

MASTERARBEIT

Programmierbarkeit der Data-Plane mittels P4



FH Salzburg

durchgeführt am
Studiengang Informationstechnik & System-Management
an der
Fachhochschule Salzburg

vorgelegt von
Michael Höflmaier

Studiengangsleiter: FH-Prof. Dipl.-Ing. Dr. Gerhard Jöchtl
Betreuer: Mag. DI Ulrich Pache, BSc

Salzburg, 29. Juli 2020

Details

Autor	Michael Höflmaier
Universität	Fachhochschule Salzburg
Studiengang	Informationstechnik & System-Management
Titel der Masterarbeit	Programmierbarkeit der Data-Plane mittels P4
Schlüsselwörter	Data-Plane Programmierung, P4 ₁₆ , P4Runtime, LonTalk, Kontrollnetzwerke
Betreuer	Mag. DI Ulrich Pache, BSc

Abstract

Within the networking domain data-plane programmability is becoming increasingly important, because state-of-the-art networking requirements have a strong dynamic aspect and administrators want to capitalize on modern network techniques. Existing SDN protocols like OpenFlow only specify a set of supported protocol headers and actions, which can be used to influence forwarding decisions of networking hardware. To solve this issue, network administrators can use an extensible, protocol and target independent high-level language to add custom functionality to networking infrastructure. The language achieving this is P4. P4 is a high-level language to define custom match-action-pipelines and a packet header parsing logic, which can be used to implement tailor-made forwarding mechanisms. Therefore, this language can enable data-plane programmability support for otherwise unsupported protocols. One such protocol is the control network protocol LonTalk. This document will cover the theoretical concepts of P4 and LonTalk. Accompanying this, a reference P4-implementation of a LonTalk-Router is presented and described. Conclusively this implementation is discussed and shortcomings, advantages and other findings are presented.

Danksagung

Einleitend möchte ich mich bei meinem Masterarbeits-Betreuer Mag. DI Ulrich Pache bedanken, der sich im Herbst 2019 kurzfristig bereit erklärte, mich bei der Themenfindung zu unterstützen und mir dabei sehr wertvollen Input lieferte. Außerdem möchte ich ihm für das Feedback und die ausführlichen Unterredungen bezüglich einiger Milestones danken. Dabei möchte ich herausstreichen, dass all diese Hilfe vor dem Hintergrund etlicher anderer zu betreuender Master- und Bachelorarbeiten, der Covid-19-Pandemie und der damit einhergehenden zusätzlichen Beanspruchungen stattfand. Außerdem möchte ich meiner gesamten Familie danken, die es mir ermöglichte, mich dank ihrer Hilfe komplett auf die Erstellung dieses Textes zu konzentrieren. Besonders möchte ich hierbei noch die Rolle meiner Partnerin Verena herausstreichen, die ihre endlose Geduld und Hilfe beim Korrekturlesen meiner Arbeit unter Beweis stellte. Schlussendlich möchte ich auch noch einigen meiner Kommilitonen danken, auf deren Hilfe und Zusammenarbeit ich bei der Modifizierung des Latex-Templates, das durch Dorian Prill zur Verfügung gestellt wurde, zählen konnte.

Eidesstattliche Erklärung

Hiermit versichere ich, Michael Höflmaier, geboren am 20. November 1994 in Salzburg, dass die vorliegende Masterarbeit von mir selbstständig verfasst wurde. Zur Erstellung wurden von mir keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Salzburg, am 29. Juli 2020



Michael Höflmaier

1810581024

Matrikelnummer

Inhaltsverzeichnis

1 Einleitung	1
1.1 Problemstellung	1
1.2 Fragestellung	4
1.3 Voraussetzungen und verwendete Tools	4
2 Data-Plane-Programmierung	6
3 Stand der Technik	8
4 P4	9
4.1 Was ist P4?	9
4.2 Was führte zur Entwicklung von P4?	10
4.3 Ziele, Vorteile und Nutzen von P4	13
4.3.1 Ursprüngliche Ziele von P4	13
4.3.2 Vorteile der Nutzung von P4	14
4.3.3 Nutzen für P4-Programmierer*innen	15
4.4 Evolution von P4	15
4.5 Zusammenspiel der Komponenten	17
4.6 Architekturen	18
4.6.1 Allgemeines	18
4.6.2 PISA	20
4.6.3 P4 ₁₄ -Switch-Modell	21
4.6.4 V1-Modell	22
4.6.5 PSA	23
4.6.6 Andere Architekturen	25
4.7 P4-Compiler	25
4.7.1 Entwicklung des P4-Referenzcompilers	25
4.7.2 Architektur des P4-Referenzcompilers	26
4.7.3 Compiler-Projekte	27
4.8 P4Runtime	28
4.9 P4-Sprachkomponenten	30
4.9.1 Allgemeines	30
4.9.2 Datentypen	31
4.9.3 Parser	33
4.9.4 Control	33
4.9.5 Deparser	35
4.9.6 Main	36
5 Kontrollnetzwerke	37
5.1 LonWorks	37
5.1.1 LonWorks-Übertragungsmedien	37
5.1.2 Logische Struktur eines LonWorks-Netzes	38
5.1.3 LonWorks-Netzwerkbausteine	39
5.1.4 LonWorks-Beispielnetzwerk	40
5.1.5 Aufbau eines LonWorks-Knotens	40

5.2	Lontalk-Protokoll	41
5.2.1	Schichten des LonTalk-Protokolls	41
5.2.2	Protocol Data Units des LonTalk-Protokolls	43
6	Grundprinzipien der Implementierung	49
6.1	Gründe für eine P4-Implementierung von LonTalk	49
6.2	Auswahl des zu entwickelnden LonWorks-Gerätes	50
7	Detailfragen im Vorfeld der Implementierung	54
7.1	Wahl des Testaufbaus	54
7.2	Auswahl des Software-Switches	56
7.3	Wahl der Ausführungsumgebung	58
7.4	Wahl der Entwicklungsplattform	61
7.5	Wahl des Nachrichtenerstellungs-Tools	63
7.6	Auswahl der Übertragungstechnologie	64
7.7	Auswahl der Topologie	66
7.8	Machbarkeitsabschätzung	68
8	Umsetzung eines LonTalk-Routers	70
8.1	Details der Ausführungsumgebung	70
8.2	Details der Scapy-Programme	75
8.2.1	Scapy-Programm <code>send.py</code>	76
8.2.2	Scapy-Programm <code>receive.py</code>	80
8.3	Designentscheidungen bezüglich der P4-Implementierung	82
8.4	P4-Implementierung des LonTalk-Routers	84
8.4.1	Allgemeine Inhalte der P4-Implementierung	84
8.4.2	P4-Definition der benötigten Header	85
8.4.3	P4-Parser	87
8.4.4	P4-Ingress-Pipeline des Routers	89
8.4.5	P4-Egress-Pipeline und Deparser	93
8.4.6	P4-Debugging-Tabelle	94
8.5	Konfiguration des LonTalk-Routers	95
8.6	Testung des LonTalk-Routers	99
8.7	Inbetriebnahme der virtuellen Maschine	103
9	Diskussion der Realisierungsergebnisse und der Forschungsfrage	105
9.1	Diskussion der Realisierungsergebnisse	105
9.1.1	Gewonnene Erkenntnisse	105
9.1.2	Einschränkungen und Limitierungen der Realisierung	106
9.1.3	Vorteile der Realisierung	107
9.2	Implementierbarkeit des LonTalk-Protokolls	108
10	Schlussfolgerungen und Aussichten	110
10.1	Zusammenfassung	110
10.2	Wichtigste Erkenntnisse	111
10.3	Ausblick auf mögliche Weiterentwicklungen	112
Abkürzungsverzeichnis		114

Abbildungsverzeichnis	116
Tabellenverzeichnis	117
Listings	118
Literatur	120

1 Einleitung

Im Rahmen des Fachhochschulschwerpunkts für Netzwerktechnik und Security beschäftigt sich der Autor im Zuge dieser Masterarbeit mit der Thematik der Programmierbarkeit der Data-Plane unter Verwendung von P4. In den folgenden Kapiteln sind die zugrunde liegenden Problemstellung und die Rahmenbedingungen und Voraussetzungen für die Erstellung dieser Arbeit ausführlich erläutert. Ergänzend sind auch die zentralen Fragestellungen vorgestellt, die in dieser Arbeit behandelt werden. Die Gesamtheit der folgenden Ausführungen sollten dem/der Leser*in als Kontext für die weiteren Kapitel dieses Schriftstücks, welche sich unter anderem mit den theoretischen und praktischen Bereichen beschäftigen, dienen.

1.1 Problemstellung

Das Themengebiet Software Defined Networking ist ein sich rapide verändernder Teilbereich der Netzwerktechnik. Unter Software Defined Networking (SDN) wird gemeinhin ein neuer Zugang zur Programmierbarkeit von Netzwerken verstanden. Der Terminus „Programmierbarkeit von Netzwerken“ bezieht sich hierbei auf die Fähigkeit, das Verhalten eines Netzwerkes, hinsichtlich Verwaltung, Veränderungen, Kontrolle und Initialisierung, über offene Schnittstellen dynamisch zu beeinflussen. Zusammengefasst bedeutet dies, dass SDN es ermöglicht, das Tun eines Netzwerkes kontrollieren und beeinflussen zu können. Darüber hinaus wird im SDN-Paradigma davon ausgegangen, dass die Funktionen des Netzwerkes in drei Ebenen abstrahiert werden kann. Diese Aufteilung ist in Abbildung 1 ersichtlich und besteht aus dem Application-Layer, der Control-Plane und der Data-Plane. [2]

Dabei bildet die Data-Plane die unterste Ebene. Die Netzwerk-Entitäten, welche auf dieser Ebene beheimatet sind, haben grundsätzlich die Aufgabe, formatierte Datenmengen, also Pakete, in einem paket-orientierten Netz weiterzuleiten [3]. Nichtsdestotrotz findet auf dieser Ebene auch die Überwachung lokaler Informationen und die Sammlung von Daten zu Zwecken der statistischen Auswertung statt. In der darüberliegenden Schicht befindet sich die Control-Plane, deren Hauptverantwortung es ist, die Data-Plane zu verwalten. Zwischen der Data- und Control-Plane befindet sich die Southbound-Schnittstelle, welche den Austausch von Nachrichten zwischen diesen Ebenen ermöglicht. Diese Schnittstelle wird hauptsächlich von Kommunikationsprotokollen, wie zum Beispiel OpenFlow oder Forwarding and Control Element Separation (ForCES), genutzt, um deren Funktionalität zu ermöglichen. Die oberste Ebene bildet der Application-Layer, der dazu verwendet wird, Steuerungsbefehle an die Control-Plane zu übersenden. Die Grundlage für diese Befehle

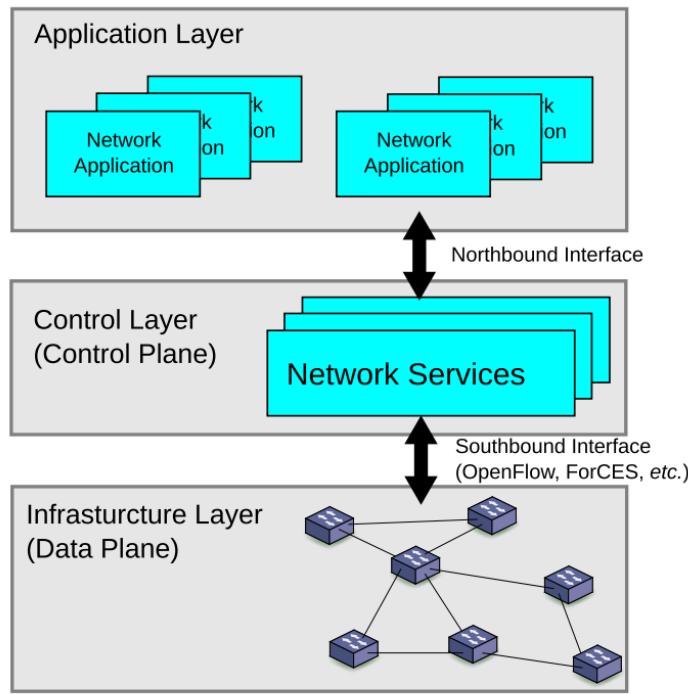


Abbildung 1: Schichten der SDN-Architektur [1]

bildet die Information, welche über die Northbound-Schnittstelle von der Control-Plane an den Application-Layer übermittelt wird. [1]

Die beiden zuerst genannten Ebenen wurden „traditionell“, bei nicht programmierbaren Netzwerkgeräten, immer auf demselben Gerät implementiert. Demgegenüber stehen programmierbare Netzwerkgeräte, welche die Aufgaben der untersten zwei Schichten nicht mehr gesammelt aufweisen, sondern die Rolle der Control-Plane an eine separate Entität abgeben. Beginnend mit der Einführung des SDN-Paradigmas und spätestens durch die erste Formulierung von OpenFlow im Jahr 2008, das eine programmierbare Schnittstelle zwischen dem Control- und Data-Plane postuliert, wurde eine Möglichkeit für zusätzliche Anpassungsfähigkeit der Netzwerkgeräte erreicht [1], [4]. Diese Anpassungsfähigkeit kann dem Betreiber von SDN-Lösungen weitreichende Vorteile hinsichtlich der Optimierung ihrer Netzwerke und Arbeitsabläufe bieten [5], [4].

Ungeachtet dieser Flexibilisierung bleibt die grundlegende Funktion von Netzwerkgeräten und der Data-Plane, das Weiterleiten von Paketen, unangetastet [6]. Dies bedeutet, dass das Verhalten der Data-Planes als fix und nur schwer veränderbar angesehen werden kann [7]. Dennoch gibt es, aufgrund von technischen und/oder wirtschaftlichen Gründen, die Notwendigkeit, neue Weiterleitungsverfahren von Paketen in Netzwerkhardware zu unterstützen. Die Implementierung dieser Verfahren kann sich laut [1] und [6] allerdings zeitlich oft als sehr langwierig herausstellen.

Ein bezeichnendes Beispiel dafür ist Virtual Extensible Local Area Network (VXLAN). Die Beschränkungen von Virtual Local Area Network (VLAN), allem voran die Anzahl der erstellbaren VLANs, führten im Jahr 2010 dazu, dass mit dem Entwurf von VXLAN begonnen wurde. Der gesamte Prozess, beginnend mit der Spezifikation und endend mit der Implementierung, konnte erst nach vier Jahren abgeschlossen werden. Dies führte dazu, dass die Vorteile von VXLAN gegenüber VLAN zeitlich sehr spät nutzbar wurden. Die Gründe für diese lange Wartezeit sind vielfältig, aber manifestierten sich vor allem dadurch, dass für Hersteller von Netzwerkgeräten die Implementierung eines solchen neuen Netzwerkprotokolls mit großen Kosten- und Ressourceneinsatz verbunden ist. Dies lässt sich darauf zurückführen, dass viele Netzwerkgeräte für die ihnen zugesetzte Aufgabe mit schnellen, anwendungsspezifischen Schaltungen, sogenannten Application-Specific Integrated Circuits (ASICs), ausgestattet werden. Unter bestimmten Umständen kann es daher voneinander sein, dass für die Implementierung eines neuen Netzwerkprotokolls eine Neuentwicklung der Hardware respektive der anwendungsspezifischen Schaltungen eines Netzwerkgeräts vorgenommen werden muss. Dieses „Bottom-Up“-Prinzip hinsichtlich der Integrierung neuer Protokolle hat daher das Potential, schnelle Veränderungen im Bereich der Netzwerktechnik zu verhindern. [6]

Generell besteht die Möglichkeit, dass sich die Abhängigkeit von „spezialisierter“ Hardware mittels generischer Hardware, in Form von Field Programmable Gate Arrays (FPGAs), lösen ließe [8]. Jedoch garantiert die Verwendung dieser wahrscheinlich keine langfristige Flexibilität, da man in diesem Fall trotzdem an deren gerätespezifischen Eigenschaften, wie zum Beispiel Throughput und Input/Output-Schnittstellen, gebunden wäre [9].

Die Verwendung einer Programmiersprache, die es Entwickler*innen ermöglicht, beliebige Protokolle sehr schnell, einfach und möglichst zielunabhängig selbst zu implementieren, könnte die Möglichkeit bieten, die Abhängigkeit von konventioneller Netzwerkhardware zu reduzieren. Dadurch könnten auch bereits existierende und bewährte Protokolle auf anderer Hardware lauffähig gemacht werden, um eine weitere Flexibilisierung und Unabhängigkeit zu erreichen. Es würde auch die Option bieten, Geräte zu entwerfen, welche nur exakt die gewünschten Funktionen implementieren. Durch das Entfernen von nicht benötigten Features könnte eine Verbesserung der Leistung gegenüber einem vergleichbaren Gerät entstehen, das das Feature aufweist. Diese Möglichkeiten sind vor allem dann sehr interessant, wenn das Protokoll von Interesse nur von einer geringen Anzahl an Herstellern implementiert wird. Dies ist zum Beispiel bei dem LonTalk-Protokoll der Fall, das unter anderem im Bereich der Power Line Communication (PLC) seinen Einsatz findet [10]. Durch eine Eigenimplementierung dieses Protokolls, unter der Zuhilfenahme der bereits skizzierten Programmiersprache, könnten Abhängigkeiten eines Unternehmens oder einer Organisation gegenüber den Geräteherstellern reduziert oder gar vermieden werden. Die Idee für eine solche Programmiersprache wurde erstmals im Jahre 2014 publiziert und

im darauffolgenden Jahr unter dem Namen Programming Protocol-Independent Packet Processors (P4) spezifiziert [11], [12].

Da diese Neuerung eine Vielzahl von interessanten, neuen Möglichkeiten bietet und bereits einige zusätzliche Entwicklungen hervorbracht hat, werden in dieser Masterarbeit mehrere wichtige Teilespekte von P4 vorgestellt und auch in einen praktischen Kontext gesetzt.

1.2 Fragestellung

Die Zielsetzung dieser Masterarbeit ist es, die folgenden Fragestellungen auf wissenschaftliche Art und Weise zu behandeln.

Allem voran steht die Frage, warum P4 durch das P4-Sprachkonsortium entwickelt wurde. Um diese Frage zu präzisieren: Aufgrund welcher Entwicklungen und Problemstellungen sahen sich Forschungsinstitute und Industrievertreter veranlasst, die Entwicklung einer Sprache, wie P4, voranzutreiben?

Der zentrale Aspekt, auf den in dieser Masterarbeit eingegangen wird, ist, ob es möglich ist, das LonTalk-Kommunikationsprotokoll durch Zuhilfenahme von P4 selbst zu implementieren. Bezugnehmend auf dieses Protokoll wird in dieser Masterarbeit die Frage behandelt, ob die öffentlich zugänglichen Protokollbeschreibungen ausreichen, um eine Implementierung von LonTalk durch P4 zu ermöglichen.

1.3 Voraussetzungen und verwendete Tools

Für die Erstellung dieser Masterarbeit bedarf es der Erfüllung einiger Rahmenbedingungen. Diese sind notwendig, um die Erarbeitung der theoretischen und praktischen Erkenntnisse zu ermöglichen.

Eine zentrale Voraussetzung stellt dabei die Literaturrecherche dar, welche im besonderen zur Thematik von P4, aber natürlich auch zu den Teilen, welche dem Bezugsrahmen von LonTalk angehören, stattfinden muss. Hierzu zählt allem voran die Header-Struktur des LonTalk-Protokolls. Auch bedarf es für die Behandlung der Forschungsfragen einiger Werkzeuge, vor allem der Virtualisierungslösung VirtualBox, bereitgestellt durch die Oracle Corporation. Diese wird benutzt, um die offizielle Tutorial-VM des P4-Sprachkonsortiums verwenden zu können und dadurch eine schnelle Wissensgewinnung und Entwicklung in praktischen Belangen zu ermöglichen. Auch wird Python, eine interpretierte Programmiersprache, eingesetzt. Außerdem kommt Scapy, ein Python-Programm, zur Anwendung, das für die Erzeugung von LonTalk-Datenframes benötigt wird. Schlussendlich sei auch die

PyCharm-Entwicklungsumgebung der Firma JetBrains erwähnt, da dieses für die Entwicklung von P4- und Python-Programme zur Anwendung kommt.

Da für die Klärung der Fragestellungen, welche das Zentrum dieser Masterarbeit bilden, ein umfassendes Hintergrundwissen über die verwendeten Technologien nötig ist, werden diese in den nachfolgenden Kapiteln beschrieben. Die Gesamtheit der Ausführungen dieser Kapitel soll dem/der Leser*in vor allem als Grundlage für den Inhalt des praktischen Teils dieser Arbeit dienen.

2 Data-Plane-Programmierung

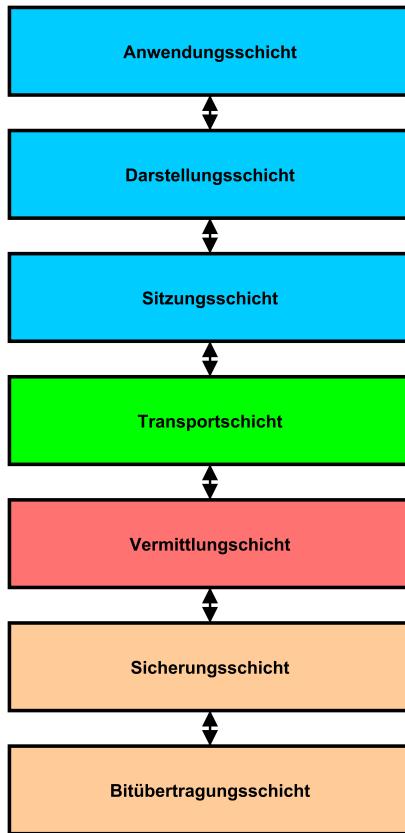


Abbildung 2: Schichten des OSI-Modells nach [13]

Das Kernthema dieser Arbeit ist die Programmierbarkeit der Data-Plane. Deshalb sollte vorab darauf eingegangen werden, warum Data-Plane-Programmierung für die Netzwerktechnik von Bedeutung ist. Wie bereits einleitend in Kapitel 1 beschrieben, wurde für die Realisierung des SDN-Paradigmas eine Trennung der Control- und Data-Plane vorgenommen. Diese Trennung war notwendig, um eine Programmierbarkeit der Control-Plane zu ermöglichen. Jedoch konnte alleine durch die Programmierbarkeit der Control-Plane keine vollständige Protokollunabhängigkeit von Netzwerkgeräten erreicht werden. Als Grund dafür sind die bereits in Kapitel 1.1 genannten Einschränkungen von OpenFlow zu identifizieren. Auch wurde bereits erwähnt, dass Programmierbarkeit eine Fülle von Vorteilen und Möglichkeiten mit sich bringt. Die Gründe für die Notwendigkeit einer Trennung eines Systems in zwei Subsysteme kann auch anhand eines generalisierten Falles betrachtet werden: Sobald ein komplexes System in zwei oder mehrere fundamentale Schichten getrennt wird, resultiert daraus, dass die einzelnen Schichten nun auch unabhängig voneinander betrachtet werden können. Diese Unabhängigkeit der Schichten voneinander bedingt nun auch, dass die Schichten einzeln verändert werden können. Dies bringt allen Schichten, als großes Ganzes betrachtet, den Vorteil von Flexibilität. Ein wohl bekanntes Beispiel hierfür ist der Aufbau von Computernetzen nach dem Open System Interconnect-Referenzmodell

(OSI), wie es in [14] beschrieben und in Abbildung 2 dargestellt ist.

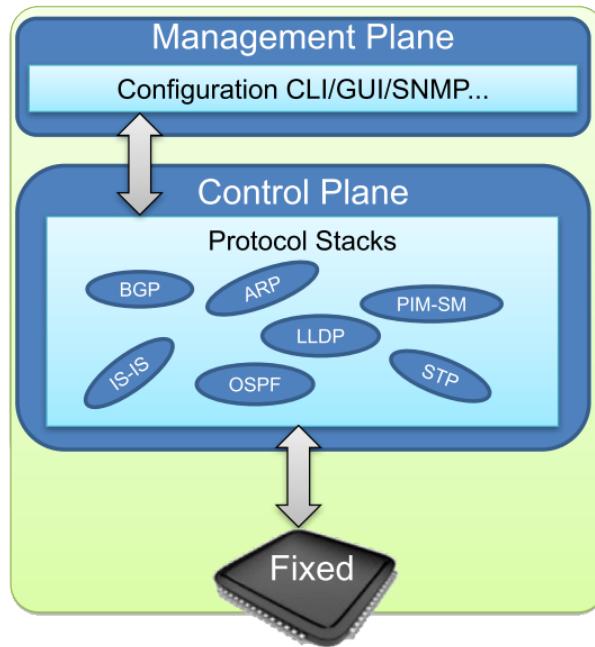


Abbildung 3: Architekturelle Bausteine traditioneller Netzwerkswitches [15]

Im Control- und Data-Plane-Universum ist der größte und grundlegendste Vorteil einer Trennung in die besagten zwei Ebenen die Programmierbarkeit [2]. Überdies kann unter Verwendung eines ASIC auf der Hardware-Ebene der Aufbau eines Netzwerkgerätes durchaus komplexe Züge aufweisen. Dies wurde bereits in Kapitel 1 kurz erwähnt und ist auch in Abbildung 3 illustriert. Hinsichtlich der Komplexität eines Netzwerkgeräts auf der Data-Plane kann durch den Gebrauch einer Abstraktionsmöglichkeit Abhilfe geschaffen werden. Ein Werkzeug hierfür kann laut [16] die Verwendung einer höheren Programmiersprache sein. Die höhere Programmiersprache, welche in diesem Zusammenhang zu nennen ist, heißt P4.

3 Stand der Technik

In diesem Kapitel wird der derzeitige Stand der Technik bezogen auf Technologien, die eine konzeptionelle Ähnlichkeit mit P4 aufweisen, präsentiert. Hierfür muss erwähnt werden, dass P4 einen Mix an Zielen und Vorteilen bereithält, der in Kapitel 4.3 genauer behandelt wird. Vorab kann aber gesagt sein, dass der gesamte Katalog an Eigenschaften von P4 nicht durch die Verwendung einer anderen, bereits existierenden Technologie erreicht und umgesetzt werden kann.

Im Jahr 2000 wird in [17] erstmals eine Softwarearchitektur vorgestellt, die es erlaubt, unter der Verwendung von Paketverarbeitungsmodulen flexible und konfigurierbare Router zu erstellen. Diese Erkenntnisse wurden später verwendet, um ClickOS, eine flexible Softwareplattform für das Entwerfen von Middleboxen, zu entwickeln [18]. Die Vielseitigkeit der Verwendung ist auch ein Ziel, das durch P4 erreicht werden soll [11]. P4 teilt sein Ziel nach Hersteller- und Hardwareunabhängigkeit mit packetC, einer Low-Level-Programmiersprache, welche bereits im Jahr 2009 vorgestellt wurde. Durch die Verwendung von packetC soll es laut [19] möglich sein, eigene Protokolle zu implementieren, jedoch wird bei dieser Technologie Deep Packet Inspection (DPI) als beispielhafter Anwendungsfall gesehen. Im selben Jahr wurde Plug vorgeschlagen, das es durch den Einsatz von flexibleren beziehungsweise generischeren Lookup-Modulen erlauben soll, selbst definierbare Forwarding-Methoden zu implementieren [20]. Hier wurden jedoch die Themengebiete Parsen und Editieren von Paketen nicht genauer behandelt. Programmierbare Parser wurden daraufhin allerdings durch das Kangaroo System ermöglicht [21]. Im Jahr 2013 wurde Protocol-Oblivious Forwarding (POF) vorgeschlagen, das Protokollunabhängigkeit postuliert, jedoch das Ziel nach Plattformunabhängigkeit, das eine Kernüberlegung von P4 ist, außer Acht lässt [22]. Die Kernfrage, wie eine möglichst flexible und damit programmierbare Data-Plane erreicht werden kann, wurde 2013 in [23] erläutert. Darin wird das Konzept von Reconfigurable Match Table (RMT) vorgestellt, das es ermöglicht, viele Restriktionen von herkömmlichen Match-Tabellen zu umgehen. Die Verwendung von RMT im Zusammenhang mit dem „Match-Action“-Paradigma, welches ein Kernstück der Protocol Independent Switch Architecture (PISA) ist, führt dazu, dass eine Rekonfigurierbarkeit des betreffenden Netzwerkgeräts erreicht wird.

Die einzigartige Kombination von Eigenschaften, die in der Sprache P4 vereint sind, ermöglichte bereits die Implementierung einiger sehr bekannter und grundlegender Funktionen, Protokolle und neuer Ideen. Hier sind folgende Implementierungen beispielhaft zu nennen: NetCache [24], ein In-Network-Cache für Key-Value-Datenbanken, eine Layer-Drei-Firewall [25] und ein Load-Balancer auf Schicht vier des OSI-Modells [26]. Der von P4 erreichte Katalog an Möglichkeiten, die Flexibilität der Sprache und die Aktualität des Themas sind unter anderem Gründe warum P4 in dieser Arbeit genauer behandelt wird.

4 P4

P4 ist der wichtigste Grundstein, auf dem diese Arbeit fußt. Die Anwendung dieser Technologie bietet einige Vorteile gegenüber anderen bisherigen Sprachen, welche versuchen, teilweise ähnliche Ziele zu erreichen. Die Vorteile von P4, welche in Kapitel 4.3 vorgestellt werden, führt dazu, dass es eine breite Front an Unterstützern für P4 gibt. Dieser Kreis von Förderern besteht dabei aus prominenten, führenden Industrieunternehmen, wie zum Beispiel Xilinx¹, Cisco² und Google [29]. Aber auch eine breite Front von Forschungseinrichtung, wie zum Beispiel Salzburg Research³ und einige renommierte Universitäten, hier ist beispielhaft die Cornell University⁴ zu nennen, zählen zu den Unterstützern. Die Unterstützung, die dieses Projekt laut [11] seit 2014 erfährt, führt seither zu einer stetigen Weiterentwicklung der Sprache und zur Erstellung von Tools, welche die Entwicklung von P4-Anwendungen für Nutzer*innen vereinfachen soll.

4.1 Was ist P4?

Die Frage, was P4 eigentlich ist, lässt sich relativ einfach anhand des Namens der Sprache, welche erstmals in [11] erwähnt wurde, beantworten: P4 ist ein Akronym und steht, wie bereits in Kapitel 1 erläutert, für "Programming Protocol-independent Packet Processors". Außerdem handelt es sich hierbei um eine höhere Programmiersprache, die das Weiterleitungsverhalten von Paketen auf einem Zielgerät bestimmt [7]. In [3] wird darauf hingewiesen, dass man korrekterweise eine Unterscheidung zwischen *ein Netzwerkgerät programmieren* und *die Data-Plane eines Netzwerkgerätes programmieren* treffen muss. Denn wie bereits erwähnt, findet das Weiterleiten von Paketen auf der Data-Plane statt. Hier setzt P4 an und definiert genau diese Funktionalität. Die Sprache wird überdies auch genutzt, um Schnittstellen für den Nachrichtenaustausch zwischen der Control- und Data-Plane zu definieren. Die Nutzung von P4 beschränkt sich allerdings auf diese beiden Dinge und kann daher, laut [3], zum Beispiel nicht genutzt werden, um die Funktionsweise der Control-Plane zu bestimmen. Außerdem darf durch die Anwendung von P4 niemals der Eindruck entstehen, dass die Bedeutung der Control-Plane geringer als die der Data-Plane ist, da diese immer noch benötigt wird, um die Weiterleitungstabellen eines Netzwerkgerätes zu befüllen. Die Verwendung von P4 bedingt daher auch weiterhin entweder die Nutzung einer lokalen Control-Plane oder eines SDN-Controllers.

¹Xilinx Inc., Entwickler und Hersteller von FPGA-Lösungen [27].

²Cisco Systems, Entwickler und Hersteller von Telekommunikationslösungen [28].

³Salzburg Research Forschungsgesellschaft m.b.H., ein unabhängiges Forschungsinstitut mit Sitz in Salzburg [30].

⁴Cornell University, eine US-amerikanische Universität im Bundesstaat New York [31].

4.2 Was führte zur Entwicklung von P4?

Die Frage, welche Entwicklungen und Umstände dazu führten, dass P4 überhaupt angedacht und später auch spezifiziert wurde, ist eine der zentralen Fragestellungen dieser Masterarbeit (siehe Kapitel 1.2). Im Laufe dieses Kapitels wird versucht, diese Frage aus dem Blickwinkel eines Netzwerkbetreibers zu behandeln und fundiert zu beantworten.

Seitdem das Interesse von Netzwerkbetreibern an SDN-Lösungen aufgekommen und gestiegen ist, wurden auch die Bemühungen seitens der Hersteller von Netzwerk-Hardware intensiviert, passende Produkte anbieten zu können. Dabei setzte sich vor allen OpenFlow durch, das auf einer Vielzahl von Netzwerkgeräten verfügbar ist und sich zu dem de facto-Industriestandard entwickelte [4]. Nichtsdestotrotz wurde vonseiten der Hersteller auch noch auf andere Ansätze gesetzt. Beispieltechnologien dafür sind die Protokolle Network Configuration Protocol (NETCONF) und Representational State Transfer Configuration Protocol (RESTCONF), welche in [32] respektive [33] spezifiziert wurden und vor allem für die Netzwerkautomatisierung herangezogen werden. Daraus kann sich laut [15] aber auch ein Problem für den/die Endnutzer*in ergeben. Zum Beispiel, wenn Produkte verschiedener Hersteller betrieben werden, kann die Interoperabilität der einzelnen Netzwerkkomponenten aufgrund der Kombination von verwendeten Technologien mit großer Komplexität einhergehen oder stark eingeschränkt sein.

Wie bereits in diesem Kapitel erwähnt, entwickelte sich OpenFlow, seitdem es in [4] vorgeschlagen wurde, zu einem sehr weit verbreiteten Bestandteil von SDN-Lösungen. Jedoch ist OpenFlow auch mit einer starken Limitierungen belastet. In Kapitel 1.1 wurden die Schwierigkeit skizziert, dass OpenFlow laut [11] nur eine strikte Auswahl an Header-Feldern spezifiziert und daher auch nur diese zur Verfügung stellt. Der Fakt, dass nur eine fixe Bandbreite an Headern zur Verfügung steht und damit durch den/die Nutzer*in auch nur eine ausgewählte Menge an Protokollen verwendet werden kann, schränkt die Verwendungsmöglichkeiten von OpenFlow für den Netzwerkbetreiber stark ein. Dem gegenüber steht der Trend, dass fortlaufend neue Protokolle entwickelt werden, welche deshalb unter Umständen nicht in SDN-Lösungen eingesetzt werden können, da das gewünschte Protokoll nicht unterstützt wird. Dieser Trend lässt sich aus den Veröffentlichungszahlen für Spezifikationen und Dokumentationen, genauer Request for Comments (RFCs), herauslesen, welche durch Organisationen wie die Internet Engineering Task Force (IETF), die Internet Research Task Force (IRTF) und das Internet Architecture Board (IAB) publiziert werden. Die Anzahl der Veröffentlichungen wird jährlich auf [34] bekannt gemacht und können als ein Gradmesser für Innovation im Bereich Netzwerktechnik angesehen werden. Dabei sticht primär die große Anzahl an Veröffentlichungen heraus, welche seit der Jahrtausendwende im Durchschnitt 293 Publikationen pro Jahr beträgt. Der bisherige Höchststand wurde im 2006 erreicht, da in diesem Jahr 459 RFCs veröffentlicht wurden.

In [20] wird beschrieben, dass die fundamentale Ursache, warum nicht alle existierenden Header unterstützt werden können, in der Hardwarekonfiguration der Netzwerkgeräte zu finden ist. Da die Weiterleitungsfunktion eines Netzwerkgerätes immer auf dem Prinzip der Suche eines Eintrags in einer Tabelle aufbaut, muss für ein jedes unterstütztes Protokoll Platz in einer solchen Tabelle sein. Der Aufbau und die Struktur einer Weiterleitungstabelle muss dabei den Anforderungen des gewünschten Protokolls gerecht werden. Da viele Hersteller ihre Netzwerkgeräte mithilfe eines ASIC realisieren, bedeutet dies automatisch, dass die Struktur der Tabelle meist fix ist. Dies wiederum hat zur Folge, dass das betreffende Gerät nur Protokolle unterstützt, die auf der verwendeten Tabellenstruktur lauffähig sind. Es besteht allerdings auch die Möglichkeit, ein neues/gewünschtes Netzwerkprotokoll mithilfe des NetFPGA-Projekts, also unter der Verwendung von FPGA-basierender Hardware, zu realisieren. Dieses Projekt wurde erstmals in [8] vorgestellt. Der große Nachteil einer Implementierung auf Basis eines FPGA besteht laut [20] allerdings in der geringeren Leistungsfähigkeit und den höheren Beschaffungskosten gegenüber einer ASIC-Implementierung.

In [35] wird der Begriff „Tussle“ erstmals im Zusammenhang mit Netzwerktechnik erwähnt. Darunter wird der Kampf zwischen verschiedenen Stakeholdern verstanden, die jeweils ihre eigenen Interessen vertreten und dadurch in Konflikt mit den Vorhaben anderer geraten. Um diesen Begriff auf die Problematik der Protokolle anzuwenden: Der/die Nutzer*in, der/die die Implementierung eines bestimmten Protokolls wünscht, ist den Herstellern von Netzwerkgeräten, wenn man von Lobbying-Arbeit absieht, „ausgeliefert“. Denn für den Erhalt von hochperformanter Hardware ist er/sie auf diese angewiesen. Im Falle, dass das gewünschte Feature nun durch den Hersteller implementiert wird, muss der/die Nutzer*in, wie bereits in 1.1 beschrieben, mitunter eine längere Zeitspanne abwarten. Sobald die Netzwerkhardware, die das besagte Feature enthält, auf dem Markt erscheint, muss der/die Nutzer*in trotzdem noch meist eine wirtschaftliche Abwägung treffen, ob die Vorteile der neuen Funktion den Kosten der Neuanschaffung überwiegen. Am Ende dieses Prozesses entspricht das Produkt, das der/die Käufer*in nun in Händen hält, möglicherweise nicht genau seiner/ihrer ursprünglichen Anforderung: die Möglichkeit, ein spezifisches, neues Protokoll nutzen zu können.

Eine Lösung, um diesen vielen möglichen Problemen zu entgehen, wäre, die Verwendung von dedizierter, domänenspezifischer Hardware. Wie in [22] vorgeschlagen wird, könnte diese als „White Box“ ausgeführt sein und damit auch durch den/die Nutzer*in selbst programmierbar sein. Ähnliche Konzepte existieren in anderen Bereichen der Informatiknstechnik bereits seit langem (siehe Abbildung 4). Zum Beispiel fand eine Umsetzung dieses Grundgedankens im Bereich Machine Learning statt [37]. Hier wurden unter Verwendung sogenannter Tensor Processing Units (TPUs) und einer domänenspezifischen Sprache, in Form des TensorFlow-Frameworks (siehe Abbildung 4), deutliche Verbesserungen erreicht.

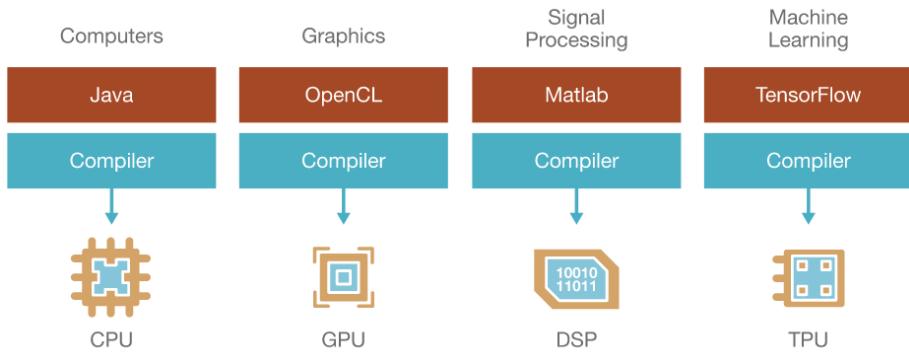


Abbildung 4: Beispiele in der Informationstechnik für die Programmierbarkeit von „White Box“-Hardware [36]

rungen gegenüber der bisher eingesetzten Hardware erzielt. Das Konzept der Verwendung von domänenspezifischer Hardware kann auch auf die Netzwerkdomäne angewendet werden. Barefoot Networks ist eine Firma, gegründet im Jahr 2013, welche aus genau dieser Idee heraus geboren wurde [6]. Barefoot Network verkauft auf die Netzwerkdomäne angepasste, programmierbare Silizium-Chips, die das Herzstück eines voll programmierbaren Netzwerkgerätes bilden können [38].

Wie bereits erwähnt, braucht es für die Entwicklung von domänenspezifischer Hardware generisch einsetzbare und flexible Tabellen, auch Lookup-Module oder RMTs genannt. Die Anwendung eines solchen Moduls erlaubt Veränderungen in den Weiterleitungstabellen, ohne eine physische Veränderung der Hardware vornehmen zu müssen. Die angesprochenen Module können laut [20] hardwareseitig sehr effizient durch den Einsatz von Ternary Content-Addressable Memory-Speicherzellen (TCAM) realisiert werden. TCAM erlaubt die Verwendung eines dritten Speicherzustands „X“, auch „don't care“ genannt, der eine sehr große Flexibilität bei der Tabellensuche erlaubt [39]. Diese Speicherart ist speziell für Anwendungen wie die Tabellensuche geeignet, da sie die Parallelisierung einer Suchabfrage erlaubt und deshalb hohe Suchgeschwindigkeiten gestattet. Der Einsatz von RMTs bedingt, nach Aussagen in [20], aber auch den Einsatz einer neuen Architektur auf Hardwareebene.

Für die Realisierung von programmierbaren Netzwerkgeräten wird dabei in [23] folgende Herangehensweise skizziert: Da das in OpenFlow verwendete „Match+Action“-Prinzip bereits weit verbreitet ist und eine stabile Ausgangslage bietet, kann es als probate Lösung herangezogen werden. Die Nachteile der Umsetzung in OpenFlow zeigen sich dadurch, dass die Auswahl von verfügbaren Aktionen auf Seiten von OpenFlow sehr gering ist und dass selbst definierbare Match-Felder meist durch die Zuhilfenahme von Multiple Match Table-Tabellen (MMT) anstatt von RMT-Tabellen implementiert werden. OpenFlow spezifiziert dabei nicht die Menge und Struktur der Tabellen, die in der verwendeten Hardware zur Verfügung stehen müssen, was einen weiteren Nachteil darstellt. Allgemein könnten unter

Verwendung von RMT aufgrund deren Vorteile, bei Anwendung des „Match+Action“-Prinzips, neue Felddefinitionen hinzugefügt und bestehende verändert werden. Außerdem kann die Anzahl, die Breite in Bit, die Tiefe und die Topologie der verwendeten Tabellen vom Nutzer konfiguriert werden. Dabei muss allerdings noch beachtet werden, dass manche Eigenschaften der Tabelle trotzdem Hardware-Einschränkungen unterliegen: Zum Beispiel kann die Breite eines Tabelleneintrags nicht größer sein, als es die physischen Speicherzellen zulassen. [23]

Die Verwendung dieser architektonischen Grundidee trug einen großen Teil dazu bei, dass es schließlich möglich wurde, programmierbare Netzwerkgeräte mit hohen Leistungskennzahlen zu bauen. Durch Anwendung des bereits erwähnten und in Abbildung 4 dargestellten Konzeptes, das Nutzer*innen erlaubt, domänenspezifische „White Box“-Systeme selbst programmieren zu können, kann festgestellt werden, dass es nur noch einer für die Netzwerkdomäne passenden Sprache bedarf. Weist die angedachte Sprache nun ähnliche Flexibilität und Vielseitigkeit wie ihr Hardware-Gegenstück auf, kann das Ziel als erreicht angesehen werden, dass Netzwerkgeräte umfassend programmierbar sind. Hierbei muss man beachten, dass dieses Ziel immer im Kontext herkömmlicher, nicht-programmierbarer Hardware betrachtet werden muss und die programmierbaren Geräte gegenüber diesen auch hinsichtlich Preis und Leistung vergleichbar sein müssen. Die Rolle einer solchen Programmiersprache für die Netzwerkdomäne erfüllt P4.

4.3 Ziele, Vorteile und Nutzen von P4

In diesem Kapitel werden die ursprünglichen Ziele, die durch die Entwicklung von P4 erreicht werden sollten, vorgestellt. Außerdem wird auch noch auf den Nutzen beziehungsweise die Vorteile eingegangen, welche sich durch die Verwendung dieser Sprache ergeben können.

4.3.1 Ursprüngliche Ziele von P4

In [11], in dem P4 erstmals skizziert wurde, sind drei Ziele erwähnt, welche durch die Formulierung der Sprache erreicht werden sollten. Zuerst wird hier die Rekonfigurierbarkeit einer Netzwerkkomponente, nachdem diese bereits im aktiven Einsatz ist, genannt. Mit Rekonfigurierbarkeit ist gemeint, dass die Weiterleitungsfunktionen und das Parsen von Netzwerkpaketen neu definiert werden kann. Zweitens ist das Erreichen von Protokollunabhängigkeit eine zentrale Anforderung, die erfüllt werden soll. Der Grundgedanke dabei ist,

dass Netzwerkgeräte nicht mehr an bestimmte, durch den Hersteller vorgegebene Protokolle gebunden sind. Diese Anforderung wird durch den Einsatz eines programmierbaren Parsers erreicht, da dieser es ermöglicht die benötigte Informationen aus den Paket-Headern zu extrahieren. Die Anzahl und der Inhalt der „Match+Action“-Tabellen soll schlussendlich dazu genutzt werden können, auf Basis der ausgelesenen Header-Information zu arbeiten. Das dritte und letzte Ziel, das Erwähnung findet, ist Geräteunabhängigkeit. Darunter wird verstanden, dass der/die Nutzer*in von P4 kein Detailwissen über die verwendete Ziel-Hardware haben muss. Denn mithilfe eines Compilers, der auf das Zielgerät spezifisch angepasst ist, soll es möglich sein, eine vom Gerät unabhängige Beschreibung des gewünschten Verhaltens in ein gerätespezifisches Programm umzuwandeln. Die Beschreibung des Verhaltens erfolgt dabei durch ein P4 Programm.

4.3.2 Vorteile der Nutzung von P4

Der Einsatz von P4 in Netzwerkgeräten birgt Potenzial für tiefgreifende Veränderungen im Bereich Netzwerktechnik, da P4-Implementierungen dank der Flexibilität und Vielseitigkeit der Sprache viele Vorteile für Nutzer*innen bereithalten können.

Einer der Vorteile von P4 ist, dass man eine unabhängiger Position gegenüber der verwendeten Hardware einnehmen kann. Ist zum Beispiel ein verwendetes Gerät zu leistungsschwach, kann ein P4-Feature auf einer leistungsstärkeren Plattform ähnlich implementiert, wenn nicht sogar wiederverwendet werden [6]. Einerseits wird die Möglichkeit geboten, Programmteile aus einer bestehenden, sehr umfangreichen P4-Implementierung zu entfernen, wenn diese für einen spezifischen Einsatzzweck nicht benötigt werden. Somit wird die Leistungsfähigkeit eines Gerätes nicht durch ungenutzte Protokolle beeinträchtigt. Außerdem kann, unter Einbeziehung von Skalierungs-Effekten, argumentiert werden, dass diese Vorgehensweise den Stromverbrauch eines Netzes senken kann [15]. Andererseits erlaubt es P4 auch, neue Features und Protokolle in einem Netz bereitzustellen. Dabei ist es unerheblich, ob das Feature nach den Vorgaben eines öffentlichen Standards entworfen oder von proprietärer Natur ist. Durch P4 können somit laut [6] viele neue Ideen, wie zum Beispiel ein selbst entworfenes Load-Balancing-Verfahren oder die Verwendung von netzinternen Tags, umgesetzt werden. Es entsteht auch der Vorteil, dass man Implementierungsfehler in P4-Programmen selbst beheben kann, ohne dabei auf Updates eines Geräteherstellers angewiesen zu sein. Ein weiterer Punkt ist die Möglichkeit, eigene Implementierungen schnell und einfach zu testen. Dies ist einerseits in Software unter Zuhilfenahme eines Software-Switches⁵ möglich. Andererseits gibt es laut [41] allerdings auch die Option, Implementierungen in Hardware, unter Verwendung des NetFPGA-Projekts, zu testen. Ein weiteres Feature, das durch P4 ermöglicht wird, ist die Erzeugung und Erfassung von Te-

⁵Behavioral Model Version 2 ist eine Referenzimplementierung eines P4-Software-Switches [40].

lemetriedaten in Echtzeit. Dies ermöglicht es, nach [6], eine umfangreiche Diagnose eines Netzwerks durchzuführen, da durch P4 für einzelne Netzwerkpakete Metadaten produziert werden können.

4.3.3 Nutzen für P4-Programmierer*innen

Abgesehen von Vorteilen, die sich durch eine Nutzung der Sprache ergeben können, hat P4 auch aus dem Blickwinkel eines Programmierers beziehungsweise einer Programmiererin einige Vorzüge.

Für den/die Programmierer*in ergibt sich dabei aus der Tatsache, dass man eine Vielzahl von Paketweiterleitungsarten durch P4 ausdrücken kann, große Flexibilität. Außerdem bietet P4 die Möglichkeit, komplexe Paketverarbeitungsverfahren mit den in der Sprache standardmäßig enthaltenen Grundoperationen zu realisieren. Dies führt dazu, dass Programme auf verschiedene Zielsysteme portiert werden können, solange diese ein und dieselbe Architektur unterstützen. Die Designentscheidung, dass der/die Programmierer*in keinerlei Interaktion mit dem Systemspeicher, wie es zum Beispiel durch Allokieren von Speicher der Fall wäre, ausgesetzt ist, vereinfacht den Entwurf von P4-Programmen noch weiter. Speicher, der innerhalb eines Programms, zum Beispiel für Internet Protocol-Adressen (IP), benötigt wird, wird durch den Compiler automatisch auf die darunterliegende Hardware gemappt. [3]

Viele dieser Designentscheidungen, die die Sprache betreffen, wurden bereits bei der erstmaligen Vorstellung von P4 getroffen [11]. Dennoch kam es im Laufe der Zeit zu mehreren Entwicklungsschritten, die in Kapitel 4.4 erörtert werden.

4.4 Evolution von P4

Wie in Kapitel 4.3 skizziert, kam es im Juli 2014 in Form eines Papers zu einer Präzisierung der Idee zu P4. Der Schritt hin zur Spezifikation dieser Programmiersprache mit der Versionsnummer 1.0.1 wurde dabei einige Monate später, im September desselben Jahres, vollzogen [12]. In den Jahren darauf wuchs die Sprache und deren Verwendung nach Angaben in [7] sehr stark. Dieser Wachstumsprozess führte dazu, dass das P4-Sprachkonsortium viel Feedback erreichte und daraus resultierend mit dem Entwurf einer neuen Version der Sprache begonnen wurde. Dieses Verfahren kam im Mai 2017, mit dem Erscheinen der ersten Spezifikation von P4₁₆, zu einem erfolgreichen Ende [42]. Die vorangegangene Version der Sprache, P4₁₄, lässt sich durch die tiefgestellten Zahl 14 von deren Evolution unterscheiden.

Bei der Weiterentwicklung zu P4₁₆ achtete man besonders darauf, bestimmte Entwicklungsziele zu verfolgen. Eines dieser Ziele war, dass P4₁₆ keine neue Sprache, sondern lediglich eine logische Evolution darstellen sollte. Darunter wird einerseits verstanden, dass die Sprache ihre grundlegenden Bestandteile und Konzepte beibehält, andererseits sollte die Sprache einfacher nutzbar gemacht werden. Dies wird durch eine klare, formal definierte Semantik erreicht. Zum Beispiel wurde in P4₁₄ nicht genau definiert, welche Folgen eine Berechnung von Werten unterschiedlicher Bitbreite oder der Aufruf des „drop“-Befehls hat. Diese und ähnliche Ungenauigkeiten sollten mit dem Sprachentwurf von P4₁₆ behoben werden. Eines der wichtigsten verfolgten Prinzipien war, die begrenzte Ausrichtung von P4 auf eine Gerätegruppe hin aufzubrechen, um damit eine wahre Zielunabhängigkeit zu erreichen. Dafür musste es möglich werden, neue Architekturen definieren und verwenden zu können. Die grundlegende Designentscheidung, die dafür getroffen werden musste, war die Trennung von Sprache und Architektur, die im Kapitel 4.6 beschrieben wird. Erst dieser Schritt führte zu einem Wandel und schlussendlich zu einer Flexibilisierung in diesem Bereich. Um etwaige Rückschritte im Evolutionsprozess zu verhindern, wurde die Einführung von neuen Konstrukten als Nicht-Ziel definiert. Das war notwendig, weil geplant war, den Umfang der Sprache zu verringern. Dies wurde erreicht, indem man möglichst viele Schlüsselwörter aus der Spezifikation entfernte. Die hier beschriebenen Maßnahmen wurden gefasst, um P4 endgültig als eine für die Netzwerkdomäne spezifische Sprache festzulegen und auszurichten. [15]

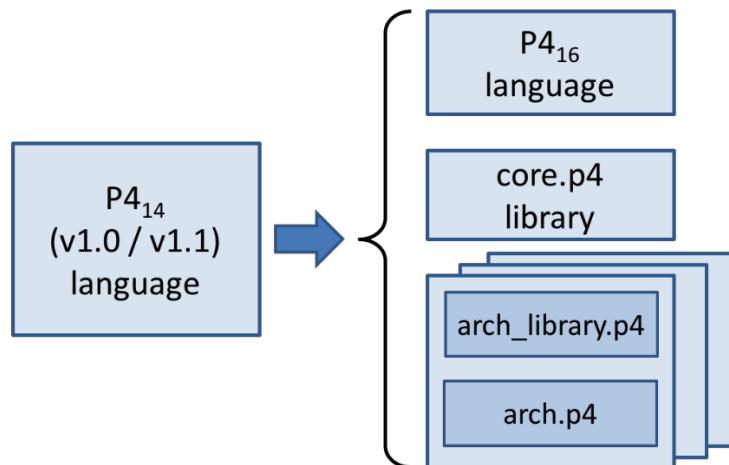


Abbildung 5: Strukturelle Veränderungen zwischen P4₁₄ und P4₁₆ nach [3]

Durch die Änderungen der Spezifikation von P4 kam es auch zu einer Veränderung der Syntax und der Art und Weise in der Programmierer*innen mit der Programmiersprache arbeiten und interagieren. Änderungen an der Semantik bewirkten, dass P4₁₆-Programme nicht mehr rückwärtskompatibel sind und daher auch nicht mehr durch einen P4₁₄-Compiler verstanden werden. In Abbildung 5 ist illustriert, dass viele Bereiche der Sprache in einzelne Module unter- bzw. verteilt wurden. P4 ist nunmehr in zwei übergeordneten Bereiche

aufgeteilt. Der erste übergeordnete Bereich ist offiziell Teil der Sprache und wird daher nur selten verändert. Dieser Teil umfasst den Sprachkern und die offizielle Bibliothek. In letztere wurden Objekte versetzt, welche nicht in jeder Architektur von Nöten sind. Den zweiten übergeordneten Bereich bilden Sprachbestandteile, die für die verwendete Architektur angepasst sind. Diese enthalten zum Beispiel Bibliotheken von Hardwareherstellern, die auf diese Weise ihre gerätespezifischen Funktionen durch P4-Konstrukte ausdrücken und dadurch dem/der Programmierer*in zur Verfügung stellen. Allgemein wurde es durch die Einführung von Bibliotheken möglich, bereits programmierte Features einfacher wiederzuverwenden und in neuen Programmen zu nutzen. [3]

4.5 Zusammenspiel der Komponenten

Bisher wurde hauptsächlich über den Nutzen, die Herkunft und die Ziele von P4 referiert. Jedoch ist um P4-Programme verstehen zu können auch ein Überblick über alle beteiligten Komponenten notwendig. Diese Veranschaulichung soll helfen, das Zusammenspiel der Bestandteile besser nachvollziehen zu können. Damit wird außerdem auch eine Basis für weitere, detailliertere Beschreibungen der einzelnen Komponenten in Folgekapiteln geschaffen. In Abbildung 6 findet sich eine schematische Darstellung aller beteiligten Komponenten.

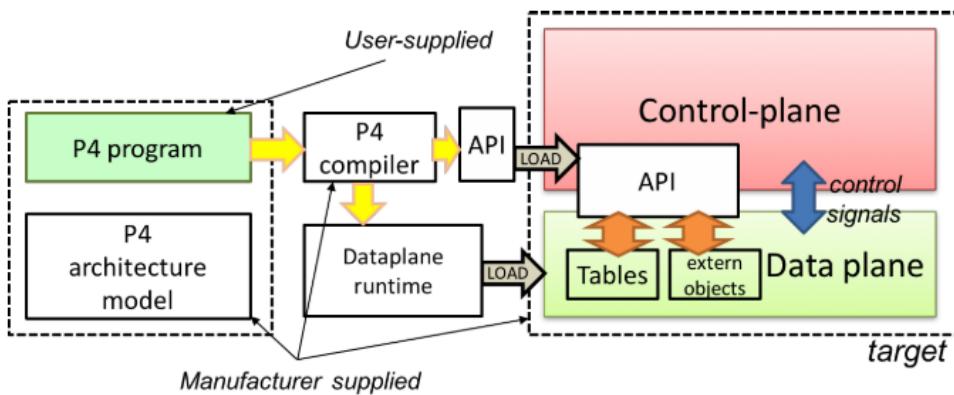


Abbildung 6: Schema des P4-Workflows [3]

Für die Arbeit eines Programmierers/einer Programmiererin müssen drei grundlegende Elementen vorhanden sein. Die wichtigste Komponente ist dabei das „Target“, das fortlaufend Zielgerät genannt wird. Das Zielgerät kann dabei sowohl eine Hardware-Plattform als auch ein Softwareimplementierung sein. Die gewählte Plattform gibt nämlich die beiden anderen Komponenten, den Compiler und das Architekturmodell vor, da diese in einer Abhängigkeit voneinander stehen [3]. Auf das Thema Architekturmodelle wird in Kapitel 4.6 genauer eingegangen. Details über Architektur und die grundlegende Funktionalität

des Compilers finden sich hingegen in Abschnitt 4.7. Das durch den/die Programmierer*in auf Basis des Architekturmodells geschriebene P4-Programm wird durch Zuhilfenahme des passenden Compilers verarbeitet. Dieser erzeugt nach [3] genau zwei Artefakte.

Einerseits wird eine Data-Plane-Konfiguration in Form einer Binärdatei erzeugt. Diese übernimmt die Interaktion mit der darunterliegenden Logik des Zielgeräts. Die Konfiguration enthält Anweisungen zur Erstellung der benötigten Tabellen und anderer, im P4-Programm beschriebenen Objekte auf dem Zielgerät. Andererseits wird auch noch ein Application Programming Interface (API) erstellt, das für die Verwaltung der Objekte auf der Data-Plane durch die Control-Plane verwendet werden kann. Über diese Kommunikationsschnittstellen können anschließend unter anderem die Inhalte der Tabellen verändert und mit externen Objekten, wie zum Beispiel Metern, interagiert werden. Bezug wird auf diese Schnittstellen in Kapitel 4.8 genommen. [3]

4.6 Architekturen

In Kapitel 4.5 wurde bereits die Wichtigkeit von Architekturmodellen unterstrichen. In diesem Kapitel folgt nun eine detaillierte Vorstellung und Beschreibung dieses Themas. Diese soll dem/der Leser*in helfen, die Vorteile von und die Erfordernisse für Architekturmodelle nachzuvollziehen. Überdies sind in diesem Kapitel auch noch verschiedene aktuell existierende Architekturmodelle veranschaulicht. In Zuge dieser Vorstellung wird auch auf die neueste Entwicklung in diesem Bereich eingegangen.

4.6.1 Allgemeines

Vor der weiteren Behandlung dieses Themas ist es wichtig, ein grundlegendes Wissen darüber zu haben, was eine Architektur beziehungsweise ein Architekturmodell ist und wie dieses definiert wird. In [15] heißt es sinngemäß, dass eine Architektur eine spezifische Auswahl von externen Objekten, deren Schnittstellen und P4-programmierbaren beziehungsweise nicht-P4-programmierbaren Komponenten ist, die dem/der P4-Programmierer*in zur Verfügung stehen. [43] bietet mehrere Sichtweisen und es werden die folgenden Beschreibungen genannt: Eine Architektur ist eine abstrakte Betrachtungsweise der gesamten Verarbeitungspipeline eines Zielgeräts. Das P4-Programm zielt dabei auf ebendiese Pipeline ab, da diese die programmierbaren Komponenten enthält. Außerdem kann eine Architektur als Abstraktion, wie ein/eine Programmierer*in über die darunterliegende Plattform denkt, gesehen werden. Bezuglich der letzten Aussage gibt [43] zu bedenken, dass sich die Komponenten der verwendeten Architektur gegenüber den physischen Komponenten des Zielgeräts unterscheiden können. Zusammenfassend lässt sich sagen, dass eine

Architektur den möglichen Aufbau und die Bestandteile der Verarbeitungspipeline eines Zielgeräts logisch zu beschreiben versucht, um dem/der Programmierer*in eine abstrakte und dadurch einfach zu benutzende Entwicklungsplattform bieten zu können.

Aus dem vorangegangenen Absatz kann bereits herausgelesen werden, warum bei P4 die Verwendung einer Architektur eine grundlegende Voraussetzung ist. Aus der Sicht eines Programmierers/einer Programmiererin, bietet ihm/ihr diese die Möglichkeit, mit dem Zielgerät zu interagieren, ohne dabei Detailwissen über dieses besitzen zu müssen. Eine Architektur ist für einen/eine Programmierer*in also eine Beschreibung der Komponenten eines Zielgeräts, die in einem P4-Programm verwendet werden können. Dieser Umstand ermöglicht es, dass sobald zwei verschiedene Zielgeräte dieselbe Architektur unterstützen, das Programm auch auf beiden Geräten implementiert werden kann, ohne das vorab Änderungen am Programm vorgenommen werden müssen [44]. Die Verwendung einer bestimmten Architektur kann die Wiederverwendbarkeit eines Programms daher auch in großem Maße beeinflussen.

In Kapitel 4.4 wurde bereits beschrieben, dass es in P4₁₆ keine Beschränkung auf eine bestimmte Architektur mehr gibt und es im Zuge der Evolution von P4₁₄ zu einer Trennung von Architektur und Zielgerät kam. Die Beschränkung auf einen Gerätetyp beziehungsweise eine Architektur war auch der Grund, dass diese Trennung vollzogen wurde [3]. Denn bisher konnte P4 laut [7] nur mit einer Gerätengruppe betrieben werden. Die besagte Gruppe umfasst programmierbare Netzwerk-Switches. Diese Geräte mussten für eine Verwendung unter P4₁₄ eine ähnliche Architektur, wie sie in der Spezifikation [45] beschrieben wird, aufweisen. Gemäß [15] unterstützt P4 durch die Trennung von Architektur und Zielgerät nun neben Switches, die zum Beispiel auf Barefoots Tofino-Chips setzen [38], auch andere Gerätetypen wie Router, Network Interface Cards (NICs) und FPGAs [41]. Das aktuelle Portfolio an unterstützten Zielen beschränkt sich aber nicht nur auf physische Netzwerkgeräte, sondern erstreckt sich auch bereits in den Bereich der Softwareimplementierung hinein. Hier finden vor allem der P4-Standard-Softwareswitch auf Basis des Behavioral Model Version 2 (BMv2) und der Open vSwitch [46] Verwendung.

Nun stellt sich allerdings die Frage, aus welchen Bestandteilen eine solche Architektur üblicherweise aufgebaut ist. Hierzu erwähnt [44] einige Komponenten, die meistens umgesetzt werden, wie zum Beispiel einen Parser, eine Ingress- und Egress-Pipeline und einen Deparser. Die wichtigsten dieser Komponenten werden nun kurz vorgestellt.

Allem voran kommt meist der Parser zum Einsatz, da in diesem die Programmierung hinsichtlich Form, Struktur und Abfolge der Header-Felder vorgenommen wird. Darauf folgt eine Ingress-Pipeline. In dieser formuliert der/die Programmierer*in die zur Verfügung stehenden Tabellen und den genauen Verarbeitungsalgorithmus für Netzwerkpakete. Hierauf folgt gewöhnlich eine Egress-Pipeline, welche ähnliche Aufgaben wie die Ingress-Pipeline

erfüllt. Innerhalb dieser beiden Komponenten wird üblicherweise entschieden, welchem Port ein Paket zugeordnet werden soll. Die meist letzte Komponente bildet der Deparser, in dem die Struktur der ausgegebenen Pakete definiert wird. Die gerade angeführte Anzahl und Abfolge von Komponenten in P4-Programmen ist allerdings nicht fix vorgegeben, sondern kann sich je nach Architektur stark unterscheiden. [43]

Um diese Variabilität zu verdeutlichen, werden in den folgenden Abschnitten die wichtigsten Architekturen vorgestellt.

4.6.2 PISA

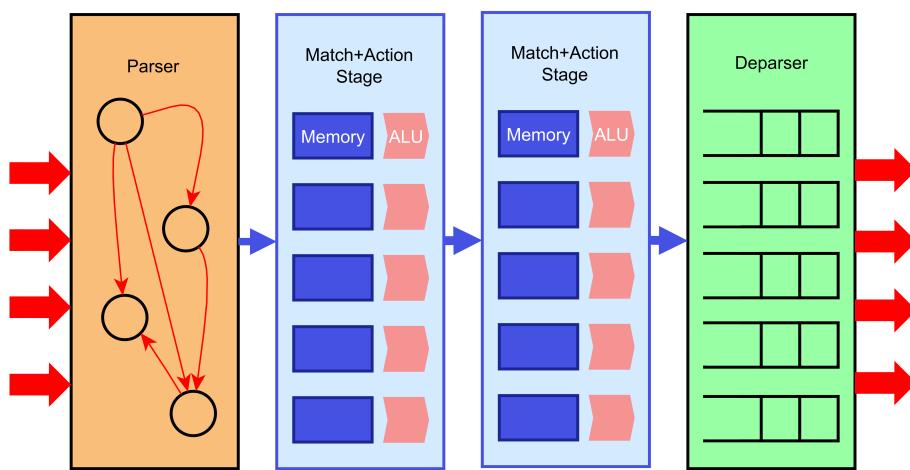


Abbildung 7: Schemenhafte Darstellung des PISA-Modells nach [15]

Das erste zu erwähnende Data-Plane-Modell ist die Protocol Independent Switch Architecture, kurz PISA. Hierbei handelt es sich um eine „Single-Pipeline“-Architektur, da diese nur eine einzige Pipeline beinhaltet [43]. Das Modell besteht dabei aus einem Parser, einer „Match+Action“-Pipeline und einem Deparser. Die formale Trennung der gesamten Pipeline in jeweils eine Ingress- beziehungsweise Egress-Pipeline ist hier nicht zu finden. Die angeführten Komponenten und deren Abfolge sind in Abbildung 7 illustriert. Die Mehrzahl der programmierbaren Netzwerk-Switches folgt dieser Architektur [47]. Dies führte laut Ausführungen in [43] dazu, dass dieses Data-Plane-Modell unter P4₁₄ quasi zu der Standard-Architektur wurde. Denn neben Tofino-Chip-basierenden Switches [38] bauen auch Switches anderer Hersteller, wie zum Beispiel Geräte von Cisco [48], auf dieser Architektur auf. Laut [49] ermöglicht diese Architektur die Flexibilität einer programmierbaren Data-Plane, ohne dabei Abstriche hinsichtlich Leistungsfähigkeit und Stromverbrauch gegenüber traditionellen Geräten gleicher Chipgröße machen zu müssen. Abschließend muss noch erwähnt werden, dass diese Architektur eine Abwandlung des in der Spezifikation von P4₁₄ [45] beschriebenen Standards ist.

4.6.3 P4₁₄-Switch-Modell

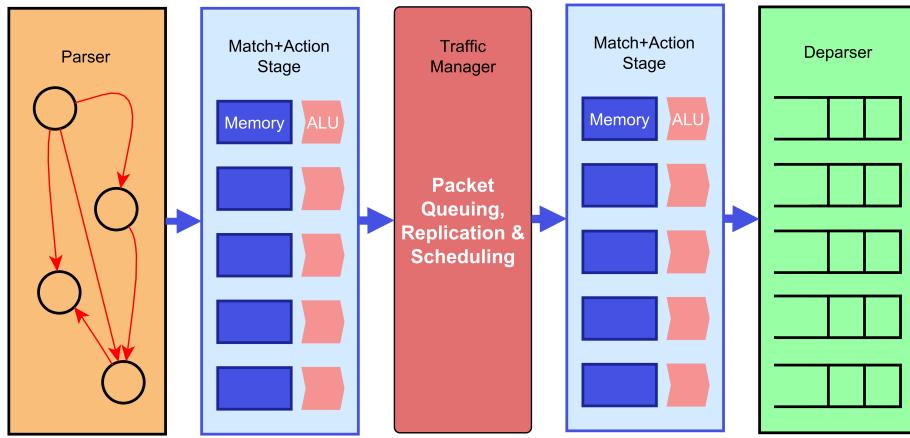


Abbildung 8: Schemenhafte Darstellung des P4₁₄-Switch-Modells nach [15]

In der Spezifikation von P4₁₄ [45] heißt es, dass das P4₁₄-Switch-Modell aus den Komponenten Parser, Ingress- und Egress-Pipeline und Deparser besteht. Des weiteren wird beschrieben, dass sich zwischen den zwei enthaltenen Pipelines ein Traffic-Manager befindet, welcher jedoch im Unterschied zu den anderen Komponenten nicht sehr ausführlich dokumentiert ist. In [15] wird dieser Traffic-Manager auch als Memory-Management-Unit oder Packet-Replication-Engine betitelt. Diese Bezeichnungen geben auch Aufschluss über die Funktion des Traffic-Managers innerhalb der Architektur. In der Spezifikation wird außerdem auch der Ablauf von und der Zusammenhang zwischen den Komponenten dieses Modells genauer beschrieben. Laut dieser generiert die Ingress-Pipeline für jedes Paket eine Egress-Spezifikation. Diese beinhaltet die Nummer des Ausgangsports an das das Paket später gesendet werden soll. Hierauf folgt der Traffic-Manager, der mehrere Instanzen dieses Pakets erzeugt, falls die Egress-Spezifikation mehr als einen Ausgangsport enthält. Außerdem wird im Traffic-Manager, falls dies notwendig ist, nun auch Buffering durchgeführt. Nach diesem Schritt wird in der Egress-Pipeline der „Match+Action“-Prozess fortgeführt. Die Spezifikation erwähnt zu diesem Schritt, dass üblicherweise der Ausgangsport eines Pakets in dieser Phase nicht mehr verändert wird. Das Verwerfen eines Pakets wird hier jedoch als eine Ausnahme für diese Faustregel aufgeführt. [45]

Die Komponenten des P4₁₄-Switch-Modells sind in Abbildung 8 veranschaulicht. Bei einer Gegenüberstellung dieser Illustration mit Abbildung 7 wird ersichtlich, dass der Traffic-Managers die Pipeline in einen Ingress- und Egress-Teil aufspaltet. Hieraus lässt sich auch der Zusammenhang zwischen dem PISA- und dem P4₁₄-Switch-Modell ablesen. Dieser liegt darin begründet, dass die zugrundeliegende Idee beider Architekturen vermutlich denselben Ursprung in [23] haben. Denn in [23] wird die Architektur eines Chips für Netzwerkswitches durch ein Modell beschrieben, dass sowohl PISA, als auch dem P4₁₄-Switch-Modell stark ähnelt.

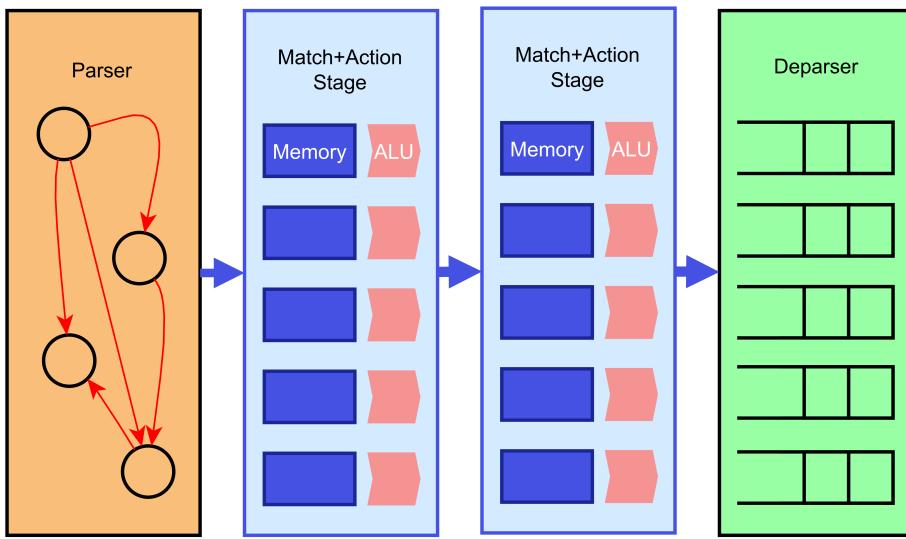


Abbildung 9: Schemenhafte Darstellung einer programmierbaren SmartNIC-Architektur nach [15]

Die Spezifikation von P4₁₄ lässt bezüglich der Gestaltung und Verwendung alternativer Modelle sehr viel Freiraum. In [15] wird neben PISA auch noch eine für SmartNICs [50] spezifische Architektur, die in Abbildung 9 dargestellt ist, als Beispiel für die Freiräume, welche die Spezifikation bietet, angeführt. Diese Architektur benötigt laut [15] keine Egress-Pipeline, da Geräte, die dieses Modell einsetzen, meist nur einen „Port“ in Richtung eines Servers, in Form eines Peripheral Component Interconnect Express-Anschlusses (PCIe), vorsehen. Die Architektur, die dadurch entsteht, ist identisch mit der des PISA-Modells (siehe Abbildung 7), da diese ebenfalls nur eine Pipeline aufweist. Daher kann gesagt werden, dass es die Spezifikation von P4₁₄ erlaubt, einzelne Komponenten der Standardarchitektur zu entfernen. Laut [15] ist es aber nicht möglich, zusätzliche Komponenten, wie zum Beispiel einen zweiten Parser oder Deparser, in das P4₁₄-Switch-Modell zu integrieren. Eine solche Möglichkeit würde zum Beispiel den Einsatz von symmetrischen und damit unabhängigen Parsern und Deparsern erlauben.

4.6.4 V1-Modell

Eine weitere wichtige Architektur ist das V1-Modell, das streng genommen keine eigenständige Architektur darstellt, da es mit dem P4₁₄-Switch-Modell identisch ist [40]. Daher ist das V1-Modell auch durch Abbildung 8 graphisch repräsentiert. Diese Ähnlichkeit bedingt laut [15] auch eine Kompatibilität der besagten Data-Plane-Modelle untereinander. Besonders wichtig hier anzuführen ist, dass das V1-Modell Kern der Referenzplattform des P4-Konsortiums, dem BMv2-Software-Switch, ist. Demzufolge ist das V1-Modell auch Teil des Referenzcompilers. Die Rolle des V1-Modells als Referenzarchitektur bewirkt überdies, dass P4₁₄-Programme, die für das P4₁₄-Switch-Modell geschrieben wurden, nun auch mit dem V1-Modell benutzt werden können. [40]

4.6.5 PSA

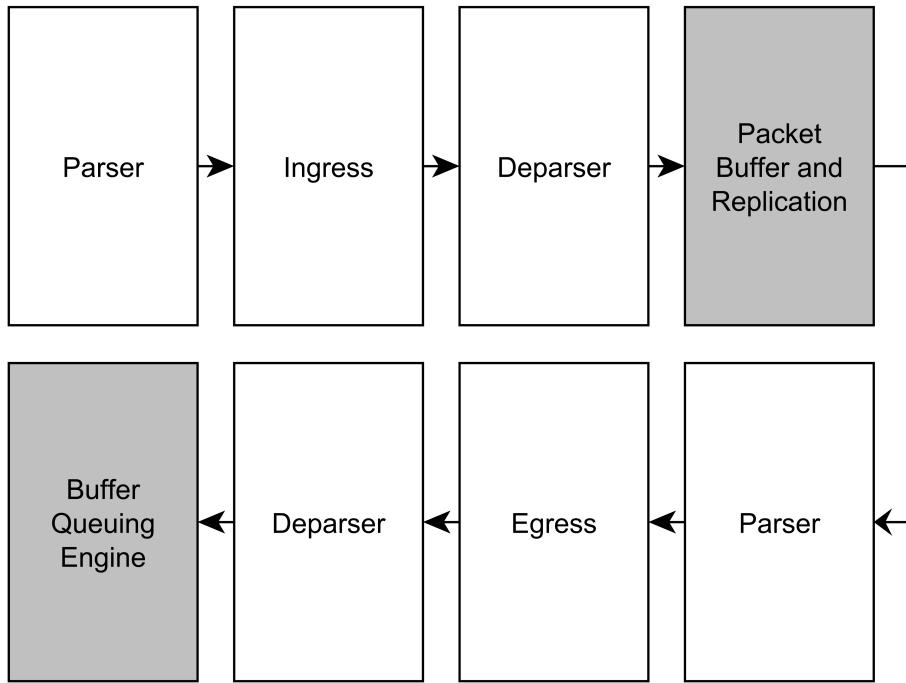


Abbildung 10: Schemenhafte Darstellung des PSA-Modells nach [51]

Das letzte hier vorgestellte Data-Plane-Modell ist die Portable Switch Architecture (PSA). Aus dem Namen lässt sich bereits entnehmen, dass auch diese Architektur primär für Netzwerkswitches gedacht ist [51]. PSA enthält hauptsächlich Bausteine, die ebenfalls in anderen Architekturen enthalten sind. Diese Aussage trifft im Besonderen auch auf die bereits in diesem Kapitel genannten Modelle zu. In Abbildung 10 sind die für das PSA-Modell benötigten Komponenten zu sehen. Die Architektur setzt sich demnach aus sechs programmierbaren und zwei nicht-programmierbaren Blöcken zusammen [51]. Nicht-programmierbar bedeutet, dass diese durch den/die Programmierer*in nicht verändert werden können, sondern dass deren Funktionen durch das Zielgerät fix vorgegeben sind. Dies bedeutet daher auch, dass diese Blöcke für die Tätigkeit des Programmierers/der Programmiererin nicht von direktem Interesse sind. Ein direktes Interesse kann jedoch bestehen, wenn Blöcke über Schnittstellen verfügen, die in einem P4-Programm die Nutzung spezieller Funktionen des Zielgeräts ermöglichen. Die Funktionen, die diese beiden nicht-programmierbaren Blöcke umsetzen, entsprechen in etwa denen des Traffic-Managers aus dem P4₁₄-Switch-Modell. Die Spezifikation des PSA-Modells [51] beschreibt den Ablauf der Behandlung eines Datenpakets durch die Blöcke der gesamten Architektur wie folgt: Ankommende Nachrichtenpakete werden im Parser hinsichtlich deren Struktur validiert und nach einer erfolgreichen Validierung der Ingress-Pipeline übergeben. In dieser erfolgen ihrerseits die „Match+Action“-Verarbeitungen, bevor die Pakete an den Ingress-Deparser übergeben werden. Der P4-Code, welcher den Deparser beschreibt, bestimmt daraufhin,

welche Paketinhalte und dazugehörige Metadaten an den Paket-Buffer übergeben werden. Dieser Buffer ist Teil des ersten nicht-programmierbaren Blocks, der auch für die optionale Replizierung von Paketen verantwortlich ist. Diese Vervielfältigung eines Pakets wird allerdings nur vorgenommen, falls ein Paket an mehrere Ausgangsports adressiert ist und daher aus diesen heraus versendet werden soll. Im nächsten Schritt, dem Egress-Parser, wird pro ausgehendem Nachrichtenpaket dessen Struktur analysiert, bevor es weiter an die Egress-Pipeline in Richtung Egress-Deparser übergeben wird. Sobald das Datenpaket den Egress-Deparser verlässt, wird es an den letzten Block übergeben. Dieser ist wiederum nicht-programmierbar und implementiert unter anderem das Queuing von Paketen, bevor diese das Netzwerkgerät verlassen. Die Spezifikation der PSA-Architektur beschränkt sich allerdings nicht nur auf die Beschreibung dieser Blöcke [51]. Laut der Spezifikation umfasst die der Architektur zugehörende P4-Bibliothek zum einen Typen, welche für jede Implementierung von PSA notwendig sind und auch teilweise gerätespezifische Eigenschaften aufweisen. Ein Beispiel für einen gerätespezifischen Datentypen ist `PortIdUint_t`, der laut [51] für die Nummerierung von Ports auf einem Gerät benutzt wird. Dieser kann unter Umständen auf einem Gerät acht und auf einem anderen wiederum 16 Bit lang sein. Hinzukommend enthält eine PSA-Bibliothek laut Spezifikation auch noch externe Objekte, die bereits erwähnt wurden. Teil dieser Bibliothek ist außerdem eine Beschreibung von „Paket-Pfaden“, die in einer Implementierung von PSA beschrieben sein müssen [51]. In Abbildung 11 sind diese Pfade in Form von Kanten mit der Bezeichnung „Pfad“ dargestellt. Die Spezifikation von PSA hat zum Ziel, dass Programme, die für diese Architektur geschrieben werden, mit möglichst vielen P4₁₆-Netzwerkswitches kompatibel sind, die diese Architektur implementieren [51]. Ein Ziel, das durch die „P4.org Architecture Working Group“ bei dieser Architektur jedoch nicht verfolgt wird, ist laut [43] „Performance portability“. Damit ist gemeint, dass ein spezifisches P4-Programm auf allen Geräten, die dieses Data-Plane-Modell unterstützen, mit gleicher Leistung läuft beziehungsweise eine gleich hohe relative Durchsatzrate erreicht.

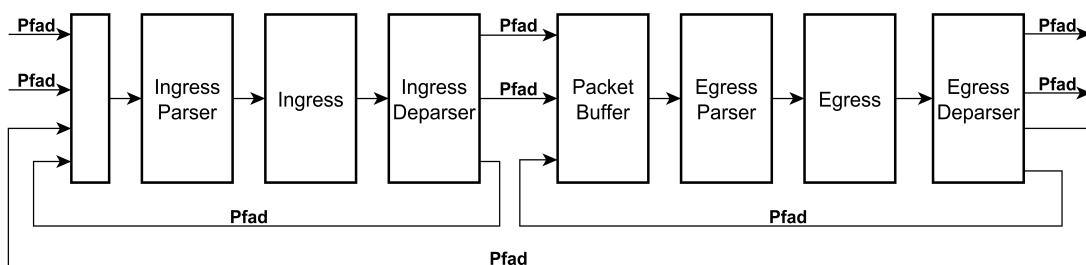


Abbildung 11: Illustration der „Paket-Pfade“ im PSA-Modell [51]

4.6.6 Andere Architekturen

Abschließend ist zu erwähnen, dass die in diesem Kapitel erwähnten Architekturen keine vollständige Übersicht aller verfügbaren Data-Plane-Modelle darstellt. Denn für viele mit P4₁₆ programmierbare Zielgeräte gibt es eine speziell auf das Ziel abgestimmte Architektur. Beispielhaft seien hier die SimpleSumeSwitch-Architektur für die Plattform NetFPGA-SUME [52] und die Architekturen XilinxSwitch und XilinxStreamSwitch für Produkte von Xilinx [53] erwähnt.

4.7 P4-Compiler

In Kapitel 4.5 wurde bereits auf das Zusammenspiel aller Komponenten bei P4 eingegangen. Eine dieser Komponenten ist der P4-Compiler. In diesem Kapitel wird nun genauer auf den Compiler eingegangen, ohne dabei zu weit ins Detail gehen zu wollen. Denn Ziel dieser Erläuterungen ist es, ein prinzipielles Verständnis über den Aufbau, die Arbeitsweise und Aufgabe des P4-Referenzcompilers [54] zu gewinnen, um dadurch P4 besser zu verstehen.

4.7.1 Entwicklung des P4-Referenzcompilers

In diesem Kapitel wird die Entwicklung des „p4c“ genannte P4₁₆-Referenzcompiler behandelt [54]. In [7] wird erwähnt, dass der bereits existierende P4₁₄-Compiler nicht mehr für P4₁₆ adaptiert werden konnte, ohne dabei einen großen Mehraufwand zu verursachen. Diese Tatsache bedingte auch die Entscheidung, dass ein neuer Compiler auf Basis von C++11 entworfen und implementiert werden sollte [7]. Außerdem beschreibt [7], dass der Compiler zeitgleich mit der Spezifikation von P4₁₆ implementiert wurde. Dies bedingte, dass sich der Implementierungsprozess und die Spezifizierung gegenseitig ergänzten und dadurch Lücken in der Spezifikation besser erkannt und behoben werden konnten.

In [7] sind einige Ziele für die Entwicklung definiert, die durch den neuen Compiler umgesetzt werden sollen:

- Es sollten vergangene, aktuelle und alle zukünftigen Versionen von P4 unterstützt werden.
- Der Compiler sollte mehrere Backends unterstützen. Passende Backends sollten beispielsweise für die folgende Zielgeräte ermöglicht werden: ASICs, NICs, FPGAs, Software-Switches

- Anknüpfungspunkte für Software-Entwicklungswerzeuge sollten bereitgestellt werden. Dies sollte die Entwicklung von Debugging- und Testtools ermöglichen.
- Erleichterung der Erstellung von Architektur-spezifischen Compilern, da das Frontend des Compilers frei verfügbar sein sollte.
- Erweiterbarkeit hinsichtlich der Architektur des Compilers. Da hier das Hinzufügen neuer Durchlaufphasen, in der Literatur [7] auch „Passes“ genannt, anderer Optimierungsschritte und neuer Midends ermöglicht werden soll.
- Hinzukommend sollte der Compiler auf modernen Standards in der Software- beziehungsweise Compiler-Entwicklung fußen. Hierzu zählen zum Beispiel unveränderbarer Zwischencode, basierend auf einem Visitor-Software-Pattern und starke Typisierung.

4.7.2 Architektur des P4-Referenzcompilers

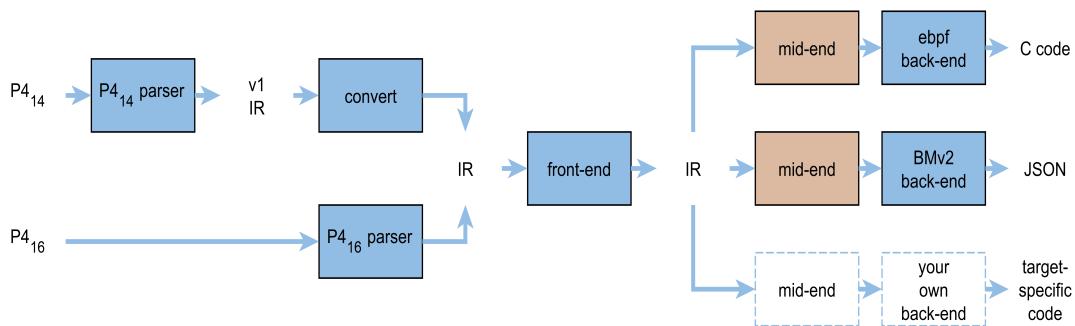


Abbildung 12: Schematische Darstellung des Datenflusses des P4₁₆-Referenzcompilers nach [55]

In diesem Kapitel wird auf die Architektur und die Bestandteile des P4₁₆-Referenzcompilers eingegangen.

Der zugrundeliegende Datenfluss und die beteiligten Komponenten sind in Abbildung 12 ersichtlich. Darin ist zu sehen, dass der Compiler mit einem Parser für jeweils P4₁₄ und P4₁₆ beginnt. Außerdem ist erkennbar, dass in P4₁₄ geschriebene Programme in P4₁₆ umgewandelt werden können [54]. Die Architektur des umgewandelten Programms entspricht daraufhin der des V1-Modells, da diese nahe miteinander verwandt sind. [7]

Die nachfolgende Stufe in Abbildung 12 bildet das Frontend. Dieses führt alle zielunabhängigen Verarbeitungen durch. Hier durchläuft der gesamte Code standardmäßig 25 Phasen. Diese dienen der Optimierung, der syntaktischen und semantischen Überprüfungen des Codes. [7]

Das darauf folgende Midend wird dafür genutzt, um den aus dem Programmcode generierten Zwischencode weiter an ein spezifisches Zielgerät anzupassen. Dies geschieht dadurch,

dass der/die Nutzer*in aus einer vordefinierten Bibliothek verschiedene Durchlaufphasen wählen kann, welche dann auf den Code angewendet werden. Außerdem gibt es für den/die Nutzer*in die Möglichkeit, selbst Durchlaufphasen zu entwerfen und diese in eine eigene Compiler-Implementierung hinzuzufügen. [7]

Abschließend findet sich in Abbildung 12 das Backend. In diesem werden nur zielspezifische Verarbeitungen durchgeführt. Die wichtigste Aufgabe ist hierbei die Erzeugung des zielspezifischen Codes. Der Programmcode wird durch die Nutzung von 25 Durchlaufphasen auf das Zielgerät abgestimmt und optimiert. Wichtig zu erwähnen ist, dass in diesem Schritt die zielspezifischen, externen Objekte behandelt und auf diese Art und Weise auf dem Zielgerät nutzbar gemacht werden. Der Referenzcompiler hat grundsätzlich zwei besonders erwähnenswerte Backends. Das erste bildet das Backend für BMv2, das ein Konfigurationsprogramm in Form einer JavaScript Object Notation-Datei (JSON) erzeugt. Diese Datei kann daraufhin mit dem P4-Standard-Softwareswitch genutzt werden. Das zweite Backend ist das „ebpf“-Backend. Dieses generiert C-Code, der wiederum zu extended Berkeley Packet Filter (eBPF) konvertiert und damit in einem Linux-Kernel lauffähig gemacht werden kann. [55]

Zwischen den einzelnen, bereits angeführten Komponenten des Compilers befindet sich jeweils Zwischencode, der den Inhalt des P4-Programms in einer transformierter Darstellung beinhaltet. In Abbildung 12 werden die einzelnen Instanzen des Zwischencodes, auch Intermediate Representation (IR) genannt, mit der Abkürzung IR gekennzeichnet. Laut [7] wurde für den Referenzcompiler beschlossen, dass ein neuer Zwischencode entworfen werden muss, da der unter dem P4₁₄-Compiler genutzter Zwischencode nicht mehr in der Lage war, die große Anzahl an verschiedenen Zielgeräten zu unterstützen. In [7] werden Ziele erwähnt, die bei dem Entwurf des neuen Zwischencodes verfolgt wurden. Grundsätzliche Ziele waren, dass diese IR abstrahierend, leicht verständlich und für die verschiedensten Backends einfach erweiterbar sein sollte. In [55] wird die Vereinfachung des Debugging von selbst-implementierten Compilern als ein Grund für diese Ziele genannt. Genauere Informationen bezüglich des Zwischencodes sind in [7] zu finden.

4.7.3 Compiler-Projekte

Seit der Implementierung des P4-Referenzcompilers entstanden einige Projekte, welche Teile des Compilers verwenden, um bestimmte Eigenschaften von P4 zu verbessern. Zwei dieser Projekte werden in diesem Kapitel kurz vorgestellt.

In [56] wird das Projekt P4LLVM vorgeschlagen. In diesem wird die Idee aufgegriffen, den regulären Zwischencode nachgeordnet noch in einen anderen, effizienteren Zwischencode

zu übersetzten. Der in diesem Projekt verwendete Zwischencode ist Low Level Virtual Machine (LLVM). Dieser ermöglicht es, einen besseren Optimierungsgrad des P4-Programms zu erreichen. [56]

Ein zweites erwähnenswertes Projekt ist Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD). In [57] wird dieses Compilersystem vorgeschlagen, um die Vorteile der höheren Programmiersprache P4 mit einer abstrakten, zielunabhängigen Open Data Plane-API (ODP) zu verschmelzen. Zuerst wird hier das Ergebnis des P4₁₆-Referenzcompiler-Frontends in Datapath-Logik, eine IR, übersetzt. Anschließend kompiliert der Kerncompiler von MACSAD diesen Zwischencode mithilfe der passenden ODP-API für ein spezifisches Zielgerät. Dadurch soll es leichter möglich sein, ein Programm für verschiedene Zielgeräte zu übersetzen. [57]

Dieses System wird in [49] implementiert. Darin wird geschlussfolgert, dass die Leistungsfähigkeit der Test-Plattform dieses Projekts ungefähr mit der anderer Software-Switches, zum Beispiel Open vSwitch [46] vergleichbar ist.

4.8 P4Runtime

In Kapitel 4.5 wurde erwähnt, dass durch das Kompilieren eines P4-Programms eine API erstellt wird. Diese dient dem Nachrichtenaustausch zwischen der Control- und Data-Plane. Die API kann jedoch verschieden ausgestaltet sein. Ein Möglichkeit diese umzusetzen ist der Entwurf einer rudimentären Kommandozeileneingabe, wie sie zum Beispiel bei BMv2, für eine schnelle und einfache Konfiguration genutzt wird [40]. Eine andere Möglichkeit bietet sich in Form der P4Runtime, deren aktuellste Version in [58] spezifiziert ist. Diese läutet laut [59] eine neue Generation von API zwischen Control- und Data-Plane ein. In diesem Abschnitt werden die Grundlagen dieser vielseitig einsetzbaren API-Technologie vorgestellt.

Zweck der Entwicklung dieser Technologie war der Entwurf und die Standardisierung einer hersteller- und protokollunabhängigen API. Diese ist für den Einsatz mit Data-Planes gedacht, die entweder durch P4 ausgedrückt werden können oder mit P4 definiert wurden [58]. Dies bedeutet laut [58], dass auch nicht programmierbare Netzwerkgeräte mit P4Runtime genutzt werden können. Darüber hinaus ermöglicht diese API die Nutzung folgender Möglichkeiten zur Laufzeit eines Gerätes:

- Das in einem P4-Programm definierte Paketweiterleitungsverfahren kann auf das Gerät transferiert werden und auf diese Weise dessen Verhalten verändern [60].
- Die Manipulation von Tabellen, deren Einträge und der gesamten Verarbeitungspipeline des Geräts [61].

- Das direkte Empfangen und Versenden von Datenpaketen [42].
- Die Interaktion mit externen Objekten [42].
- Unterstützung einer Master-Slave-Architektur der Control-Plane, um eine hohe Verfügbarkeit und Ausfalltoleranz zu erreichen. Dadurch ergibt sich unter anderem auch die Möglichkeit einen am Gerät eingebetteten SDN-Controller und mehrere externe SDN-Controller zu verwenden. [60]

Allgemein ausgedrückt werden durch die P4Runtime Nachrichten und Semantik der Schnittstellen zwischen dem Controller und dem Zielgerät definiert [58]. Die Definition der Grundelemente dieser API kann in [62] eingesehen werden und erfolgt durch Dateien im Protobuf-Format. Durch den Inhalt der Protobuf-Dateien wird das Format der Nachrichten, die zur Kommunikation verwendet werden können und die Entitäten der Data-Plane mit denen interagiert werden kann, definiert [60]. Die auf diese Weise festgelegten Nachrichten können laut [58] über google Remote Procedure Calls (gRPCs) zwischen einem gRPC-Server am Zielgerät und einem gRPC-Client am Controller ausgetauscht werden. Sowohl der Authentifizierungsmechanismus, als auch die Erstellung des Programmcodes für Client und Server wird nach Ausführungen in [60] durch gRPC ermöglicht.

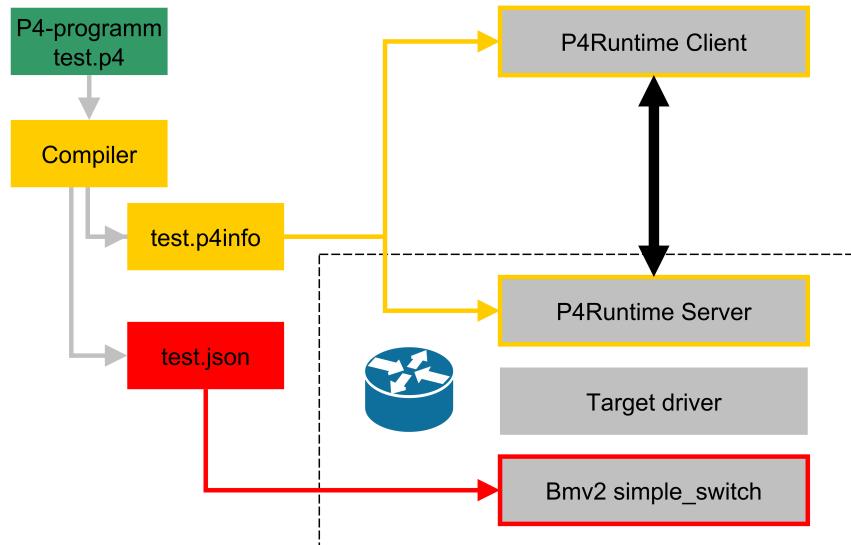


Abbildung 13: Übersicht über P4Runtime-Komponenten nach [60]

In Abbildung 13 werden die beiden für P4Runtime wichtigsten Artefakte, die bereits in Kapitel 4.5 erwähnt wurden, an einem Beispiel dargestellt. Die „P4 Device Config“, die das ziel- und architekturnspezifische Weiterleitungsverhalten und somit das „Programm“ des Zielgeräts enthält, ist in Rot dargestellt. Für das in diesem Beispiel verwendete Zielgerät, ein BMv2-Softwareswitch, liegt diese Konfigurationsdatei im JSON-Format vor. In Gelb ist die P4Info-Datei dargestellt. Diese enthält P4-Entitäten, welche schlussendlich durch die API angesprochen werden können [58]. Eine solche Entität kann zum Beispiel

eine Instanz einer im P4-Programm definierten Tabelle sein. Allgemein gesehen muss diese Datei sowohl auf dem Runtime-Server als auch auf dem Runtime-Client vorhanden sein, um beispielsweise die Konfiguration der Tabellen des Zielgeräts zu erlauben. Ist dies nicht der Fall, besteht keine gemeinsame Kommunikationsbasis und eine Konfiguration des Zielgeräts durch P4Runtime ist nicht möglich. Es ist jedoch nicht zwingend erforderlich, dass die Konfigurationsdatei eines P4-Gerätes dem/der Nutzer*in zur Verfügung stehen muss [58]. Dies kann zum Beispiel der Fall sein, wenn der Hersteller eines Gerätes das bereits fest vorinstallierte P4-Programm nicht offenlegen möchte, weil der Inhalt des Programms Geschäftsgeheimnis ist. Ein weiterer Grund, warum die Konfigurationsdatei nicht verfügbar ist, kann sein, dass das Weiterleitungsverhalten eines Geräts nicht durch P4 vorgegeben ist, sondern auf anderem Wege, zum Beispiel durch eine fixe Hardwarefunktionen, definiert ist.

Die in diesem Kapitel enthaltenen Ausführungen sind eine kurz gehaltene Zusammenfassung des Themas P4Runtime. Detailliertere Ausführungen zu diesem Thema finden sich in [58], [61], [42], [59], [62] und [60]. Es ist jedoch sehr wichtig, die grundlegende Funktionsweise dieser API zu erwähnen, weil sie sehr stark mit P4 verknüpft ist und bei neuen Entwicklungen in der Netzwerktechnik [63] zum Einsatz kommt.

4.9 P4-Sprachkomponenten

In diesem Kapitel soll ein Verständnis für die Sprachbestandteile von P4 gewonnen werden. Die hier vorgestellten Sprachkonstrukte sind elementar für das Verständnis von allen folgenden Ausführungen praktischer Natur. Als kontextuelle Grundlage für dieses Kapitel dienen alle bereits im Kapitel 4 vorgenommenen Erläuterungen, im Speziellen die des Kapitels 4.6.

4.9.1 Allgemeines

P4 wird in [7] als eine relativ einfache, statisch typisierte Sprache vorgestellt, deren Syntax sich sehr an der weit verbreiteten Programmiersprache C orientiert. Darauf, dass Objekte vor der Verwendung definiert werden müssen und auf weitere Parallelen zwischen diesen beiden Sprachen wird in [15] hingewiesen.

```

1 #define <Makro_ohne_Parameter>
2 #include <Datei>
3 #include "Datei"

```

Listing 1: Präprozessor-Befehle [3]

Auch die Benutzung von P4-Präprozessor-Befehlen unterscheidet sich in der Handhabung für den/die C-Programmierer*in nur gering. Wie in Listing 1 zu sehen, gibt es auch in P4 die Möglichkeit, den Compiler mehrere Quelldokumente in ein Programm einbinden zu lassen. Weitere Befehle, die auch im C-Präprozessor enthalten sind, werden laut P4₁₆-Spezifikation [3] unterstützt, wie zum Beispiel **#define**. [7] beschreibt, dass die meisten P4-Programme mit einem **#include**-Befehl beginnen, um die Standardbibliothek und die jeweilige Zielarchitektur-Bibliothek einzubinden. Auch können selbst geschriebene Bibliotheken und Programmblöcke auf diese Weise eingebunden werden.

4.9.2 Datentypen

```

1 8w180          //8-bit unsigned Wert:180
2 8w0b10110100  //8-bit unsigned Wert:180
3 32w0xB4       //32-bit unsigned Wert:180
4 32s0xB4       //32-bit signed Wert:180

```

Listing 2: Integer-Literale

Bevor in diesem Abschnitt die wichtigsten Datentypen und deren Verwendung vorgestellt werden, ist es wichtig, zuerst auf die Repräsentation und Schreibweise von Integer-Literalen einzugehen. Neben der Standardmöglichkeit, Zahlen durch das Dezimalsystem auszudrücken, gibt es auch die Möglichkeit, eine hexadezimale, oktale oder duale Basis zu verwenden. In Listing 2 sind Beispiele für die verschiedenen Schreibweisen enthalten. Dabei ist zu erkennen, dass die Zahl, welche vor dem ersten Buchstaben steht, die Länge des Literal-Wertes in Bit angibt. Darauf folgt entweder ein **w** oder ein **s**. Dadurch wird angezeigt, ob ein Wert kein beziehungsweise ein Vorzeichen enthält. Anschließend gibt eine Folge aus zwei Zeichen, beginnend mit einer Null, die Basis des Literals an. Für hexadezimale Zahlen folgt nach der Null der Buchstabe **x**, für oktale Zahlen ein **o** und für Zahlen des Binärsystems ein **b**. Wie in Listing 2 zu sehen, werden diese beiden Zeichen für Zahlen des Dezimalsystems jedoch ausgespart. Danach folgt der Zahlenwert durch die zuvor definierte Basis. [3]

```

1 bit<32> x = 8;
2 int<16> y = 101;
3 int ytwo = 5;
4 varbit<48> z;

```

Listing 3: Einige grundlegende Datentypen

In Listing 3 erkennt man einige der wichtigsten, grundlegenden Datentypen. Der Datentyp **bit** wird mit einer fixen Bit-Anzahl definiert und enthält immer einen Zahlenwert ohne Vorzeichen. Des weiteren gibt es auch einen Datentyp mit Vorzeichen: **int**. Dieser kann sowohl mit als auch ohne eine fix definierten Bitlänge angeben werden. Dabei ist jedoch darauf zu achten, dass Zahlen mit einer definierte Bitlänge auf Basis des Dualsystems

angegeben werden müssen. Zahlen des Datentyps `int` ohne definierte Bitlänge werden hingegen im Dezimalsystem angegeben. Die Längenangaben der Datentypen `bit<N>` und `int<N>` müssen jedoch bereits vor dem Kompilieren des Programms definiert werden. Dies ist bei `varbit<N>` nicht der Fall, da dieser Datentyp eine dynamische, also zur Laufzeit bestimmte, Länge aufweist und in der Bitbreite nur durch die in `N` definierte Bitlänge begrenzt wird. [3]

```

1 typedef bit<48> Ethernet_address;
2
3 header Ethernet_h{
4     Ethernet_address dstAddr;
5     Ethernet_address srcAddr;
6     bit<16> ether_type;
7 }
8
9 struct Parsed_packet{
10     Ethernet_h ethernet;
11     IPv4_h ip;
12 }
```

Listing 4: Beispiele für abgeleitete Datentypen [7]

In Listing 4 werden einige entscheidende, abgeleitete Datentypen beispielhaft aufgeführt. Darin ist zuerst das Sprachelement `typedef` zu sehen, das benutzt wird, um alternative Namen für Datentypen zu vergeben und das die Wiederverwendung des definierten Datentyps erlaubt [3]. In Zeile drei findet sich ein Header, der dazu benutzt wird die Felder eingehender und ausgehender Pakete zu definieren [7]. Laut [7] ist ein `struct` konzeptionell mit einem C-Konstrukt vergleichbar. Abschließend findet sich ein `struct` mit dem Namen `Parsed-packet`, das zum Beispiel dazu genutzt wird, alle Header, die durch den Parser erkannt werden sollen, an diesen zu übergeben [3].

Zuletzt seien auch noch Metadaten erwähnt. Unter diesem Begriff versteht [3] temporäre Daten, die während des Programmablaufs erzeugt werden. Man muss hier jedoch noch zwischen intrinsischen Metadaten und durch den/die Benutzer*in erstellte Metadaten unterscheiden. Denn laut [7] versteht man unter intrinsischen Metadaten paketspezifische Informationen, die durch das Zielgerät bereitgestellt und verarbeitet werden, während benutzerdefinierte Metadaten unabhängig von dem jeweiligen Zielgerät festgelegt werden können.

Einige prädestinierte Beispiele für intrinsische Metadaten finden sich im V1-Modell:

- `ingress_port`: Für jedes neu ankommende Paket wird durch das V1-Modell ein Datum erstellt, das die Nummer des Eingangsports enthält [64].
- `egress_port`: Ähnlich der Funktionsweise von `ingress_port`, jedoch mit dem Unterschied, dass hier der Eingangsport der Egress-Verarbeitungspipeline eingetragen wird [64].

- **egress_spec:** Bereits im Kapitel 4.6.3 unter dem Begriff „Egress-Spezifikation“ erwähnt. Enthält den Ausgangsport, aus dem das Paket gesendet werden soll [64].

4.9.3 Parser

```

1 parser Parser(packet_in b,
2                 out Parsed_packet packet) {
3     state start {
4         b.extract(packet.ethernet);
5         transition select(packet.ethernet.ether_type) {
6             0x0800: parse_ipv4;
7             default: reject;
8         }
9     }
10    state parse_ipv4 {
11        b.extract(packet.ip);
12        transition accept;
13    }
14 }
```

Listing 5: Beispiele für einen Parser [7]

Dieses Teilprogramm hat laut [7] die Aufgabe, gültige Abfolgen von Headern innerhalb eines Pakets zu erkennen und zu extrahieren.

Ein Parser ist vergleichbar mit einem endlichen Automaten. Wie dieser hat er einen Anfangszustand und mehrere Endzustände. Im Fall von P4 gibt es jedoch genau zwei Endzustände. Entweder wird ein eingehendes Paket erfolgreich geparsst, repräsentiert durch den Zustand **accept**, oder das Paket kann nicht geparsst werden, was durch den Zustand **reject** gekennzeichnet wird. Zustände zwischen dem Anfangs- und den Endzuständen können durch den/die Nutzer*in frei definiert werden. Innerhalb eines solchen Zustands können gleichzeitig ein oder mehrere Header behandelt werden. Außerdem kann ein Übergang eines Zustands in einen anderen, aufgrund der im Paket enthaltenen Header-Daten, definiert werden. [3]

In Listing 5 ist ein Parser definiert, der, wenn er ein Paket **b** erhält, darauffolgend versucht, eine Struktur des Typs **Parsed_packet** (ersichtlich in Listing 4) zu erzeugen. Diese Struktur enthält sowohl einen Ethernet- als auch einen Internet Protocol Version 4-Header (IPv4).

4.9.4 Control

```

1 control Ctr(inout Parsed_packet headers,
2               in InControl iCtr,
3               out OutControl oCtr) {
4     IPv4_address nextHop;
5     action Drop_action() {oCtr.port = DROP_PORT;}
6     action Set_nhopt(IPv4_address ipv4_dest, PortId port) {
7         nextHop = ipv4_dest;
8         oCtr.output_port = port;
9     }
```

```

10
11     table ipv4_match {
12         key = { headers.ip.dstAddr: lpm; }
13         actions = { Drop_action; Set_nhop; }
14         default_action = Drop_action;
15     }
16
17     apply {
18         ipv4_match.apply();
19         if (oCtr.output_port == DROP_PORT)
20             return;
21     }
22 }
```

Listing 6: Ausschnitt eines Control-Blocks [7]

Anschließend an den Parser findet meist eine Verarbeitung und Manipulation der Datenpakete durch einen Control-Block statt. Innerhalb dieses Blocks werden diese Transformationen durch das „Match+Action“-Prinzip vollzogen [3]. Der Ablauf eines Control-Blocks ist dabei dem eines imperativen Programms nachempfunden, da hier eine Sequenz von definierten „Match+Action“-Aktionen hintereinander abgearbeitet wird [7]. In den folgenden Absätzen werden die wichtigsten Komponenten eines Control-Blocks beschrieben.

Die erste hier vorgestellte Komponente ist **action**. Diese beschreibt die Art und Weise, wie Header-Felder und Metadaten eines Pakets transformiert werden [3]. Diese Komponente ist daher meist hauptverantwortlich für die Veränderung des Weiterleitungsverhalten der Data-Plane. Wie in Listing 6 zu sehen, ähneln Aktionen syntaktisch einer C-Funktion ohne Rückgabewert. Im Gegensatz zu diesem Beispiel kann **action** auch außerhalb eines Control-Blocks implementiert werden, was dazu führt, dass diese Implementierung dann auch innerhalb eines anderen Control-Blocks genutzt werden kann [3]. Auf die Besonderheiten bezüglich Aktionen-Parameter wird in einem späteren Absatz eingegangen.

table bildet das Kernstück des „Match+Action“-Prozesses. Dafür wird mit Tabellen gearbeitet, in denen mittels Keys passende Einträge gesucht werden. Das Ergebnis dieser Suche ist der Aufruf einer **action**. Gedanklich kann eine Tabelle als eine Abfolge von Zeilen mit jeweils einem Key-Action-Paar visualisiert werden. Wie in der Tabellenimplementierung in Listing 6 zu erkennen, ist einem Key, in diesem Beispiel die IP-Zieladresse eines Pakets, immer eine Konstante, hier **lpm**, zugeordnet. Mit deren Hilfe erfolgt die Tabellensuche mit einem spezifizierten Algorithmus. Diese Konstante wird **match_kind** genannt. Innerhalb der P4-Standardbibliothek sind drei dieser Konstanten definiert (**exact**, **ternary** und **lpm**). Durch die Verwendung einer Architekturbibliothek können jedoch auch noch weitere hinzugefügt werden. Wie in Listing 6 außerdem zu erkennen ist, gibt es die Möglichkeit, eine Standard-Aktion zu definieren, falls die Suche anhand des definierten Keys nicht erfolgreich sein sollte. Das Eintragen und Manipulieren von Einträgen in Tabellen wird durch die Control-Plane ermöglicht. [3]

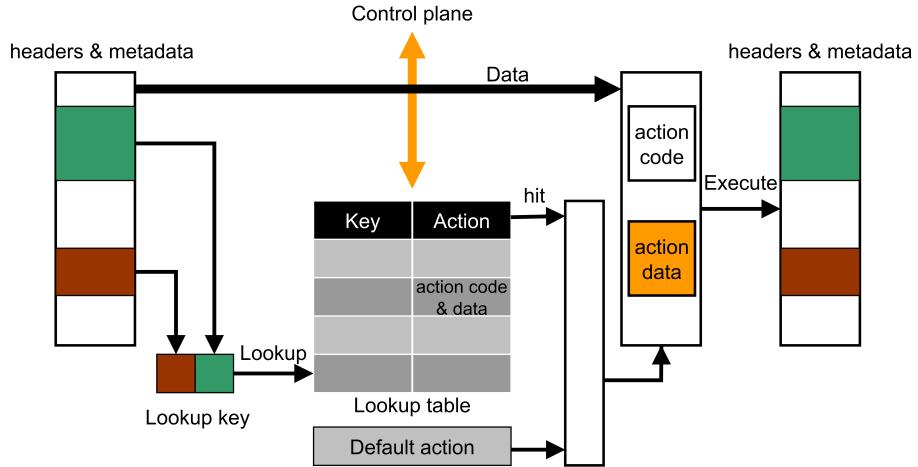


Abbildung 14: „Match+Action“-Prozess nach [3]

Der gesamte „Match+Action“-Prozess wird nochmals in Abbildung 14 zusammenfassend visualisiert. Darin ist auch gut ersichtlich, dass eine **action** mehrere Parameter aus verschiedenen Quellen aufweisen kann, zum einen gerichtete Parameter, deren Inhalt aus der Data-Plane selbst stammt. Ein Beispiel hierfür sind Informationen, die aus dem Header oder den Metadaten eines Pakets herausgelesen werden können. Gerichtete Parameter sind in P4 immer an deren Präfix (`in`, `out` oder `inout`) erkennbar. Zum anderen gibt es auch richtungslose Parameter, die laut [15] in der Praxis tendenziell öfter verwendet werden als gerichtete Parameter. [3]

Zuletzt muss noch auf den `apply`-Abschnitt des Control-Blocks eingegangen werden. Dieser enthält laut [7] ein schleifenfreies, imperatives Programm, in dem die Behandlungsabfolge der Tabellen definiert ist. Ein Beispiel für diesen Abschnitt lässt sich im untersten Teil von Listing 6 erkennen. In diesem Anwendungsfall wird jedoch nur eine Tabelle (`ipv4_match`) verwendet.

4.9.5 Deparser

```

1 control Deparser(in Parsed_packet packet, packet_out b) {
2     apply {
3         b.emit(packet.ethernet);
4         b.emit(packet.ip);
5     }
6 }
```

Listing 7: Deparser-Blocks [7]

Der letzte Block in der Verarbeitungspipeline ist ein Control-Block, der dazu genutzt wird, um Deparsing durchzuführen. Darunter versteht man, dass die zur Verfügung stehenden Header eines Datenpakets wieder zusammengefügt werden. Dies ermöglicht, dass Pakete korrekt serialisiert aus dem Gerät verschickt werden können. Dieser Deparsing-Vorgang ist

in Listing 7 zu beobachten. Dabei wird in diesem Beispiel zuerst ein Ethernet-Header und darauffolgend ein IPv4-Header an das ausgehende Paket (`packet_out`) angefügt. [3]

4.9.6 Main

```
1 Switch(Parser(), Ctr(), Deparser()) main;
```

Listing 8: Main-Deklaration [7]

Schlussendlich muss noch ein Package mit einer Variable `main` definiert werden. Dies ist in Listing 8 zu sehen. Der Name dieses Packages (`Switch`) ist durch die jeweilige Architektur vorgegeben. Durch die Verwendung dieses Objekts wird festgelegt, in welcher Reihenfolge die einzelnen Komponenten des P4-Programms zusammengefügt werden müssen, sodass diese für die Instanziierung des Netzwerkgerätes verwendet werden können. [7]

Leider lässt es der Umfang von P4 nicht zu, alle Objekte dieser Sprache vorzustellen. Darunter fallen zum Beispiel `counter`, `meter`, `registers`, einige andere externe Objekte, Expressions und Statements. Darüber hinaus wird auch auf das Thema Architektur-Definition nicht eingegangen, da das Ziel dieser Masterarbeit nicht darin besteht eine eigene, individuelle Architektur und den dazugehörigen Compiler zu schreiben.

5 Kontrollnetzwerke

Kontrollnetzwerke sind Systeme, die eine hohe Übertragungsfrequenz kleiner Nachrichtenpakete ermöglichen. Um diese Eigenschaft zu ermöglichen, benötigen Kontrollnetzwerke hohe Zuverlässigkeit und geringen Protokoll-Overhead. Diese Netze enthalten intelligente Geräte, die Kontrollfunktionen implementieren und durch ein gemeinsames Kommunikationsprotokoll kommunizieren. In einem Kontrollnetzwerk können sowohl einfache Sensoren, als auch komplexe Datenerfassungsgeräte existieren. Diesen Beschreibungen eines Kontrollnetzwerkes entspricht auch LonWorks. LonWorks und das diesem System zugeordnete LonTalk-Protokoll bilden das Fundament, um Kontext und Implementierung des LonTalk-Protokolls auf einer Data-Plane unter Zuhilfenahme von P4 zu verstehen. Daher werden in diesem Kapitel auch die grundlegenden Eckpunkte der LonWorks-Technologie vorgestellt. Bisherige P4-Implementierungen zielen meist auf die traditionelle Netzwerktechnik, im Umfeld von Ethernet, ab. Das zeigt sich dadurch, dass die meisten P4-Zielgeräte, die bereits in Kapitel 4 erwähnt wurden, auch für diesen Bereich entworfen und entwickelt werden. Dass in dieser Arbeit das LonTalk-Protokoll implementiert wird, soll aufzeigen, dass es auch noch andere Anwendungsbereiche für P4 gibt, die bisher nicht im Fokus standen. [10]

5.1 LonWorks

Das Netzwerkprotokoll LonTalk gehört dem Themenbereich LonWorks an. Daher ist es auch unerlässlich, ein prinzipielles Verständnis für diese Technologie zu vermitteln. Allgemein gesehen bildet das Feldbussystem Local Operating Network (LON) den eigentlichen Kern von LonWorks. Die Aufgabe von LON ist es, eine Vielzahl von kostengünstigen, dezentralen Knotenpunkten über ein Bussystem zu verbinden. Die Hauptbestandteile der LON-Technologie wurden in der Standardfamilien ISO/IEC 14908 festgelegt und beschrieben. [65]

5.1.1 LonWorks-Übertragungsmedien

Grundsätzlich ist es bei LonWorks möglich, eine Vielzahl an verschiedenen Übertragungsmedien zu verwenden. Darunter fallen laut [65] auch Funkstrecken, Zwei-Draht- und Glasfaserleitungen. Die folgenden Übertragungsoptionen werden hier genauer behandelt:

- Free Topology Twisted-Pair: Im Gegensatz zu konventionellen Bus-Topologien muss hier keine bis gar keine Beachtung auf die Wahl der Topologie gelegt werden. Diese Channel-Art ist durch ISO/IEC 14908-2 standardisiert. [10]

Band	Frequenzbereich	Verwendungszweck
A	3 kHz – 95 kHz	Überwachung und Kontrolle des Niederspannungsnetzes
B	95 kHz – 125 kHz	Kunden ohne weitere Einschränkungen
C	125 kHz – 140 kHz	Kunden unter Verwendung des CSMA Protokolls
D	140 kHz – 148,5 kHz	Kunden ohne weitere Einschränkungen

Tabelle 1: EN 50065 Frequenzen [66]

- IP-852: Dieses Medium ermöglicht die Übertragung von LonTalk-Paketen über bestehende IP-Netzwerke. Dieser Tunneling-Prozess ist in ISO/IEC 14908-4 standardisiert. Die Verbindung zwischen einem beliebigen Übertragungsmedium und einem IP-Netzwerk geschieht dabei über einen Router, der beide Medien unterstützt. [65]
- PLC: Der Begriff PLC beschreibt die digitale Übertragung von Nachrichten über Leitungsnetze, die eigentlich der Energieversorgung dienen [67]. Diese Channel-Art ist durch ISO/IEC 14908-3 standardisiert und auf die europäische Norm EN 50065 [68] abgestimmt [10]. Laut [69] erlaubt diese Norm die Nutzung von Frequenzen zwischen 3 kHz und 148,5 kHz. Dieser Frequenzbereich ist, wie in Tabelle 1 zu erkennen, in Bänder mit dazugehörigem Verwendungszweck aufgeteilt. Durch die Abstimmung des ISO-Standards auf die europäische Norm wird eine automatische Beschränkung der mit LonWorks verwendbaren Frequenzen erreicht. Die bewirkt, dass LonWorks dadurch auch global eingesetzt werden kann, weil andere PLC-Standards dieses Frequenzband meist ebenfalls beinhalten [70]. Der Vorteil dieses Mediums ist, dass meist bestehende Infrastruktur verwendet werden kann und dadurch der Aufwand und die Kosten für die Herstellung eines neuen Übertragungsweges entfallen.

5.1.2 Logische Struktur eines LonWorks-Netzes

Um ein besseres Verständnis für die einzelnen Elemente eines Netzes unter LonWork zu gewinnen, werden diese in den greifbareren, metaphorischen Kontext einer Stadt mit Stadtteilen und Bewohnern eingebettet. Die Struktur eines Netzes kann sich in folgende logische Elemente unterteilen:

- Knoten: Ein Knoten ist ein logisches Objekt, das sich laut [65] auf einem physischen Gerät befindet, dass die Schichten eins bis sieben des LonTalk-Protokolls implementiert. Ein Smart Meter oder eine Jalousien-Steuerungseinheit sind Beispiele für Knoten. Laut [10] besitzt ein Knoten einen sieben Bit langen Identifikator. Das physische Gerät, das den Knoten beinhaltet, besitzt außerdem noch eine 48 Bit lange, global einzigartige Identifikationsnummer, welche generell als Unique ID beziehungsweise bei Neuron-Chips als Neuron ID bezeichnet wird. Diese wird, derselben Quelle

zufolge, oftmals für die initiale Konfiguration des Gerätes verwendet.

In der angesprochenen Metapher können Knoten als Einwohner einer Stadt gesehen werden.

- Subnetz: Diese logische Struktur ermöglicht die Unterteilung aller Knoten einer Domäne in bis zu 255 Subnetze. Dabei kann ein jedes Subnetz maximal 127 Knoten beinhalten. Dadurch, dass die Struktur an eine Domäne gebunden ist, kann sich diese auch nicht in eine andere Domäne hinaus erstrecken. [65]

In der bereits skizzierten Metapher nimmt ein Subnetz den Platz eines Bezirks beziehungsweise Stadtteils ein.

- Domäne: Eine Domäne ist eine Struktur, die sich über mehrere Übertragungsmedien hinweg erstrecken kann. Sie ermöglicht, dass Knoten, die sich in derselben Domäne befinden, miteinander kommunizieren können. Um eine Kommunikation zwischen Knoten verschiedener Domänen zu ermöglichen, bedarf es eines gemeinsamen Gateways, das als Applikation implementiert ist. [65]

Bildlich gesehen ist eine Domäne einer Stadt gleichzusetzen.

- Gruppe: Gruppen sind eine Sammlung von Knoten einer Domäne. Eine Domäne kann dazu in bis zu 256 Gruppen unterteilt werden. Gruppen können innerhalb eines Netzes dazu genutzt werden, die für die Broadcast-Kommunikation notwendige Paketanzahl einzuschränken. [65]

In der erwähnten Metapher kann eine Gruppe als Freundeskreis, der innerhalb einer Stadt verteilt wohnt, gesehen werden.

5.1.3 LonWorks-Netzwerkbausteine

Der Aufbau eines Netzes besteht neben Knoten-Geräten und Übertragungsmedien aus den folgenden Bausteinen:

- Repeater: Ein Repeater wird dazu genutzt, mehrere gleich- beziehungsweise verschie- denartige Übertragungsmedien miteinander zu verbinden. Dieser kann auch dazu genutzt werden, die Signalform zu regenerieren. [65]
- Bridge: Eine Bridge stellt eine erweiterte Form eines Repeaters dar, der auch in der Lage ist, Domäneninformation zu berücksichtigen. [65]
- Learning Router: Diese Gerätegruppe lernt den Netzaufbau auf Domain- und Subnetzebene durch den vorkommenden Datenverkehr und vermittelt Pakete basierend auf der gelernten Adressinformation weiter. Gruppeninformationen werden hier allerdings nicht berücksichtigt sondern ignoriert. [65]

- Configured Router: Im Gegensatz zu einem Learning Router vermittelt ein Configured Router nur Pakete, die eine entsprechende Adressübereinstimmung in der Vermittlungstabelle haben. Diese Tabelle kann auch mit Gruppeninformationen befüllt werden und dadurch auch Vermittlungen auf Gruppenebene durchführen. Die Konfiguration erfolgt dabei über das für den Router jeweilige Werkzeug, das durch den Hersteller des Geräts bereitgestellt wird. [65]

5.1.4 LonWorks-Beispielnetzwerk

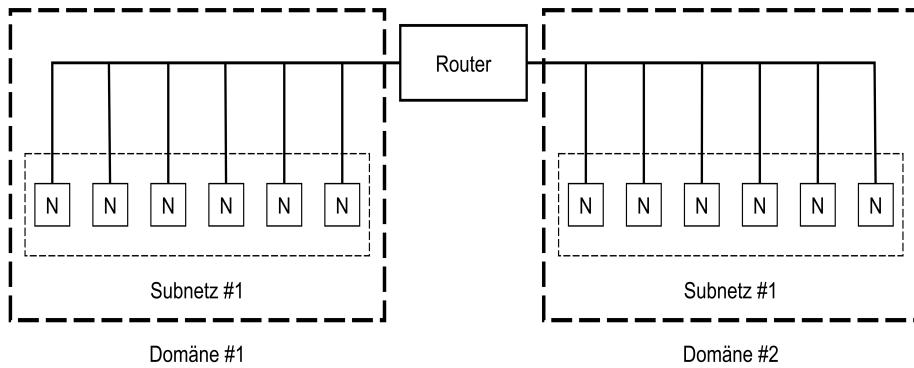


Abbildung 15: Aufbau eines LonWorks-Netzes

In Abbildung 15 ist beispielhaft ein Netzwerk abgebildet, das aus Komponenten, die in den Kapiteln 5.1.2 und 5.1.3 vorgestellt wurden, besteht. Darin sind zwei Domänen zu erkennen, die jeweils ein Subnetz aufweisen. Jedes Subnetz besteht dabei aus sechs Knoten, die sich ein Übertragungsmedium teilen. Die beiden Domänen werden in diesem Beispiel von einem Router getrennt. Dieser ist jedoch nicht nötig, wenn in beiden Domänen dasselbe Übertragungsmedium verwendet wird. Der Grund dafür ist, dass auf einem Medium laut [65] mehrere Domänen vorhanden sein können.

5.1.5 Aufbau eines LonWorks-Knotens

Bei LonWorks gibt es mehrere Möglichkeiten, wie ein Knoten aufgebaut sein kann. Wie bereits in Kapitel 5.1.2 beschrieben, muss ein Knoten lediglich die Schichten eins bis sechs des LonTalk-Protokolls implementieren.

In [71] werden mehrere dieser Möglichkeiten skizziert. Einerseits kann der Neuron-Chip⁶ zum Einsatz kommen, denn dieser erlaubt es relativ einfach, die Protokoll-Schichten zwei bis sieben zu implementieren. Für die Umsetzung der Funktionalität der Schicht eins muss

⁶Ein Neuron-Chip ist ein SoC, das über mehrere Prozessoren, Speicher und Kommunikations- und Input/Output-Schnittstellen verfügt und für LonWorks-Knoten vorgesehen ist [10].

allerdings ein passender Transceiver verwendet werden. Bei Verwendung eines Neuron-Chips ist es auch möglich, auf dessen Funktionalität auf Schicht sieben zu verzichten, denn diese kann auch durch die Verwendung eines Mikroprozessors implementiert werden. Andererseits kann auch vollständig auf den Neuron-Chip verzichtet werden. Dafür bedarf es aber einer alternativen Implementierung der Protokollsichten zwei bis sieben in Software oder Hardware. Dies ist zum Beispiel bei den LonWorks-Knoten des Herstellers Loytec⁷ der Fall. Hierbei werden die Protokoll-Schichten zwei bis drei in Hardware und die Schichten vier bis sechs durch eine C-Implementierung festgelegt. [71]

5.2 Lontalk-Protokoll

LonTalk ist das Kommunikationsprotokoll innerhalb der LonWorks-Technologie [73]. Laut [10] ist der Name LonTalk Eigenname der Firma Echelon⁸. Unter diesem Namen wird deren Implementierung des ISO/IEC 14908-1-Standards verstanden. Allgemein wird LonTalk auch noch als Control Network Protocol (CNP) betitelt [10]. In diesem Kapitel sind die einzelnen Schichten des Protokolls und deren wichtigste Aufgaben erklärt.

5.2.1 Schichten des LonTalk-Protokolls

Der Aufbau von LonTalk ist stark an die Struktur des OSI-Modells angelehnt, da sich die Funktionen des Protokolls ebenfalls in diese Schichten unterteilen lassen [65]. In Tabelle 2 sind dieser Zusammenhang und die wichtigsten LonTalk-Funktionen dargestellt. In den folgenden Absätzen werden die einzelnen Schichten und deren Aufgaben überblicksmäßig beleuchtet.

Schicht eins definiert, wie die Übertragung der Daten-Bits über das Übertragungsmedium vonstatten geht und das korrekte Erkennen versendeter Bits auf Seiten des Empfängers. Wie bereits in Kapitel 5.1.1 erwähnt und vorgestellt, können unter dieser Technologie viele Übertragungsmedien verwendet werden. [10]

Auf Schicht zwei sind Methoden der Medienzugriffssteuerung implementiert, um eine gezielte Nutzung des Übertragungsweges zu gewährleisten. Hier erfolgt ebenfalls eine Fehlerüberwachung auf Basis eines Cyclic Redundancy Checks (CRCs). Außerdem wird auf dieser Ebene ein Mechanismus zur Priorisierung von Datenframes implementiert. [10]

Schicht drei widmet sich dem Thema Routing und definiert dadurch, wie Nachrichten von einer Quelladresse zu einer oder mehreren Zieladressen übertragen werden. Dies betreffend

⁷LOYTEC electronics GmbH, Entwickler und Hersteller von LonWorks-Lösungen [72].

⁸Echelon Corporation, Entwickler der LON-Technologie und des Neuron-Chips, nunmehr Teil von Adesto Technologies [74].

OSI-Schicht	Bedeutung	LonTalk Service
7 Application	Kompatibilität auf Applikationsebene	Objektdefinition, Standard-Netzwerkvariablen, Netzwerkmanagement
6 Presentation	Interpretation	Transport von beliebigen Telegramm-Rahmen
5 Session	Aktion	Request-Response-Mechanismus
4 Transport	Zuverlässigkeit	Übertragung mit/ohne Quittung, Einzel- und Gruppenadressierung, Authentifizierung, Duplikat-Erkennung
3 Network	Zieladressierung	Broadcast-Meldungen, Routing
2 Link	Medienzugriff und Rahmenprüfung	CRC-16, CSMA, Kollisionsvermeidung durch Zuteilung von Zeitschlitten, Encodierung der Daten, Rahmenprüfung
1 Physical	Elektrische Verbindung	Unterstützung für diverse Übertragungsmedien

Tabelle 2: LonTalk-Ebenen nach [65]

wird hier auch auf den besonderen Fall eingegangen, dass Geräte sich auf unterschiedlichen Übertragungsmedien befinden können. Adressen, die für das Routing benötigt sind, werden ebenfalls hier definiert. [10]

Auf Schicht vier beschäftigt man sich mit der Erkennung von Duplikaten und zuverlässiger Zustellung von Pakten. Dafür stehen dem/der Nutzer*in die folgenden drei Übertagungsmethoden zur Verfügung. [10]

- Acknowledged: Versendete Nachrichten erwarten eine Bestätigung für den Erhalt der Nachricht durch den Empfänger. Dies ermöglicht eine zuverlässige Kommunikation. [10]
- Unacknowledged / repeated: Hierbei wird keine Empfangsbestätigungen erwartet. Jedoch werden, um eine geringfügige Zuverlässigkeit zu bieten und die Empfangswahrscheinlichkeit einer Nachricht zu erhöhen, Frames mehrfach versendet. [10]
- Unacknowledged: Hier wird ebenfalls keine Antwort durch den Empfänger erwartet, jedoch wird auch keine mehrfache Sendung eines Datenpakets durchgeführt. Dieser Modus ist nicht zuverlässig und daher nur für Anwendungen gedacht, die einer solchen Zuverlässigkeit nicht bedürfen. Eine Beispielanwendung für diesen Modus ist die Übertragung von Temperatursensordaten eines Smart Meters. [10]

Schicht fünf implementiert einen Request-Reponse-Mechanismus. Dieser ermöglicht es Geräten, unter anderem bestimmte Daten von anderen Knoten abzufragen. Außerdem wird

auf dieser Ebene ein Authentifizierungsverfahren definiert, das einem Empfänger einer Nachricht ermöglicht, zu überprüfen, ob der Sender autorisiert ist, eine Nachricht zu senden. [10]

Auf Schicht sechs werden die auszutauschenden Datenpakete in sechs verschiedene Nachrichtentypen unterteilt: „User Application-“, „Standard Application-“, „Foreign Frame-“, „Network Diagnostic-“, „Network Management-“ und „Network Variable Message“. Zwei dieser Nachrichtentypen werden nunmehr beispielhaft herausgegriffen und kurz vorgestellt. [10]

- Nachrichten des Typs „Network Variable Message“ enthalten spezifizierte Standard Network Variable Types (SNVTs). Die Struktur und der Inhalt eines jeden dieser Nachrichtentypen ist genau definiert, um die Interoperabilität zwischen Knoten verschiedener Hersteller zu gewährleisten. SNVTs werden dabei durch die Organisation LonMark⁹ definiert. Dies ermöglicht es zum Beispiel, dass ein bestimmtes SNVT dazu genutzt werden kann, auf allen Geräten eines Netzes die Temperatur auf gleiche Weise, mit gleicher Skala (Fahrenheit, Celsius) und vorhersehbarem Format abzufragen. [10]
- Bei Nachrichten des „Foreign Frame Message“-Typs werden Bytes durch eine Anwendung versendet. Die Interpretation des Nachrichteninhalts obliegt dabei der jeweiligen Applikation. Dieser Typ kann zum Beispiel für die Implementierung eines eigenen Protokolls genutzt werden. [10]

Die Anwendungsschicht (Ebene sieben des OSI-Modells) sieht die Standardisierung von Diensten vor, die zum Beispiel für Netzwerkkonfiguration, Diagnosedienste und Scheduling von Nöten sind. Dabei stützt sie sich auf die Definition der darunterliegenden Schichten. Auch auf dieser Ebene kann erst durch die Spezifizierung der Dienste die Interoperabilität zwischen Geräten und Software-Tools verschiedener Hersteller gewährleistet werden. [10]

5.2.2 Protocol Data Units des LonTalk-Protokolls

In Kapitel 5.2.1 wurde bereits ein grober Blick auf die Aufgaben und Inhalte einer jeden Schicht des LonTalk-Protokolls gewährt. In diesem Kapitel wird nunmehr auf einige Protocol Data Unit-Formate (PDU) des LonTalk-Protokolls genauer eingegangen. Dies ist vor allem notwendig, um den Ausführungen des Kapitels 8.4 folgen zu können. In diesem Kapitel wird den PDU-Formaten auf den Schichten zwei und drei Beachtung geschenkt, weil diese für die praktische Implementierung, die in dieser Arbeit veranschaulicht wird, den

⁹LonMark International Inc., Organisation für die Verbreitung von Interoperabilität im LonWorks-Bereich [75].

größten Stellenwert haben. Für alle nachfolgend besprochenen PDU-Formate des LonTalk-Protokolls gilt laut Annex D in [76], dass nach der Methode „most significant first“ verfahren wird. Dies bedeutet, dass das Most Significant Bit (MSB) eines jeden Bytes zuerst übertragen wird [76]. Das MSB befindet sich in den folgenden Abbildungen dieses Kapitels immer links oben.

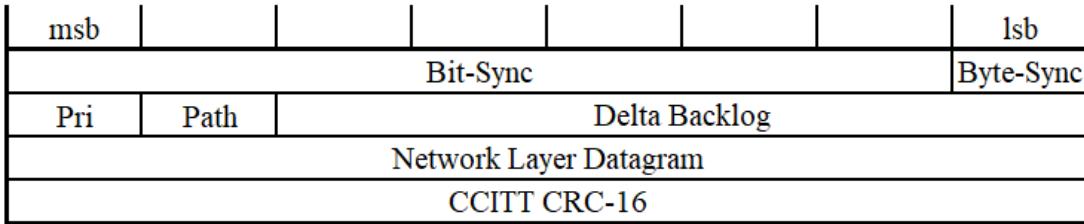


Abbildung 16: LonTalk-Nachricht nach [10]

Das PDU-Format des LonTalk-Protokolls ist in Abbildung 16 dargestellt. Die ersten beiden Felder dieser PDU lassen bereits erkennen, dass eine LonTalk-Nachricht mit einer Präambel beginnt. Bei der Betrachtung der Abbildung muss jedoch beachtet werden, dass die Gesamtlänge der Präambel beeinflusst und daher variiert werden kann. Diese Variabilität ergibt sich aus der Konfigurierbarkeit der Länge des Bit-Sync-Feldes, das neben dem Byte-Sync-Feld Bestandteil der Präambel ist [10]. Da laut [10] das Byte-Sync-Feld nur aus einem differenziell manchester-codierten Null-Bit besteht und das Bit-Sync-Feld mindestens sechs Eins-Bits enthalten muss, kann davon ausgegangen werden, dass die Länge der Präambel meist ein ganzzahliges Vielfaches eines Bytes ist. [76] beschreibt, dass die Länge dieser Präambel für ein jedes Übertragungsmedium spezifisch ist. Aus [10] geht außerdem hervor, dass diese Präambel dazu genutzt wird, um in asynchronen Netzen eine Synchronisierung der Bit- beziehungsweise Byte-Samplingtime zu ermöglichen. Das Byte-Sync-Feld markiert für einen LonTalk-Knoten das Ende der Präambel und damit den Beginn des Pri-Feldes. Dieses wird dazu genutzt, um den Prioritätsstatus einer Physical Protocol Data Unit (PPDU) festzulegen. Wenn der zu versendeten Nachricht eine Priorität gegenüber anderen Nachrichten ohne Priorität gewährt werden soll, wird der Wert dieses Bits auf eins gesetzt [10]. In [10] wird außerdem erwähnt, dass dieser Prioritätsstatus Einfluss, auf den bei LonTalk verwendeten Carrier Sense Multiple Access- Algorithmus (CSMA) hat, weil in einem solchen Fall sogenannte Beta 2-Priority-Slots anstatt von herkömmlichen Beta 2-Slots herangezogen werden. [10] beschreibt, dass das Path-Feld herangezogen werden kann, sobald ein Knoten über mehrere Übertragungswege verfügt. Über dieses Feld kann in einem solchen Fall das gewünschte Übertragungsmedium festgelegt werden. Der Grundgedanke hinter diesem Feld ist in [10] geschildert: Das Path-Feld dient einzig dazu, dass sobald eine Nachrichtenübertragung am primären Übertragungsmedium fehlschlägt, auf das andere Medium umgeschaltet werden kann. Wie in Abbildung 16 zu erkennen ist, folgt auf das Path-Feld das Delta Backlog-Feld, welches wiederum aus sechs Bit besteht. Dieses steht

wiederum im Zusammenhang mit dem Channel Backlog. Der Channel Backlog eines Geräts gibt eine lokale Schätzung an, wie viele Geräte im nächsten CSMA-Paketzyklus ein Paket senden werden [10]. Auf dieser Schätzung beruht laut [10] darauffolgend die Berechnung der Größe und Anzahl der Zeitslots (Beta 2-Slots) eines CSMA-Paketzyklus. Sobald ein Knoten nun eine LonTalk-PPDU erhält, wird der Wert des Delta Backlog-Feldes zum aktuellen Channel Backlog hinzu addiert und somit für den nächsten Paketzyklus aktualisiert [10]. Die Ermittlung des Wertes des Delta Backlog-Feldes beruht dabei auf der vom Sender erwarteten Anzahl von Acknowledgements, die durch das Versenden einer Nachricht erzeugt werden [10]. Wie in Abbildung 16 zu erkennen, folgt auf das Delta Backlog-Feld die Network Protocol Data Unit NPDU, welche weitere Headerdaten und die Nutzdaten einer Nachricht beinhaltet. Diese kann laut einer Abbildung in [10] eine Länge zwischen sechs und 246 Byte annehmen. Abschließend folgt noch eine zyklische Redundanzprüfung mittels eines CCITT CRC-16-Polynoms. Dieser Algorithmus stützt sich dabei auf den gesamten Header der OSI-Schicht zwei und die zu transportierende NPDU [10].

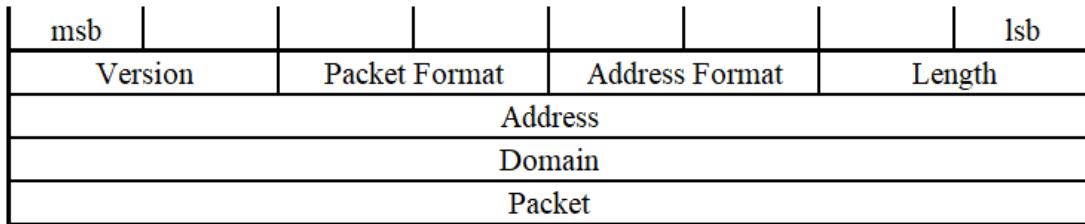


Abbildung 17: LonTalk-NPDU nach [10]

Abbildung 17 zeigt den Aufbau der besagten NPDU. Der Header der NPDU, die auch Datagramm genannt wird, beginnt mit dem Version-Feld. Dieses ist zwei Bit lang und nimmt immer den Wert Null an. Darauffolgend beginnt bereits das Packet Format-Feld, das ebenfalls aus zwei Bit besteht und das Format des in diesem Header eingeschlossenen Pakets anzeigt. Dieses Feld dient daher dazu, Netzwerknoten anzuzeigen, welche Paketart auf Schicht vier transportiert wird. Die verfügbaren Paketarten und die korrespondierenden Werte des Packet Format-Feldes sind in Tabelle 3 ersichtlich. Details zu diesen Paketformaten sind in [10] zu finden. [10]

Wert	Paketformat
0	Transport Packet
1	Session Packet
2	Authenticated Packet
3	Presentation Packet

Tabelle 3: Übersicht über das Packet Format-Feld nach [10]

Das darauffolgende Feld des Headers enthält Informationen zu dem Format der Adressin-

formation der Nachricht. Hierbei werden in Tabelle 4 die möglichen Werte dieses Feldes mit einem Adressformat verknüpft. Die Zusammensetzung der einzelnen Adressformate wird an späterer Stelle in diesem Kapitel noch erläutert. [10]

Wert	Adressformat
0	Subnet Broadcast
1	Group
2	Subnet/Node oder Group Acknowledgement
3	Unique ID

Tabelle 4: Übersicht über das Address Format-Feld nach [10]

Das letzte Feld mit einer fixer Länge von zwei Bit innerhalb dieses Headers ist das Length-Feld. Dieses Feld gibt die Anzahl der Bytes an, aus denen das Domain-Feld besteht. Die Aufschlüsselung der Werte mit der korrespondierenden Länge des Domain-Feldes findet sich in Tabelle 5. [10]

Wert	Länge des Domain-Feldes
0	0 Byte
1	1 Byte
2	3 Byte
3	6 Byte

Tabelle 5: Übersicht über das Length-Feld nach [10]

Auf das Length-Feld folgt nun das längenvariable Address-Feld. Die variable Länge dieses Feldes ergibt sich aus dem unterschiedlichen Inhalt des Feldes. Der in diesem Feld enthaltene Adressstyp wird, wie schon erläutert, durch den Wert des Address Format-Feldes (siehe Tabelle 4) angegeben. In den Abbildungen 18, 19, 20, 21 und 22 ist der Aufbau der einzelnen Adressierungsarten dargestellt. Das Address-Feld einer Nachricht dient dazu, eine Adressierbarkeit eines oder mehrerer Zielgeräte zu ermöglichen. [10]

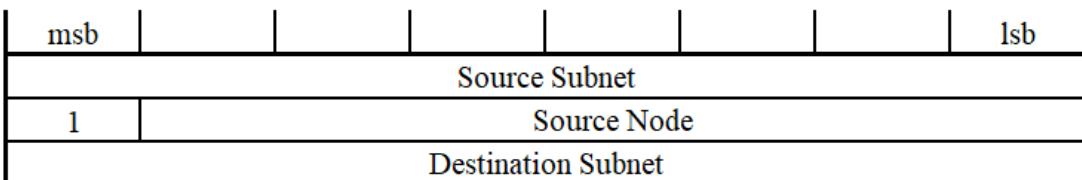


Abbildung 18: Adressformat „Subnet Broadcast“ nach [10]

Das Adressformat des Typs „Subnet Broadcast“, das in Abbildung 18 dargestellt ist, beginnt mit einer acht Bit langen Subnetznummer. Darauffolgend wird ein Füllbit mit dem Wert eins angefügt. Die Verwendung eines Füllbits röhrt in diesem Fall daher, dass nach

diesem Bit auftretende Feld über eine Länge von nur sieben Bit verfügt. Allgemein ist es bei allen Adressarten des Address-Feldes notwendig, dass die Gesamtlänge des Address-Feldes immer ein ganzzahliges Vielfaches eines Bytes ist. Abschließend folgt bei diesem Adressart ein acht Bit langes Destination Subnet-Feld. [10]

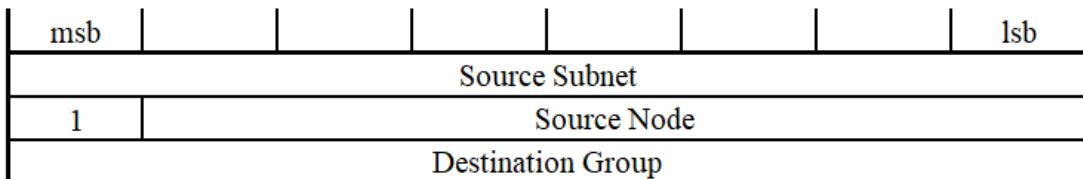


Abbildung 19: Adressformat „Group“ nach [10]

In Abbildung 19 ist das Adressformat des Typs „Group“ dargestellt. Dessen erste 16 Bit sind nach demselben Schema wie der Adresstyp „Subnet Broadcast“ aufgebaut. Der Unterschied liegt nur im abschließenden Feld des Adressformats. Dieses enthält ein acht Bit langes Feld mit einer Destination Group-Nummer. [10]

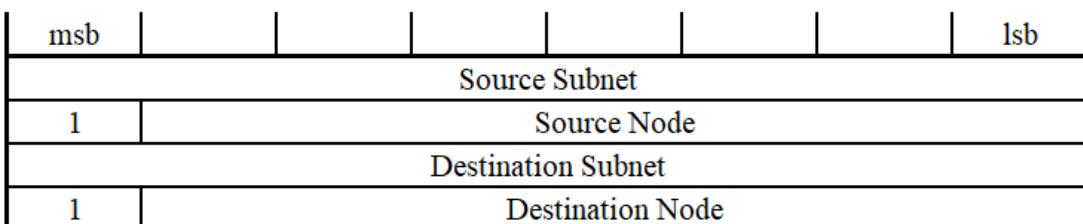


Abbildung 20: Adressformat „Subnet/Node“ nach [10]

Das Adressformat des Typs „Subnet/Node“ ist in Abbildung 20 einsehbar. Dieses ist nahezu identisch mit dem Adressformat des Typs „Subnet Broadcast“, jedoch werden an dessen Adressformat noch zwei weitere Felder hinzugefügt. Hierbei bewirkt wiederum die abschließende, sieben Bit lange Destination Node-Nummer, dass diesem Feld ein Füllbit vorangestellt wird. [10]

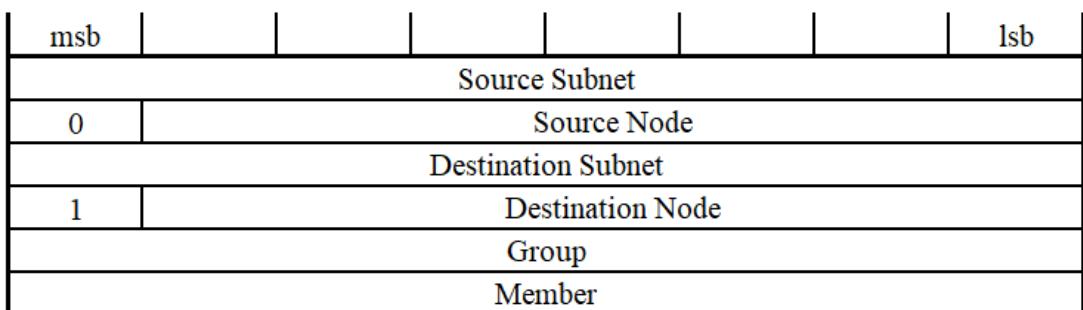


Abbildung 21: Adressformat „Group Acknowledgement“ nach [76]

In Abbildung 21 ist das Adressformat des Typs „Group Acknowledgement“ dargestellt. Aus dessen Zusammensetzung lässt sich erkennen, dass sich dieser Typ von den anderen

bisher vorgestellten Adressformaten, hauptsächlich durch das neunte Bit und die letzten beiden Felder unterscheidet. Das neunte Bit, das Füllbit, nimmt bei diesem Adressotyp den Wert null an. Die beiden letzten, bisher noch nicht näher beschriebenen Felder sind das Group- und das Member-Feld. Diese sind jeweils ein Byte lang. Dies trägt dazu bei, dass dieses Adressformat mit einer Gesamtlänge von sechs Byte das zweitlängste Adressformat des LonTalk-Protokolls ist. [10]

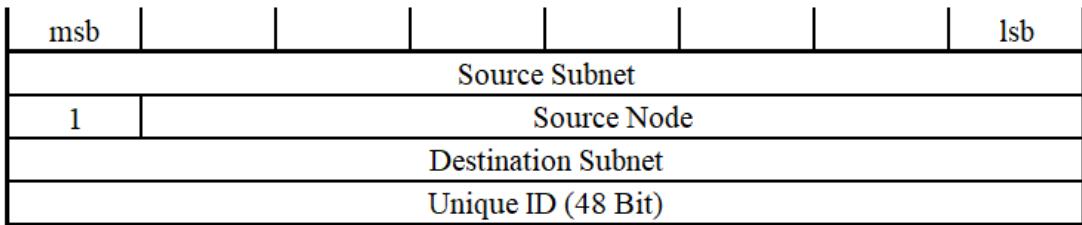


Abbildung 22: Adressformat „Unique ID“ nach [10]

Das letzte noch vorzustellende Adressformat ist „Unique ID“. Dieses ist in Abbildung 22 dargestellt. Mit einer Gesamtlänge von neun Byte ist dieses Adressformat das mit der höchsten Gesamtlänge. Diese ergibt sich vor allem daraus, dass das Unique ID-Feld bereits eine Länge von 48 Bit erfordert. Der restliche Aufbau dieses Formats ist wiederum mit den Komponenten des Adressstyps „Subnet Broadcast“ (ersichtlich in Abbildung 18) identisch. [10]

Nunmehr werden die letzten beiden Felder der NPDU, welche in Abbildung 17 dargestellt sind, erläutert. An den Adressteil einer NPDU schließt sich das Domain-Feld an, dessen Länge durch den Wert des Length-Feldes vorgegeben wird. Die möglichen Längen in Byte sind in Tabelle 5 ersichtlich. Das die NPDU abschließende Feld ist das Packet-Feld. Darin befinden sich die zu übermittelnden Nutzdaten. Das Format der enthaltenen Nutzdaten ist für einen Knoten anhand des Packet Format-Feldes ersichtlich. [10]

Weitere Details hinsichtlich des Aufbaus einer LonTalk-Nachricht finden sich in [10], [65], [66], [71] und [76].

6 Grundprinzipien der Implementierung

Bezüglich der praktischen Implementierung dieser Arbeit, wird in diesem Kapitel darauf eingegangen, welche grundsätzlichen Inhalte mittels P4 beschrieben werden sollten. Dabei steht vor allem folgende Frage im Vordergrund: Warum sollte das LonTalk-Kommunikationsprotokoll implementiert werden? Außerdem wird darauf eingegangen, welche Grundgedanken für die Realisierung eines LonTalk-Routers sprechen und warum sich eine Implementierung eines solchen Gerätetyps in dieser Arbeit gegenüber den anderen LonTalk-Geräten durchsetzen konnte.

6.1 Gründe für eine P4-Implementierung von LonTalk

Wie auch bereits einleitend geäußert, soll die Frage beantwortet werden, warum das Kommunikationsprotokoll LonTalk für die praktische Implementierung dieser Arbeit herangezogen wird. Gibt es doch eine Fülle von Protokollen, die noch nicht mithilfe von P4 implementiert und frei veröffentlicht wurden.

Um diese Entscheidung vollständig nachvollziehen zu können, muss erwähnt sein, dass zum Zeitpunkt der Erstellung dieser Masterarbeit, trotz umfangreicher Recherchetätigkeit, alle Indizien darauf hindeuteten, dass öffentlich zugängliche P4-Implementierungen meist auf Basis von Ethernet vorgenommen werden. Zu den Indizien zählt, dass lediglich eine niedrige Anzahl von Quellen, die meist nur für Lern- und Bildungszwecke gedacht sind, P4-Implementierungen zeigen. Beispiele hierfür sind [77], [78], [79], [80] und [81]. Für die P4-Implementierungen, die in diesen Quellen gezeigt werden, gilt, dass diese Protokolle behandeln, welche meist sehr gut öffentlich dokumentiert sind und daher auch tendenziell einfacher zu implementieren sind. Als Beispiel hierfür dienen Ethernet und IP. Dem Gegenüber steht die Dokumentation von Protokollen, wie zum Beispiel LonTalk, deren volle Dokumentation schlicht in einem ungenügenden Umfang verfügbar ist. Als Extremfall sind proprietäre Protokolle anzusehen, weil deren Dokumentation meist nicht frei einsehbar ist und daher das Protokollverhalten nur schwer reproduziert werden kann. Hinzukommend lässt sich vermuten, dass Tools, die bei der P4-Programmierung zum Einsatz kommen, beispielhaft ist hier BMv2 zu erwähnen, auf weit verbreitete, wohlbekannte Protokolle wie Ethernet ausgelegt sind. Diese Vermutung lässt sich äußern, weil die bereits erwähnte Sammlung von öffentlichen P4-Implementierungen auf diesen Tools aufbaut. Dieser und die anderen genannten Umstände bewirken, dass die Behandlung von noch nicht in P4 implementierten Protokollen neue Erkenntnisse und zusätzliches Wissen hervorbringen kann. Aus diesem Grund wurde für diese Masterarbeit beschlossen, das LonTalk-Kommunikationsprotokoll zu behandeln. Aus einem solchem Experiment ergibt

sich auch die Möglichkeit, Limitierungen der unter P4 verwendeten Tools und der Sprache selbst aufzudecken.

6.2 Auswahl des zu entwickelnden LonWorks-Gerätes

In der Einleitung des Kapitels 6 wurde bereits angeschnitten, dass das Verhalten eines LonTalk-Routers verwirklicht werden soll. In diesem Kapitel wird beschrieben, warum sich die Möglichkeit der Implementierung eines Configured Routers gegen die Implementierung eines anderen LonWorks-Netzwerkbausteins durchsetzen konnte. Daher wird in diesem Kapitel für jeden Netzwerkbaustein analysiert, warum dieser für eine Implementierung interessant sein könnte.

- Endknoten: Ein LonTalk-Endknoten arbeitet meist auf allen Schichten des LonTalk-Protokolls und hätte dadurch den Anspruch auf eine komplette Implementierung des LonTalk-Protokolls. Jedoch ist es das Ziel dieser Arbeit, Data-Plane-Programmierung anhand von P4 zu zeigen. Die Programmierung der Data-Plane durch P4 ist jedoch nur für Netzwerkgeräte vorgesehen [3]. Netzwerkgeräte sind laut [2] Geräte, deren Operationen Paketmanipulation und Paketweiterleitung umsetzen. Dies ist bei einem LonTalk-Endknoten jedoch nur bedingt der Fall, da sich die Netzwerkfunktionalität eines solches Endknotens auf das Versenden und Empfangen von Nachrichten beschränkt. Es ist daher davon abzusehen, einen Endknoten mittels P4 zu implementieren, da andere LonWorks-Netzwerkbausteine mehr Netzwerkfunktionalität bieten.
- Repeater: Auf diesen Anwendungsfall trifft die lateinische Redensart „nomen est omen“ voll und ganz zu. Denn wie die Gerätbezeichnung vermuten lässt, werden hier keine Weiterleitungsentscheidungen getroffen, da hier nur eine „Übersetzung“ zwischen Übertragungsmedien oder eine Regenerierung der Signalform vollzogen wird (siehe Kapitel 5.1.3). Dies bedeutet, dass bei diesem Gerätetyp Nachrichten auf Schicht eins des OSI-Modells bearbeitet werden. Daher ist auch dieser Anwendungsfall für eine P4-Implementierung nicht sonderlich interessant.
- Bridge: In Kapitel 5.1.3 wurde bereits erläutert, dass eine Bridge nur eine Erweiterung eines Repeaters ist. Der Unterschied zwischen den beiden Gerätarten ist, dass eine Bridge auch auf Schicht drei des OSI-Modells, der Vermittlungsschicht, arbeitet. Diese Erweiterung ist erforderlich, da bei LonTalk-Nachrichten auf dieser Schicht meist die Domäneninformation enthalten ist. Die Weiterleitungsentscheidungen einer Bridge stützen sich auf ausschließlich ebendiese Information. Da jedoch ein Router

neben anderen Weiterleitungsentscheidungen auch die Domäneninformation berücksichtigt, ist der Wissensgewinn, der durch die Implementierung eines Routers erzielt werden kann, sicherlich größer, als wenn eine „funktionsärmere“ Bridge implementiert werden würde.

Abschließend ist es noch notwendig, eine Entscheidung zwischen den Unterarten der existierenden LonTalk-Routern zu treffen. In diesem Abschnitt wird argumentiert, warum die Implementierung eines Configured Router für diese Arbeit mit mehr Relevanz verbunden ist, als die Implementierung eines Learning Routers. Grundsätzlich ist festzuhalten, dass die Grundkomponenten der Programmiersprache und die zu Verfügung stehenden Entwicklungstools für die Implementierung eines Learning Routers vor allem eine Herangehensweise zulassen. Diese würde die Implementierung einer eigenen Control-Plane bedingen und wäre daher erheblich aufwändiger, als die Implementierung eines Configured Router. Sollte eine solche Control-Plane umgesetzt werden, könnte beispielsweise ein Controller Verwendung finden, der unter Zuhilfenahme von P4Runtime mit dem Learning Router kommuniziert. Die Verwendung eines Controllers ist bei diesem Lösungsansatz darauf zurückzuführen, dass dieser dazu genutzt werden kann, die Weiterleitungstabellen mit Einträgen zu befüllen. In [82] ist eine beispielhafte Implementierung enthalten, wie diese Herangehensweise mit einem Controller funktionieren könnte. Konzeptionell würde der Learning Router für ein eingehendes LonTalk-Paket eine Entsprechung in den Weiterleitungstabellen suchen. Wenn eine solche Entsprechung nicht gefunden wird, würde der Router das betreffende Paket weiter an den Controller senden, der den Inhalt dieses Pakets analysieren würde. Das Ergebnis dieser Analyse wäre, dass die in der Weiterleitungstabellen des Routers nötigen Ergänzungen durch den Controller an den Router übermittelt würden. Bei einer solchen Herangehensweise würde sich jedoch der Fokus deutlich von der Programmierung der Data-Plane auf die Implementierung eines Controllers verschieben. Da in dieser Arbeit der Programmierung der Data-Plane die Hauptaufmerksamkeit geschenkt werden soll, wird darauf verzichtet, einen Learning Router zu implementieren.

Das bisher in diesem Kapitel durchgeführte Ausschlussverfahren lässt nunmehr nur noch die Implementierung eines Configured Routers als Option zu. Die folgenden Punkte enthalten die grundlegenden Vorteile des Configured Routers gegenüber dem Learning Router. Diese Vorteile sollen verdeutlichen, warum die Wahl des zu implementierenden Gerätetyps auf den Configured Router fällt:

- In der initialen Lernphase eines Learning Router kann es, je nach Netzwerktopologie, zu einer großen Flut an Nachrichten kommen, die im Netzwerk weiterverbreitet werden. Diese Überlastsituation kann dazu führen, dass es in dieser Phase zu einer Verringerung der Netto-Durchsatzleistung des Netzwerkes kommt und dadurch die Menge der zu übertragenden Nutzdaten stark verringert wird. Diese Situation

kann bei einem Configured Router aufgrund der nicht existierenden Lernphase nicht auftreten. [10]

- Bei PowerLine-basierenden Netzwerken ist es häufig der Fall, dass innerhalb einer Netzwerktopologie Schleifen vorhanden sind. Deren Existenz kann bei Learning Router dazu führen, dass diese ein inkorrektes Abbild der vorherrschenden Topologie erlernen. Denn im Gegensatz zu herkömmlichen Switch-Infrastrukturen existiert innerhalb von LonTalk keine mit dem Spanning Tree Protocol (STP) vergleichbare Funktion. Bei der Verwendung von Configured Routern kann auf in der Topologie existierende Schleifen manuell eingegangen werden, um diese zu entschärfen. [10]
- Der letzte und wohl wichtigste Vorteil, den ein Configured Router bietet, ist, dass dieser auch in der Lage ist, Gruppeninformationen zu berücksichtigen. Im Gegensatz dazu stehen Learning Router, die nicht dazu im Stande sind Weiterleitungsentscheidungen, aufgrund der in Paketen enthaltenen Gruppeninformation, zu treffen. Dieser Vorteil des Configured Routers stärkt dessen Berechtigung, durch P4 implementiert zu werden zusätzlich. [10]

Hinsichtlich des Configured Router gibt es jedoch weitere grundsätzliche Entscheidungen, die getroffen werden müssen. Die erste Entscheidung dreht sich um die Frage: Auf welchen Schichten des OSI-Modells arbeitet ein Configured Router? Auf diese Frage findet sich in der Literatur eine eindeutige Antwort, denn laut [10] wird bei Configured Routern die Konfiguration des Gerätes von einer zentralen Stelle im Netzwerk aus vorgenommen. Dies hat zur Folge, dass ein Configured Router bis auf Schicht sieben des OSI-Modells tätig ist, weil diese Konfiguration, wie bereits in Kapitel 5.2.1 beschreiben, auf dieser OSI-Schicht mithilfe von speziellen LonTalk-Nachrichten erfolgt. Der Erhalt solcher Nachrichten kann implementiert werden, jedoch gibt es bei der ausgewählten P4-Implementierungsplattform (siehe Kapitel 7.2), laut weitreichender Recherhetätigkeit nicht die Möglichkeit, die Weiterleitungstabellen aus dem P4-Programm heraus zu befüllen. Als Workaround für diese Problematik könnte auf die bereits vorgestellte Learning Router-Lösung unter Zuhilfenahme eines Controller zurückgegriffen werden, jedoch ist diese Lösung ihrerseits, wie auch bereits erläutert, mit Nachteilen verbunden. Daher ist die Entscheidung hinsichtlich der gewählten Konfigurationsoptionen auch vor diesem Hintergrund zu bewerten. Für die ausgewählte P4-Implementierungsplattform (siehe Kapitel 7.2) stehen die folgenden zwei Optionen bereit:

- Die erste Option bildet die Verwendung eines Schnittstellenprogramms, das es ermöglicht, den Inhalt einer Konfigurationsdatei an die Router-Software zu übermitteln.
- Die Konfiguration des Routers durch Verwendung der P4Runtime stellt die zweite Option dar.

Die beide gewählten Optionen und deren Implementierung werden in Kapitel 8 noch im Detail behandelt. Für eine Implementierung deines Configured Router stellt sich die Frage, auf welcher Schicht des OSI-Modells die Forwarding-Entscheidungen getroffen werden. Hierzu kann auf Basis der untersuchten Literatur keine eindeutige Aussage getroffen werden. Dies liegt vor allem darin begründet, dass, wie bereits in diesem Kapitel beschrieben, ein Configured Router Nachrichten empfangen kann, die seiner Konfiguration dienen. Dies würde bedeuten, dass eine Nachricht durch den Configured Router bis auf Schicht sieben inspiziert werden muss. Die Paket-Informationen, die für Forwarding-Entscheidungen benötigt werden, stehen jedoch bereits auf Schicht drei bereit. Dies lässt sich behaupten, da laut [76] ein Configured Router diese Entscheidungen nur aufgrund von Gruppen-, Domänen- und Subnetzinformationen trifft. Daher kann die These vertreten werden, dass Forwarding-Entscheidungen auch auf ebendieser Schicht vorgenommen werden, da es für die Leistungsfähigkeit eines Gerätes nachteilig wäre, weitere Paket-Schichten zu analysieren. Diese These wird auch dadurch unterstützt, dass ein Configured Router auch gezielt adressiert werden kann und auf diese Weise auch die gesamte Funktionalität eines LonTalk-Endknoten implementieren muss [10]. Empfängt ein Configured Router nunmehr ein an ihn selbst adressiertes Paket, muss dieses nicht weitergeleitet werden, sondern wird daraufhin durch das Gerät selbst, auf den Schichten vier bis sieben bearbeitet. Dies hat zur Folge, dass ab dieser Schicht ein Configured Router ein Paket nunmehr wie ein LonTalk-Endknoten, dessen Implementierung in diesem Kapitel abgelehnt wurde, bearbeiten würde. Aus diesen Gründen werden in dieser Arbeit die Protokollsichten zwei und drei eines Configured Router mittels P4 implementiert.

7 Detailfragen im Vorfeld der Implementierung

Da in Kapitel 6 die grundlegende Entscheidungen festgehalten sind, werden in diesem Kapitel nun weitere Implementierungsentscheidungen getroffen. Diese besitzen im Gegensatz zu den grundlegenden Entscheidungen jedoch geringer wiegende Auswirkungen und beeinflussen die Natur der P4-Implementierung deshalb auch nicht mehr in einem definierenden Ausmaß. Diese Entscheidungen umfassen zum Beispiel den physischen beziehungsweise virtuellen Testaufbau, die Struktur der Laufzeitumgebung des P4-Programms und das Erstellen beziehungsweise Empfangen von LonTalk-Nachrichten. Abschließend wird dieses Kapitel mit einer Abschätzung der Realisierbarkeit dieser Entscheidungen enden.

7.1 Wahl des Testaufbaus

In diesem Kapitel wird behandelt, welche Art des Aufbaus für die weiteren Implementierungsschritte gewählt wird und diesen damit zu Grunde liegt. Grundlegend stehen hier drei Optionen zur Auswahl, die sorgfältig gegeneinander abgewägt werden müssen, um einen stabilen Implementierungsverlauf gewährleisten zu können.

Die erste zu erwähnende Option ist der physische Aufbau. Bei diesem würde der gesamte Aufbau und alle Komponenten der LonTalk-Testumgebung, die aus mehreren Hosts und einem LonTalk-Router besteht, ausschließlich mit physischen Komponenten realisiert werden. Um diese Umgebung zu verwirklichen, bedarf es einiger grundlegender Entscheidungen. Eine dieser Entscheidungen betrifft die Wahl der Router-Hardware.

Hierfür würde es sich anbieten, auf den von der Fachhochschule angebotenen Zugang zu deren programmierbarer Netzwerkhardware, zurückzugreifen. Der damit verfügbare Hardware-Switch auf Basis von Barefoots Tofino-Chip könnte für die Implementierung herangezogen werden. Dieser besitzt hardware-spezifische Eigenschaften, wie zum Beispiel ein eigenes Architekturmodell und eine eigene Library, die einige externe P4-Funktionen bereitstellt. Diese Eigenschaften würden daher auch zu Unterschieden in der Implementierung gegenüber anderen Hardware- beziehungsweise Software-Switches führen.

Bei einem komplett physischen Aufbau ist auch zu bedenken dass die Hostgeräte, welche für das Versenden und Empfangen von LonTalk-Nachrichten genutzt werden, physischer Natur sind. Hier stehen wiederum zwei Optionen zur Verfügung:

- Die erste Option bildet die Verwendung von Smart Metern. Diese müssten ebenfalls durch die Fachhochschule Salzburg bereitgestellt werden und wurden bereits

für einen Versuchsaufbau, siehe [66], verwendet. Wie jedoch auch aus dieser Quelle hervorgeht, ist es durch die Verwendung von Smart Metern nur schwer möglich, eine zielgerichtete Analyse der übertragenen LonTalk-Nachrichten durchzuführen, da hier das eigentliche Format einer LonTalk-PDU nicht ausreichend inspiert werden kann. Auch sind die Testmöglichkeiten, die diese Herangehensweise bietet, nicht genügend, weil es die Testung der Funktionsweise des LonTalk-Routers erfordert, manuell LonTalk-Nachrichten zu erstellen. Außerdem wird in [66] erläutert, dass in einem Netzwerk mit Smart Metern auch noch andere Nachrichten übertragen werden, die nicht dem LonTalk-Protokoll zurechnet werden könnten. Solche Nachrichten würden daraufhin in einer eigenen Implementierung nicht mehr durch den LonTalk-Router weitervermittelt und damit verworfen werden. Der Grund dafür ist, dass diese für den Router keinem bekannten Format entsprechen würden. Dies wiederum könnte im schlimmsten Fall dazu führen, dass eine Kommunikation zwischen den einzelnen Empfängern und Sendern nicht mehr möglich ist beziehungsweise ein korrekter Betrieb dieser Smart Meter nicht mehr stattfinden kann.

- Eine zweite Option ergibt sich aus der Verwendung von handelsüblichen Einplatinencomputern, welche die Rolle von LonTalk-Endgeräte übernehmen. Diese Option ist allerdings für eine Implementierung nicht sehr vorteilhaft, weil hier der Managementaufwand der einzelnen Geräte hinsichtlich Setup und Sende- respektive Empfangsprogrammen nicht unerheblich wäre.

Durch die Verwendung eines physischen Aufbaus entsteht jedoch die Problematik der zu verwendenden Übertragungstechnologie. Denn durch die Auswahl der Endgeräte und des Hardware-Switches wird in diesem Fall bereits implizit eine Entscheidung hinsichtlich der genutzten Übertragungstechnologie getroffen. Denn die beteiligten Geräte implementieren üblicherweise unterschiedliche Schnittstellen. Ein Smart Meter würde zum Beispiel ein PLC-Interface bereitstellen, während ein Einplatinencomputer meist über ein Ethernet-fähiges Interface verfügt. Je nach Kombination der Geräte der LonTalk-Testumgebung müssten hier Medienkonverter verwendet werden, um eine Interoperabilität der einzelnen Geräte zu ermöglichen. Hinzukommend ist in diesem Fall die Verfügbarkeit von Medienkonvertern ein beachtenswerter Punkt. Steht ein Konverter nicht zur Verfügung, müsste hier eine Lösung selbst entworfen und umgesetzt werden. Dies wiederum steht jedoch dem Fokus dieser Arbeit entgegen, da das Hauptaugenmerk auf der Programmierung der Data-Plane und nicht auf der Implementierung eines Medienkonverters liegen sollte.

Die zweite Option ist die Realisierung eines virtuellen Aufbaus. Hierbei würde auf einem Rechner die gesamte LonTalk-Testumgebung realisiert werden. Die Entscheidung hätte die Verwendung von virtualisierten Endgeräten und Übertragungswegen sowie eines Routers auf Basis eines Software-Switches bedingt. Diese Option würde auch bewirken, dass all

diese Komponenten, die auf einem Rechner ausgeführt werden würden, einfach zu Verwalten wären und dass das Testen des gesamten Aufbaus sehr einfach wäre. Der größte Vorteil dieser Option ist allerdings, dass diese voraussichtlich mehr Flexibilität für schnelle Veränderungen der Testumgebung bietet.

Die dritte und letzte Option besteht aus einer Mischung eines virtuellen und physischen Aufbaus. Ein Beispiel hierfür wäre die Verwendung eines virtualisierten LonTalk-Routers auf Basis eines Software-Switches und die Zuhilfenahme von mehreren physischen Hosts, deren Funktionen durch mehrere Einplatinencomputer umgesetzt werden würden. Hierbei besteht jedoch die Gefahr, dass die nachteiligen Eigenschaften des physischen Aufbaus einfließen, sodass diese Option nur einen geringfügig größeren Implementierungsanreiz gegenüber einem rein physischen Aufbau bietet.

Abschließend ist noch auszuführen, welche dieser drei Optionen umgesetzt werden sollten. Dabei überwiegen klar die Vorteile des virtuellen Aufbaus, da diese Option durch Flexibilität und Simplizität überzeugen kann. Der Nachteil der komplett physischen Option liegt vor allem in der Problematik des verwendeten Übertragungswegs begründet. Auch verkompliziert sich in diesem Fall die Entwicklung unnötigerweise, was besonders auf die Verwendung von Smart Metern, aber auch auf Einplatinencomputer zurückzuführen ist. Schlussendlich spielt auch die Verfügbarkeit und der Zugang zu den Smart Metern und dem Hardware-Switch eine große Rolle, da in den ersten zwei Quartalen des Jahres 2020 der Zugang zu diesen aufgrund der vorherrschenden Covid-19-Pandemie länger eingeschränkt war. Aus dieser Vielzahl von Gründen wurde beschlossen, die Implementierung auf Basis eines virtuellen Aufbaus zu realisieren.

7.2 Auswahl des Software-Switches

Dieses Kapitel widmet sich der Frage, auf welche Art und Weise der LonTalk-Router umgesetzt werden soll. In Kapitel 7.1 wurde bereits die Entscheidung getroffen, dass dieser virtualisiert und daher nicht auf einer Hardware-Plattform, wie zum Beispiel einem Tofino-Chip basierendem Switch, realisiert werden soll. Um dieses Vorhaben umzusetzen, bedarf es eines Software-Switches, der in diesem Kapitel ausgewählt wird.

Hier gibt es grundsätzlich zwei Optionen, die als Basis für eine Umsetzung des LonTalk-Router dienen können: BMv2 [40] und Open vSwitch [46]. Wie das Projekt PISCES [83] verdeutlicht, kann auch die Data-Plane eines Open vSwitch mittels P4 programmiert werden. Dennoch wurde für diese Arbeit beschlossen, auf eine auf dem BMv2 basierende Lösung zurückzugreifen.

Die Entscheidung wurde zugunsten BMv2 gefällt, weil dieser die Referenzimplementierung eines P4-Software-Switches ist. Es ist daher auch möglich, wie in Kapitel 4.7.2 beschrieben, P4-Programme direkt mit dem Referenzcompiler für BMv2 zu kompilieren, ohne einen anderen nicht-offiziellen Compiler zu verwenden. Die Nachteile, die eine Verwendung eines BMv2-Switches mit sich bringt, sind für diese Arbeit nicht ausschlaggebend. Dies liegt darin begründet, dass der zu entwerfende LonTalk-Router nicht für kommerzielle Gründe gedacht ist und daher auch nicht Anforderungen wie hoher Throughput und niedrige Latenz erfüllen muss [40]. Würden diese Anforderungen gestellt, müsste auf einen Switch mit hohem Reifegrad wie Open vSwitch zurückgegriffen werden. BMv2 ist hingegen für Entwicklungstätigkeiten, Testen und Debugging von P4-Programmen gedacht [40]. Außerdem bedingt der hohe Verbreitungsgrad aufgrund dessen Status als Referenz-P4-Software-Switch auch eine ausreichend vorhandene Dokumentation. Dies wiederum hat zur Folge, dass der Einstieg in die P4-Programmierung, trotz des großen Funktionsumfangs von BMv2, einfach gelingen sollte.

Auf Basis des BMv2 können mehrere Programme, nunmehr P4-Ziele genannt, gewählt werden, um den LonTalk-Router zu implementieren. [84] zufolge bieten sich hier die folgenden fünf Optionen, welche kurz vorgestellt werden:

- `simple_router` und `l2_switch`: Diese P4-Ziele sind Beispiele für die Vielseitigkeit der BMv2-Library und sollen zeigen, dass auch die Umsetzung verschiedenster Architekturmodelle mit BMv2 möglich ist [84]. Diese Ziele sind jedoch für die Implementierung eines LonTalk-Routers nicht vorteilhaft, da die Menge der hier verwendbaren intrinsischen Metadaten nicht sehr groß ist (siehe `simple_router.cpp` beziehungsweise `l2_switch.cpp` in [84]). Aus den genannten Dateien geht beispielsweise hervor, dass diese P4-Ziele die Verwendung von Multicastgruppen nicht unterstützen. Außerdem besteht laut [84] in diesem Fall zum momentanen Zeitpunkt nicht die Möglichkeit, P4₁₆-Programme auszuführen und damit eine faktische Beschränkung auf P4₁₄. Aus diesen Gründen ist von der Verwendung dieser P4-Ziele generell abzusehen.
- `psa_switch`: Die neueste Option bietet sich in Form von `psa_switch`. Dieses P4-Ziel wurde entwickelt, um das PSA-Architekturmodell umzusetzen. Da jedoch die Arbeiten an der ersten Version der PSA-Spezifikation laut [51] erst im November 2018 abgeschlossen wurden, ist die Implementierung dieses P4-Ziels zum Erstellungszeitpunkt dieser Arbeit noch nicht abgeschlossen. Dies liegt laut [84] auch daran, dass dafür weitere Arbeiten an dem P4-Referenzcompiler nötig sind. Aus diesen Gründen wird für diese Arbeit von der Verwendung dieses P4-Ziels abgesehen.
- `simple_switch`: `simple_switch` stellt laut [85] die de-facto Lösung für alle Nutzer*innen dar. Dies lässt sich unter anderem auch auf dessen breiten Funktionsumfang (zum

Beispiel Multicasting) und die Einfachheit der Verwendung zurückführen. Auch unterstützt dieses P4-Ziel weiterhin nahezu alle Programme auf Basis von P4₁₄ und natürlich auch neuere P4₁₆-Programme [84]. Dieses P4-Ziel setzt außerdem auf das bereits in Kapitel 4.6.4 erwähnte V1-Modell und setzt damit die Funktionen des P4-Sprachkerns und des V1-Modells um [64]. Mit dieser Architektur und einer Vielzahl von grundlegenden, externen Funktionen, wie zum Beispiel dem Klonen von Paketen und/oder dem Update von Header-Checksummen, lassen sich fundamentale Programme umsetzen. Eine detaillierte Übersicht über alle externen Funktionen und verfügbaren Metadaten findet sich in [64] beziehungsweise [85]. Außerdem bietet dieses P4-Ziel die Möglichkeit, einen Controller über eine Thrift-API anzubinden, um die Weiterleitungstabellen des Ziels zu befüllen [84]. All diese Fähigkeiten und Funktionen sprechen daher für die Verwendung von simple_switch.

- simple_switch_grpc: Das nunmehr letzte P4-Ziel ist simple_switch_grpc. Dieses basiert auf simple_switch und hat daher auch denselben Funktionsumfang und dieselben Fähigkeiten [84]. Das Merkmal, das diese beiden Ziele unterscheidet, ist, dass simple_switch_grpc Transmission Control Protocol-Verbindungen (TCP) zulässt [84]. Dies wiederum ermöglicht den Austausch von Nachrichten mit einem Controller und die Verwendung der P4Runtime. Diese Fähigkeiten geben daher auch den Ausschlag dafür, dass die simple_switch_grpc-Option in dieser Arbeit Anwendung findet.

7.3 Wahl der Ausführungsumgebung

In diesem Kapitel wird eine Auswahl der in dieser Arbeit verwendeten Ausführungsumgebung getroffen. Diese dient einerseits dazu, ein P4-Programm zu kompilieren. Andererseits wird diese auch dazu benötigt, um den LonTalk-Router und die den Router umgebende Topologie zu starten.

In Rahmen dieser Arbeit werden in Kapitel 7.4 zwei Entwicklungsplattformen genauer vorgestellt. Die beiden Entwicklungsplattformen enthalten jeweils eine eigene Ausführungsumgebung, die den Start-Prozess eines P4-Beispiels vereinfachen soll. Diese Umgebungen ähneln sich sehr und unterscheiden sich lediglich in Details. Da die Wahl der zu verwendeten Ausführungsumgebung auch ein Auswahlkriterium für eine der beiden Plattformen ist, werden diese nunmehr genauer beleuchtet.

```

1 {
2   "program": "reflector.p4",
3   "switch": "simple_switch",
4   "compiler": "p4c",
5   "options": "--target bmv2 --arch v1model --std p4-16",
6   "switch_cli": "simple_switch_CLI",
7   "cli": true,

```

```

8   "pcap_dump": true,
9   "enable_log": true,
10
11 ...
12 ...
13
14 "topology": {
15   "assignment_strategy": "12",
16   "links": [["h1", "s1"]],
17   "hosts": {
18     "h1": {
19       }
20   },
21   "switches": {
22     "s1": {
23       }
24   }
25 }
26 }
```

Listing 9: p4app.json-Datei enthalten in [86]

- Ausführungsumgebung der Entwicklungsplattform der ETH Zürich [78]: Um die P4-Beispielprogramme auszuführen, die in dieser Entwicklungsplattform enthalten sind, wird auf eine eigene Bibliothek zurückgegriffen. Diese wird P4-utils genannt und hat laut [87] die Aufgabe P4-Anwendungen und Test-Netzwerke einfacher zu entwerfen, zu starten und das Debugging zu erleichtern. Bei P4-utils handelt es sich um eine Erweiterung für Mininet, sodass für Mininet auch die Integration von P4-Geräten ermöglicht wird. Diese Ausführungsumgebung lehnt sich stark an die offizielle Ausführungsumgebung namens p4app [88] an. Grundlegend unterstützt P4-utils, in seiner aktuellen Fassung, die Verwendung der P4Runtime nicht [88]. Um mithilfe dieser Umgebung eine P4-Programmumgebung zu starten, ist es lediglich notwendig, den Befehl `sudo p4run` auszuführen. Dies bewirkt, dass eine Mininet-Umgebung nach den Vorgaben, welche in der Datei `p4app.json` festgehalten sind, bereitgestellt wird. Diese Mininet-Umgebung enthält bereits auch die gewünschten P4-Switches, sofern diese in `p4app.json` definiert wurden. Ein einfaches Beispiel für den Inhalt einer solchen Datei findet sich in Listing 9. Diese Datei ist Teil des „Packet Reflector“-Beispiels (siehe [86]). In dieser JSON-Datei ist unter anderem der Name des zu verwendenden P4-Programms, die Variante des BMv2-Switches, etwaiger Compileroptionen und eine Beschreibung der zu erzeugenden Topologie enthalten. Jedoch sind bei dieser Ausführungsumgebung die Auswirkungen der Parameter, mit der die beteiligten Programme, darunter der Software-Switch und der für diesen vorgesehene Compiler, gestartet werden, schwer ersichtlich, da diese in mehreren Dateien beschrieben werden. Die Ausführung des Befehls `sudo p4run` mündet in einer Mininet-Konsole, die für weitere Eingaben, wie das Testen der Topologie, genutzt werden kann. Grundsätzlich ist anzumerken, dass die Unterschiede zwischen p4app und P4-utils gering sind. Ein Beispiel hierfür ist: Um innerhalb einer erstellten Topologie Befehle auf einzelnen enthaltenen Hosts auszuführen, enthält P4-utils einen

neuen Befehl namens `mx`. Dieser wird, gefolgt von einem Hostbezeichner und einem herkömmlichen Linux-Befehl genutzt. Dies hat gegenüber p4app den Vorteil, dass kein neues Konsolenfenster gestartet werden muss, um einen Befehl auf einem Host auszuführen.

- Ausführungsumgebung der offiziellen Entwicklungsplattform [77]: Diese basiert ebenfalls auf der Verwendung und Integration von P4-Geräten in Mininet. Die Überschneidungen dieser Ausführungsumgebung mit p4app und P4-utils sind gravierend, obwohl hier das Starten einer P4-Programmumgebung anders umgesetzt wurde. Dies zeigt sich durch die Integration des Build-Management-Tools `Make` [89]. Außerdem wird hier auch standardmäßig die Verwendung der P4Runtime unterstützt. Um eine P4-Programmumgebung zu starten, ist es lediglich notwendig, den Befehl `sudo make run` auszuführen [77]. Dieser Befehl kompiliert das P4-Programm, startet die Mininet-Umgebung mit einer definierten Topologie und konfiguriert alle Host und Switches automatisch mit den in einer JSON-Datei gemachten Angaben. Der Inhalt einer dieser JSON-Dateien ist in Listing 10 zu finden. Diese JSON-Datei ist Teil des Beispiels „Implementing Basic Forwarding“, das wiederum in der offiziellen Entwicklungsplattform enthalten ist. Bei dieser Ausführungsumgebung sind die Parameter des Referenzcompilers außerdem einfacher veränderbar, als dies bei der Ausführungsumgebung der Entwicklungsplattform der ETH Zürich der Fall ist. Hinzukommend lassen sich die Auswirkungen der genutzten Parameter, ohne eine Analyse von weiteren Quelldateien der Ausführungsumgebung bestimmen.

```

1 {
2   "hosts": {
3     "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
4            "commands": ["route add default gw 10.0.1.10 dev eth0",
5                          "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"
6                         ]},
7     ...
8     ...
9     "h4": {"ip": "10.0.4.4/24", "mac": "08:00:00:00:04:44",
10           "commands": ["route add default gw 10.0.4.40 dev eth0",
11                          "arp -i eth0 -s 10.0.4.40 08:00:00:00:04:00"
12                         ]}
13   },
14   "switches": {
15     "s1": { "runtime_json" : "pod-topo/s1-runtime.json" },
16     "s2": { "runtime_json" : "pod-topo/s2-runtime.json" },
17     "s3": { "runtime_json" : "pod-topo/s3-runtime.json" },
18     "s4": { "runtime_json" : "pod-topo/s4-runtime.json" }
19   },
20   "links": [
21     ["h1", "s1-p1"], ["h2", "s1-p2"], ["s1-p3", "s3-p1"], ["s1-p4",
22       "s4-p2"],
23     ["h3", "s2-p1"], ["h4", "s2-p2"], ["s2-p3", "s4-p1"], ["s2-p4",
       "s3-p2"]
24   ]
25 }
```

Listing 10: JSON-Konfigurationsdatei enthalten in [90]

Bezüglich der Gemeinsamkeiten der vorgestellten Ausführungsumgebungen wurde bereits erwähnt, dass diese jeweils auf Mininet aufbauen. Dies bewirkt auch, dass Mininet für jedes Gerät der Topologie einen eigenen Namespace vorsieht. Der Vorteil der Verwendung von Netzwerk Namespaces ist laut [87], dass diese Geräte nunmehr einen eigenen, von den anderen getrennten, Network Stack besitzen. Ein Nachteil von Mininet ist jedoch, dass die ausgewählte Übertragungstechnologie nicht mehr frei gewählt werden kann, denn Mininet verwendet standardmäßig virtuelle Ethernet-Interfaces, die im Kernel der Linux-Distribution enthalten sind [87].

Um diesem Nachteil entgegenzutreten, ist es möglich, gänzlich auf die Verwendung von Mininet zu verzichten. Der Verzicht auf Mininet kann durch einen manuellen Aufbau der Topologie, das händische Kompilieren des P4-Programms und das Starten von P4-Switches über einen Konsolenbefehl erreicht werden. Dieses Konzept ist in [91] prinzipiell dargestellt. Diese Herangehensweise, eine P4-Programmumgebung zu erstellen, besticht vor allem dadurch, dass im Vergleich zu den „automatisierten“ Methoden alle ausgeführten Befehle direkt einsehbar sind. Dies führt auch dazu, dass alle Parameter hinsichtlich des Starts des Switches und der Kompilierung des P4-Programms auf leichte Art und Weise veränderbar sind. Neben einer deutlichen Vereinfachung wird dadurch, wie einleitend erläutert, die Verwendung anderer Übertragungstechnologien ermöglicht. Eine Parallele, die sich zu den Mininet-Lösungen zeigt, ist die Verwendung von Network Namespaces. Die in dieser Arbeit vorgenommene Implementierung einer solchen Ausführungsumgebung findet sich in Kapitel 8.1.

7.4 Wahl der Entwicklungsplattform

In diesem Kapitel wird eine Entwicklungsplattform ausgewählt. Diese dient dazu, alle benötigten Programme, wie zum Beispiel den Software-Switch, andere Hosts und die Programme zum Senden beziehungsweise Empfangen von LonTalk-Nachrichten, ausführen und anwenden zu können. Den Kern der Entwicklungsplattform stellt in jedem Fall eine Linux-Distribution dar, da diese beispielsweise für den Software-Switch und andere beteiligte Programme benötigt wird.

Die bevorzugte Umsetzung einer Entwicklungsplattform basiert darauf, dass das grundlegende Betriebssystem, eine Linux-Distribution, als virtuelle Maschine (VM) konzipiert ist. Eine Herangehensweise auf dieser Basis ist, alle benötigten Programme und Pakete manuell auf dieser VM zu installieren. Dies würde bedeuten, dass beispielsweise Programmpakete wie Mininet, Protobuf, gRPC, BMv2, P4Runtime, der P4-Referenzcompiler und viele ihrer Dependencies händisch installiert werden müssten. Diese Herangehensweise bietet den

Nachteil, dass eine Out-of-the-box-Funktionalität nicht gegeben ist und ein funktionierendes Zusammenspiel all dieser Programme nicht garantiert werden kann. Überdies hat ein manueller Aufbau das Potenzial, hinsichtlich Fehlersuche und Troubleshooting aufwendig beziehungsweise kompliziert zu sein, wenn eine einwandfreie Interaktion aller Programme nicht gegeben ist. Diese Gründe bewirken, dass ein manueller Aufbau einen komplexen Einstieg verursacht und daher nicht die erste Wahl bezüglich der Entwicklungsplattform sein sollte.

Allerdings gibt es neben der manuellen Realisierung der Entwicklungsplattform auch noch andere, praktischere Möglichkeiten. Diese bieten sich in der Gestalt von bereits vorgefertigten Plattformen, auf die für den Einstieg in die P4-Entwicklung zurückgegriffen werden kann. Eine Auseinandersetzung mit vorgefertigten Plattformen wird in dieser Masterarbeit anhand zweier Vertreter vorgenommen, die bereits in Kapitel 7.3 erstmalig erwähnt wurden. Diese Plattformen können auf automatisierte Art bereitgestellt beziehungsweise realisiert werden. Der allgemeine Vorteil dieser Realisierungsmethode ist vor allem deren Simplizität. Um auf diese Weise eine Plattform zu realisieren, ist unter Verwendung zweier Programme nur ein Kommandozeilenbefehl nötig: `vagrant up`. Die beteiligten Programme dabei sind VirtualBox (siehe Kapitel 1.3) und Vagrant [92]. Vagrant ist eine Programmumgebung, die es auf einfache Art und Weise ermöglicht, virtuelle Maschinen zu provisionieren. Ein weiterer Vorteil von bereits vorgefertigten Plattformen ist, dass die benötigten Programme bereits installiert sind. Daher ist hier auch eine Out-of-the-box-Funktionalität gegeben. Außerdem verfügen diese Plattformen über einige Beispiel-P4-Programme, die ein schnelles Erlernen der Sprache ermöglichen. Folgend werden nunmehr die Unterschiede und Grundzüge zweier vorgefertigten Plattformen vorgestellt.

- Die erste Option ergibt sich aus der Nutzung der offiziellen Tutorial-VM [77]. In dieser durch das P4-Sprachkonsortium veröffentlichten VM sind bereits alle für die diversen Programme benötigten Dependencies enthalten. Diese VM legt seinen Fokus vor allem auf die Verwendung von `simple_switch_grpc` anstatt von `simple_switch`. Dieser Umstand ermöglicht auch die Nutzung der P4Runtime, ohne eine erstmalige, manuelle Kompilierung von `simple_switch_grpc` vornehmen zu müssen.
- Als zweite Option bietet sich die VM der Eidgenössischen Technischen Hochschule Zürich an [78]. Diese VM wird von deren Networked Systems Group betrieben, gewartet und auch für die Lehre herangezogen [78]. Ein Vorteil dieser VM ist, dass bereits eine größere Anzahl von P4-Beispielprogrammen enthalten ist. Jedoch sind einige dieser Programme bereits älter und daher auch nicht immer mit neueren Versionen des BMv2-Switches oder des P4-Referenzcompilers kompatibel [78]. Aus einigen der enthaltenen Beispiele ist jedoch die Verwendung von externen Funktionen des `simple_switch` gut ersichtlich. Da diese Umgebung gänzlich auf die Verwen-

dung des simple_switch ausgerichtet ist, kann diese, ohne eine Kompilierung von simple_switch_grpc, nicht mit der P4Runtime genutzt werden. Wie sich bei einer Untersuchung zeigte, ist eine Abänderung dieser Plattform nicht ohne Hindernisse möglich. Dies liegt darin begründet, dass die Entwicklungsplattform Programme enthält, deren Versionen aufeinander abgestimmt sein müssen. Daraus resultiert, dass es teilweise nicht möglich war, einzelne Programme zu aktualisieren. Dies war zum Beispiel für die aktuellsten Versionen von BMv2 und des P4-Referenzcompilers der Fall. Versuche, die aktualisierten Versionen dieser Programme zu nutzen, scheiterten an einer fehlenden Kompatibilität mit anderen Programmen, im speziellen mit gRPC. Diese neuesten Versionen von BMv2 und des dazugehörigen Compilers unterscheiden sich allerdings nicht in einem großen Ausmaß von den genutzten, im Herbst 2019 veröffentlichten Versionen. Veränderungen zwischen den Versionen sind beispielsweise im verwendbaren Befehlssatz bemerkbar. Dies drückt sich in dem Befehl `log_msg()` aus, der laut [93] in einer neueren Version hinzukam und unter anderem ein einfacheres Debugging ermöglicht. Da die Nutzung eines solchen Features für die Implementierungstätigkeiten dieser Arbeit nicht ausschlaggebend ist, wurde mit den auf den virtuellen Maschinen vorinstallierten Programmversionen gearbeitet.

Im Verlauf des Analyseprozesses der beiden VMs und deren Ausführungsumgebungen wurde mit beiden Entwicklungsplattformen gearbeitet. Um die P4Runtime nutzen zu können und dafür keine Veränderungen an der Entwicklungsplattform der ETH Zürich vornehmen zu müssen, wurden die Implementierungsarbeiten, die in dieser Arbeit vorgestellt werden, auf der offiziellen VM des P4-Sprachkonsortiums durchgeführt. Diese Entscheidung wurde außerdem dadurch begünstigt, dass die Nutzung einer der in Kapitel 7.3 vorgestellten, VM-spezifischen Ausführungsumgebungen durch die Verwendung einer manuellen Ausführungsumgebung nicht mehr nötig war. Aus den in der VM der ETH Zürich enthaltenen P4-Programmen ließ sich nichtsdestotrotz Wissen für die Implementierung eines LonTalk-Routers gewinnen.

7.5 Wahl des Nachrichtenerstellungs-Tools

In diesem Kapitel wird die Auswahl des Tools, das zur Erstellung und zum Empfang der LonTalk-Nachrichten dienen soll, getroffen. Dieses Tool wird dazu benötigt, um durch Hosts, die in der Topologie enthaltenen sind, Nachrichten über den LonTalk-Router weitervermitteln und um damit die korrekte Funktionsweise des Routers testen zu können.

Eine einfache Lösung, um Pakete zu versenden und zu empfangen, bietet sich in der Verwendung von Scapy [94]. Scapy ist ein Programm beziehungsweise eine Bibliothek, die in

der Programmiersprache Python geschrieben wurde und einen großen Funktionsumfang besitzt. Dies drückt sich laut [94] darin aus, dass mit Scapy Netzwerkpakete gesendet, bestehender Netzwerkverkehr untersucht und modifiziert werden kann. Das Ziel hinter der Entwicklung von Scapy war eine „domänenspezifische Sprache“ zu entwickeln, die es ermöglicht, alle Arten von Datenpakete zu beschreiben [94].

Der wichtigste Grund, der für eine Verwendung von Scapy spricht, ist, dass damit Pakete nach eigenen Vorstellungen erstellt werden können. Obgleich in Scapy bereits eine Vielzahl von verwendbaren Protokollen beschrieben ist, kann durch die Möglichkeit, den Aufbau eines Pakets selbst zu beschreiben, ein Kommunikationsprotokoll wie LonTalk erst umgesetzt werden. Um diesen Zweck zu erfüllen, stehen unter Scapy einige Klassen und Funktionen zur Verfügung. Mit deren Hilfe können Felder einer PDU selbst definiert werden. Überdies kann diese PDU daraufhin versendet und empfangen werden, sodass eine Ende-zu-Ende-Lösung für die Erstellung und Entwicklung von eigenen Kommunikationsprotokollen entsteht. Dieser große Funktionsumfang bewirkt, dass die Wahl des Tools, das zur Erstellung von LonTalk-Nachrichten genutzt wird, auf Scapy fällt. Abschließend ist anzumerken, dass für Scapy eine begrenzte Anzahl von Übertragungsmedien zur Verfügung stehen. Diese Auswahl lässt sich in [94] einsehen. Sie bietet Beispiele für unterstützte Übertragungsmedien, wie: Controller Area Network (CAN), Ethernet und Bluetooth.

7.6 Auswahl der Übertragungstechnologie

In diesem Kapitel wird die Entscheidung getroffen, welche Übertragungstechnologie in der Topologie, die zu Testzwecken zum Einsatz kommt, als Basis für die Kommunikation zwischen den Geräten verwendet wird.

Hier liegen Einschränkungen vonseiten des Software-Switches, von Scapy und der Ausführungsumgebung, in Form der für die Erstellung von Netzwerk-Interfaces benötigten Befehlsfamilie `ip link`, vor. Details zu Scapy und `ip link` finden sich in [94] beziehungsweise [95]. Die besagten drei Teile des Testaufbaus müssen jeweils mit der ausgewählten Übertragungstechnologie interagieren können. Außerdem muss die Übertragungstechnologie auch eine Interaktion zwischen den einzelnen beteiligten Komponenten, wie Hosts und Software-Switches, gewährleisten. Da diese Bedingungen, vor allem auf Seiten Scapys, die Bandbreite der verwendbaren Übertragungsmedium bereits sehr einschränken, gibt es die Möglichkeit, zwischen zwei Optionen zu wählen.

- CAN: Wie aus [10], [65] und anderer Literatur zum Thema LonTalk hervorgeht, ist die Verwendung eines Bus-Systems bei LonTalk-Anwendungen äußerst üblich und daher die naheliegende Wahl. Eine LonTalk-Anwendung, die in [66] im Fokus steht,

ist der Bereich Smart Metering. Dabei werden üblicherweise die zu übertragenden Daten über ein Stromnetz transportiert. Dabei liegen logisch gesehen alle Geräte meist auf demselben Leitungsabschnitt. In LonWorks wird ein solcher Leitungsabschnitt üblicherweise als Segment tituliert [10]. Ein solches Übertragungsmedium kann auch als Bus-System bezeichnet werden. Um ein Bus-System als Basis der Netzwerktopologie festlegen zu können, gibt es unter Scapy bereits eine Option: CAN. Dies bedeutet, dass Scapy dieses Übertragungsmedium nutzen kann, um selbst erstellte PDUs zu übertragen.

- Ethernet: Eine Lösung, die ebenfalls durch Scapy unterstützt wird, basiert auf Ethernet. Hier entfallen jedoch die für LonTalk üblichen Eigenschaften eines Bus-Systems. Dies bedeutet, dass für das Design der Topologie die Möglichkeit entfällt, mehrere Hosts an ein Switch-Interface anzuschließen und somit eine direkte Assoziation eines Segments mit einem einzigen Host erzeugt wird. Der Vorteil dieser Lösung ist jedoch, dass sowohl der Software-Switch als auch Scapy standardmäßig Ethernet verwenden und dass dessen Verwendung auch bereits in deren jeweiliger Dokumentation, [40] respektive [94], ausführlich beschrieben ist.

Allgemein musste erst überprüft werden, ob die praktische Verwendung von CAN mit dem Software-Switch BMv2 möglich ist. In eigenen Experimenten wurde ersichtlich, dass es möglich ist, ein virtuelles CAN-Interface erfolgreich mit dem BMv2-Switch zu verbinden und darauffolgend auch durch Scapy erzeugte Nachrichten mit diesem zu nutzen. Jedoch gibt es bei dieser Interface-Art die Einschränkung, dass lediglich acht Byte Nutzdaten in einem Frame transportiert werden können [96]. Um diese Problematik zu veranschaulichen: Bei LonTalk ist bereits alleine das Adressformat des Typs „Unique ID“, siehe Abbildung 22, mit neun Byte deutlich zu lang. Werden die restlichen Teile des Headers auf Schicht zwei und drei hinzugerechnet, wird dieses Limit noch weiter überschritten. Abhilfe für diese Begrenzung bietet die Nutzung von Controller Area Network Flexible Data-Rate (CAN FD) [97]. Diese CAN-Erweiterung ermöglicht die Nutzung von bis zu 64 Byte Nutzdaten statt der bisherigen acht Byte. Neben `ip link` ermöglicht auch Scapy die Nutzung dieses Modus, sodass nunmehr auch Frames mit mehr als acht Byte Nutzdaten versendet werden können. Wie jedoch bereits in Kapitel 5.2.2 erläutert, kann die NPDU bei LonTalk bis zu 246 Byte lang sein, sodass ein Paket maximaler Länge auch nicht mittels CAN FD transportiert werden kann. Zum Erstellungszeitpunkt dieser Arbeit ist es leider nicht möglich gewesen, die Maximum Transmission Unit (MTU) eines virtuellen CAN-Interfaces auf über 72 Byte, das entspricht maximal 64 Byte Nutzdaten, festzulegen. Aus diesen Gründen wurde entschieden, für die Topologie (siehe Abbildung 23) Ethernet als primäre Übertragungstechnologie zu nutzen, da bei dieser auch die maximale Länge einer LonTalk-Nachricht übermittelt werden kann.

Abschließend muss in diesem Kapitel noch erwähnt werden, dass es durch die Verwendung von Ethernet, einer für LonTalk-Nachrichten unüblichen Übertragungstechnologie, zu der Einschränkung kommt, dass ergänzend zu dem auf Schicht zwei befindlichen Header des LonTalk-Protokolls noch Felder des Ethernet-Headers derselben Schicht definiert werden müssen. Aus demselben Grund enthalten die auf Ethernet basierenden LonTalk-Nachrichten auf OSI-Schicht eins auch keine für LonTalk typische Präambel. Diese kann auch nicht innerhalb eines P4-Programms implementiert werden, da diese Schicht für den/-die P4-Programmierer*in verborgen ist und die Aufgaben dieser OSI-Schicht bei Verwendung des BMv2-Switches durch das darunterliegende Linux-Betriebssystem übernommen werden. Würde in dieser Arbeit kein Software-Switch zugrunde liegen, wäre die Behandlung der Schicht Aufgabe und Teil der verwendeten Router-Hardware.

7.7 Auswahl der Topologie

In diesem Kapitel werden die Grundsätze für den Aufbau und Entwurf der Topologie festgelegt. Darüber hinaus wird auf dieser Basis die Testtopologie erstellt und präsentiert.

Das Ziel der Topologie ist es, die Funktionsweise des LonTalk-Routers ausführlich testen zu können, um etwaige Lücken in dessen Implementierung aufzudecken. Dabei muss vor allem der Adressinformation, aufgrund dieser der Router seine Weiterleitungsentscheidungen trifft, besondere Aufmerksamkeit gewidmet werden. Daraus ergibt sich, dass Gruppen-, Subnetz- und Domäneninformation den Hosts der Topologie so zugeteilt werden muss, dass diese getestet werden kann. Diese Entscheidung hat auch zur Folge, dass die Anzahl der im Netz enthaltenen Hosts zu definieren ist.

In Abbildung 23 findet sich die Topologie, mit der versucht wird, diesen Grundsätzen gerecht zu werden und sie umzusetzen. Dafür sind sechs Hosts, sechs Subnetze und zwei Domänen vorgesehen. Bezüglich der Subnetzinformation ist zu erkennen, dass ein jedes Übertragungssegment jeweils mit einem eigenen Subnetz assoziiert ist, weil sich ein Subnetz laut [10] nicht über einen intelligenten Router hinweg erstrecken kann. Dies bedeutet, dass ein Subnetz einer Domäne maximal auf einem Segment existieren kann. Durch eine faktische Verkettung eines Hosts mit einem zugehörigen Segment repräsentiert ein Host alle auf einem Segment möglichen Hosts. Hypothetisch gesehen würden, wenn ein Host dieser Topologie eine LonTalk-Nachricht empfängt, alle Hosts, welche sich auf demselben Segment befinden, diese Nachricht ebenfalls empfangen. Dies würde zur Folge haben, dass alle Hosts dieses Segments anhand der in der Nachricht enthaltenen Adressinformation individuell entscheiden müssten, ob sie das Paket akzeptieren oder verwerfen. Wird dieser Entscheidungsmechanismus in dieser Arbeit nicht implementiert, kann ein Host jedes auf

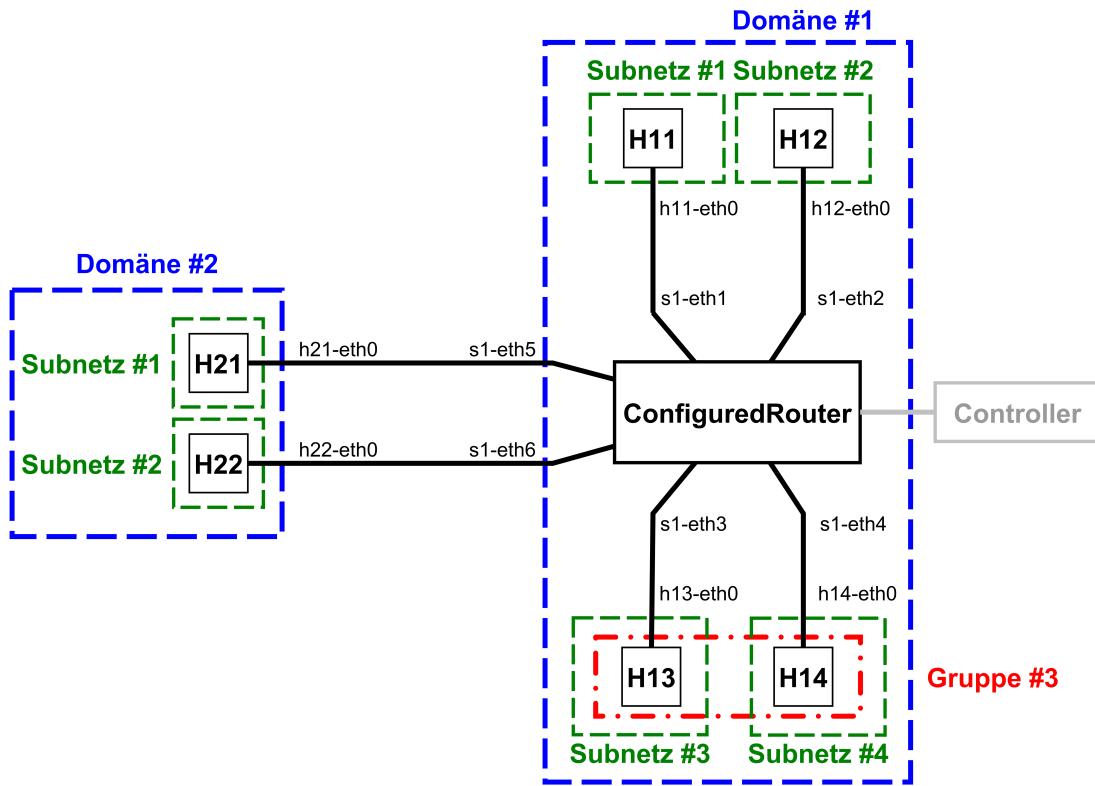


Abbildung 23: Aufbau der Topologie

seinem Segment empfangene Paket anzeigen. Die hohe Zahl an Subnetzen dient vor allem dazu, das Forwarding-Verhalten des Routers besser testen zu können und damit etwaige Implementierungsfehler offenzulegen. Die zuvor erläuterte Verkettung begründet sich dadurch, dass das gewählte Transportmedium nur eine Zuordnung von maximal einem Host pro Übertragungsstrecke beziehungsweise Segment zulässt. Die Auswahl der in der Topologie verwendeten Transporttechnologie wurde in Kapitel 7.6 im Detail erläutert.

In Abbildung 23 ist außerdem erkennbar, dass eine logische Gruppe, mit der dazugehörigen Nummer drei, definiert wurde. Diese wurde integriert, um die Multicast-Funktionalität des Routers bei der Verarbeitung von Gruppeninformationen zu testen. Diese Verarbeitungstätigkeit tritt bei der Verwendung des Adressformats „Group“ auf. Die jeweiligen Adressformate, welche für das Testen von Nachrichten, die an Subnetze beziehungsweise Gruppen adressierte sind, findet sich in Kapitel 5.2.2. Um die Domäneninformationen testen zu können, wurden die Hosts der Topologie in zwei Domänen unterteilt. Die Hosts H21 und H22 sind daher der Domäne zwei zuzuordnen und liegen, wie die Hosts der Domäne eins, jeweils in eigenständigen Subnetzen. Erst durch die Aufteilung aller Hosts in zwei getrennte Domänen wird es möglich, das Versenden von Nachrichten zwischen Hosts verschiedener Domänen zu testen. Im rechten Teil der Abbildung 23 findet sich außerdem noch ein Controller, in Grau dargestellt, der für die Konfiguration durch die P4Runtime genutzt wird. Der verwendete Controller wird in Kapitel 8.5 noch genauer behandelt.

7.8 Machbarkeitsabschätzung

In diesem Kapitel wird eine Vorabschätzung getroffen, ob die bisher in den Kapiteln 6 und 7 festgelegten Herangehensweisen implementierbar sind und welche Hindernisse noch vor der Implementierungsphase auftreten könnten.

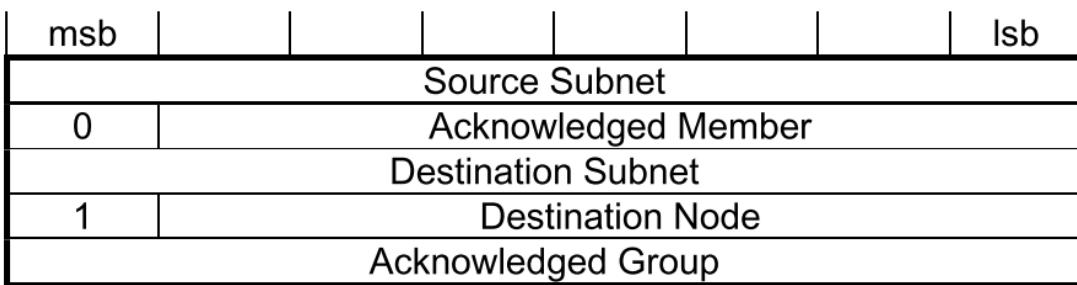


Abbildung 24: Adressformat „Group Acknowledgement“ nach [10]

Eingangs wurde in Kapitel 6.1 erwähnen, dass die Auswahl des LonTalk-Protokolls als Objekt einer P4-Implementierung ein noch nicht beschrittener Weg ist und damit voraussichtlich einige Einschränkungen einhergehen werden. Einschränkungen, die bereits diskutiert wurden, sind die Auswahl des zu implementierenden LonWorks-Gerätetypen und die damit einhergehende Konfigurationsproblematik. Außerdem ist es wahrscheinlich, dass durch eine praktische Auseinandersetzung mit der gewählten P4-Implementierungsplattform (siehe Kapitel 7.2) und anderer Entwicklungstools neue Hindernisse entstehen können, die bisher noch nicht abzuschätzen sind. Auch ist fragwürdig, welche Schichten des LonTalk-Protokolls tatsächlich standardkonform implementiert werden können. Es kann jedoch davon ausgegangen werden, dass die Programmierung der Schichten zwei und drei bewerkstelligt werden kann. Eine Problematik, die jedoch bereits bei Recherchearbeiten zu den PDUs dieser Schichten zutage getreten ist, ist, dass der offizielle Standard [76] nicht sehr umfangreich in seiner Ausführung ist und nicht vollständig auf einzelne Details eingeht. Ein Beispiel für diesen Umstand sind die Adressformate des LonTalk-Protokolls. Diese werden im Standardisierungsdokument [76] nur sehr oberflächlich beleuchtet. Außerdem ergab sich diesbezüglich auch bereits ein Problem hinsichtlich eines kohärenten Bildes aller beteiligten Quellen: [76], [10], [71], [65] und [66]. Diese stellen beispielsweise für den Adressstyp zwei beziehungsweise 2b eine unterschiedliche Feldstruktur dar. Die Unterschiede zweier Quellen hinsichtlich dieses Beispiels sind in den Abbildungen 21 und 24 abgebildet. In den Bildunterschriften der beiden Abbildungen ist die jeweils zugehörige Quelle vermerkt. An diesen Abbildungen erkennt man, dass [10] bei diesem Adressstyp das letzte Feld, **Member**, auslässt. Auch für die Erstellung der LonTalk-Nachrichten mittels Scapy gibt es einige Punkte, die nur durch eine experimentelle Implementierung gelöst werden können, da die Dokumentation von Scapy [94] zwar sehr umfassend ist, aber Detailaspekte, wie zum Beispiel die Erstellung von Headerfeldern, welche kein Vielfaches von

acht Bit sind, vernachlässigt. Darüber hinaus wird sich erst im Verlauf der Implementierungsarbeiten zeigen, ob ein LonTalk-Router auf Basis des BMv2 realisierbar ist.

Abschließend muss erwähnt werden, dass die Durchführbarkeit zum aktuell vorherrschenden Wissensstand gegeben ist. Jedoch ist es bei innovativen Ideen und Herangehensweisen wahrscheinlich, dass Hindernisse und Herausforderungen entstehen, die sich üblicherweise erst in der Implementierungsphase offenbaren und zu Veränderungen in der Herangehensweise führen. Sollten Veränderungen in der Herangehensweise nötig sein, werden diese in den weiteren Abhandlungen dieser Masterarbeit diskutiert.

8 Umsetzung eines LonTalk-Routers

In diesem Kapitel werden die in dieser Arbeit umgesetzten Implementierungen vorgestellt. Dabei wird die im Verlauf dieser Arbeit angewendete Ausführungsumgebung präsentiert. Außerdem werden Details zu den genutzten Scapy-Programmen genannt. Bezüglich der P4-Implementierung des LonTalk-Routers werden genauere Einzelheiten präsentiert und anhand einiger Codeausschnitte des Programms visualisiert. Außerdem werden in diesem Kapitel noch die Arten der Router-Konfiguration detailliert beschrieben. Abschließend wird neben der Testung des LonTalk-Routers auch die Inbetriebnahme der genutzten virtuellen Maschine ausführlich beschrieben.

8.1 Details der Ausführungsumgebung

In diesem Kapitel wird die final zur Anwendung gekommene Ausführungsumgebung vorgestellt. Diese basiert auf dem in Kapitel 7.3 erläuterten Konzept eines manuellen Setups. Dieses Kapitel hat zum Ziel, alle Befehle der Ausführungsumgebung, die für den Start der Topologie, in welcher der LonTalk-Router zur Anwendung kommt, nötig sind, darzustellen und deren Auswirkung zu erklären.

Die realisierte Ausführungsumgebung baut innerhalb der verwendeten Entwicklungsplattform immer auf einer vordefinierten Ordnerstruktur auf. Diese muss, damit das manuelle Setup erfolgreich ausgeführt werden kann, in Form der folgenden Dateiordner vorhanden sein:

- **build:** Der Ordner `build` enthält alle P4-Dateien, die für die Realisierung des LonTalk-Routers benötigt werden.
- **logs:** In diesem Ordner werden die durch die Konfiguration und durch den BMv2-Switch erstellten Log-Dateien gespeichert.
- **python:** Der Ordner `python` enthält Python-Programme, die für das Versenden und Empfangen von LonTalk-Nachrichten auf den Hosts benötigt werden. Außerdem ist in diesem Ordner auch noch das Controller-Programm abgelegt, das für die Konfiguration des Routers über die P4Runtime verantwortlich ist.
- **switch_config:** Dieser Ordner enthält die für die Konfiguration benötigten Dateien, welche die Konfigurationsanweisungen enthalten.

Der Ordner, welcher diese vier Unterordner und das Bash-Skript, in dem die Ausführungsumgebung definiert wurde, enthält, wird im weiteren Verlauf dieser Arbeit mit dem Titel „Projektordner“ bezeichnet.

Um das besagte Bash-Skript zu starten, muss aus dem Projektordner heraus lediglich der Befehl `sudo ./run_demo.sh` zusammen mit zwei notwendigen Argumenten ausgeführt werden. Das erste benötigte Argument gibt den Namen des P4-Programms an, auf dem der Router basieren sollte. Beispielsweise muss der Inhalt dieses Arguments `lontalk` sein, damit der Router auf der Datei `lontalk.p4` basierend ausgeführt wird. Der Name der P4-Datei muss für diese Ausführungsumgebung auch mit den Namen der Konfigurationsdateien übereinstimmen. Dies würde für das zuvor erwähnte Beispiel bedeuten, dass die Konfigurationsdateien `lontalk.json` beziehungsweise `lontalk.txt` heißen müssten. Auch muss ein Teil der Inhalte der JSON-Konfigurationsdatei auf das erste Argument angepasst sein. Dies betrifft jedoch lediglich den Namen der Dateien, welche die „P4 Device Config“ und P4Info (siehe Kapitel 4.8) repräsentieren. Das zweite benötigte Argument gibt an, welcher Konfigurationsmodus verwendet werden soll. Hier sind lediglich die Optionen `p4runtime` und `cli` erlaubt. Der Inhalt des ersten Arguments wird innerhalb des Bash-Skripts mittels `$1` aufgerufen. Für den Inhalt des zweiten Argument wird `$2` benutzt. Sind die beiden benötigten Argumente gesetzt, werden die Inhalte des Bash-Skriptes abgehandelt.

Das Skript beginnt damit, dass alle Prozesse, Konsolenfenster und Namespaces, die in einer vormaligen Instanz der Ausführungsumgebung nicht korrekt gelöscht wurden, entfernt werden, sodass diese neu erstellt werden können.

```
1 sudo ip netns add h11
2 sudo ip netns add h12
```

Listing 11: Erstellung der Network Namespaces für die Endgeräte H11 und H12

Daraufhin werden für Hosts benötigte Network Namespaces mittels des Packages `iproute2` erstellt. Dies ist notwendig, da Network Namespaces es ermöglichen, einen neuen Network Stack zu definieren. Dies bedeutet, dass ein Network Namespace eigene Interfaces und Routing-Tabellen besitzen kann, die vom globalen Network Stack des darunterliegenden Betriebssystems abgetrennt sind. Diese Isolierung ist maßgeblich für die Realisierung der einzelnen Hosts, die in der benutzten Topologie enthalten sind. Die Befehle, die für die Erstellung der dementsprechenden Network Namespaces genutzt werden, sind in Listing 11 visualisiert. Darin werden jedoch lediglich die Network Namespaces für die Endgeräte H11 und H12 erstellt. Weitere Informationen zu Namespaces und im besonderen zu Network Namespaces finden sich in [98] und [99].

```
1 sudo ip netns exec h11 ifconfig lo up
2 sudo ip netns exec h12 ifconfig lo up
```

Listing 12: Aktivierung der Loopback-Interfaces für die Endgeräte H11 und H12

Darauffolgend muss für jeden jeden der erstellten Hosts das Loopback-Interface aktiviert werden. Dieser Vorgang ist in Listing 12 für die Endgeräte H11 und H12 dargestellt. Die

Aktivierung dieses Interfaces ist für die Ausführung der auf Scapy basierenden Sende- und Empfangsprogramme nötig. Wird dieses Interface nicht bereitgestellt, ist die Ausführung dieser Programme nicht möglich.

```
1 sudo ip link add h11-eth0 type veth peer name s1-eth1
2 sudo ip link add h12-eth0 type veth peer name s1-eth2
```

Listing 13: Erstellung der Übertragungswege für die Endgeräte H11 und H12

In Listing 13 ist der nächste Befehl, der durch das Bash-Skript mithilfe des Tools `ip` ausgeführt wird, dargestellt. Dieser Befehl wird für ein jedes Endgerät ausgeführt und erstellt einen Ethernet-Übertragungsweg. Die Auswahl von Ethernet als Übertragungsweg für die zugrundeliegende Topologie wurde in Kapitel 7.6 diskutiert. Der erstellte Ethernet-Übertragungsweg besteht dabei lediglich aus zwei Ethernet-Interfaces. Aus dem angeführten Beispiel ist zu erkennen, dass jeweils das erste Interface dem involvierten Endgerät, zum Beispiel H11, und das zweite dem Software-Switch zuzuordnen ist.

```
1 sudo ip link set h11-eth0 netns h11
2 sudo ip link set h12-eth0 netns h12
```

Listing 14: Zuweisung der erstellten Interfaces zu den Endgeräten H11 und H12

Die Zuordnung des in Listing 13 erstellten Interfaces zu dem korrespondierenden Network Namespace beziehungsweise Endgerät erfolgt jedoch erst in Listing 14. Diese Zuweisung ist für die Interfaces des Software-Switches nicht notwendig. Dies bedeutet, dass die betroffenen Interfaces des Switches im globalen Namespace des Betriebssystems verbleiben können. Das stellt laut [91] jedoch kein Problem dar, da der verwendete Software-Switch keine eigenen, isolierten Network Stack benötigt, weil dieser lediglich Interfaces für die Weiterleitung von Paketen benötigt.

```
1 declare -a arr=("h11-eth0" "h12-eth0" "h13-eth0" "h14-eth0" "h21-eth0" "
    h22-eth0" "s1-eth1" "s1-eth2" "s1-eth3" "s1-eth4" "s1-eth5" "s1-eth6"
")
2 for intf in "${arr[@]}"
3 do
4     sysctl net.ipv6.conf.${intf}.disable_ipv6=1
5 done
```

Listing 15: Deaktivierung von IPv6 auf den erstellten Interfaces

In einem nächsten Schritt wird auf allen definierten Interfaces das Internet Protocol Version 6 (IPv6) deaktiviert. Das ist vorteilhaft, da der IPv6-Stack eines Interfaces andernfalls automatisch Nachrichten versendet, wie zum Beispiel IPv6 Multicast Domain Name System-Nachrichten (DNS). Diese Nachrichten werden daraufhin durch den verwendeten Software-Switch detektiert und verarbeitet, sodass diese nunmehr auch in der Log-Datei ersichtlich werden. Die angesprochenen Nachrichten haben dadurch einen negativen Effekt

auf die Übersichtlichkeit der Log-Datei und erschweren die Fehlersuche in der betroffenen Datei. Daher wurde entschieden, wie in Listing 15 dargestellt, IPv6 auf den erstellten Interfaces zu deaktivieren.

```

1 sudo ip netns exec h11 ifconfig h11-eth0 hw ether 00:00:0a:00:00:01
  10.0.1.1/24 up
2 sudo ip link set dev s1-eth1 up
3 sudo ip netns exec h12 ifconfig h12-eth0 hw ether 00:00:0a:00:00:02
  10.0.1.2/24 up
4 sudo ip link set dev s1-eth2 up

```

Listing 16: Aktivierung der erstellten Netzwerkinterfaces

Im einem nächsten Schritt werden die erstellten Netzwerkinterfaces aktiviert. Dies ist in Listing 16 beispielhaft für die Übertragungsstrecke der Endgeräte H11 und H12 dargestellt. Darin ist auch der Aktivierungsbefehl visualisiert, der die Endgeräte betrifft und in dem jeweiligen Network Namespace, unter Angabe einer IP- und einer Media Access Control-Adresse (MAC), auszuführen ist. Diese angegebenen Adressen werden dabei lediglich benötigt, um das betreffende Interface zu aktivieren und werden in dieser Arbeit nicht weiter verwendet. Die Aktivierung der Router- Interfaces ist hingegen ohne diese Adressen möglich und wird über das Tool `ip link` abgewickelt.

```

1 sudo p4c --p4runtime-files build/$1.p4.p4info.txt --target bmv2 --arch
  v1model --std p4-16 build/$1.p4 -o ./build

```

Listing 17: Kompilierung des P4-Programms

Der folgende Schritt betrifft bereits das P4-Programm selbst. In Listing 17 ist der Befehl abgebildet, der dazu genutzt wird, das P4-Programm zu kompilieren, um dadurch alle für den Betrieb des LonTalk-Routers benötigten Dateien zu erhalten. In diesem Listing ist zu erkennen, dass ein Argument gesetzt wurde, das zur Erstellung der P4Info-Datei dient. Außerdem ist in dem Befehl angegeben, für welche Plattform und Architektur das Programm kompiliert werden soll. Dies sind in diesem Fall der BMv2-Switch und das V1-Modell. Darüber hinaus wird noch die Verwendung der aktuellen Sprachversion (P4₁₆), das zu kompilierende P4-Programm und ein Ausgabeordner, der das Ergebnis der Kompilierung enthält, angegeben. Möchte sich ein/e Nutzer*in einen Überblick über alle weiteren, möglichen Argumente des Referenzcompilers verschaffen, so kann dazu der Befehl `p4c --help` genutzt werden.

```

1 sudo simple_switch -i 1@s1-eth1 -i 2@s1-eth2 -i 3@s1-eth3 -i 4@s1-eth4 -
  i 5@s1-eth5 -i 6@s1-eth6 --thrift-port 9090 --log-console >logs/
  switch.log --device-id 0 build/$1.json &
2 .
3 .
4 .
5 sudo simple_switch_grpc -i 1@s1-eth1 -i 2@s1-eth2 -i 3@s1-eth3 -i 4@s1-
  eth4 -i 5@s1-eth5 -i 6@s1-eth6 --log-console >logs/switch.log --
  device-id 0 --no-p4 -- --grpc-server-addr localhost:50051 &

```

Listing 18: Starten des Software-Switches

Auf die Kompilierung des P4-Programms folgt der Start des Software-Switches. Die dazu genutzten Befehle sind in Listing 18 veranschaulicht. Der erste, in Zeile eins, abgebildete Startbefehl wird ausgeführt, wenn die Konfigurationsart `cli` ausgewählt ist, während der zweite Befehl, der in der letzten Zeile des Listings abgebildet ist, für die Konfigurationsart `p4runtime` genutzt wird. Die beiden Befehle enthalten einige Gemeinsamkeiten, die nun beschrieben werden. Dabei sticht vor allem der Parameter `-i` heraus, der eine Liste der durch den Switch zu nutzenden Netzwerkinterfaces angibt. Dabei wird beispielsweise der Switch-Port eins mit dem Ethernet-Interface `s1_eth1` assoziiert. Eine weitere Gemeinsamkeit ist die Nutzung von Logging-Meldungen, die über den Parameter `--log-console` aktiviert und in einer dedizierten Textdatei gespeichert werden können. Der letzte Parameter, der beiden Befehlen gemein ist, ist `--device-id`. Dieser und die dabei vergebene Geräte ID dienen dazu, um mehrere Software-Switches unterscheiden zu können. Neben den vorgestellten Gemeinsamkeiten gibt es außerdem einige Unterscheidungen zwischen beiden Befehlen:

- Konfigurationsart `cli`: Um den Switch nach seinem Start auch konfigurieren zu können, wird in Listing 18, mittels des Parameters `--thrift-port`, ein Thrift-Port definiert, über den die Konfiguration abgewickelt werden kann. Am Ende des Befehls ist außerdem noch ersichtlich, dass das nach der Kompilierung nunmehr im JSON-Format vorliegende P4-Programm angegeben ist.
- Konfigurationsart `p4runtime`: Neben den allgemeinen Startparametern gibt es auch für die Konfigurationsart `p4runtime` einige Ergänzungen. Eine auffällige Änderung ist dass der Befehl `simple_switch_grpc` anstatt des Befehls `simple_switch` genutzt wird. Dies liegt darin begründet, dass `simple_switch_grpc` die Vergabe einer gRPC-Server-Adresse erlaubt. Diese Adresse, ein Tupel, das aus einer IP-Adresse und einem TCP-Port besteht, ermöglicht, dass der erstellte Server durch einen Controller angeprochen werden kann. Der Controller kann, nach dem Start des Software-Switches, dazu genutzt werden, das P4-Programm auf diesem zu installieren. Diese Möglichkeit zeigt sich durch den Parameter `--no-p4`, der es ermöglicht, den Software-Switch ohne ein initiales P4-Programm zu starten.

Weitere Startoptionen der Programme `simple_switch` und `simple_switch_grpc` sind durch Zuhilfenahme des Parameters `--help` zu finden.

```

1 simple_switch_CLI --thrift-port 9090 < switch_config/$1.txt
2 .
3 .
4 .
5 sudo ./python/mycontroller.py --table_entries "./switch_config/$1.json"

```

Listing 19: Konfigurationsoptionen des Software-Switches

In Listing 19 sind nunmehr die Befehle dargestellt, die für die Konfiguration des Software-Switches genutzt werden. Der erste in diesem Listing dargestellte Befehl wird ausgeführt, wenn die Konfigurationsoption `cli` gewählt wurde. In diesem Befehl wird das Programm `simple_switch_CLI` genutzt, das den Software-Switch über bereits zuvor, in Listing 18, definierten Thrift-Port konfiguriert. Die zu übermittelnde Konfiguration ist in der angegebenen Textdatei enthalten. Der zweite, in der letzten Zeile des Listings 19 angeführte, Befehl dient dazu, bei gesetzter Konfigurationsoption `p4runtime` einen Python-Programm zu starten. Dieses Programm implementiert einen simplen Controller, der dazu genutzt wird, eine Konfiguration des Software-Switches nach den Angaben in einer JSON-Datei vorzunehmen. Der Pfad dieser Datei wird dem Python-Programm über den Parameter `--table_entries` übergeben. Auf die Inhalte dieser JSON-Datei wird in Kapitel 8.5 noch im Detail eingegangen.

```
1 xterm -xrm 'XTerm.vt100.allowTitleOps: false' -T H11 -e ip netns exec
   h11 bash &
2 xterm -xrm 'XTerm.vt100.allowTitleOps: false' -T H12 -e ip netns exec
   h12 bash &
```

Listing 20: Start von Eingabefenstern für die Endgeräte H11 und H12

Nachdem der Software-Switch gestartet ist, wird für einen jeden Host der verwendeten Topologie je ein Konsolenfenster gestartet. Diese `xterm`-Eingabefenster ermöglichen das Ausführen von einzelnen Befehlen und das Starten der in Kapitel 8.2 beschriebenen Scapy-Programme. In Listing 20 sind die Befehle, die für den Start der Konsolenfenster der Endgeräte H11 und H12 benötigt werden, angegeben.

Um gesamte Ausführungsumgebung wieder zu schließen beziehungsweise zu beenden, muss durch den/die Nutzer*in in dem Konsolenfenster, das dazu genutzt wurde, das Bash-Skript zu starten, lediglich eine beliebige Taste gedrückt werden. Ein getätigter Tastendruck bewirkt, dass abermals ein Cleanup-Prozess in Gang gesetzt wird, der alle laufenden Prozesse des Software-Switches, alle Network Namespaces der gesamten Topologie, alle Konsolenfenster, die diesen Namespaces zugerechnet werden können, beendet werden. Außerdem werden innerhalb dieses Cleanup-Prozesses alle Log-Dateien und alle Dateien, die durch die Kompilierung erstellt wurden, gelöscht.

8.2 Details der Scapy-Programme

In Kapitel 8.1 ist beschrieben, dass durch die Ausführungsumgebung pro Host ein Konsolenfenster gestartet wird. Das Konsolenfenster kann daraufhin genutzt werden, um LonTalk-Nachrichten mit dem Programm `send.py` zu versenden. Das Empfangen von LonTalk-Nachrichten wird, innerhalb dieser Konsolenfenster, durch das Programm

`receive.py` ermöglicht. Eine Verwendung beider Programme ermöglicht daher eine Kommunikation zwischen zwei Host-Endgeräten. Die Eigenschaften von Scapy, die die Grundlage der Programme, die in diesem Kapitel vorgestellt werden, bilden, wurden bereits eingehend in Kapitel 7.5 erläutert.

8.2.1 Scapy-Programm `send.py`

Das in diesem Kapitel vorgestellte Programm dient dem Versenden von LonTalk-Nachrichten auf einem Host-Endgerät und ist Teil des Projektunterordners `python`.

Um dieses Programm aus einem in der Ausführungsumgebung erstellten Konsolenfenster zu starten, muss der Befehl `sudo ./python/send.py` aufgerufen werden. Dieses Programm kann allerdings auch mit einigen optionalen Parametern gestartet werden:

- **--interface:** Der Parameter `--interface` muss im Programm nicht gesetzt werden, da der erwartete Inhalt des Parameters durch die am Host vorhandenen Interfaces automatisch ermittelt wird. Nichtsdestotrotz ist es durch diesen Parameter möglich, das Programm auch mit Ethernet-Interfaces, die nicht in der Topologie enthalten sind, zu nutzen.
- **--type:** Das Argument des Parameters `--type` gibt das Adressformat der zu versendenden Nachricht an. Die verfügbaren Adressformate wurden bereits in Kapitel 5.2.2 genauer beschrieben. Die Werte, die dieser Parameter für die verfügbaren Adressformate entgegennehmen kann, entsprechen den in Tabelle 4 angeführten Werten.
- **--domainlength:** In Kapitel 5.2.2 wurde bereits erwähnt, dass es bei LonTalk eine Abstufung hinsichtlich der Länge der in einer LonTalk-Nachricht enthaltenen Domäneninformation gibt. Tabelle 5 bietet eine Übersicht über die für diesen Parameter möglichen Werte.
- **--payload:** Mit dem Parameter `--payload` kann der Inhalt, der in einer Nachricht enthalten sein soll, bestimmt werden. Wird dieser Parameter nicht gesetzt, enthält die versendete Nachricht neben einer Sequenz von Buchstaben auch eine Zufallszahl, die nützlich sein kann, um die versendete Nachricht am Empfänger von anderen empfangenen Nachrichten zu unterscheiden.
- **--priority:** Der Parameter `--priority` dient dazu, bestimmen zu können, welchen Wert dem Priority-Feld einer versendeten Nachricht zugewiesen wird. Die Werte, die diesem Feld zugewiesen werden können, sind eins und null. Wird das Programm ohne den Parameter `--priority` gestartet, wird das Feld auf den Wert null gesetzt, das hat

zur Folge, dass das Paket durch den LonTalk-Router keine priorisierte Behandlung erfährt.

```

1 class Ethertunnel(Packet):
2     name = "Ethertunnel"
3     fields_desc = [DestMACField("dst"),
4                     SourceMACField("src"),
5                     XShortField("type", 34848)]
6
7 class LonTalkL2(Packet):
8     name = "LonTalk_L2"
9     fields_desc = [BitField("Pri", 0, 1),
10                  BitField("Path", 0, 1),
11                  BitField("Backlog", 0, 6)]
12     def guess_payload_class(self, payload):
13         return LonTalkL3
14
15 class LonTalkL3(Packet):
16     name = "LonTalk_L3"
17     fields_desc = [BitField("Version", 0, 2),
18                    BitField("PacketFormat", 0, 2),
19                    BitField("AddressFormat", 3, 2),
20                    BitField("Length", 0, 2)]
21     def guess_payload_class(self, payload):
22         return SourceAddress

```

Listing 21: Definition einiger Header entnommen aus der Datei `send.py`

Ein Kernaspekt von Scapy sind Klassen, die auf der Basisklasse `Packet` fußen, denn durch diese Klassen kann das Format eines Pakets beziehungsweise einer PDU definiert werden. Beispiele hierzu finden sich in Listing 21. Darin sind die drei Klassen `Ethertunnel`, `LonTalkL2` und `LonTalkL3` enthalten, die Teile des Headers von Ethernet und LonTalk beschreiben. Die abgebildete Klasse `Ethertunnel` wird benötigt, weil mit Scapy erstellte Pakete entweder über Ethernet- oder über IP versandt beziehungsweise empfangen werden können. Ethernet hat diesbezüglich den Vorteil, dass auf dem Software-Switch weniger Header definiert werden müssen, die nicht LonTalk zuzurechnen sind, als dies bei der Verwendung von IP der Fall wäre, sodass sich das resultierende P4-Programm dadurch übersichtlicher gestaltet. Aus diesem Grund basiert das Versenden von LonTalk-Nachrichten auf Ethernet. Die in Listing 21 abgebildete Klasse `Ethertunnel` besteht aus jeweils einer Source- und einer Destination-MAC-Adresse. Die Adresse, welche für jede versendete LonTalk-Nachricht im Feld der Destination-MAC-Adresse vergeben wird, ist `FF:FF:FF:FF:FF:FF` und entspricht daher der Broadcast-MAC-Adresse. Das abschließende Feld dieser Klasse bildet `type` und repräsentiert den EtherType eines Ethernet-Headers. Da der Wert dieses Feldes den im Ethernet-Frame gekapselten Inhalt anzeigt, wird für alle in dieser Arbeit vorgenommenen Implementierungen das LonTalk-Protokoll mit dem EtherType 34848 assoziiert. Die `LonTalkL2` und `LonTalkL3`, die zusätzlich in Listing 21 dargestellt sind, implementieren Teile des LonTalk-Headers der OSI-Schichten zwei und drei. Um nach dem Versenden einer LonTalk-Nachricht den Inhalt der in dieser Nachricht enthaltenen Header korrekt darzustellen, benötigt Scapy eine „Verknüpfung“ der betroffenen Header-Klassen. Eine solche „Verknüpfung“ kann durch die Verwendung der Klassen-

methode `guess_payload_class` erreicht werden. Dazu muss lediglich der Rückgabewert dieser Klassenmethode dem Klassennamen der auf diese Header-Klasse folgenden Klasse entsprechen. In Listing 21 ist beispielsweise in der Klassenmethode der Klasse `LonTalkL2` definiert, dass die darauffolgende Klasse `LonTalkL3` sein soll. Hierbei muss allerdings beachtet werden, dass die Verwendung dieser Methode für die Klasse `Ethertunnel` nicht möglich ist, da deren Header innerhalb einer versendeten LonTalk-Nachricht immer an der Spitze des Header-Stacks steht. Um dennoch eine „Verknüpfung“ zwischen dieser Klasse und der darauffolgenden Klassen `LonTalkL2` herstellen zu können, wird auf eine andere Herangehensweise, welche auf der Methode `bind_layers()` basiert, zurückgegriffen. Diese beiden Herangehensweise werden genutzt, um alle Header-Klassen dieses Programms zu verbinden.

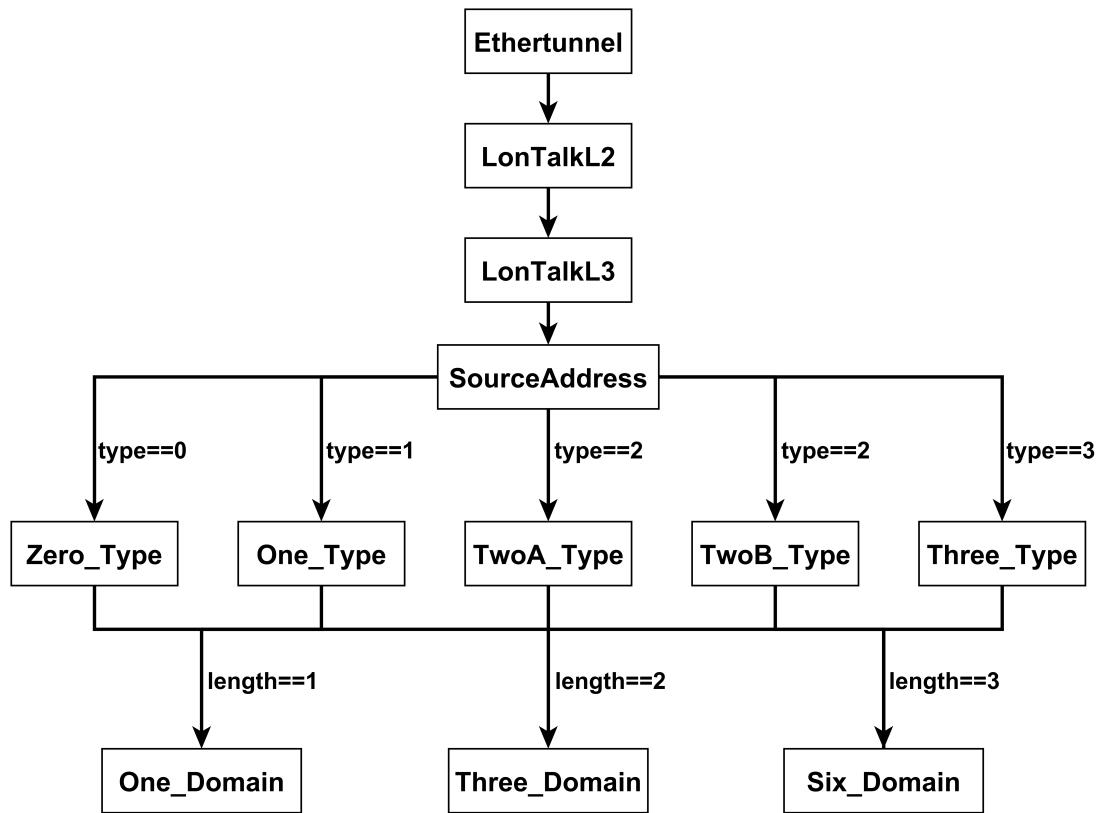


Abbildung 25: Übersicht über die Header-Klassen des Nachrichtenerstellungs-Programms

Die Verknüpfungen zwischen den, in diesem Programm definierten, Header-Klassen werden in Abbildung 25 dargestellt. Die Abbildung enthält neben den Verknüpfungen zwischen den Klassen selbst auch noch Bedingungen, aufgrund derer die jeweilige Klassenmethode (`guess_payload_class`) entscheidet, mit welcher Header-Klasse eine Verknüpfung erstellt werden soll. Um diese Abbildung besser interpretieren zu können, wird im Folgenden ein Beispiel daraus vorgestellt: Wenn das Feld `type`, das den Adressstyp einer LonTalk-Nachricht angibt, den Wert null enthält, die Header-Klasse `SourceAddress` mit der Header-Klasse `Zero_Type` verknüpft wird. Aus Abbildung 25 kann auch ge-

schlussfolgert werden, dass die Header-Klassen `EtherTunnel`, `LonTalkL2`, `LonTalkL3` und `SourceAddress` bei der Erstellung eines jeden Pakets, das mit diesem Programm erstellt wird, zur Anwendung kommen. Die Klasse `SourceAddress` kann dabei immerzu angewendet werden, weil sie Header-Felder, wie beispielsweise `SourceSubnet`, definiert, die in einem jeden Adressstypen beziehungsweise in einer jeden Adressierungsart enthalten sind.

```

1 if length == 0:
2     pktdomain=""
3 elif length ==1:
4     domain = int(raw_input("Domain number [0-255]:\n>>>"))
5     if domain > 255 or domain < 0:
6         print(cross + "Error: domain number not in range")
7         return
8     pktdomain=One_Domain(Domain=domain)
9 elif length ==2:
10    domain = int(raw_input("Domain number [0-65535]:\n>>>"))
11    if domain >65535 or domain<0:
12        print(cross + "Error: domain number not in range")
13        return
14    pktdomain=Three_Domain(Domain=domain)
15 elif length == 3:
16    domain = int(raw_input("Domain number [0-281474976710655]:\n>>>"))
17    if domain >281474976710655 or domain<0:
18        print(cross + "Error: domain number not in range")
19        return
20    pktdomain = Six_Domain(Domain=domain)

```

Listing 22: Erstellung des domänenspezifischen Paketbestandteils in `send.py`

Neben diesen immer vorkommenden und damit „fixen“ Paketbestandteilen gibt es noch andere Klassen, deren Verwendung nicht statisch vorgegeben ist. Dazu zählen die Klassen, die benötigt werden, um restliche Adress- und Domäneninformationen in das Paket zu integrieren. Die Erstellung des Paketteils, der die Domäneninformation enthält (siehe Variable `pktdomain`), ist in Listing 22 ersichtlich. Dabei ist anzumerken, dass sobald die Programmvariable `length`, die die Länge der Domäneninformation angibt, null ist, keine domänenspezifische Header-Klasse benötigt wird, weil dies bedeutet, dass das Paket keine diesbezügliche Headerinformation enthalten soll.

```

1 if type == 0:
2     print("Destination-subnet needed!")
3     destinationsubnet = int(raw_input("Destination-subnet number
4         [0-255]:\n>>>"))
5     if destinationsubnet >255 or destinationsubnet<0:
6         print(cross + "Error: subnet number not in range")
7         return
8     print(checkmark + 'All required arguments available!')
9     pkttpe = SourceAddress(SourceSubnet=sourcesubnet, Bit=1, SourceNode
10      =sourcenode) / \
11          Zero_Type(DestSubnet=destinationsubnet)
12 elif type == 1:
13     print("Destination-group needed!")
14     group = int(raw_input("Group number [0-255]:\n>>>"))
15     if group > 255 or group < 0:
16         print(cross + "Error: group number not in range")
17         return
18     print(checkmark + 'All required arguments available!')

```

```

17     pkttype = SourceAddress(SourceSubnet=sourcesubnet, Bit=1, SourceNode
18         =sourcenode) / \
                      One_Type(DestGroup=group)

```

Listing 23: Erstellung von Headern der Adressformate „Subnet Broadcast“ und „Group“

Die größte Variabilität des Programms entsteht allerdings durch die Nutzung der verfügbaren LonTalk-Adressformate. In Listing 23 ist daher der Python-Code für die Erstellung der Adressformate null und eins beziehungsweise „Subnet Broadcast“ und „Group“ dargestellt. Der somit erstellte Paketeil behandelt alle verfügbaren Adressformate und wird für den weiteren Programmverlauf in der Variable `pkttype` gespeichert.

```

1     pkt = pktheaders / pkttype / pktdomain / pktpayload
2     sendp(pkt, iface=iface, verbose=False)
3     pkt.show2()

```

Listing 24: Versenden einer LonTalk-Nachricht mittels `send.py`

In der ersten Zeile des Listings 24 ist zu erkennen, dass eine zu versendende LonTalk-Nachricht durch Zusammensetzen von bereits erstellten Programmteilen erstellt werden kann. Dazu werden die „fixen“ Bestandteile des Paketheaders, die in der Variable `pktheaders` gespeichert sind, durch das Zeichen `/` mit dem darauffolgenden Paketeil verbunden. Auf diese Weise werden auch die erstellen Paketeile `pkttype` und `pktdomain`, die in den Listings 22 und 23 erstellt wurden, verbunden. Hinzukommend kann durch diese Vorgangsweise auch der eigentliche Inhalt des Pakets, repräsentiert durch die Variable `pktpayload`, an die konstruierte Nachricht angefügt werden. Die so erstellte LonTalk-Nachricht wird daraufhin durch die Methode `sendp()` über ein spezifiziertes Interface versendet und dem/der Nutzer*in aufgrund des letzten, in Listing 24 abgebildeten, Befehls in dem entsprechenden Konsolenfenster angezeigt.

8.2.2 Scapy-Programm `receive.py`

Das in diesem Kapitel vorgestellte Programm dient dem Empfang von LonTalk-Nachrichten, die durch das Programm `send.py` auf einem Host-Endgerät erstellt wurden, und ist Teil des Projektunterordners `python`. Für dieses Programm muss außerdem angemerkt werden, dass es starke Parallelen zu dem in Kapitel 8.2.1 vorgestellten Programm aufweist.

Um dieses Programm zu starten, wird der Befehl `sudo ./python/receive.py` benötigt. Im Gegensatz zu dem Programm, das für die Erstellung von LonTalk-Nachricht genutzt wird, kann in diesem Programm lediglich der Parameter `--interface` angepasst werden. Die Angabe eines Interfaces ist auch in dieser Programmroutine optional und wird, wenn der Parameter nicht gesetzt wurde, automatisch durch das Programm bestimmt.

```

1 class LonTalkL3(Packet):
2     name = "LonTalk_L3"
3     fields_desc = [BitField("Version", 0, 2),
4                     BitField("PacketFormat", 0, 2),
5                     BitField("AddressFormat", 3, 2),
6                     BitField("Length", 0, 2)]
7     def guess_payload_class(self, payload):
8         global type
9         global length
10        type=self.AddressFormat
11        length=self.Length
12        return SourceAddress
13    .
14    .
15    .
16 sniff(iface = iface, prn = handle_pkt)

```

Listing 25: Ausschnitte des Programms receive.py

Der Kern des Programms bildet die Methode `sniff()`, welche in Listing 25 dargestellt ist und auf einem definierten Interface alle eingehenden Ethernet-Frames erfasst. Innerhalb der Methode wird außerdem noch ein Callback auf die Methode mit dem Namen `handle_pkt` definiert, die für einen jeden neu eingehenden Frame ausgeführt wird. Die Methode `handle_pkt` hat in diesem Programm lediglich die Funktion, die eingehenden LonTalk-Nachrichten zu visualisieren, damit der/die Nutzer*in den Erhalt einer versendeten Nachricht bestätigen und den Inhalt auf Korrektheit überprüfen kann. Zu diesem Zweck wird bei einer eingehenden Nachricht lediglich mithilfe einer Scapy-Methode überprüft, ob die in dieser Datei definierten Header-Klassen in der zu bearbeitenden Nachricht enthalten sind. Ist dies der Fall, werden die dementsprechenden enthaltenen Teile der jeweiligen Header-Klasse in einem Konsolenfenster visualisiert. Die angesprochenen Header-Klassen dieser Datei, die mithilfe der Basisklasse `Packet` erstellt wurden, beziehungsweise die Headerstruktur, die in diesen Klassen definiert ist, unterscheiden sich von den Klassen des Programms, das für die Versendung der LonTalk-Nachrichten genutzt wird, nur geringfügig. Beispielsweise muss für das Empfangen einer Nachricht kein Ethernet-Header mehr definiert werden, da dieser automatisch aufgrund der Methode `sniff()` detektiert wird und in Scapy daher bereits enthalten ist. Die Definition einer eigenen Klasse, die mit der in Kapitel 8.2.1 vorgestellten Klasse `Ethertunnel` vergleichbar ist, ist auf Seiten des Empfängers nicht sinnvoll, da trotz umfangreicher Recherhetätigkeit keine Möglichkeit gefunden wurde, das auf diese Weise definierte Headerformat anstelle des automatisch von Scapy verwendeten Ethernet-Headers zu detektieren. Ein weiterer Unterschied, der auch in Listing 25 dargestellt ist, betrifft die Klasse `LonTalkL3`. In der Klasse wird innerhalb der Scapy-Methode `guess_payload_class` das Adressformat und die Länge der Domäneninformation ausgelesen und in die globalen Variablen `type` beziehungsweise `length` gespeichert. Diese werden im weiteren Programmverlauf dazu benötigt, um die korrekten Klassen der Headerbestandteile, die das Adressformat beziehungsweise die Domäneninformation enthalten, zu visualisieren. Eine Parallele, die zwischen dem Versende- und Emp-

fangsprogramm besteht, ist die Nutzung der Methode `bind_layers()`, die auch in diesem Programm benötigt wird, um die Scapy-Klasse des Ethernet-Headers mit der darauffolgenden, selbst definierten Klasse (`LonTalkL2`) zu verknüpfen.

8.3 Designentscheidungen bezüglich der P4-Implementierung

In Kapitel 6.2 wurde bereits die Entscheidung begründet, welche Bestandteile eines Configured Routers implementiert werden sollen. Vor der P4-Implementierung der betroffenen Router-Bestandteile sind noch einige Designentscheidungen und Vорbedingungen, denen die Implementierung unterliegt, zu treffen, die in diesem Kapitel enthalten sind.

Die These, dass die grundlegenden Forwarding-Entscheidungen eines Configured Routers auf Schicht drei getroffen werden, wurde in Kapitel 6.2 aufgestellt und bildet die Grundlage der in dieser Arbeit vorgenommenen Implementierung. Das Ziel der Implementierung ist, dass der zu implementierende LonTalk-Router mit der in Kapitel 7.7 vorgestellten Topologie und der zugrundeliegenden Übertragungstechnologie funktioniert. Eine Abstimmung des P4-Programms auf die verwendete Übertragungstechnologie ist nötig, da ein mit dieser Übertragungstechnologie einhergehender Header Teil des Programms sein muss. In Kapitel 8.2 wurde die Entscheidung getroffen, dass die Programme, die für das Empfangen und Versenden von LonTalk-Nachrichten zuständig sind, einen Ethernet-Tunnel für die Übermittlung der LonTalk-Pakete nutzen sollen. Ein Ethernet-Tunnel bietet gegenüber einem Tunnel auf der Basis von IP den Vorteil, dass innerhalb des Parsers des P4-Programms nur ein Ethernet-Header definiert werden muss und daher der IP-Header nicht zu implementieren ist. Um innerhalb dieses P4-Programms die Funktionalität des Pri-Feldes, welches sich auf Schicht zwei des LonTalk-Protokolls befindet, zu unterstützen, muss vorab der Priorisierungsmechanismus des `simple_switch_grpc` aktiviert werden. Für die Aktivierung des Mechanismus müssen auf der Entwicklungsplattform mehrere Befehle ausgeführt werden. Diese Befehle basieren auf der in Kapitel 8.7 erstellten virtuellen Maschine. Dazu müssen die im Ordner `behavioral-model` befindlichen Dateien `simple_switch.h` und `simple_switch.cpp` modifiziert werden.

```
1 // #define SSWITCH_PRIORITY_QUEUEING_ON
```

Listing 26: Präprozessor-Direktive deren Kommentar zu entfernen ist

In der zuerst genannten Datei ist bei der in Listing 26 angeführten Präprozessor-Direktive der vorangestellte Kommentar zu entfernen.

```

1 #ifdef SSWITCH_PRIORITY_QUEUEING_ON
2     bm::Logger::get()->info("Using priority queueing!");
3 #endif

```

Listing 27: Für Aktivieren des Priorisierungsmechanismus nötige Programmzeilen

In der erwähnten C++-Datei sind in der Funktion `start_and_return_()` die in Listing 27 abgebildeten Zeilen zu ergänzen. Diese Ergänzung bewirkt, dass durch die in der Log-Datei enthaltenen Worte „Using priority queueing!“ angezeigt wird, dass der Priorisierungsmechanismus aktiv ist. Daraufhin ist es jeweils noch erforderlich, in den Ordnern `simple_switch` und `simple_switch_grpc`, die jeweils Unterordner des `behavioral-model`-Ordners sind, die Befehle `make` und `make install` auszuführen. Erst die Ausführung dieser Befehle bewirkt, dass der Priorisierungsmechanismus aktiviert wird und funktionstüchtig ist.

Das benutzte Architekturmodell, V1-Modell, bietet die Möglichkeit, Multicasting nativ in ein P4-Programm zu integrieren. Dazu wird das intrinsische Metadatum `mcast_grp` verwendet. Für dieses Metadatum steht unter `simple_switch_grpc` standardmäßig die Vergabe eines bis zu 16 Bit langen Wertes zur Verfügung. Laut [85] gibt es für das Metadatum `mcast_grp` die Einschränkung, dass der Wert null nicht verwendet werden kann, um eine Multicastgruppe zu identifizieren, da null bedeutet, dass für ein Paket kein Multicasting vorzunehmen ist. Die Zuweisung eines beziehungsweise mehrerer Interfaces zu einer Multicastgruppe ist in Kapitel 8.5 dargestellt.

Wie bereits in Kapitel 6.2 erwähnt, nutzt der LonTalk-Router für Forwarding-Entscheidungen die Domänen-, Gruppen- und Subnetzinformationen eines Pakets. Bei der Implementierung muss vor allem der Domänen- und der Subnetzinformation besondere Beachtung geschenkt werden, weil die Subnetzinformation betreffend zum Beispiel der Fall auftreten kann, dass das Zielsubnetz-Feld einer Nachricht den Wert null trägt. Dies führt dazu, dass die betroffene Nachricht an alle Subnetze dieser Domain weitergeleitet wird. Bei der Domäneninformation muss beispielsweise darauf geachtet werden, dass der Router keine Pakete von Hosts einer Domäne in eine andere weiterleitet. Dies ist notwendig, da laut [76] nur Pakete zwischen Hosts gleicher Domäne versendet werden dürfen. Am Router kann es daher vorteilhaft sein, solche Pakete zu verwerfen und damit nicht weiterzuleiten, um eine Belastung des Netzwerkes zu verhindern. Um diese Funktionalität sicherzustellen, wird auf dem Router eine Whitelist erstellt, die alle pro Router-Interface erlaubten Domänen auflistet. Die erstellte Whitelist kann dann dazu genutzt werden, dass der Router nur Nachrichten aus Domänen weiterleitet, die in dieser Whitelist korrekt definiert wurden.

Wie bereits in Kapitel 8.2 angemerkt, besitzt der dieser Implementierung zugrundeliegende Ethernet-Header ein Feld mit dem Titel `EtherType`, das dazu genutzt wird, das im Ethernet-Frame enthaltene Protokoll anzuzeigen. Da dem Feld `EtherType` standardmäßig für LonTalk kein Wert zugewiesen ist, wurde in Kapitel 8.2 dafür der Wert 34848 definiert.

Dieser Wert wird für jede mittels Scapy versendete Nachricht gesetzt und beim Empfang am Router überprüft.

Für die P4-Implementierung muss außerdem eine Standard-Domäne definiert werden, da es in der Topologie möglich sein sollte, Pakete sowohl mit als auch ohne Domäneninformation zu versenden. Diese Flexibilität begründet sich im length-Feld, da der hier eingetragene Wert die Länge der Domäneninformation angibt. Wird vom Router nun ein Paket ohne Domäneninformation empfangen, wird das Paket für dessen weitere Behandlung innerhalb des Routers mit einer Standard-Domäne assoziiert.

In Kapitel 6.2 wurde bereits erläutert, dass ein Router von einem Netzteilnehmer adressiert werden kann. An dieser Stelle wird darauf hingewiesen, dass in Kapitel 6.2 beschlossen wurde, an den Router adressierte Pakete durch das P4-Programm nicht weiter zu behandeln, jedoch ist es wichtig zu erkennen, wenn der Router adressiert wurde. Um auch dies in der Implementierung abzubilden, kann für den Router eine konstante Gruppennummer, eine Knotennummer und eine Unique ID vergeben werden. Die Adressierung des Routers sollten in einen zusätzlichen Parser-Zustand münden, sodass die Adressierung des Routers durch ein spezifisches Paket auch in der Log-Datei des Routers ersichtlich wird.

Eine weitere in diesem Kapitel vorgestellte Designentscheidung betrifft Debugging-Tabellen, da in Kapitel 7.4 bereits ersichtlich gemacht wurde, dass die Nutzung des Befehls `log_msg()` auf der verwendeten Entwicklungsplattform nicht möglich ist, sollten Debugging-Tabellen Teil des P4-Programms werden, um die Funktionalität dieses Befehls zu ersetzen. Debugging-Tabellen können nämlich dazu genutzt werden, bestimmte Werte beziehungsweise Inhalte eines Pakets in der Log-Datei ersichtlich zu machen. Die Nutzung von Debugging-Tabellen sollte jedoch optional sein, weil die Log-Datei des Routers dadurch unübersichtlicher werden könnte.

8.4 P4-Implementierung des LonTalk-Routers

In diesem Kapitel werden Details zur Implementierung des LonTalk-Router genannt, erklärt und einige Teile der Implementierung anhand von Codeausschnitten vorgestellt. Das vorgestellte Programm basiert dabei auf den in Kapitel 8.3 getroffenen Entscheidungen und unterliegt auch den Vorbedingungen, die in besagtem Kapitel erwähnt sind.

8.4.1 Allgemeine Inhalte der P4-Implementierung

```

1 V1Switch(
2 MyParser(),
3 MyVerifyChecksum(),
4 MyIngress(),

```

```

5 MyEgress(),
6 MyComputeChecksum(),
7 MyDeparser()
8 ) main;

```

Listing 28: Programmierbare architekturelle Bestandteile des V1-Modells

Für diese Implementierung wurde die P4-Version P4₁₆ verwendet. In Kapitel 4.6.4 wurde bereits beschrieben, dass der genutzte BMv2-Switch das V1-Modell verwendet. Dies bedeutet, dass das Programm neben dem Sprachkern, der in der Datei `core.p4` enthalten ist, auch noch die Datei `v1model.p4` einbinden muss. In Listing 28 sind die grundlegenden programmierbaren Elemente dieser Architektur ersichtlich, die in den weiteren Kapiteln genauer beschrieben werden.

```

1 typedef bit<48> macAddr_t;
2 typedef bit<8> subnetAddr_t;
3 typedef bit<7> nodeAddr_t;

```

Listing 29: Einige wiederverwendbare Datentypen der P4-Implementierung

In dem P4-Programm des Routers ist eine umfangreiche Anzahl an Headern mit dazugehörigen Feldern definiert, daher ist es vorteilhaft, einige wiederverwendbare Datentypen zu definieren. Einige dieser Datentypen sind in Listing 29 visualisiert, darunter ist zum Beispiel ein Datentyp, der für die Definition der MAC-Adressfelder verwendet wird.

Für die Realisierung der bereits angesprochenen Debugging-Tabelle wurden im gesamten Programm Präprozessordirektiven, wie beispielsweise `define` oder `ifdef` platziert, sodass die Verwendung dieser Tabelle für den/die Nutzer*in optional bleibt.

Im Programm wurden auch noch einige globale Konstanten definiert, aufgrund der im weiteren Programmverlauf Entscheidungen getroffen werden. Die Gründe für die Verwendung einiger Konstanten, darunter `ETHERTYPE_LONTALK` und `STANDARD_DOMAIN`, wurden bereits in Kapitel 8.3 behandelt. In diesem Kapitel wurde außerdem die Adressierbarkeit des Routers thematisiert. Für diesen Zweck sind im Programm die Konstanten `ADDR_ROUTER_GROUP`, `ADDR_ROUTER_NODE` und `ADDR_ROUTER_UNIQUEID` enthalten, die die Adressen des Routers repräsentieren. Darüber hinaus wurde auch noch die Konstante `DROP_PORT` definiert, deren Verwendung in Kapitel 8.4.4 erläutert wird.

8.4.2 P4-Definition der benötigten Header

In Kapitel 8.2.1 wurde die Definition der benötigten Header auf Seiten des Scapy-Programms, das zum Versenden von LonTalk-Nachrichten dient, dargelegt. In diesem Kapitel werden unter anderem die für dieses P4-Programm benötigten Headerdefinitionen vorgestellt.

```

1 header ethernettunnel_t {
2     macAddr_t dstAddr;
3     macAddr_t srcAddr;
4     bit<16> etherType;
5 }
6 header layer2_t {
7     bit<1> priority;
8     bit<1> path;
9     bit<6> deltaBacklog;
10}
11 header layer3_t {
12     bit<2> version;
13     bit<2> packetFormat;
14     bit<2> addrFormat;
15     bit<2> length;
16     subnetAddr_t sourceSubnet;
17 }
18 header address_zero_type_t {
19     bit<1> oneBit;
20     nodeAddr_t sourceNode;
21     subnetAddr_t destSubnet;
22 }
```

Listing 30: Einige Header der P4-Implementierung

Listing 30 gibt Einsicht in einen Teil der gesamten Headerdefinitionen dieses Programms. In diesem Listing ist zu erkennen, dass ein Ethernet-Header, ein Header für die Felder des LonTalk-Protokolls auf OSI-Schicht zwei und abschließend noch ein weiterer Header für die Protokollschicht drei definiert ist. Die Definition des zuletzt genannte Headers (`layer3_t`) unterscheidet sich allerdings von der Scapy-Implementierung dieses Headers dadurch, dass bei der P4-Implementierung das Feld `sourceSubnet` angefügt worden ist. Dieses Feld ist in der Scapy-Implementierung Teil der Header-Klasse `SourceAddress`, die zusätzlich noch ein Feld für ein einzelnes Bit und ein sieben Bit langes Feld für die Adresse des Quellknoten einer Nachricht enthält. Diese beiden Felder sind in der Implementierung des LonTalk-Routers nunmehr Teil der Header, die die Adressinformation des Ziels enthält. In Listing 30 ist das anhand der Headerdefinition `address_zero_type_t` ersichtlich. Diese Umstrukturierung der Felder des LonTalk-Headers ist nötig, um im P4-Parser eine einfachere Unterscheidung der Adresstypen „Subnet/Node“ beziehungsweise „Group Acknowledgement zu ermöglichen. Eine detaillierte Begründung dazu findet sich in Kapitel 8.4.3.

```

1 struct metadata {
2     subnetAddr_t destSubnet;
3     bit<48> domainField;
4     groupAddr_t destGroup;
5     bit<48> checkAddress;
6 }
7 struct headers {
8     ethernettunnel_t ethertunnel;
9     layer2_t          12;
10    layer3_t          13;
11    address_zero_type_t zero_type;
12    address_one_type_t one_type;
13    address_twoa_type_t twoa_type;
14    address_twob_type_t twob_type;
15    address_three_type_t three_type;
```

```

16     domain_one_t          domain_one;
17     domain_three_t        domain_three;
18     domain_six_t          domain_six;
19 }
```

Listing 31: **struct**-Definitionen innerhalb des P4-Programms

In Listing 31 findet sich ein Überblick über die für dieses Programm definierten Header und Metadaten-Felder. Die in der Struktur `metadata` definierten Metadaten-Felder werden verstärkt für den Control-Block `MyIngress` und den Parser benötigt. Die Definition der in Listing 31 enthaltenen Strukturen ist notwendig, um eine Verwendung der Header beziehungsweise der einzelnen Metadaten-Felder im Parser und damit im gesamten weiteren Programmverlauf zu ermöglichen.

8.4.3 P4-Parser

In Kapitel 4.9 wurde bereits die Aufgabe und Funktionsweise des Parsers dargelegt. In diesem Kapitel werden daher nur die Besonderheiten dieses Teils des P4-Programms ergänzend erwähnt. Um diesen Vorgang zu unterstützen, findet sich in Abbildung 26 ein Zustandsübergangsdiagramm, das durch das Graphs Backend [100], das Teil des offiziellen Compilers ist, erstellt wurde. In dieser Abbildung werden die Zustände in Form von Rechtecken visualisiert. Kanten, die dazu genutzt werden, um Übergänge zwischen zwei Zuständen darzustellen, sind durch Pfeile repräsentiert. Ein jeder in dieser Abbildung enthaltener Pfeil trägt eine Bezeichnung, diese beschreibt eine Bedingung die erfüllt sein muss, damit der Übergang zwischen Zuständen erfolgen kann.

In Abbildung 26 ist der Startzustand des Parsers, der laut [3] einem endlicher Automaten entspricht, dargestellt. In diesem Zustand wird überprüft, ob bei einer eingehenden Nachricht der aus dem Header-Feld `etherType` extrahierte EtherType dem festgelegten Wert `34848` entspricht. Tritt dieser Fall nicht ein und das Feld `etherType` enthält einen anderen Wert, geht der Parser implizit in den Zustand `reject` über. Eine für den Verlauf des Programms wichtige Parser-Entscheidung findet im Zustand `parse_13` statt, denn in diesem Zustand wird aufgrund des Adressformats ein mit dem jeweiligen Adressformat korrespondierender, nachfolgender Zustand gewählt. Entspricht also das Adressformat, repräsentiert durch `hdr.13.addrFormat`, dem Wert null (siehe Tabelle 4), wird in den Zustand `parse_zero_type` übergegangen. Der Parser weicht von diesem Schema lediglich bei dem Wert zwei ab, da durch diesen Wert zwei verschiedene Subadressformate angeprochen werden: „Subnet/Node“ und „Group Acknowledgement“. Diese beiden Adressformate haben eine logische Umstrukturierung der Header innerhalb des P4-Programms zur Folge, die bereits in Kapitel 8.4.2 eingangs erläutert wurde. Bisher wurden aus einem

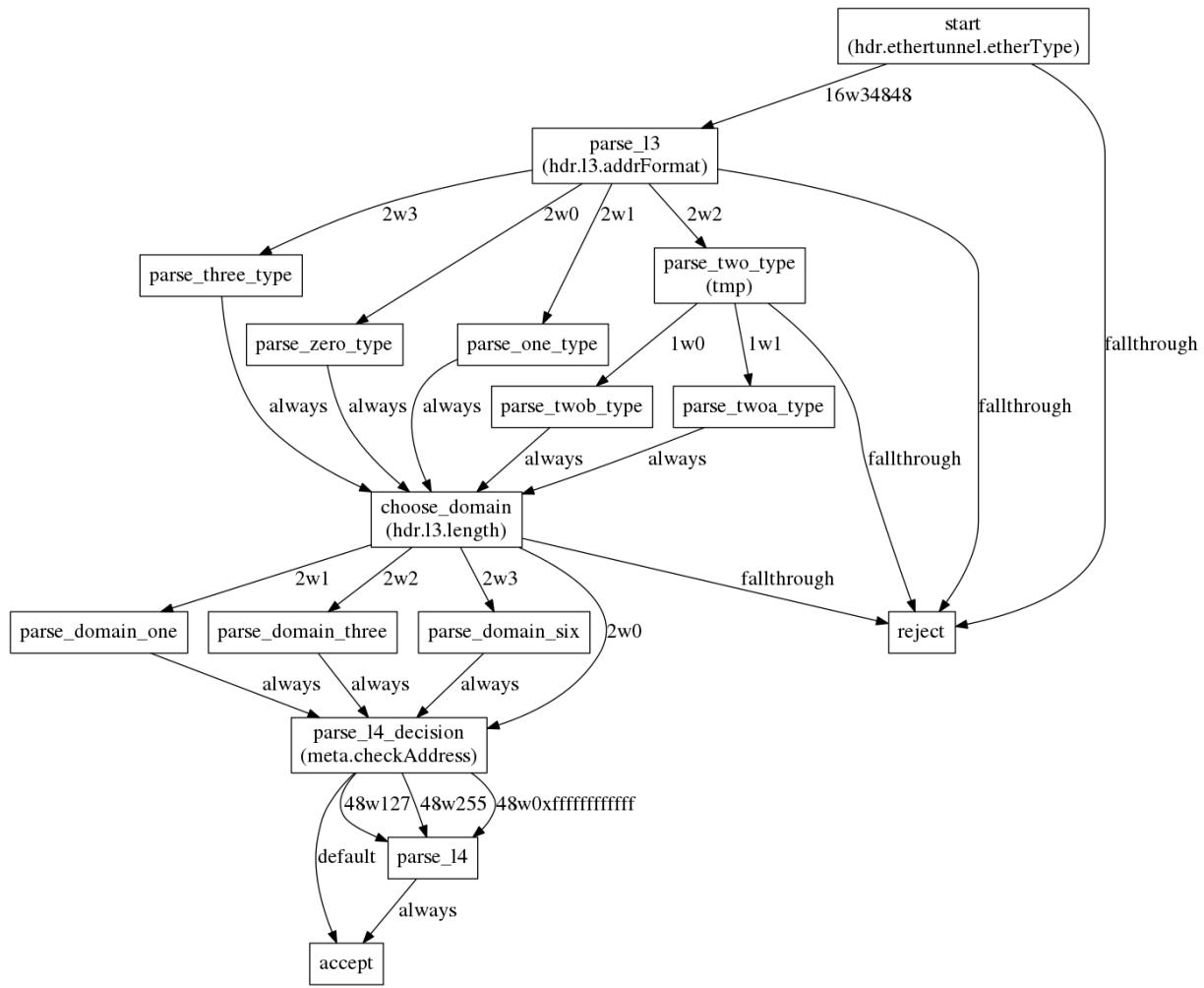


Abbildung 26: Durch ein offizielles Compiler-Backend erstelltes Zustandsübergangsdiagramm

eingehenden Paket die Headertypen `ethernettunnel_t`, `layer2_t` und `layer3_t` extrahiert, sodass nunmehr in einem jedem Paket als nächstes ein Feld mit einer Länge von einem Bit folgt. Dessen Wert wird im Zustand `parse_two_type` benötigt, um zwischen den Adressformaten „Subnet/Node“ und „Group Acknowledgement“ unterscheiden zu können. Um den Wert des besagten Feldes auszulesen, aber nicht extrahieren zu müssen, wird die Methode `lookahead` verwendet. Der so erhaltene Wert wird in die Variable `temp` gespeichert und daraufhin genutzt, um in den jeweils passenden Zustand (`parse_twob_type` oder `parse_twa_type`) überzuleiten. Das Namensschema der Zustände röhrt daher, dass diese Subadressstypen auch 2a beziehungsweise 2b genannt werden können [76]. In dem jeweiligen resultierenden Zustand wird der dementsprechende Header extrahiert. Ein jeder der fünf Zustände, in dem ein Adressformat extrahiert wird, verweist auf den nächsten Zustand: `choose_domain`. In diesem wird, basierend auf dem Wert des Feldes `length` des Headertyps `layer3_t`, ein spezifischer Domain-Header extrahiert und daraufhin in den Zustand `parse_l4_decision` übergeleitet. In Abbildung 26 ist erkennbar, dass lediglich für den `length`-Wert null direkt in den Zustand `parse_l4_decision` übergegangen wird.

```

1 state parse_14_decision{
2     transition select(meta.checkAddress){
3         (bit<48>) ADDR_ROUTER_NODE: parse_14;
4         (bit<48>) ADDR_ROUTER_GROUP: parse_14;
5         ADDR_ROUTER_UNIQUEID: parse_14;
6         default: accept;
7     }
8 }
```

Listing 32: Überprüfung des Parsers hinsichtlich Routeradressierung

In `parse_14_decision` wird überprüft, ob mit dem erhaltenen Paket der Router selbst adressiert wird. In Listing 32 ist zu erkennen, dass sobald das Metadatum `checkAddress` einer der als Konstanten definierten Adressen des Routers entspricht, in den Zustand `parse_14` übergegangen wird. Der Inhalt des Metadatums `checkAddress` wird in den Zuständen `parse_one_type`, `parse_twoa_type`, `parse_twob_type` und `parse_three_type` bestimmt, da hier die jeweilige Zieladresse aus dem Paket ausgelesen werden kann. Wie in Abbildung 26 zu erkennen ist, wird aus diesem Zustand heraus auf keinen weiteren, selbst definierten Zustand verwiesen, jedoch könnte dies für eine weitere Behandlung von Nachrichten, die an den Router adressiert sind, neu aufgegriffen werden. Da der Fokus dieser Arbeit jedoch nicht auf einer solchen weiteren Behandlung liegt, mündet der Zustand `parse_14` direkt in den Endzustand `accept`. Dieser Zustand bewirkt, dass die extrahierten Header des Pakets zusammen mit den generierten Metadaten an die Ingress-Pipeline übergeben werden.

8.4.4 P4-Ingress-Pipeline des Routers

In diesem Kapitel wird auf die Inhalte der Ingress-Pipeline eingegangen. Die extrahierten Bestandteile eines Pakets werden hier erstmals bearbeitet, da im eigentlich vorangestellten Control-Block `MyVerifyChecksum` keine Verarbeitungstätigkeiten stattfinden.

```

1 action drop_n_exit() {
2     standard_metadata.egress_spec = DROP_PORT;
3     exit;
4 }
5 action subnet_forward(egressSpec_t port) {
6     standard_metadata.egress_spec = port;
7 }
8 action group_forward(mcast_grp_t mcast_grp) {
9     standard_metadata.mcast_grp = mcast_grp;
10}
11 action broadcast_to_domain(mcast_grp_t mcast_grp) {
12     standard_metadata.mcast_grp = mcast_grp;
13}
```

Listing 33: `action`-Abschnitte der Ingress-Pipeline

Zuerst sind die in diesem Control-Block enthaltenen `action`-Abschnitte zu behandeln. Diese sind für die grundlegende Weiterleitung von Paketen verantwortlich und in Listing 33

abgebildet. Die erste vorgestellte **action** ist **drop_n_exit**, welche die Variable **DROP_PORT** verwendet. Die Variable wurde vorab, wie in Kapitel 8.4.1 beschrieben, als globale Konstante definiert, um auf diese Art und Weise einfacher für das gesamte Programm verändert werden zu können. Eine solche Veränderung kann nötig sein, weil der Drop-Port beim Start des Software-Switches über einen Parameter verändert werden kann. Ein Aufruf der **action drop_n_exit** hat zum Ziel, ein Paket zu verwerfen. Dazu wird das intrinsische Metadatum **egress_spec**, das den Port markiert, aus dem das Paket versendet werden soll, auf den Wert der Variable **DROP_PORT** gesetzt. Die Variable **DROP_PORT** enthält dabei den Standard-Drop-Port, der in der Datei **simple_switch.h** mit den Wert 511 festgelegt ist. Da das Verwerfen eines Pakets erst am Ende der Egress-Pipeline stattfindet, kann es durch weitere Bearbeitungsschritte der Ingress-Pipeline dieses P4-Programms dazu kommen, dass das Metadatum **egress_spec** überschrieben wird. Um einer solchen Überschreibung vorzubeugen wird am Ende der **action drop_n_exit** das namensgebende Schlüsselwort **exit** gesetzt. Dieses Schlüsselwort führt dazu, dass bei dessen Aufruf sofort an das Ende der Ingress-Pipeline gesprungen wird, sodass im Control-Block **MyIngress** keine weiteren Verarbeitungsaktionen das Metadatum **egress_spec** überschreiben können. Der zweite vorzustellende **action**-Abschnitt ist **subnet_forward**, der für Forwarding auf Basis der Subnetzinformation eines Pakets genutzt wird. Bei dieser **action** wird, wie in Listing 33 abgebildet, der Wert des Ausgangsports durch eine Weiterleitungstabelle übergeben, sodass dieser Wert lediglich **standard_metadata.egress_spec** zugewiesen werden muss. Für die beiden **action**-Abschnitte **broadcast_to_domain** und **group_forward** gilt, dass diese der **action subnet_forward** sehr ähneln. Allerdings arbeiten diese beiden **action**-Abschnitte nicht auf Basis eines Ausgangsports, sondern mit Multicast-Gruppennummern, die ebenfalls durch Weiterleitungstabellen bereitgestellt und dem intrinsischen Metadatum **mcast_grp** zugewiesen werden.

```

1 table port_domain_whitelist_exact {
2     key = {
3         standard_metadata.ingress_port : exact;
4         meta.domainField : exact;
5     }
6     actions = {
7         NoAction;
8         drop_n_exit;
9     }
10    size = 1024;
11    default_action = drop_n_exit();
12 }
```

Listing 34: Definition der Tabelle **port_domain_whitelist_exact**

In diesem Programm werden, um die erwähnten **action**-Abschnitte aufzurufen, zumeist Weiterleitungstabellen verwendet. Für die Funktionalität dieser P4-Implementierung sind drei verschiedene Tabellen definiert, die nunmehr vorgestellt werden. Die erste Tabelle, **port_domain_whitelist_exact**, ist in Listing 34 abgebildet. Der Key der Tabelle setzt

sich aus dem Ingress-Port, auf dem der Router eine Nachricht empfangen hat, und einem eigens erstellten Metadatum `domainField` zusammen. Bei einem Aufruf der Tabelle wird eine von zwei `action`-Abschnitten, die in Listing 34 dargestellt sind, aufgerufen. Wird durch die ausgelöste Tabellensuche keine Entsprechung in der Tabelle gefunden, führt dies dazu, dass das Paket durch die `action drop_n_exit` verworfen wird. Diese Funktionalität wird genutzt, um die in Kapitel 8.3 aufgestellte Bedingung zu erfüllen, dass ein durch den Router empfangenes Paket nur dann weitergeleitet werden soll, wenn in der Tabelle ein passender Eintrag enthalten ist. Empfängt der Router beispielsweise ein Paket dessen Domäne auf dem Eingangsinterface der Nachricht nicht zu erwarten ist, wird das Paket verworfen. Um mit diesem Mechanismus zu verhindern, dass Pakete von Hosts einer Domäne auch zu Hosts einer anderen Domänen versendet werden können, muss für einen Router daher immer eine korrekte Konfiguration dieser Weiterleitungstabelle vorliegen. Ein Beispiel eines korrekt konfigurierten Tabelleneintrags findet sich in Kapitel 8.5.

Die zweite Tabelle dieser Implementierung ist `destSubnet_type_exact`, in der ein Paket anhand seiner enthaltenen Domänen- und der Subnetzinformation weitergeleitet wird. Findet sich in der Tabelle eine Entsprechung, wird einer der in der Weiterleitungstabelle angeführten `action`-Abschnitte unter Zuhilfenahme einer in der Tabelle enthaltenen Variable ausgeführt. Die Variable kann dabei sowohl ein Ausgangsinterface als auch eine Multicastgruppe repräsentieren, um mithilfe dieser Tabelle Pakete nicht nur durch die `action subnet_forward` in ein Subnetz versenden, sondern auch mittels der `action broadcast_to_domain` an alle Subnetze einer Domäne weiterleiten zu können. Wird in dieser Tabelle keine Entsprechung gefunden, wird das Paket standardmäßig verworfen.

Die dritte und letzte Tabelle dieser Implementierung ist `destGroup_type_exact`. Diese Tabelle ist vergleichbar mit der Funktionalität der Tabelle `destSubnet_type_exact` und ermöglicht es einem Paket, anhand seiner Domänen- und Gruppeninformation eine Multicast-Gruppennummer zuzuordnen. Diese Nummer ist durch die Konfiguration des Routers mit einer Gruppe von Interfaces assoziiert, auf welchen die Teilnehmer der Multicastgruppe zu finden sind. Die für diese Funktionalität genutzte `action` ist `group_forward`. Wird in der besagten Tabelle keine Entsprechung gefunden, wird das Paket verworfen.

```

1 if (hdr.domain_one.isValid()) {
2     meta.domainField=(bit<48>)hdr.domain_one.domainField;
3     port_domain_whitelist_exact.apply();
4 }
5 else if(hdr.domain_three.isValid()){
6     meta.domainField=(bit<48>)hdr.domain_three.domainField;
7     port_domain_whitelist_exact.apply();
8 }
9 else if(hdr.domain_six.isValid()){
10    meta.domainField=hdr.domain_six.domainField;
11    port_domain_whitelist_exact.apply();
12 }
13 else{

```

```

14     meta.domainField=STANDARD_DOMAIN;
15     port_domain_whitelist_exact.apply();
16 }
```

Listing 35: Behandlung der Domäneninformation in der Ingress-Pipeline

Der Definition der besprochenen Tabellen folgt der `apply`-Abschnitt, der dazu dient, die Tabellen aufzurufen, Metadaten zu generieren und Header-Felder zu modifizieren. In Listing 35 finden sich die ersten Bestandteile des `apply`-Abschnittes. Darin ist zu erkennen, dass dem Metadatum `domainField` Domäneninformation, die für die zuvor genannten Tabellen wichtig ist, zugewiesen wird. Da bei LonTalk der Fall auftreten kann, dass eine Nachricht keinerlei Domäneninformation enthält, wird auch dieses Szenario in den letzten Zeilen des Listings 35 behandelt. Diesen Zeilen kann entnommen werden, dass wenn in einem Paket kein Domain-Header vorhanden ist, dem Metadatum `domainField` die Konstante `STANDARD_DOMAIN` zugewiesen wird. Der Grund für diese Zuweisung und die Verwendung der Konstante `STANDARD_DOMAIN` wurde bereits in Kapitel 8.3 dargelegt. Daraufhin wird die Tabelle `port_domain_whitelist_exact` auf die betroffenen Pakete angewendet. Dies ist möglich, da dem Paket-Metadatum `domainField` ein Wert zugewiesen wurde.

```

1 if (hdr.three_type.isValid()) {
2     meta.destSubnet = hdr.three_type.destSubnet;
3     destSubnet_type_exact.apply();
4 }
5 else if (hdr.twoa_type.isValid()) {
6     meta.destSubnet = hdr.twoa_type.destSubnet;
7     destSubnet_type_exact.apply();
8 }
9 .
10 .
11 .
12 else if (hdr.one_type.isValid()) {
13     meta.destGroup = hdr.one_type.destGroup;
14     destGroup_type_exact.apply();
15 }
```

Listing 36: Behandlung der Subnetz- und Gruppeninformation in der Ingress-Pipeline

Listing 36 zeigt, wie daraufhin die für die Weiterleitung zuständigen Tabellen `destSubnet_type_exact` und `destGroup_type_exact` bei den unterschiedlichen Adressierungsarten aufgerufen werden. Da das Adressformat „Group“ aufgrund der Gruppeninformation eines Pakets weitergeleitet wird, hebt sich das Adressformat durch die Verwendung der Tabelle `destGroup_type_exact` gegenüber den anderen Adressierungsarten, bei denen die Tabelle `destSubnet_type_exact` zur Anwendung kommt, ab. All diesen Adressierungsarten ist jedoch gemein, dass dem für die Tabellensuche benötigte Metadatenfeld `destSubnet` respektive `destGroup` ein Wert zugewiesen werden muss. Diese Zuweisung ist ebenfalls in Listing 36 ersichtlich.

```

1 if (hdr.12.priority==0){
2   standard_metadata.priority = (bit<3>)0;
3 }
4 else{
5   standard_metadata.priority = (bit<3>)7;
6 }
```

Listing 37: Behandlung des Pri-Feldes in der Ingress-Pipeline

In diesem `apply`-Abschnitt findet sich außerdem noch ein Programmteil, in dem der Prioritätsstatus eines Pakets behandelt wird. Wie dieser in diesem P4-Programm implementiert wird, kann Listing 37 entnommen werden. In der Implementierung wird, wenn das Pri-Bit, das sich im Header auf Schicht zwei befindet, den Wert eins enthält, dem intrinsischen Metadatum `priority`, das für die Priorisierung eines Pakets genutzt werden kann, der Wert sieben zugewiesen. Dieser Wert ist der höchste Wert, der für den Priorisierungsmechanismus des Routers zur Verfügung steht [101]. Diese Quelle gibt auch an, dass der niedrigste Priorisierungswert null ist. In der Log-Datei des Routers kann die jeweils getroffene Zuweisung auch eingesehen werden. Ob eine Priorisierung des Pakets innerhalb des Routers aufgrund dieser Zuweisungen stattfindet, lässt sich jedoch nicht überprüfen, da es keinen eindeutigen Nachweis gibt [102].

8.4.5 P4-Egress-Pipeline und Deparser

In diesem Kapitel wird auf die Inhalte der Egress-Pipeline eingegangen, da diese im Programmverlauf an die Ingress-Pipeline anschließt. Die Tätigkeiten, die in der Egress-Pipeline vorgenommen werden, beschränken sich auf ein Mindestmaß, da hier nur noch ausgehende Nachrichten verändert werden. Der Begriff „ausgehende Nachrichten“ beinhaltet neben eingehenden Nachrichten auch durch Multicasting erstellten Nachrichten. Neben den Inhalten der Egress-Pipeline werden in diesem Kapitel auch die Inhalte des Deparsers beschrieben.

Die Egress-Pipeline enthält lediglich die `action drop`, die einen ähnlichen Zweck wie die Ingress-`action drop_n_exit`. Diese beiden `action`-Abschnitte unterscheiden sich lediglich dadurch, dass für die Egress-`action` der Befehl `exit` nicht nötig ist, da in diesem Fall die Pipeline nicht verlassen werden muss. Werden Pakete dem Multicasting-Prozess des LonTalk-Routers ausgesetzt, hätte dies normalerweise zur Folge, dass die Nachricht auch aus dem Ingress-Port versendet werden würde. Dies liegt im Verhalten und dem Umgang des Software-Switches mit dem zugrundeliegenden Metadatum `mcast_grp` begründet. Um diesem Umstand entgegenzuwirken, wird ein jedes Paket durch die `action drop` verworfen, das an einen Host des Ursprungssegments adressiert ist. Dies bewirkt, dass der Egress-Port einer Nachricht niemals dem eigenen Ingress-Port gleicht.

```

1 control MyDeparser(packet_out packet, in headers hdr) {
2     apply {
3         packet.emit(hdr.ethertunnel);
4         packet.emit(hdr.12);
5         packet.emit(hdr.13);
6         packet.emit(hdr.zero_type);
7         packet.emit(hdr.one_type);
8         packet.emit(hdr.twoa_type);
9         packet.emit(hdr.twob_type);
10        packet.emit(hdr.three_type);
11        packet.emit(hdr.domain_one);
12        packet.emit(hdr.domain_three);
13        packet.emit(hdr.domain_six);
14    }
15 }
```

Listing 38: Deparser der Router-Implementierung

Da der auf die Egress-Pipeline folgende Control-Block `MyComputeChecksum` in dieser Implementierung nicht genutzt wird, werden die Header eines Pakets nunmehr direkt dem Deparser übergeben. Der Deparsingvorgang wurde bereits allgemein in Kapitel 4.9 beschrieben. Listing 38 enthält alle Header, die für diesen Vorgang innerhalb dieses P4-Programms genutzt werden und daher Bestandteil eines ausgehenden Pakets sein können.

8.4.6 P4-Debugging-Tabelle

Die Gründe, die für die Nutzung einer Debugging-Tabelle sprechen, wurden bereits in Kapitel 8.3 erläutert. In Kapitel 8.4.1 wird behandelt, dass durch das Hinzuziehen von Präprozessorbefehlen die Verwendung von Debugging-Tabellen für den/die Nutzer*in optional bleibt.

```

1 control essential_debug(in headers hdr, in standard_metadata_t
2   standard_metadata, in metadata meta)
3 {
4     table dbg_table {
5         key = {
6             standard_metadata.egress_spec : exact;
7             standard_metadata.mcast_grp: exact;
8             standard_metadata.priority : exact;
9             meta.destSubnet : exact;
10            meta.domainField : exact;
11            meta.destGroup : exact;
12            meta.checkAddress : exact;
13        }
14        actions = { NoAction; }
15        const default_action = NoAction();
16    }
17    apply {
18        dbg_table.apply();
19    }
}
```

Listing 39: Definition der Debugging-Tabelle in `debug.p4`

Der für die Implementierung einer Debugging-Tabelle verwendete Control-Block wurde in diesem P4-Programm in der separaten Datei `debug.p4` definiert (siehe Listing 39). Um die Debugging-Tabelle im LonTalk-Programm nutzen zu können, wird der Inhalt der betroffenen Datei mithilfe des Präprozessorbefehls `include` integriert. Dieser Vorgang kann im P4-Programm allerdings erst nach dem Parser stattfinden, da manche Paketinformationen frühestens nach diesem verfügbar sind. Ein Beispiel hierfür ist das Metadatum `checkAddress`, dessen Inhalt erst durch die Vorgänge des Parsers befüllt wird. Die Ausführung der Debugging-Tabelle als eigenständiger Control-Block, dem die Gesamtheit aller Paketinformationen übergeben wird, hat zum Vorteil, dass die Debugging-Tabelle innerhalb des P4-Programms mehrfach instanziert werden kann. Die Definition der Debugging-Tabelle enthält einen Key, welcher die darzustellenden Informationen repräsentiert. Sollte etwa der Prioritätsstatus einer jeden, den Software-Switch durchlaufenden Nachricht in Erfahrung gebracht werden, muss lediglich das entsprechende Metadatum in der Tabelle als Teil des Keys festgelegt werden (siehe Listing 39). Da das Ziel dieser Tabelle lediglich darin liegt, Werte darzustellen und da keine `action` auszuführen ist, muss als Standard-`action NoAction` definiert werden. Die Verwendung von `NoAction` und der für die Ausführung benötigte Aufruf der Tabelle im `apply`-Abschnitt des Control-Blocks ist in Listing 39 visualisiert. Um die Debugging-Tabelle nun auch im P4-Programm nutzen zu können, muss am Beginn von „MyIngress“ eine Instanz des in der Datei `debug.p4` enthaltenen Control-Blocks erstellt werden. Diese Instanz kann daraufhin im gesamten `apply`-Abschnitt der Ingress-Pipeline mittels der Funktion `apply` aufgerufen werden. Bei einem Aufruf der Tabelle werden nunmehr alle im Key repräsentierten Felder und deren Inhalte in die Log-Datei des Routers geschrieben. Durch ein mehrfaches Aufrufen der Debugging-Tabelle kann auch eine Veränderung von Werten genauer untersucht und analysiert werden.

8.5 Konfiguration des LonTalk-Routers

In diesem Kapitel werden die Möglichkeiten, wie der Router konfiguriert werden kann, genauer vorgestellt. Die für die Konfiguration der Tabellen genutzten Befehle werden für beide Optionen in diesem Kapitel anhand von Beispielen dargestellt.

Die einfachste Option bietet die Verwendung des vorgefertigten Programms `simple_switch_CLI`. Wie bereits in Kapitel 8.1 erläutert, implementiert dieses Programm einen Controller und nutzt einen durch den Software-Switch bereitgestellten Thrift-Port, um den Switch zu konfigurieren. Für diese Zwecke kann, wie in Listing 19 ersichtlich, auch eine Textdatei verwendet werden, sodass der Software-Switch ohne eine zeilenweise Eingabe von Befehlen wie gewünscht konfiguriert werden kann. Enthält die besagte Textdatei Leerzeilen, führt dies dazu, dass in der Log-Datei des Zielgeräts

pro Leerzeile jeweils eine Fehlermeldung hinzugefügt wird. Jedoch hat eine enthaltene Leerzeile keine praktischen Auswirkungen auf die Konfiguration des Software-Switches.

Eine weitere Option bietet sich durch die Verwendung der P4Runtime. Bei dieser kann durch ein Controller-Programm der Inhalt der für das Forwarding genutzten Tabellen festgelegt und verändert werden. Außerdem bietet die P4Runtime, wie in Kapitel 4.8 beschrieben, die Möglichkeit, das durch in einem kompilierten P4-Programm festgelegte Weiterleitungsverhalten zur Laufzeit auf das Gerät zu übersenden. Der in dieser Arbeit zum Einsatz kommende Controller basiert auf einem Beispielprogramm, das bereits auf der offiziellen virtuellen Maschine des P4-Sprachkonsortiums enthalten ist. Folgend werden nunmehr die wichtigsten Ausschnitte dieses Controller-Programms vorgestellt.

Beim Start des Controller-Programms wird eine JSON-Datei benötigt, die die gesamte gewünschte Konfiguration des Zielgeräts enthält. Diese Datei muss den Pfad zu den in Kapitel 4.8 beschriebenen Dateien enthalten. Bei den benötigten Dateien handelt es sich um die „P4 Device Config“ und die P4Info-Datei.

```

1 switch = p4runtime_lib.bmv2.Bmv2SwitchConnection(
2     name='switch',
3     address='127.0.0.1:50051',
4     device_id=0,
5     proto_dump_file='logs/switch-p4runtime-requests.txt')
6 switch.MasterArbitrationUpdate()
7 switch.SetForwardingPipelineConfig(p4info=p4info_helper.p4info,
    bmv2_json_file_path=bmv2_file_path)

```

Listing 40: Diverse die Konfiguration ermögliche Programmteile aus `mycontroller.py`

Wie aus Inhalten des Listing 40 geschlussfolgert werden kann, werden alle P4Runtime Nachrichten, die den adressierten Switch betreffen, in einer Log-Datei abgespeichert. Außerdem wird der Controller, wie in der vorletzte Zeile des Listings 40 abgebildet, für den betreffenden Software-Switch als Master festgelegt. Dieser Vorgang ist notwendig, um eine Schreiboperation durchführen zu können und ergibt sich aus der Nutzung der P4Runtime. In der letzten Zeile des Listings ist zu sehen, wie daraufhin mithilfe der Python-Funktion `SetForwardingPipelineConfig` das Verhalten des P4-Programms auf dem Zielgerät installiert wird. Um die einzelnen Tabelleneinträge, die in der dem Programm übergebenen JSON-Datei enthalten sind, auf das Zielgerät zu übersenden, muss der Inhalt dieser Datei zuvor noch geparsst werden. Auf diese Weise werden für das Programm relevante Informationen extrahiert.

```

1 if default_action_type is True:
2     table_entry = p4info_helper.buildTableEntry(
3         table_name=table_name,
4         default_action=True,
5         action_name=action_name,
6         action_params=action_params)
7 else:
8     table_entry = p4info_helper.buildTableEntry(
9         table_name=table_name,

```

```

10     match_fields=match,
11     action_name=action_name,
12     action_params=action_params)
13 switch.WriteTableEntry(table_entry)

```

Listing 41: Erstellen eines Tabelleneintrags in mycontroller.py

Daraufhin kann, wie in Listing 41 dargestellt, ein Tabelleneintrag erstellt und an den BMv2-Switch gesendet werden. In diesem Listing ist zu erkennen, dass normale Tabelleneinträge anders zu behandeln sind als Tabelleneinträge, welche die `default-action` einer Tabelle festlegen.

```

1 mc_entry = p4info_helper.buildMulticastGroupEntry(
2     k["multicast_group_id"],
3     k['replicas'])
4 switch.WriteMulticastGroupEntry(mc_entry)

```

Listing 42: Erstellen eines Multicastgruppen-Eintrags in mycontroller.py

Die Konfiguration der Multicast-Engine des Zielgeräts funktioniert nach einem ähnlichen Muster. Dieses ist in Listing 42 einzusehen und stützt sich auf die Funktionen `buildMulticastGroupEntry` und `WriteMulticastGroupEntry`.

Um die Konfigurationsdateien besser vergleichen zu können und die Tabellen, die in Kapitel 8.4.4 vorgestellt sind, greifbarer zu machen, werden Ausschnitte der beteiligten Konfigurationsdateien in diesem Kapitel präsentiert.

```

1 {
2     "table": "MyIngress.destSubnet_type_exact",
3     "default_action": true,
4     "action_name": "MyIngress.drop_n_exit",
5     "action_params": {}
6 },
7 {
8     "table": "MyIngress.destSubnet_type_exact",
9     "match": {
10         "meta.destSubnet": 1,
11         "meta.domainField": 1
12     },
13     "action_name": "MyIngress.subnet_forward",
14     "action_params": {
15         "port": 1
16     }
17 }

```

Listing 43: Teil einer Tabellenkonfiguration in lontalk.json

```

1 table_set_default destSubnet_type_exact drop_n_exit
2 table_add MyIngress.destSubnet_type_exact subnet_forward 0x1 0x1 => 1

```

Listing 44: Teil einer Tabellenkonfiguration in lontalk.txt

Bei einem Vergleich der Listings 43 und 44 lässt sich erkennen, dass die Konfiguration eines Tabelleneintrags mithilfe der Option `cli`, abgebildet in Listing 44, deutlich kompakter ausfällt. Dies ist vor allem dem JSON-Format geschuldet, das bei der Konfiguration

mittels P4Runtime verwendet wird. In beiden Listings ist zuerst die Konfiguration der `default-action` für Tabelle `destSubnet_type_exact` angeführt. Darauf folgt jeweils ein Eintrag, der für ein empfangenes Paket mit der Domäneninformation eins und dem enthaltenen Zielsubnetz eins dem Paket das Ausgangsinterface eins zuweist. Dies geschieht durch Verwendung der `action subnet_forward`. Die in den beiden Listings vorgestellten Schemata sind für alle weiteren Tabellen ident und erfordern lediglich ein Anpassen der jeweiligen Konfigurationsparameter.

```

1 "multicast_group_id" : 3,
2 "replicas" : [
3     {
4         "egress_port" : 3,
5         "instance" : 3
6     },
7     {
8         "egress_port" : 4,
9         "instance" : 3
10    }
11 ]
12 }
```

Listing 45: Definition der Multicastgruppe drei in `lontalk.json`

```

1 mc_mgrp_create 3
2 mc_node_create 6 3
3 mc_node_create 7 4
4 mc_node_associate 3 6
5 mc_node_associate 3 7
```

Listing 46: Definition der Multicastgruppe drei in `lontalk.txt`

Die Konfiguration der Multicast-Engine wird in diesem Kapitel anhand der Einträge in den Listings 45 und 46 vorgestellt. In diesen Konfigurationsbeispielen ist jeweils die nötige Konfiguration enthalten, um die Multicastgruppe drei für die Gruppenadressierung der in der Topologie enthaltenen Hosts H13 und H14 zu erstellen. Für die Konfiguration mittels P4Runtime, siehe Listing 45, wird die Definition einer `id` für die entsprechende Multicastgruppe benötigt. Der dafür vergebene Wert spiegelt sich daraufhin auch in den Werte-Feldern des Schlüssels `instance` wieder. Wie in Listing 45 außerdem ersichtlich ist, werden die Switch-Ports drei und vier dieser Multicastgruppe zugeordnet. Diese Switch-Ports können durch Inspektion des Listings 18 und der Abbildung 23 eindeutig den Hosts H13 und H14 zugerechnet werden. Um dieselbe Konfiguration mittels der Option `cli` vorzunehmen, muss, wie in Listing 46 dargestellt, zuerst eine Multicastgruppe erstellt werden. Daraufhin werden die betroffenen Interfaces jeweils mit einem einzigartigen Identifikator, in diesem Fall sind dies die Zahlen sechs und sieben, assoziiert, um einen Multicast-Knoten zu erstellen. Mithilfe der Anweisung `mc_node_associate` können diese Knoten der Multicastgruppe drei zugewiesen werden.

8.6 Testung des LonTalk-Routers

Kapitel 8.4 beschreibt die Inhalte des P4-Programms, sodass in diesem Kapitel nunmehr die für diese Arbeit genutzten Testabläufe beschrieben und anhand ausgewählter Beispiele dargestellt werden können.

```
1 [11:13:09.709] [bmv2] [I] [thread 3249] Using priority queueing!
```

Listing 47: Log-Datei zeigt Aktivierung des Priorisierungsmechanismus an

Allem voran sind in diesem Kapitel die Gründe für die Testung des Programms anzuführen. Eine Testung wurde durchgeführt, um sicherstellen zu können, dass das im P4-Programm festgehaltene Verhalten des LonTalk-Routers auch die beabsichtigten Ergebnisse hervorbringt. Außerdem konnte erst durch eine ausführliche Testung die korrekte Konfiguration des Routers verifiziert werden. Darüber hinaus konnte durch eine erfolgreiche Testung auch der korrekte Aufbau der in Kapitel 7.7 beschriebenen Topologie bestätigt werden. Die Gesamtheit der durchgeföhrten Tests lässt sich dabei in zwei Phasen einordnen. Erstens wurde bereits im Laufe der Programmertätigkeit zu testen begonnen, um neu hinzukommende Funktionen erstmalig auf deren Funktion zu überprüfen. Zweitens wurde der LonTalk-Router auch nach Beendigung der Programmertätigkeiten einem abschließenden manuellen Testlauf unterzogen, um sicherstellen zu können, dass auch das Zusammenspiel der bereits überprüften Programmfeatures gegeben ist. Ein Feature des LonTalk-Routers, das beispielsweise bereits während der Programmertätigkeit getestet wurde, ist die Funktionalität des Pri-Feldes und der damit verbundenen Priorisierungsmechanismus des Routers. Diese Überprüfung war notwendig, da untersucht werden musste, ob der Mechanismus, wie in Kapitel 8.3 beschrieben, erfolgreich aktiviert wurde. Das Ergebnis eines dieser Tests lässt sich in Listing 47 erkennen. Diese Abbildung zeigt einen Ausschnitt der Log-Datei des BMv2-Switches und enthält die Worte „Using priority queueing!“, die eine erfolgreiche Aktivierung angeben. Beispiele für andere Features, die peu à peu implementiert und getestet wurden, sind die verschiedenen Adressformate des LonTalk-Protokolls und die möglichen Arten von Domäneninformation. Bei dem abschließende Test wurde versucht die verschiedenen Forwarding-Szenarien abzudecken. Dabei wurde beispielsweise das Verhalten des Routers aufgrund von Domäneninformation untersucht. Dies geschah einerseits durch die Nutzung der verschiedenen Längen von Domäneninformation, andererseits aber auch durch den Versuch Nachrichten zwischen verschiedenen Domänen zu übermitteln. Nunmehr werden zwei Testbeispiele genutzt, um dem/der Leser*in einen Eindruck über den Ablauf eines Test zu bieten.

Das erste Testbeispiel widmet sich dem Versuch, Nachrichten zwischen Hosts, die verschiedenen Domänen angehören, zu übermitteln. Bei diesem Versuch

```
###[ Ethertunnel ]###
    dst      = ff:ff:ff:ff:ff:ff
    src      = 00:00:0a:00:00:02
    type     = 0x8820
###[ LonTalk_L2 ]###
    Pri      = 0L
    Path     = 0L
    Backlog  = 0L
###[ LonTalk_L3 ]###
    Version   = 0L
    PacketFormat= 0L
    AddressFormat= 2L
    Length    = 3L
###[ Source_Address_Information ]###
    SourceSubnet= 2L
    Bit       = 1L
    SourceNode= 1L
###[ 2a_Type ]###
    DestSubnet= 2L
    StuffBit  = 1L
    DestNode  = 42L
###[ Domain_6Byte ]###
    Domain    = 2L
###[ Raw ]###
    load      = 'Testnachricht von H12 an H22!'
```

Abbildung 27: Inhalt einer Testnachricht von H12 an H22

wird erwartet, dass die betroffene Nachricht durch den LonTalk-Router verworfen wird. Um diesen Test durchführen zu können, müssen auf den betroffenen Hosts die jeweiligen Scapy-Programme gestartet werden. Da H12 der Sender dieser Nachricht sein sollte, wird hierzu das Programm `send.py` mit dem Befehl `sudo ./python/send.py -type 2 -payload "Testnachricht von H12 an H22!"` gestartet. H22, der Empfänger, startet daraufhin das Empfangsprogramm mit dem Befehl `sudo ./python/receive.py`. Daraufhin werden am Sender noch einige Zusatzinformationen, wie die zu verwendende Domänennummer, der genaue Adresstyp, die Zielsubnetznummer und die Zielknotenadresse benötigt. Das Format der durch den Sender erstellten Nachricht stellt sich darauffolgend wie in Abbildung 27 dar. Hier ist zu beachten, dass die dargestellte Nachricht aus den in Scapy definierten Headern und dem Inhalt der Nachricht besteht. Diese Bestandteile bauen dabei von oben nach unten aufeinander auf, sodass die Testnachricht mit dem Header `Ethertunnel` beginnt. Auf `Ethertunnel` folgt der Header `LonTalk_L2`, der dem Bereich `Raw` voransteht. Dieser abschließende Bereich enthält den Inhalt der Nachricht.

```
1 Processing packet received on port 2
2 ...
3 ...
4 build/lontalk.p4(343) Primitive meta.domainField=hdr.domain_six.
  domainField
5 Applying table 'MyIngress.port_domain_whitelist_exact'
```

```

6| Looking up key:
7| * standard_metadata.ingress_port: 0002
8| * meta.domainField : 000000000002
9| Table 'MyIngress.port_domain_whitelist_exact': miss
10| Action entry is MyIngress.drop_n_exit -
11| Action MyIngress.drop_n_exit
12| ...
13| ...
14| Egress port is 511
15| Dropping packet at the end of ingress

```

Listing 48: Log-Datei zeigt Ausschnitt der Verarbeitung einer Testnachricht von H12 an H22

In Listing 48 sind die Verarbeitungsschritte des LonTalk-Routers zu erkennen. Sobald der Router die Nachricht auf Port zwei erhält, wird der Parser in Gang gesetzt, um in der darauffolgenden Ingress-Pipeline die Tabelle `port_domain_whitelist_exact` auszuführen zu können. Aus Listing 48 außerdem ist ersichtlich, dass bei der darauffolgenden Tabellensuche keine Entsprechung gefunden wird, sodass die Nachricht mithilfe der action `drop_n_exit` verworfen wird. Das Verwerfen der Nachricht hat erwartungsgemäß auch zur Folge, dass im Konsolenfenster von H22 keine Nachricht entgegengenommen wird.

```

###[ Ethertunnel ]###
    dst      = ff:ff:ff:ff:ff:ff
    src      = 00:00:0a:00:00:02
    type     = 0x8820
###[ LonTalk_L2 ]###
    Pri      = 0L
    Path     = 0L
    Backlog  = 0L
###[ LonTalk_L3 ]###
    Version   = 0L
    PacketFormat= 0L
    AddressFormat= 3L
    Length    = 3L
###[ Source_Address_Information ]###
    SourceSubnet= 2L
    Bit       = 1L
    SourceNode= 1L
###[ 3_Type ]###
    DestSubnet= 1L
    UniqueAddress= 42L
###[ Domain_6Byte ]###
    Domain    = 1L
###[ Raw ]###
        load      = 'Testnachricht von H12 an H11!'

```

Abbildung 28: Inhalt einer Testnachricht von H12 an 11 (Sender)

Durch das zweite hier angeführte Testbeispiel soll eine Nachrichtenübermittlung zwischen H12 und H11 dargestellt werden. Um die Testnachricht am Empfänger (H11) visualisieren zu können, muss auf diesem der Befehl `sudo ./python/receive.py` ausgeführt werden. Der Sender (H12) benötigt wiederum den folgenden Befehl:

```
sudo ./python/send.py -type 3 -payload "Testnachricht von H12 an H11!".
```

Wie dies bereits bei dem ersten Testbeispiel der Fall war, muss nunmehr auch für diesen Test zusätzliche Information eingegeben werden. Benötigt wurde für die abgesetzte Nachricht die im Paket enthaltene Domäneninformation, die Zielsubnetzadresse und eine für das Adressformat „Unique ID“ benötigte Adresse. Diese Informationen werden darauffolgend auch im Konsolenfenster des Senders, wie in Abbildung 28 visualisiert, dargestellt.

```

1 Processing packet received on port 2
2 ...
3 ...
4 build/lontalk.p4(343) Primitive meta.domainField=hdr.domain_six.
   domainField
5 Applying table 'MyIngress.port_domain_whitelist_exact'
6 Looking up key:
7 * standard_metadata.ingress_port: 0002
8 * meta.domainField : 000000000001
9 Table 'MyIngress.port_domain_whitelist_exact': hit with handle 1
10 Dumping entry 1
11 Match key:
12 * standard_metadata.ingress_port: EXACT      0002
13 * meta.domainField : EXACT      000000000001
14 Action entry: NoAction -
15 Action entry is NoAction -
16 Action NoAction
17 ...
18 ...
19 Applying table 'MyIngress.destSubnet_type_exact'
20 Looking up key:
21 * meta.domainField : 000000000001
22 * meta.destSubnet : 01
23 Table 'MyIngress.destSubnet_type_exact': hit with handle 0
24 Dumping entry 0
25 Match key:
26 * meta.domainField : EXACT      000000000001
27 * meta.destSubnet : EXACT      01
28 Action entry: MyIngress.subnet_forward - 1,
29 Action entry is MyIngress.subnet_forward - 1,
30 Action MyIngress.subnet_forward
31 build/lontalk.p4(270) Primitive standard_metadata.egress_spec = port
32 build/lontalk.p4(387) Condition "hdr.12.priority==0" (node_28) is true
33 ...
34 ...
35 Transmitting packet of size 60 out of port 1

```

Listing 49: Log-Datei zeigt Ausschnitt der Verarbeitung einer Testnachricht von H12 an H11

In Listing 49 sind einige Verarbeitungsschritte des Routers abgebildet die bei diesem Testfall ausgeführt werden. Die ersten Schritte decken sich dabei mit den Schritten des ersten Testfalls, wobei in diesem Fall die Tabellensuche `port_domain_whitelist_exact` positiv verläuft und die Nachricht somit nicht verworfen wird. Daraufhin wird aufgrund des verwendeten Adresstyps eine Entsprechung in der Tabelle `destSubnet_type_exact` gesucht. Die erfolgreiche Suche hat zur Folge, dass in der `action subnet_forward` der Ausgangsport, der für diese Nachricht bestimmt ist, den Wert eins erhält. Abschließend ist in Listing 49 zu erkennen, dass diese Zuweisung ausgeführt und das Paket erfolgreich aus dem Port eins versendet wird.

```

###[ Ethernet ]###
dst      = ff:ff:ff:ff:ff:ff
src      = 00:00:0a:00:00:02
type     = 0x8820
-----
###[ LonTalk_L2 ]###
Pri     = 0L
Path    = 0L
Backlog = 0L
-----
###[ LonTalk_L3 ]###
Version = 0L
PacketFormat= 0L
AddressFormat= 3L
Length   = 3L
-----
###[ Source_Address_Information ]###
SourceSubnet= 2L
Bit      = 1L
SourceNode= 1L
-----
###[ 3_Type ]###
DestSubnet= 1L
UniqueAddress= 42L
-----
###[ Domain_6Byte ]###
Domain   = 1L
-----
###[ Raw ]###
load     = 'Testnachricht von H12 an H11!'

```

Abbildung 29: Inhalt einer Testnachricht von H12 an 11 (Empfänger)

Die vom Sender versendete Testnachricht wird daraufhin durch H11, der mit dem Ausgangsport des LonTalk-Routers verbunden ist (siehe Abbildung 23), visualisiert. Der Aufbau dieser Nachricht verläuft nach demselben Prinzip wie dies bereits am Sender vorgestellt wurde. Bei einem Vergleich der Abbildung 29, welche das Nachrichtenformat aus Sicht des Empfängers darstellt, mit Abbildung 28, die das Format des Senders abbildet, ist zu erkennen, dass die generelle Header-Abfolge gleich ist. Außerdem kann durch eine genaue Betrachtung dieser Abbildung geschlussfolgert werden, dass die Inhalte der jeweiligen Header sich gleichen und die Nachricht daher korrekt durch den Router weitergeleitet wurde. Aus diesem Grund ist dieser Test als positiv zu bewerten und zeigt an diesem konkreten Beispiel, dass Implementierung des LonTalk-Routers wie beabsichtigt umgesetzt wurde.

8.7 Inbetriebnahme der virtuellen Maschine

In diesem Kapitel werden für den Zweck der Reproduzierbarkeit die einzelnen Schritte dargelegt, die nötig sind, um den LonTalk-Router auf der verwendeten Entwicklungsplatt-

form zu starten. Diese Schritte werden auch in der Datei `createvm.pdf`, die der gedruckten Fassung der Arbeit beiliegt, aufgelistet.

In Kapitel 7.4 wurden bereits die beiden für die Installation nötigen Programme VirtualBox und Vagrant erwähnt. Nachdem diese installiert sind kann mit den für die Reproduktion der Entwicklungsplattform nötigen Schritten fortgefahren werden. Im Zentrum steht dabei das Github-Repository [77], das die offizielle Tutorial VM enthält. Dieses muss heruntergeladen und entpackt werden. Nachdem in den Unterordner `vm` navigiert wurde, kann der Befehl `vagrant up` ausgeführt werden. Dieser Befehl erstellt daraufhin eine virtuelle Maschine, die nach einiger Wartezeit genutzt werden kann. Der erste Schritt auf der erstellten virtuellen Maschine ist das Wechseln auf den User `vagrant`. Dies kann durch den Befehl `sudo su - vagrant` erreicht werden. Daraufhin sollte für deutschsprachige Nutzer*innen das voreingestellte Tastaturlayout mittels dem Befehl `sudo dpkg-reconfigure keyboard-configuration` geändert werden. Als nächster Schritt wird der Inhalt des Ordners `tutorials` mit dem Befehl `sudo cp -r /home/p4/tutorials /home/vagrant` in das Home-Verzeichnis des Users `vagrant` übernommen. Im Home-Verzeichnis des Users `vagrant` wird daraufhin ein Ordner erstellt, der die Projektdateien beinhalten soll. Für das Hinzufügen der Projektdateien gibt es zwei verschiedene Quellen. Einerseits enthält der der gedruckten Arbeit beiliegende Datenträger diese Dateien, andererseits sind die Dateien auch als Repository auf dem GitLab des Studiengangs ITS (Informationstechnik und Systemmanagement)¹⁰ verfügbar. Die Dateien müssen lediglich in den zuvor erstellten Ordner platziert werden. Ist dies geschehen, kann der Standarduser dieser VM mithilfe des Befehls `sudo nano /etc/lightdm/lightdm.conf.d/10-lightdm.conf` verändert werden. Dazu muss lediglich der Wert des Schlüssels `autologin-user` von `p4` auf `vagrant` geändert werden. Wurde diese Änderung getätigt, kann die virtuelle Maschine neu gestartet werden, sodass nunmehr der User `vagrant` aktiv ist. Daraufhin muss mit dem Befehl `sudo pip install termcolor` noch eine benötigte Python-Bibliothek installiert werden. Um das manuelle Setup ausführen zu können, bedarf es noch zweier Befehle. Der erste Befehl `sudo chown -R vagrant ~` ändert den Eigentümer-Benutzer aller im Home-Verzeichnis enthaltenen Dateien. Der zweite Befehl `sudo chmod -R 777 ~` wird dazu benutzt, um darauffolgend die Zugriffsrechte aller Dateien des aktuellen Home-Verzeichnisses zu modifizieren. Sind diese beiden Befehle erfolgreich ausgeführt, kann noch der Priorisierungsmechanismus, wie in Kapitel 8.3 beschrieben, aktiviert werden. Um die Ausführungsumgebung zu starten, ist es nötig, den Befehl `sudo ./run_demos.sh` auszuführen. Wird diese gestartet, ist es auch möglich, die in Kapitel 8.2 beschriebenen Programme erfolgreich zu nutzen.

¹⁰Das GitLab-Repository, zu finden unter dem Namen `LonTalk-Router`, ist für alle internen Nutzer*innen des Studienganges ITS einsehbar.

9 Diskussion der Realisierungsergebnisse und der Forschungsfrage

In diesem Kapitel werden die Ergebnisse und Schlussfolgerungen thematisiert, die sich aus der Implementierung des LonTalk-Routers ergeben. Außerdem wird die in Kapitel 1.2 dargelegte Fragestellung, die bisher nicht thematisiert wurde, behandelt.

9.1 Diskussion der Realisierungsergebnisse

Dieses Kapitel wird genutzt, um die Ergebnisse der in Kapitel 8 vorgestellten Implementierungsarbeiten zu diskutieren und aufzuzeigen. Kernelemente dieses Kapitels sind daher Errungenschaften und Misserfolge, die bei den Entwicklungstätigkeiten zutage getreten sind. In diesem Kapitel wird außerdem ein Schwerpunkt auf die Limitierungen und Vorteile der getätigten Realisierung gelegt.

9.1.1 Gewonnene Erkenntnisse

Zusammenfassend werden in diesem Kapitel die wichtigsten Erkenntnisse der gesamten Implementierungsarbeiten dargelegt.

Der zentralste Punkt, der seine Umsetzung fand, betrifft das Weiterleitungsverhalten des LonTalk-Routers. Für dieses Gerät war es möglich, das grundlegende Forwarding, das durch einen Configured Router auf der OSI-Schicht drei des LonTalk-Protokolls bereitgestellt wird, zu implementieren. Um das Weiterleitungsverhalten zu testen, war es nötig, auch das Versenden und Empfangen von LonTalk-Nachrichten zu implementieren. Dieses Ziel wurde, wie in Kapitel 8.2 dargelegt, durch Verwendung von Scapy und Python erreicht. Die daraus resultierenden Programme können innerhalb der Topologie verwendet und daher auch von dem/der Nutzer*in dazu genutzt werden, die Implementierung des Routers ausführlich zu testen. Erst die Entwicklung und Nutzung dieser Programme machte eine Untersuchung und eine allfällige Anpassung des implementierten Weiterleitungsverhaltens möglich. Diese Programme dienten auch dazu, eine stärkere Auseinandersetzung mit den Inhalten eines LonTalk-Frames zu fördern. Ein weiteres Ergebnis der Implementierungsarbeiten ist der Entwurf und die Umsetzung einer funktionalen Topologie, die die Einbettung und damit auch die Testung des LonTalk-Routers in einem Netzwerk ermöglicht. Um diese Topologie zu erstellen, wurde ein simples Bash-Skript erstellt. Dieses bietet auch die Möglichkeit, Modifizierungen vorzunehmen, um damit an die Bedürfnisse des Nutzers beziehungsweise der Nutzerin angepasst zu werden. In diesem Bash-Skript ist auch die

Konfiguration des Routers behandelt. Um diese durchführen zu können, ist es gelungen, zwei verschiedene Konfigurationsoptionen zu integrieren: Einerseits die Konfiguration über eine CLI und andererseits die Nutzung der P4Runtime-Technologie zusammen mit einem auf Python basierenden Controller.

9.1.2 Einschränkungen und Limitierungen der Realisierung

In diesem Kapitel werden Einschränkungen und Limitierungen aufgeführt, die durch die Implementierung und die gewählte Herangehensweise des gesamten Aufbaus bedingt sind.

Eine Limitierung ist beispielsweise, dass die neueste Version des Software-Switches und des dazugehörigen Compilers nicht genutzt wurde. Diesbezüglich wurde, jedoch bereits in Kapitel 7.4 erwähnt, dass dies ein zu vernachlässigender Faktor ist und daher keine weiteren größeren Auswirkungen hat. Ein weiterer Aspekt ist, dass sich durch die Nutzung des V1-Modells die Portabilität des P4-Programms deutlich verringert. Dies liegt vor allem darin begründet, dass sich das Programm oftmals auf für dieses Architekturmodell spezifischen intrinsischen Metadaten stützt. Beispiele dafür sind die Nutzung von `egress_spec`, `mcast_grp` und `priority`. Außerdem wird durch die verwendete Architektur auch eine vordefinierte Programmabfolge (siehe Listing 28) vorgeschrieben, die auch eingehalten werden muss. Daraus ergibt sich, dass das P4-Programm adaptiert werden müsste, damit ein anderes Zielgerät, das das genutzte Architekturmodell nicht unterstützt, mit derselben Programmfunctionen betrieben werden kann. Durch die Nutzung einer unter LonTalk nicht üblichen Übertragungstechnologie wurden in das P4-Programm Headerfelder integriert, die bei der Verwendung eines für LonWorks üblicheren Mediums nicht nötig gewesen wären. Dies bedeutet, dass das P4-Programm mit der Wahl, kein Bus-System, sondern Ethernet zu verwenden, auf die verwendete Topologie angepasst wurde. Würde nunmehr eine üblichere Übertragungstechnologie genutzt, müsste das Programm modifiziert und neue Versende- beziehungsweise Empfangsprogramme konzipiert werden. Eine weitere Einschränkung geht mit dem Entwurf einher, dass der Router den Nachrichtenverkehr zwischen Domänen einschränken kann. Diese Einschränkung betrifft dabei die Konfiguration des Routers, denn wird diese nicht korrekt vorgenommen, kann es geschehen, dass Nachrichten von ganzen Segmenten, die an den Router angeschlossen sind, von diesem nicht weitergeleitet, sondern verworfen werden. Ein Aspekt, der durch den/-die Leser*in als Einschränkung angesehen werden kann, jedoch kein Ziel dieser Arbeit ist, ist, dass die Implementierung der Scapy-Programme nicht komplett ist und nicht alle Schichten des LonTalk-Protokolls abbildet. Beispielhaft ist hier die fehlende Implementation der OSI-Schichten vier bis sieben zu nennen. Darüber hinaus werden in diesen

Programmen grundlegende, aber für diese Arbeit nicht ausschlaggebende und teilweise für das Debugging sogar hilfreiche Features nicht implementiert. Beispielhaft dafür steht, die Überprüfung ob ein am Host eingehendes Paket an diesen adressiert ist. Diese wurde nicht implementiert, weil die Implementierung der Hosts nicht im Fokus dieser Arbeit liegt, sodass jeder Host jede von ihm empfangene Nachricht anzeigen kann. Eine weitere einschränkende Vorbedingung, die für die gesamte Implementierung getroffen wurde, ist, dass davon ausgegangen wird, dass alle Teilnehmer der Topologie bereits mit Adressen konfiguriert sind. Diese Annahme stellt jedoch nur dann einen Nachteil dar, wenn im zugrundeliegenden Netzwerk Nachrichten ohne Subnetzinformationen versendet werden würden. Denn enthalten die in einem unkonfigurierten Netz ausgetauschten Nachrichten eine Zielsubnetzinformation und wurden die Weiterleitungstabellen des Routers bereits korrekt konfiguriert, ist die volle Funktionsfähigkeit des implementierten Routers gegeben. Eine Einschränkung, welche die verwendete Plattform betrifft, dass für einen Broadcast auf allen Interfaces, die derselben Domäne zugerechnet werden können, pro Domäne eine Multicastgruppe erstellt werden muss. Dies ist durch die Realisierung von Multicasting innerhalb des BMv2-Software-Switches bedingt. Details zu der Verwendung von Multicasting mithilfe des verwendeten Switches finden sich in Kapitel 8.3. Eine Einschränkung des Programms selbst ist, dass für drei von vier möglichen Domänenlängen ein eigener Header erstellt werden musste. Dies ist notwendig, da die Nutzung eines für alle Domänenlängen gemeinsamen und durch den Datentyp `varbit` variablen Headers nicht möglich ist. Laut [3] liegt diese Limitierung darin begründet, dass dieser Dateityp nicht mit allen Operationen genutzt werden kann. Beispielsweise war es nicht möglich, diesen direkt an eine Tabelle zu übergeben. Ein weitere Limitierung wurde bereits in Kapitel 8.4.4 erläutert. In diesem Kapitel wurde ausgeführt, dass bei dem genutzten Software-Switch hinsichtlich der Priorisierung von einzelnen Nachrichten nicht überprüft werden kann, ob diese Priorisierung auch tatsächlich durchgeführt wird, weil abgesehen von Einträgen in die zum Software-Switch gehörige Log-Datei keine eindeutige Rückmeldung an den/die Programmierer*in erfolgt.

9.1.3 Vorteile der Realisierung

In diesem Kapitel werden Vorteile aufgeführt, die sich durch die vorgenommene Implementierung und die anfangs ausgewählte Herangehensweise des Aufbaus bedingt sind.

Ein Vorteil des Routers ist, dass dieser durch die in Kapitel 8.3 vorgestellte Whitelist die Möglichkeit hat, Nachrichten bestimmter Domänen und Interfaces nicht weiterleiten zu müssen. Dies hat zur Folge, dass an den Router angeschlossene Netzsegmente entlastet werden. Würde dieser Mechanismus nicht bestehen, würden durch den Router auch

Pakete weitergeleitet werden, die an Hosts einer anderen Domäne adressiert sind. Dies hätte wiederum zur Folge, dass diese Pakete erst an den empfangenden Hosts und nicht an dem auf dem Nachrichtenweg voranstehenden Router verworfen werden würden. Ein weiterer Vorteil, der durch die Implementierung und den genutzten Software-Switch entsteht ist, dass mehr Segmente an einen Router angeschlossen werden können. Laut [76] sind an einen jeden LonTalk-Router immer lediglich exakt zwei Segmente angeschlossen, die durch den Router getrennt werden. Die Zahl an angeschlossenen Segmenten kann durch den in dieser Implementierung genutzten Software-Switch erheblich erhöht werden, sodass die Nutzung dieses Routers die Möglichkeit bietet, flexiblere Topologien zu nutzen. Allgemein wird durch die Nutzung von P4 und die damit ermöglichte Programmierbarkeit der Data-Plane eines LonTalk-Routers eine Flexibilisierung des Forwardings erreicht. Dies eröffnet beispielsweise die Möglichkeit, neben der Domänen-, Subnetz- und Gruppeninformation auch die Unique ID oder andere Paketinformationen für ein detailliertes Forwarding von Paketen zu nutzen. Auch könnte durch die Programmierbarkeit des Routers eine Blockierung von bestimmten Nachrichten beziehungsweise Nachrichtenhalten oder auch eine Verschlüsselung von LonTalk-Nachrichten zwischen zwei Routern auf Schicht zwei erreicht werden. Die Nutzung von P4 bietet daher auch für das Kommunikationsprotokoll LonTalk sehr viele Möglichkeiten für eine innovative Weiterentwicklung des Protokolls.

9.2 Implementierbarkeit des LonTalk-Protokolls

In diesem Kapitel wird die noch nicht behandelte Fragestellung des Kapitels 1.2 thematisiert und auf Basis des Wissens, dass durch die Implementierungsarbeiten dieser Masterthesis gewonnen wurde, beantwortet. Durch die Fragestellung wird diskutiert, ob es möglich ist, das LonTalk-Kommunikationsprotokoll durch Zuhilfenahme von P4 selbst zu implementieren. Außerdem wird bei dieser Frage präzisierend behandelt, ob die öffentlich zugänglichen Protokollbeschreibungen ausreichen, um eine Implementierung von LonTalk durch P4 zu ermöglichen.

Um diese Frage zu beantworten, ist es nötig, zu wissen, für welche Zwecke das LonTalk-Protokoll implementiert werden soll. Dies liegt darin begründet, dass für die Implementierung eines LonTalk-Endgeräts andere Netzwerkfunktionen umgesetzt werden müssen, als dies für einen LonTalk-Router der Fall ist. Wurde eine Wahl getroffen, welche Gerätefunktion man umsetzen möchte, muss nunmehr entschieden werden, welches P4-Zielgerät man für diese Implementierung heranziehen möchte. Stellt sich die Aufgabe, einen physischen LonTalk-Router umzusetzen, würde dies vermutlich mehr Aufwand und damit Entwicklungszeit bedeuten, als bei der Entwicklung eines auf Software basierenden LonTalk-Routers. Dies begründet sich vor allem in den Problemen, die mit einem physischen Gerät

einhergehen, welche bereits in Kapitel 7.1 erläutert wurden. Sollte nunmehr ein LonTalk-Router auf Basis eines Software-Switches, wie es in dieser Arbeit geschehen ist, entwickelt werden, ist dies mit weniger Problemen verbunden. In dem für diese Arbeit erstellten P4-Programm, dessen Inhalte in Kapitel 8.4 beschrieben sind, wurde die Implementierbarkeit der Schichten zwei und drei eines Configured Router untersucht. Das für die vorgenommene Implementierung notwendige Wissen stützt sich auf mehrere Quellen. Unter diesen Quellen befinden sich unter anderem Dokumente, die von einem Hersteller [10] und einer anderen dem LonTalk-Protokoll verschriebenen Organisation [65] verfasst wurden. Der Schwerpunkt dieser Dokumente liegt allerdings meist nicht auf der Implementierung und Beschreibung des Geräteverhaltens selbst. Die Implementierung und Beschreibung des Geräteverhaltens ist jedoch ein Schwerpunkt, der in dem Standardisierungsdokument [76] gesetzt wurde, denn darin befindet sich unter anderem eine Referenzimplementierung eines LonTalk-Endknotens. Das für diese Arbeit interessante Verhalten eines Routers wird in diesem Dokument allerdings nicht behandelt, sodass aus der Gesamtheit aller in den Quellen enthaltenen Informationen eine These betreffend des Weiterleitungsverhaltens synthetisiert werden musste (siehe Kapitel 6.2). Daraus kann geschlossen werden, dass die verfügbaren Informationen für die Implementierung eines Configured Routers anhand der Programmiersprache P4 nicht ausreichen. Dass für diese Masterarbeit die Zuhilfenahme einer These verwendet wird, birgt die Gefahr, dass diese widerlegt werden könnte. Dies würde bedeuten, dass dadurch die in dieser Arbeit vorgenommene P4-Implementierung inkorrekt ist. Selbst wenn dieser Fall nie eintreten sollte, gibt es einige Probleme, die einer vollständigen Implementierung eines LonTalk-Router mittels P4 innerhalb dieser Arbeit im Wege stehen. Einige dieser Probleme wurden bereits im Kapitel 9.1.2 ausführlich dargelegt. Auf Basis der in dieser Arbeit vorgenommenen Implementierung kann zum Schluss gekommen werden, dass die Umsetzung des LonTalk-Protokolls auf Basis eines Configured Routers teilweise, aber nicht vollständig möglich ist.

Aus den in dieser Implementierung gewonnenen Erkenntnissen kann außerdem geschlussfolgert werden, dass die Implementierbarkeit des LonTalk-Protokolls mit P4 von einigen variablen Umständen abhängig ist. Allem voran steht dabei die Kenntnis über das Verhalten und die korrekte Implementierung des LonTalk-Protokolls. Ein weiterer Faktor, von dem die Implementierung des LonTalk-Protokolls abhängt, ist, der zu realisierende Gerätetyp und das dafür verwendete P4-Zielgerät. Steht für das verwendete Zielgerät die Nutzung eines dem Protokoll angepassten Architekturmodells zur Verfügung und sind in diesem Modell die für das Protokoll nötigen externen Objekte, die essentielle Protokollfunktionen umsetzen, bereits enthalten, so kann zum vorliegenden Erkenntnisstand davon ausgegangen werden, dass eine standardkonforme Umsetzung möglich ist.

10 Schlussfolgerungen und Aussichten

In diesem Kapitel werden die wichtigsten Inhalte dieser Arbeit zusammengefasst dargestellt. Außerdem werden die wichtigsten Erkenntnisse dieser Arbeit vorgestellt. Den Abschluss dieses Kapitels bildet ein Ausblick auf etwaige Weiterentwicklungsmöglichkeiten, die auf den Erkenntnissen dieser Arbeit basieren.

10.1 Zusammenfassung

Der Hauptzweck dieser Arbeit war zu zeigen, ob eine selbstständige Implementierung des LonTalk-Kommunikationsprotokoll durch Zuhilfenahme von P4 möglich ist. Dazu wurde im ersten Kapitel die Forschungsthematik, die zugrundeliegende Problemstellung und die für diese Arbeit zentralen Fragestellungen vorgestellt. Dabei wurde auch erstmals die in dieser Masterarbeit verwendete Programmiersprache einführend erwähnt, die dazu genutzt wird, um LonTalk auf einer Data-Plane zu realisieren. Das Ergebnis dieses Kapitels ist eine Beschreibung der Ziele, in Form von Forschungsfragen, die in dieser Arbeit behandelt werden. Die formulierten Forschungsfragen wurden im Laufe dieser Masterarbeit thematisiert und beantwortet. Das Kapitel 2 stellte den Begriff Data-Plane und deren Programmierung vor. Daraufhin wurde in Kapitel 3 der aktuelle Stand der Technik und Technologien aufgeführt, die eine konzeptionelle Ähnlichkeit mit P4 aufweisen. Die als Grundstein für diese Arbeit dienende Technologie P4 wurde im Kapitel 4 detailliert auf Basis einer umfassenden Literaturrecherche präsentiert. Dabei wird geklärt, welche Rolle P4 erfüllen soll und welche Gründe zur Entwicklung dieser Programmiersprache führten. Außerdem wurde thematisiert, welchen Nutzen und welche Vorteile sich aus der Nutzung von P4 ergeben. Zusätzlich wurden die verschiedenen Evolutionsstufen der Sprache vorgestellt. Ergänzend dazu wurde außerdem das Zusammenspiel der für ein P4-Programm nötigen Komponenten dargelegt. Dies hatte zum Ziel, dass der/die Leser*in den Zusammenhang zwischen den daraufhin vorgestellten Thematiken Architekturmodelle, Programm-Compiler und P4Runtime versteht. Ferner wurden die wichtigsten in P4 nutzbaren Sprachbestandteile anhand von einfachen Beispielen vorgestellt. Kapitel 5 widmete sich dem LonTalk-Protokoll und der dazugehörigen Technologie LonWorks und veranschaulichte unter anderem das Format von LonTalk-Nachrichten auf den OSI-Schichten zwei und drei. In Kapitel 6 werden vorbereitend auf die praktische Implementierung dieser Masterarbeit die Fragen beantwortet, warum LonTalk für die Implementierung herangezogen wird und welches LonWorks-Gerät als Basis für diese Implementierung dient. Weitere Implementierungsentscheidungen werden in Kapitel 7 festgehalten. Dabei wird in diesem Kapitel vor allem argumentiert, wie der gesamte Testaufbau, in dem die Implementierung des LonTalk-Protokolls stattfand, aufgebaut und

realisiert werden sollte. Kapitel 8 zeigte die Ergebnisse der Implementierungsarbeiten und wie diese tatsächlich umgesetzt wurden. Außerdem wird in diesem Kapitel noch der Ablauf der Testung des LonTalk-Routers dargelegt. Abschließend wurde in diesem Kapitel angeführt, wie die genutzte Entwicklungsplattform modifiziert wurde, sodass diese auch von dem/der Leser*in reproduziert werden kann. In Kapitel 9 werden daraufhin die Ergebnisse der Implementierung diskutiert und die gewonnenen Erkenntnisse vorgestellt. Überdies wird in diesem Kapitel die noch ausstehende Forschungsfrage beantwortet.

10.2 Wichtigste Erkenntnisse

Dieser Abschnitt steht zusammenfassend für die in Kapitel 9 detailliert aufgeführten Erkenntnissen, die durch die Implementierungstätigkeiten ersichtlich wurden.

Das zentrale Ergebnis dieser Masterarbeit ist die Umsetzung eines LonTalk-Routers. Dies zeigte sich durch die Möglichkeit, grundlegendes Forwarding eines Routers auf Basis des LonTalk-Protokolls mit P4 zu implementieren. Ein weiteres zentrales Ergebnis ist, dass die Testung und Einbettung des Routers durch die Erstellung einer rudimentären Topologie möglich sind. Die Einschränkung der Portabilität von P4-Programmen durch das verwendete Architekturmodell, ist ebenfalls ein wichtige Erkenntnis dieser Arbeit. In der vorgenommenen Implementierung wird dies vor allem dadurch sichtbar, dass intrinsische Metadaten zur Anwendung kommen, die lediglich unter dem genutzt V1-Modell verwendet werden können. Eine Tatsache, die Auswirkung auf den Versuchsaufbau hatte, ist, dass kein P4-programmierbares Gerät verwendet wurde, das perfekt auf die Bedürfnisse eines LonTalk-Routers zugeschnitten ist. Dies drückt sich zum Beispiel dadurch aus, dass keine für LonWorks übliche Übertragungstechnologie genutzt werden konnte, sodass eine mit dem genutzten Software-Switch verwendbare Übertragungstechnologie verwendet wurde. Dies resultierte wiederum darin, dass diese Übertragungstechnologie auch zentraler Bestandteil des Versuchsaufbaus wurde. Wie sich während der Implementierungsarbeiten herausstellte, sind nicht alle vorhanden Features des verwendeten BMv2-Switches vollständig ausgereift. Stellvertretend dafür steht der Mechanismus, der für die Priorisierung von einzelnen Paketen genutzt wird. Hier ist beispielsweise nicht eindeutig ersichtlich, ob dieser Mechanismus voll funktionsfähig ist. Nichtsdestotrotz stellte sich durch die in dieser Masterarbeit umgesetzt Programmierung einer Data-Plane heraus, dass P4 ein mächtiges Werkzeug ist und dazu genutzt werden kann, auch der klassischen Netzwerkdomäne fremde Protokolle umzusetzen.

10.3 Ausblick auf mögliche Weiterentwicklungen

Diese Masterarbeit sollte als Basis für weitere tiefergehende Forschungsbemühungen im Bereich der Programmierung von Data-Planes und einer kompletten Implementierung von LonTalk angesehen werden. Damit etwaige Forschungsbemühungen auf den Forschungsergebnissen dieser Arbeit aufbauen können, ist sicherzustellen, dass die Erkenntnisse und Ergebnisse dieser Masterarbeit dokumentiert und reproduzierbar sind.

Allem voran ist das Ausloten von Grenzen und Möglichkeiten im Zusammenhang mit frei verfügbaren P4-Werkzeugen und Zielgeräten ein Forschungsfeld, dass weiter behandelt werden sollte. Durch weitere Forschungstätigkeit könnte auch ein Beitrag zur Weiterentwicklung der noch jungen Programmiersprache P4 geleistet werden. Ein mögliches Forschungsfeld, das P4-Zielgeräte betrifft, ist die Konzeption beziehungsweise Entwicklung eines Software-Switches, dessen Augenmerk auf P4-Programmierbarkeit, Containerisierung, Effizienz, Stabilität und Wartbarkeit liegt. Ein Beispiel für weitere Untersuchungen, das sowohl mit dieser Masterarbeit als auch mit einem frei verfügbaren P4-Zielgerät in Zusammenhang steht, ist die Analyse des durch den BMv2-Switch bereitgestellten Priorisierungsmechanismus. Hier könnte neben einer Analyse der Funktionsweise beispielsweise auch untersucht werden, wie dieser Mechanismus modifiziert werden könnte.

Überdies gibt es Forschungsthemen, die sich auf die Implementierung dieser Masterarbeit stützen und darauf aufbauen könnten. Beispielsweise wäre es interessant, ein physisches Testnetz aufzubauen, dessen Kern eine Hardwareimplementierung des auf P4 basierenden LonTalk-Routers bildet. Dieses Testnetz könnte in einem weiteren Schritt auch zertifizierte LonTalk-Hardware enthalten, die für das Erstellen und Empfangen von LonTalk-Nachrichten genutzt wird. Eine weitere Möglichkeit auf der Realisierung dieser Masterarbeit aufzubauen, wäre die Erweiterung der P4-Implementierung, sodass der LonTalk-Router auch an ihn selbst adressierte Nachrichten bearbeiten und darauf reagieren kann. Hierfür würde sich vor allem der Authentifizierungsmechanismus des LonTalk-Protokolls anbieten.

In diesem Zusammenhang könnte auch untersucht werden, wie die Entwicklung von externen Objekten beziehungsweise Funktionen für den BMv2-Switch vereinfacht werden könnte. Dies könnte dazu dienen, den Funktionsumfang und die Weiterentwicklung des besagten Zielgeräts langfristig zu erhöhen und zu beschleunigen. Überdies konnte versucht werden, eine Erweiterung des genutzten Software-Switches vorzunehmen, sodass dieser besser auf die Anforderungen des LonTalk-Protokolls ausgerichtet ist und trotzdem den grundlegenden P4-Support bietet. Diesbezüglich könnte der Funktionsumfang des Switches angepasst und die Nutzung von für LonWorks üblichen Übertragungstechnologien ermöglicht werden. Bei dieser Forschungstätigkeit könnte beispielsweise versucht werden, eine direkte Konfigurierbarkeit der Weiterleitungstabellen eines BMv2-Switches aus dem P4-Programm heraus

umzusetzen. Dies würde eine LonTalk-konforme Konfiguration des Switches ermöglichen, die mit dem BMv2-Switch zum momentanen Stand nicht möglich ist. Außerdem wäre es interessant zu untersuchen, ob es durch die Abänderung des BMv2-Switches möglich wäre, den Trailer einer Nachricht zu verarbeiten, da P4 für dies nicht ausgelegt ist [103]. Durch eine solche Funktionalität könnte es darauffolgend auch möglich sein, die in einem LonTalk-Paket auf OSI-Schicht zwei enthaltene CCITT CRC-16-Checksumme zu berechnen. [3] legt nahe, dass die Berechnung und Validierung einer solchen im Trailer befindlichen, Checksumme durch nicht-programmierbare Teile des verwendeten Zielgeräts geschieht.

Abkürzungsverzeichnis

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

BMv2 Behavioral Model Version 2

CAN Controller Area Network

CAN FD Controller Area Network Flexible Data-Rate

CNP Control Network Protocol

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

DNS Domain Name System

DPI Deep Packet Inspection

eBPF extended Berkeley Packet Filter

ForCES Forwarding and Control Element Separation

FPGA Field Programmable Gate Array

gRPC google Remote Procedure Call

IAB Internet Architecture Board

IETF Internet Engineering Task Force

IP Internet Protocol

IPv4 Internet Protocol Version 4

IPv6 Internet Protocol Version 6

IR Intermediate Representation

IRTF Internet Research Task Force

ITS Informationstechnik und Systemmanagement

JSON JavaScript Object Notation

LLVM Low Level Virtual Machine

LON Local Operating Network

MAC Media Access Control

MACSAD Multi-Architecture Compiler System for Abstract Dataplanes

MMT Multiple Match Table

MSB Most Significant Bit

MTU Maximum Transmission Unit

NETCONF Network Configuration Protocol

NIC Network Interface Card

NPDU Network Protocol Data Unit

ODP Open Data Plane

OSI Open System Interconnect

P4 Programming Protocol-Independent Packet Processors

PCIe Peripheral Component Interconnect Express

PDU Protocol Data Unit

PISA Protocol Independent Switch Architecture

PLC Power Line Communication

POF Protocol-Oblivious Forwarding

PPDU Physical Protocol Data Unit

PSA Portable Switch Architecture

RESTCONF Representational State Transfer Configuration Protocol

RFC Request for Comments

RMT Reconfigurable Match Table

SDN Software Defined Networking

SNVT Standard Network Variable Types

SoC System on a Chip

STP Spanning Tree Protocol

TCAM Ternary Content-Addressable Memory

TCP Transmission Control Protocol

TPU Tensor Processing Unit

VLAN Virtual Local Area Network

VM Virtuelle Maschine

VXLAN Virtual Extensible Local Area Network

Abbildungsverzeichnis

1	Schichten der SDN-Architektur [1]	2
2	Schichten des OSI-Modells nach [13]	6
3	Architekturelle Bausteine traditioneller Netzwerkswitches [15]	7
4	Beispiele in der Informationstechnik für die Programmierbarkeit von „White Box“-Hardware [36]	12
5	Strukturelle Veränderungen zwischen P4 ₁₄ und P4 ₁₆ nach [3]	16
6	Schema des P4-Workflows [3]	17
7	Schemenhafte Darstellung des PISA-Modells nach [15]	20
8	Schemenhafte Darstellung des P4 ₁₄ -Switch-Modells nach [15]	21
9	Schemenhafte Darstellung einer programmierbaren SmartNIC-Architektur nach [15]	22
10	Schemenhafte Darstellung des PSA-Modells nach [51]	23
11	Illustration der „Paket-Pfade“ im PSA-Modell [51]	24
12	Schematische Darstellung des Datenflusses des P4 ₁₆ -Referenzcompilers nach [55]	26
13	Übersicht über P4Runtime-Komponenten nach [60]	29
14	„Match+Action“-Prozess nach [3]	35
15	Aufbau eines LonWorks-Netzes	40
16	LonTalk-Nachricht nach [10]	44
17	LonTalk-NPDU nach [10]	45
18	Adressformat „Subnet Broadcast“ nach [10]	46
19	Adressformat „Group“ nach [10]	47
20	Adressformat „Subnet/Node“ nach [10]	47
21	Adressformat „Group Acknowledgement“ nach [76]	47
22	Adressformat „Unique ID“ nach [10]	48
23	Aufbau der Topologie	67
24	Adressformat „Group Acknowledgement“ nach [10]	68
25	Übersicht über die Header-Klassen des Nachrichtenerstellungs-Programms	78
26	Durch ein offizielles Compiler-Backend erstelltes Zustandsübergangsdiagramm	88
27	Inhalt einer Testnachricht von H12 an H22	100
28	Inhalt einer Testnachricht von H12 an 11 (Sender)	101
29	Inhalt einer Testnachricht von H12 an 11 (Empfänger)	103

Tabellenverzeichnis

1	EN 50065 Frequenzen [66]	38
2	LonTalk-Ebenen nach [65]	42
3	Übersicht über das Packet Format-Feld nach [10]	45
4	Übersicht über das Address Format-Feld nach [10]	46
5	Übersicht über das Length-Feld nach [10]	46

Listings

1	Präprozessor-Befehle [3]	30
2	Integer-Literale	31
3	Einige grundlegende Datentypen	31
4	Beispiele für abgeleitete Datentypen [7]	32
5	Beispiele für einen Parser [7]	33
6	Ausschnitt eines Control-Blocks [7]	33
7	Deparser-Blocks [7]	35
8	Main-Deklaration [7]	36
9	p4app.json-Datei enthalten in [86]	58
10	JSON-Konfigurationsdatei enthalten in [90]	60
11	Erstellung der Network Namespaces für die Endgeräte H11 und H12	71
12	Aktivierung der Loopback-Interfaces für die Endgeräte H11 und H12	71
13	Erstellung der Übertragungswege für die Endgeräte H11 und H12	72
14	Zuweisung der erstellten Interfaces zu den Endgeräten H11 und H12	72
15	Deaktivierung von IPv6 auf den erstellten Interfaces	72
16	Aktivierung der erstellten Netzwerkinterfaces	73
17	Kompilierung des P4-Programms	73
18	Starten des Software-Switches	73
19	Konfigurationsoptionen des Software-Switches	74
20	Start von Eingabefenstern für die Endgeräte H11 und H12	75
21	Definition einiger Header entnommen aus der Datei send.py	77
22	Erstellung des domänenspezifischen Paketbestandteils in send.py	79
23	Erstellung von Headern der Adressformate „Subnet Broadcast“ und „Group“	79
24	Versenden einer LonTalk-Nachricht mittels send.py	80
25	Ausschnitte des Programms receive.py	81
26	Präprozessor-Direktive deren Kommentar zu entfernen ist	82
27	Für Aktivieren des Priorisierungsmechanismus nötige Programmzeilen	83
28	Programmierbare architekturelle Bestandteile des V1-Modells	84
29	Einige wiederverwendbare Datentypen der P4-Implementierung	85
30	Einige Header der P4-Implementierung	86
31	struct-Definitionen innerhalb des P4-Programms	86
32	Überprüfung des Parsers hinsichtlich Routeradressierung	88
33	action-Abschnitte der Ingress-Pipeline	89
34	Definition der Tabelle port_domain_whitelist_exact	90
35	Behandlung der Domäneninformation in der Ingress-Pipeline	91
36	Behandlung der Subnetz- und Gruppeninformation in der Ingress-Pipeline	92
37	Behandlung des Pri-Feldes in der Ingress-Pipeline	93
38	Deparser der Router-Implementierung	94
39	Definition der Debugging-Tabelle in debug.p4	94
40	Diverse die Konfiguration ermöglichte Programmteile aus mycontroller.py	96
41	Erstellen eines Tabelleneintrags in mycontroller.py	96
42	Erstellen eines Multicastgruppen-Eintrags in mycontroller.py	97
43	Teil einer Tabellenkonfiguration in lontalk.json	97
44	Teil einer Tabellenkonfiguration in lontalk.txt	97
45	Definition der Multicastgruppe drei in lontalk.json	98
46	Definition der Multicastgruppe drei in lontalk.txt	98

47	Log-Datei zeigt Aktivierung des Priorisierungsmechanismus an	99
48	Log-Datei zeigt Ausschnitt der Verarbeitung einer Testnachricht von H12 an H22	100
49	Log-Datei zeigt Ausschnitt der Verarbeitung einer Testnachricht von H12 an H11	102

Literatur

- [1] W. Braun und M. Menth, „Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices,“ *Future Internet*, Jg. 6, Nr. 2, S. 302–336, 2014. DOI: 10.3390/fi6020302.
- [2] E. Haleplidis, K. Pentikousis, S. Denazis et al., „RFC 7426: Software-Defined Networking (SDN): Layers and Architecture Terminology,“ RFC Editor, RFC 7426, 2015. Adresse: <https://tools.ietf.org/rfc/rfc7426.txt>.
- [3] *P4_16 Language Specification v1.2.0*, Mai 2018. Adresse: <https://p4.org/p4-spec/docs/P4-16-v1.2.0.html> (besucht am 03.02.2020).
- [4] N. McKeown, T. Anderson, H. Balakrishnan et al., „OpenFlow: Enabling Innovation in Campus Networks,“ *ACM SIGCOMM Computer Communication Review*, Jg. 38, Nr. 2, S. 69, März 2008. DOI: 10.1145/1355734.1355746.
- [5] J. Saldana, F. Pascual, D. De Hoz et al., „Optimization of low-efficiency traffic in OpenFlow Software Defined Networks,“ in *Proceedings of the 2014 International Symposium on Performance Evaluation of Computer and Telecommunication Systems, SPECTS 2014 - Part of SummerSim 2014 Multiconference*, Juli 2014, S. 550–555. DOI: 10.1109/SPECTS.2014.6879992.
- [6] N. McKeown, *Programmable forwarding planes are here to stay*. Adresse: <https://conferences.sigcomm.org/sigcomm/2017/workshop-netpl.html> (besucht am 18.01.2020). *Vortrag auf dem ACM SIGCOMM 2017 The Third Workshop on Networking and Programming Languages (NetPL 2017)*.
- [7] M. Budiu und C. Dodd, „The P4_16 programming language,“ in *ACM SIGOPS Operating Systems Review*, Bd. 51, Association for Computing Machinery, Sep. 2017, S. 5–14. DOI: 10.1145/3139645.3139648.
- [8] J. W. Lockwood, N. McKeown, G. Watson et al., „NetFPGA—An Open Platform for Gigabit-Rate Network Switching and Routing,“ in *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, IEEE, Juni 2007, S. 160–161. DOI: 10.1109/MSE.2007.69. Adresse: <http://ieeexplore.ieee.org/document/4231497/>.
- [9] P. Benacek, „Generation of High-Speed Network Device from High-Level Description,“ Diss., Czech Technical University in Prague, Prague, 2016. DOI: 10.5121/ijngn.2012.4307.
- [10] *Introduction to the LonWorks® Platform*. Adresse: https://www.echelon.com/assets/blt893a8b319e8ec8c7/078-0183-01B_Intro_to_LonWorks_Rev_2.pdf (besucht am 18.01.2020).
- [11] P. Bosshart, D. Daly, G. Gibb et al., „P4: Programming Protocol-Independent Packet Processors,“ *SIGCOMM Comput. Commun. Rev.*, Jg. 44, Nr. 3, S. 87–95, Juli 2014. DOI: 10.1145/2656877.2656890. Adresse: <https://doi.org/10.1145/2656877.2656890>.
- [12] The P4 Language Consortium, *P4 Language and Related Specifications*. Adresse: <https://p4.org/specs/> (besucht am 09.03.2020).

- [13] Y. Li, D. Li, W. Cui et al., „Research based on OSI model,“ in *2011 IEEE 3rd International Conference on Communication Software and Networks*, Mai 2011, S. 554–557. DOI: 10.1109/ICCSN.2011.6014631.
- [14] J. D. Day und H. Zimmermann, „The OSI reference model,“ *Proceedings of the IEEE*, Jg. 71, Nr. 12, S. 1334–1340, Dez. 1983. DOI: 10.1109/PROC.1983.12775.
- [15] V. Gurevich, *P4_16 Introduction*, 2017. Adresse: https://www.youtube.com/playlist?list=PLxpC4CQ_nvdRbb65eMmC0pa4c0-PTvb0 – (besucht am 18.01.2020). *Vortrag auf dem P4 Developer Day 2017*.
- [16] S. Shenker, *The future of networking and the past of protocols*, Okt. 2011. Adresse: <https://www.youtube.com/watch?v=YHeyuD89n1Y> (besucht am 15.02.2020). *Vortrag auf dem Open Networking Summit 2011*.
- [17] E. Kohler, R. Morris, B. Chen et al., „The Click Modular Router,“ *ACM Trans. Comput. Syst.*, Jg. 18, Nr. 3, S. 263–297, Aug. 2000. DOI: 10.1145/354871.354874. Adresse: <https://doi.org/10.1145/354871.354874>.
- [18] J. Martins, M. Ahmed, C. Raiciu et al., „ClickOS and the Art of Network Function Virtualization,“ in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, Ser. NSDI’14, USA: USENIX Association, 2014, S. 459–473.
- [19] R. Duncan und P. Jungck, „PacketC language for high performance packet processing,“ in *2009 11th IEEE International Conference on High Performance Computing and Communications, HPCC 2009*, 2009, S. 450–457. DOI: 10.1109/HPCC.2009.89.
- [20] L. De Carli, Y. Pan, A. Kumar et al., „PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-Speed Routers,“ *SIGCOMM Comput. Commun. Rev.*, Jg. 39, Nr. 4, S. 207–218, Aug. 2009. DOI: 10.1145/1594977.1592593. Adresse: <https://doi.org/10.1145/1594977.1592593>.
- [21] C. Kozanitis, J. Huber, S. Singh et al., „Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System,“ in *2010 Proceedings IEEE INFOCOM*, März 2010, S. 1–9. DOI: 10.1109/INFCOM.2010.5462139.
- [22] H. Song, „Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane,“ in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, Ser. HotSDN ’13, New York, NY, USA: Association for Computing Machinery, 2013, S. 127–132. DOI: 10.1145/2491185.2491190. Adresse: <https://doi.org/10.1145/2491185.2491190>.
- [23] P. Bosshart, G. Gibb, H. S. Kim et al., „Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN,“ in *SIGCOMM 2013 - Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2013, S. 99–110. DOI: 10.1145/2486001.2486011.
- [24] X. Jin, X. Li, H. Zhang et al., „NetCache: Balancing Key-Value Stores with Fast In-Network Caching,“ in *Proceedings of the 26th Symposium on Operating Systems Principles*, Ser. SOSP ’17, New York, NY, USA: Association for Computing Machinery, 2017, S. 121–136. DOI: 10.1145/3132747.3132764. Adresse: <https://doi.org/10.1145/3132747.3132764>.

- [25] P. Vörös und A. Kiss, „Security Middleware Programming Using P4,“ in *Human Aspects of Information Security, Privacy, and Trust*, T. Tryfonas, Hrsg., Cham: Springer International Publishing, 2016, S. 277–287.
- [26] R. Miao, H. Zeng, C. Kim et al., „SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs,“ in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, Ser. SIGCOMM ’17, New York, NY, USA: Association for Computing Machinery, 2017, S. 15–28. DOI: 10.1145/3098822.3098824. Adresse: <https://doi.org/10.1145/3098822.3098824>.
- [27] Xilinx, *Xilinx - Adaptable. Intelligent*. Adresse: <https://www.xilinx.com/> (besucht am 20.03.2020).
- [28] Cisco, *Cisco – Österreich*. Adresse: https://www.cisco.com/c/de_at/index.html (besucht am 20.03.2020).
- [29] The P4 Language Consortium, *P4 Contributors*. Adresse: <https://p4.org/contributors/> (besucht am 09.03.2020).
- [30] Salzburg Research, *Salzburg Research Forschungsgesellschaft - Die Landesforschungsgesellschaft von Salzburg*. Adresse: <https://www.salzburgresearch.at/> (besucht am 20.03.2020).
- [31] Cornell University, *Cornell University*. Adresse: <https://www.cornell.edu> (besucht am 12.05.2020).
- [32] R. Enns, Juniper Networks, M. Björklund et al., „Network Configuration Protocol (NETCONF),“ RFC Editor, RFC 6241, 2011. Adresse: <https://www.rfc-editor.org/rfc/rfc6241.txt>.
- [33] A. Bierman, YumaWorks, M. Björklund et al., „RESTCONF Protocol,“ RFC Editor, RFC 8040, 2017. Adresse: <https://www.rfc-editor.org/rfc/rfc8040.txt>.
- [34] RFC Editor, *Number of RFCs Published per Year*. Adresse: <https://www.rfc-editor.org/rfcs-per-year/> (besucht am 11.03.2020).
- [35] D. D. Clark, J. Wroclawski, K. R. Sollins et al., „Tussle in cyberspace: defining tomorrow’s Internet,“ *IEEE/ACM Transactions on Networking*, Jg. 13, Nr. 3, S. 462–475, Juni 2005. DOI: 10.1109/TNET.2005.850224.
- [36] Barefoot Networks, *Barefoot Technology*. Adresse: <https://www.barefootnetworks.com/technology/> (besucht am 12.03.2020).
- [37] N. P. Jouppi, C. Young, N. Patil et al., „In-Datacenter Performance Analysis of a Tensor Processing Unit,“ *SIGARCH Comput. Archit. News*, Jg. 45, Nr. 2, S. 1–12, Juni 2017. DOI: 10.1145/3140659.3080246. Adresse: <https://doi.org/10.1145/3140659.3080246>.
- [38] Barefoot Networks, *Barefoot Networks Unveils Tofino™ 2, the Next Generation of the World’s First Fully P4-Programmable Network Switch ASICs*. Adresse: <https://www.barefootnetworks.com/press-releases/barefoot-networks-unveils-tofino-2-the-next-generation-of-the-worlds-first-fully-p4-programmable-network-switch-asics/> (besucht am 12.03.2020).
- [39] M. Irfan, Z. Ullah und R. C. C. Cheung, „D-TCAM: A High-Performance Distributed RAM Based TCAM Architecture on FPGAs,“ *IEEE Access*, Jg. 7, S. 96 060–96 069, 2019. DOI: 10.1109/ACCESS.2019.2927108.

- [40] *The reference P4 software switch.* Adresse: <https://github.com/p4lang/behavioral-model> (besucht am 16.03.2020).
- [41] The P4 Language Consortium, *P4→NetFPGA: A low-cost solution for testing P4 programs in hardware.* Adresse: <https://p4.org/p4/p4-netfpga-a-low-cost-solution-for-testing-p4-programs-in-hardware.html> (besucht am 04.03.2020).
- [42] N. Zilberman, *P4 Tutorial*, 2018. Adresse: https://github.com/p4lang/tutorials/raw/master/P4_tutorial.pdf (besucht am 01.03.2020).
- [43] C. Cascaval und D. Daly, *P4 Architectures.* Adresse: <https://www.youtube.com/watch?v=zEERS2bzij0> (besucht am 21.03.2020). *Vortrag auf dem P4 Workshop 2017.*
- [44] G. Brebner, *P4 Language Design Working Group Update*, 2017. Adresse: <https://www.youtube.com/watch?v=dFh0G-tJsgk> (besucht am 12.01.2020). *Vortrag auf dem P4 Workshop 2017.*
- [45] *The P4 Language Specification Version 1.0.5*, Nov. 2016. Adresse: <https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf> (besucht am 20.03.2020).
- [46] B. Pfaff, J. Pettit, T. Koponen et al., *The Design and Implementation of Open vSwitch*, 2015. Adresse: <https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf> (besucht am 10.01.2020).
- [47] Z. Hang, M. Wen, Y. Shi et al., „Programming protocol-independent packet processors high-level programming (P4HLP): Towards unified high-level programming for a commodity programmable switch,“ *Electronics (Switzerland)*, Jg. 8, Nr. 9, Sep. 2019. DOI: [10.3390/electronics8090958](https://doi.org/10.3390/electronics8090958).
- [48] Cisco, *Cisco Nexus 34180YC and 3464C Programmable Switches Data Sheet*, 2019. Adresse: <https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/datasheet-c78-740836.html> (besucht am 16.03.2020).
- [49] P. G. K. Patra, F. E. Cesen, J. S. Mejia et al., „Toward a sweet spot of data plane programmability, portability, and performance: On the scalability of multi-architecture P4 pipelines,“ *IEEE Journal on Selected Areas in Communications*, Jg. 36, Nr. 12, S. 2603–2611, Dez. 2018. DOI: [10.1109/JSAC.2018.2871288](https://doi.org/10.1109/JSAC.2018.2871288).
- [50] Netronome, *P4 Programmability for the Netronome Agilio SmartNIC.* Adresse: <https://www.netronome.com/blog/p4-programmability-for-the-netronome-agilio-smartnic/> (besucht am 18.03.2020).
- [51] The P4.org Architecture Working Group, *P4_16 Portable Switch Architecture (PSA)*, 2018. Adresse: <https://p4.org/p4-spec/docs/PSA.pdf> (besucht am 02.02.2020).
- [52] *SimpleSumeSwitch Architecture (v1.2.1 and Earlier)*, 2018. Adresse: [https://github.com/NetFPGA/P4-NetFPGA-public/wiki/SimpleSumeSwitch-Architecture-\(v1.2.1-and-Earlier\)](https://github.com/NetFPGA/P4-NetFPGA-public/wiki/SimpleSumeSwitch-Architecture-(v1.2.1-and-Earlier)) (besucht am 16.03.2020).
- [53] Xilinx, *P4-SDNet User Guide*, 2018. Adresse: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1252-p4-sdnet.pdf (besucht am 16.03.2020).

- [54] *P4_16 prototype compiler*. Adresse: <https://github.com/p4lang/p4c> (besucht am 02.04.2020).
- [55] M. Budiu und C. Doss, *The Software Architecture of the P4_16 Reference Compiler*, 2017. Adresse: <https://www.youtube.com/watch?v=nHYHaAgFsEs> (besucht am 24.03.2020). *Vortrag auf dem P4 Workshop 2017*.
- [56] T. K. Dangeti, V. Keerthy S. und R. Upadrasta, „P4LLVM: An LLVM Based P4 Compiler,“ in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, S. 424–429. DOI: 10.1109/ICNP.2018.00059.
- [57] P. G. Patra, C. E. Rothenberg und G. Pongrácz, „MACSAD: Multi-Architecture Compiler System for Abstract Dataplanes (Aka Partnering P4 with ODP),“ in *Proceedings of the 2016 ACM SIGCOMM Conference*, Ser. SIGCOMM ’16, New York, NY, USA: Association for Computing Machinery, 2016, S. 623–624. DOI: 10.1145/2934872.2959077. Adresse: <https://doi.org/10.1145/2934872.2959077>.
- [58] The P4.org API Working Group, *P4Runtime Specification version 1.1.0*, Feb. 2020. Adresse: <https://p4.org/p4runtime/spec/v1.1.0/P4Runtime-Spec.html> (besucht am 24.03.2020).
- [59] N. McKeown, J. Wanderer, T. Sloane et al., *P4 Runtime: Putting the Control Plane in Charge of the Forwarding Plane*, 2017. Adresse: https://www.youtube.com/watch?v=oEV_qY4r0Wg (besucht am 25.03.2020). *Im Rahmen eines LightReading Webinars*.
- [60] B. O’Connor und C. Cascone, *Next-Gen SDN Tutorial*, 2019. Adresse: <https://docs.google.com/presentation/d/1XLmqiZ1Gq9dkn8QjLmvvr13YexVb5PoX9kLYv74j57Y/edit?usp=sharing> (besucht am 25.03.2020). *Vortrag auf der ONF Connect 2019*.
- [61] The P4 Language Consortium, *Announcing the P4Runtime v1.0 release*, 2019. Adresse: <https://p4.org/p4/runtime-v1-release> (besucht am 25.03.2020).
- [62] *Specification documents for the P4Runtime control-plane API*. Adresse: <https://github.com/p4lang/p4runtime> (besucht am 25.03.2020).
- [63] Open Networking Foundation, *Stratum Project Main Repository*. Adresse: <https://github.com/stratum/stratum> (besucht am 25.03.2020).
- [64] A. Fingerhut und A. Bas, *The BMv2 Simple Switch target*. Adresse: https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md (besucht am 02.04.2020).
- [65] C. Brönnimann, *Technische Grundlagen zur LonWorks® Technologie*, Ettenhausen, 2018. Adresse: http://www.lonmark.ch/pdf/Technische_Grundlagen_zur_LonWorks.pdf (besucht am 05.01.2020).
- [66] A. Gurtner, *Vergleichende Bewertung von Meter Daten in Smart Metern, Konzentratoren und Backend Lösungen*, Bachelor’s Thesis, Salzburg, 2017.
- [67] E. Plasser, „Prinzipien der Powerline Communication,“ *e&i Elektrotechnik und Informationstechnik*, Jg. 118, Nr. 6, S. 311–315, 2001. DOI: 10.1007/BF03158911.
- [68] ÖVE/ÖNORM EN 50065-1: 2012 03 01: *Signalübertragung auf elektrischen Niederspannungsnetzen im Frequenzbereich 3 kHz bis 148,5 kHz - Teil 1: Allgemeine Anforderungen, Frequenzbänder und elektromagnetische Störungen*. Austrian Standards International.

- [69] A. A. Amarsingh, H. A. Latchman und D. Yang, „Narrowband Power Line Communications: Enabling the Smart Grid,“ *IEEE Potentials*, Jg. 33, S. 16–21, 2014. DOI: 10.1109/MPOT.2013.2249691.
- [70] L. Lampe und L. Berger, „Power line communications,“ *Academic Press Library in Mobile and Wireless Communications*, S. 621–659, Jan. 2016. DOI: 10.1016/B978-0-12-398281-0.00016-8.
- [71] H. Rabbie, *KGQ - Implementing the LonTalk Protocol for Intelligent Distributed Control*, 1998. Adresse: <http://www.stitcs.com/cn/lonworks/LonTalk%20Protocol%20Seminar.pdf> (besucht am 18.01.2020). *Vortrag auf der Embedded Systems Conference 1998*.
- [72] LOYTEC electronics GmbH, *LOYTEC electronics Home Page*. Adresse: <https://www.loytec.com/> (besucht am 03.04.2020).
- [73] Woo-Seop Kim, Lok-Won Kim, Chang-Eun Lee et al., „A control protocol architecture based on LonTalk protocol for power line data communications,“ in *2002 Digest of Technical Papers. International Conference on Consumer Electronics (IEEE Cat. No.02CH37300)*, Juni 2002, S. 310–311. DOI: 10.1109/ICCE.2002.1014043.
- [74] Adesto Technologies, *Adesto Technologies Home Page*. Adresse: <https://www.adestotech.com/> (besucht am 04.04.2020).
- [75] LonMark International Inc., *LonMark Home Page*. Adresse: <https://www.lonmark.org/> (besucht am 04.04.2020).
- [76] ÖVE/ÖNORM EN 14908-1: 2014 09 15: *Offene Datenkommunikation für die Gebäudeautomation und Gebäudemanagement - Gebäude-Netzwerk-Protokoll - Teil 1: Datenprotokollsichtenmodell*. Austrian Standards International.
- [77] *P4 Tutorial*. Adresse: <https://github.com/p4lang/tutorials> (besucht am 18.05.2020).
- [78] *P4 Learning*. Adresse: <https://github.com/nsg-ethz/p4-learning> (besucht am 18.05.2020).
- [79] *Consolidated switch repo (API, SAI and Netlink)*. Adresse: <https://github.com/p4lang/switch> (besucht am 18.05.2020).
- [80] *DC.p4 Repository*. Adresse: <https://github.com/p4lang/papers/tree/master/sosr15> (besucht am 21.05.2020).
- [81] *P4 Applications Repository*. Adresse: <https://github.com/p4lang/p4-lang/p4-applications> (besucht am 18.05.2020).
- [82] *P4 Learning L2_Learning-Example*. Adresse: https://github.com/nsg-ethz/p4-learning/tree/master/exercises/04-L2_Learning (besucht am 19.05.2020).
- [83] M. Shahbaz, S. Choi, B. Pfaff et al., „PISCES: A programmable, protocol-independent software switch,“ in *SIGCOMM 2016 - Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, Association for Computing Machinery, Inc, Aug. 2016, S. 525–538. DOI: 10.1145/2934872.2934886.
- [84] *Behavioral model targets*. Adresse: <https://github.com/p4lang/behavioral-model/blob/master/targets/README.md> (besucht am 22.05.2020).

- [85] *Simple Switch*. Adresse: <https://github.com/nsg-ethz/p4-learning/blob/master/documentation/simple-switch.md> (besucht am 22.05.2020).
- [86] *Packet Reflector*. Adresse: <https://github.com/nsg-ethz/p4-learning/tree/master/exercises/01-Reflector> (besucht am 23.05.2020).
- [87] *P4-Utils*. Adresse: <https://github.com/nsg-ethz/p4-utils> (besucht am 23.05.2020).
- [88] *p4app*. Adresse: <https://github.com/p4lang/p4app> (besucht am 21.05.2020).
- [89] F. S. Foundation, *GNU Make*. Adresse: <https://www.gnu.org/software/make/> (besucht am 23.05.2020).
- [90] *Implementing Basic Forwarding*. Adresse: <https://github.com/p4lang/tutorials/tree/master/exercises/basic> (besucht am 23.05.2020).
- [91] *Manual Setup*. Adresse: https://github.com/nsg-ethz/p4-learning/tree/master/examples/manual_setup (besucht am 21.05.2020).
- [92] *Vagrant*. Adresse: <https://www.vagrantup.com/> (besucht am 23.05.2020).
- [93] *Can't use log_msg() & Debugging #346*. Adresse: <https://github.com/p4lang/tutorials/issues/346> (besucht am 08.06.2020).
- [94] *Scapy 2.4.3 documentation*. Adresse: <https://scapy.readthedocs.io/en/latest/> (besucht am 25.05.2020).
- [95] *ip-link: network device configuration - Linux Man Pages*. Adresse: <https://www.systutorials.com/docs/linux/man/8-ip-link/> (besucht am 08.06.2020).
- [96] Robert Bosch GmbH, *CAN Specification Version 2.0*, 1991. Adresse: <http://esd.cs.ucr.edu/webres/can20.pdf> (besucht am 08.06.2020).
- [97] ISO/TC 22/SC 31, *ISO 11898-1:2015: Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signaling*, 2015. Adresse: <https://www.iso.org/standard/63648.html>.
- [98] A. Grattafiori, *NCC Group Whitepaper - Understanding and Hardening Linux Containers*, 2016. Adresse: https://www.nccgroup.com/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding-hardening_linux_containers-1-1.pdf (besucht am 09.06.2020).
- [99] B. Lantz, B. Heller und N. McKeown, „A Network in a Laptop: Rapid Prototyping for Software-Defined Networks,“ in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Ser. Hotnets-IX, New York, NY, USA: Association for Computing Machinery, 2010. DOI: 10.1145/1868447.1868466. Adresse: <https://doi.org/10.1145/1868447.1868466>.
- [100] *Graphs Backend*. Adresse: <https://github.com/p4lang/p4c/tree/master/backends/graphs> (besucht am 11.06.2020).
- [101] A. Bas, *Multi-Queue and Packet Scheduling in P4 Behavior Models*. Adresse: http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2016-May/000314.html (besucht am 12.06.2020).
- [102] *New question concerning priority queueing · Issue #896*. Adresse: <https://github.com/p4lang/behavioral-model/issues/896> (besucht am 12.06.2020).

- [103] M. Budiu, *Programming networks with P4 - VMware Research*, 2017. Adresse: <https://blogs.vmware.com/research/2017/04/07/programming-networks-p4/> (besucht am 16.06.2020).