundlage für die Nachrichten aus Events. Getter/setter: Nachrichten um benötigte Werte beim Anbieter auslesen oder ändern zu können.

Für jede ECU wird definiert, welche Services sie zur Verfügung stellt und diese bekommen Identifikationsnummern, die Service IDs. Die gleichen Services werden teils von mehreren ECUs angeboten oder eine ECU stellt mehrere Services für verschiedene

Abnehmer bereit. Um sie eindeutig unterscheiden zu können wurde der Begriff Serviceinstanzen (Service Instance) definiert. Die ECUs die Serviceinstanzen anbieten werden Server genannt und die ECUs die Serviceinstanzen benötigen werden als Client betitelt. ECUs sind hierdurch häufig Client und Server zugleich, da sie häufig Daten benötigen um bestimmte Services anzubieten. Mit der Service ID wurden die Services benannt, aber woher weiß nun eine ECU wo sie eine Service ID 1234 findet oder für welche ECU sie einen Service anbietet? Es gibt spezielle Multicasts welche dem Netzwerk mitteilen, dass eine besimmte Serviceinstanz gesucht wird (Request). Dementsprechend existiert der Multicast auch für Angebote von Serviceinstanzen (Offer). Um das SOME/IP Protokoll zu entlasten wurde für die Erkennung von Services ein explizites neues Protokoll entwickelt - die SOME/IP Service Discovery (SOME/IP-SD). Sie setzt als Modul auf dem SOME/IP-Protokoll auf.

Die Service Discovery übernimmt die Funktionalität verfügbare Services innerhalb des Automotive-Netzwerkes anzubieten und zu finden. Als Umsetzung des Publish-Subscribe-Musters nutzt sie IP-Multicast und definiert eigene SOME/IP-SD Nachrichten. Jede ECU kann mithilfe des Service Discovery Modules Services anbieten, suchen und auch das Angebot des Services zurücknehmen. Weiterhin sorgt die Service Discovery für die Anmeldung, das Akzeptieren der Anmeldung und die Abmeldung in Eventgruppen. Diese Gruppen können nach Bedarf ein- oder ausgeschaltet werden.

Abbildung 2.1 zeigt den Header jeglicher SOME/IP Nachrichten, welcher in dem SOME/IP Protokoll definiert wird [7], d.h. auch für Nachrichten der Service Discovery wird der Header des zugrunde liegenden Protokolls genutzt. Dieser ist wichtig für die Interoperabilität verschiedener Versionen. Anhand der Reihenfolge in der Abbildung werden die einzelnen Felder erläutert:

Message ID: Jede Message ID soll innerhalb des Systems Fahrzeug einzigartig sein. 32 Bit lang soll der Benutzer die Belegung bewerkstelligen, in der Service Discovery Spezifikation wird sie noch näher definiert.

2.2 Software-Defined Networking(SDN)

Software-Defined Networking (SDN) ist eine Erweiterung der Netzwerkarchitektur um eine zentrale Softwarekomponente. Diese Komponente, der SDN-Controller, übernimmt die Steuerung des Netzwerkes und kontrolliert den Datenfluss, indem er den Switches

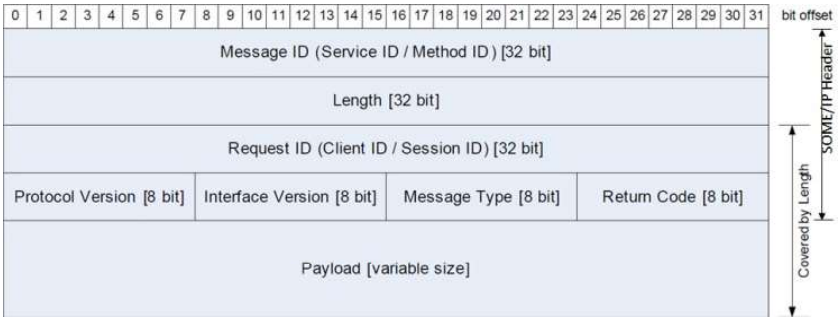

Abbildung 2.1: SOME/IP Header Format

Routingwege vorschreibt. Für die Übernahme der Steuerung müssen beteiligte Switches ein SDN-Protoll unterstützen, welches in dieser Arbeit das OpenFlow-Protokoll ist.

2.2.1 Klassische Netzwerke

Die Weiterleitung von Paketen innerhalb eines Netzwerkes sowie die Steuerung der Wegfindung wird in klassischen Netzwerken von den Routern/Switches übernommen. Es sind die wichtigsten Funktionen der Vermittlungsschicht (Networklayer) des ISO/OSI-Models, mit der Aufgabe, Pakete von ihrer Quelle zu einem Ziel zu transportieren. Der Prozess der Weiterleitung verwaltet jedes ankommende Paket und sucht die zu verwendende Ausgangsleitung anhand seiner Routing-Tabelle. [26] Ein Beispiel eines dynamischen Routing-Weges ist in Abbildung 2.2 dargestellt.

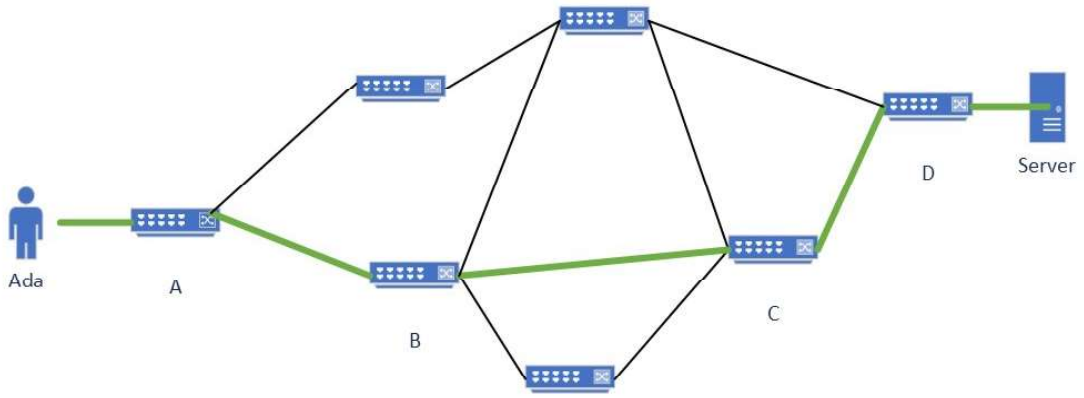

Abbildung 2.2: Routing im klassischen Netzwerk

Jeder Router verwaltet seine Routing-Tabelle, die für jeden im Netz vorhandenen Router einen Eintrag erhält. Dieser Eintrag besteht aus zwei Teilen: Zum einen aus der Ausgangsleitung(Port) zu diesem Router, zum zweiten die geschätzte Übertragungszeit zu diesem Ziel.(Bellmann-Ford-Algorithmus) Alle paar Millisekunden sendet jeder Router eine Liste mit geschätzten Übertragungszeiten an seine direkten Nachbarn und erhält seinerseits ihre Listen. Durch diesen Austausch werden die optimalen Wege zu einem Ziel ständig neu berechnet und in aktualisierten Routing-Tabellen festgehalten.

Dieser dynamische Aspekt der ständigen Aktualisierung, sorgt dafür, dass der Ausfall eines entfernten Knotens erst nach mehreren Aktualisierungen der Routing-Tabellen einem Router bekannt wird. Diese Problematik gilt des Weiteren für ähnliche Algorithmen und Weiterentwicklungen die in der heutigen Zeit im Bereich des Internet- und Netzwerkverkehrs eingesetzt werden.

Als eine Client-Server-Struktur im Internet vorherrschte war eine hierarchisch aufgebaute Netzwerkarchitektur aus Ethernet-Switches sinnvoll. [22] Einige wichtige Trends in der Nutzung der Netzwerke zeigen der Idee des selbstverwaltenden Netzwerkes allerdings die Grenzen auf. Die gestiegene Anzahl an Netzwerkgeräten und ein sich änderndes Muster bei der Datennutzung, bei dem Nutzer über verschiedene Geräte auf Unternehmensdaten zugreifen, wirken sich nachhaltig aus. Netzwerk-Administratoren sollen einerseits jedem Mitarbeiter auf jedem Gerät Unternehmensdaten zur Verfügung stellen, andererseits diese Geräte und Daten aber schützen. Zusätzlich sorgen Cloud-Dienste zu einer weiteren Erhöhung der Komplexität. Management-Tools auf Geräteebene und manuelle Prozesse sind die Herausforderungen der IT-Abteilungen. [16] So erhöht sich der Druck die Netzwerkprotokolle weiter zu entwickeln, um den Ansprüchen an höhere Leistung, Zuverlässigkeit, breitere Konnektivität und strengere Sicherheit gerecht zu werden. [22] Auf Grund der Trägheit aktueller IP-Netzwerke kann es fünf bis zehn Jahre dauern, bis ein neues Routing-Protokoll eingeführt wird. Wie zum Beispiel die Umstellung von IPv4 auf IPv6, die bereits zwanzig Jahre dauert und teils immer noch nicht abgeschlossen ist. [16] Eine Erneuerung des Internet Protocols(IP) wird als so eine gewaltige Aufgabe angesehen, dass sie in der Praxis als nicht umsetzbar gilt.[24]

2.2.2 Netzwerke mit SDN-Controller

Die zentrale Idee des Sofware-Defined Networking(SDN) ist die Steuerungslogik von den zugrunde liegenden Routern und Switches zu trennen und damit die vertikale Integration

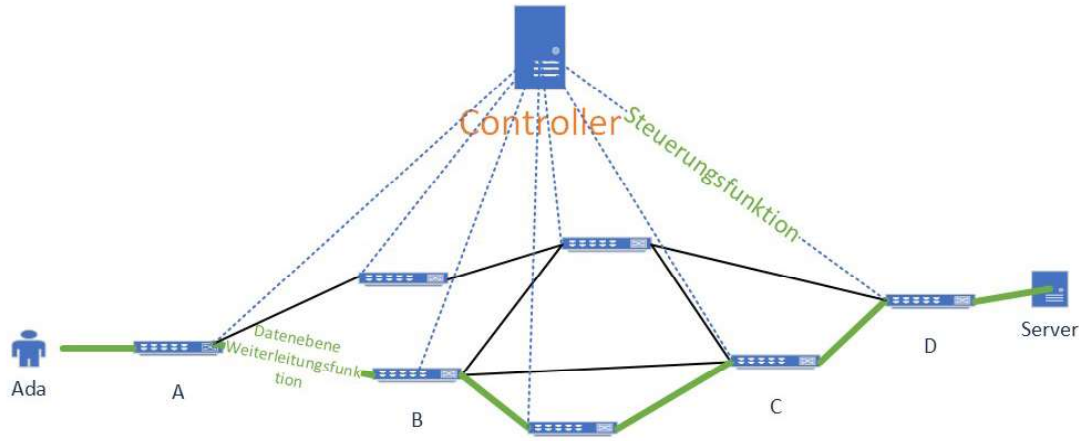

Abbildung 2.3: Einbindung eines SDN-Controllers

aufzubrechen. Die Steuerungsebene wird zentralisiert logisch gebündelt und ermöglicht somit die Programmierung des Netzwerkes. [16] Die Switches werden zu reinen Weiterleitungsgeräten und die Kontrolllogik wird in einem Controller implementiert oder einem NOS(network operating system). Dies vereinfacht nicht nur die Umsetzung von Richtlinien, sondern auch die (Neu-)Konfigurationen und die Weiterentwicklung von Netzwerken. [13] Auch SDN-Netzwerke greifen auf verteilte Steuerungsebenen zurück und sind somit nicht an ein physisches zentralisiertes System gebunden. [15] [11] In Abbildung 2.3 sieht man die Erweiterung des Beispiels zu klassischen Netzwerken um einen SDN-Controller.

Die Trennung der Steuerungs- von der Weiterleitungsebene kann durch eine klar definierte Programmierschnittstelle(API) zwischen den Switches und dem SDN-Controller erreicht werden. Der Controller erlangt über diese API direkte Kontrolle über die Elemente der Datenebene. Eine solche API ist OpenFlow. Ein OpenFlow-Switch verfügt über eine oder mehrere Tabellen mit Regeln zur Paketverarbeitung(flow table). Jede Regel gleicht eine Teilmenge des Datenverkehrs ab und führt bestimmte Aktionen(Löschen, Weiterleiten, Ändern usw.) für den Datenverkehr aus. Je nach den vorgegebenen Regeln des SDN-Controllers kann sich ein OpenFlow-Switch wie ein Router, Switch oder eine Firewall verhalten oder sogar andere Rollen übernehmen (z.B. Load Balancer, Traffic Shaper oder eine einfach Middlebox). [16]

2.2.3 Die vier Säulen des SDN[16]

Eine Definition von SDN als Netzwerkarchitektur umfasst vier Säulen:

1. Die Steuerebene und Datenebene sind entkoppelt. Die Steuerungsfunktionalität wird von Netzwerkgeräten entfernt, die zu einfachen (Paket-)Weiterleitungsgeräten werden.

2. Weiterleitungsentscheidungen sind nun flussbasiert anstelle von zielbasiert. Dieser Fluss wird durch einige Werte in den Feldern der Pakete, die als Übereinstimmungskriterium (Filter) dienen und einige Aktionen (Anweisungen) definiert. In dem SDN/OpenFlow-Kontext ist ein Fluss eine Folge von Paketen zwischen einer Quelle und einem Ziel. Alle Pakete des Flusses bekommen identische Service-Richtlinien in den Weiterleitungsgeräten. [8] [21] Diese Flussabstraktion ermöglicht die Vereinheitlichung des Verhaltens der verschiedenen Arten von Netzwerkgeräten, einschließlich Routern, Switches, Firewalls und Middleboxen. [12] Die Flussprogrammierung stellt eine unerreichte Flexibilität zur Verfügung, welche nur durch die Fähigkeiten der Flusstabellen begrenzt wird. [19]

3. Die Steuerlogik wird auf eine externe Instanz verlagert, einen SDN-Controller oder ein NOS. Das NOS ist eine Softwareplattform, die auf normaler Servertechnologie läuft und die wesentlichen Ressourcen und Abstraktionen bereitstellt, um die Programmierung von Weiterleitungsgeräten auf der Grundlage einer logisch zentralisierten, abstrakten Netzwerkansicht zu erleichtern. Sein Zweck ähnelt daher dem eines herkömmlichen Betriebssystems.

4. Das Netzwerk ist über Softwareanwendungen programmierbar, die auf dem NOS ausgeführt werden um mit den darunter liegenden Geräten der Datenebene zu interagieren. Dies ist ein grundlegendes Merkmal von SDN, welches als der wichtigste Mehrwert angesehen wird.

Die logische Zentralisierung bietet noch weitere Vorteile. Zum einen können Netzwerkrichtlinien einfacher und weniger fehleranfällig umgesetzt oder geändert werden. Zum anderen kann auf fehlerbehaftete Änderungen automatisiert hingewiesen und reagiert werden. Des Weiteren befähigt die globale Kenntnis des Netzwerkzustandes der Steuerungslogik zu anspruchsvolleren Netzwerkfunktionen, -diensten und -anwendungen.

Diese Flexibilität und die Programmierbarkeit des Netzwerkes sind gerade in der gut bekannten Netzwerkarchitektur von Fahrzeugen herausragende Eigenschaften. So ermöglicht SDN eine Rekonfigurierbarkeit und Flexibilität, die sich an Veränderungen im Netzwerk anpassen kann, z.B. durch Softwareupdates oder heruntergeladene Fahrassistenzsysteme. [9] Es ebnet den Weg für einen ganz neuen Markt, vergleichbar mit der vor einigen Jahren noch undenkbaren App-Vielfalt auf modernen Smartphones. So könnten zertifizierte Huptöne, Stimmungs-Innenraumbeleuchtungen oder sogar bezahlbare Aufhebung der Geschwindigkeitsbegrenzungen in Echtzeit für bestimmte Fahrzeuge aktiviert werden.

2.3 OpenFlow

OpenFlow wurde für Tests von experimentellen Protokollen für Netzwerke entwickelt. Es basiert auf einem einfachen Ethernet-Switch mit einer internen Flusstabelle. In Kombination mit einem standardisierten Interface können Flusseinträge hinzugefügt oder gelöscht werden. 2008 wurde eine Bitte an die Hersteller von Netzwerkgeräten veröffentlicht OpenFlow in ihre Produkte zu integrieren. [20] Eine Handvoll Anbieter reagierten mit einer OpenFlow-Option. Experimentelle Einsätze auf Universitätsgeländen und die Anlaufstelle ONF für die wachsende OpenFlow Gemeinschaft setzten eine SDN-Transformation in Gang. Bereits 2012 sprechen große Cloud-Anbieter über SDN-basierte Infrastrukturen und Bekanntheit und Einsatzmöglichkeiten steigen unaufhörlich. [17]

Im Kontext des Automotive Ethernets verbinden sich die Netzwerkkomponenten bei Systemstart. Wie der SDN-Controller das Netzwerk kennen lernt und wie sich die Pakete durch ein OpenFlow-Netzwerk bewegen wird hier erklärt. Die grundsätzlichen Aufgaben und Funktionsweisen wurden bereits im vorherigen Kapitel beschrieben.[1]

Der OpenFlow-Kanal ist das Interface, welches jeden Switch mit einem Controller verbindet. Über dieses Interface konfiguriert und verwaltet der Controller den Switch, er empfängt Ereignisse vom Switch und sendet Pakete zu ihm. Auch wenn die Schnittstelle zwischen dem Datenpfad und dem OpenFlow-Kanal implementierungsspezifisch ist, müssen die Kanalnachrichten selber gemäß des OpenFlow-Protokolls formatiert sein. Dieser Kanal ist normalerweise mit TLS verschlüsselt, kann aber auch direkt über TCP ausgeführt werden.[1]

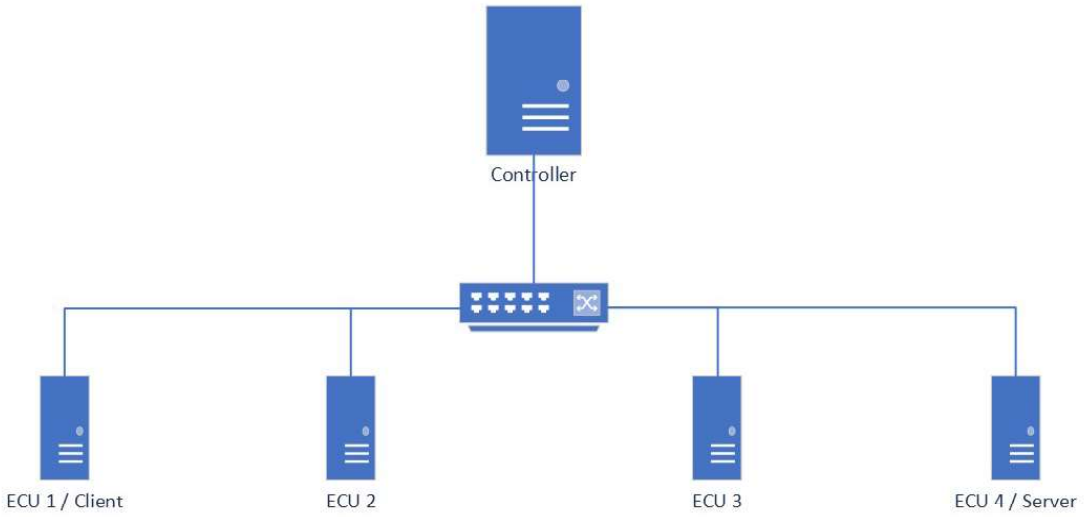

Abbildung 2.4: Exemplarisches Netzwerk mit einem Switch

Das Beispiel in Abbildung 2.4 soll den Nachrichtenaustausch verdeutlichen. In diesem Netzwerk sind neben dem Controller und dem Switch noch exemplarisch vier ECUs zu sehen. ECU 1 nimmt die Rolle eines Clients an und soll eine Nachricht vom Typ „suche Service“ senden. Der ECU 4 fungiert als Anbieter des Services und erhält somit die Rolle des Servers. Als erstes wird folgende symmetrische OpenFlow-Nachricht beim Hochfahren des Netzwerkes gesendet:

Hello: Hello Nachrichten werden bei Systemstart zwischen jedem Switch und dem Controller ausgetauscht.

Danach informiert der Switch über erreichbare Knotenpunkte mit der asynchronen Nachricht:

Port-Status: Mit diesem Befehl werden Portkonfiguration, Portstatus und Änderungen an diesen beiden gesendet. Eine Änderung wäre zum Beispiel, wenn der Link ausfällt.

Nun ist das Netzwerk betriebsbereit und die einzelnen Komponenten sind dem Controller bekannt.

ECU 1 benötigt nun für seinen Betrieb eine bestimmte Serviceinstanz und sendet hierzu die Service Discovery-Nachricht Request per UDP-Multicast in das Netzwerk. Angekommen beim Switch durchsucht dieser seine lokalen Flusstabellen, ob ein Eintrag für folgende Werte vorliegt: Die MAC Quell- und Zieladresse und die IP Quell- und Zieladresse.

Da dies die erste Nachricht in dem konstruierten Beispiel ist, sind die Flusstabellen noch leer. Der Zustand in dem kein Treffer in den Flusstabellen vorhanden ist wird Table-miss genannt. Dieses Request-Paket wird vom Switch asynchron an den Controller gesendet, damit Verhaltensregeln definiert werden.

Packet-in: Die Kontrolle eines Pakets wird an den Controller übertragen, dafür wird das Paket über einen reservierten Port mit einem Table-miss Flusseintrag an den Controller gesendet. Dies geschieht mit TCP und wird meinem ACK quittiert. (Es existiert auch die Möglichkeit das Paket beim Switch zu buffern, dann wird nur eine Buffer-ID gesendet und der Controller antwortet mit den Regeln und dem Hinweis auf die zugehörige Buffer-ID. Im Konzept der Arbeit wird darauf verwiesen, dass der Controller die Pakete zur Verarbeitung benötigt und daher wird nicht näher auf diese Möglichkeit des Bufferns eingegangen.)

Der Controller hat nun verschieden Möglichkeiten zu reagieren. In diesem Fall sendet er ein Packet-Out, welches gleich erklärt wird. Alternativ kann er eine Flow-MOD(Flussmodifizierungs)-Nachricht senden, mit neuen Regeln, welche in die Flusstabellen eingetragen werden. Diese Regeln beziehen sich auf Treffer und Masken in den Flusstabellen mit gleichen Quell- und Zieladressen. Eine weitere Alternative wäre die Flussmodifizierung, dass er diese Art von Paketen künftig wie Pakete ohne OpenFlow behandeln soll und nach eigenen Routing-Tabellen vorgeht.

Packet-Out: Das UDP-Request-Paket innerhalb des TCP-Packet-in-Pakets wird an den Switch zurück gesendet - versehen mit den Regeln über welche Ports der Multicast zu erfolgen hat. Formal heißt es: Die Nachricht muss eine Liste mit Regeln beinhalten, welche in der angegebenen Reihenfolge angewendet werden. Eine leere Aktionsliste führt zu einem drop des Pakets, es wird gelöscht.

Nachdem der Erhalt des Packet-Out mit einem ACK bestätigt wurde, arbeitet der Switch nun die Instruktionen ab und versendet den Multicast klassisch per UDP an ECU 2-4. Das Request wird nun von ECU 4 mit einem Serviceangebot(OfferService) beantwortet. Per UDP gesendet, führt dieses Paket erneut zu einem Table-miss und wird wie oben beschrieben an den Controller gesendet, kommt zurück zum Switch und wird dann an ECU 1 geschickt.

In folgendem Kapitel wird der Ist-Zustand mit SOME/IP und der Service Discovery bechrieben. Vorweg genommen: Die Service Discovery von ECU 1 antwortet mit einem Subscribe Eventgroup und ECU 4 akzeptiert mit einem Subscribe Eventgroup ACK.

Dann wird die Kommunikation von dem zugrunde liegende SOME/IP-Protokoll übernommen und die Flusseinträge für die Eventgroup eingetragen. Sobald diese Einträge in den Flusstabellen vorhanden sind, werden zyklische Nachrichten vom Switch direkt an die Empfänger geleitet.

3 Ist-Analyse und Problemstatement

Mit der gestiegenen Anzahl an Steuerungsgeräten ist die Suche nach einer höheren Bandbreite mit Einführung des Ethernetstandards 100Base T1 und 1000Base T2 bestimmt nicht abgeschlossen. Die signifikant erhöhte Geschwindigkeit des Netzwerkverkehrs erlaubt nun auch die Nutzung von Technologien, welche vor einigen Jahren noch undenkbar gewesen wären. Beispielsweise die Echtzeitübertragung von Videodaten über das fahrzeuginterne Netzwerk, wäre mit dem CAN-Bus unmöglich gewesen. Autonome Fahrzeuge benötigen neben Radar, Lidar und anderen Sensoren auch die Auswertung von Echtzeit-Videostreams. Das heisst auf der einen Seite wird weiter an immer schnelleren Übertragungsgeschwindigkeiten geforscht und auf der anderen Seite muss versucht werden bestehende Basistechnologien zu verschlanken.

Die Steuerungseinheiten bieten Dienste an und benötigen andere Dienste, um Daten bereitstellen zu können. Die Identifikation und das Auffinden dieser Dienste erfolgt in jeder Electronic Control Unit (ECU) durch den Einsatz einer SOME/IP-Service Discovery (SD), welche klar definierte Service Discovery-Nachrichten versendet. Im folgenden wird eine SD-Nachricht analysiert anhand der autosar-Protokoll Spezifikation (PRS) und der zugehörigen Nummer.

3.1 SOME/IP-Header und SOME/IP-SD-Header

Der SOME/IP-Header der Nachricht ist in der SOME/IP Spezifikation definiert und muss übernommen werden. [PRS_SOMEIPSD_00250] Einige variable Felder werden allerdings in der Spezifikation der Service Discovery festgelegt. [PRS_SOMEIPSD_-00151] bis [PRS_SOMEIPSD_00164] Das Format des SOME/IP-SD-Header wird im Anschluss um einige Felder erweitert.

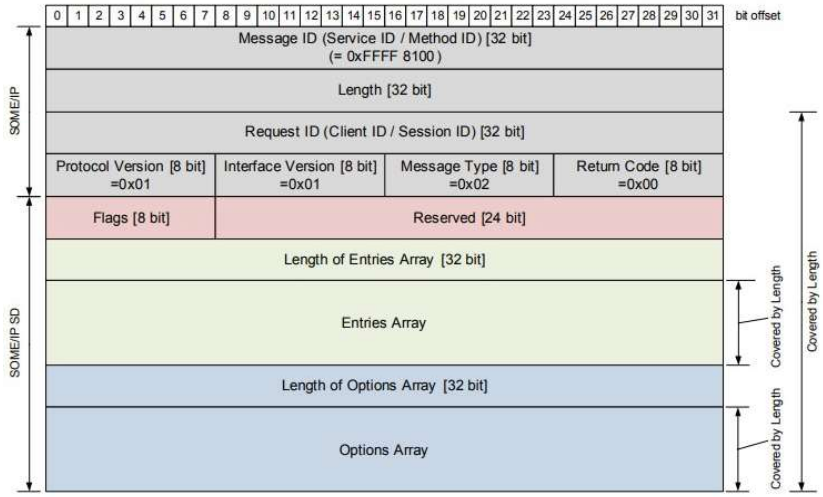

Abbildung 3.1: SD Nachrichten Format

So wird die Service ID auf 0xFFFF und die Method ID auf 8100 festgelegt. Dies ergibt dann eine konstante Message ID für sämtliche Nachrichten der Service Discovery (0xFFFF 8100).

Ein uint 32 gibt die Länge der SOME/IP-SD Nachricht in Byte an, beginnend mit dem Byte nach dem Length-Feld und endend mit dem letzten Byte der SD Nachricht. Die Client ID wird mit 0x0000 bestimmt, da nur eine Instanz der SOME/IP-SD existiert.

Für die Session ID gelten mehrere Vorgaben. Erstens soll sie mindestens eins sein und 16 Bit lang sein, das ergibt bei der 65535. Nachricht muss dieser Wert wieder auf eins gesetzt werden.

Der Protokollversion und der Interfaceversion werden je die Werte 0x01 zugeordnet und der Nachrichtentyp soll von der Art Benachrichtigung(notification) sein, also 0x02. Auch der Returncode wird vorgegeben mit 0x00.

Dies ergibt das nur zwei sich ändernde Werte im Header der SOME/IP-SD Nachricht vorliegen. Zum einen die Länge der Nachricht und zum anderen die Request ID, welche sich mit jeder Nachricht durch die Session ID ändert.

Die Erweiterung des SOME/IP-SD-Headers beginnt nach dem SOME/IP-Header mit einem Feld von Flags. Danach folgen 24 reservierte Bits, ein Array aus den Einträgen und ein Array mit den zugehörigen Optionen der Einträge.

[PRS_SOMEIPSD_00254] bis [PRS_SOMEIPSD_00258] & [PRS_SOMEIPSD_00631] Das acht Bit lange Feld Flags beginnt mit dem Reboot-Flag. Dieses wird gesetzt, wenn die Session-ID auf Eins zurückgestellt wurde. Das Reboot-Flag soll für jede Sender und Empfänger Beziehung gesetzt werden. Dies bedeutet, dass es getrennte Zähler geben muss. Auf Seiten des Senders existiert ein Zähler für Multicasts und je einen für jede aktuelle Unicastverbindung. Auf Seiten des Empfängers existiert je ein Zähler für jede Multicastverbindung und je ein Zähler für jede Unicastverbindung.

[PRS_SOMEIPSD_00259] & [PRS_SOMEIPSD_00540] Das zweite Bit der Flags ist ein historisches Überbleibsel, es soll auf eins gesetzt werden und bedeutet, dass der Empfang von Unicast unterstützt wird.

[PRS_SOMEIPSD_00702] Bit drei bis acht der Flags sind bisher undefiniert und sollen beim Senden auf null gesetzt werden und beim Empfang ignoriert werden.

[PRS_SOMEIPSD_00261] Im SD-Header folgt ein Feld der Länge 24 Bit mit dem Titel reserviert(Reserved).

[PRS_SOMEIPSD_00262] bis [PRS_SOMEIPSD_00267] Auf den Header folgen ein Array für Einträge und ein Array für Optionen zu den Einträgen, wobei jeweils vorher ein 32-Bit Feld für die Angabe der Länge der Arrays vorgesehen ist. Die Reihenfolge der Abarbeitung der Einträge soll der Reihenfolge in dem Array entsprechen. Dabei existieren zwei Eintragstypen, zum einen Diensteinträge und zum anderen Eventgroupeinträge. Hiervon können beliebig viele in einer SD-Nachricht zusammengefasst werden. Die Größe der Arrays wird in Byte angegeben.

3.1.1 Format der Diensteinträge

[PRS_SOMEIPSD_00268] Jede Nachricht vom Typ Diensteintrag soll 16 Byte lang sein und umfasst zehn Felder von Informationen. (s. Abb. 3.2)

Type Feld: Ein uint8 belegt für SucheDienst(FindService) 0x00, BieteDienst(OfferService) 0x01 und StoppBieteDienst(StopOfferService) 0x01.

Index 1st options Feld: Ein uint8 mit dem Index des Optionsarrays an welchem der Optionsdurchlauf eins für diesen Eintrag beginnt.

Index 2nd options Feld: Ein uint8 mit dem Index des Optionsarrays an welchem der Optionsdurchlauf zwei für diesen Eintrag beginnt.

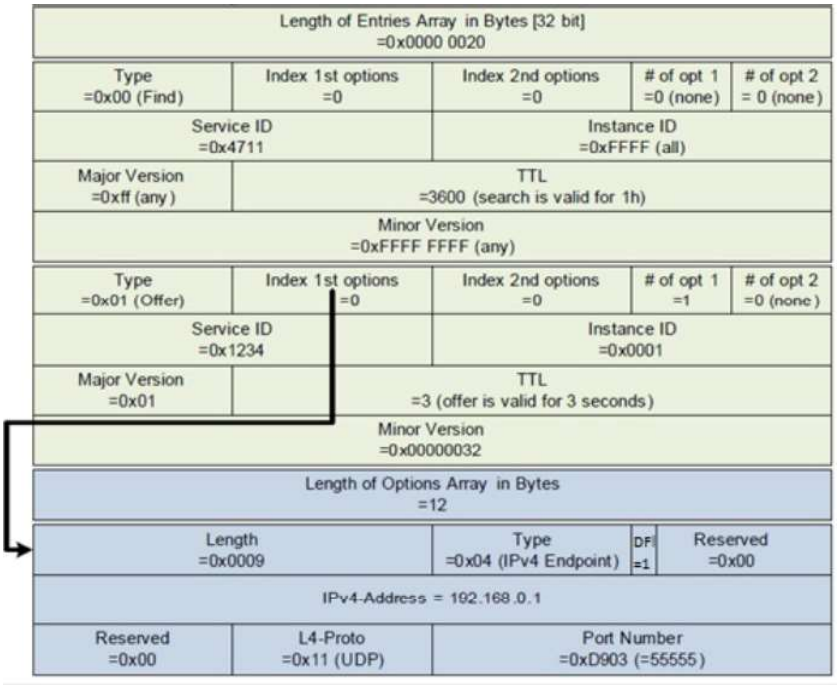

Abbildung 3.2: SD Einträge und Optionen

# of opt 1 Feld: Ein uint4 für die Anzahl der Optionen des ersten Optionsdurchlaufs.

# of opt 2 Feld: Ein uint4 für die Anzahl der Optionen des zweiten Optionsdurchlaufs.

Service-ID Feld: Ein uint16 für die angebotene oder gesuchte Dienst-ID des Eintrags.

Instance-ID Feld: Ein uint16 für die Instanz-ID der Dienstinstanz des Eintrags oder 0xFFFF, wenn alle Instanz-IDs des Dienstes gemeint sind.

Major Version Feld: Ein uint8 für die höchste Version des Dienstes der gesucht wird.

TTL Feld: Ein uint24 für die Lebenszeit des Eintrags in Sekunden.

Minor Versions Feld: Ein uint32 für die Mindestversion des Dienstes der gesucht wird.

[PRS_SOMEIPSD_00270] Jede Nachricht vom Typ Eventgruppeneintrag soll 16 Byte lang sein und umfasst zwölf Felder von Informationen. (s. Abb. 3.2)

Type Feld: Ein uint8 belegt für anmelden(Subscribe) 0x06, abmelden(StopSubscribeEventgroup) 0x06, Anmeldebestätigung(SubscribeAck) 0x07 und Abmeldebestätigung(SubscribeEventgroupNack) 0x07.

Index 1st options Feld: Ein uint8 mit dem Index des Optionsarrays an welchem der Optionsdurchlauf eins für diesen Eintrag beginnt.

Index 2nd options Feld: Ein uint8 mit dem Index des Optionsarrays an welchem der Optionsdurchlauf zwei für diesen Eintrag beginnt.

# of opt 1 Feld: Ein uint4 für die Anzahl der Optionen des ersten Optionsdurchlaufs.

# of opt 2 Feld: Ein uint4 für die Anzahl der Optionen des zweiten Optionsdurchlaufs.

Service-ID Feld: Ein uint16 für die angebotene oder gesuchte Dienst-ID des Eintrags.

Instance-ID Feld: Ein uint16 für die Instanz-ID der Dienstinstanz des Eintrags oder 0xFFFF, wenn alle Instanz-IDs des Dienstes gemeint sind.

Major Version Feld: Ein uint8 für die Hauptversion der Dienstinstanz zu welcher die Eventgruppe gehört.

TTL Feld: Ein uint24 für die Lebenszeit des Eintrags in Sekunden.

Reserved Feld: Ein uint12 welches auf 0x0000 gesetzt werden soll.

Counter Feld: Ein uint4 Zähler, der genutzt wird wenn der gleiche Abonnent sich mehrfach bei der Eventgruppe anmeldet, bei Nichtverwendung auf 0x0 setzen.

Eventgroup ID Feld: Ein uint16 für die ID der Eventgruppe.

3.2 Optionen der Einträge

Die SOME/IP-Service Discovery Nachrichten sollen Informationen enthalten, wie der Kommunikationspartner kontaktiert werden kann. Diese Kontaktinformationen werden anhand der Optionen mitgeteilt. Zum Erreichen einer Dienstinstanz müssen IP-Adresse, Transportprotokoll und Portnummer in den Optionen hinterlegt sein. [PRS_SOMEIPSD_00833] Hierbei werden zwei Optionsarten unterschieden. Zum einen gemeinsame Optionen für mehrere SOME/IP-SD-Einträge und zum anderen Optionen die für jeden SOME/IP-SD-Eintrag unterschiedlich sind. Diese in zwei Optionsdurchläufe zu trennen scheint der effizienteste Weg beide Formate zu unterstützen und die Netzwerklast zu begrenzen.

Typ	Optionsart	Format beschrieben in:
0x01	Konfiguration	[PRS_SOMEIPSD_00276]
	Namen des Dienstes oder zusätzliche Konfigurationsmöglichkeiten	
0x02	Lastenausgleich	[PRS_SOMEIPSD_00544]
	Priorisierung von Dienstinstanzen bei mehrfachem Angebot	
0x04	IPv4-Endpunkt	[PRS_SOMEIPSD_00307]
	Serverseitig: Quelle der Ereignisse/Events Clientseitig: Ziel der Ereignisse/Events	
0x06	IPv6-Endpunkt	[PRS_SOMEIPSD_00315]
	s. IPv4-Endpunkt	
0x14	IPv4-Multicast	[PRS_SOMEIPSD_00326]
	Serverseitig: Quelle der Multicastevents oder -mitteilungen Clientseitig: Ziel der Multicastevents oder -mittelungen	
0x16	IPv6-Multicast	[PRS_SOMEIPSD_00333]
	s. IPv4-Multicast	
0x24	IPv4 SD Endpunkt	[PRS_SOMEIPSD_00552]
	genutzt für ECUs ohne Service Discovery, über welche SD sie zu erreichen sind	
0x26	IPv6 SD Endpunkt	[PRS_SOMEIPSD_00559]
	s. IPv4 SD Endpunkt	
Quelle und Ziel bestehen jeweils aus IP-Adresse, Protokoll der Transportschicht und Portnummer		

Abbildung 3.3: Optionen und ihre Typen

In Tabelle 3.3 sind die verschiedenen Optionsarten mit den zugehörigen Typnummern aufgelistet. Jeweils darunter sind typische Anwendungsmöglichkeiten zu finden.

3.3 Startverhalten

[PRS_SOMEIPSD_00833] Die Service Discovery muss für jede Dienstinstanz folgende drei Phasen durchlaufen:

- Anfängliche Wartephase (Initial Wait Phase)
- Wiederholungsphase (Repetition Phase)
- Hauptphase (Main Phase)

3.3.1 Zustände der Service Discovery eines Serverprozesses

In Abbildung 3.4 werden Zustände und Übergänge der Service Discovery beispielhaft anhand eines Serverprozesses skizziert:

Vor der anfänglichen Wartephase wird sichergestellt, dass alle benötigten Komponenten und Dienste zur Verfügung stehen und erreichbar sind. Wenn dies Prüfung abgeschlossen ist und notwendige Konfigurationen durchgeführt wurden, kann ein Serverprozess seinen Dienst anbieten. So wird gewährleistet, dass sich die Dienstanbieter in einem stabilen Ausgangszustand befinden um fragile Kommunikationszustände zu vermeiden.

[PRS_SOMEIPSD_00416] Nach dem Eintritt in den Zustand "Ready"wird eine randomisierte Verzögerung abgewartet um die erste Nachricht zu versenden. Diese anfängliche Wartephase (Initial Wait Phase) dient der Entlastung des Netzwerkes. Es soll verhindert werden, dass sämtliche Prozesse gleichzeitig ihre initialen Nachrichten versenden. In Abb. 3.4 ist zu sehen, wie ein /send(OfferService) durch den abgelaufenen Timer ausgelöst wird und den Übergang in die zweite Phase kennzeichnet.

Zusammen mit der zweiten Phase, der Wiederholungsphase (Repetition Phase) wird sichergestellt, dass auch der Verlust einiger UDP-Pakete abgefangen wird. So wird mit jeder beantworteten Nachricht sowohl die Menge an Dienstangeboten sukzessiv gemindert als auch die Netzwerklast verringert. Die Anzahl der Wiederholungen wird begrenzt und nach jedem Senden eines Dienstangebotes wird ein Timer gesetzt, wann die nächste Wiederholung erfolgt. Durch das Erhalten einer Dienstanfrage (receive(FindService)) wird dieser Timer zurückgesetzt nachdem mit einem Dienstangebot geantwortet wurde. Sei die Wiederholungsanzahl auf Null gesetzt oder durch Ablauf der Timer die Maximalanzahl der Durchläufe erreicht, wird in die nächste Zustandsphase gewechselt.

Die Hauptphase (Main Phase) setzt die zweite Phase mit leichten Änderungen fort. Der Timer wird auf eine konstante zyklische Mitteilungsverzögerung gesetzt und bei Ablauf ein Dienstangebot gesendet. Bei eingehenden Suchanfragen wird analog zur Phase zwei reagiert, der Timer wird zurückgesetzt und mit einem Dienstangebot geantwortet.

3.3.2 Zustände der Service Discovery eines Clientprozesses

Die Zustände der Service Discovery eines Clientprozesses beinhalten für jeden benötigten Dienst die gleichen Phasen und Zustände wie bei einem Serverprozess beschrieben,

sobald er gestartet wird. Der Unterschied besteht in der Hauptphase. Hier befindet sich der Clientprozess im Zustand Dienst bereit (Service Ready) und erneuert bei jedem eingehenden Dienstangebot seine Lebenszeit(TTL). Sollte die Lebenszeit ablaufen wird erneut nach dem benötigten Dienst gesucht. Zwei Ereignisse sorgen für ein Stoppen der Hauptphase. Zum einen kann die Anzahl der maximalen Wiederholungen in Phase zwei erreicht werden, das bedeutet keine Instanz des benötigten Dienstes ist erreichbar. Zum anderen der Erhalt der Nachricht, dass ein benötigter Dienst nicht mehr angeboten wird(receive(StopOfferService)) . Diese Nachricht sorgt auch für den Stop des Dienstes, wenn er sich bereits in der Hauptphase befindet.

Wenn keine benötigte Dienstinstanz verfügbar ist, muss seine Service Discovery warten. Dieser Zustand wird als Nicht Angefragt(Not Requested) bezeichnet und verharrt in einer Dienst-nicht-gesehen-Position. Um wieder in die Hauptphase eintreten zu können muss die Service Discovery ein Dienstangebot einer passenden Dienstinstanz erhalten.

3.4 Publish/Subscribe mit SOME/IP und SOME/IP-SD

Die SOME/IP Spezifikation [7] bietet ein grundlegendes Anfrage/Antwort-Konzept (4.2.2 Request/Response Communication), Anfragen ohne benötigte Antwort (4.2.3 Fire&Forget Communication) und Mitteilungsereignisse (4.2.4 Notification Events). Hier wird darauf verwiesen, dass das Konzept des Publish/Subscribe Musters in der SOME/IP-Service Discovery implementiert ist. Im Gegensatz zu diesen Mechanismen existieren Anwendungsfälle in welchen ein Client regelmäßig einen Satz Parameter von einem Server benötigt, aber diese nicht jedes Mal von dem Server anfragen soll. Diese werden Benachrichtigungen genannt und betreffen Ereignisse oder Felder.

[PRS_SOMEIPSD_00443] & [PRS_SOMEIPSD_00446] & [PRS_SOMEIPSD_00446] Es sollen sich alle Clients, welche Ereignisse oder Benachrichtigungen benötigen, zur Laufzeit beim Server registrieren. So sei der OfferService-Eintrag einer serverseitigen SOME/IP-SD nicht nur als Angebot Mitteilungen zu pushen, sondern als Trigger zum Abonnieren zu verstehen. Jeder an dem Angebot interessierte Client hat mit dem SOME/IP-SD-Eintrag Abonniere Eventgruppe zu antworten. Sobald der Client angemeldet ist, können OfferServices so lange verworfen werden, bis das Reboot-Flag gesetzt wurde. Zu jedem Neustart des Servers muss auch erneut ein Abonnement des Services erfolgen.

3.5 Problemstatement

Aus den Zustandsdiagrammen wird klar ersichtlich mit welchen Problemen die Service Discovery hauptsächlich zu kämpfen hat. In der anfänglichen Wartephase wird eine Zufallszahl gewählt, damit nicht alle Dienstinstanzen gleichzeitig ihre Nachrichten für Dienstangebote und Dienstgesuche versenden. Die Wiederholungsphase dient vorrangig dem Zweck Knotenpunkte im Netzwerk zu erreichen, die erst später zum Status "Ready"wechseln. Aber weiterhin werden durch die Wiederholungsphase Fehler abgefangen, welche durch verlorene UDP-Pakete entstehen. Jede Steuerungseinheit soll für jede Dienstinstanz, die sie anbietet oder anfragt, diese drei Phasen durchlaufen, die im Zustandsdiagramm visualisiert werden. Die Nachrichten vom Typ OfferService und FindService werden per Multicast in das Netzwerk gesendet. Multicast nutzt klassisch das UDProtokoll, ein verbindungsloses Protokoll, welches keine Möglichkeit liefert die Paketverluste zu überprüfen.

Ein Problem besteht also in der Menge der versandten Multicast-Nachrichten, welche mit jeder neuen ECU und jeder neuen Abhängigkeit von anderen Sensoren weiter steigt.

Ein weiterer kritischer Aspekt ist das Thema Sicherheit. Jede ECU darf Dienstinstanzen anbieten, anfragen und nach Bestätigung der Eventgroup-Aufnahme auch Remote Procedure Calls durchführen. Um auszuschließen, dass beispielsweise ein Infotainmentsystem auf Bremsen oder Gaspedal Einfluss nehmen darf, müssten in den Service Discoveries Blacklists geführt werden. Auf diesen Blacklists müsste festgehalten werden, welche Services für welche Steuerungseinheiten nicht zur Verfügung stehen. Bei jedem neuen Sensor oder jeder neuen ECU müssten demnach bestehende Rechte angepasst werden. Das System Auto ist auf Gewichtsreduktion, Geschwindigkeit beim Systemstart und Stromsparen ausgelegt. Viele Einheiten, die zum Erhalt ihrer variablen Blacklists Strom benötigen, auch wenn das Fahrzeug mehrere Wochen nicht genutzt wird, passen nicht in das Konzept.

Als Sicherheitsmechanismus könnte auch eine Verschlüsselung der Nachrichten und eine Authentifizierung der Kommunikationspartner eingesetzt werden. Auch hier stellt sich die Frage, wie sich nachträglich eingeführte Sicherheitsmaßnahmen auf das S von SOME/IP auswirken - die Skalierbarkeit. Wirken sich Systemerweiterungen nachteilig auf die Netzwerklast aus? Führen sie zu höheren Latenzzeiten.

Viele der Aspekte lassen sich durch die Einführung einer bestehenden Technologie zentralisiert lösen. Das Software Defined Networking(SDN) in Kombination mit einer Service Discovery im SDN-Controller wird im Konzept vorgestellt.

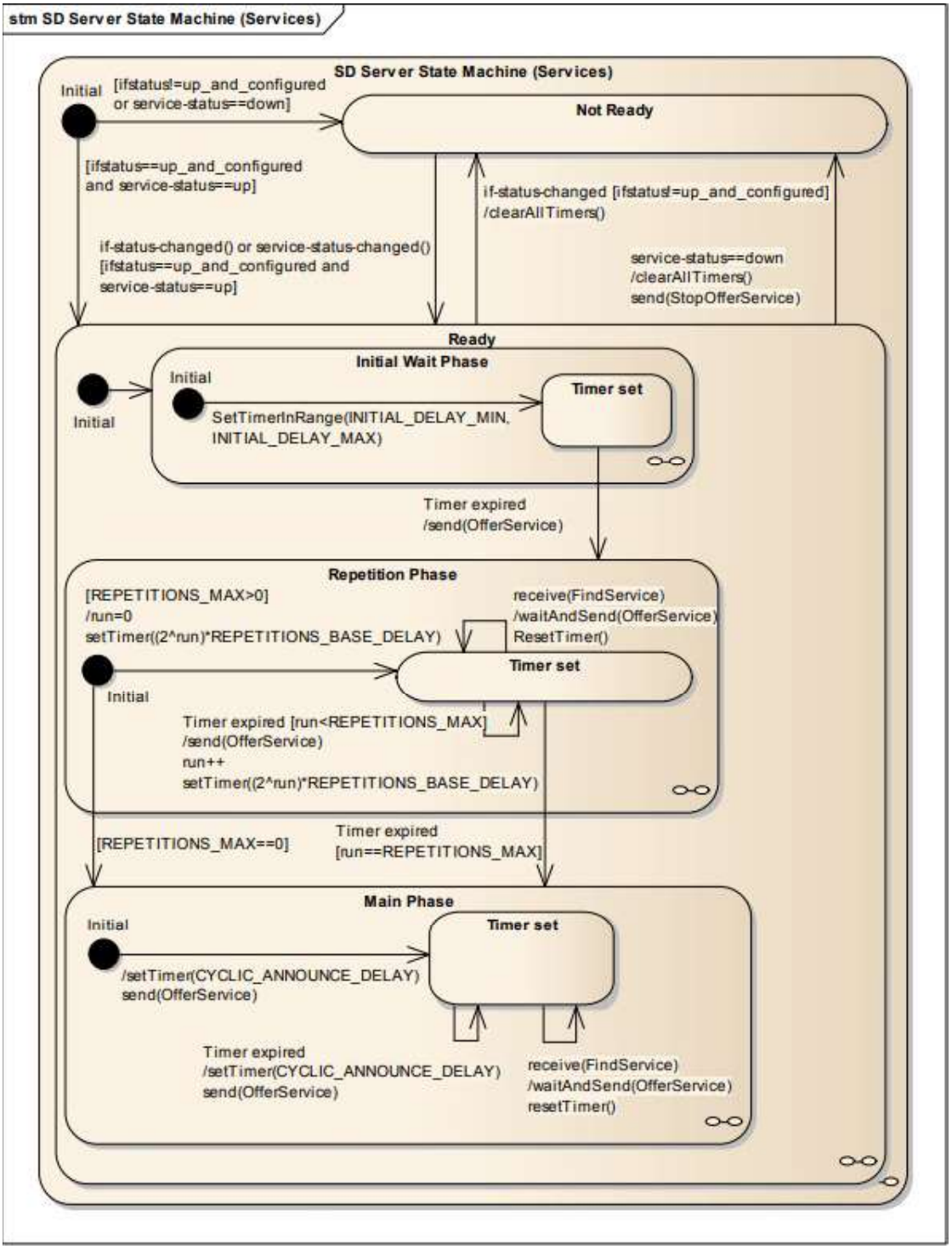

Abbildung 3.4: SD Server Zustandsdiagramm

4 Konzept

Die Programmierung des Netzwerkes, die vierte Säule des Software-Defined Networkings [16], erlaubt neue Ansätze der Optimierung des Netzwerkverkehrs. Zwei Eigenschaften des Zusammenspiels aus SOME/IP und SDN-Controller bilden die Grundlage dieses Konzepts. Erstens werden die SOME/IP-SD-Nachrichten, gekapselt in den OpenFlow-Paketen, an den SDN-Controller mitgesendet. Der Controller kann somit die Routingentscheidungen für die Switches treffen, erhält aber auch Einblick in die SOME/IP-SD Kommunikation. Zweitens existiert durch die Programmierung des Netzwerks die Möglichkeit diese Pakete zu verarbeiten und nur erforderliche Weiterleitungen durchzuführen.

Der SDN-Controller wird um eine SOME/IP-Service Discovery erweitert. Erhält der Controller OpenFlow-Pakete werden diese nun an die Service Discovery übermittelt. Die erste Aufgabe der Service Discovery besteht darin, aus den OpenFlow-Paketen die SD-Nachrichten herauszufiltern und zu verarbeiten. Aus diesen Nachrichten werden relevante Informationen herausgearbeitet und gecached. Anstatt nur die Routing-Entscheidungen der einzelnen Pakete zu treffen, können somit Anfragen/Angebote direkt aus diesem Cache beantwortet werden. Einerseits soll mit dieser Anwendung eine Reduzierung der Netzwerklast und andererseits eine bessere Skalierbarkeit erreicht werden.

In einer Simulation soll evaluiert werden wie nachhaltig sich dieser Cache auf die Anzahl der Nachrichten im Netzwerk auswirkt.

4.1 Vorhandene Infrastruktur

Das Konzept und die Implementierung fügen sich in eine Simulationsumgebung der CoRE-Group der HAW-Hamburg [5] ein. CoRE steht für "Communication over Real-Time Ethernet" und beinhaltet den Umgang mit zeitkritischen und sicherheitskritischen Anwendungen auf der Grundlage von Ethernet-Netzwerktechnologien. Die CoRE-Group

erforscht seit 2008 Technologien für den Einsatz in anspruchsvollen Echtzeitumgebungen von Flugzeugen und Fahrzeugen.

Als Ausgangspunkt diente das INET Framework für ereignisbasierte Simulationen in Ethernet-Netzwerken innerhalb der Simulationsumgebung OMNeT++[23]. Hieraus hat sich ein beeindruckendes Werk aus Frameworks zur Echtzeitsimulation von Technologien in der Automobilbranche entwickelt. Unter anderem stehen die Frameworks SDN4CoRE (Software-Defined Networking for Communication over Realtime Ethernet) und OpenFlow zur Verfügung, welche zu der Idee inspiriert haben, den SDN-Controller um eine Service Discovery zu bereichern.

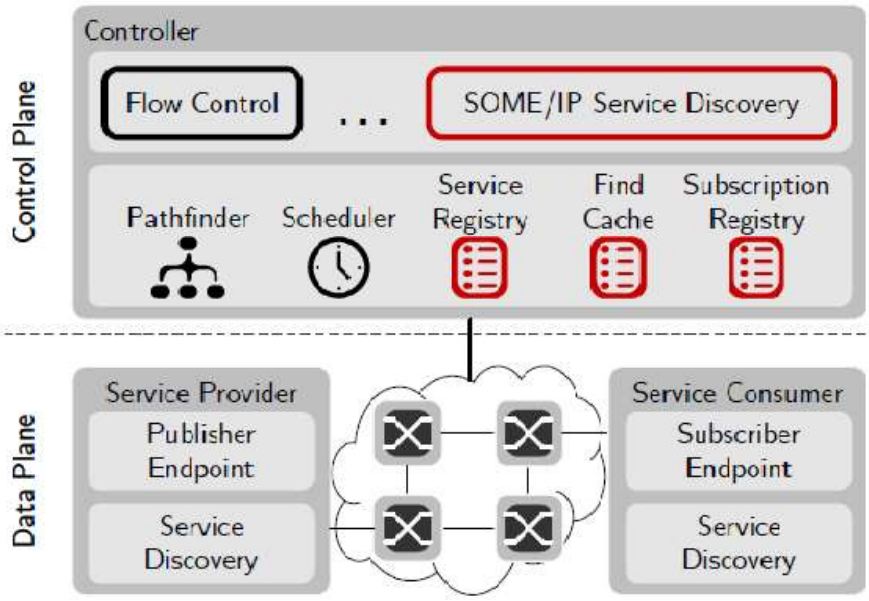

Abbildung 4.1: Konzept für eine SDN-unterstützte SOME/IP [9]

Abb. 4.1 zeigt: Mit dem Einsatz des SDN-Controllers erreichen wir eine klare Trennung zwischen der Datenebene und der Steuerungsebene des Netzwerks. Die Switches übernehmen ausschließlich die Aufgabe der Datenweiterleitung und deligieren die Routing-Entscheidungen an den Controller. Auf der Datenebene verbinden die Switches die dienstanbietenden und die dienstsuchenden ECUs.

4.2 Nachrichtenverlauf mit einem SDN-Controller

Aus der Umsetzung des SOME/IP-Protokolls ergibt sich, dass jeder Knoten eine Service Discovery(SD) besitzt oder durch eine solche erreichbar ist. In Abb 4.1 sind exemplarisch je ein Service Provider und ein Service Consumer dargestellt. Im folgenden wird veranschaulicht, welche Service Discovery-Nachrichten zwischen dem Service Consumer und dem Service Provider ausgetauscht werden. Außerdem werden die Auswirkungen durch die Trennung der Datenebene und Steuerungsebene des Netzwerkes gezeigt und wie der SDN-Controllers die Steuerung ausführt.

Abb. 4.2 zeigt beispielhaft die Anfrage eines Services und die notwendigen Nachrichten bis hin zu der Antwort durch ein Serviceangebots. Wir betrachten das Netzwerk bei Systemstart, d.h. die Flusstabellen der Switches beinhalten noch keine Einträge. Für eine bessere Verständlichkeit des Kapitels Implementation wird folgend die eingedeutschte Benennung der Komponenten eingeführt. Die eigentlichen Nachrichten werden als ein SOME/IP-SD-Header versendet. In diesem Header sind die Diensteinträge(Service-Entries) mit zugehörigen Optionen(options) gekapselt. Sowohl die Service-Entries als auch die options werden als eigene doppel-verlinkte Liste versendet. In den einzelnen Service-Entries stehen die Angaben an welcher Stelle die zugehörigen options beginnen und ihre Anzahl. Des Weiteren werden die Service-Entries in einer ServiceInstanceTable gespeichert.

Der gleiche Service kann in dem Netzwerk von verschiedenen ECUs angeboten werden, um sie zu unterscheiden hat jeder Knoten unterschiedliche Instanz-Identifikationsnummern (Instance-IDS). Wird eine ganz bestimmte Instanz angefragt, so spricht man von einer Service-Instance. Sollte die Instance-ID nicht von belang sein, so werden alle Services angefragt und die Instance-ID auf 0xFFFF gesetzt.

Die Suche nach einem Dienstangebot wird im Service-Entry als vom Typ Find eingetragen, daher wird die Nachricht als Ganzes künftig als Find bezeichnet und ggf. in einer RequestTable gespeichert.Das Angebot eines Dienstes wird im Service-Entry als vom Typ Offer eingetragen, daher wird die Nachricht als Ganzes künftig als Offer bezeichnet.Die Open-Flow-Flusstabellen der Switches werden künftig als flow tables bezeichnet. Die Nachricht zur Anmeldung bei einer Eventgroup wird fortan als SubscribeEventgroup bezeichnet und die Bestätigung der Anmeldung als SubscribeEventgroupAck.

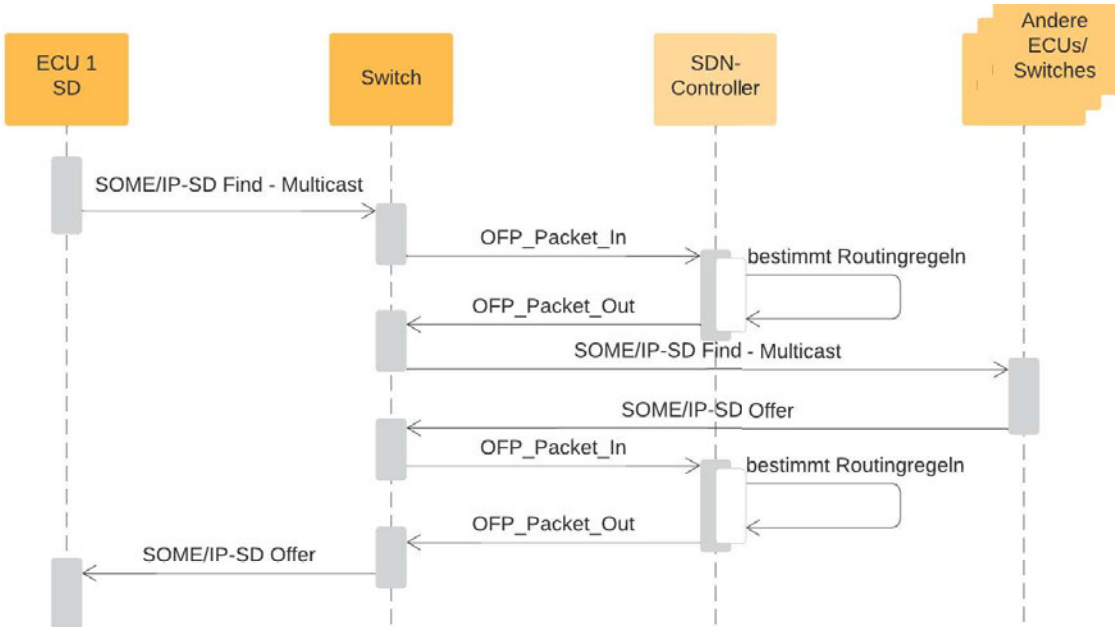

Abbildung 4.2: Sequenzdiagramm der Servicesuche bis zur Antwort

4.2.1 Beispielsequenz einer Find-Nachricht

Die Service Discovery des Consumers ECU1 sendet eine Find-Nachricht. Diese Nachricht wird als UDP-Multicast versendet. Sie erreicht den Switch und für dieses Dienstangebot existiert im Switch noch kein flow table entry. Daher kapselt der Switch das Find in ein OpenFlow-Paket vom Typ OFP_Packet_In und sendet es an den SDN-Controller um Steuerungsbefehle zu erhalten, bzw die Kontrolle des Pakets an den Controller zu übertragen. OpenFlow nutzt für die Übertragung TCP, somit wird das anfängliche UDP-Paket nun in einem TCP-Paket gekapselt. Bei Erhalt des Pakets entscheidet der SDN-Controller, dass der enthaltene Multicast ausgeführt werden soll. Er kapselt den Multicast in ein OpenFlow-Paket vom Typ OFP_Packet_Out mit den Anweisungen den Multicast auszuführen. Hierbei übernimmt er die die Switch-Control-Information, an welchem Port das Find angekommen ist und klammert ihn bei den Anweisungen für die Weiterleitung aus. Multicasts an den eigentlichen Urheber sind per Definition ausgeschlossen.

Das Find des ECU1 erreicht zum zweiten Mal den Switch und wird gemäß den Controller-Anweisungen an alle Ports, exklusive des Ports von ECU1, weitergeleitet. Sollte an einem Port ein weiterer Switch verbunden sein, so wird das Paket auch hier per OFP_Pa-

cket_In und OFP_Packet_Out mit Steuerungsanweisungen vom Controller versehen. In diesem Fall würde der Port zum ersten Switch ausgenommen werden.

Die SD eines Producers erhält diese Serviceanfrage schlussendlich und antwortet mit einer Offer-Nachricht. Natürlich können auch mehrere Producer auf die Anfrage antworten, der Ablauf wäre der Gleiche und der Consumer würde sich anhand einer Priorisierung eine Service-Instance auswählen. Dieses Offer durchläuft als UDP-Unicast das gleiche Procedere. Für jedes Switch auf dem Weg, wird es als OFP_Packet_In an den Controller gesendet und als OFP_Packet_Out mit den Steuerungsanweisungen zurück gesendet. Der einzige Unterschied besteht darin, dass der SDN-Controller nun den Weg zum Endpunkt kennt und die Steuerungsentscheidungen aufgrund des kürzesten Weges oder anderer Entscheidungsmöglichkeiten trifft.

Das Offer kommt beim ECU1 an und triggert eine Anmeldung bei der Eventgroup des Anbietenden. Folglich wird ein SubscribeEventgroup an den Serviceanbieter versendet, dies geschieht wieder als als UDP-Unicast. Dieses Paket wird auf nun bekanntem Weg zum Ziel geleitet: OFP_Packet_In an den Controller, OFP_Packet_Out mit Steuerungsentscheidungen zurück an den Switch, ggf bei mehreren Switches bis es den Publisher erreicht. Die SD des Producers sendet nach Anmeldung des Consumers ein SubscribeEventgroup ACK als UDP-Unicast - ein letztes Mal produzieren die Service Discoveries die OFP_Packets bei Switches.

Dieser Subscriber Endpoint erhält nun über die Eventgroup sämtliche Nachrichten des Publisher Endpoint. Diese Nachrichten werden nun auf Basis des SOME/IP Protokolls erstellt und sobald die Verknüpfung steht, wird für die Kommunikation der Eventgroup ein Flow-Table-Entry erzeugt, d.h. die Switches wissen wie sie die Pakete routen sollen. Der Controller kann diese Regeln jederzeit ändern, aber wird vorerst nicht mehr um Steuerungsregeln gebeten. Die Nachricht des Controllers um Einträge der Flow-Table zu erzeugen oder sie zu ändern werden FlowMod-Einträge (für Flow Modifikation) genannt. Bei dem Transfer der Kontrolle über ein Paket an den Controller spricht man von einem packet-in event.(Manche Switches haben die Möglichkeit die ankommenden Pakete zu buffern, dann würden sie nur Teile des Paket-Headers und eine buffer ID an den Controller senden. Diese Möglichkeit wird in dieser Arbeit explizit nicht berücksichtigt.)

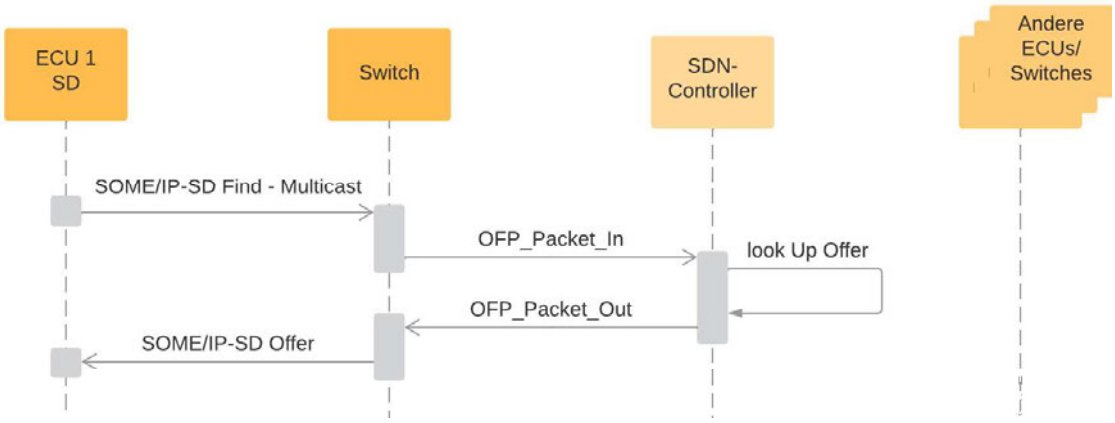

Abbildung 4.3: Der SDN-Controller antwortet mit dem Offer auf den gesuchten Service

4.2.2 Beispielsequenz einer Find-Nachricht mit einer ServiceDiscovery

Abb. 4.3 zeigt die gleiche Anfrage wie im vorherigen Beispiel mit dem Unterschied, dass Requests und Offer nun in einer Service Discovery verarbeitet werden. Wir betrachten das Netzwerk wieder bei Systemstart, d.h. die Flusstabellen der Switches beinhalten noch keine Einträge.

Die Service Discovery des Consumers ECU1 sendet eine Find-Nachricht. Diese Nachricht wird als UDP-Multicast versendet. Sie erreicht den Switch und für dieses Dienstangebot existiert im Switch noch kein flow table entry. Daher kapselt der Switch das Find in ein OpenFlow-Paket vom Typ OFP_Packet_In und sendet es an den SDN-Controller um Steuerungsbefehle zu erhalten, bzw die Kontrolle des Pakets an den Controller zu übertragen. OpenFlow nutzt für die Übertragung TCP, somit wird das anfängliche UDP-Paket nun in einem TCP-Paket gekapselt. Bei Erhalt des Pakets verarbeitet der SDN-Controller nun das Paket. In diesem Fall ist bereits ein Offer eines Publishers eingegangen und gecached worden. Resultierend daraus ist der SDN-Controller nun in der Lage das eingehende Find direkt zu beantworten. Das OFP_Packet_Out enthält somit einerseits das Offer und andererseits den Steuerungsbefehl das Paket direkt an ECU1 weiter zu leiten.

Das Offer kommt beim ECU1 an und triggert eine Anmeldung bei der Eventgroup des Anbietenden. Folglich wird ein SubscribeEventgroup an den Serviceanbieter versendet, dies geschieht wieder als als UDP-Unicast und erreicht den Controller wieder als OFP_-Packet_In. Nun verarbeitet der Controller zweierlei. Als erstes verwaltet er sein subscriptionTable, er updated seinen Eintrag wenn er schon vorhanden ist oder fügt den

neuen Eintrag hinzu. Als zweites erzeugt er den gleichen SubscribeEventGroupEntry und versendet ihn an den Publisher. Dies ist notwendig, da dem Controller die Rechte vorbehalten sind eingehende Pakete zuerst zu verarbeiten und dann als Originalpaket weiter zu senden.

Dieses Paket wird auf bereits bekanntem Weg zum Ziel geleitet: OFP_Packet_Out mit Steuerungsentscheidungen zurück an den Switch, ggf bei mehreren Switches bis es den Publisher erreicht. Die SD des Producers sendet nach Anmeldung des Consumers ein SubscribeEventgroup ACK als UDP-Unicast. Auch dieses Ack wird beim Erreichen des Controllers in den subscriptionTable eingetragen und mit einem installFlowForSubscription an den/die Switch/es gesendet. Somit wird auch diese Kommunikation wieder an das SOME/IP Protokoll übergeben und damit auch der Nachrichtenaustausch betrieben, einerseits zyklisch und andererseits auch RPC-Aufrufe.

4.3 Die Erweiterung des SDN-Controllers

Aus den Nachrichtensequenzen der Beispiele ergeben sich folgende Aufgaben: Als erstes die Implementation einer Erweiterung des SDN-Controller, eine Service Discovery App. Diese App muss ankommende OpenFlow-Pakete empfangen und SomeIpSDHeader identifizieren und verarbeiten. In den Headern gekapselte Service Offer werden gespeichert und Service Requests beantwortet, d.h. als zweites muss eine Struktur zum Cachen der Nachrichten implementiert werden. Sollte kein Offer zu einem Request vorliegen, wird diese Anfrage in diesem Chache gespeichert und eine Find-Nachricht seitens der App gesendet (Der Initialanfragende wird aus dem Multicast ausgenommen). Wird dieses Find mit einem eingehenden Angebot beantwortet, wird die Liste der unbeantworteten Anfragen durchsucht. Alle Treffer werden mit dem Offer beantwortet und danach aus dem Cache entfernt. Um die erforderlichen Nachrichten zu erstellen muss ein Konstruktor für OFP_Packet_Out vorhanden sein.

Es folgt die Verarbeitungsmöglichkeit von Eventgroup-Entries in einem subscriptionTable. Hier wird zusätzlich noch die Option für den installFlowForSubscription benötigt.

4.3.1 Mögliche Zustände des Controllers

Die ServiceInstanceTable: Der Cache wird als ServiceInstanceTable realisiert und besteht aus zwei verschachtelten HashMaps. Eingehende Offer sollen in diesem ServiceTable gespeichert werden. Bei eingehenden Requests wird diese Mapstruktur durchsucht, ob die Kombination aus angefragter ServiceID und InstanceID vorhanden ist. Die HashMaps sind aufgrund der schnellen Zugriffszeiten und schneller Einfüge- und Entfernen-Operationen ideal geeignet.

Letztendlich soll eine Struktur gespeichert werden, welche die Informationen zu einem einzelnen Offer mit zugehörigen Optionen enthält(ein Service-Entry) um eine ServiceInstance abzubilden. Diese Struktur wird als Wert einer InstanceMap gespeichert. Mit der InstanceID als Schlüssel wird eine Map implementiert, welche alle verfügbaren Instances eines Services beinhaltet. Diese Map ist nach den InstanceIDs sortiert.

Die zweite HashMap, die ServiceInstanceMap, beinhaltet nun als Schlüssel die ServiceID und als Wert die InstanceMap. Ein Vorteil dabei ist, dass der Suchvorgang abgeschlossen wird, sobald keine ServiceID in dem ServiceTable vorhanden ist. Ein weiterer Vorteil ist, dass häufig alle Instances eines Services angefragt werden, woraufhin der Client die Instance mit seiner höchsten Priotität wählt. Zur Beantwortung wird über die InstanceMap iteriert und schnell eine Liste erstellt mit allen verfügbaren Instances zu einer ServiceID.

Die RequestMap: Ein weiterer Cache wird für die Find-Nachrichten zur Verfügung gestellt. Mit der ServiceID als Schlüssel und einer Liste unbeantworteter Finds als Wert. Hintergrund ist die Reaktion des Controllers auf Requests die er nicht beantworten kann. In diesem Fall übernimmt der Controller selbst die Rolle eines Clients, anstatt den Mutlicast des Anfragenden zu gestatten. Die Request wird gespeichert und eine eigene Find-Nachricht erstellt. Daraus ergibt sich, dass Offer-Antworten, welche direkt an den Controller gerichtet sind immer eine Antwort auf bisher offene Requests sind. Spätestens jetzt kann der Controller offene Requests beantworten.

Natürlich kann es auch sein, dass zwischenzeitlich ein Multicast-Offer eingegangen ist. Daher wird bei jedem Offer geprüft, ob hierzu ein Eintrag in der RequestMap vorliegt.

Einträge in der RequestMap werden bei Beantwortung gelöscht.

Durch die verschiedenen Herangehensweisen beim Empfangen von Find- und Offer-Nachrichten müssen bei der Implementation alle Kombinationen aus bekannten und

Fall		Service	Instance	Request
1	find empfangen	unbekannt	unbekannt	vorhanden
2		bekannt	bekannt	vorhanden
3		bekannt	unbekannt	vorhanden
4		unbekannt	unbekannt	nicht vorhand
5		bekannt	bekannt	nicht vorhand
6		bekannt	unbekannt	nicht vorhand
7	offer empfangen	unbekannt	unbekannt	vorhanden
8		bekannt	bekannt	vorhanden
9		bekannt	unbekannt	vorhanden
10		unbekannt	unbekannt	nicht vorhand
11		bekannt	bekannt	nicht vorhand
12		bekannt	unbekannt	nicht vorhand

Abbildung 4.4: Mögliche Zustände des SDN-Controllers beim Empfangen einer SD-Nachricht

unbekannten ServiceID, InstanceIDs und Requests berücksichtigt werden. Abb. 4.4 gibt eine Übersicht darüber.

Zu Testzwecken wird hier definiert, wie sich der Controller bei diesen zwölf Fällen jeweils zu verhalten hat.

1. Fall: Die Serviceanfrage wurde vom ECU wiederholt - der anbietende Knoten ist nicht bereit/hat noch nicht geantwortet.

   Reaktion des Controllers: Eigene Serviceanfrage wird erneut versendet.

2. Fall: Ein Requesteintrag wird beim Beantworten gelöscht. Daher kann dieser Fall nicht eintreten.

   Reaktion des Controllers: Serviceangebot würde erneut gesendet.

3. Fall: Siehe Fall 1, Serviceanfrage wurde wiederholt - ein anbietender Knoten dieser spezifischen InstanceID ist nicht bereit/hat noch nicht geantwortet.

   Reaktion des Controllers: Eigene Serviceanfrage wird erneut gesendet.

4. Fall: Die Serviceanfrage wurde vom ECU initial gesendet - anbietende Knoten sind noch nicht bereit.

Reaktion des Controllers: Eigene Serviceanfrage wird gesendet und ECU-Serviceanfrage wird als Request gespeichert.

5. Fall: Die Serviceanfrage wurde vom ECU initial gesendet - ein anbietender Knoten hat diesen Service bereits angeboten.

   Reaktion des Controllers: Serviceangebot wird gesendet und Serviceanfrage wird verworfen.

6. Fall: Die Serviceanfrage wurde vom ECU initial gesendet - ein anbietender Knoten der spezifischen InstanceID ist nicht bereit.

   Reaktion des Controllers: Eigene Serviceanfrage wird gesendet und ECU-Serviceanfrage wird als Request gespeichert.

7. Fall: Ein Serviceangebot wurde als Antwort auf eine Serviceanfrage gesendet.

   Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und Requests werden beantwortet und dann gelöscht.

8. Fall: Siehe Fall 2 - Ein Serviceangebot wird im ServiceTable eingetragen und Requests beantwortet und gelöscht, daher dürfte keine Request bei bekannten Services existieren.

   Reaktion des Controllers: Serviceangebot würde im ServiceTable gespeichert und Requests würden beantwortet und dann gelöscht werden.

9. Fall: Ein Serviceangebot mit noch unbekannter InstanceID wurde als Antwort auf eine Serviceanfrage gesendet oder der Publisher ist gerade (wieder) bereit.

   Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und Requests werden beantwortet und dann gelöscht.

10. Fall: Ein Serviceangebot wurde initial versendet.

    Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und der Multicast wird per OFP_Packet_Out fortgesetzt.

11. Fall: Ein Serviceangebot wurde wiederholt gesendet.

    Reaktion des Controllers: Serviceangebot wird im ServiceTable aktualisiert und ein eigenes Serviceangebot per Multicast versendet.

12. Fall: Ein Serviceangebot einer bisher unbekannten InstanceID wurde initial gesendet.

    Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und ein eigenes Serviceangebot per Multicast versendet.

4.4 Konzeptionelle Vorgaben zur Implementierung

Aus den aufgeführten Anforderungen und Spezifikationen ergeben sich viele Vorgaben für die Implementierung. In diesem Kapitel werden diese Vorgaben und Entscheidungen aufgezählt und erklärt. Abb 4.5 zeigt eine Übersicht, welche Pakete in welcher Reihenfolge in dem eingehenden OpenFlow-Paket gekapselt sind. Zusätzlich sind die für das Konzept wichtigen Informationen der Pakete aufgeführt.

Der Controller versendet eigene Find-Nachrichten, dazu müssen wir aus den Grundlagen den [PRS_SOMEIPSD_00158] beachten und für die SessionIDs einen Zähler initialisieren, der nicht Null werden darf.

Zur Beantwortung der Nachrichten oder ihrer Weiterleitung wird der Eingangsport des Switches benötigt, an welchem die Nachricht angekommen ist. So können einerseits die Steuerung der Pakete als Unicast-Antwort realisiert werden und andererseits kann dieser Port bei Multicasts exkludiert werden. Die Switch-Informationen erhalten wir durch eine Funktion des Controllers.

Die Speicherung der Offer-Nachrichten verlangt mehrere Faktoren. Zum Ersten muss die ServiceInstance gespeichert werden können. Zum Zweiten wird diese ServiceInstance in einer InstanceMap verwaltet, mit der InstanceID als Schlüssel. Zum Dritten wird eine ServiceInstanceMap benötigt, welche als Schlüssel die ServiceID und als Wert die InstanceMap hat. Diese Struktur soll eine schnelle Suche nach Wichtigkeit und Vorkommen ermöglichen. Der Zugriffspunkt ist die ServiceID, sollte sie nicht in der Map vorhanden sein, kann die Suche abgebrochen werden. Wenn die ServiceID vorliegt, wird nach der benötigten InstanceID gesucht, bzw eine Liste aller InstanceIDs erstellt.

Die ServiceInstance besteht aus dem gesamten ServiceEntry des eingegangenen Offer und der Liste der zugehörigen Optionen, der optionList. Die ServiceInstance soll um eine Komponente von layeredInformation erweitert werden.

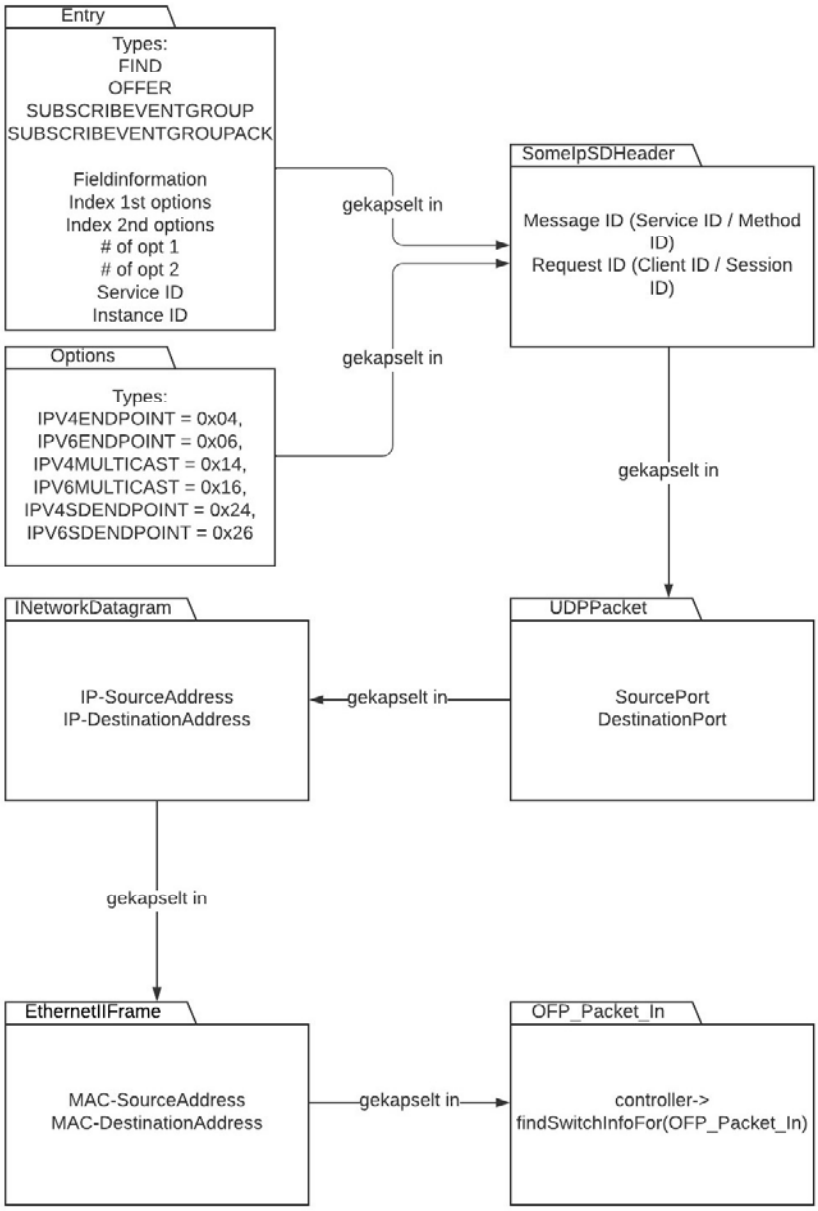

Abbildung 4.5: Reihenfolge der Paketkapselung und wichtige Informationen

Hintergrund: Die benötigten Informationen zur Bearbeitung und Beantwortung der Nachrichten sind auf viele gekapselte Pakete aufgeteilt. Es wird also eine Möglichkeit benötigt, diese Informationen zu sammeln, hierfür bestehen mehrere Optionen. Nach Abwägung der Vor- und Nachteile ist die Entscheidung zugunsten einer ControlInfo getroffen worden. Mit der Funktion setControlInfo ist es möglich eine Datenstruktur an eine Nachricht oder ein Objekt anzuheften. In diesem Fall heften wir die layeredInformation an den SOME/IP-SD-Header an. Der große Vorteil besteht darin, dass die zugehörigen Informationen direkt an das Objekt angehängt werden, welches im Zuge der Verarbeitung immer als Parameter übergeben wird. D.h. wiederum, es besteht nicht die Notwendigkeit die Anzahl der Parameter in den Funktionen zu erhöhen. Das sorgt für eine bessere Lesbarkeit des Codes einerseits und andererseits müssen vorhandene Klassen nicht mit einer Erweiterung versehen werden um die Information tragen zu können. Diese control info wird häufig genutzt, wenn Pakete zwischen Protokollschichten übergeben werden. Die control info Objekte werden beim Löschen des Trägerobjekts mit gelöscht, das heißt auch hier muss man sich um die Reinheit des Codes zur Laufzeit keine Gedanken machen.

Zu beachten ist, dass nur eine control info angeheftet werden kann, sollte versucht werden eine Zweite anzuhängen wird es zu einem Error kommen.

Der Nachteil dieser Entscheidung betrifft das Kopieren des Objekts, hierbei ist zu beachten, dass die control info auf den nullptr gesetzt wird. Es muss darauf geachtet werden die control info explizit zu kopieren und wieder an die Objektkopie anzuhängen.

LayeredInformation sollen folgende, vollständige Informationen zu den eingehende OpenFlow-Paketen enthalten:

Die MAC Quell- und Zieladresse,

die IP Quell- und Zieladresse,

den UDP Quell- und Zielport,

den Eingangsport des Controllers und

die Switch-Informationen des sendenden Switches.

Die Speicherung der Find-Nachrichten soll nur kurzzeitig geschehen. In einer RequestTable werden alle Requests gespeichert, deren Beantwortung aussteht. Zur schnellen Suche wird auch hier eine Map verwendet, mit der ServiceID als Schlüssel und einer Liste aller FindRequests als Wert.

Die Speicherung der Eventgroup-Einträge soll in einer ServiceInstanceSubscriptionList erfolgen. Auch hier werden zwei geschachtelte HashMaps mit den Schlüsseln ServiceID und InstanceID benötigt um eine Liste mit Eventgroup-Verbindungen speichern zu können.

Verarbeitung der Einträge im SomeIpSDHeader: Vorerst sollen die vier verschiedenen Typen der Entries verarbeitet werden, wie in Abb 4.5 dargestellt.

Fall 1: FIND-Eintrag

Fall 2: OFFER-Eintrag

Fall 3: SUBSCRIBEVENTGROUP-Eintrag

Fall 4: SUBSCRIBEVENTGROUPACK-Eintrag

Hierzu soll eine Liste der vorhandenen Einträge erstellt und die Einträge nacheinander abgearbeitet werden. Die Einträge müssen hierbei zusammen mit dem SomeIpSDHeader als Parameter übergeben werden, da die Optionen zu den Einträgen in diesem gekapselt sind. Der Einsatz einer Dispatch-Tabelle eignet sich hier um eine klare Struktur in den Code zu bringen. Die Vorteile dieser Struktur liegen in der Erweiterbarkeit, sollten neue Fälle hinzugefügt werden und der Austauschbarkeit, wenn verarbeitende Funktionen überarbeitet werden, können sie in der zugehörigen Switch-Anweisung leicht ausgetauscht werden. Somit wird der Code leichter wartbar.

Fall 1: FIND-Eintrag Ein Find soll mit einer Liste von ein oder mehreren Offern beantwortet werden. Wurde eine bestimmte InstanceID angefragt, so enthält diese Liste nur das Ergebnis zu dieser InstanceID. Sonst werden alle vorhandenen Einträge zu der ServiceID in der Liste übergeben. Sollte kein Eintrag der ServiceID in der ServiceInstanceMap vorhanden sein, wird diese Anfrage einerseits im RequestTable gespeichert und andererseits dieses Find als Anfrage des Controllers versendet.

Fall 2: OFFER-Eintrag Offer werden zyklisch versendet, wie in den Grundlagen beschrieben. Da die Spezifikation dies vorgibt muss auch der Controller die zyklische Weiterleitung umsetzen. Für mögliche Änderungen ist ein boolscher Wert einzusetzen für forwardOfferMulticast. Mit diesem Schalter könnte man zukünftig als überflüssig identifizierte Multicasts aussetzen.

Aus der Umsetzung der Spezifikation ergibt sich nun als erstes eine Weiterleitung des Multicasts. Hierzu setzen wir den Controller als Absender ein, damit die getriggerten

SubscribeEventgroup-Einträge direkt zur Bearbeitung an den Controller gesendet werden und somit das Netzwerk weiter entlastet wird. Als zweites werden vom Controller gesendete Offer direkt gedropt. Als drittes sollen einerseits die ServiceInstanceMap geupdated werden und andererseits offene Requests des RequestTables beantwortet werden.

Fall 3: SUBSCRIBEVENTGROUP-Eintrag Die SubscribeEventgroup-Einträge müssen eindeutige Informationen zu spezifischer ServiceID und InstanceID, sowie zu der genutzten EndpointOption enthalten. Dies ist zu prüfen bevor die subscription in der SubscriptionTable eingefügt wird, mit dem Verweis auf ein ausstehendes acknowledgement. Ein SubscribeEventgroup mit den Daten des Absenders und den Daten des Offers in der ServiceInstanceMap soll erstellt und versendet werden.

Fall 4: SUBSCRIBEVENTGROUPACK-Eintrag Für die SubscribeEventgroupAck-Einträge werden genutzt um die ausstehenden acknowledgement in der SubscriptionTable zu ändern. Für die zukünftige Kommunikation zwischen dem Consumer und dem Provider muss den Switches ein FlowTable-Eintrag der beiden Parteien übergeben werden.

5 Implementierung

Die Implementierung ist in eine Simulationsumgebung der CoRE-Group der HAW-Hamburg [5] eingebettet. Als Ausgangspunkt diente das INET Framework für ereignisbasierte Simulationen in Ethernet-Netzwerken innerhalb der Simulationsumgebung OMNeT++[23]. Hieraus hat sich ein beeindruckendes Werk aus Frameworks zur Echtzeitsimulation von Technologien in der Automobilbranche entwickelt. Unter anderem stehen die Frameworks SDN4CoRE (Software-Defined Networking for Communication over Realtime Ethernet) und OpenFlow zur Verfügung, welche zu der Idee inspiriert haben, den SDN-Controller um eine Service Discovery zu bereichern. Zur besseren Speicherverwaltung sollen möglichst Zeiger auf die Objekte verwendet werden.

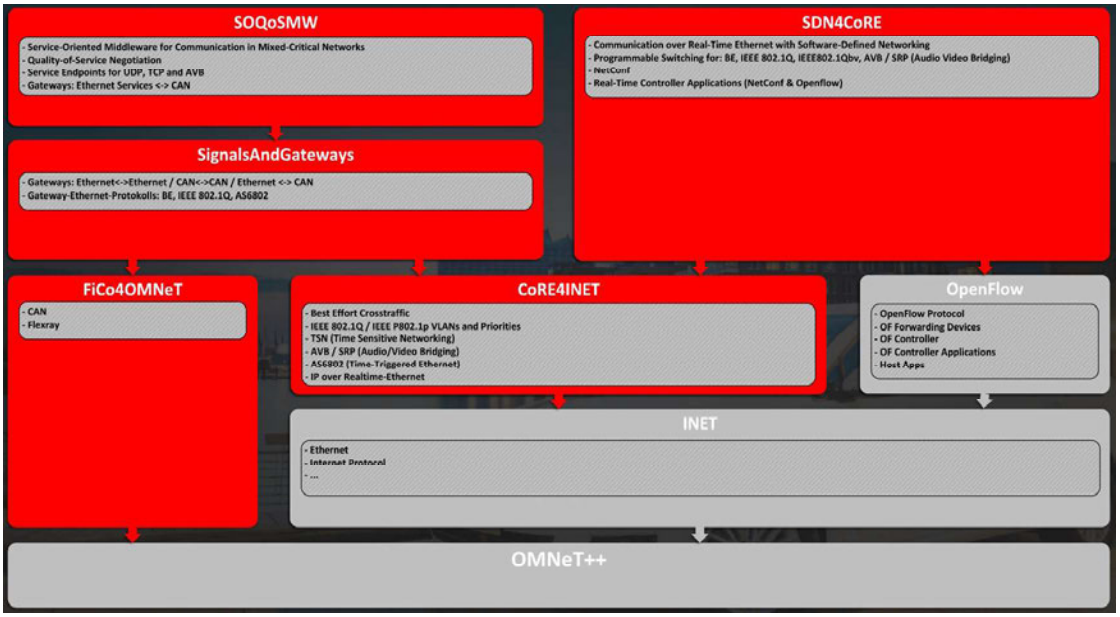

Abbildung 5.1: Framework-Übersicht [5]

5.1 Umsetzung des Konzepts

Grundlegend wird eine Service Discovery als Anwendung in dem SDN4CoRE-Framework implementiert. Die Umsetzung des Konzepts soll hier schrittweise erklärt werden:

Als erstes werden die geforderten Caches bereitgestellt. Jeder angebotene Service wird durch eine ServiceID und eine InstanceID identifiziert. Für eine schnelle Suche werden zwei HashMaps verwendet. Die erste mit der ServiceID als Schlüssel und der zweiten HashMap als Wert. Die zweite HashMap hat als Schlüssel die InstanceID und relevante Informationen als Wert. Diese Map wird als ServiceInstanceMap initialisert und als ServiceTable benannt. Jedes eintreffende Angebot von Services wird In dieser ServiceTable gespeichert. Der Zugriff auf folgende interne Objekttypen bleibt dem Controller vorbehalten und wird in der Header Datei der App als protected gekennzeichnet und wie folgt benannt:

ServiceInstanceMap serviceTable: Nach ServiceIDs sortierte Map mit der InstanceMap als Wert.

RequestMap requestTable: Kurzfristige Speicherung aller unbeantworteten Find-Nachfragen.

SubscriptionMap subscriptionTable: Übersicht über aktive Publish/Subscriber-Verkettungen.

LayeredInformation myLayeredInformation: die Controller-Informationen in Form der LayeredInformation(Erklärung folgt).

uint16_t controllerRequestID: Der Zähler für die selbsterstellten Nachrichten des Controllers.

Als zweites wird die Filterung der SOME/IP-SD-Nachrichten umgesetzt. Hierzu werden eingehende OpenFlow-Pakete verarbeitet.

SomeipSDControllerApp::processPacketIn(OFP_Packet_In packet_in_msg)*

Diese Funktion entkapselt den SomeIpSDHeader aus dem OFP_Packet_In. Die Reinhefolge der Kapselung ist in der Abb. 4.5 des Konzepts übersichtlich dargestellt. Der SomeIpSDHeader wurde im UDPPacket gekapselt, dieses in dem INetworkDatagram, dieses wiederum im EthernetIIFrame und dieser letztendlich im OFP_Packet_In.

Vorweg werden die Switch-Informationen des sendenden Switches vom Controller abgerufen. Danach werden die verschiedenen Layer entkapselt und um Versionskonflikte

auszuschließen auf die im Konzept vorgeschlagenen Versionen gecastet. Mit der Kontrollstruktur der bedingten Anweisung, werden alle Pakete nicht weiter verarbeitet, die nicht unseren Kriterien entsprechen. Damit wird sichergestellt, dass wir nur die SomeIpSDHeader verarbeiten. Somit ist die Forderung des Konzepts nach Filterung der Header erfüllt. Des Weiteren werden auch Pakete verworfen, welche vom Controller selber erstellt wurden und der HostTable des Controllers geupdated.

Die geforderten Informationen der layeredInformation werden zusammengetragen und an den SomeIpSDHeader mit setControlInfo angeheftet. Jeder valide SomeIpSDHeader ist nun mit umfassenden Informationen bestückt und wird der nächsten Funktion verarbeitet.

void SomeipSDControllerApp::processSomeIpSDHeader(SomeIpSDHeader someIpSDHeader)*

Die zu verarbeitenden SomeIpSDEntries liegen gekapselt in dem SomeIpSDHeader als Liste vor, diese wird zur Verarbeitung entkapselt. Über diese Liste wird iteriert und jeder Eintrag nacheinander bearbeitet. Dabei werden die verschiedenen Typen in verschiedenen Funktionen verarbeitet. Die Dispatch-Tabelle wird als Datenstruktur umgesetzt um die Funktionen basierend auf dem Typ des SomeIpSDEntry auszuführen. Der Steuerungsmechanismus Switch wird für die vier Fälle FIND, OFFER, SUBSCRIBEVENTGROUP und SUBSCRIBEVENTGROUPACK angewendet und je nach Typ werden die weiterverarbeitenden Funktion innerhalb eines cases aufgerufen.

void SomeipSDControllerApp::processFindEntry(SomeIpSDEntry findInquiry, SomeIpSDHeader* someIpSDHeader)*

Diese Funktion verarbeitet ankommende Anfragen wie folgt: Vorweg werden die LayeredInformation aus der ControlInfo des SomeIpSDHeader ausgelesen, da sie für die Verarbeitung benötigt werden. Danach wird unsere ServiceInstanceMap nach passenden Einträgen zu ServiceID und InstanceID durchsucht und diese in einer Liste gesammelt. Hiernach wird als erstes der Fall abgefangen, dass ein vom Controller gesendetes Find eingegangen ist. Wenn die Liste Einträge haben sollte, müssen diese Einträge nach dem Senden des Finds eingegangen sein und damit auch beantwortet. Bei Eingang eines Offers werden offene Requests beantwortet. Sollte die Liste weiterhin leer sein, simulieren wir die Repetition-Phase und senden das Find erneut als Multicast.

Der zweiter Fall beschreibt den Umstand das kein Offer zu ServiceID und InstanceID vorliegt, d.h. die Liste ist leer und der Absender ist eine ECU. Der Controller erstellt

daraufhin ein neues Find mit den wichtigen Feldern aus seinen eigenen LayeredInformation. Die Informationen des Absenders sind im eingegangen someIpSDHeader enthalten. Somit können wir einen Multicast der Find-Nachricht versenden und den Port des Absender-ECUs ausnehmen. Seine Request wird daraufhin inklusive der Optionen in der RequestTable eingefügt.

Der dritte und letzte Fall ist die schöne Möglichkeit das Find beantworten zu können. Ein Offer wird erstellt und an den Anfragenden ECU per Unicast gesendet.

void SomeipSDControllerApp::processOfferEntry(SomeIpSDEntry offerEntry,SomeIpSDHeader* someIpSDHeader)*

Diese Funktion verarbeitet ankommende Angebote wie folgt: Vorweg werden die LayeredInformation aus der ControlInfo des SomeIpSDHeader ausgelesen, da sie für die Verarbeitung benötigt werden. Der eingesetzte Boolean für forwardOfferMulticast wird bei der Initialisierung auf true gesetzt. Der eingegangene SomeIpSDHeader wird dupliziert, die vorhandene ControlInfo wird gelöscht und durch die LayeredInformation des Controllers ersetzt. Die Nachricht wird per Multicast gesendet und dabei die eigentliche Quelle ausgenommen. Sollte diese Nachricht vom Controller stammen, so wird aus der Funktion direkt returned.

Danach wird die ServiceInstanceMap geupdated, dabei wird ein vorhandener Eintrag durch den jüngeren Eintrag ersetzt und neue, zugehörige Optionen hinzugefügt. Neue Einträge werden hinzugefügt, entweder in die InstanceMap einer vorhandenen ServiceID oder als neuer Eintrag in der ServiceInstanceMap.

Als letztes wird die RequestTable durchsucht, ob mit diesem Offer eine Anfrage beantwortet werden kann.

SomeipSDControllerApp::processSubscribeEventGroupEntry(SomeIpSDEntry entry,SomeIpSDHeader* someIpSDHeader)*

Diese Funktion verarbeitet ankommende SubscribeEventGroup wie folgt: Vorweg werden die LayeredInformation aus der ControlInfo des SomeIpSDHeader ausgelesen, da sie für die Verarbeitung benötigt werden. In der ServiceInstanceMap wird nun nach einer exakten Übereinstimmung der ServiceID, der InstanceID und den EndpointOptions gesucht. Bei Erfolg wird die SubscriptionTable verwaltet um den Status der verschiedenen Abonnements für Dienste zu verfolgen. Bei einer neuen Subscription wird ein neuer

Eintrag hinzugefügt und schließlich eine SubscribeEventGroup-Anfrage mit den Empfängerdaten des Offers erstellt und an den Publisher des Dienstes gesendet. Bei diesem neuen Eintrag wird ein Flag waitingForAck auf true gesetzt um zu kennzeichnen, dass der Anmeldevorgang noch nicht abgeschlossen ist.

void SomeipSDControllerApp::processSubscribeEventGroupAckEntry(SomeIpSDEntry entry, SomeIpSDHeader* someIpSDHeader)*

Diese Funktion verarbeitet ankommende SubscribeEventGroupAck wie folgt: Vorweg werden die LayeredInformation aus der ControlInfo des SomeIpSDHeader ausgelesen, da sie für die Verarbeitung benötigt werden. In der SubscriptionTabel wird nach der Subscription für die im SubscribeEventGroupAck bestätigten ServiceID und InstanceID gesucht. Diese Suche wird eingegrenzt auf Einträge, in denen das Flag waitingForAck auf true gesetzt ist. Mit einem installFlowForSubscription werden die Regeln zur Datenweiterleitung den beteiligten Switches mitgeteilt. Somit bleibt nur noch, dass der Subscriber informiert wird, daher wird eine neue SubscribeEventGroupAck-Nachricht mit allen nötigen Daten erzeugt und versendet. Danach vermerkt der Controller in der SubscriptionTabel diese Eventgroup-Kombination als aktiv.

5.2 Hilfsfunktionen

Um die Verständlichkeit und Wartbarkeit zu gewährleisten wurden viele häufig genutzte Funktionen ausgelagert. So wurden die Konstruktoren der einzelnen Service Discovery-Nachrichten buildFind, buildOffer, buildSubscribeEventGroup und buildSubscribeEventGroupAck in eigene Funktionen ausgelagert. Desweiteren wurden die Verwaltungen der benötigten Tabellen in eigene Funktionen kodiert, sowie die Verwaltung der Optionen zu den Einträgen.

Exemplarisch wird das buildFind vorgestellt:

SomeIpSDHeader SomeipSDControllerApp::buildFind(SomeIpSDHeader* findSource, SomeIpSDEntry* findInquiry)*

SomeIpSDHeader header = new SomeIpSDHeader(SSOME/IP SD - FIND");*

Es wird neues Objekt vom Typ SomeIpSDHeader erstellt, das für den Aufbau der Find-Anfrage verwendet wird. Der SomeIpSDHeader ist die zu versendende Datei, welche den Eintrag und zugehörige Optionen kapselt.

header->setRequestID(controllerRequestID++);

Die Funktion verwendet eine Request-ID (controllerRequestID), die für jede SomeIpSD-Nachricht inkrementiert wird.

SomeIpSDEntry newFind = findInquiry->dup();*

Ein neuer Find-Eintrag wird erstellt, indem die eingegangene Nachricht dupliziert wird. Dies ist notwendig, da der Controller nicht die Rechte an der eingegangenen Nachricht besitzt und somit keine Änderungen vornehmen darf. Die Nachricht soll vom Controller gesendet werden und muss daher folgende Werte ändern.

int optionQuantity = findInquiry->getNum1stOptions();

newFind->setIndex1stOptions(0);

newFind->setNum1stOptions(optionQuantity);

Die Anzahl der Optionen des eingegangenen Finds werden ausgelesen und in den Eintrag übernommen.

if (controllerRequestID == 0) controllerRequestID = 1;

[PRS_SOMEIPSD_00159] Die Session-ID darf nicht 0 sein, daher wird hier sie auf 1 gesetzt, falls die Request-ID 0 ist.

header->encapEntry(newFind);

Das neue Find wird in dem SomeIpSDHeader gekapselt.

std::list<SomeIpSDOption> oldOptionList = getEntryOptions(findInquiry, findSource);*

*for (auto it = oldOptionList.begin(); it != oldOptionList.end(); it++) header->encapOption(*it);*

Die Options aus der ursprünglichen Find-Anfrage werden in den Header der neuen Find-Anfrage eingefügt.

return header;

Schließlich wird der erstellte SomeIpSDHeader zurückgegeben.

Ähnlich werden die anderen Einträge aufgebaut und an die aufrufende Funktion zurückgegeben. Weitere wichtige Funktionen sind die Verwaltung der Maps, wie:

list<SomeipSDControllerApp::ServiceInstance> SomeipSDControllerApp::lookUpServiceInMap(uint16_t requestedServiceId, uint16_t requestedInstanceId)

Diese Funktion erstellt eine Liste von SericeInstances aus der ServiceInstanceMap und unterscheidet, ob eine bestimmte Instance oder alle Instances einer ServiceID angefragt wurden.

void SomeipSDControllerApp::updateServiceTable(ServiceInstance& newInfo)

Diese Funktion aktualisiert Service- und Instance-Einträge oder fügt sie hinzu. Bei vorhandenen Einträgen werden vorhandene Optionen überprüft und aktualisiert und die alte Instanz wird bereinigt. Dabei werden die neuen Informationen übernommen und alte Optionen hinzugefügt, wenn sie fehlen sollten.

list<SomeIpSDOption> SomeipSDControllerApp::getEntryOptions(SomeIpSDEntry* xEntry, SomeIpSDHeader* header)*

Diese Funktion übernimmt die knifflige Aufgabe die Optionen der Einträge aus der Optionsliste herauszufiltern. In den Einträgen wird auf den Index und die Anzahl der Optionen verwiesen. Hier wird dafür gesorgt, dass eine neue Optionsliste erstellt und nur mit den Optionen des Eintrags gefüllt wird.

void SomeipSDControllerApp::sendOffer(SomeIpSDHeader offer, SomeIpSDHeader* findSource, LayeredInformation* infoFind, LayeredInformation* infoOffer)*

Diese Funktion sendet Unicast-Offer als Antwort auf Find-Nachrichten. Alle anderen Nachrichten sind Multicasts und werden mit folgender Methode gesendet:

void SomeipSDControllerApp::sendMulticast(SomeIpSDHeader mcSrc, SomeIpSDHeader* mcDst)*

6 Qualitätssicherung

Im Konzept sind die zwölf verschiedenen Fälle aufgeführt in welchen Zuständen sich der Controller befinden kann. Diese Fälle wurden als Grundlage für Konfigurationen des Netzwerkes genutzt, um sie bewusst herbeizuführen. Somit kann die Implementierung automatisiert getestet werden, ob Vorgaben aus Konzept und Spezifikationen korrekt umgesetzt wurden. Einige Fälle können während des normales Betriebes des Netzwerks eigentlich nicht auftreten, dies wird hier bestätigt. Die eingehenden Entries der Service Discovery-Nachrichten sind wie eine Batch-Datei und werden nacheinander abgearbeitet. So können keine Effekte entstehen, die bei einer Parallelverarbeitung zu Komplikationen führen. Im folgenden werden die einzelnen Testfälle anhand der Testkonfiguration, der erwarteten Reaktion des Controllers und des Testergebnisses beschrieben.

[Config case 1]

Testkonfiguration: Ein Subscriber sendet ein Find und wiederholt dies in der Repetitionphase, die Publisher-Dienste werden deaktiviert.

Reaktion des Controllers: Eigene Serviceanfrage wird erneut versendet.

Testergebnis: Der Controller empfängt das erste Find des Subscribers, speichert es in der RequestTable und sendet ein eigenes Find per Multicast mit der RequestID=1. Der Controller empfängt das zweite Find des Subscribers und sendet ein eigens Find per Multicast mit der RequestID=2.
Der Controller reagiert, wie gefordert.

[Config case 2]

Testkonfiguration: Dieser Zustand ist nicht erreichbar, da im Controller nur ein Ereignis zur Zeit verarbeitet wird, sollte es ein Find sein und ServiceID und InstanceID bekannt, so wird der Eintrag in der RequestTable gelöscht und eine Offer-Antwort verfasst. Sollte es ein Offer ein, so wird der Eintrag in der RequestTable gelöscht und eine Offer erstellt.

[Config case 3]

Testkonfiguration: Ein Subscriber sendet ein Find mit einer speziellen InstanceID und wiederholt dies in der Repetitionphase, ein Publisher-Dienst mit einer anderen InstanceID wird aktiviert.

Reaktion des Controllers: Eigene Serviceanfrage wird erneut gesendet.

Testergebnis: Der Controller empfängt das erste Find des Subscribers mit der InstanceID=1 und sendet ein eigenes Find per Multicast mit der RequestID=1. Der Controller empfängt das Offer mit der InstanceID=2 und trägt es in die ServiceInstanceMap ein. Der Controller empfängt das zweite Find des Subscribers und sendet ein eigenes Find per Multicast mit der RequestID=2.
Der Controller reagiert, wie gefordert.

[Config case 4]

Testkonfiguration: Ein Subscriber sendet ein Find, die Publisher-Dienste werden deaktiviert.

Reaktion des Controllers: Eigene Serviceanfrage wird gesendet und ECU-Serviceanfrage wird als Request gespeichert.

Testergebnis: Der Controller empfängt das Find des Subscribers, speichert es in der RequestTable und sendet ein eigenes Find per Multicast mit der RequestID=1.
Der Controller reagiert, wie gefordert.

[Config case 5]

Testkonfiguration: Ein Publisher sendet ein ein Offer mit ServiceID und InstanceID. Ein Subscriber sendet ein ein Find mit gleicher ServiceID und InstanceID.

Reaktion des Controllers: Serviceangebot wird gesendet und Serviceanfrage wird verworfen.

Testergebnis: Der Controller empfängt das Offer des Publishers und speichert es in der ServiceInstanceMap. Der Controller empfängt das Find und beantwortet es mit dem Eintrag aus der ServiceInstanceMap. Das Find wird nicht gespeichert und verworfen.
Der Controller reagiert, wie gefordert.

[Config case 6]

Testkonfiguration: Ein Publisher sendet ein Offer mit der InstanceID=1. Ein Subscriber sendet ein Find mit der InstanceID=2. Die ServiceIDs sind gleich.

Reaktion des Controllers: Eigene Serviceanfrage wird gesendet und ECU-Serviceanfrage wird als Request gespeichert.

Testergebnis: Der Controller empfängt das Offer mit der InstanceID=1 und trägt es in die ServiceInstanceMap ein. Der Controller empfängt das Find des Subscribers mit der InstanceID=2. Er kontrolliert die RequestTable, erhält keinen Treffer und speichert somit das Find dort ab. Danach sendet er ein eigenes Find.
Der Controller reagiert, wie gefordert.

[Config case 7]

Testkonfiguration: Ein Subscriber sendet ein Find. Ein Publisher-Dienst sendet ein Offer mit gleicher ServiceID und InstanceID.

Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und Requests werden beantwortet und dann gelöscht.

Testergebnis: Der Controller empfängt das Find des Subscribers mit der und kontrolliert die RequestTable, erhält keinen Treffer und speichert somit das Find dort ab. Danach sendet er ein eigenes Find. Der Controller empfängt das Offer hierzu und speichert es in der ServiceInstanceMap. Er kontrolliert die RequestTable, beantwortet die Anfrage und löscht den Eintrag aus der RequestTable.
Der Controller reagiert, wie gefordert.

[Config case 8]

Testkonfiguration: Dieser Zustand ist nicht erreichbar, da im Controller nur ein Ereignis zur Zeit verarbeitet wird, sollte es ein Find sein und ServiceID und InstanceID bekannt, so wird der Eintrag in der RequestTable gelöscht und eine Offer-Antwort verfasst. Sollte es ein Offer ein, so wird der Eintrag in der RequestTable gelöscht und eine Offer erstellt.

[Config case 9]

Testkonfiguration: Ein Publisher sendet ein Offer mit gleicher Service-ID und InstanceID=1. Ein Subscriber sendet ein Find mit gleicher ServiceID und InstanceID=2. Ein zweiter Publisher sendet ein Offer mit gleicher Service-ID und InstanceID=2.

Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und Requests werden beantwortet und dann gelöscht.

Testergebnis: Der Controller empfängt das erste Offer mit InstanceID=1 und speichert es in der ServiceInstanceMap. Der Controller empfängt das Find mit der InstanceID=2 und kontrolliert die RequestTable, erhält keinen Treffer zu der InstanceID und speichert somit das Find dort ab. Der Controller empfängt das zweite Offer mit InstanceID=2 und speichert es in der ServiceInstanceMap. Er kontrolliert die RequestTable, beantwortet die Anfrage und löscht den Eintrag aus der RequestTable.
Der Controller reagiert, wie gefordert.

[Config case 10]

Testkonfiguration: Ein Publisher sendet ein Offer.

Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und der Multicast wird per OFP_Packet_Out fortgesetzt.

Testergebnis: Der Controller empfängt das Offer und speichert es in der ServiceInstanceMap. Er kontrolliert die RequestTable, wird nicht fündig und beendet die Bearbeitung des Offers.
Der Controller reagiert, wie gefordert.

[Config case 11]

Testkonfiguration: Ein Publisher sendet ein Offer und wiederholt dies in der Repetitionphase.

Reaktion des Controllers: Serviceangebot wird im ServiceTable aktualisiert und ein eigenes Serviceangebot per Multicast versendet.

Testergebnis: Der Controller empfängt das Offer und speichert es in der ServiceInstanceMap. Der Controller empfängt das zweite Offer und updated den Eintrag in der ServiceInstanceMap.
Der Controller reagiert, wie gefordert.

[Config case 12]

Testkonfiguration: Ein Publisher sendet ein Offer mit gleicher ServiceID und der InstanceID=1. Ein zweiter Publisher sendet ein Offer mit gleicher ServiceID und der InstanceID=2.

Reaktion des Controllers: Serviceangebot wird im ServiceTable gespeichert und ein eigenes Serviceangebot per Multicast versendet.

Testergebnis: Der Controller empfängt das erste Offer und speichert es in der ServiceInstanceMap. Der Controller empfängt das zweite Offer und speichert es mit der InstanceID=2 zu der ServiceID in der ServiceInstanceMap.
Der Controller reagiert, wie gefordert.

Anhand dieser Testfälle können neue Implementationen oder Codeanpassungen zielgerichtet auf Fehler kontrolliert werden. Gleichzeitig wurde mit diesem Testdurchlauf gezeigt, dass die bisherige Implementation die gewünschten Verhaltensweisen des Controllers umsetzt. Gerade der minimalistische Ansatz, nur die benötigten Publisher- oder Subscriber-Dienste laufen zu lassen, schafft eine Übersichtlichkeit der zugrunde liegenden Testfrage. Somit wird der Fokus gezielt auf die wichtigen Funktionen geleitet und Ablenkungen minimiert.

7 Evaluation

Die Leistungsfähigkeit des Konzepts mit einer inkludierten Service Discovery wird anhand eines Vergleichs gemessen. Zum einen im Vergleich zu einem herkömmlichen Netzwerk, welches aktuell in Fahrzeugen eingesetzt wird, SOME/IP auf der Basis eines Ethernet-Netzwerks. Zum anderen im Vergleich zu einem Netzwerk, welches nur einen SDN-Controller besitzt. Dabei wird eine steigende Anzahl von 1 bis 50 Publishern und Subscribern genutzt, sowie und eine unterschiedliche Anzahl an Switches (1 bis 5) verglichen. Gemessen wird die vergangene Zeit von der ersten Publisher-Nachricht bis zur letzten Einrichtung der Eventgroup-Verbindung.

Grundlage der Simulation ist die Implementierung sowie der vorgestellte OMNeT++ Simulator[23], mit dem genannten INET-Framework und dem Open-FlowOMNeTSuite[14].

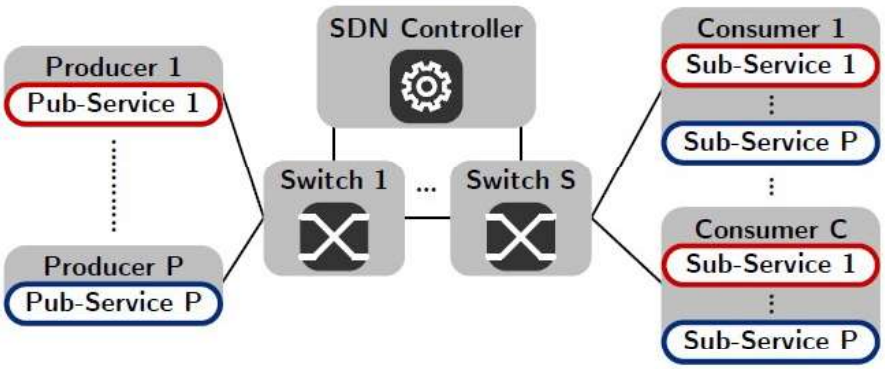

Abbildung 7.1: Topologie für den Skalierbarkeitsvergleich von 1 bis 5 Switches (S), 1 bis 50 Publishern (P) und 1 bis 50 Subscribern (C) mit einem Subscriber pro Publisher[9].

Für den Vergleich wurde die Topologie aus Abb. 7.1 genutzt. Verbunden werden die Knoten durch 1GBit/s Ethernet Leitungen. Es wurden 192 Parameterkombinationen simuliert und die Zeit bis zur Einrichtung aller Subscriptions gemessen. Der SDN-Controller nutzt das OpenFlow-Protokoll um die Switches zu konfigurieren, dies dauert 100 μs[25].

Die gleiche Zeit ist für die Switch-Nachrichtenverarbeitung anzunehmen und hier gilt noch eine Hardwareverzögerung von 8 μs.[9]

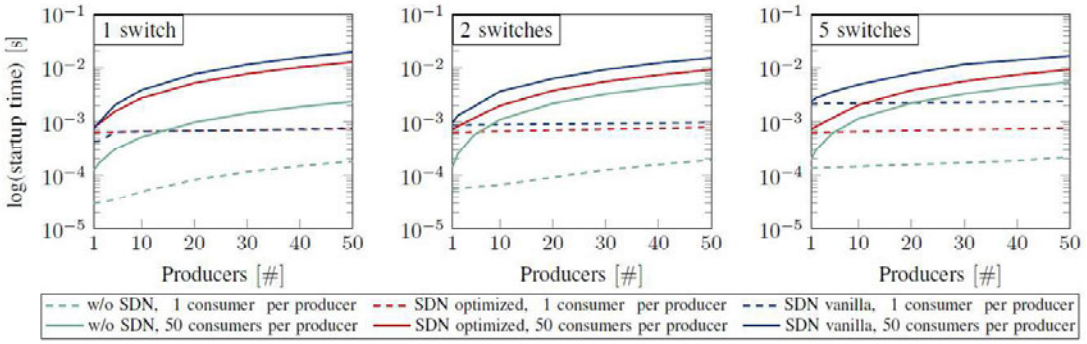

Abbildung 7.2: Vergleich der Netzwerke ohne SDN (w/o SDN), mit SDN und SD(SDN optimized) und SDN ohne SD(SDN vanilla). Die Gesamtzeit aller Subscriptions wird logarithmisch dargestellt. [9].

Abb. 7.2 zeigt die Ergebnisse der Versuchssimulation. Ein SDN-Controller legt die Steuerung für jedes Paket fest, dieser zusätzliche Datentransfer zwischen Switches und SDN-Controller ist hier klar zu erkennen. Bei zwei oder mehr Switches halbiert sich die zusätzliche Verarbeitungszeit mit einer Service Discovery und zunehmenden Publishern. Mit Optimierungen der OpenFlow-Kommunikation zwischen Controller und Switches könnten sich die Zeiten durchaus noch weiter angleichen.

Zwei Werte für einen direkten Vergleich: Bei fünf Switches und einer Subscription benötigen die Netzwerke 0,1 ms(Ethernet), 0,6 ms (SDN mit SD) und 2,2 ms (SDN vanilla) bis die Subscriptions fertiggestellt sind. Bei jeweils 50 Publishern und Subscribern sind es bis dahin 5,5 ms, 9,3 ms und 16,4 ms. Somit haben wir teilweise eine Verbesserung der Leistung um 50% durch den Einsatz einer Service Discovery im SDN-Controller.

Für sicherheitskritische Dienste, wie z.B. Kollisionserkennung könnte die erhöhte Migrationszeit knapp werden, wenn der Dienst vorher nicht bekannt war. Allerdings werden die meisten Dienste beim Systemstart angeboten und gesucht. Hier sollen alle Services in unter 200ms verfügbar sein. Mit einem Wert von 2500 Services in unter 10ms ist die Einrichtungszeit für alle Dienstleistungen, auch sicherheitskritischer Natur, vollkommen akzeptabel.

8 Fazit

In dieser Arbeit wurde eine Verbesserung eines SDN-Netzwerkes durch eine Service Discovery im Controller vorgestellt. Ziel sollte eine bessere Skalierbarkeit und eine verringerte Netzwerklast sein. Die Evaluation zeigt, dass diese Ziele erreicht wurden, sogar mit einer bis zu 50% igen Geschwindigkeitssteigerung. Allerdings ist anzumerken, dass diese Verbesserung noch nicht mit der Geschwindigkeit eines reinen SOME/IP-Netzwerkes mithalten kann.

Aussichten: Durch eine sichere Verarbeitung der initialen Find- und Offer-Nachrichten könnte man über eine Abweichung von der SOME/IP-Spezifikation nachdenken. Die Wiederholungen in der Repetitionphase könnten einen zeitlich höheren Abstand bekommen. Es könnte eine Optimierungsmöglichkeit geben, wenn dadurch Antworten ankommen bevor die Wiederholung ausgelöst wird. Dazu müsste untersucht werden, wie viele Wiederholungen mit und ohne SDN vorliegen und wie oft die Pakete verloren gehen.

Die Anzahl der ECUs in Fahrzeugen steigt weiterhin rasant, daher ist jeder Schritt in Richtung Entlastung des Netzwerkverkehrs wichtig und richtig. Ich sehe die Vorteile einer Service Discovery App in einem SDN-Netzwerk im Fahrzeug noch an anderen Stellen. Zum einen ist es möglich die Tabelle der Serviceinstanzen zu speichern, sei es direkt im Fahrzeug(der Stromverbrauch wäre minimal), sei es im Schlüssel oder sei es im Handy des Nutzers. So könnte bereits bei Systemstart versucht werden die Dienste aktiv anzubieten, erst bei erreichen der EventGroup-Anfrage müssten die Anbieter betriebsbereit sein. Fehler bei den Nachricht-Sequenzen werden jetzt schon abgefangen. Die Dienste anbieten zu können, ohne dass sie Nachrichten verschickt haben, setzt voraus, dass die ECUs feste MAC- und IP-Adressen haben, die Ports sind meist feste Steckverbindungen und sollten die gleichen Nummern bekommen. Somit würden auch böswillige Veränderungen an der Hardware schnell auffallen und auch Ausfälle von ECUs. An dieser Stelle kann auch definiert werden, welches Bauteil mit welchem kommunizieren darf, damit sich Manipulationen beispielsweise des Infotainment-Systems nicht auf die Betriebsbe-

reitschaft der Fahrfunktionen auswirken können. Die ServiceDiscovery wäre somit nicht nur eine Entlastungsinstanz, sondern auch eine Kontrollinstanz.

Je nach Fahrer, Fahrpraxis oder Geldbeutel könnten den Nutzern durch Staffelung der Services verschiedene Grenzen erlaubt werden. So könnte das Familienauto mit der Mutter als Fahrerin 200 km/h, mit dem 20-jährigen Sohn als Fahrer nur 120 km/h, erreichen dürfen. Für den Verkehr gesperrte Wohngebiete könnten mit Lizenzen für Anwohner befahrbar werden. Tempolimit 130 auf Autobahnen - Eine Erweiterung auf 180 km/h könnte gebucht werden, sollte man zu spät zum Geschäftsmeeting kommen. Autonomes Kolonne-fahren oder Stau-Auflösungs-Modus könnte die Service Discovery mit den Service Discoveries der umgebenden Autos vernetzen und so geeignete Geschwindigkeiten umsetzen, unter Berücksichtigung des Fahrzeuggewichts, Steigung oder Gefälle der Straße oder anderer Fahrbahnbeschaffenheiten.

Literaturverzeichnis

[1] Open Networking Foundation: *OpenFlow switch - open networking foundation*. Apr 2015. – URL https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[2] AUTOSAR: *Requirements on SOME/IP Protocol*. Nov 2021. – URL https://www.autosar.org/fileadmin/standards/R21-11/FO/AUTOSAR_RS_SOMEIPProtocol.pdf

[3] AUTOSAR: *Specification on SOME/IP Transport Protocol*. Nov 2021. – URL https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_SOMEIPTransportProtocol.pdf

[4] Autosar: *Specification of Service Discovery*. Nov 2022. – URL https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_ServiceDiscovery.pdf

[5] CoRE-Group: *Communication over Real-Time Ethernet Group*. 2023. – URL https://core.informatik.haw-hamburg.de/. – [Online; accessed 29-November-2023]

[6] Dr. Völker, Lars: *What is Automotive Ethernet?* Sep 2021. – URL https://youtu.be/KwNAEdjd7d0?si=p4YOcx3aVe4zt3Xj

[7] Dr. Völker, Lars: *SOME/IP Protocol Specification*. Nov 2022. – URL https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPProtocol.pdf

[8] Gude, Natasha ; Koponen, Teemu ; Pettit, Justin ; Pfaff, Ben ; Casado, Martín ; McKeown, Nick ; Shenker, Scott: NOX: Towards an Operating System for Networks. In: *SIGCOMM Comput. Commun. Rev.* 38 (2008), jul, Nr. 3, S. 105–110. – URL https://doi.org/10.1145/1384609.1384625. – ISSN 0146-4833

[9] Häckel, Timo ; Meyer, Philipp ; Mueller, Mehmet ; Schmitt-Solbrig, Jan ; Korf, Franz ; Schmidt, Thomas C.: *Dynamic Service-Orientation for Software-Defined In-Vehicle Networks.* 2023. – URL https://doi.org/10.48550/arXiv.2303.13903

[10] IEEE: *P802.3bw/D3.2.1, July 2015 - IEEE Draft Standard for Ethernet Amendment: Physical Layer Specifications and Management Parameters for 100 Mb/s Operation over a Single Balanced Twisted Pair Cable (100BASE-T1).* Jan 2015. – URL https://ieeexplore.ieee.org/servlet/opac?punumber=7177014

[11] Jain, Sushant ; Kumar, Alok ; Mandal, Subhasree ; Ong, Joon ; Poutievski, Leon ; Singh, Arjun ; Venkata, Subbaiah ; Wanderer, Jim ; Zhou, Junlan ; Zhu, Min ; Zolla, Jon ; Hölzle, Urs ; Stuart, Stephen ; Vahdat, Amin: B4: Experience with a Globally-Deployed Software Defined Wan. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM.* New York, NY, USA : Association for Computing Machinery, 2013 (SIGCOMM '13), S. 3–14. – URL https://doi.org/10.1145/2486001.2486019. – ISBN 9781450320566

[12] Jamjoom, Hani ; Williams, Dan ; Sharma, Upendra: Don't Call Them Middleboxes, Call Them Middlepipes. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking.* New York, NY, USA : Association for Computing Machinery, 2014 (HotSDN '14), S. 19–24. – URL https://doi.org/10.1145/2620728.2620760. – ISBN 9781450329897

[13] Kim, Hyojoon ; Feamster, N.: Improving network management with software defined networking. In: *Communications Magazine, IEEE* 51 (2013), Nr. 2, S. 114–119. – ISSN 0163-6804

[14] Klein, Dominik ; Jarschel, Michael: An OpenFlow Extension for the OMNeT++ INET Framework. In: *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques.* Brussels, BEL : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013 (SimuTools '13), S. 322–329. – ISBN 9781450324649

[15] Koponen, Teemu ; Casado, Martin ; Gude, Natasha ; Stribling, Jeremy ; Poutievski, Leon ; Zhu, Min ; Ramanathan, Rajiv ; Iwata, Yuichiro ; Inoue, Hiroaki ; Hama, Takayuki ; Shenker, Scott: Onix: A Distributed Control Platform for Large-Scale Production Networks. In: *Proceedings of the 9th USENIX Conference*

on Operating Systems Design and Implementation. USA : USENIX Association, 2010 (OSDI'10), S. 351–364

[16] Kreutz, D. ; Ramos, F.M.V. ; Esteves Verissimo, P. ; Esteve Rothenberg, C. ; Azodolmolky, S. ; Uhlig, S.: Software-Defined Networking: A Comprehensive Survey. In: *Proceedings of the IEEE* 103 (2015), Januar, Nr. 1, S. 14–76. – ISSN 0018-9219

[17] Larry Peterson: *OpenFlow: Catalyst that Kickstarted the SDN Transformation*. – URL https://opennetworking.org/news-and-events/blog/openflow-catalyst-that-kickstarted-the-sdn-transformation/. – [Online; accessed 26-November-2023]

[18] McKeown, Nick: Software-defined Networking. In: *INFOCOM Keynote Talk* 17 (2009), 01, S. 30–32

[19] McKeown, Nick ; Anderson, Tom ; Balakrishnan, Hari ; Parulkar, Guru ; Peterson, Larry ; Rexford, Jennifer ; Shenker, Scott ; Turner, Jonathan: OpenFlow: Enabling Innovation in Campus Networks. In: *SIGCOMM Comput. Commun. Rev.* 38 (2008), mar, Nr. 2, S. 69–74. – URL https://doi.org/10.1145/1355734.1355746. – ISSN 0146-4833

[20] McKeown, Nick ; Anderson, Tom ; Balakrishnan, Hari ; Parulkar, Guru ; Peterson, Larry ; Rexford, Jennifer ; Shenker, Scott ; Turner, Jonathan: OpenFlow: Enabling Innovation in Campus Networks. In: *SIGCOMM Comput. Commun. Rev.* 38 (2008), mar, Nr. 2, S. 69–74. – URL https://doi.org/10.1145/1355734.1355746. – ISSN 0146-4833

[21] Newman, Peter ; Minshall, Greg ; Lyon, Thomas L.: IP Switching—ATM under IP. In: *IEEE/ACM Trans. Netw.* 6 (1998), apr, Nr. 2, S. 117–129. – URL https://doi.org/10.1109/90.664261. – ISSN 1063-6692

[22] ONF: Software-Defined Networking: The New Norm for Networks. Open Networking Foundation, 2012. – Forschungsbericht. – URL https://opennetworking.org/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf

[23] OpenSim Ltd.: *OMNeT++ Discrete Event Simulator and the INETFramework*. 2023. – URL https://omnetpp.org/. – [Online; accessed 29-November-2023]

[24] Raghavan, Barath ; Casado, Martín ; Koponen, Teemu ; Ratnasamy, Sylvia ; Ghodsi, Ali ; Shenker, Scott: Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure. In: *Proceedings of the 11th ACM Workshop on Hot*

Topics in Networks. New York, NY, USA : Association for Computing Machinery, 2012 (HotNets-XI), S. 43–48. – URL https://doi.org/10.1145/2390231.2390239. – ISBN 9781450317764

[25] Rotermund, Randolf ; Häckel, Timo ; Meyer, Philipp ; Korf, Franz ; Schmidt, Thomas C.: Requirements Analysis and Performance Evaluation of SDN Controllers for Automotive Use Cases. In: *2020 IEEE Vehicular Networking Conference (VNC)*, 2020, S. 1–8

[26] Tanenbaum, Andrew S. ; Wetherall, David J.: *Computer Networks*. 5th. Prentice Hall, 2011. – ISBN 978-0-13-212695-3

[27] Wikipedia contributors: *CAN bus — Wikipedia, The Free Encyclopedia*. 2023. – URL https://en.wikipedia.org/w/index.php?title=CAN_bus&oldid=1140539606. – [Online; accessed 22-February-2023]

A Anhang

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen 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