Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Umsetzung von Event-basiertem Hardwarefiltering unter Nutzung von P4

Patrick Schneefuss

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer/in:	Dr. rer. nat. Sukanya Bhowmik
Beginn am:	4. April 2018
Beendet am:	4. Oktober 2018

Kurzfassung

Publish/Subscribe Systeme gewinnen durch ihre Flexibilität gegenüber wechselnden Netzwerktopologien ein immer höheres Ansehen. Ein Empfänger von Daten wird hier Abonnent genannt. Dieser kann sein Interesse an bestimmten Paketinhalten aussprechen, und wird folglich auch nur solche Pakete erhalten. Um dies zu erreichen, müssen Pakete auf ihrem Weg durch das Netzwerk gefiltert und je nach Inhalt über die entsprechenden Links weitergeleitet werden. Das Filtern in Software erfüllt strikte Anforderungen an die Latenz solcher Systeme kaum. Diese Arbeit wird zunächst auf Systeme eingehen, die diesen Filtervorgang bereits auf Hardwareebene durchführen. Hierdurch entstehen allerdings in all diesen Systemen gewisse Ungenauigkeiten, wie zum Beispiel das gelegentliche Weiterleiten von Events an Abonnenten, die kein Interesse an deren Informationen verkündet haben. Ziel dieser Arbeit ist es also, ein Publish/Subscribe System zu entwerfen, welches den Filtervorgang auf Hardwareebene durchführt, diese Ungenauigkeiten aber komplett vermeidet. Dies wird vor allem durch das Programmieren der Datenebene eines softwaredefinierten Netzwerkes anhand der Programmiersprache P4 ermöglicht. Hierdurch können nun statt nur diversen vordefinierten Headerfelder alle Daten eines Paketes für dieses Filtervorgang in Betracht gezogen werden. Die Kernpunkte dieser Arbeit sind somit sowohl der Entwurf eines geeigneten P4 Programmes, sowie der Logik eines Controllers, welcher die Filter auf Hardwareebene unter sich ändernden Abonnements dynamisch aktualisiert. Untersucht wird hierbei vor allem, wie sich diese neuen Ansätze auf die Skalierbarkeit und Performanz dieses Systems auswirken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einleitung

Durch die rasante Entwicklung des Internets über die vergangenen Jahre steigen vor allem auch die Anforderungen an Internet Anwendungen stetig an. Informationen müssen heutzutage nichtmehr nur über kleine, zentralisierte Systeme verteilt werden. Besonders in großen, verteilten Systemen ist es durchaus denkbar, dass Informationen zwischen Millionen Entitäten ausgetauscht werden müssen. Diese Entitäten können hier über die ganze Welt verteilt sein. Auch die Orte einzelner Entitäten wechseln unter Umständen von Zeit zu Zeit. Notwendig ist also ein skalierbares, effizientes System, welches sich dieser ständig im Wandel befindenden Netzwerktopologie anpasst. So genannte Publish/Subscribe Systeme (u.a [1][2]) gewinnen hier immer mehr an Popularität. Eine Entität hat in einem solchen System die Möglichkeit, anderen Entitäten ihr Interesse an bestimmten Informationen zu übermitteln. Diese Entitäten werden hier als Abonnent (engl. Subscriber) bezeichnet. Sogenannte Veröffentlicher oder Publisher senden Pakete über das Netzwerk. Ein durch einen Publisher versendetes Paket wird hier als Event bezeichnet. Die Aufgabe besteht nun darin, Events eines Publishers über das Netzwerk an alle Abonnenten zu leiten, die ihr Interesse an Informationen dieses Events verkündet haben. Insbesondere wird hierdurch vermieden, dass ein Abonnent solche Events erhält, die für ihn persönlich irrelevant sind. Unterschieden wird hierbei zwischen themenbasierten und inhaltsbasierten Publish/Subscribe Systemen. In einem themenbasierten Publish/Subscribe System kann ein Abonnent Interesse an einem bestimmten Themengebiet verkünden. Folglich erhält er in Zukunft nur Events, die Informationen zu diesem Themengebiet tragen. In einem inhaltsbasierten System hat der Abonnent die Möglichkeit, seine Interessen deutlich präziser zu spezifizieren. Beispielsweise ist es ihm hier möglich, nur bestimmte Teilmengen aller Events eines Themengebietes zu abonnieren. Die unterschiedlichen Informationen zu einem bestimmten Thema werden hier als Attribute bezeichnet. Einem Abonnenten ist es hierbei zum Beispiel möglich, nur Events zu abonnieren, deren Attribute jeweils einen bestimmten Wert besitzen. Das Filtern der Events übernehmen hier sogenannte Software-Broker auf dem Pfad zwischen Publisher und Abonnent. Publisher müssen nun vor allem keine Kenntnis über eventuelle Empfänger ihrer versendeten Events besitzen. In Bereichen wie Online-Gaming sind die Anforderungen an ein solches System in Sachen Latenz und Durchsatz mittlerweile allerdings sehr hoch. Das Vergleichen von einem Event mit allen aktuell vorliegenden Abonnements durch einen Broker kann diesen Anforderungen nicht gerecht werden, da Vergleiche auf Softwareebene zu hohe Latenzen mit sich bringen. Erstrebenswert wäre es also, das Vergleichen eingehender Events gegen die vorliegenden Abonnements direkt auf der Hardwareebene durchzuführen. Durch den Ansatz der Softwaredefinierten Netzwerke[3] eröffnen sich gerade hierfür einige neue Möglichkeiten. In Softwaredefinierten Netzwerken wird die Kontrollebene von der Datenebene getrennt. Die Kontrollebene besteht hier gewöhnlich aus leistungsstarken Servern, die Datenebene aus handelsüblichen Switches. Kontroll- und Datenebene kommunizieren hier über Kommunikationsprotokolle wie beispielweise *OpenFlow*[4]. Aufgabe der Kontrollebene ist die Steuerung der Datenebene. Hierbei kann die Kontrollebene über *OpenFlow* beispielsweise die Routingtabellen eines Switches der Datenebene modifizieren. Im Kontext eines Publish/Subscribe Systems ist es die Aufgabe der Kontrollebene, diese Tabellen so zu manipulieren, dass ein gesendetes Event von den Switches der Datenebene immer zu allen interessierten Abonnenten

geleitet wird. Unter *OpenFlow* werden auf Hardwareebene allerdings nur Vergleiche mit einer festen Anzahl an vordefinierten Headerfeldern [5] ermöglicht. Nötig ist es deshalb, die Werte aller Attribute eines Events zu kodieren und in das Format eines dieser Headerfelder umzuwandeln.

Diese Arbeit wird auf einige bereits existierende Publish/Subscribe Systeme eingehen, die sich Filter auf Hardwareebene und den Aspekt des softwaredefinierten Netzwerkens zu Nutzen machen. LIPSIN [6] präsentiert hier einen neuen Ansatz für einen performanten Multicast auf Netzwerkebene in einem themenbasierten Publish/Subscribe System. In *PLEROMA*[7][8] wird nun durch die Realisierung eines inhaltsbasierten Publish/Subscribe System eine höhere Aussagekraft der Abonnements, sowie durch den Verzicht auf Source-Routing eine höhere Flexibilität in Bezug auf die Netzwerktopologie ermöglicht. Vergleich von Eventinformation mit installierten Filtern findet hier direkt im ternären inhaltsadressierbaren Speicher (engl. Ternary Content Adressable Memory, kurz TCAM) eines Switches statt, wodurch eine sehr performante Weiterleitung von Events ermöglicht wird [9]. Die Umformung der für das Filtern relevanten Informationen in das Format eines unterstützten Headerfeldes, um Vergleiche auf Hardwareebene zu ermöglichen, bringt allerdings in beiden Ansätzen einige Nachteile mit sich. Logischerweise stellt die Größe des verwendeten Headerfeldes hier eine Beschränkung der maximalen Anzahl unterschiedlicher Kodierungen dar. Vor allem die Zustellung von Events an uninteressierte Abonnenten, sogenannte Falsch-Positive, ist ein hierdurch entstehendes Problem. Durch die Entstehung dieser Falsch-Positive wird vor allem die Bandbreite nicht optimal ausgenutzt.

Durch den technischen Fortschritt in der Entwicklung programmierbarer Switches, sowie der Vorstellung einer neuen Programmiersprache *P4*[10], anhand welcher das Verhalten eines Switches komplett nach Wünschen des Nutzers definiert werden kann, eröffnen sich hier diverse neue Möglichkeiten zuvor beschriebene Probleme zu vermeiden. Unter Verwendung von P4 ist man insbesondere nicht an die Verwendung von vordefinierten Headerfeldern für das Filtern der Events angewiesen. Es lassen sich problemlos eigene Header definieren, deren Felder für besagte Vergleiche ebenfalls verwendet werden können. Ziel dieser Arbeit ist es, aufbauend auf *PLEROMA* und unter der Nutzung von P4, ein Publish/Subscribe System für softwaredefinierte Netzwerke zu entwerfen, welches Falsch-Positive komplett vermeidet, um somit die Bandbreite optimal auszunutzen. Schwerpunkt hierbei ist vor allem die Logik zur dynamischen Aktualisierung der Datenebene bei sich zur Laufzeit ändernden Abonnements. Da das in dieser Arbeit entworfene System der erste Ansatz zu einer Realisierung eines Publish/Subscribe System in Verbindung mit einer programmierbaren Datenebene ist, ist hier besonders interessant, wie sich diese neuen Ansätze auf die Skalierbarkeit und Latenz eines solchen Systems auswirken.

Gliederung

Die Arbeit ist wie folgt gegliedert:

Kapitel 2 – Technischer Hintergrund wird alle für das Verständnis dieser Arbeit relevanten Grundlagen erläutern.

Kapitel 3 – Verwandte Arbeiten gibt einen Überblick über für diese Arbeit relevante, bereits existierende Werke und Systeme.

Kapitel 4 – P4 Publish/Subscribe stellt den Hauptteil dieser Arbeit dar. Dieses Kapitel beschreibt den Aufbau und die Funktionsweise des in dieser Arbeit entworfenen Systems.

Kapitel 5 – Auswertung befasst sich mit der Auswertung des in Kapitel 4 vorgestellten Systems.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse und Erkenntnisse aus den vorigen Kapiteln zusammen und stellt Anknüpfungspunkte vor.

2 Technischer Hintergrund

2.1 Publish/Subscribe

Publish/Subscribe[1][2] ist ein Mechanismus, um Informationen in einem Netzwerk flexibel von einem Publisher an einen Abonnenten zu leiten. Ein Publisher ist hier eine Entität des Netzwerkes, die Informationen in Form von Paketen versendet. Diese Pakete werden hier als ein Event bezeichnet. Ein Abonnent hat nun die Möglichkeit, bestimmte Teilmengen aller veröffentlichten Events zu abonnieren. Eine Netzwerkentität kann hier sowohl als Abonnent oder Publisher agieren, als auch beide Rollen zu gleicher Zeit einnehmen.
In einem Netzwerk können mehrere Publisher und Abonnenten zur selben Zeit aktiv sein. Auch kann ein Abonnent über mehrere aktive Abonnements verfügen. Ziel eines Publish/Subscribe System ist es nun, jedem Abonnenten auch nur diese Events zuzustellen, für die dieser aktuell ein Abonnement besitzt. Publisher benötigen keine Kenntnis über den Zielort oder die Zieladresse eines Abonnenten. Hierdurch lässt sich die Netzwerktopologie sehr flexibel ändern.
Unterschieden wird hier zwischen themenbasiertem und inhaltsbasiertem Publish/Subscribe. In einem themenbasierten System gehört jedes veröffentlichtes Event einem entsprechendes Themengebiet an. Abonnenten können nun spezifizieren, welche Themengebiete für sie relevant sind. Nur Events dieser abonnierten Themengebiete werden also an den jeweiligen Abonnenten übermittelt. Bei inhaltsbasiertem Publish/Subscribe besteht die Möglichkeit, deutlich präzisere Abonnements einzureichen. Relevant sind hier für einen Abonnenten weniger das Thema, als die Informationen eines Events. So trägt ein Event zum Thema „Konzertkarten“ also zum Beispiel Informationen über den Preis einer Karte, sowie die Kategorie des Platzes. Anstatt hier alle Events zu diesem Thema zu abonnieren, besitzt ein Abonnent beispielsweise die Möglichkeit nur Events mit Informationen über Karten in einer bestimmten Kategorie und einer bestimmten Preisspanne zu abonnieren. Jede Information eines Events wird hier als ein Attribut bezeichnet. Events in obigem Beispiel besitzen also zwei Attribute. Ein solches Event kann nun als ein Punkt in einem Koordinatensystem, auch Eventraum genannt, dargestellt werden. Die Dimension dieses Koordinatensystems entspricht genau der Anzahl der Attribute eines Events, wobei jedes Attribut einer der Achsen des Koordinatensystems zugewiesen wird. Das erste Attribut eines Events wird deshalb im Weiteren auch als die erste Dimension eines Events bezeichnet, analoges gilt für alle weiteren Attribute. Ein Event mit Informationen zu einer Konzertkarte der Kategorie 3 und dem Verkaufspreis von 20 Euro wird hier in Abbildung 2.1 im Eventraum als roter Punkt dargestellt.

Analog hierzu lassen sich Abonnements in einem zweidimensionalen Eventraum als eine Fläche darstellen, in einem dreidimensionalen Eventraum als ein Körper. Abbildung 2.1 enthält hier zum Beispiel ein Abonnement zu Events mit Kartenpreisen zwischen 10 und 30 Euro, sowie einer Sitzplatzkategorie zwischen 2 und 4. Relevant für einen Abonnenten sind also alle Events, die in diesem Eventraum in einer Fläche liegen, die ein Abonnement dieses Abonnenten repräsentiert.

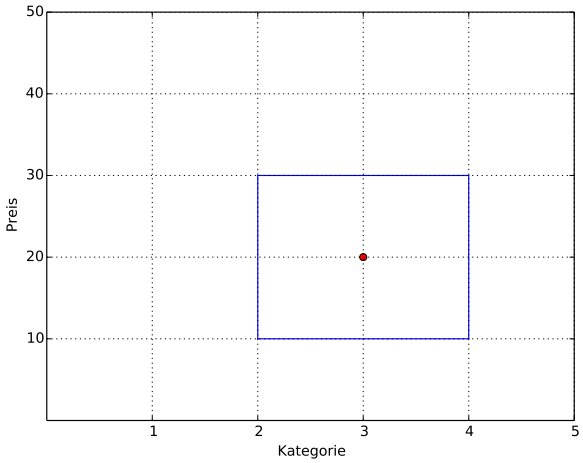

Abbildung 2.1: Events und Abonnements im Eventraum

Abonnements unterschiedlicher Abonnenten können sich hier überlagern oder überschneiden. Events, welche in einer Teilfläche liegen, die sich mehrere Abonnements teilen, müssen folglich auch allen zugehörigen Abonnenten zugestellt werden.

Zuständig für die Berechnung des Pfades, beziehungsweise der Pfade, über die ein Event nun von einem Publisher zu einem Abonnenten geleitet werden muss, sind sogenannte Software-Broker, kurz Broker. Diese Broker sind hier eigenständige Entitäten des Netzwerkes und können also weder die Rolle eines Abonnenten, noch die eines Publishers einnehmen. Ein jeder Broker benötigt einen Überblick über alle Abonnements eines jeden Abonnenten des Netzwerkes. Geht nun ein Event an einem Broker ein, so kann dieser die Werte einer jeden Dimension des Events mit allen vorliegenden Abonnements vergleichen, und die Zieladresse eines Events entsprechend modifizieren. Untenstehende Abbildung 2.2 demonstriert ein sehr einfaches Publish/Subscribe Szenario bestehend aus einem Publisher und zwei Abonnenten. Der zwischen Publisher und Abonnent geschaltete Broker B trifft hier die Entscheidung, an welche Abonnenten ein eingehendes Event weitergeleitet werden muss.

2.2 Softwaredefinierte Netzwerke

Gewöhnliche Netzwerke bestehen aus einer unter Umständen sehr hohen Anzahl verschiedener Hardware zue Weiterleitung von Paketen, auch von unterschiedlichen Herstellern. Die Logik des Netzwerkes ist über alle diese Switches oder Router verteilt. Je nach Hersteller ist hier eine eigene Implementierung dieser Logik von Nöten. Dies erschwert die Erweiterung solcher Netzwerke um zusätzliche Hardware immens. Solche Netzwerke werden aktuellen Anforderungen nichtmehr gerecht. Softwaredefiniertes Netzwerken (SDN)[3] ist ein neues Konzept zum Entwerfen von Computer-Netzwerken. Wichtigster Aspekt ist hierbei die Trennung von Kontrollebene und Datenebene. Die Kontrollebene ist nun für die Steuerung des Netzwerkes, die Datenebene für die Weiterleitung von Paketen zuständig. Hierfür besteht die Kontrollebene für gewöhnlich aus leistungsstarken Servern, wohingegen handelsübliches Switches und Router die Datenebene bilden. Ein solcher

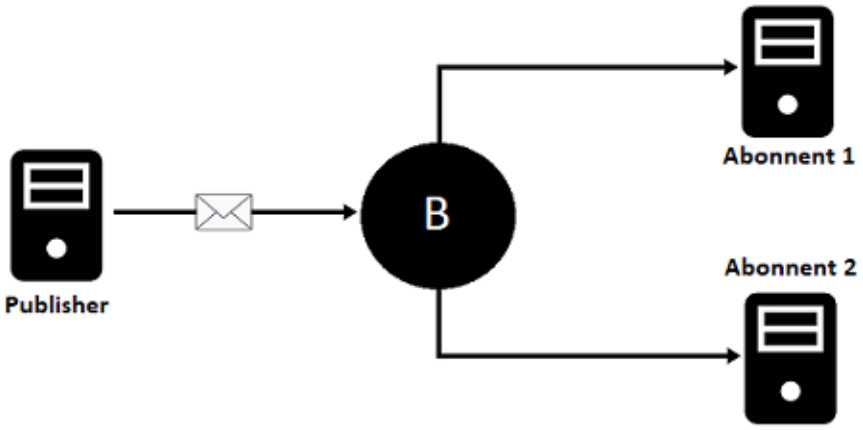

Abbildung 2.2: Publish/Subscribe System mit Broker

Server der Kontrollebene wird auch Controller genannt. Einem Controller ist es nun möglich, die Datenebene zu konfigurieren. Hierbei können beispielsweise die Einträge der Routingtabellen eines Routers modifiziert werden. Ein Eintrag in diesen Tabellen gibt an, über welchen Port bestimmte Pakete weitergeleitet werden sollen. Diese Einträge werden jeweils auch als *Flow* bezeichnet. Durch diese Zentralisierung der Logik eines Netzwerkes kann dieses problemlos um weitere Hardware erweitert werden. Die neue Hardware führt ebenfalls die Befehle der Kontrollebene aus. Vor allem ist ein softwaredefiniertes Netzwerk hierdurch nicht mehr an Hardware spezieller Hersteller gebunden. Auf je nach Hardware unterschiedliche Implementierungen der Netzwerklogik kann hier verzichtet werden. Die Kommunikation zwischen Kontroll- und Datenebene findet hier über Kommunikationsprotokolle wie zum Beispiel OpenFlow[4] statt.

2.3 SDN im Kontext von Publish/Subscribe

Die Anforderungen an Netzwerke sind über die letzten Jahre sehr stark gestiegen. Gerade in Bereichen wie Online-Gaming werden sehr niedrige Latenzzeiten gefordert. Die Verwendung von Brokern in Publish/Subscribe Systemen kann diesen Anforderungen nichtmehr gerecht werden. Das Filtern eines jeden Events in Software weist hier zu hohe Latenzen auf. Das Konzept des softwaredefinierten Netzwerkes ermöglicht es nun allerdings, das Filtern direkt in der Hardware der Datenebene durchzuführen. Der Controller des Netzwerkes ist also dafür zuständig, die Flow-Tabellen eines jeden Switches des Netzwerkes so zu konfigurieren, dass ein jedes gesendetes Event bei allen interessierten Abonnenten, aber auch nur bei diesen, ankommt. Hierfür senden alle Abonnenten ihre Abonnements, beziehungsweise die Kündigung eines zuvor eingereichten Abonnements, direkt an den Controller. Da die Anzahl der Flow-Einträge eines Switches aber begrenzt ist, sollen nur die wirklich nötigen Flows auf einem Switch installiert werden. Ein Flow ist hier nötig, wenn tatsächlich auch Events versendet werden, die zu diesem Flow passen. Hierfür muss jeder Publisher dem Controller in Form eines *Advertisments* zusätzlich mitteilen, welche Informationen er in Zukunft veröffentlichen wird. Ein Advertisment kann also wie auch ein Abonnement als eine Fläche oder ein Körper im Eventraum repräsentiert werden. Der Controller gleicht nun alle vorliegenden Advertisments mit allen vorliegenden Abonnements ab und konfiguriert

die Flow-Tabellen entsprechend. Die Zeit, die ein Controller zum Erstellen der Flow-Tabellen benötigt, beeinflusst hierbei vor allem die schlussendliche Latenz versendeter Events nicht. Beim Filtern auf Hardwareebene müssen nun die Werte einer jeden Dimension eines Events mit den entsprechenden Werten eines Flow-Eintrages verglichen werden. Liegen die Werte des Events in den von einem Flow-Eintrag spezifizierten Bereichen, so erzielt dieser Eintrag für dieses Event einen Treffer. Leider ist es mit herkömmlichen Kommunikationsprotokollen wie OpenFlow[4] und nicht programmierbaren Switches nur möglich, Vergleiche einer festen Anzahl an vordefinierten Headerfeldern durchzuführen. Hierzu gehört zum Beispiel die IP-Adresse. In diesem Fall wird also die IP-Adresse des eingehenden Events mit der IP-Adresse eines Flow-Eintrages verglichen. Nötig ist es hierfür also, die Informationen zu jeder Dimension eines Events zu kodieren und die Sammlung der Kodierungen aller Dimensionen in eine solche IP-Adresse umzuwandeln. Auch Abonnements müssen analog umgewandelt werden, bevor sie durch einen Floweintrag repräsentiert werden können.

2.4 Programmierung der Datenebene mit P4

Die Notwendigkeit der Kodierung eines Abonnements oder der Informationen eines Events bringt durch die Größe des hierfür verwendeten Headerfeldes eine Beschränkung der maximalen Anzahl möglicher, verschiedener Kodierungen mit sich. Man ist also auf eine maximale Anzahl an verschiedenen Abonnements beschränkt, und folglich lassen sich auch nicht alle Abonnements so präzise wie gewünscht darstellen. In Kapitel 3 wird weiter darauf eingegangen, welche negativen Effekte diese Beschränkung mit sich bringt. Wünschenswert wäre es, die Werte einer jeden Dimension eines Events mit den Flow-Einträgen eines Switches zu vergleichen, ohne sie vorher zu kodieren. Hierfür müsste der Switch aber die Möglichkeit besitzen, die jeweilige Felder des Paketes, welche relevante Information für das Publish/Subscribe System tragen, auch zu identifizieren. Bisher wurde immer davon ausgegangen, dass die Datenebene aus nicht programmierbarer Hardware mit fester Funktion besteht. Dies hat vor allem damit zu tun, dass solche Switches gegenüber einem programmierbaren Switch bisher über eine deutlich höhere Geschwindigkeit bei der Paketverarbeitung verfügten. Über die letzten Jahre wurden nun allerding programmierbare Switches entwickelt, die ihrem Gegenüber hier mindestens ebenbürtig sind. Die Programmiersprache *P4*[10] ermöglicht nun genau das oben beschriebene, erwünschte Szenario. *P4* ist eine protokollunabhängige, deklarative Programmiersprache, die es ermöglicht, das Verhalten eines programmierbaren Switches nach Belieben des Nutzers zu programmieren. Hierdurch kann der Nutzer direkten Einfluss darauf nehmen, wie eingehende Pakete von einem Switch verarbeitet und weitergeleitet werden sollen, auch nachdem besagter Switch bereits im Netzwerk eingesetzt wurde. Außerdem ist die Funktionalität und Implementierung des P4 Programmes unabhängig von der Hardware des Zielswitches oder Zielrouters. Wichtigster Aspekt für das in dieser Arbeit entworfene System ist die Möglichkeit, in P4 eigene Header zu definieren und daraufhin auch Vergleiche anhand der Felder dieses Headers durchzuführen. Genauer lässt sich hier aber auch die gesamte Pipeline zur Paketverarbeitung und Weiterleitung definieren. Aktuell existieren zwei Versionen von P4, $P4_{14}$ [11] und $P4_{16}$ [12]. Die Listings dieses Kapitels zeigen hierbei immer Beispielimplementierungen in $P4_{14}$. Semantik dieser Implementierungen unterscheiden sich von Version zu Version kaum, lediglich in der Syntax existieren gewisse Unterschiede. $P4_{16}$ verfügt hier über diverse Verbesserungen gegenüber $P4_{14}$, welche für die Zwecke dieser Arbeit allerdings kaum relevant sind. Anhand diverser Compiler

lassen sich $P4_{14}$ Programme problemlos in ein äquivalentes $P4_{16}$ Format konvertieren. Auf für Publish/Subscribe relevante Unterschiede zwischen den beiden Versionen wird in diesem Kapitel bei Bedarf eingegangen.

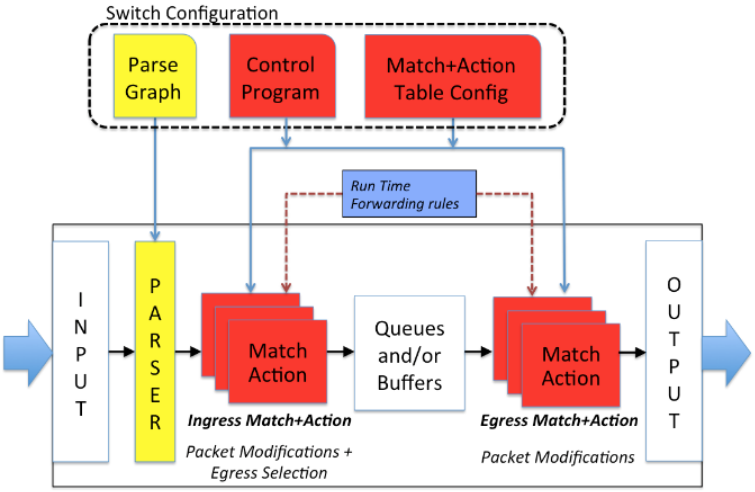

Abbildung 2.3: Abstraktes Weiterleitungsmodell[11] in $P4_{14}$

Das abstrakte Modell zur Weiterleitung von Paketen ist in Abbildung 2.3 dargestellt. Die wichtigsten Komponenten sind hier der Paketparser, sowie die Ingress und Egress Pipelines. Letztere bestehen aus jeweils einer oder mehreren Flowtabellen, hier auch *Match&Action*-Tabellen genannt. Ein Kontrollprogramm definiert hier die Reihenfolge aller Stationen, die ein Paket währen der Verarbeitung und Weiterleitung durchläuft. Die Tabelleneinträge können entweder statisch definiert werden, oder aber auch nach Inbetriebnahme eines Switches hinzugefügt werden. Diese Aufgabe wird für gewöhnlich von einem SDN-Controller übernommen.

2.4.1 Parser

Die Aufgabe des Paketparsers ist es, die gewünschten Headerfelder eines eingehenden Events zu extrahieren und eine geparste Repräsentation all dieser Headers zu erzeugen. Üblicherweise ist hier je ein Parser für einen Paketheader zuständig. Um alle gewünschten Header zu extrahieren müssen also mehrere Parser in Reihe ausgeführt werden. In einem P4-Programm muss nun der Startpunkt für den Parse-Vorgang definiert werden. Gewöhnlicherweise ist das der Ethernet Parser. Wurde dieser Header extrahiert, kann der nächste Parser aufgerufen werden. Dieser kann entweder fest definiert sein, es besteht allerdings auch die Möglichkeit die Wahl des nächsten Parsers von den Werten der Felder des eben geparsten Headers abhängig zu machen. Beim Parsen des Ethernet-Header kann hier beispielsweise anhand des EtherType des eingehenden Paketes entschieden werden, ob hiernach der IPv4- oder IPv6-Header geparst werden muss. Die Reihenfolge, in der die einzelnen Parser durchlaufen werden, kann hier auch, wie in Abbildung 2.4 gezeigt, als Parse-Graph dargestellt werden.

```
parser start {
    return parse_ethernet;
}

#define ETHERTYPE_IPV4 0x0800
header ethernet_t ethernet;

parser parse_ethernet {
    extract(ethernet);
    return select(latest.etherType) {
        ETHERTYPE_IPV4 : parse_ipv4;
        default: ingress;
    }
}
```

Listing 2.1: Ethernet Parser in $P4_{14}$

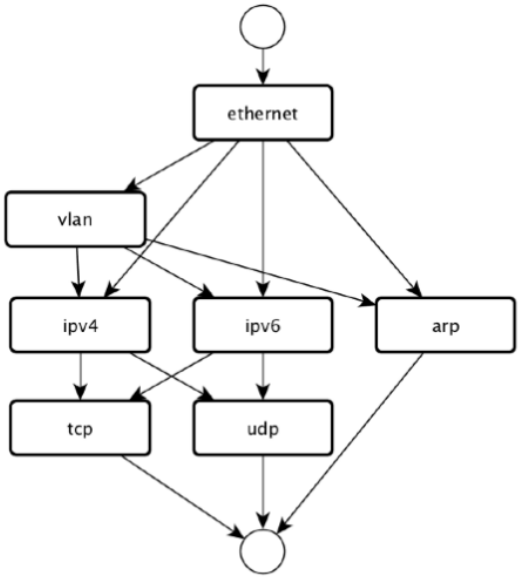

Abbildung 2.4: Beispiel eines Parse-Graphen

2.4.2 Header und Felder

Um die Korrektheit des Parsens zu garantieren, muss der Parser Kenntnis über die Anzahl der Felder des zu extrahierenden Headers besitzen. Um die Übergänge zwischen den Felder erkennen zu können, muss er zusätzlich die Größe eines jeden dieser Felder kennen. Hierfür müssen alle zu extrahierenden Header im zugehörigen P4 Programm definiert werden. Die Größe der Headerfelder wird hier in der Anzahl der jeweils verwendeten Bits angegeben. Wichtig ist hierbei, dass jedes Feld, welches schlussendlich auch im Header eines eingehenden Pakets präsent ist, auch definiert ist. Wichtig ist außerdem die korrekte Anordnung dieser Felder. Ist diese nicht gegeben, werden folglich

Headerinformationen in falschen Felder gespeichert. Eine Beispieldefinition eines Ethernet-Header ist Listing 2.2 zu entnehmen.

```
header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}
```

Listing 2.2: Ethernet-Header Definition in $P4_{14}$

Hierdurch ergibt sich die Möglichkeit, einen neuen Header zu definieren, der alle für Publish/Subscribe relevanten Informationen trägt. Im Kontext des inhaltsbasierten Publish/Subscribe muss dieser Header also für jede Dimension des Eventraumes ein separates Feld besitzen. Diese Informationen können nun an festgelegter Stelle in die Nutzdaten der Events geschrieben werden, am Switch werden diese dann als genau dieser Header erkannt. Da die Anzahl der Headerfelder statisch im P4 Programm definiert sind, ist hier eine Änderung der Dimensionsanzahl zur Laufzeit nicht möglich. Auch ist hier festzustellen, dass die Verwendung von Dezimalzahlen in P4 nicht unterstützt wird. Wertebereiche von Abonnements lassen sich also nur durch ganze Zahlen repräsentieren. Der Parser speichert nun die geparste Repräsentation eines Headers in einer Instanz des entsprechenden Headertyps. Analog zu der beschriebenen Headerdefinition lassen sich auch, falls benötigt, zusätzliche Metadaten definieren. Hierdurch lässt sich einem Paket beispielweise ein Zustand zuordnen, der beim Durchlaufen der Pipeline modifiziert oder gelesen werden kann.

```
header_type custom_metadata_t {
    fields {
        state : 16;
    }
}
```

Listing 2.3: Metdaten Definition in $P4_{14}$

Standard Metadaten wie der Ausgangs- oder Eingansport eines Paketes müssen allerdings nicht explizit definiert werden. Eine solche Instanz der *standard_metadata* wird standardmäßig jedem eingehenden Paket zugeordnet. Die Metadaten enthalten gewöhnlich Informationen, welche benötigt werden, während das Paket die Pipeline des Switches durchläuft. Das letztendlich ausgehende Paket enthält diese Metadaten logischerweise nichtmehr.

2.4.3 Tabellendefinition

Hauptbestandteil der Ingress- und Egress Pipeline sind die *Match&Action*-Tabellen. Beide Pipelines können eine oder mehrere solcher Tabellen enthalten. Diese können in Reihe oder auch parallel zueinander angeordnet sein. Bei der Definition einer solchen Tabelle müssen die Felder angegeben werden, die für das Filtern in dieser Tabelle verwendet werden. Für gewöhnlich sind dies Felder der geparsten Header oder der Metadaten eines Pakets. Ein Tabelleneintrag besteht

später aus einem Wert, oder einem Wertebereich in Form von Unter- und Obergrenze, jeder dieser gewählten Felder. Für jedes dieser Felder muss das Kriterium für einen Treffer definiert werden. Hier kann entschieden werden, ob der Wert eines eingehenden Paketes *exakt* mit dem Wert des Tabelleneintrages übereinstimmen muss, oder ob er in einem bestimmten Wertebereich (*range*) liegen soll. Auch ternäre Vergleiche sind hier möglich. Hierbei wird jedem Tabelleneintrag zusätzlich eine Bitmaske zugeordnet. Die Länge dieser Bitmaske entspricht der Länge des jeweiligen Feldes. Um einen Treffer zu erzielen, müssen hier nur die Bits des Headerfeldes mit denen des Tabelleneintrages übereinstimmen, die in der Bitmaske auch gesetzt sind. Analog hierzu lässt sich auch das längste Präfix als Trefferkriterium wählen. Um die Fläche eines Abonnements hier präzise als Floweintrag zu repräsentieren, eignet sich die Wahl des Wertebereiches (*range*) als Trefferkriterium perfekt. Zusätzlich hierzu müssen auch alle Operationen angegeben werden, die bei einem eventuellen Treffer eines Eintrags dieser Tabelle ausgeführt werden können. Teil eines Tabelleneintrages ist neben den Werten aller Felder später nun auch die bei einem Treffer dieses Eintrages auszuführende Operation. Operationen, auch Aktionen genannt, werden in P4 als Funktionen definiert. Diese Funktionen setzten sich für gewöhnlich aus einer oder mehreren primitiven Aktionen zusammen. Primitive Aktionen sind hier zum Beispiel *modify_field, add_to_field* oder *bit_and*. Untenstehendes Listing zeigt hier eine Beispieldefinition einer Tabelle, die den Eingangsport eines Events ausliest und dieses dann je nach gelesenem Wert entweder fallenlassen oder weiterleiten kann.

```
table in_tbl {
  reads {
    standard_metadata.ingress_port : exact;
  }
  actions {
    fwd_act;
    drop_act;
  }
}
```

Listing 2.4: Beispieldefinition einer Tabelle in $P4_{14}$

Für den Fall, dass ein eingehendes Event mit keinem der Tabelleneinträge einen Treffer erzielt, muss für jede Tabelle eine Standardaktion definiert werden. Diese wird in gerade genanntem Fall auf das Paket angewendet. Im Kontext von Publish/Subscribe ist diese Standardaktion gewöhnlich für das Fallenlassens eines Pakets zuständig. Die Entscheidung, welche der in der Tabellendefinition aufgeführten Aktionen als Standardaktion dient, wird hier zur Laufzeit getroffen.

2.4.4 Kontrollfluss

Das Kontrollprogramm, beziehungsweise der Kontrollfluss, eines P4 Programmes spezifiziert, wie ein eingehendes, bereits geparstes Paket behandelt werden soll. Definiert werden muss hier vor allem die Reihenfolge, in der die zuvor definierten *Match&Action*-Tabellen durchlaufen werden sollen. Hier besteht auch die Möglichkeit, Pakete mit unterschiedlichen Formaten oder Werten unterschiedliche Tabellen durchlaufen zu lassen. So ist es beispielsweise sinnvoll, ein eingehendes Paket mit erfolgreich geparstem IPv4-Header an eine Tabelle zur Weiterleitung von IPv4 Paketen zu übergeben, Pakete mit erfolgreich geparstem IPv6-Header hingegen an eine Tabelle zur Weiterleitung von IPv6 Paketen. Die verschiedenen Tabellen, die ein Paket also durchlaufen soll, können hier im

Kontrollprogramm anhand eines *if/else* Konstruktes definiert werden. $P4_{16}$ ermöglicht es außerdem Aktionen direkt im Kontrollprogramm aufzurufen. In der älteren Version von P4, $P4_{14}$, lassen sich solche Aktionen jedoch nur durch einen Treffer einer $Match\&Action$-Tabelle ausführen. Um also ein für das System irrelevantes Paket fallenzulassen, muss dieses in $P4_{14}$ an eine Dummy-Tabelle übergeben werden, welche über keine Tabelleneinträge, sondern nur über die Standardaktion zum Fallenlassen eines Paketes verfügt. Alternativ besteht in beiden Versionen die Möglichkeit, ein irrelevantes Paket direkt beim Parsen der Header fallenzulassen, indem es an einem Punkt weder an den Ingress Kontrollfluss, noch an einen weiteren Parser übergeben wird. In P4 ist für die Ingress und Egress Pipeline jeweils ein eigenständiger Kontrollfluss zu definieren. Wichtige Informationen zu Weiterleitung des Events, wie zum Beispiel der Ausgangsport, müssen hierbei schon nach Durchlaufen der Ingress Pipeline festgelegt werden. Änderung dieser Felder während Durchlaufen der Egress Pipeline werden auf die Weiterleitung des Pakets keinen Einfluss mehr nehmen.

3 Verwandte Arbeiten

3.1 P4CEP

Complex Event Processing (CEP) ist eine Technologie, die eine massive Menge an eingehenden Informationen, auch einfache Events genannt, in Echtzeit verarbeitet und analysiert. Ziel ist es hierbei, anhand von gewissen Mustern eine kausale oder temporale Relation zwischen diesen eingehenden Events herzustellen. Die Ergebnisse dieser Analyse werden als komplexe Events bezeichnet und können bei CEP End-Hosts eine entsprechende Reaktion auslösen. Beispielsweise können die eingehenden Messdaten verschiedener Feuermelder ein komplexes Event in Form eines Feueralarmes auslösen.

P4CEP[13] verbindet diese Technologie nun mit einer durch P4 programmierten Datenebene. Ein durch P4 programmierter Switch, oder auch P4-Target, wird hier mit Regeln bestückt, die für bestimmte Abfolgen von erhaltenen einfachen Events die jeweiligen reaktiven komplexen Events definieren. Umgesetzt wird dies durch die Implementierung einer Zustandsmaschine. Die Aktionen, die ein eingehendes einfaches Event auslöst, hängen hier direkt von dem aktuellen Zustand des P4-Targets ab. Abbildung 3.1 zeigt P4CEP's Systemmodell. P4CEP koexistiert hier mit den gewöhnlichen Netzwerkdiensten.

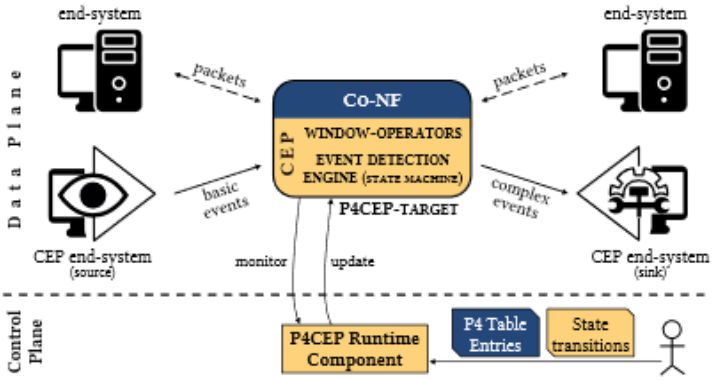

Abbildung 3.1: P4CEP Systemmodell[13]

P4CEP beschäftigt sich im Gegensatz zu dieser Arbeit nicht mit dem Thema Publish/Subscribe und auch der Aspekt der dynamischen Änderungen im System steht hier weniger im Vordergrund. Allerdings werden hier, wie auch in inhaltsbasiertem Publish/Subscribe , ebenfalls Aktionen abhängig von den jeweiligen Informationen eines eingehenden Events ausgelöst. Dies in Kombination mit der Implementierung der Weiterleitungspipeline in P4 stellen somit trotz allem eine gewisse Relevanz für diese Arbeit dar.

3.2 LIPSIN

Line-Speed Publish/Subscribe Inter-Networking (LIPSIN[6]) präsentiert einen neuen Ansatz des Multicast auf der Netzwerkschicht in einem themenbasierten Publish/Subscribe System. Zwar beschäftigt sich diese Arbeit nicht mit themen-, sondern inhaltsbasiertem Publish/Subscribe, allerdings sind die Ergebnisse von LIPSIN in Bezug auf die Performanz des Systems durch die Nutzung von Filtern auf Hardwareebene für diese Arbeit trotz allem sehr interessant. Grundsätzlich wird hier zwischen zwei Phasen unterschieden. Erstere wird als *recursive bootstrapping*[6] bezeichnet. Hierbei sammeln alle Knoten des Netzwerkes durch den Austausch untereinander Informationen über das Netzwerk, bis ein jeder Knoten über eine globale Übersicht über die gesamte Netzwerktopologie verfügt. Die zweite Phase befasst sich nun mit der Weiterleitung eines Events durch dieses Netzwerk. Um sicherzustellen, dass ein Event allen interessierten Abonnenten zugestellt wird, wird hier jeder Link des Netzwerkes durch eine Link ID repräsentiert. Ein Link ist hier unidirektional. Das bedeutet, dass eine Verbindung zwischen zwei Knoten A und B hier durch zwei verschiedene Link IDs repräsentiert wird. Eine Link ID ist ein 256 Bit langer Bitstring. Zu Beginn wird hier festgelegt, wie viele Bits einer Link ID gesetzt werden. Diese Anzahl variiert nie von Link ID zu Link ID. In LIPSIN werden für gewöhnlich 5 Bits in einer jeden Link ID gesetzt. Welche 5 Bits in einer Link ID nun schlussendlich gesetzt werden kann unter anderem zufällig bestimmt werden. Der Publisher muss nun vor Versenden eines Events Kenntnis über alle relevanten Abonnenten besitzen, und kreiert anhand seiner globalen Netzwerkübersicht einen Spannbaum bestehend aus allen Links, die für die korrekte Zustellung des Events an alle Abonnenten verwendet werden müssen. Diese Sammlung an Link IDs muss nun dem Paketheader des Events hinzugefügt werden. Hierfür macht sich LIPSIN Bloom Filter[14] zu Nutze. Dieser Bloom Filter ist ebenfalls ein 256 Bit großer Bitstring, und wird hier als $zFilter$ bezeichnet. Initial ist in diesem $zFilter$ kein Bit gesetzt. Eine Link ID kann einem $zFilter$ nun durch die Verwendung einer bitweisen Oder-Operation hinzugefügt werden. Hierbei werden also alle Bits des $zFilter$ gesetzt, welche auch in der Link ID gesetzt sind. Ist dieses Bit im $zFilter$ bereits gesetzt, muss hier auch keine Änderung vorgenommen werden. Wurden alle Link IDs dem $zFilter$ hinzugefügt, so wird dieser dem Paketheader des Events angeheftet. Durch die Verwendung von 256 Bit großen $zFiltern$ bieten sich hierfür die Felder der Start- und Zieladresse des IPv6-Headers an. An einem jeden Knoten kann nun der $zFilter$ eines eingehenden Events mit den Link IDs aller von diesem Knoten ausgehenden Verbindungen durch eine bitweise Und-Operation verglichen werden. Sind hierbei alle Bits des $zFilters$ gesetzt, die auch in der jeweiligen Link ID gesetzt sind, so wird das Event auch über diesen Link weitergeleitet. Hierdurch wird sichergegangen, dass jeder Link des ursprünglichen Spannbaumes auch verwendet wird, also dass keine Falsch-Negative entstehen. In der Natur eines Bloom-Filters liegt es allerdings, dass es zwar möglich ist, eindeutig zu erkennen ob ein Element diesem Filter nicht hinzugefügt wurde, jedoch nicht möglich ist, mit eindeutiger Sicherheit zu sagen, ob ein bestimmtes Element diesem Filter hinzugefügt wurde. Am Beispiel der Link ID könnten alle dieser Link ID entsprechenden Bits eines $zFilter$ auch durch das Hinzufügen diverser anderer Link IDs gesetzt worden sein. In Abbildung 3.2 wurden Link ID 1 und Link ID 2 dem $zFilter$ hinzugefügt. Nun wird Link ID 3 mit dem $zFilter$ durch ein bitweises Und verglichen, um zu überprüfen, ob Link ID 3 in diesem $zFilter$ enthalten ist. Das Resultat dieser bitweisen Und-Operation entspricht hier also genau Link ID 3, obwohl diese dem $zFilter$ selbst nicht hinzugefügt wurde. Hierdurch werden also auch Links für die Weiterleitung verwendet, die nicht Teil des Spannbaumes sind. Abonnenten erhalten somit unter Umständen auch Falsch-Positive, also für sie irrelevante Events.

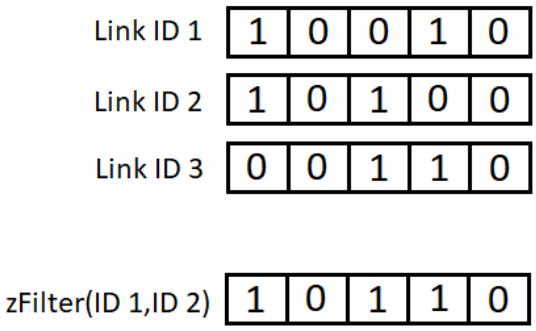

Abbildung 3.2: Bloom Filter

3.3 PLEROMA

Die Verwendung von Source-Routing in LIPSIN[6] bringt mit sich, dass jeder Publisher über alle Abonnenten und ihren Ort, sowie über alle Links des Netzwerkes Bescheid wissen muss. Dies erschwert vor allem Änderungen an der aktuellen Netzwerktopologie. Außerdem bieten themenbasierte Publish/Subscribe Systeme generell nicht die gewünschte Aussagekraft eines Abonnements. Die *PLEROMA* Middleware[8][7][15][16][17][9][18][19][20] liefert nun eine Realisierung eines inhaltsbasierten Publish/Subscribe Systems für softwaredefinierte Netzwerke mit hoher Performanz in Hinsicht auf die Nutzung der verfügbaren Bandbreite und der Ende-zu-Ende-Latenz. Im Besonderen benötigen Publisher hier keine Kenntnis über weitere Entitäten des Netzwerkes. Da das in dieser Arbeit entworfene Publish/Subscribe System stark auf *PLEROMA* aufbaut, wird in diesem Abschnitt detailliert auf das Systemmodell und die Funktionsweise von *PLEROMA* eingegangen. Die untenstehende Abbildung 3.3 zeigt eben erwähntes Systemmodell. Publisher teilen dem SDN-Controller in Form von Advertisments mit, welchen Inhalt sie von nun an über das Netzwerk senden werden. Abonnenten teilen dem Controller in Form von Abonnements mit, an welchem Inhalt der zugehörige End-Host aktuell Interesse hat. Der Controller unterteilt sich bei *PLEROMA* in zwei Hauptkomponenten, den *Dispatcher* und den *Configurator* [7]. Der *Dispatcher* nimmt Advertisments und Abonnements der Hosts entgegen und verarbeitet diese, worauf er sie dann an den *Configurator* weiterleitet.

Die Aufgabe des *Configurators* ist es nun, zu ermitteln, ob es bereits aktive Abonnenten gibt, die von einem eingehenden Advertisment betroffen sind, oder umgekehrt, ob es aktuell Publisher gibt, die bereits passende Events zu einem eben eingegangenen Abonnement senden. Ist dies der Fall, so ist es auch die Aufgabe des *Configurators*, die Datenebene so zu aktualisieren, dass zukünftige Events auch immer alle korrekten End-Hosts erreichen. Der *Configurator* ermittelt hierfür jeweils den Pfad von einem Publisher zu einem interessierten Abonnenten und aktualisiert über OpenFlow[4] Nachrichten die Flowtabellen aller Switches auf diesem Pfad.

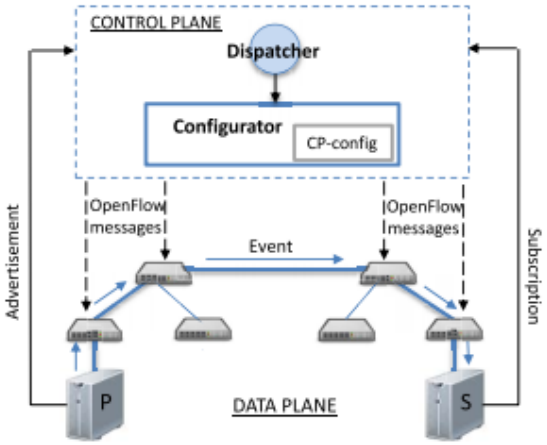

Abbildung 3.3: PLEROMA Systemmodell[7]

3.3.1 Events, Abonnements und Advertisments

Um Abonnements oder Events einem Bereich im Eventraum zuzuordnen verwendet *PLEROMA* das Prinzip räumlicher Indizes [21]. Hierbei wird der komplette Eventraum Schritt für Schritt in eine Rasterhierarchie unterteilt. In einem zweidimensionalen Eventraum wird jede Fläche, beginnend mit der Fläche des gesamten Eventraumes, hier in jedem Schritt der Indizierung an einer der beiden Achsen in zwei Teilflächen unterteilt. Die Trennung findet hier genau am Mittelpunkt der Fläche an der jeweiligen Achse statt. Von Schritt zu Schritt alternieren nun die Achsen, an denen die Trennung der vorhandenen Flächen durchgeführt wird. Aus dem ersten Schritt resultieren also zwei Teilflächen, die im zweiten Schritt jeweils erneut, an der anderen Achse, in zwei weitere Teilflächen unterteilt werden. Nach zwei Schritten der Indizierung wurde der Eventraum folglich in 4 Teilflächen mit jeweils gleichem Flächeninhalt unterteilt. Abbildung 3.4 verdeutlicht diesen Vorgang anhand eines Beispiels. Hier wurde ein zweidimensionaler Eventraum mit einem Wertebereich von 0 bis 100 pro Dimension in eben zwei Schritten in vier Teilflächen unterteilt.

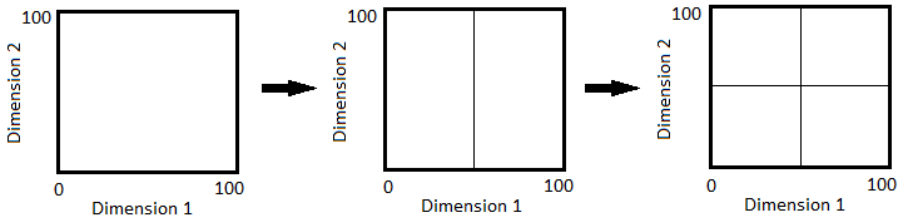

Abbildung 3.4: Räumliche Indizierung eines zweidimensionalen Eventraums

Bei jeder Unterteilung einer Fläche lassen sich die beiden resultierenden Flächen durch ein Bit unterscheiden. Die Fläche mit dem niedrigeren Wertebereich an der entsprechenden Achse, im Beispiel nach dem ersten Schritt der Indizierung also die Fläche, die sich über die x-Werte 0 bis 50 spannt, wird hier durch ein 0 Bit repräsentiert, die Fläche mit dem höheren Wertebereich durch ein 1

Bit. Mit jeder weiteren Unterteilung wird dem bisherigen Bitstring nun das jeweilige Bit angehängt. Jede Fläche wird somit von einem eindeutigen Bitstring repräsentiert, welcher in *PLEROMA* auch als dz-Ausdruck, oder kurz dz, bezeichnet wird [8]. Abbildung 3.5 zeigt hier jeweils die Zuordnung eines solchen dz-Ausdrucks zu allen aus der Indizierung in Abbildung 3.4 entstandenen Flächen.

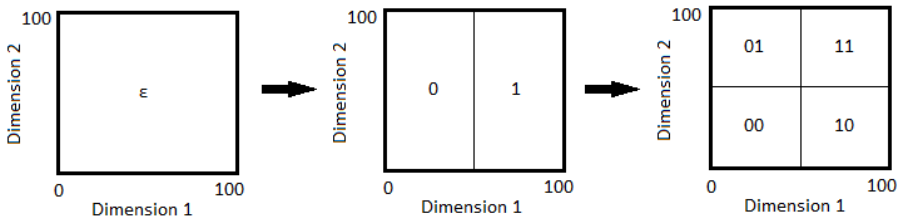

Abbildung 3.5: Räumliche Indizierung mit zugeordnetem dz

In einem zweidimensionalen Eventraum liefert also jedes Bit an ungerader Position eine Information darüber, welchen Bereich die Fläche in Dimension 1 abdeckt, jedes Bit an gerader Position gibt Aufschluss über den abgedeckten Bereich der zweiten Dimension. Betrachtet man hier den Bitstring *1001*, so geben das erste und dritte Bit Aufschluss über den Wertebereich der ersten Dimension der Fläche. Im Eventraum mit Werten von 0 bis 100 pro Dimension würde dies in diesem Beispiel also den Bereich von 50-75 liefern. Das erste Bit indiziert, dass in der ersten Unterteilung hier die Fläche mit dem höherwertigen Bereich, also von 50-100 gewählt wird. Das dritte Bit indiziert, dass bei der nächsten Unterteilung dieser Fläche an der gleichen Achse der niederwertigere Bereich, also 50 bis 75, gewählt wird. Gleiche Informationen über den Wertebereich der zweiten Dimension lassen sich hier dem zweiten und vierten Bit entnehmen. Offensichtlich steigt die Länge eines dz-Ausdrucks mit jedem Schritt der Indizierung an. Je kleiner die durch den dz-Ausdruck repräsentierte Fläche, desto länger ist dieser also. Logischerweise werden mit steigender Anzahl an Dimensionen auch mehr Schritte benötigt, um eine ähnlich kleine Fläche, beziehungsweise einen Raum, zu erhalten. Ziel ist es nun, ein jedes Abonnement und Advertisment, so wie auch jedes Event, durch eben eine solche, durch die räumliche Indizierung entstandene Fläche zu repräsentieren. Ein Event ist im Eventraum allerdings als ein Punkt, und weniger als eine Fläche zu betrachten. Um diesen Punkt möglichst präzise zu repräsentieren muss folglich eine maximal kleine Fläche gewählt werden, somit also auch ein dz-Ausdruck maximaler Länge. Abonnements und Advertisments hingegen haben in einem zweidimensionalen System bereits die Form einer Fläche. Allerdings ist nicht gegeben, dass durch die räumliche Indizierung zu jedem Abonnement eine äquivalente Fläche entstanden ist. Abbildung 3.6 zeigt hier ein Abonnement, welches unmöglich durch nur einen dz-Ausdruck repräsentiert werden kann. Folglich kann es notwendig sein, ein Abonnement durch mehrere dz-Ausdrücke zu repräsentieren. Dies ist ebenfalls in Abbildung 3.6 zu sehen. Jeder dieser dz-Ausdrücke repräsentiert in diesem Fall eine Teilfläche des gesamten Abonnements. Analoges gilt für ein Advertisment. Die Sammlung aller dzs, die ein Abonnement eines Abonnenten A darstellen, wird hier als $DZ(A)$ bezeichnet. Die Sammlung aller dzs, die ein Advertisment eines Publisher P darstellen, wird analog mit $DZ(P)$ bezeichnet [8].

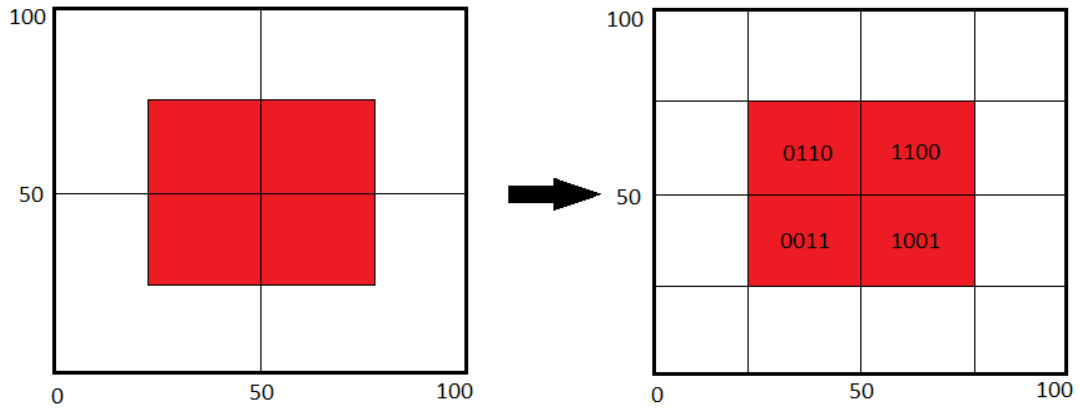

Abbildung 3.6: Unterteilung eines Abonnements in mehrere dz-Ausdrücke

3.3.2 Hierarchie von dz-Ausrücken

Die räumliche Indizierung liefert, wie zuvor erwähnt, eine Rasterhierarchie. Jedes Feld dieses Rasters wird durch ein dz-Ausdruck repräsentiert. Jede dieser Flächen enthält folglich alle Flächen, die durch weitere rekursive Schritte der Indizierung besagter Fläche entstehen. Diese Hierarchie lässt sich auch aus den jeweiligen dz-Ausdrücken der Flächen ableiten. Da die Größe einer Fläche sich auf die Länge des jeweiligen dz-Ausdrucks auswirkt, ist es nicht möglich, dass eine Fläche eine weitere Fläche enthält, deren dz-Ausdruck kürzer als ihr eigener ist. Ein dz_i enthält hier ein weiteres dz_j genau dann, wenn dz_i ein Präfix von dz_j ist. Man schreibt hier $dz_i \succ dz_j$. Decken also zwei verschiedene dz-Ausdrücke dz_i und dz_j eine gemeinsame Teilfläche ab, so folgt , dass entweder $dz_i \succ dz_j$ oder $dz_j \succ dz_i$. Die gemeinsame Teilfläche dieser beiden dzs entspricht hier genau der durch den kürzeren der beiden dz-Ausdrücke repräsentierten Fläche. Durch die rekursive Indizierung folgt vor allem, dass sich zwei hierdurch entstehende Flächen nie miteinander überschneiden können. Überschneiden bedeutet hier, dass zwei verschiedene dzs dz_i und dz_j zwar eine gemeinsame Teilfläche haben, aber weder $dz_i \succ dz_j$ noch $dz_j \succ dz_i$ gilt. Sind zwei dz-Ausdrücke dz_i und dz_j identisch, so wird $dz_i = dz_j$ geschrieben [8]. Alle dzs, die ein Abonnement eines Abonnenten A repräsentieren, also alle dzs in $DZ(A)$, können vor allem nicht miteinander in einer der eben beschriebenen Relationen stehen.

3.3.3 Spannbaum

Die Bestimmung des Pfades, über welches ein Event von einem Publisher zu einem relevanten Abonnenten geleitet werden soll, wird hier vor allem von zwei Kriterien beeinflusst, namentlich der Latenz und der Bandbreite. Erstrebenswert ist es logischerweise, die zu installierenden Flows so anzubringen, dass ein Event eines Publishers jeweils zu jedem Abonnent über den Pfad mit der minimalen Latenzzeit geleitet wird. Wird das Event allerdings über sehr viele verschiedene Pfade zu den unterschiedlichen Abonnenten geleitet, so kann die Bandbreite nicht effizient genutzt werden.

Wünschenswert ist es also, gemeinsame Teilpfade zu verwenden. Hierfür verfügt der *Configurator* in *PLEROMA* über einen Spannbaum bestehend aus allen Switches des Netzwerkes und Pfaden zwischen diesen mit geringer Latenz [7]. Hierbei ist auch der Aufwand bei Rekonfiguration der Flowtabellen deutlich reduziert. Dieser beschränkt sich nämlich lediglich auf die Kanten, welche im Spannbaum vertreten sind.

3.3.4 Eingang eines Advertisments / eines Abonnements am Controller

Plant eine Entität A des Netzwerkes nun, gewisse Daten über das Netzwerk zu senden oder das Senden gewisser Daten über das Netzwerk zu beenden, so sendet diese zuvor ein Advertisment, beziehungsweise die Kündigung eines solchen, an den SDN-Controller. Will eine Entität B des Netzwerkes künftig bestimmte Events erhalten, oder ein aktuelles Abonnement kündigen, so sendet diese analog ein Abonnement, beziehungsweise die Kündigung eines solchen, an den SDN-Controller. Sowohl Advertisment und Abonnement, als auch die jeweiligen Kündigungen, bestehen hier also aus den dz-Ausdrücken, die die neu angebotene oder abonnierte Fläche, oder eine bereits angebotene beziehungsweise abonnierte, zu entfernende Fläche, repräsentieren. Der *Dispatcher* nimmt eine solche Nachricht in zuvor beschriebener Manier entgegen, und identifiziert zunächst, ob es sich hierbei um ein Abonnement, ein Advertisment, die Kündigung eines Abonnements oder um die Rücknahme eines Advertisments handelt. Daraufhin leitet er die Anfrage an der *Configurator* weiter. War die Entität, von der ein solches Abonnement oder ein Advertisment ausging, zuvor weder als Publisher noch als Abonnent vertreten, muss diese nun dem Spannbaum hinzugefügt werden. Im Falle einer Kündigung muss die Entität stattdessen aus dem Spannbaum entfernt werden, solange sie aktuell keine weiteren Events versendet, beziehungsweise keine weiteren aktiven Abonnements hält. Einer jeden Entität in diesem Spannbaum werden ihre jeweiligen Abonnements und Advertisments zugeordnet. Kommt hier also nun ein neues Abonnement hinzu, so gleicht der *Configurator* dieses mit allen im Spannbaum vorliegenden Advertisments ab, um festzustellen, ob aktuell überhaupt Events versendet werden, die für dieses neue Abonnement relevant sind [7]. Ist dem nicht der Fall, so ist das neue Abonnement zu diesem Zeitpunkt redundant und folglich müssen auch keine neuen Flows auf den Netzwerkswitches installiert werden. Hier besteht natürlich die Möglichkeit, dass dieses Abonnement durch den späteren Eingang eines neuen Advertisments wieder relevant wird. Der Eingang eines solchen Advertisments wird analog wie der eines Abonnements behandelt. Hier wird dieses stattdessen lediglich mit allen vorliegenden Abonnements abgeglichen. Im Folgenden wird dieser Abgleich für ein neu eingehendes Abonnement eines Abonnenten A genau beschrieben. Bis auf kleine, eben beschriebene Unterschiede weicht die Verarbeitung eines neu eingehenden Advertisments hiervon aber nicht ab. Zunächst muss klar sein, dass hier jedes dz_i aus $DZ(A)$ sequentiell betrachtet und verarbeitet wird. Um zu entscheiden, ob auf einem oder mehreren Switches ein neuer Flow zu installieren ist, muss jedes dz_i aus $DZ(A)$ mit allen zu einem aktuell vorliegenden Advertisment gehörenden dz_j abgeglichen werden. Auf den Switches zwischen Abonnent A und dem Publisher B eines solchen Advertisments müssen ein oder mehrere Flows installiert werden, wenn in $DZ(B)$ mindestens ein dz_j existiert, für welches entweder $dz_i \succ dz_j$, $dz_j \succ dz_i$ oder $dz_i = dz_j$ gilt. Die von einem Flow repräsentierte Fläche muss hier genau der gemeinsamen Teilfläche des dz_i aus $DZ(A)$ und des dz_j aus $DZ(B)$ entsprechen. Besitzt ein dz_i aus $DZ(A)$ eine gemeinsame Teilfläche mit mehreren dz_j aus $DZ(B)$, so muss für jede dieser Teilflächen ein separater Flow installiert werden. Gilt hier $dz_i = dz_j$, so ist für den neuen Abonnenten A jedes von Publisher P gesendete Event in dz_j relevant. Die von dem zu installierenden Flow abzudeckende Fläche entspricht hier also dz_i beziehungsweise dz_j. Gilt $dz_i \succ dz_j$, so ist für den Abonnenten A ebenfalls jedes von Publisher

P gesendetete Event in dz_j relevant. Da aber nur Events im Bereich von dz_j gesendet werden, ist die verbleibende Fläche von dz_i redundant. Die von dem zu installierenden Flow abgedeckte Fläche entspricht hier also genau dz_j. Gilt $dz_j \succ dz_i$, so ist für den Abonnenten A nun nicht jedes von Publisher P gesendete Event in dz_j, sondern lediglich die Teilmenge an Events, die in dz_i liegt, relevant. Von P versendete Events, die nicht in dz_i liegen, müssen also folglich auch nicht an diesen Abonnenten weitergeleitet werden. Der zu installierende Flow repräsentiert hier also genau dz_i. Selbes Verfahren kann auch verwendet werden, um zu ermitteln, welche Flows gelöscht werden müssen, falls ein aktives Abonnement gekündigt wurde oder ein Advertisment zurückgezogen wurde. Tritt nun einer dieser drei soeben beschriebenen Fälle ein, so ermittelt der *Configurator* alle Switches auf dem Pfad im Spannbaum zwischen betroffenem Abonnenten und Publisher, sowie pro Switch den jeweiligen Ausgangsport, um den nächsten Switch auf dem Pfad zu erreichen. Die Flowtabellen aller dieser betroffenen Switches müssen nun um den neuen Flow erweitert werden. Hierbei wird jedem Flow der berechnete Ausgangsport zugeordnet. Allerdings ist es nicht immer nur genüge, den neuen Flow hinzuzufügen. Gegebenenfalls müssen hierbei auch bereits installierte Flows gelöscht, beziehungsweise modifiziert werden. Somit ist es also notwendig, dass der *Configurator* Kenntnis über den aktuellen Stand der Flowtabellen aller Switches besitzt. Im Besonderen kann sich hier das Hinzufügen eines neuen Flows auf die zu verwendenden Ausgangsports eines bereits installierten Flows auswirken. Wird die Fläche eines neuen Flows beispielsweise bereits von einem installieren Flow abgedeckt, dessen Ausgangsport mit dem des neuen Flows übereinstimmt, so ist auch hier keine Addition des neuen Flows notwendig. Im folgenden Verlauf dieser Arbeit wird mit der Bezeichnung Flow auch direkt die von diesem Flow abgedeckte Fläche bezeichnet.

3.3.5 Umwandlung eines dz-Ausdruckes in eine IPv6-Adresse

In dieser Arbeit wurde zuvor bereits kurz darauf eingegangen, dass durch OpenFlow[4] nur Vergleiche mit vordefinierten Headerfeldern ermöglicht werden. Folglich ist es nicht möglich das dz eines Events in die Paketdaten zu schreiben und diesen Wert dann mit dem dz eines Floweintrages zu vergleichen. Die Information des dz-Ausdrucks muss so umgewandelt werden, dass sie in einem von OpenFlow akzeptierten Headerfeld speicherbar ist. *PLEROMA*[7] verwendet hierfür IPv6-Multicast-Adressen [7]. dzs werden also in IPv6-Adressen umgewandelt und in das entsprechende Headerfeld eingetragen. Auch die Flowtabellen der Switches werden so nicht mit den Bitstrings, sondern mit den entsprechenden IPv6-Multicast-Adressen befüllt. Hierbei wird das dz in Blöcke aus jeweils 4 Bit unterteilt. Ist die Länge des dzs kein Vielfaches von 4, so wird der letzte Block mit 0 Bits aufgefüllt. Jeder der entstandenen Blöcke wird nun in eine Hexadezimalzahl konvertiert. Das Präfix *ff0e* sowie die aneinandergereihten Hexadezimalzahlen bilden die resultierende IPv6-Multicast-Adresse. Beispielsweise wird der dz-Ausdruck *101101* somit in die Hexadezimalzahlen B (*1011*) und 4 (*0100*) konvertiert. Die zugehörige IPv6 Adresse lautet also *ff0e:b400::/22*. Nach dem gleichen Prinzip wird der dz-Ausdruck *101* somit in die IPv6 Adresse *ff0e:a000::/19* umgewandelt [7]. Ein Treffer beim Vergleichen von Headerfeld und Tabelleneintrag kommt hier nun zustande, wenn das Netzwerkpräfix der IPv6-Adresse des Tabelleneintrages ein Präfix des Netzwerkpräfixes der IPv6-Adresse des versendeten Events ist. Somit passt der Tabelleneintrag *ff0e:a000::/19* also zu einem eingehenden Event mit IP-Adresse *ff0e:b400::/22*. Gewählt wird hier im Falle mehrerer Treffer der Eintrag, dem die höchste Priorität zugeteilt wurde.

3.3.6 Dimensionsauswahl

In den bisherigen Beispielen wurde einfachheitshalber nur von zweidimensionalen Eventräumen ausgegangen. Mit der Erhöhung der Dimensionen steigt aber auch die Länge der *dzs*, solange jede Dimension bei der Indizierung in gleich viele verschiedene Bereiche wie zuvor unterteilt werden soll. Wird jede Dimension in einem zweidimensionalen Raum einmal unterteilt, so enstehen hierdurch vier Flächen. Um jeder Fläche ihr einzigartiges *dz* zuzuordnen, werden hierfür 2 Bit benötigt. Im dreidimensionalen Raum entstehen durch gleiches Vorgehen bereits acht Unterräume, folglich werden hier 3 Bit benötigt, um jedem Unterraum ein eindeutiges *dz* zuzuteilen. Außerdem ist ein *dz* wie erwähnt länger, je kleiner die durch es repräsentierte Fläche, beziehungsweise der repräsentierte Raum ist. Offensichtlich ist es nicht möglich, beliebig lange *dzs* für das Filtern zu verwenden. Somit kann also immer eine gewisse Ungenauigkeit in der Repräsentation eines Abonnements oder eines Advertisments existieren. Durch die Nutzung eines Headerfeldes zum Filtern auf Hardwareebene wird diese Beschränkung der Länge eines *dzs* allerdings noch weiter verschärft. Eine IPv6-Adresse besteht aus 128 Bit. Die ersten 16 Bit sind hier für das Multicast Präfix in Verwendung. Verbleibend sind also 112 Bit, die für die Kodierung eines *dz* genutzt werden können. Offensichtlich impliziert dies eine maximal mögliche Länge eines *dz*, welches trotz allem noch in eine eindeutige IPv6-Adresse konvertierbar ist. Daraus folgen eine Beschränkung der Dimensionen sowie eine maximal mögliche Anzahl an Schritten der räumlichen Indizierung. Je mehr Dimensionen betrachtet werden, in desto weniger Flächen kann der Eventraum unterteilt werden. Hierdurch werden Abonnements von sehr kleinen Flächen gegebenenfalls nur sehr unpräzise repräsentiert. Auch Events werden wie bereits erwähnt als ein *dz* maximaler Länge dargestellt. Auch diese werden folglich mit zunehmender Anzahl an Dimensionen immer unpräziser repräsentiert. Da dies nicht wünschenswert ist, werden in *PLEROMA* nicht alle möglichen Dimensionen für das Filtern in Betracht gezogen und die räumliche Indizierung somit auch nur auf eine Teilmenge der Dimensionen angewendet [7]. Ein Nebeneffekt hiervon sind Falsch-Positive. Ein Falsch-Positiv ist ein Event, dass an einen Abonnenten geleitet wird, obwohl dieser aktuell für den Inhalt des Paketes kein Abonnement getätigt hat. Betrachten wir nun Events bestehend aus 3 verschiedenen Attributen A, B und C mit einem Wertebereich von jeweils 0 bis 100. Werden für das Filtern nun nur Attribute A und B in Betracht gezogen und ein Abonnent ist interessiert an Events im Bereich *A = 0–50, B = 0–50, C = 0–50*, so wird dieser hierdurch zwangsweise alle Events im Bereich *A = 0–50, B = 0–50 ,C = 0–100* erhalten. Zwar wird er hierdurch keine für ihn interessanten Events verpassen, alle Events mit Werten des Attributes C im Bereich von 51 und 100 sind für ihn aber irrelevant. Erstrebenswert ist eine Minimierung dieser Falsch-Positiv-Rate. Deshalb werden die Dimensionen, die beim Filtern relevant sind, nicht zufällig gewählt. Das entscheidende Kriterium ist hier die Aussagekraft einer Dimension, also die Fähigkeit einer Dimension Falsch-Positive zu vermeiden. Diese wird vor allem von zwei Faktoren beeinflusst. Einerseits ist das die Verteilung der verschiedenen Abonnements in dieser Dimension. Wenn alle Abonnements in einer Dimension den gleichen Bereich abdecken, so ist diese Dimension deutlich weniger relevant, als eine solche, in der die Abonnements in ihren jeweils abgedeckten Bereichen stark variieren. Andererseits spielt auch die Anzahl der an der Dimension interessierten Abonnenten eine Rolle. Natürlich kann es sein, dass Abonnenten existieren, für die die Werte einer bestimmten Dimension nicht relevant sind und diese folglich einfach den gesamten Wertebereich dieser Dimension abonnieren. Gehören nun alle Abonnements, die in dieser Dimension nur Teilbereiche abonnieren, dem gleichen Abonnenten an, so ist diese Dimension auch bei hoher Verteilung der verschiedenen Abonnemente weniger

relevant. Zur Bekämpfung dieser Falsch-Positive wird in *PLEROMA* außerdem bereits ein hybrider Ansatz[19] präsentiert, der das Filtern zwischen Daten- und Kontrollebene aufteilt. Hierdurch wird ein Kompromiss zwischen Genauigkeit und Latenz des Systems angestrebt.

4 P4 Publish/Subscribe

4.1 Systemmodell

Die grundlegende, im vorherigen Kapitel behandelte Logik, um zu bestimmen, welchen Switches eines Netzwerkes ein bestimmter Flow bei eingehendem Abonnement oder Advertisment hinzugefügt werden soll, beziehungsweise von welchen Switches dieser bei einer Abonnementkündigung gelöscht werden muss, bleibt auch für das in diesem Kapitel beschriebene System weitestgehend analog. Einzig das Format der eingehenden Abonnements und Advertisments, sowie die Repräsentation eines Flows müssen hierfür modifiziert werden. Aus diesen Gründen ist das Systemmodell sehr ähnlich zu dem von *PLEROMA*[7][8] in Kapitel 3 gehalten. Durch die Verwendung von P4-programmierbaren Switches ist es nun möglich, auf räumliche Indizes zu verzichten. Ein eingehendes Abonnement muss somit also nichtmehr auf die durch diese Indizierung entstandenen Flächen aufgeteilt werden. Stattdessen ist es nun möglich, ein solches Abonnement direkt durch die jeweiligen Unter- und Obergrenzen einer jeden Dimension darzustellen. Aus diesem Grund können alle Abonnements und Events sehr präzise repräsentiert werden. Die Werte der Attribute eines Events werden in einen speziell hierfür angelegten Header geschrieben und am Switch mit den vorliegenden Floweinträgen verglichen. Ein solcher Floweintrag ist nun für ein System mit 2 Dimensionen von der Form $A = 10 - 50 \&\& B = 50 - 70$. Für jedes Attribut eines Events wird also überprüft, ob dessen Wert in dem durch den Floweintrag vorgegebenen Wertebereich liegt. Ist dies bei einem bestimmten Floweintrag für jedes Attribut eines Events der Fall, so wird hier von einem Treffer gesprochen.

Weiterhin besteht die Middleware hier aus einem *Dispatcher* und *Configurator* [7]. Ersterer nimmt die Advertisments angehender Publisher, sowie die Abonnements angehender Abonnenten entgegen. Sowohl bei Advertisments, als auch bei Abonnements, schreibt der zugehörige Publisher oder Abonnent die Werte, welche die angebotene oder abonnierte Fläche repräsentieren, in die Nutzdaten eines Paketes. Bei Eingang eines solchen Paketes am Controller liest der *Dispatcher* diese Werte aus und identifiziert außerdem, ob ein Advertisment oder ein Abonnement vorliegt. Hiernach wird das Abonnement oder das Advertisment an den *Configurator* übergeben. Dieser berechnet, wie auch in *PLEROMA*, die für die Installation relevante Fläche, und die zugehörigen relevanten Switches. Der *Configurator* verfügt hier für jeden Switch des Netzwerkes über einen sogenannten *Container*, in dem er einen Überblick über alle auf dem jeweiligen Switch installierten Flows bewahrt. Wird nun ein neuer Flow an einen dieser *Container* weitergeleitet, so müssen unter Umständen Modifikationen an bereits installierten Flows getätigt werden. Diese hierfür zuständige, in der Kontrollebene angesiedelte Logik stellt den Hauptteil dieser Arbeit dar und wird im weiteren Verlauf dieses Kapitels detailliert beschrieben.

Wünschenswert für ein solches System ist logischerweise die dynamische Modifikation der Flowtabellen eines Switches bei neu eingehenden Abonnements oder Kündigung eines solchen während der Laufzeit. Mit P4 ist es generell zunächst nicht möglich, einen zentralen Controller die Tabellen eines durch P4 konfigurierten Switches zur Laufzeit verwalten zu lassen. *P4Runtime*[22]

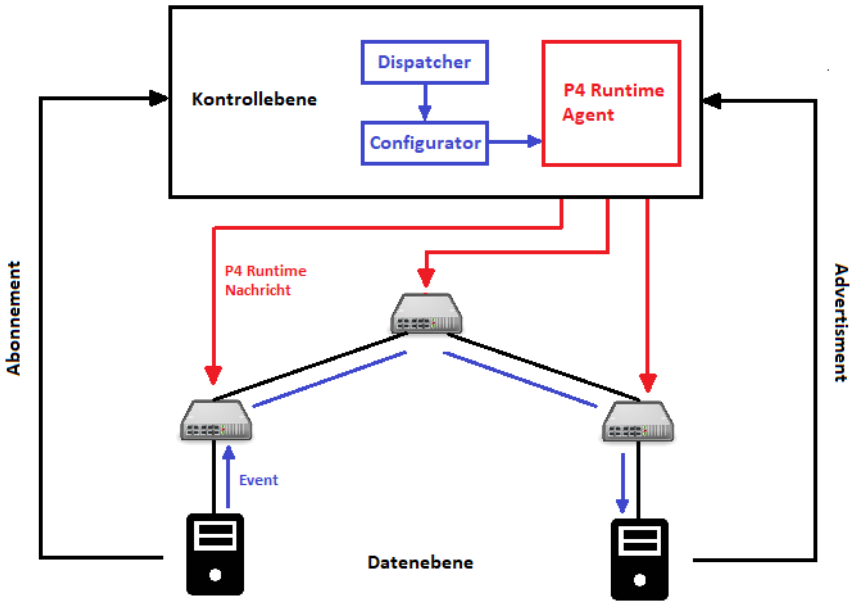

Abbildung 4.1: P4 Publish/Subscribe Systemmodell

übernimmt hier die Funktion, die in der Implementierung von *PLEROMA* von OpenFlow[4] erfüllt wurde, und stellt somit eine API zu Verfügung, über welche der Kontrollebene die Kommunikation mit einem durch P4 programmierten Switch ermöglicht wird. Kompiliert ein P4 Compiler ein P4 Programm, so identifiziert dieser hierbei automatisch die Teile des Programmes, auf die eventuell zur Laufzeit zugegriffen werden muss. Des Weiteren besteht die Möglichkeit, die bereits vorliegende P4 Pipeline hier durch weitere Tabellen zu erweitern. In unserem Fall ist hier nur die Rekonfiguration der bereits definierten Tabellen, beziehungsweise deren Floweinträgen, relevant. Hierfür erstellt P4Runtime also für jede Tabelle ein *Protobuf* Schema. *Protobuf* ist ein flexibles Datenformat, welches zur Serialisierung mit einer Schnittstelle verwendet wird. Soll nun beispielweise ein Eintrag zu einer Tabelle hinzugefügt werden, muss dieser Eintrag zunächst in besagtes Protobuf Schema umgewandelt werden. Spezifiziert werden müssen hier vor allem die Tabellen ID, die ID der auszuführenden Aktion und in dem Fall dieses Systems, die Wertebereiche einer jedem Dimension, mit denen eingehende Events verglichen werden sollen. Wurden nun nach dem Eingang eines neuen Flows alle betroffenen Container aktualisiert, so müssen nun sämtliche hier entstandenen Modifikationen über die P4Runtime API an die Datenebene weitergeleitet werden.

4.2 P4 Programm

Die Korrektheit des Systems wird maßgeblich von dem auf den Switches installierten P4 Programm beeinflusst. Hierbei wird natürlich auf allen Switches eines Netzwerkes dasselbe Programm aufgespielt. Für die Zwecke dieser Arbeit kann besagtes Programm weitestgehend simpel gehalten werden. Folgender Abschnitt beschreibt den Aufbau des verwendeten Programmes und rechtfertigt, warum das verwendete Design für dieses System die meisten Vorteile bringt.

4.2.1 Publish/Subscribe-Header

Essentiell ist hier zunächst die Definition eines für das Publish/Subscribe System ausgelegten Headers, um das spätere Vergleichen von eingehenden Events mit den in diesem Header spezifizierten Feldern zu ermöglichen. Der Publish/Subscribe-Header für ein zweidimensionales System ist untenstehendem Listing 4.1 zu entnehmen.

```
header_type pubsub_t {
    fields {
        value_a : 16;
        value_b : 16;
    }
}
```

Listing 4.1: Defintion des Publish/Subscribe-Header in $P4_{14}$

Hierbei müssen alle Felder dieses Headers definiert werden, sowie jedem dieser Felder auch die jeweilige Länge in Bits zugeteilt werden. Ansonsten ist es dem Parser nicht möglich, den Paketheader bei Empfang eines Paketes zu extrahieren, beziehungsweise die empfangenen Daten den jeweilig korrekten Headerfeldern zuzuordnen. Die Anzahl der benötigten Headerfelder entspricht im Kontext von Publish/Subscribe der Anzahl der Dimensionen des Systems. Diese Anzahl muss somit zu Beginn festgelegt werden und ist nur durch weitere Kompiliervorgänge änderbar. Der Header in Listing 4.1 enthält hier 2 verschiedene Felder. Das System verfügt somit über 2 Dimensionen. Die gewählte Länge von 16 Bit eines jedes Feldes beschränkt den Bereich des Eventraumes auf das Intervall zwischen 0 und 65.535 pro Dimension, kann aber nach Belieben angepasst werden. Auch dies ist allerdings folglich nicht dynamisch ohne erneute Kompilierung möglich.

4.2.2 Parser

Nach Definition des Headers muss der Parser, sowie die gewünschte Parse-Reihenfolge spezifiziert werden. Die Parse-Reihenfolge wird maßgeblich vom Format der für das System relevanten Pakete beeinflusst. Diese sind logischerweise solche Events, die von Publisher zu Abonnent geleitet werden müssen und bestehenden in der Evaluation in Kapitel 5 aus dem Ethernet-Header, dem IPv4-Header, einem UDP-Header, gefolgt von dem eigendefinierten Publish/Subscribe-Header, der die Informationen über die Position des Events im Eventraum trägt. Enthält ein Paket die vom Parser spezifizierte Headerreihenfolge nicht, so ist es hier auch nicht relevant für dieses System.

Die eben aufgelisteten Header werden hier in Reihe geparst, beginnend mit dem Ethernet-Header. Der Ethernet Parser muss hierfür als Startparser definiert werden. Die Abfolge der verschiedenen Parser wird in Abbildung 4.2 als Parse-Graph dargestellt. Nachdem dieser Header extrahiert und geparst wurde, kann nun der Wert des EtherTypes überprüft werden. Indiziert dieser einen folgenden IPv4-Header, so kann mit dem IPv4 Parser fortgesetzt werden. Anderenfalls kann das Packet direkt dem Ingress Kontrollfluss übergeben und dort fallengelassen, oder wie bereits in Kapitel 2 erwähnt, direkt durch keine Weiterleitung an Parser oder Kontrollfluss ignoriert werden. Analoges gilt daraufhin für das Parsen der weiteren Header. Folgendes Listing zeigt hier die Definition des Parsers für den Publish/Subscribe-Header.

```
header pubsub_t pubsub;

parser parse_pubsub {
    extract(pubsub);
    return ingress;

}
```

Listing 4.2: Publish/Subscribe Parserdefinition in $P4_{14}$

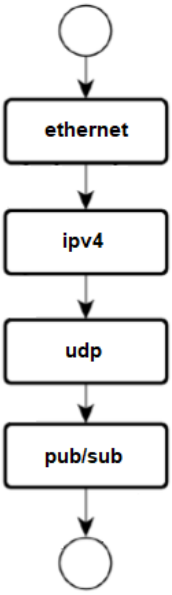

Abbildung 4.2: P4 Parse-Graph für ein Publish/Subscribe System

4.2.3 Tabellen

Die erste größere Designentscheidung fällt hier nun bei der Definition der für alle eingehenden Events zuständigen Tabelle, beziehungsweise Tabellen, an. Hierfür werden zwei verschiedene Herangehensweisen skizziert. Einerseits ist es möglich, die Ingress Pipeline für Events auf nur eine einzige Tabelle zu reduzieren. In dieser Tabelle müssen folglich die Werte jeder Dimension eines Events ausgewertet werden. In einem System mit zwei Dimensionen A und B hätte ein Eintrag in der Flowtabelle eines Switches hier also die Form $A = 20–80 \;\&\&\; B = 0–100$. Die Definition einer solchen Tabelle kann folgendem Listing 4.3 entnommen werden.

```
table publish_subscribe {
    reads {
        pubsub.value_a : range;
        pubsub.value_b : range;
    }
    actions {
        set_outport;
        multicast;
        _drop;
    }
    size: 65535;
}
```

Listing 4.3: Publish/Subscribe Tabellendefinition in $P4_{14}$

Definiert werden müssen hier also zunächst alle Felder der Header des eingehenden Paketes, die für das Filtern in dieser Tabelle relevant sind. Außerdem müssen schon hier alle Aktionen festgelegt werden, die bei eventuellen Treffern beim Vergleichen ausgeführt werden können.
Einen Nachteil, den diese Herangehensweise mit sich bringt, ist die Anzahl an Vergleichen, die getätigt werden muss, bis erkannt werden kann, dass ein Event zu keinem Tabelleneintrag passt und somit fallengelassen werden kann. Bei 100 Tabelleneinträgen in einem System mit 2 Dimensionen wird das Event hier also mit allen Einträgen abgeglichen. Pro Eintrag werden 2 Attribute verglichen. Nach 200 Vergleichen besteht also erst Klarheit, dass dieses Paket für keinen Abonnenten relevant ist. Die zweite Herangehensweise splittet ein Abonnement hier in seine unterschiedlichen Attribute auf. Hierfür wird pro Dimension eine eigene Tabelle definiert. Diese Tabellen können nun in Reihe durchlaufen werden. Jede Tabelle ist hierbei für die Auswertung einer Dimension zuständig, vergleicht also nur ein einziges Attribut des Events mit ihren Tabelleneinträgen. Begonnen wird hier mit der Tabelle, die das Attribut der ersten Dimension auswertet. Logischerweise wird das Setzen des Ausgangsportes eines jeden Paketes, das alle Tabellen durchlief, durch die Kombination der Treffer einer jeden dieser Tabellen bestimmt. Genauer gesagt, auch wenn zwei verschiedene Pakete in der letzten Tabelle einen Treffer mit dem gleichen Tabelleneintrag erzielen, werden diese nicht zwangsweise auch über den gleichen Port weitergeleitet, da sie in den vorherigen Tabellen unterschiedliche Treffer erzielt haben könnten. Hierdurch muss also auch beim Durchlaufen der letzten Tabelle noch Kenntnis über alle vorherigen Treffer bestehen. Es bietet sich hierfür also an, einem jeden Paket einen Zustand zuzuteilen, der nach dem Durchlaufen einer jeden Tabelle entsprechend aktualisiert wird. In Betracht gezogen werden nun also für alle Tabellen, außer logischerweise der ersten, nicht nur das jeweilige Attribut, sondern zusätzlich auch der aktuelle Zustand des Paketes. Die erste Tabelle nimmt nun ein Paket entgegen, vergleicht das jeweilige Attribut mit ihren Tabelleneinträgen und weist dem Paket daraufhin einen entsprechenden Zustand zu. Die folgenden Tabellen werten nun das jeweilige Attribut und den aktuellen Zustand aus, und modifizieren den Zustand entsprechend. Die letzte Tabelle wertet Zustand und Attribut aus, und weist daraufhin einen entsprechenden Ausgangsport zu. Hierdurch ist es möglich, dass im Vergleich zu erster Herangehensweise in manchen Fällen deutlich weniger Vergleiche notwendig sind, bis ein Paket fallengelassen werden kann. Geht man hier davon aus, dass jede Tabelle in einem System mit 2 Dimensionen über 100 Einträge verfügt, so kann ein Event, dessen Attribut der ersten Dimension bereits zu keinem Eintrag dieser Tabelle passt, bereits nach dem Abarbeiten der 100 Einträge der ersten Tabelle fallengelassen werden. Da die Attribute hier aber über die Tabellen verteilt sind, sind hierfür nun im Gegensatz zu vorherigem Ansatz nur 100 Vergleiche

notwendig. Muss allerdings eine für Attribut B zuständige Tabelle eine bestimmte Operation für alle eingehenden Pakete mit B = 20-30 ausführen, so kann dies in Abhängigkeit von der Anzahl möglicher Zustände nach vorheriger Tabelle zu einer eventuell großen Anzahl an Einträgen führen. Dieser Anstieg wirkt sich besonders stark auf die am Ende der Ausführreihenfolge stehenden Tabellen aus. Dies soll durch ein kurzes Beispiel vereinfacht dargestellt werden: Es liegen an einem Switch zwei Abonnements mit $A = 0-30 \&\& B = 20-40$ und $A = 0-40 \&\& B = 20-30$ vor. Beide Abonnements gehören unterschiedlichen Abonnenten an, die über unterschiedliche Ausgangsports zu erreichen sind. Ersteres Abonnement soll hier über Port 1 geleitet werden, letzteres über Port 2. Folglich müssen Events mit Werten im Bereich $A = 0-30 \&\& B = 20-30$ hier über beide Ausgangsports geleitet werden, da sie zu beiden Abonnements passen, sprich die abgedeckte Fläche beider Abonnements überschneidet sich hier in genau diesem Bereich. Events mit Werten im Bereich $A = 0-40 \&\& B = 31-40$ hingegen sind hier nur für das erste der beiden Abonnements relevant. Auch hier gibt es zwei verschiedene Varianten, die jeweiligen Tabellen zu bestücken. Offensichtlich muss hier in der ersten Tabelle zwischen den Bereichen 0-40 und 0-30 unterschieden werden. Einerseits können hier also zwei Einträge mit 0-30 und 31-40 erstellt werden, andererseits ist es auch möglich die Einträge 0-30 und 0-40 hinzuzufügen, und dem ersten Eintrag (0-30) eine höhere Priorität zuzuordnen. Diese Priorität gibt also an, welcher Eintrag gewählt werden soll, wenn Werte eines Events zu mehreren Tabelleneinträgen passen. Letztere Variante vereinfacht die Rekonfiguration der Tabellen bei neu eingehenden Abonnements stark, zur Vereinfachung und besseren Verständlichkeit wird für dieses Beispiel in 4.3 trotz allem die erste Variante verwendet. Die beiden oben genannten Abonnements liefern also folgende Tabellen. Grau unterlegt sind hier pro Tabelle der Zustand eines eingehenden Events, sowie der Zustand eines ausgehenden Events.

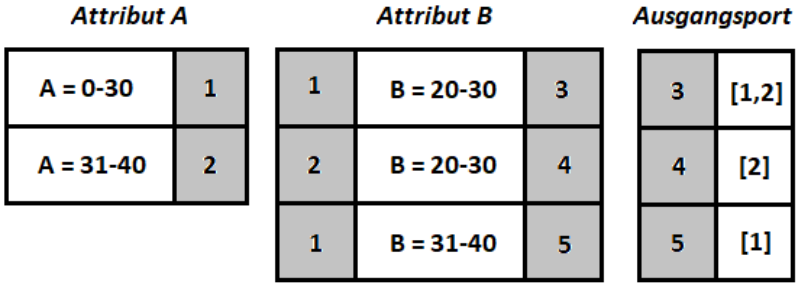

Abbildung 4.3: Verwendung einer Tabelle pro Attribut

Dieses Beispiel zeigt bereits, dass schon bei sehr wenigen Abonnements die Anzahl der Einträge von Tabelle zu Tabelle ansteigt. Dieser Anstieg kann im schlechtesten Fall sogar exponentiell sein. Zusätzlich ist auch das Aktualisieren dieser Tabellen sehr kompliziert und bringt einen großen Mehraufwand mit sich. Vor allem wirkt sich hier die Änderung an einer Tabelle aufgrund der nun veränderten Zustände eventuell stark auf alle folgenden Tabellen und deren Einträge aus. Aus diesem Grund ist die Verwendung einer einzigen Tabelle für das in dieser Arbeit entworfene P4 Publish/Subscribe System deutlich praktikabler.

4.2.4 Aktionen

Abhängig von dem Eintrag der Tabelle, der bei einem eingehenden Paket einen Treffer erzielt, muss auch eine entsprechende Aktion ausgeführt werden. Für unsere Zwecke ist dies für gewöhnlich das Setzen des gewünschten Ausgangsports bei Übereinstimmung mit einem der Floweinträge, oder das Fallenlassen eines Paketes, falls kein Treffer erzielt werden kann. Diese Aktionen können in P4 wie folgt definiert werden.

```
action _drop() {
    drop();
}
```

Listing 4.4: Aktion zum Fallenlassen von Paketen in $P4_{14}$

Die Drop-Aktion ruft hierbei lediglich die bereits vordefinierte primitive Aktion drop() auf.

```
action set_outport(port) {
    modify_field(standard_metadata.egress_spec, port);
    add_to_field(ipv4.ttl, -1);
}
```

Listing 4.5: Aktion zur Weiterleitung ueber gewuenschten Port in $P4_{14}$

Für das Setzen des Ausgangsports wird der gewünschte Port der Aktion hier als Parameter übergebenen. Ein jeder Tabelleneintrag enthält hierfür neben den Feldern zum Filtern der Pakete auch die für diesen Eintrag auszuführende Aktion, sowie die entsprechenden Parameter, die dieser Funktion hierfür übergeben werden sollen. Um letztendlich zu spezifizieren, über welchen Port das Paket die Egress Pipeline verlassen soll, muss hierfür das Feld $egress\_spec$ mit dem gewünschten Port überschrieben werden. Dieses Feld ist Teil der Standard Metadaten eines jeden Paketes, das die P4 Pipeline durchläuft. Zugriff auf dieses Feld erfolgt hierbei also über $standard\_metadata.egress\_spec$. Das zugehörige Listing zeigt außerdem, wie analog zum Setzen des Ausgangsports weitere Operationen, wie das Dekrementieren des Time-to-Live-Feldes des IPv4-Headers, durchgeführt werden können. Das Setzen nur eines Ausgangsports ist für ein Publish/Subscribe System allerdings nicht immer genüge. Um ein Event mehreren interessierten Abonnenten zuzustellen, ist es in P4 einerseits möglich, ein Paket zu klonen und daraufhin das ursprüngliche Paket weiterzuleiten, das geklonte Paket jedoch wieder in die Ingress oder Egress Pipeline einzureihen. Hiermit ist Multicast bereits realisierbar. Allerdings bringt diese Herangehensweise einerseits einen starken Mehraufwand in der Implementierung, sowie eine variierende Latenz der Pakete. Logischerweise erreicht das ursprüngliche Paket den ersten Abonnenten vergleichsweise schnell, wohingegen die erneut eingereihten, geklonten Pakete mit jedem erneuten Klonvorgang eine höhere Latenz aufweisen werden. Um dies zu vermeiden, ermöglicht P4 es außerdem, einem Paket eine sogenannte Multicast Gruppe zuzuordnen.

4.2.5 Multicast

Die ID einer Multicastgruppe wird einem Paket in P4 anstatt eines Ausgangsports übergeben. Jeder Multicast Gruppe können hier die benötigten Ports zugeteilt werden. Zu betrachten ist hier die *instrinsic_metadata* eines Paketes. Diese sind eine spezielle Instanz der Standard Metadaten eines Paketes. Das relevante Feld dieser Metadaten ist hier vor allem die $mgrp$. Hieraus leitet sich die Notwendigkeit einer weiteren Aktion ab. Diese funktioniert weitestgehend analog zu dieser, die den Ausgangsport eines Paketes setzen soll. Einziger Unterschied ist hier nun, dass der Aktion die ID der Multicast Gruppe übergeben werden muss, welche in oben genanntem Feld gespeichert wird. Das Erstellen einer Multicast Gruppe ist allerdings nicht statisch über das P4 Programm möglich. Diese müssen dem Switch analog zu allen weiteren Einträgen der Flowtabellen über die P4Runtime API übergeben werden. Aufgabe des Controllers ist es hier also ebenfalls, zu überprüfen, welche installierten Flows über mehrere Ausgangsports geleitet werden müssen, und daraufhin für all diese Flows eine entsprechende Multicast Gruppe zu erstellen und an den jeweiligen Switch weiterzuleiten. Auch die Funktionalität des Multicast Mechanismus an sich wird hier nicht in P4 spezifiziert. Diese ist abhängig von der Zielhardware und ist von dieser bereitzustellen.

4.2.6 Kontrollfluss

Nachdem der Parse-Vorgang abgeschlossen ist, wird das Paket, beziehungsweise die geparste Repräsentation, an die Ingress Pipeline übergeben. Der Kontrollfluss definiert hierbei, welche $Match\&Action$-Tabellen ein Paket durchläuft, und vor allem in welcher Reihenfolge dies geschehen soll. Natürlich müssen hier nur für das System relevante Pakete an die Tabellen übergeben werden. Deshalb ist es hierbei wichtig zu überprüfen, ob das geparste Paket auch über einen Publish/Subscribe-Header verfügt. $valid()$ erfüllt hier genau diese Aufgabe. Ist der Publish/Subscribe-Header also vorhanden und gültig, so kann das Paket an die Tabellen übergeben werden, anderenfalls muss es nicht weiterverarbeitet werden, beziehungsweise kann fallengelassen werden. Dies geschieht in P4 14, wie in Kapitel 2 erwähnt, zum Beispiel durch das Weiterleiten an eine Dummy-Tabelle. Eine solche Definition des Ingress Kontrollflusses mit Weiterleitung an eine Dummy-Tabelle drp sieht also wie folgt aus:

```
control ingress {

    if(valid(pubsub)) {
        apply(publish_subscribe);
    } else {
        apply(drp);
    }

}
```

Listing 4.6: Ingress Kontrollfluss fuer das Publish/Subscribe System in $P4_{14}$

Wurden alle nötigen Tabellen durchlaufen, so wird das Paket an die Egress Pipeline übergeben. Für die Zwecke dieser Arbeit sind hier allerdings keine weiteren Modifikationen des Paketes notwendig, weshalb dieser Schritt hier nicht weiter behandelt wird.

4.3 Rekonfiguration der Flowtabellen

Wird nun ein Advertisment, beziehungsweise ein Abonnement, vom Controller entgegengenommen, so wird daraufhin unter Verwendung des vorliegenden Spannbaumes und der Kenntnis über bereits vorliegende Advertisments und Abonnements berechnet, welche Teilfläche der von dem eingehenden Abonnement oder dem eingehenden Advertisment abgedeckten Fläche nun relevant ist. Relevant bedeutet auch hier, dass es nun einen Publisher gibt, der Events mit Werten in dieser Teilfläche sendet, sowie einen Abonnenten, der an Events in dieser Teilfläche Interesse hat. Diese Teilfläche wird nun an die *Container* der jeweiligen Switches auf dem Pfad zwischen Publisher und Abonnent in Form eines Flows weitergeleitet. Dieser Flow enthält die jeweiligen Unter- und Obergrenzen jeder Dimension. Diese stellen also die abgedeckte Fläche dar. Außerdem enthält dieser Flow zusätzlich den zugehörigen Ausgangsport. Ein solcher Flow, der dem *Container* übergeben wird, wird im weiteren Verlauf dieses Kapitels als *Basisflow* bezeichnet. Zur besseren Verständlichkeit des folgenden Kapitels werden nun zunächst einige grundlegende Begriffe definiert. Besonders wird hier auf die Relation zweier Flows zueinander eingegangen.

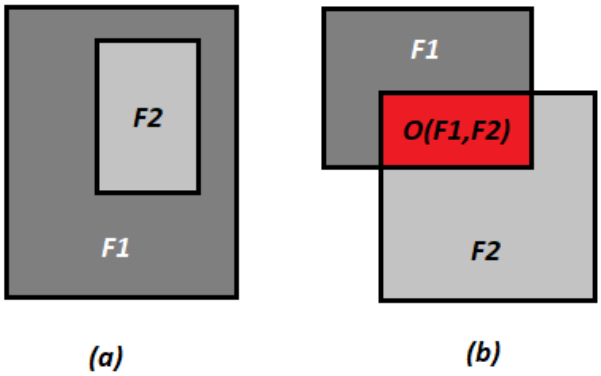

Abbildung 4.4: Relation zweier Flows F1 und F2

Zwei Flows F1 und F2 werden im Folgenden als gleichwertig bezeichnet, wenn deren abgedeckte Fläche identisch ist. Dies bedeutet genauer, dass sowohl Unter-, als auch Obergrenze einer jeden Dimension identisch sind. Insbesondere sind hierbei die Ausgangsports beider Flows irrelevant. Zwei Flows gelten als gleichwertig, selbst wenn sie jeweils verschiedene Ausgangsports besitzen. Ein Flow F1 enthält einen Flow F2, wenn die die von Flow F2 abgedeckte Fläche eine Teilfläche der von Flow F1 abgedeckten Fläche ist (Abbildung 4.4 (a)). Die ist genau dann der Fall, wenn F1 und F2 nicht gleichwertig sind und für jede Dimension k gilt:

$$F1[k].Untergrenze <= F2[k].Untergrenze \;\&\&\; F1[k].Untergrenze >= F2[k].Obergrenze \tag{4.1}$$

Ein Flow F1 liegt folglich in einem Flow F2, wenn dieser Flow F2 Flow F1 enthält. Zwei Flows F1 und F2 überschneiden sich, wenn beide Flows eine gemeinsame Teilfläche besitzen, allerdings weder gleichwertig sind, noch einer der beiden Flows den jeweils anderen enthält (Abbildung 4.4

(b)). Dies ist genau dann der Fall, wenn für jede Dimension wenigstens eines der folgenden drei Kriterien gilt und beide Flows zusätzlich nicht gleichwertig sind:

$$F1[k].Untergrenze >= F2[k].Untergrenze \;\&\&\; F1[k].Untergrenze <= F2[k].Obergrenze \tag{4.2}$$

$$F1[k].Obergrenze >= F2[k].Untergrenze \;\&\&\; F1[k].Obergrenze <= F2[k].Obergrenze \tag{4.3}$$

$$F1[k].Untergrenze < F2[k].Untergrenze \;\&\&\; F1[k].Obergrenze > F2[k].Obergrenze \tag{4.4}$$

Die Teilfläche der beiden überschneidenden Flows wird hier als der *Overlap* bezeichnet. Alle Flows eines Containers können somit in *Basisflows* und *Overlaps* aufgeteilt werden. Ein *Overlap* entsteht hierbei entweder durch den Schnitt zweier *Basisflows*, durch den Schnitt eines *Basisflows* und eines weiteren *Overlaps*, oder aber durch den Schnitt zweier *Overlaps*. Im weiteren Verlauf dieser Arbeit schreiben wir für den *Overlap* zweier Flows *F1* und *F2* auch *O(F1,F2)*. Die Ausgangsports eines *Overlaps* setzen sich hierbei aus den Ausgangsports der beiden Flows zusammen, die diesen *Overlap* verursachen. Zwei Flows F1 und F2 stehen nicht miteinander in Relation, wenn beide Flows keine gemeinsame Teilfläche besitzen, also genau dann, wenn keine der oben genannten Relationen zutrifft.

Alle wichtigen Aspekte, die bei einer Änderung der Flowtabellen eines Switches in Betracht gezogen werden müssen, sollen nun an einem einfach gehaltenen Beispiel erläutert werden. Wir betrachten hierfür erneut ein System mit zwei Dimensionen. Vorhanden sind hier an einem Container zwei *Basisflows* mit *B1: A = 10-50 && B = 10-50* sowie *B2: A = 30-70 && B = 30-70*. Abbildung 4.5 zeigt die Lage der von den *Basisflows B1* und *B2* abgedeckten Flächen im zweidimensionalen Eventraum.

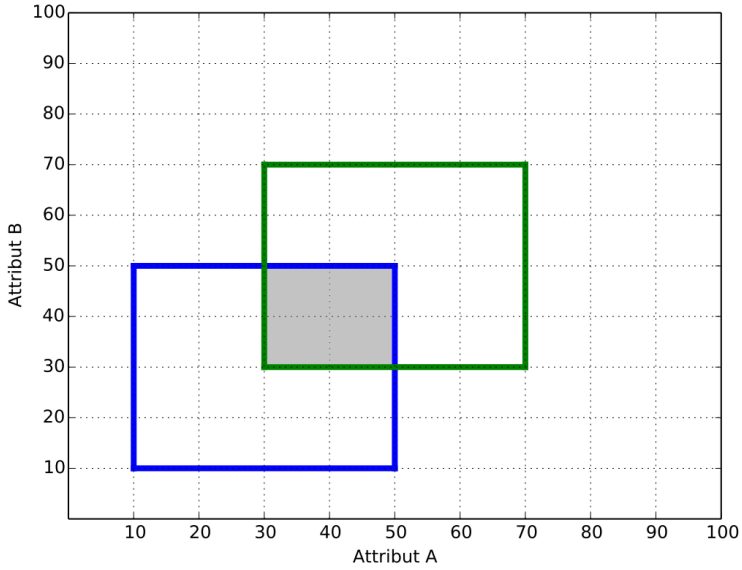

Abbildung 4.5: Zwei sich überschneidende Flows

Offensichtlich überschneiden sich *B1* und *B2*. Die gemeinsame Teilfläche, also *Overlap O(B1,B2)* ist hier gegeben durch *O(B1,B2) : A = 30-50 && B = 30-50*. Events, die also in der Fläche dieses *Overlaps* liegen, müssen sowohl über den Ausgangsport von *B1*, als auch über den von *B2* geleitet

werden. Einfach nur *B1* und *B2* auf dem Switch zu installieren ist also nicht ausreichend, denn hierdurch können genau solche Events einen Treffer mit beiden dieser Tabelleneinträge erzielen. Wichtig ist es allerdings, für jedes Event einen eindeutigen Treffer zu erzielen. Folglich muss auch der *Overlap* von *B1* und *B2* als ein dritter Flow installiert werden. Dies löst genanntes Problem zunächst aber nicht. Genaugenommen existieren nun sogar 3 Tabelleneinträge, mit denen ein Event dieser Teilfläche einen Treffer erzielen könnte. Es muss also eindeutig spezifiziert werden, welcher dieser Tabelleneinträge für solch ein Event zu wählen ist. Hierfür werden den Tabelleneinträgen Prioritäten zugeordnet. Die Priorität des *Overlaps* muss also immer höher sein, als die von *B1* oder *B2*. Dadurch ist die Eindeutigkeit der Tabelle wieder garantiert. Enthält ein Flow A einen weiteren Flow B, so muss analog zu eben genanntem Beispiel auch die Priorität von Flow B höher gesetzt werden, als die von Flow A. Grundsätzlich kann gesagt werden, dass die Priorität eines Flows A höher ist, je mehr weitere Flows diesen Flow A enthalten.

4.3.1 Hinzufügen eines neuen Flows

Wird nun ein neuer *Basisflow* B mit der Intention, diesen hinzuzufügen, an einen *Container* übergeben, so kann der hierdurch ausgelöste Vorgang grob in drei Schritte gegliedert werden:

- Die Suche nach einem gleichwertigen, bereits installierten Flow
- Die Aktualisierung aller Flows, die B enthalten, oder von B enthalten werden
- Die Berechnung aller durch das Hinzufügen von B entstehenden *Overlaps*

Diese Schritte werden nun im Folgenden detailliert erklärt. Beim Hinzufügen eines neuen *Basisflows* besteht natürlich die Möglichkeit, dass durch diese Operation keine Änderungen in Betracht der Flächen der bereits installierten Flows durchgeführt werden müssen. Insbesondere müssen in diesem Fall auch keine weiteren Flows mit neuen Flächen hinzugefügt werden. Dieses Szenario tritt dann ein, wenn bereits ein zu dem neuen *Basisflow* gleichwertiger Flow installiert ist. Es ist also garantiert, dass alle *Overlaps*, die durch die Fläche des neuen *Basisflows* verursacht werden, bereits durch den schon installierten Flow erzeugt wurden. Existiert also bereits ein solcher gleichwertiger Flow, so reicht es folglich aus, diesem Flow die Ausgangsports des neuen *Basisflows* hinzuzufügen. Außerdem müssen in diesem Fall eben diese Ports auch allen Flows hinzugefügt werden, die von dem gleichwertigen Flow enthalten werden. Insbesondere muss der neue *Basisflow* nicht zusätzlich zur Tabelle hinzugefügt werden. Es können also mehrere *Basisflows* existieren, die in der Tabelle eines Switches von demselben Flow repräsentiert werden, welcher die Kombination der Ausgangsports aller dieser *Basisflows* enthält. Für das spätere Löschen eines *Basisflows*, was auch im folgenden Unterkapitel detailliert behandelt wird, ist es wichtig zu wissen, dass ein Flow also die Information trägt, welche *Basisflows* ihm zugeordnet sind, also welche *Basisflows* gleichwertig mit diesem sind. Ein solcher, bereits existierender, gleichwertiger Flow kann natürlich auch ein *Overlap* zweier weiterer *Basisflows* sein. Auch einem solchen *Overlap* können gleichwertige *Basisflows* zugeordnet werden. Alle eingehenden *Basisflows* werden in einer eigenen Liste gespeichert. Existiert aber nicht bereits ein gleichwertiger Flow, so besteht die Möglichkeit, dass durch den neuen *Basisflow* auch neue *Overlaps* entstehen. In diesem Fall ist es nötig, zuerst einen solchen gleichwertigen Flow zu erstellen, der daraufhin der Tabelle hinzugefügt wird. Dieser neue Flow wird dem *Basisflow* nun zugeordnet. Ein *Basisflow* wird also nie selbst der Tabelle hinzugefügt, er wird lediglich am Controller hinterlegt und verweist auf einen installierten Flow, der ihn repräsentiert. Dies dient

der einfacheren Verwaltung der Tabellen beim Löschen von *Basisflows*. Abbildung 4.6 zeigt ein Szenario, in dem drei *Basisflows B1*, *B2* und *B3* einem *Container* hinzugefügt und jeweils einem gleichwertigen Flow zugeordnet werden. *B1* und *B2* werden hier dem selben Flow zugeordnet. Wenn im Folgenden bei der Aktualisierung der Tabellen von einem Basisflow gesprochen wird, ist hier somit ebenfalls der ihm zugeordnete, installierte Flow gemeint.

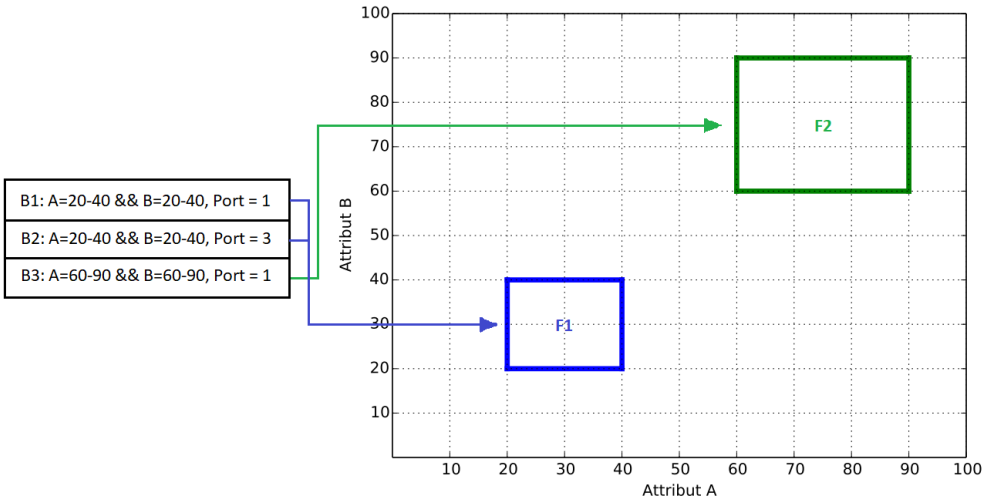

Abbildung 4.6: Zuordnung der Basisflows

Nun müssen alle Flows identifiziert werden, die in Relation zu dem neuen *Basisflow*, beziehungsweise seinem zugeordneten Flow, stehen. Dies sind also alle Flows, die sich mit diesem überschneiden, ihn enthalten, oder von ihm enthalten werden. Die Flows, die im neuen *Basisflow*, beziehungsweise dem diesem zugeordneten Flow, enthalten sind müssen auch hier um den Ausgangsport des *Basisflows* erweitert werden. Dem neuen *Basisflow* müssen alle Ausgangsports der Flows, die ihn enthalten, hinzugefügt werden. Nun werden alle *Overlaps* berechnet, die durch den Schnitt des *Basisflows* und jeweils allen Flows, die den neuen *Basisflow* schneiden, entstehen. Für jeden entstandenen *Overlap* muss auch hier geprüft werden, ob bereits ein Flow existiert, der gleichwertig ist. Es ist hier allerdings nicht nötig, jeden bereits installierten Flow auf Gleichwertigkeit zu überprüfen. Es ist ausreichend zu überprüfen, ob einer der beiden Flows, welche den *Overlap* verursachen, einen solchen gleichwertigen Flow enthält. Um eine Übersicht behalten zu können, welche *Overlaps* durch den Schnitt welcher Flows entstanden sind, enthält jeder *Overlap* Verweise auf die verursachenden Flows. Da logischerweise die gleiche Fläche durch den Schnitt unterschiedlicher Flow-Paare entstehen kann, ist es auch möglich, dass ein Flow also Verweise auf mehrere Flow-Paare besitzt. Existiert also wie zuvor erwähnt ein Flow, der zu einem neuen *Overlap* gleichwertig ist, so wird diesem lediglich ein Verweis auf das Flow-Paar, welches den neuen *Overlap* verursachte, zugeordnet. Der neue *Overlap* muss hierbei also nicht separat hinzugefügt werden. Der eben erwähnte gleichwertige Flow kann hier natürlich auch ein *Basisflow* sein. Folglich kann ein Flow also entweder ein *Basisflow* sein, dem gegebenenfalls auch mehrere weitere *Basisflows* zugeordnet sind, er kann ein *Overlap* sein, der gegebenenfalls durch mehrere verschiedene Flow-Paare verursacht wurde, aber er kann auch sowohl

Basisflow als auch *Overlap* sein. Existiert ein solcher gleichwertiger Flow nicht, wird hier der neue *Overlap* der Tabelle hinzugefügt. Auf die Ausgangsports der in diesem *Overlap* enthaltenen Flows wirkt sich dies allerdings nicht aus, da diese logischerweise auch in den beiden Flows enthalten sind, welche den *Overlap* verursachten. Die Ausgangsports wurden also bereits zuvor aktualisiert. Nach dem Hinzufügen der neuen *Overlaps* muss ebenfalls überprüft werden, ob sich diese *Overlaps* eventuell untereinander überschneiden. Hierbei können allerdings keine *Overlaps* entstehen, zu denen es nicht bereits einen gleichwertigen Flow gibt. Man gehe von 3 *Basisflows B1,B2* und *B3* aus, die sich jeweils gegenseitig überschneiden. *B1* und *B2* seien dem *Container* bereits hinzugefügt. Kommt nun der dritte *Basisflow B3* hinzu, so entstehen folglich zwei *Overlaps O(B1,B3)* und *O(B2,B3)* mit jeweils den anderen beiden *Basisflows B1* und *B2*, sowie ein *Overlap O(B3,O(B1,B2))* mit dem *Overlap* der beiden anderen *Basisflows B1* und *B2*. Erstere beiden *Overlaps* überschneiden sich logischerweise. Die hierdurch entstehende Fläche ist folglich eine Teilfläche aller drei *Basisflows*. Somit ist diese auch eine Teilfläche des *Overlaps O(B1,B2)* der beiden *Basisflows B1* und *B2*. Diese Teilfläche ist aber zwangsweise gleichwertig mit *Overlap O(B3,O(B1,B2)*, welcher bereits zuvor installiert wurde. Abbildung 4.7 veranschaulicht dieses Szenario. Die grau unterlegte Fläche stellt hier sowohl *O(B3,O(B1,B2))* als auch *O(O(B1,B3),O(B2,B3))* dar.

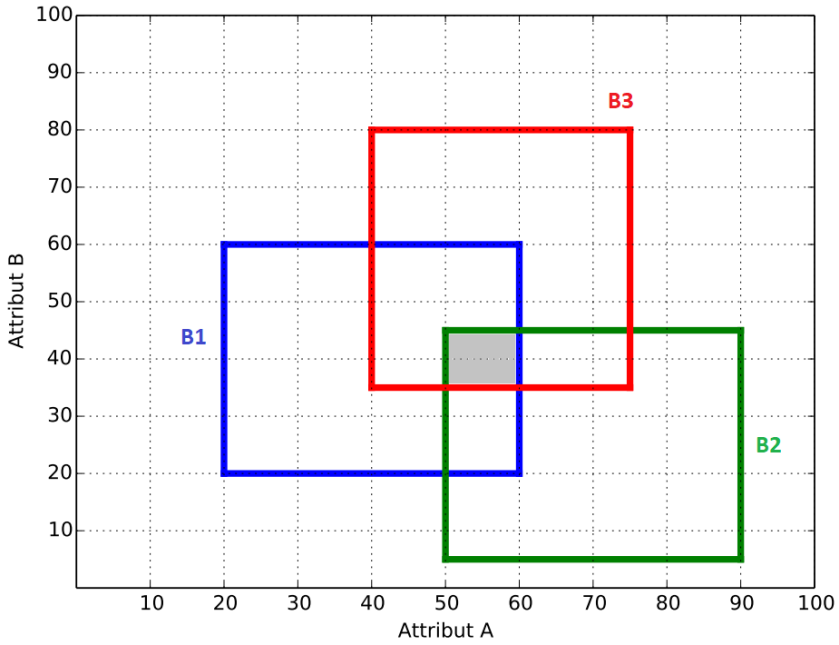

Abbildung 4.7: Entstehen neuer Overlaps

Allerdings können durch das Überschneiden zweier dieser neuen *Overlaps* zu einem bereits existierenden Flow weitere Flow-Paare hinzugefügt werden müssen, welche diesen bestimmten *Overlap* verursachten.
Das identifizieren aller installierten Flows, welche zu einem neuen *Basisflow* in Relation stehen, kann bei einer hohen Anzahl an Tabelleneinträgen sehr laufzeitintensiv sein, da hier jedes Mal alle dieser Flows überprüft werden müssen. Um die Laufzeit hier zu senken, bietet es sich an, dass jeder Flow Verweise auf alle anderen Flows, die ihn enthalten, trägt, sowie Verweise auf alle Flows, die er selbst enthält. Hierdurch entsteht eine hierarchische Anordnung aller Flows. Es ist somit ausreichend, zunächst zu prüfen, mit welchen anderen Flows, denen mindesten ein *Basisflow*

zugeordnet ist, ein neuer *Basisflow* in Relation steht. Relevant sind also offensichtlich all diese betroffenen Flows, sowie alle weiteren Flows, die sie enthalten. Alle Flows, die beispielsweise von einem Flow mit zugeordneten *Basisflows* enthalten sind, der in keiner Relation zu einem neuen *Basisflow* steht, müssen hier gar nicht betrachtet werden. Sie können logischerweise ebenfalls nicht in Relation mit dem neuen *Basisflow* stehen. Abbildung 4.8 stellt hier eine solche hierarchische Baumstruktur dar, wobei jeder Flow mit mindestens einem zugeordneten *Basisflow* jeweils den Wurzelknoten seines Baumes bildet. Diese Struktur muss natürlich beim Hinzufügen eines neuen *Basisflows* ebenfalls entsprechend aktualisiert werden.

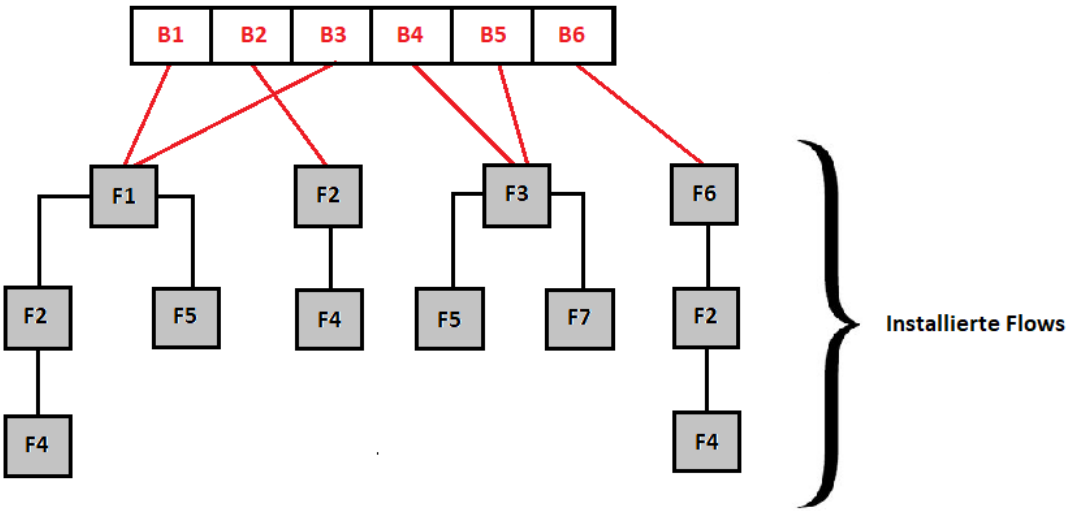

Abbildung 4.8: Hierarchische Anordnung der Flows

Nachteil hierbei ist eine eher schwere Datenstruktur bestehend aus sehr vielen Zeigern oder Verweisen auf weitere Flows. Hierfür kann unter Umständen sehr viel Haldenspeicher in Anspruch genommen werden. Ist dies unerwünscht, kann logischerweise auf die eben beschriebene hierarchische Anordnung der Flows verzichtet werden. Stattdessen muss in diesem Fall also immer für jeden installierten Flow überprüft werden, ob dieser in Relation zu dem neuen *Basisflow* steht. Zuletzt müssen die Prioritäten aller betroffenen Flows aktualisiert werden. Wird bereits zu Beginn, am Anfang dieses Unterkapitels als Schritt 1 bezeichnet, ein gleichwertiger Flow gefunden, so werden keine weiteren Flows hinzugefügt. Die Flächen im Eventraum ändern sich folglich nicht, die Priorität von keinem Flow muss hier also aktualisiert werden. Werden aber ein neuer *Basisflow* und gegebenenfalls entstehende *Overlaps* hinzugefügt, muss hier zunächst die Priorität der neu hinzukommenden Flows gesetzt werden. Daraufhin ist es nötig, die Priorität aller Flows, die von einem neu hinzukommenden Flow enthalten werden, entsprechend zu erhöhen. Generell könnte hier als Kriterium für die Priorität eines jeden Flows die Anzahl an Flows, welche diesen enthalten, gewählt werden. Wird ein Flow von 2 weiteren Flows enthalten, würde ihm so die Priorität 3 zugeordnet. Hierdurch ist es aber möglich, dass unnötige Lücken entstehen. Beispielsweise haben hier zwei sich überschneidende Flows die Priorität 1, deren *Overlap* folglich die Priorität 3. Allerdings wäre es deutlich intuitiver, diesem *Overlap* hier die Priorität 2 zuzuordnen. Um dies zu erreichen kann die Priorität eines jeden Flows mit zugeordneten *Basisflows* anhand der ihn enthaltenden Flows, denen ebenfalls mindestens einen *Basisflow* zugeordnet ist, gewählt werden. Analog zu oben hat ein Basisflow, beziehungsweise ein Flow dem mindestens ein *Basisflow* zugeordnet ist, der von

zwei weiteren Flows mit mindestens einem zugeordneten *Basisflow* enthalten wird, die Priorität 3. Die Priorität eines *Overlaps* wird hier hingegen von den Prioritäten der Flows beeinflusst, die diesen *Overlap* verursachten. Aus allen diesen verursachenden Flow-Paaren wird der Flow mit der maximalen Priorität gewählt. Hat dieser die Priorität 4, so erhält der *Overlap* die nächsthöhere Priorität 5. Dies gilt aber nur, wenn dem *Overlap* nicht gleichzeitig auch ein oder mehrere *Basisflows* zugeordnet sind.

4.3.2 Löschen eines installierten Flows

Auch muss es einem Abonnenten oder Publisher natürlich möglich sein, dynamisch bereits existierende Abonnements oder Advertisments wieder zurückzuziehen. Anhand des Spannbaumes wird hier nun berechnet, welche Teilfläche eines zurückzunehmenden Abonnements auf welchen Switches des Netzwerkes installiert ist, sowie welche Ausgangsports hierfür jeweils verwendet werden. Ein *Basisflow* mit dieser Fläche und dem entsprechenden Port wird nun an die *Container* aller betroffenen Switches übergeben. Zunächst kann überprüft werden, ob ein solcher *Basisflow* überhaupt existiert. In der Liste aller *Basisflows* eines *Containers* wird also nach einem gleichwertigen Flow gesucht, dessen Ausgangsport zusätzlich identisch mit dem des zu löschenden *Basisflows* ist. Geht man davon aus, dass das restliche System korrekt funktioniert, sowie Publisher und Abonnenten keine unsinnigen Abonnementkündigungen einsenden, so ist dieser Vorgang nicht zwingend notwendig. Um den *Basisflow* nun zu löschen, muss der installierte Flow F identifiziert werden, der diesem *Basisflow* zugeordnet ist. Ist dieser gefunden, so wird der *Basisflow* aus der Liste der dem Flow F zugeordneten *Basisflows* sowie der Liste aller *Basisflows* des *Containers* gelöscht. Auch der Ausgangsport des zu löschenden *Basisflows* muss hier bei dem installierten Flow F entfernt werden, wenn keiner der Flows, welche Flow F enthalten, und keiner der übrigen dem Flow F zugeordneten *Basisflows* diesen Port ebenfalls verwendet. Gleiche Aktualisierung der Ports wird auch auf alle Flows angewendet, die in Flow F enthalten sind. Nachdem dies geschehen ist, muss der Flow F genau dann ebenfalls komplett entfernt werden, wenn ihm nun kein weiterer *Basisflow* mehr zugeordnet ist und er zusätzlich nicht durch den Schnitt zweier weiterer Flows entstanden ist. Geschieht dies, muss bei allen durch diesen Flow F verursachten *Overlaps* jedes verursachende Flow-Paar entfernt werden, dass Flow F enthält. Auch hier muss der *Overlap* dann komplett gelöscht werden, wenn ihm kein *Basisflow* zugeordnet ist, und es keine weiteren Flow-Paare gibt, die diesen *Overlap* verursachen. Rekursiv wird dann gleiches Prinzip auf alle durch diesen *Overlap* verursachten *Overlaps* angewendet. Außerdem muss folglich die Priorität aller Flows aktualisiert werden, die zuvor in Flow F enthalten waren. Angewendet wird hier die beim Hinzufügen eines Flows beschriebene Vorgehensweise für die Berechnung der Priorität. Muss Flow F nicht komplett gelöscht werden, so muss folglich nur dessen Priorität angepasst werden. Auch das ist nur notwendig, wenn ihm nun keine *Basisflows* mehr zugeordnet sind, allerdings weiterhin Flow-Paare existieren, die diesen Flow F durch ihren Schnitt verursachen. Die Priorität wird hier wie gewöhnlich bei *Overlaps* durch die maximale Priorität aller Flows, welche diesen Flow F durch Überschneiden mit einem weiteren Flow verursachen, bestimmt. Da Flow F hier weiterhin installiert bleibt, hat das Löschen des *Basisflows* hier also keinen Einfluss auf die Priorität der in F enthaltenen Flows.

4.3.3 Vermeidung redundanter Basisflows

Die Anzahl der auf einem Switch installierbaren Flows ist begrenzt. Wünschenswert ist es also, redundante Flows möglichst komplett zu vermeiden. Ein *Basisflow* A wird hier als redundant bezeichnet, wenn ein weiterer *Basisflow* vorliegt, der gleichwertig zu *Basisflow* A ist oder *Basisflow* A enthält und außerdem auch alle Ausgangsports von *Basisflow* A besitzt. Würde *Basisflow* A nun hinzugefügt werden, so könnte dies zwar eine Änderung der Tabelle zufolge haben, das Verhalten des Switches bliebe aber weiterhin identisch. *Basisflow* A wird also, statt hinzugefügt zu werden, lediglich in eine Warteschlange eingereiht. Auch besteht natürlich beim Hinzufügen eines *Basisflows* die Möglichkeit, dass weitere bereits vorliegende *Basisflows* hierdurch redundant werden. Diese *Basisflows* müssen zunächst gelöscht, und daraufhin der Warteschleife hinzugefügt werden.

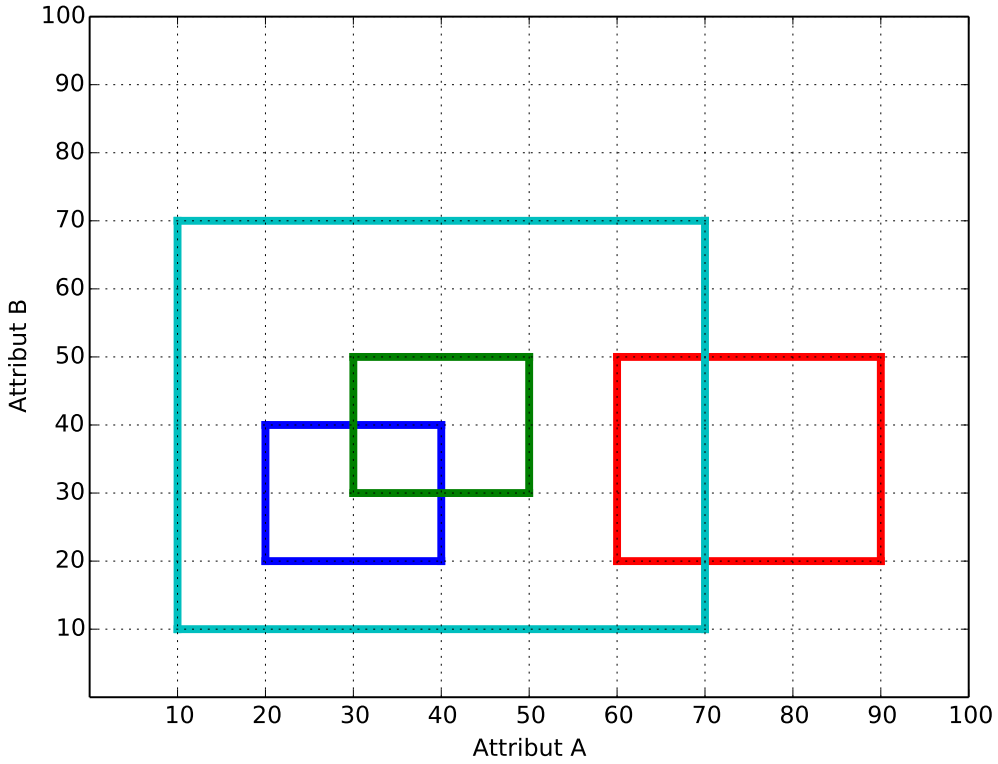

Abbildung 4.9: Neuer Basisflow führt zu Redundanz installierter Flows

Abbildung 4.9 verdeutlicht dies noch einmal. Hier seien bereits drei *Basisflows* auf einem Switch installiert. Diese Flows werden hier von der rot, blau und grün umrandeten Fläche dargestellt. Alle drei *Basisflow* haben hier den selben Ausgangsport 1. Wird nun ein neuer Basisflow, repräsentiert durch die zyanfarbene Fläche, mit ebenfalls Ausgangsport 1 hinzugefügt, so werden der blaue und grüne *Basisflow* redundant. Selbst wenn diese *Basisflows* wegfallen, werden weiterhin alle Events in deren Bereichen über Ausgangsport 1 geleitet. Das Entfernen der beiden *Basisflows* wirkt sich hier also nicht auf das Verhalten des Switches beim Weiterleiten von Events aus. Der rote *Basisflow* ist hier allerdings nicht redundant, da er weiterhin eine Teilfläche besitzt, die von keinem anderen *Basisflow* mit gleichen Ausgangsports abgedeckt wird. Soll ein *Basisflow* gelöscht werden,

muss hierbei die Warteschlange der zuvor redundanten *Basisflows* überprüft werden. Befindet sich der zu löschende *Basisflow* in dieser Warteschlange, so wird er einfach hieraus entfernt und der Löschvorgang ist komplett abgeschlossen. Ist dem nicht der Fall, so muss geprüft werden, ob durch das Löschen dieses *Basisflows* eventuell weitere *Basisflows* aus der Warteschlange nicht mehr redundant sind. Solche *Basisflows* müssen daraufhin hinzugefügt werden, und somit auch aus der Warteschlange entfernt werden.

5 Auswertung

Dieses Kapitel beschäftigt sich nun mit der Evaluation des in Kapitel 4 realisierten Systems. Im Fokus stehen hier sowohl korrekte Funktionalität des Systems als auch dessen Performanz. Hierfür wird die Rate der Falsch-Positive und Falsch-Negative dokumentiert. Außerdem soll überprüft werden, wie sich die Anzahl an Abonnements auf die schlussendliche Größe der Flowtabellen eines Switches auswirkt. Diese Aspekte werden im Unterkapitel Funktionalität untersucht. Den Schlusspunkt setzt eine Ende-zu-Ende-Latenzmessung. Hier ist vor allem interessant, wie sich Faktoren wie Tabellengröße auf diese Latenz auswirken. Auch soll überprüft werden, ob der Multicast eines Events eine höhere Latenz als die gewöhnliche Weiterleitung über nur einen Ausgangsport mit sich bringt. Diese Auswertung der Latenz wird im Unterkapitel Performanz untersucht. Da sich die P4Runtime API[22] zurzeit noch in der Entwicklungsphase befindet, ist es leider nicht möglich, diese für die Auswertung in das Publish/Subscribe System zu integrieren. Stattdessen werden dem Controller hierfür sequentiell alle Abonnements übergeben. Dieser fügt diese dann dem Container des jeweiligen Switches hinzu und generiert somit alle benötigten Tabelleneinträge. Danach generiert der Controller für jeden Switch eine Ausgabedatei, welche alle Kommandos enthält, um die Tabellen der Switches manuell über das Kommandozeileninterface der verwendeten Switches mit genau den eben generierten Tabelleneinträgen zu füllen. Auf die Funktionalität und die Latenzmessungen hat dies keine Auswirkung.

5.1 Funktionalität

5.1.1 Testsetup

Für sämtliche Tests und Auswertungen zur Funktionalität des im vorigen Kapitel entworfenen Systems wird hier der bmv2 Softwareswitch[23] sowie Mininet[24] verwendet. Bmv2 stellt hier diverse verschiedene Switches bereit. Verwendet wurde hier der *simple_switch*. Die durch Mininet simulierte Fat-Tree Netzwerktopologie wird in Abbildung 5.1 dargestellt.

Host 1 übernimmt hierbei die Rolle eines Publishers, Host 2 – 7 jeweils die Rolle eines Abonnenten. Für die Angaben der Tabellengröße in Kontrast zu den eingereichten Abonnements wird hier im Besonderen die Tabelle von Switch 1 in Betracht gezogen. Dies hat den einfachen Grund, dass jedes Abonnement aller Abonnenten für diesen Switch relevant ist. Somit stellt die Anzahl der Tabelleneinträge dieses Switches die maximale Anzahl der Tabelleneinträge eines Switches in diesem Netzwerk dar. Die Generation der Abonnements für einen Test basiert auf zwei unterschiedlichen Verteilungen. Bei einer uniformen Verteilung der Abonnements sind diese für jeden Host uniform über den kompletten Eventraum verteilt. Bei einer Zipf-Verteilung wird jedem Abonnenten ein sogenannter Hotspot zugeteilt. Alle für diesen Abonnenten generierten Abonnements konzentrieren

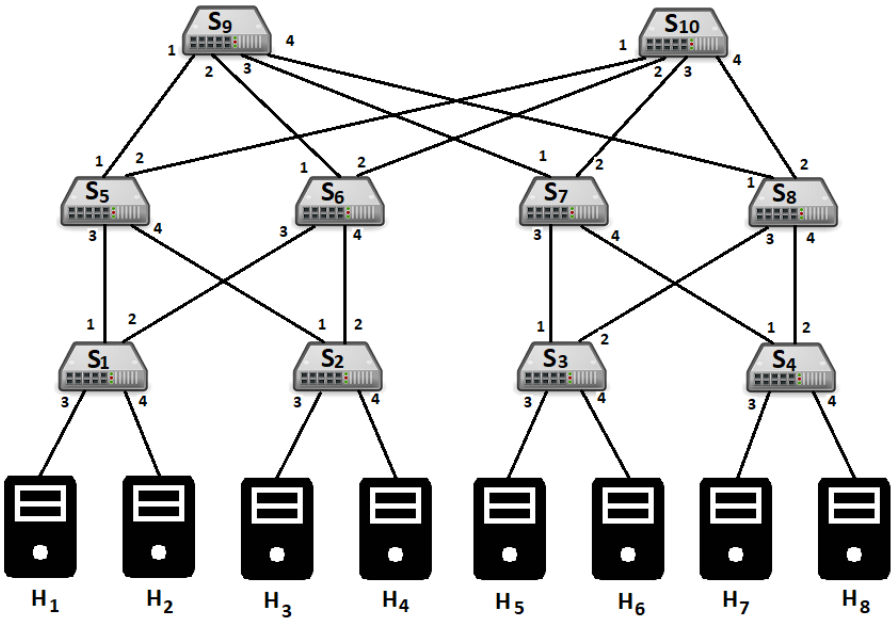

Abbildung 5.1: Fat-Tree Testtopologie

sich nun um seinen Hotspot. Hier besteht auch die Möglichkeit, dass die Hotspots mehrerer Abonnenten im Eventraum sehr nah beieinander liegen. Bei der Analyse der Tabellengröße sollte also auch die gewählte Verteilung der Abonnements in Betracht gezogen werden.

5.1.2 Tabellengröße

Abbildung 5.2 zeigt hier das Verhalten der Größe der Tabelle bei steigender Anzahl an Abonnements. Dargestellt wird die Tabellengröße hier für 20,40,60,80 und 100 Abonnements. Für jede Abonnementanzahl wurden hier verschiedene Sammlungen an Abonnements generiert, diese werden unter den 7 Abonnenten aufgeteilt. Der Wert der Tabellengröße ihr hierbei der Durchschnittswert aller Tabellengrößen dieser unterschiedlichen Sammlungen. Dies gilt sowohl für uniforme, als auch für die Zipf-Verteilung.

Es ist zu erkennen, dass unter Zipf-Verteilung generierte Abonnements deutlich weniger Tabelleneinträge verursachen. Dies ist darauf zurückzuführen, dass sich die Abonnements verschiedener Abonnenten durch die Konzentration um den jeweiligen Hotspot hier seltener überschneiden. Hierdurch werden folglich weniger Overlaps erzeugt. Bei uniformer Verteilung besteht die Möglichkeit, dass sich im schlechtesten Fall alle Abonnements paarweise miteinander überschneiden. Hierdurch werden eine immense Anzahl an Overlap und somit auch Tabelleneinträgen erzeugt. Generell ist

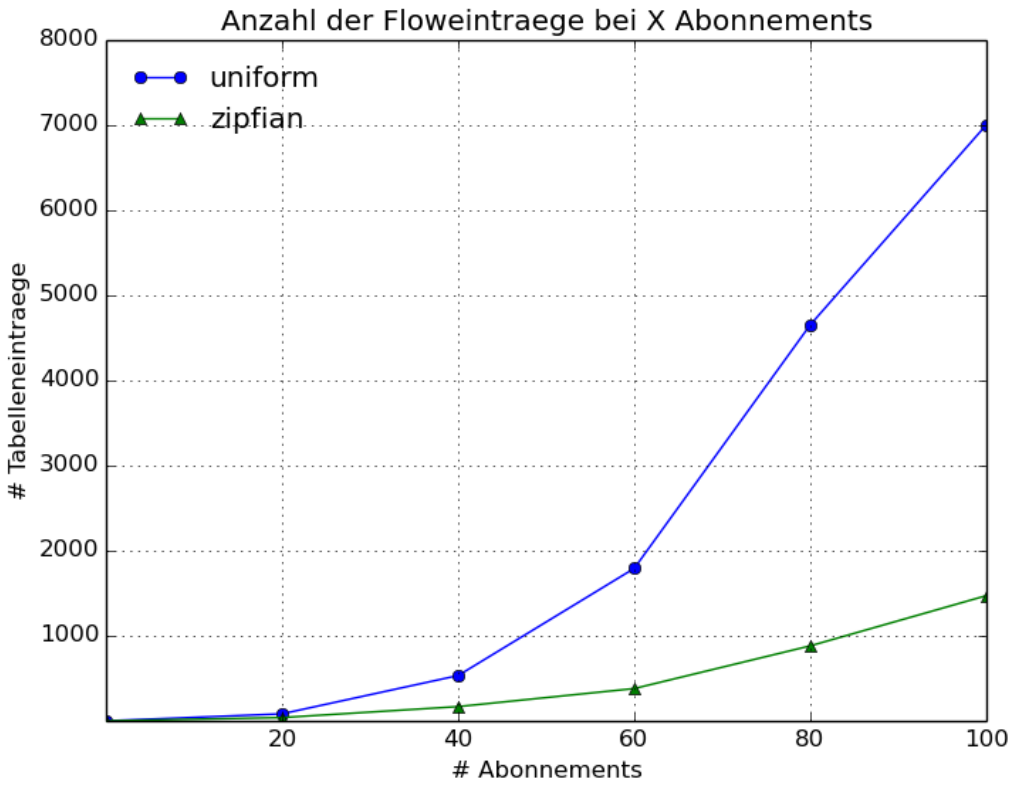

Abbildung 5.2: Anzahl der Einträge einer Flowtabelle

hier für beide Verteilungen ein starker Anstieg der Tabellengröße zu verzeichnen. Dieser ist für die uniforme Verteilung aus eben beschriebenem Grund deutlich stärker ausgeprägt. Hier wächst die Anzahl der Tabelleneinträge mit steigender Anzahl an Abonnements fast exponentiell an. Besonders gravierend ist hier beispielweise der Anstieg von 1795 Einträgen für 60 Abonnements auf 4655 Einträge bei 80 Abonnements unter uniformer Verteilung. Unter Zipf-Verteilung ist hier vergleichsweise nur ein kubischer Anstieg zu dokumentieren. 100, durch uniforme Verteilung generierten Abonnements, verursachen hier im Durchschnitt knapp 7000 Tabelleneinträge. Dies ist fast das sechsfache der durchschnittlichen Tabelleneinträge, die durch Zipf-Verteilung bei gleicher Abonnementanzahl verursacht werden. Zu betonen ist hier auch, dass für die Anzahl der Einträge bei einer bestimmten Anzahl an Abonnements keine feste Ober- und Untergrenze angegeben werden kann. Je nach den Verteilung der eingereichten Abonnements kann hier im Bestfall, so unwahrscheinlich dieser auch ist, schon ein einziger Eintrag ausreichen. Im schlechtesten Fall kann die Anzahl der Einträge hingegen auch deutlich über dem zuvor angegebenen Durchschnitt liegen. Faktoren hierfür sind neben der Abonnementanzahl unter anderem auch die Anzahl der verschiedenen Abonnenten. Die Größe der Tabellen wurde hier nur bis zu einer maximalen Anzahl an 100 Abonnements ausgewertet. Aufgrund der großen Anzahl an Overlaps und das damit verbundene exponentielle Wachstum der Tabelleneinträgen wird bei einer höheren Abonnementzahl der Controllermehraufwand zu hoch. Anhand eines üblichen Haushaltsrechners lassen sich deshalb keine Werte für eine größere Abonnementzahl ermitteln. Je nachdem, ob hier die in Kapitel 4 beschriebene Datenstruktur oder aber die ebenfalls skizzierte Variante mit etwas leichterer Datenstruktur verwendet wurde, stellt

hier entweder der Haldenspeicher oder die Dauer des Hinzufügens eines neuen Abonnements eine Beschränkung dar. Die Skalierbarkeit des Systems ist also sehr stark begrenzt. Mögliche Lösungen werden in der Zusammenfassung dieser Arbeit skizziert.

5.1.3 Falsch-Positive & Falsch-Negative

Hauptgrund für den Entwurf dieses Systems ist allerdings die Vermeidung von Falsch-Positiven und Falsch-Negativen. Folgende Tests sollen verifizieren, dass das in dieser Arbeit entworfene System dieses Ziel in der Tat erfüllt. Für die folgenden Tests wurde ebenfalls die zu Beginn dieses Kapitels gezeigte Topologie verwendet. Weiterhin werden Abonnements sowohl mit Zipf-, als auch mit uniformer Verteilung generiert. Auch die Anzahl an Abonnements, die für die jeweiligen Testläufe den Switches hinzugefügt werden, bleibt analog. Nachdem die Tabellen aller relevanten Switches der Topologie mit ihren jeweiligen Einträgen gefüllt sind, werden hier 10000 Events von Host 1, also dem Publisher, versendet. Diese Events wurden ebenfalls entweder mit uniformer oder Zipf-Verteilung generiert. Der Wertebereich für Attribute generierter Events entspricht hier dem gewählten maximalen Wertebereich einer jeden Dimension der generierten Abonnements. Wurden die Abonnement mit Zipf-Verteilung generiert, so werden folglich auch durch Zipf-Verteilung generierte Events gesendet. Diese sammeln sich also ebenfalls um einen Hotspot. Hier besteht folglich eine sehr große Wahrscheinlichkeit, dass manche Abonnenten nur sehr wenige bis keine Events erhalten werden, da ihr Hotspot und der Hotspot der Events im Eventraum nicht nahe beieinander liegen. Vor Senden der 10000 Events werden diese zuerst mit den jeweiligen Abonnements eines jeden Hosts abgeglichen und somit ermittelt, welcher Host bei korrekter Funktion des Systems welche Events erhalten sollte. Nun werden genau diese 10000 Events von Host 1 gesendet. Jeder der Abonnenten zeichnet alle erhaltenen Events auf. Diese erhaltenen Events eines jeden Hosts werden schlussendlich mit den zu erhaltenden Events dieses Hosts abgeglichen. Abbildungen 5.3 bis 5.6 zeigen hier, dass sowohl für Zipf-, als auch für uniforme Verteilung der Events und Abonnements für keine Abonnementzahl Falsch-Positive oder Falsch-Negative zu verzeichnen sind.

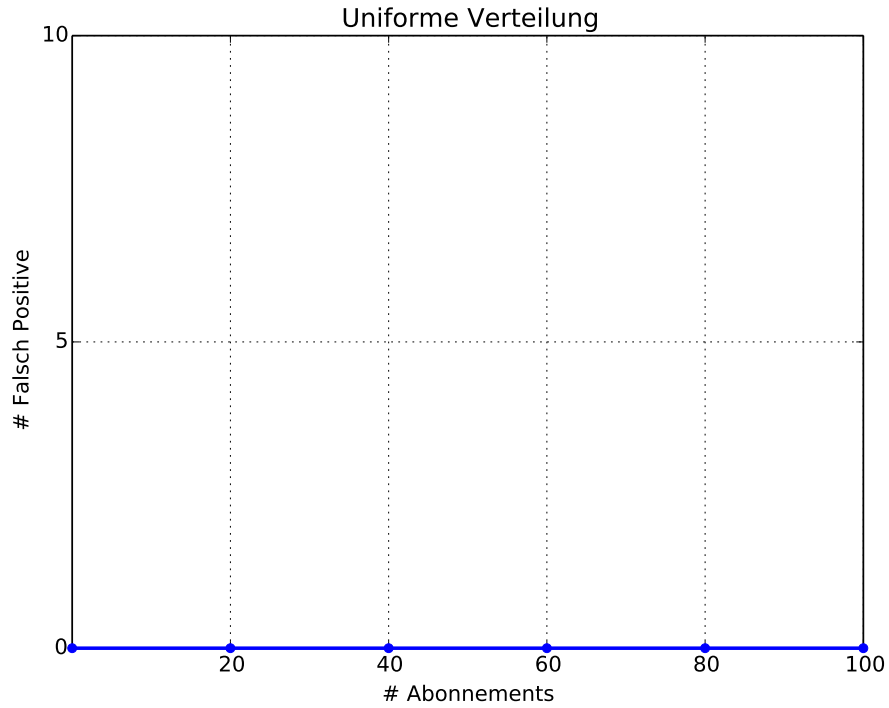

Abbildung 5.3: Falsch-Positive bei Uniformer Verteilung

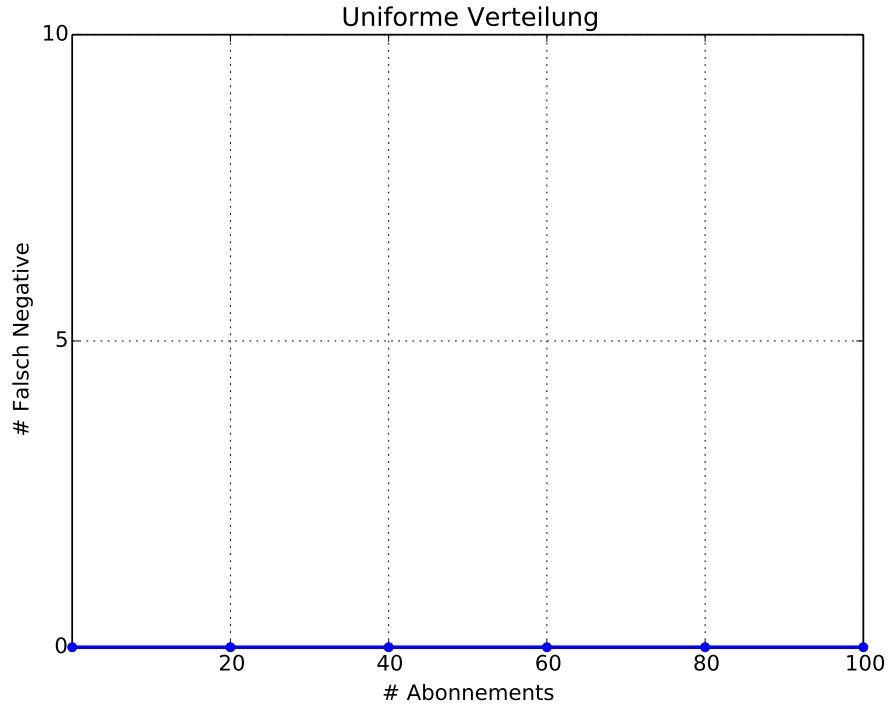

Abbildung 5.4: Falsch-Negative bei Uniformer Verteilung

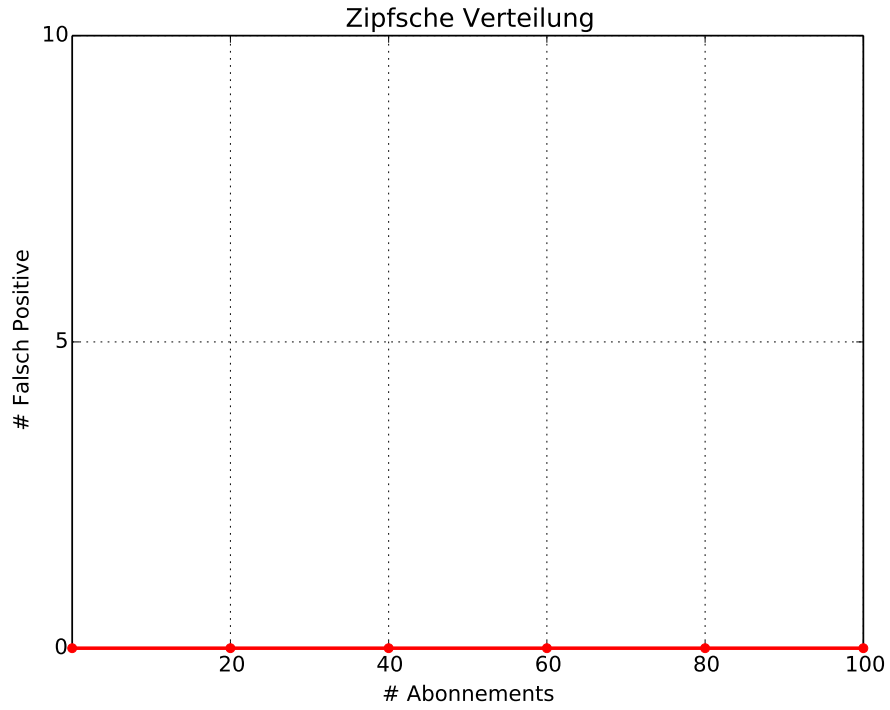

Abbildung 5.5: Falsch-Positive bei Zipf-Verteilung

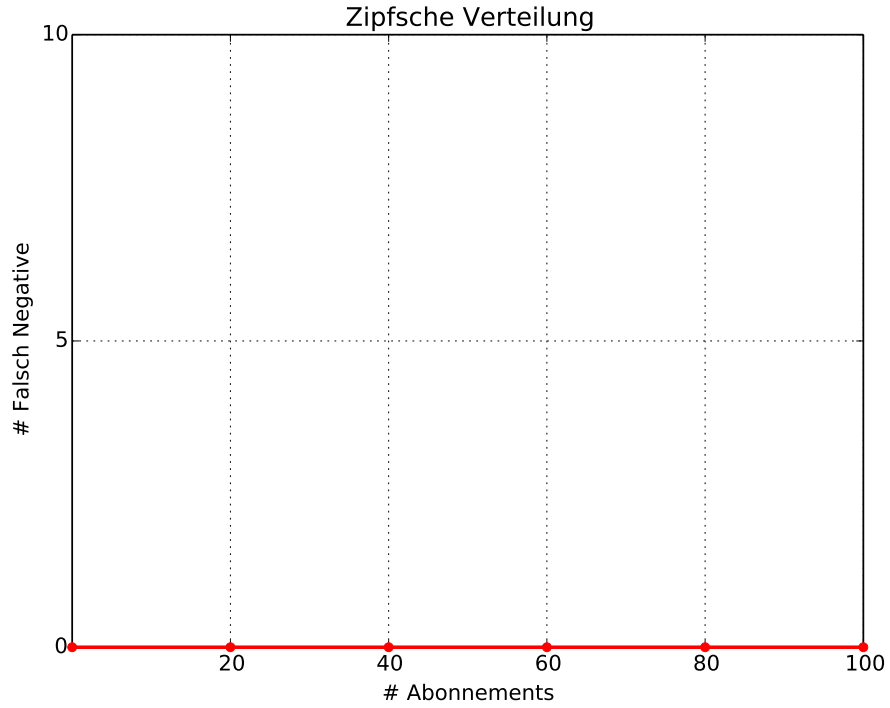

Abbildung 5.6: Falsch-Negative bei Zipf-Verteilung

5.2 Performanz

5.2.1 Testsetup

Für die Auswertung der Performanz des Systems wird hier eine programmierbare Netronome Netzwerkkarte des Modells 'Agilio CX 2x10GbE' verwendet. Die Weiterleitungspipeline dieser Karte wird anhand des in Kapitel 4 beschrieben P4 Programmes konfiguriert und die Flowtabellen über das von Netronome bereitgestellte Kommandozeileninterface gefüllt. Die für die Generation der Tabelleneinträge verantwortlichen Abonnements werden analog zu vorherigem Unterkapitel gebildet. Auf die Generation von Abonnements mit Zipf-Verteilung wird hier allerdings verzichtet, da für die Latenz nur die schlussendliche Anzahl an Einträgen, nicht aber die Lage der einzelnen Abonnements, eine Rolle spielt. Die Netzwerkkarte verfügt über zwei Ports p0 und p1, welche jeweils mit einer 10GbE Solarflare Netzwerkkarte mit Fähigkeit zum Setzen von Hardware-Zeitstempeln verbunden sind. Diese hierdurch gegebene Topologie ist Abbildung 5.7 zu entnehmen.

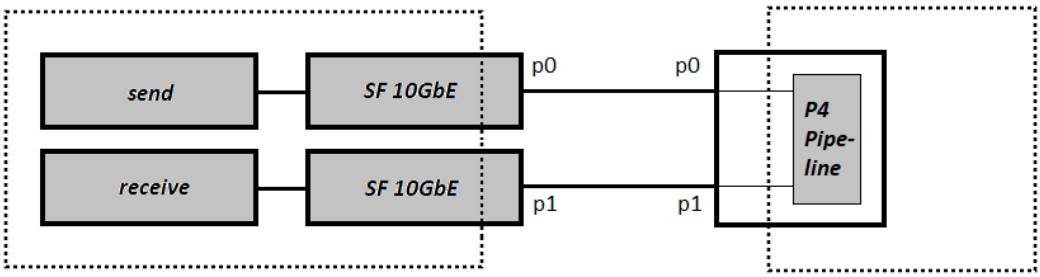

Abbildung 5.7: Topologie für Performanzevaluation

Der Host der Solarflare Netzwerkkarten sendet nun Events in Form von UDP Paketen über eine der beiden Solarflare Adapter an Port p0 der Netronome Netzwerkkarte. Ein solches Event enthält hier neben den für UDP Paketen gängigen Headern den Publish/Subscribe-Header sowie eine Sequenznummer. Das Event wird daraufhin von der Netronome Netzwerkkarte verarbeitet und dessen Attributwerte mit den Flowtabelleneinträgen abgeglichen. Entsprechend des Ergebnisses dieses Abgleiches wird das Event nun entweder fallengelassen, oder über Port p1 weitergeleitet und geht an der zweiten Solarflare Netzwerkkarte ein.

Gesendet werden hier in jedem Testlauf 10000 UDP Pakete mit einer Nutzdatengröße von 512 Byte in einem Intervall von 2 Millisekunden. Der Publish/Subscribe-Header ist hier in diesen 512 Byte einbegriffen. Werte des Publish/Subscribe-Headers werden auch hier in einem vorgegebenen Wertebereich zufällig generiert. Um die Zeit zu bestimmten, die von der Netronome Karte zur Verarbeitung und Weiterleitung eines Events benötigt wird, wird hier die Differenz der Hardwarezeitstempel der sendenden und empfangenden Solarflare Netzwerkkarte bestimmt. Die Übertragungsverzögerung kann hier vernachlässigt werden. Hauptsächlich wird im Folgenden das Verhalten der Latenz bei verschiedenen Tabellengrößen, sowie bei Multicast und Unicast betrachtet.

Zwar verfügt die Netronome Netzwerkkarte nur über zwei Ports, da es aber für diese Auswertung sehr interessant ist, wie sich die Latenz bei einem Multicast verhält, wird hier ein System mit einem Publisher und drei Abonnenten simuliert. Soll ein Event hier nur an einen dieser Abonnenten weitergeleitet werden, wird es per Unicast über p1 gesendet. Soll es an mehrere Abonnenten zugestellt werden, so wird es hier an eine Multicastgruppe gesendet, die schlichtweg ebenfalls nur p1 enthält.

Eine weitere Limitation bei Verwendung der Netronome Netzwerkkarte im Vergleich zur Verwendung des bmv2 Softwareswicthes ist die Beschränkung auf die Verwendung eines einzigen Feldes pro Tabelle für den Vergleich mit einem Wertebereich. Für das in dieser Arbeit vorgestellte Publish/Subscribe System ermöglicht dies also nur Latenzmessungen unter Verwendung eines eindimensionalen Eventraumes. Insbesondere verhindert dies leider die Ermittlung des Verhaltens der Latenz bei ansteigender Anzahl an Dimensionen.

Logischerweise ist die Anzahl der entstehenden Tabelleneinträge für ein eindimensionales System sehr viel geringer als in mehrdimensionalen Systemen. Trotz allem lassen sich durch die folgenden Messungen deutliche Muster für das Verhalten der Latenz bei variierenden Tabellengrößen erkennen.

5.2.2 Latenz

Abbildung 5.8 zeigt die gemessenen Latenzen über 10000 Events bei einer Flowtabellengröße von 200 Einträgen. Deutlich zu erkennen sind hier zwei Häufungspunkte bei 230 µs und im Bereich von 6-7 µs. Vor allem ist auch zu erkennen, dass mit steigender Anzahl gesendeter Events die Dichte am unteren Häufungspunkt ansteigt. Es war zu beobachten, dass ein Event genau dann eine durchschnittliche Latenz von 6,4 µs aufweist, wenn zuvor bereits ein weiteres Event mit gleichen Werten der für das Filtern relevanten Attribute von der Netronome Netzwerkkarte verarbeitet wurde. Die Information, welcher Tabelleneintrag für Pakete mit diesen Werten gewählt wird, scheint hier also in einem Cache zwischengespeichert zu werden. Werden die Tabellen zur Laufzeit modifiziert, muss dieser Cache natürlich wieder gelöscht werden. Auch ist zu erwähnen, dass die Position des gewählten Tabelleneintrages die Latenz hier nicht beeinflusst. Auch bei einem Treffer müssen aufgrund der zugewiesenen Prioritäten alle weiteren Tabelleneinträge ebenfalls auf einen weiteren Treffer mit höherer Priorität untersucht werden.

Abbildung 5.9 zeigt nun die durchschnittliche Latenz eines Events bei unterschiedlichen Tabellengrößen. Die blaue Linie zeigt hier für jede Tabellengröße die durchschnittliche Latenz über alle 10000 gesendeten Events. Die rote Linie hingegen berücksichtigt hier identische Pakete nicht. Zeitersparnisse durch Caching werden hier also ausgeschlossen. Die durchschnittliche Grundlatenz für nur einen Eintrag beträgt hier 6,4 µs und entspricht somit mit der durchschnittlichen Latenz eines bereits im Cache vorgemerkten Events. Die durchschnittliche Latenz bei einer Tabellengröße von 530 beträgt ohne Berücksichtigung des Caches bereits 600 µs. In beiden Fällen ist ein linearer Anstieg der Latenz bei wachsender Tabellengröße zu erkennen. Durch Fortsetzen dieses Musters lässt sich vermuten, dass die durchschnittliche Latenz ohne Caching bei 10000 Tabelleneinträgen, was eine durchaus realistische Zahl ist, bereits auf knapp 11 Millisekunden ansteigen könnte.

Zuletzt soll überprüft werden, inwiefern die durchschnittliche Latenz der durch Unicast versendeten Pakete von den durch Multicast versendeten Paketen abweicht. Hierfür werden jeweils 10000 identische Events versendet, welche an der Netronome Netzwerkkarte die gewünschte Operation

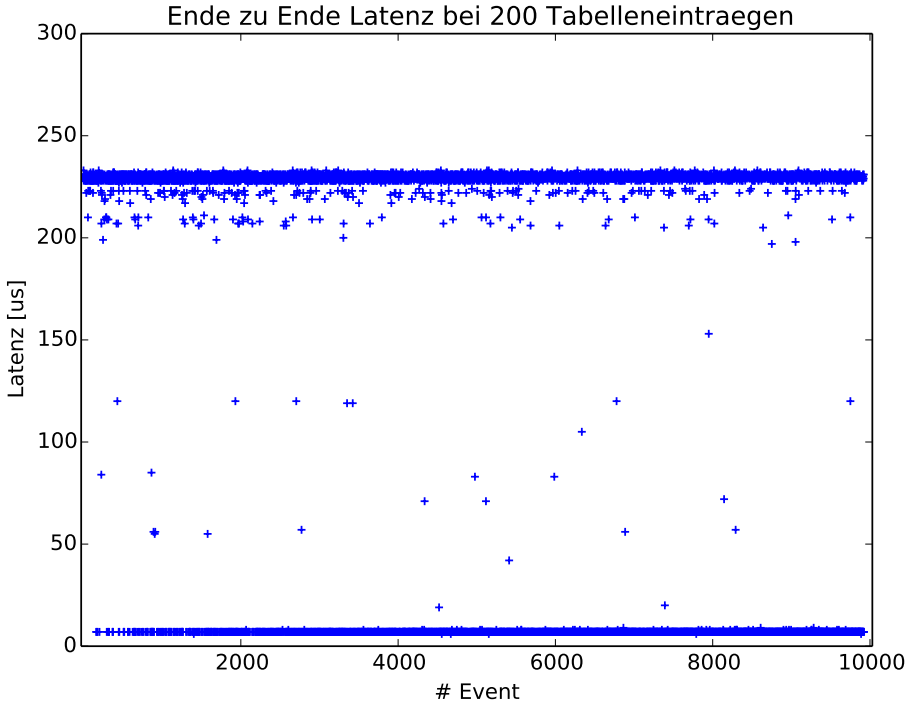

Abbildung 5.8: Durchschnittliche Latenz bei 200 Tabelleneinträgen

auslösen. Durch das Versenden identischer Events wird die durchschnittliche Latenz hier durch die Verwendung des Caches stark reduziert. Dies trifft aber auf die Resultate beider Operationen zu, weshalb eventuell variierende Werte hier trotzdem direkt auf die jeweilig ausgeführte Operation zurückzuführen sind. Erwartungsgemäß weisen über Unicast weitergeleitete Events hier eine niedrigere Latenz als ihr Gegenüber auf. Die durchschnittliche Latenz von durch Unicast versendeten Events liegt hier bei 6,4 µs. Ausführen einer Multicast Operation führt im Schnitt zu einer Latenz von 7,2 µs. Wie Abbildung 5.10 zu entnehmen ist, ist hier also ein Mehraufwand von durchschnittlich 0,8 µs zu verzeichnen.

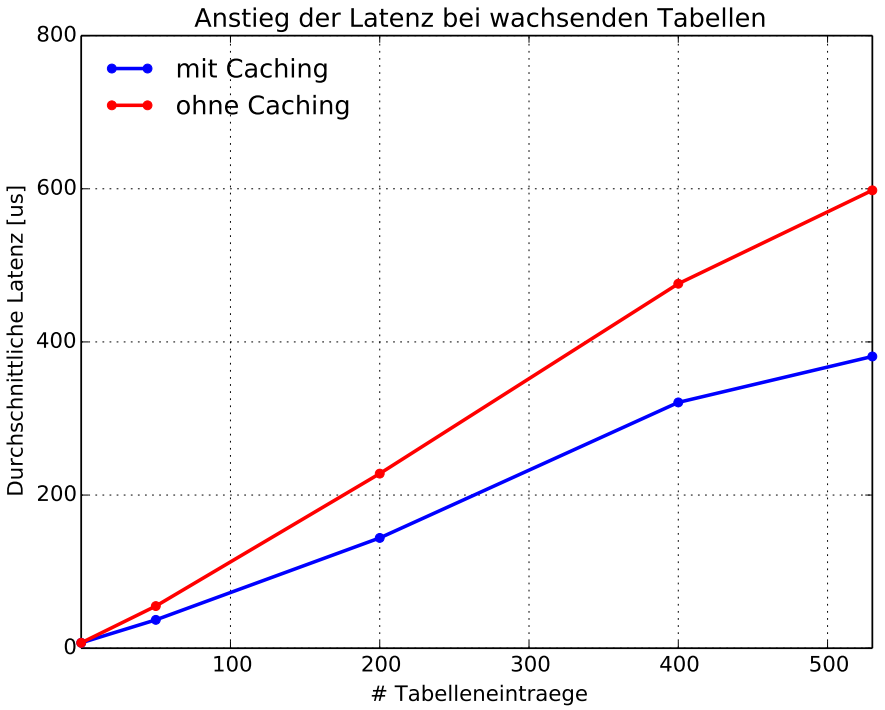

Abbildung 5.9: Durchschnittliche Latenz im Vergleich zur Tabellengröße

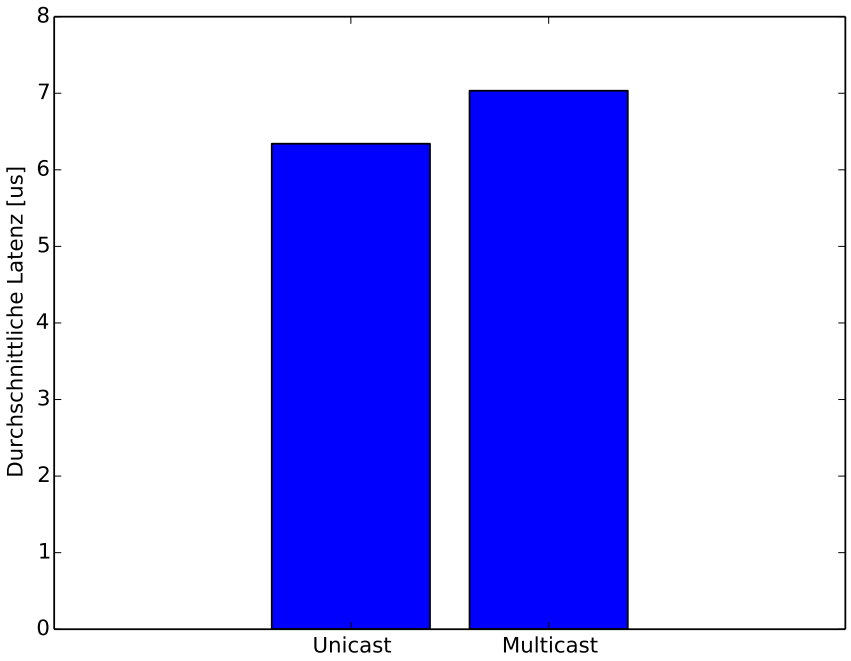

Abbildung 5.10: Latenz bei Unicast und Multicast

6 Zusammenfassung und Ausblick

Diese Arbeit zeigt, dass es unter Nutzung von P4 in der Theorie tatsächlich möglich ist, ein Publish/Subscribe System zu entwerfen, welches im Gegensatz zu verwandten Arbeiten das Auftreten von Falsch-Positiven komplett vermeidet. Praktisch hat sich hier aber gezeigt, dass der Wechsel von räumlichen Indizes zu Wertebereichen für die Repräsentation eines Flows zu einer enormen Anzahl an Tabelleneinträgen führen kann. Die Verwendung von nur 100 Abonnements ist in einem realen System natürlich kaum denkbar. Zwar kann in realen Systemen häufig von einer Zipf-Verteilung der Abonnements ausgegangen werden, und auch die Nutzung von sehr leistungsstarken Controllern wird diese Grenze sicherlich anheben können, allerdings sind hier trotz allem weitere Maßnahmen erforderlich, um den Controller zu entlasten. *PLEROMA*[7] diskutiert hier bereits einige Ansätze, wie zum Beispiel die Aufteilung des gesamten Netzwerkes in mehrere Partitionen. Hierbei ist nun jeweils ein Controller für eine Partition zuständig. Jeder Controller verfügbt nur über Informationen zur Erreichbarkeit der jeweiligen Nachbarpartitionen. Ein weiterer Ansatz ist hier eine Aufteilung des Eventraumes. Hierbei ist je ein Controller für einen Bestimmten Bereich dieses Eventraumes zuständig. Er verarbeitet also auch nur all diese Abonnements, die in besagtem Bereich liegen.
Zusätzlich ist festzustellen, dass sich die Entwicklung rund um P4 zurzeit noch in den Kinderschuhen befindet. Die schlussendliche Funktionalität hängt hier teils noch von der verwendeten Hardware ab. Essentiell für ein Publish/Subscribe System muss hier auf jeden Fall das Vergleichen mehrerer Attribute mit in Floweinträgen aufgeführten Wertebereichen sein. Die Netronome Netzwerkkarte ermöglicht dies noch nicht. Auch P4Runtime steckt zurzeit noch in der Entwicklungsphase. In der Auswertung zwar außenvor gelassen, ist die Integration von P4Runtime[22] in beschriebenes System für den Einsatz in einem realen System natürlich zwingend notwendig. Spannend ist es also, die weitere Entwicklung dieser API zu verfolgen.
In Sachen Performanz lässt sich durch entstehende Overlaps die große Anzahl and Tabelleneinträgen kaum vermeiden. Denkbar ist hier eine Erweiterung der Flow-Reduktion, sodass nicht nur redundante Basisflows entfernt werden, sondern generell jeder redundante Flow. Hier kann beispielsweise auch überprüft werden, ob die Fläche eines Flows komplett von weiteren Flows mit höherer Priorität abgedeckt wird. Höherer Fokus auf besagte Flow-Reduktion führt konsequenterweise allerdings auch zu einem erneut stark erhöhten Controllermehraufwand.
Die in vorigem Kapitel aufgeführten Latenzen weisen zwar eine Verbesserung im Vergleich zu Filtern in Software auf, können mit der Performanz von *PLEROMA* erwartungsgemäß allerdings nicht mithalten. In *PLEROMA* liegt die maximale Zeit, die ein Event über den längsten Pfad der in Kapitel 5 vorgestellten Fat-Tree Topologie von Publisher zu dem jeweiligem Abonnent benötigt bei nur knapp 100 µs [19]. Gravierend wird der Unterschied zwischen beiden Systemen vor allem mit steigender Anzahl an Tabelleneinträgen. Die Ende-zu-Ende-Latenz wird in *PLEROMA* von diesem Aspekt nicht beeinflusst und bleibt konstant. Die Kernfrage ist also, ob für ein System die Genauigkeit oder die Ende-zu-Ende-Latenz der einzelnen Events eine wichtigere Rolle spielen. Abhängig ist dies schlichtweg vom Einsatzgebiet des Publish/Subscribe System. In großen Netzwerken mit hohen Ansprüchen an die Latenz könnten die hier gemessenen Ergebnisse allerdings bereits sehr

kritisch sein. Auch interessant wird hier eine Untersuchung des Durchsatzes des in dieser Arbeit präsentierten Systems, sowie das Verhalten der Latenz bei steigender Dimensionsanzahl und somit einer höheren Anzahl an Vergleichen mit verschiedenen Wertebereichen pro Floweintrag.

Literaturverzeichnis

[1] P. Robert Pietzuch. „Hermes: A Scalable Event-Based Middleware“. In: (Juli 2004) (zitiert auf S. 11, 15).

[2] A. Carzaniga, D. S. Rosenblum, A. L. Wolf. „Design and evaluation of a wide-area event notification service“. In: *ACM Trans. Comput. Syst.* 19.3 (Aug. 2001), S. 332–383. ISSN: 0734-2071. DOI: 10.1145/380749.380767 (zitiert auf S. 11, 15).

[3] K. Benzekki, A. El Fergougui, A. El Belrhiti El Alaoui. „Software-defined networking (SDN): A survey“. In: (Feb. 2017) (zitiert auf S. 11, 16).

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner. „OpenFlow: Enabling Innovation in Campus Networks“. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (März 2008), S. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: http://doi.acm.org/10.1145/1355734.1355746 (zitiert auf S. 11, 17, 18, 27, 32, 36).

[5] *OpenFlow management and configuration protocol*. Version 1.1.1. Open Network Foundation, März 2013. URL: https://www.opennetworking.org/wp-content/uploads/2013/02/of-config-1-1-1.pdf (zitiert auf S. 12).

[6] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, P. Nikander. „LIPSIN: line speed publish/subscribe inter-networking“. In: *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. SIGCOMM '09. Barcelona, Spain: ACM, 2009, S. 195–206. ISBN: 978-1-60558-594-9. DOI: 10.1145/1592568.1592592. URL: http://doi.acm.org/10.1145/1592568.1592592 (zitiert auf S. 12, 26, 27).

[7] S. Bhowmik, M. A. Tariq, B. Koldehofe, F. Dürr, T. Kohler, K. Rothermel. „High Performance Publish/Subscribe Middleware in Software-Defined Networks“. In: *IEEE/ACM Transactions on Networking* 25.3 (2017), S. 1501–1516 (zitiert auf S. 12, 27, 28, 31–33, 35, 63).

[8] M. A. Tariq, B. Koldehofe, S. Bhowmik, K. Rothermel. „PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware“. In: *Proceedings of 15th International Middleware Conference*. Bordeaux, France, 2014. ISBN: 978-1-4503-2785-5. DOI: 10.1145/2663165.2663338 (zitiert auf S. 12, 27, 29, 30, 35).

[9] S. Bhowmik, M. A. Tariq, A. Balogh, K. Rothermel. „Addressing TCAM Limitations of Software-Defined Networks for Content-Based Routing“. In: *Proceedings of the 11th ACM International Conference on Distributed Event-based Systems*. DEBS 2017. 2017 (zitiert auf S. 12, 27).

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker. „P4: Programming Protocol-independent Packet Processors“. In: *SIGCOMM Comput. Commun. Rev.* 44.3 (Juli 2014), S. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: http://doi.acm.org/10.1145/2656877.2656890 (zitiert auf S. 12, 18).

[11] *The P4 Language Specification*. Version 1.0.4. The P4 Language Consortium, Mai 2017. URL: https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf (zitiert auf S. 18, 19).

[12] *The P4$_{16}$ Language Specification*. Version 1.0.0. The P4 Language Consortium, Mai 2017. URL: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf (zitiert auf S. 18).

[13] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, K. Rothermel. „P4CEP: Towards In-Network Complex Event Processing“. In: *Proceedings of the 2018 Morning Workshop on In-Network Computing*. NetCompute '18. Budapest, Hungary: ACM, 2018, S. 33–38. ISBN: 978-1-4503-5908-5. DOI: 10.1145/3229591.3229593. URL: http://doi.acm.org/10.1145/3229591.3229593 (zitiert auf S. 25).

[14] B. H. Bloom. „Space/Time Trade-Offs in Hash Coding With Allowable Errors“. In: 13 (Juli 1970), S. 422–426 (zitiert auf S. 26).

[15] S. Bhowmik. „Content-based routing in software-defined networks“. Diss. University of Stuttgart, 2017 (zitiert auf S. 27).

[16] S. Bhowmik, M. A. Tariq, B. Koldehofe, A. Kutzleb, K. Rothermel. „Distributed Control Plane for Software-defined Networks: A Case Study Using Event-based Middleware“. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15. Oslo, Norway, 2015 (zitiert auf S. 27).

[17] S. Bhowmik. „Distributed control algorithms for adapting publish/subscribe in software-defined networks“. Magisterarb. University of Stuttgart, Nov. 2013 (zitiert auf S. 27).

[18] S. Bhowmik, M. A. Tariq, J. Grunert, K. Rothermel. „Bandwidth-efficient Content-based Routing on Software-defined Networks“. In: *Proceedings of the 10th ACM International Conference on Distributed Event-based Systems*. DEBS 2016. 2016 (zitiert auf S. 27).

[19] S. Bhowmik, M. A. Tariq, L. Hegazy, K. Rothermel. „Hybrid Content-based Routing Using Network and Application Layer Filtering“. In: *Proceedings of 36th IEEE International Conference on Distributed Computing Systems*. ICDCS '16. 2016 (zitiert auf S. 27, 34, 63).

[20] S. Bhowmik, M. A. Tariq, J. Grunert, D. Srinivasan, K. Rothermel. „Expressive Content-Based Routing in Software-Defined Networks“. In: PP (Mai 2018), S. 1–1 (zitiert auf S. 27).

[21] B. Koldehofe, F. Dürr, M. A. Tariq, K. Rothermel. *The Power of Software-defined Networking: Line-rate Content-based Routing Using OpenFlow*. Dez. 2012 (zitiert auf S. 28).

[22] *P4Runtime Specification*. Version 1.0.0-rc2. The P4.org API Working Group, Aug. 2018. URL: https://s3-us-west-2.amazonaws.com/p4runtime/docs/v1.0.0-rc2/P4Runtime-Spec.pdf (zitiert auf S. 35, 53, 63).

[23] *Behavioral Model*. URL: http://www.bmv2.org/ (zitiert auf S. 53).

[24] *Mininet*. URL: http://mininet.org/ (zitiert auf S. 53).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift