Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik
Arbeitsbereich Kommunikationsnetze

Masterarbeit Informatik

Implementierung von Per-Stream Filtering and Policing für Time-Sensitive Networking auf einem 100G-fähigen Switch mithilfe von P4

Fabian Ihle

07.03.2023

Gutachter

Prof. Dr. habil. Michael Menth
Fachbereich Informatik
Arbeitsbereich Kommunikationsnetze
Universität Tübingen

Jun. Prof. Dr.-Ing. Setareh Maghsudi
Fachbereich Informatik
Research Group Decision Making
Universität Tübingen

Betreuer

Steffen Lindner M.Sc.

Fabian Ihle:
Implementierung von Per-Stream Filtering and Policing für Time-Sensitive Networking auf einem 100G-fähigen Switch mithilfe von P4

Masterarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.11.2022 - 07.03.2023

Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, den 07.03.2023

(Fabian Ihle)

Inhaltsverzeichnis

Ethernet-basierte Netzwerke halten Einzug in diverse Industrieanwendungen, sowie dem autonomen Fahren und dem Internet-of-Things. Um den hohen, zeitkritischen Anforderungen dieser Systeme gerecht zu werden, erweitert Time-Sensitive Networking das Best-Effort-Routing von Ethernet und garantiert so eine deterministische Latenz. Um dies zu erreichen schließen Teilnehmer eines Netzwerks einen Vertrag mit diesem, in dem die Menge der Daten, sowie deren Übertragungszeitpunkte vereinbart werden. Eine Schwachstelle dieses Ansatzes ist, dass die Einhaltung der abgeschlossenen Verträge nicht kontrolliert wird. Ein fehlkonfigurierter, oder böswilliger Host ist daher in der Lage die garantierten Latenzen des gesamten Netzwerkes zu gefährden, indem er die reservierten Ressourcen aufbraucht. *Per-Stream Filtering and Policing (PSFP)* ist ein Mechanismus, der dieses Problem verhindern kann. Er überwacht die Streams im Netzwerk auf die Einhaltung von Verträgen und schließt fehlverhaltende Hosts von der Kommunikation im Netzwerk aus.

Im Kontext von Software Defined Networking erläutert diese Arbeit die Grundlagen von Network Softwarization bezüglich der Programmierung der Control- und Data Plane, sowie die Grundzüge der P4-Programmiersprache. Außerdem wird der Nutzen, sowie der Aufbau der PSFP-Komponenten im Zusammenhang mit Time-Sensitive Networking aufgezeigt. Des Weiteren wird eine Implementierung des PSFP-Mechanismus in der P4-Programmiersprache auf einem 100G Intel® Tofino Switch-ASIC beschrieben. Die korrekte Funktion der einzelnen PSFP-Komponenten konnte durch die Durchführung isolierter Testfälle in einer Simulation verifiziert werden. Das Resultat der Arbeit, bestehend aus dem P4-Programm der Data-Plane, sowie dem Python3-Programm der Control-Plane, wird zusammengefasst und ein Ausblick auf weitere mögliche Arbeiten in diesem Bereich gegeben.

1. Einführung

In der Vergangenheit wurden zur Umstellung von analoger auf digitale Technik spezielle Systeme für die Kommunikation zeitkritischer Anwendungen entwickelt. Mit zunehmender Digitalisierung in den letzten Dekaden haben sich Ethernet-Netzwerke als kostengünstiger und praktikabler als speziell entwickelte Systeme erwiesen [19]. Für diverse zeitkritische Industrieanwendungen [10], z.B. zur Koordinierung von Maschinen, dem Internet-of-Things, oder dem autonomem Fahren, ist eine garantierte Zustellung von Frames mit einer deterministischen Latenz essentiell. Übertragungen in Ethernet-Netzwerken, die per Best-Effort-Routing erfolgen, können dies nicht gewährleisten. Die im Standard 802.1D [1] eingeführten Verkehrsklassen priorisieren zwar zeitkritischen Verkehr, fügen jedoch eine zusätzliche Latenz bei Verwendung mehrerer Flows hinzu [10]. Daher musste eine neue Lösung gefunden werden, um Ethernet-Netzwerke auch für zeitkritische Industrieanwendungen zur Kommunikation nutzen zu können [19].

Eine mögliche Lösung dieser Problematik bietet *Time-Sensitive Networking (TSN)*. TSN ermöglicht durch die Interaktion verschiedener, neuer Standards [3, 4, 21] ein Versprechen hinsichtlich einer garantierten Zustellung unter deterministischer Latenz zu geben [10]. Um dies zu erreichen, teilt TSN die Übertragung in periodische Zeitabschnitte ein und schließt einen Vertrag mit jedem TSN-Teilnehmer. In diesem Vertrag wird festgelegt, welcher Teilnehmer innerhalb eines bestimmten Intervalls des periodisch geplanten Zeitplans eine bestimmte Menge an Daten senden darf. Im Gegenzug reserviert das TSN-Netzwerk die Ressourcen für jeden Teilnehmer. In einem derart aufgeteilten und geplanten Netzwerk kann zumindest theoretisch niemals ein Paketverlust durch Stau auftreten, da die verfügbaren Kapazitäten von den TSN-Teilnehmern niemals überschritten werden [19].

In der Praxis überwacht ein reines TSN-Netzwerk die Einhaltung der geschlossenen Verträge nicht. Dies ermöglicht es, Teilnehmern des Netzwerks durch Fehlkonfiguration, oder gezielte Angriffe die reservierten Ressourcen anderer Teilnehmer für sich zu beanspruchen. *Per-Stream Filtering and Policing (PSFP)* adressiert dieses Problem, in dem es Mechanismen und Algorithmen zur Identifikation von Frames, sowie zur Überwachung der zugesicherten Bandbreiten und Zeitintervalle bereitstellt. Bei Verstoß eines TSN-Teilnehmers gegen seinen Vertrag kann PSFP eingreifen und den Teilnehmer permanent, oder temporär von der Kommunikation ausschließen [4].

Im traditionellen Networking liefern die Hersteller den Endanwendern Netzwerkgeräte mit einem festen Funktionsumfang. Diese lassen sich vom Anwender zwar konfigurieren, aber durch die enge Kopplung der Kontrolllogik an die Forwarding-Logik nicht um zusätzliche Funktionalität erweitern und selbst programmieren. Durch die-

se Herstellerbindung mit proprietärer Soft- und Hardware limitiert der Hersteller den Einsatz neuer Technologien durch Vorgaben und lange Entwicklungszyklen. *Software-Defined Networking (SDN)* trennt die Kontrolllogik der Control Plane von der Forwarding-Logik der Data Plane und lagert die Kontrolllogik in einen zentralisierten Controller aus. Dies ermöglicht darauf aufbauend eine eigene Programmierung und somit eine Spezifikation der Funktionalität von Netzwerkgeräten. Zusätzlich führt SDN eine Schnittstelle ein, die es ermöglicht, die Data Plane mit einer Programmiersprache, wie z.B. *P4*, zu beschreiben. Somit können eigene Algorithmen, wie z.B. PSFP in Kombination mit TSN, auf Netzwerkgeräten implementiert werden.

1.1. Ziel der Thesis

Diese Arbeit implementiert die Funktionalität von PSFP zur Überwachung der Einhaltung von TSN-Verträgen mithilfe der P4-Programmiersprache und den Konzepten von SDN auf einem programmierbaren Intel® Tofino Switch-ASIC. Ziel dieser Arbeit ist es, die Möglichkeit der Umsetzung von PSFP in P4 auf eigener Hardware zu demonstrieren und die korrekte Funktionsweise des Mechanismus zu verifizieren.

1.2. Struktur der Thesis

Kapitel 2 schafft zunächst eine Basis der Grundlagen hinsichtlich der Themengebiete Network Softwarization, der P4-Programmiersprache, TSN und schlussendlich PSFP, die für diese Arbeit notwendig sind. In Kapitel 3 werden verwandte Arbeiten vorgestellt, insbesondere eine Simulation von PSFP im Framework OMNeT++, sowie DPTP, einer Zeitsynchronisierung im Kontext zu SDN und TSN. Anschließend beschreibt Kapitel 4 die Implementierung von PSFP in der P4-Programmiersprache auf einem 100G Intel® Tofino Switch-ASIC. Die korrekte Funktionsweise der PSFP-Implementierung wird in Kapitel 5 verifiziert und erläutert. Abschließend werden in Kapitel 6 die Ergebnisse der Arbeit zusammengefasst und ein Ausblick in die Zukunft gegeben.

2. Grundlagen

Dieses Kapitel erläutert die theoretischen Grundlagen, die für die Umsetzung dieser Arbeit benötigt werden. Zunächst werden die grundlegenden Konzepte der Control- und Data Plane im Kontext von Network Softwarization und SDN erklärt. Anschließend wird auf die Spezifikation der P4-Sprache zur Programmierung der Data Plane eingegangen. Abschließend werden die Ideen hinter Time-Sensitive Networking (TSN) im Kontext von Per-Stream Filtering and Policing (PSFP) zusammengefasst und die Funktionalität anhand des schematischen Aufbaus von PSFP verdeutlicht.

2.1. Network Softwarization

In diesem Abschnitt werden die Prinzipien von Software-Defined Networking (SDN), insbesondere der Control- und Data Plane, vorgestellt. Anhand der Programmiersprache P4 wird anschließend aufgezeigt, wie sich diese Prinzipien in der Praxis implementieren lassen.

Klassisches Networking verwendet meist proprietäre Soft- und Hardware, die von einem Netzwerkadministrator manuell konfiguriert werden muss. Die speziellen Anforderungen an das Netzwerk, insbesondere in großen Rechenzentren, bedeuten einen erheblichen Zeit- und Kostenaufwand für die Entwicklung neuer Technologien durch die Hersteller. Durch Konzepte wie SDN und Network Softwarization kann dies durch die Verwendung offener Standards, wie z.B. OpenFlow (OF) der Open Networking Foundation, reduziert werden [18].

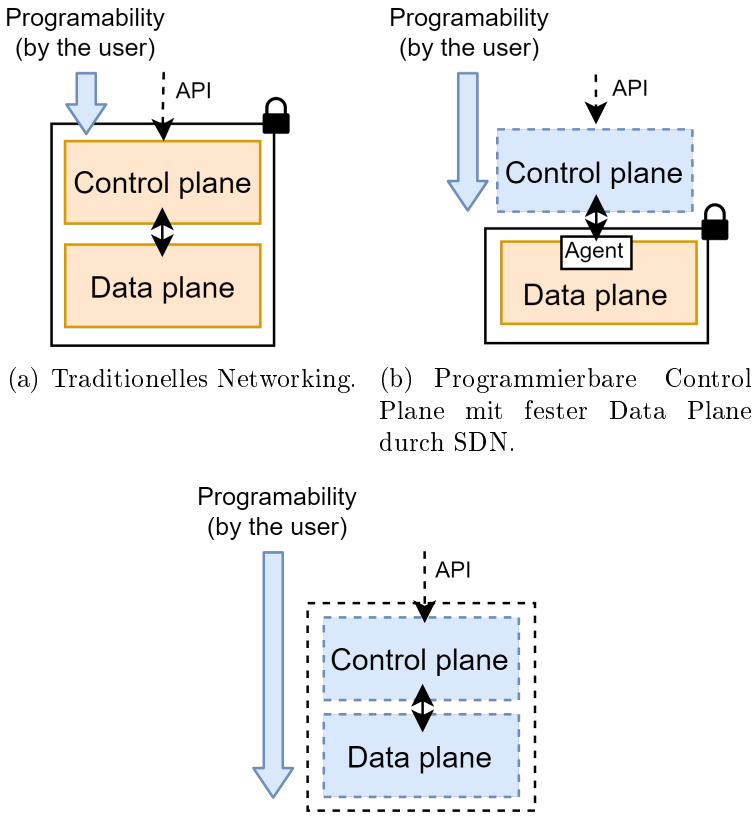
Konventionelle Algorithmen zur Paketverarbeitung in Netzwerkgeräten lassen sich in drei Klassen einteilen: *Data Plane*, *Control Plane* und *Management Plane*, wobei letztere den geringsten Einfluss auf die Paketverarbeitung hat [12].

In einer herkömmlichen Umgebung sind die Management Plane und Eigenschaften der Control Plane vom Benutzer verwalt- und konfigurier-, jedoch nicht programmierbar. Hierfür wird z.B. oft das SNMP Protokoll verwendet. Die *Data Plane* beschreibt Algorithmen zur Weiterleitung auf Basis der Informationen, die die *Control Plane* bereitstellt. Die zugrundeliegenden Algorithmen, insbesondere die der Data Plane, können nicht vom Anwender verändert werden, sondern werden ausschließlich vom Hersteller implementiert.

Network Softwarization trennt in erster Linie die Control Plane von der Data Plane und ermöglicht so durch weitere Standards, wie OF, eine Kommunikation zwischen diesen. Die Kommunikation mit den beiden Planes ermöglicht deren Programmierbarkeit. Hauser et al. definieren den Begriff der *Network Programmability* in ihrer

Arbeit [12] als die Fähigkeit, die Algorithmen der Data- und Control Plane nach eigenem Ermessen vollständig zu definieren. Beide Planes können vom Anwender mit eigenen Algorithmen programmiert werden, ohne dass der Hersteller involviert werden muss. So kann das Netzwerk genau nach den Anforderungen an Performance und Skalierbarkeit aufgebaut werden. Der Unterschied zwischen den verschiedenen Konzepten ist in Abbildung 2.1 illustriert.

Um dies zu erreichen, führt *SDN* zunächst ein Application Programming Interface (API) ein, das es ermöglicht, die Algorithmen der Control Plane durch eigene Algorithmen zu ersetzen und in einen zentralisierten Controller auszulagern. Vergleichbar dazu erlaubt es die Programmierung der Data Plane, z.B. mithilfe von *P4*, eigene Algorithmen für die Data Plane zu definieren.



(a) Traditionelles Networking.

(b) Programmierbare Control Plane mit fester Data Plane durch SDN.

(c) Network Programmability.

Abbildung 2.1.: Networking Konzepte nach Hauser et al. [12].

2.1.1. Control Plane

Die *Control Plane* hat die Aufgabe, die Data Plane mit den notwendigen Informationen zu versorgen, um Entscheidungen über die Verarbeitung und Weiterleitung von Paketen zu treffen. Hierfür verwendet SDN eine API, die es einerseits ermöglicht, die Control Plane mit Daten zu versorgen und andererseits das Netzwerk von einem zentralen Controller aus zu verwalten. Zentralisiert verwaltete Algorithmen haben sich als wesentlich einfacher und effizienter erwiesen, als konventionelle, dezentralisierte Algorithmen [12].

Ein Beispiel für die Programmierbarkeit der Control Plane ist die manuelle Implementierung von Layer 2 Switching, bei der die Abbildung der Quelladresse auf die Zieladresse und den Switchport im zentralen Controller verwaltet wird. Ein Host sendet dabei eine Address Resolution Protocol (ARP)-Anfrage an eine SDN kontrollierte Bridge, welche diese zunächst an den Controller weiterleitet. Der Controller kann daraus die Adresse und den Port des Hosts lernen und seinerseits eine erneute ARP-Anfrage über alle Ports der Bridge senden, woraus der Controller ebenfalls die Adresse und den Port des Ziel-Hosts lernt. Daraus resultierend lässt sich eine zentrale ARP-Tabelle und Topologie generieren.

Ein SDN-basierter Controller kann zwar die Control Plane frei gestalten, die Herausforderung bei der Standardisierung der SDN-API war jedoch die Annahme, dass die Data Plane, passend zur Control Plane, eine einheitliche Funktionalität bietet. Dies ist jedoch nicht der Fall, so dass eine mögliche Programmierung der Data Plane unumgänglich wurde, um die gewünschte Performance und Flexibilität der *Network Programmability* zu erreichen [12].

2.1.2. Data Plane

Die *Data Plane* ist für die Verarbeitung aller Pakete, die das Netzwerkgerät durchlaufen, verantwortlich und definiert somit durch ihre Algorithmen die Funktionalität, Performance und Skalierbarkeit des Netzwerks. Durch die Programmierbarkeit der Data Plane hat der Anwender die Möglichkeit, das Netzwerk und dessen Funktionalität genau an seine Bedürfnisse anzupassen. Dies ist insbesondere in großen Rechenzentren ein großer Vorteil. Theoretisch könnten diese Algorithmen auch in der Control Plane implementiert werden, was jedoch in der Praxis zu massiven Performanceeinbußen führen würde [12].

Ein praktisches Beispiel für die Programmierung der Data Plane der L2 Switching Control Plane aus dem vorigen Abschnitt ist die Forwarding Logik. Diese extrahiert zunächst die entsprechenden Felder aus den Paket-Headern und trifft anschließend eine Entscheidung anhand der Daten aus der Control Plane, ob und wohin das Paket weitergeleitet werden soll. Auch wenn dieses Beispiel trivial erscheint, kann die Data Plane beliebig um weitere Algorithmen erweitert werden, wie z.B. Ethernet Protection Switching [7], oder IPv6in4 Tunneling [24].

Aufgrund der inzwischen stark gestiegenen Leistungsanforderungen an die Prozessoren durch weitaus höhere Bandbreiten ist eine General-Purpose CPU heute nicht mehr ausreichend für die Paketverarbeitung. Stattdessen müssen spezielle Chips (*Application-specific integrated circuits (ASICs)*) eingesetzt werden [12]. Im Gegensatz zu General-Purpose Prozessoren stellt die Programmierbarkeit auf ASICs eine Herausforderung aufgrund von hardwarespezifischen Besonderheiten dar, die berücksichtigt werden müssen. Um dennoch Algorithmen in gängigen Programmiersprachen definieren zu können, existieren verschiedene Data Plane Modelle (Architekturen), für die die Algorithmen abstrahiert dargestellt werden können. Der abstrahierte Code kann dann für ein Netzwerkgerät, das die entsprechende Architektur in der Hardware implementiert, kompiliert werden. Ein solches Netzwerkgerät wird als *Target* bezeichnet [12].

2.2. Die P4 Programmiersprache

Dieser Abschnitt erläutert die für die Implementierung dieser Arbeit relevanten Komponenten der P4-Programmiersprache am Beispiel der Portable Switch Architecture (PSA). Zudem werden abschließend die Besonderheiten der verwendeten Tofino Native Architecture (TNA) aufgezeigt, die eng verwandt mit der PSA ist.

Programming protocol-independent packet processors (P4) ist einerseits eine höhere Programmiersprache zur Beschreibung der Data Plane, andererseits aber auch ein auf Architekturen basierendes Programmiermodell. Vereinfacht gesagt werden in einem P4-Programm Paketverarbeitungsoperationen definiert. Die Data Plane wird dadurch vollständig programmierbar, wodurch die Funktionalität des Netzwerks nur noch durch die P4-Sprache, nicht aber durch den Hersteller der Hardware limitiert wird. $P4_{16}$ ist die derzeit aktuellste Spezifikation der Sprache, die im Jahr 2017 veröffentlicht wurde und die vorige Spezifikation $P4_{14}$, die erstmals 2015 veröffentlicht wurde, ablöst. Die aktuellste Version 1.2.2 von $P4_{16}$ wurde im Mai 2021 veröffentlicht [25]. Im Folgenden wird mit P4 die Spezifikation $P4_{16}$ bezeichnet, die derzeit die am weitesten verbreitete, abstrakte Programmiersprache zur Beschreibung der Data Plane ist.

Ein P4-Programm wird immer target-spezifisch für eine bestimmte Architektur entwickelt und kann daher auf jedem Target ausgeführt werden, das die gleiche Architektur umsetzt. Ein Target kann hardwarebasiert (FPGA, ASIC), oder softwarebasiert sein. Der Hersteller des Targets bietet, zusätzlich zur Architektur, einen P4-Compiler an, der das P4-Programm in target-spezifischen Code umwandelt, welcher dann vom Target ausgeführt wird. Auf diese Weise können verschiedene Compiler ein und dasselbe P4-Programm für verschiedene Targets erzeugen. Zusätzlich generiert der P4-Compiler eine Laufzeitumgebung, die es der Control Plane ermöglicht, mit der Data Plane zu kommunizieren [12, 26]. Dieser Prozess ist in Abbildung 2.2 dargestellt.

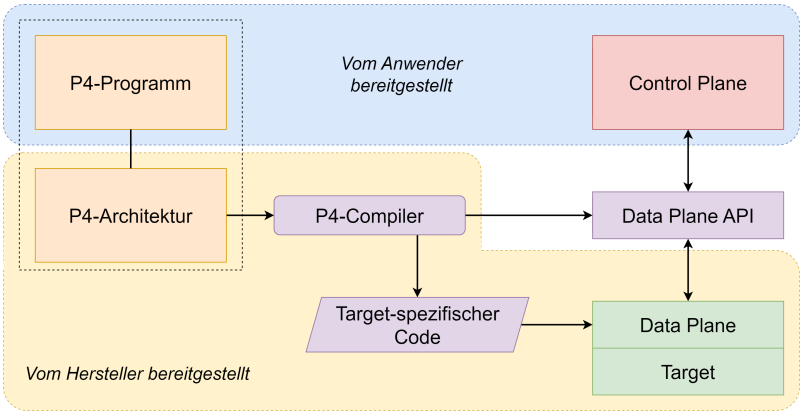


Abbildung 2.2.: P4-Workflow nach [12, 26].

2.2.1. Aufbau eines P4 Programms

Im Folgenden werden die allgemeine Struktur und der Ablauf eines P4-Programms auf Basis der *Portable Switch Architecture (PSA)* beschrieben. Diese Architektur definiert verschiedene Datentypen, den In- und Egress-Kontrollblock, den Parser- bzw. Deparser-Kontrollblock, target-spezifische Methoden (Externs) und Pfade für die Paketverarbeitung innerhalb eines Switches mit mehreren Interface-Ports. Diese werden in diesem Abschnitt erläutert [27].

Die PSA besteht aus sechs programmierbaren P4- und zwei festen Blöcken, die in einer Pipeline organisiert sind. Abbildung 2.3 zeigt die Bearbeitungsreihenfolge der einzelnen Kontrollblöcke innerhalb der Pipeline. Der Ingress- bzw. Egress-*Parser* extrahiert zunächst die Header-Felder eines eingehenden Pakets und speichert diese temporär ab, um sie im *Ingress*- bzw. *Egress*-Kontrollblock referenzieren zu können. Im Ingress-Block befinden sich mehrere Match-Action Einheiten, die eine Entscheidung über die Weiterleitung oder Manipulation des Pakets treffen. Anschließend wird das Paket im *Deparser* serialisiert und im *Packet Buffer* zwischengespeichert. Im Egress befinden sich weitere Match-Action Einheiten, die eine Entscheidung hinsichtlich der Queue-Zuordnung des Pakets treffen können. Schließlich wird das Paket vom Egress-Deparser serialisiert, gequeued und versendet. Der *Packet Buffer* und die *Queuing Engine* sind feste Blöcke und nicht programmierbar, sondern nur durch die Control Plane konfigurierbar [12, 27].

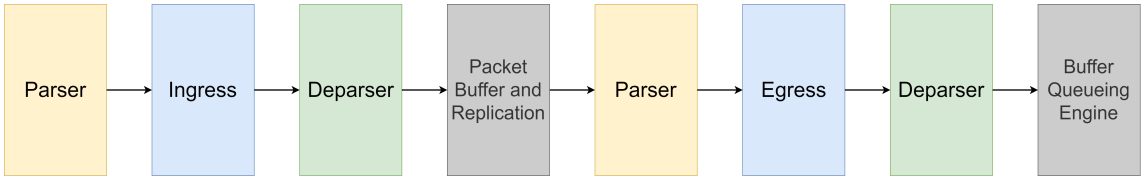


Abbildung 2.3.: Portable Switch Pipeline der PSA [27].

2.2.1.1. Datentypen

Innerhalb eines P4-Programms können Daten verschiedener Datentypen gehalten und verarbeitet werden. Die PSA unterstützt die in der P4-Sprache spezifizierten, *grundlegenden* Datentypen, wie `bool` und `signed` bzw. `unsigned int`. Die Größe von Variablen des Typs `int` ist bitweise definiert. Es können also beispielsweise Felder der Größe `bit<1>`, `bit<48>` oder größer deklariert werden.

Aus den grundlegenden Datentypen werden weitere Datentypen abgeleitet. Der `Header`-Typ beschreibt einen Paketheader, der sich aus den einzelnen Bitfeldern zusammensetzt. Zusätzlich besitzt ein `Header`-Typ ein implizites *valid* Flag, das zu Beginn auf ungültig gesetzt ist und beim Extrahieren des Pakets vom Parser auf gültig gesetzt wird. Ein `struct`-Typ fasst, darauf aufbauend, mehrere einfache und `Header`-Typen zusammen, um beispielsweise den kompletten Header-Stack eines Pakets zu beschreiben. Listing 2.1 zeigt die Definition eines 48 Bit breiten Typs für eine MAC-Adresse, die in einem Ethernet-`Header` verwendet wird. In diesem Beispiel beschreibt ein `struct`-Typ, bestehend aus dem Ethernet-`Header` und einem Ethernet 802.1Q-`Header`, den Header-Stack eines Pakets.

```
typedef bit<48> mac_addr_t;

header ethernet_t {
    mac_addr_t dst_addr;
    mac_addr_t src_addr;
    bit<16> ether_type;
}
header eth_802_1q_t {
    bit<3> pcp;
    bit<1> dei;
    bit<12> vid;
    bit<16> ether_type;
}

struct header_t {
    ethernet_t ethernet;
    eth_802_1q_t eth_802_1q;
}
```

Listing 2.1: Beispielhafter *struct*-Typ, bestehend aus mehreren *Header*-Typen.

Das Target selbst generiert, abgesehen von den geparsten `Header`-Feldern, zusätzliche, sogenannte *intrinsische Metadaten* und speichert diese für die Verarbeitung eines Pakets. Deren Struktur ist durch die Architektur vorgegeben und enthält unter anderem Werte für den Ingress- bzw. Egress-Port, den Zeitstempel, die Queue-Zuordnung, die Paketlänge und weitere Flags.

In einem weiteren `struct` können *benutzerdefinierte Metadaten* gespeichert werden. Diese werden zum Lesen und Schreiben von Werten während der Paketverarbeitung verwendet. Im Gegensatz zu `Header`-Feldern verlassen die Metadaten den Ingress- bzw. Egress-Kontrollblock nicht und besitzen nur dort ihre Gültigkeit. Werden dennoch Metadaten aus dem Ingress im Egress benötigt, so müssen diese über einen *Bridge Header* dem Header-Stack des Pakets im Ingress hinzugefügt, vom Ingress-Deparser ausgegeben und vom Egress-Parser wieder entnommen werden.

2.2.1.2. Parser

Der *Parser* hat die Aufgabe, die Header-Felder eines Pakets zu extrahieren und in den zuvor deklarierten Datenstrukturen, die in der Regel aus abgeleiteten Datentypen bestehen, zu speichern. Dabei setzt er das *valid*-Flag des jeweils korrespondierenden `Headers` auf gültig.

Der Kontrollfluss des Parsers, dargestellt als endlicher Automat, weist einen expliziten Startzustand, mehrere selbst definierte Zustände und zwei Endzustände, *accept* und *reject*, auf. Die nach Erreichen des Zustandes *accept* verbleibenden, nicht geparsten Daten werden als Nutzdaten behandelt und an das Paket angehängt. Auf diese kann im P4-Programm nicht zugegriffen werden.

Listing 2.2 zeigt einen beispielhaften Ingress-Parser für ein Paket, bei dem der Parser zunächst den Ethernet-Header extrahiert und speichert. Ein Paket in diesem P4-Programm kann einen optionalen 802.1Q-Header enthalten, der abhängig des Felds `ether_type` im Ethernet-Header geparst wird, sofern er existiert. In beiden Fällen muss anschließend ein IPv4-Header folgen, sonst verwirft der Parser das Paket. Abbildung 2.4 zeigt den Ablauf des Parsers als endlichen Automaten, Listing 2.2 den zugehörigen P4-Code.

```
parser IngressParser(packet_in pkt, out headers hdr) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        pkt.extract(hdr);
        transition select (hdr.ethernet.ether_type){
            0x0800: parse_ipv4;
            0x8100: parse_8021q;
            default: reject;
        }
    }
    state parse_8021q {
        pkt.extract(hdr.eth_802_1q);
        transition select (hdr.ethernet.ether_type){
            0x800: parse_ipv4;
            default: reject;
        }
    }
    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition accept;
    }
}
```

Listing 2.2: Beispielhafter Ingress-Parser in P4.

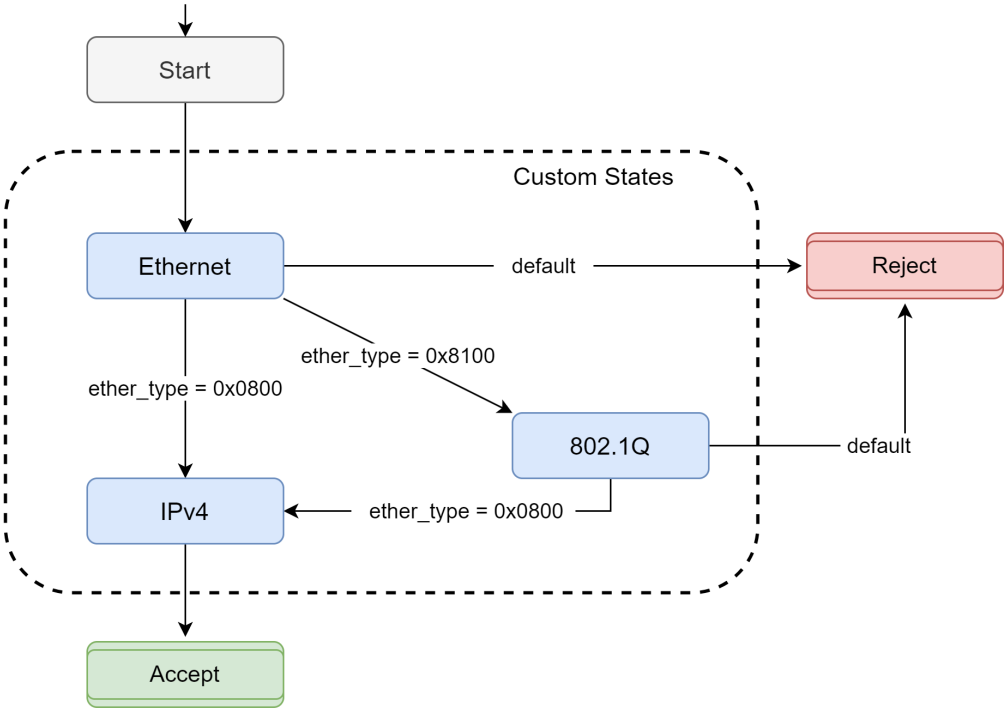


Abbildung 2.4.: Endlicher Automat des Beispielparsers.

2.2.1.3. Ingress- und Egress-Kontrollblöcke

In- und Egress definieren verschiedene Kontrollblöcke, die Operationen auf Paketen durchführen können. Abhängig von der P4-Architektur sind unterschiedliche, intrinsische Metadaten im Ingress und Egress verfügbar. Beispielsweise kann ein Kontrollblock nur im Ingress eine Paketweiterleitungsentscheidung auf der Grundlage des Ingress-Ports treffen. Ein Kontrollblock beginnt mit dem Schlüsselwort `control` und enthält verschiedene Ressourcendefinitionen im Rumpf des Blocks. Diese können aus *Tabellen*, *Actions*, oder der Instanziierung von *Externs* bestehen.

Ein Kernelement der Kontrollblöcke stellen die *Match-Action Units* dar. Diese bestehen aus einer *Match-Action-Table (MAT)* und mehreren definierten *Actions*. Eine MAT definiert Schlüsselfelder, gegen die sie Header- oder Metadatenfelder eines Pakets prüft. Bei einer Übereinstimmung führt das P4-Programm eine der in der MAT verknüpften Actions aus. *Actions* sind Codefragmente, die mit Funktionen in konventionellen Programmiersprachen vergleichbar sind. Sie können Header- oder Metadatenfelder lesen und schreiben, besitzen jedoch keinen Rückgabewert. Der Kontrollblock ruft eine Action entweder direkt mit ihrem Namen auf, oder weist sie einer MAT zu. Für die Prüfung auf Übereinstimmung existieren in der P4-Spezifikation drei Typen: `exact`, `ternary` und `LPM`.

Im Falle einer `ternary`-Übereinstimmung spezifiziert die Control Plane zusätzlich eine Bitmaske, die die MAT auf das zu prüfende Feld anwendet. Die MAT repräsentiert die Einträge als geordnete Liste und überprüft diese sequentiell. Der erste Eintrag, der auf diese Bitmaske zutrifft, wird als Treffer gewertet und die dazugehörige Action ausgeführt. Ein Spezialfall des ternären Typs ist der Typ *Longest-Prefix-Match (LPM)*, der als Bitmaske konsekutiver Einsen definiert ist. Der Eintrag, dessen längste Folge von Einsen mit dem definierten Präfix übereinstimmt, erhält dabei einen Treffer. Der `exact`-Typ kann als Sonderfall einer `LPM` Übereinstimmung mit einem Präfix maximaler Länge betrachtet werden.

Der Inhalt einer MAT, inklusive der zugewiesenen Action und ihrer Parameter bei einem Treffer, ist nicht Teil des P4-Programms, sondern wird von der Control Plane mit Daten befüllt. Darüber hinaus können der Tabelle im P4-Programm weitere Parameter zugewiesen werden, wie z.B. die maximale Tabellengröße oder eine Standard-Action, die ausgeführt werden soll, wenn keine Übereinstimmung gefunden wird.

Zusätzlich zu den Ressourcendefinitionen muss der Kontrollblock eine `apply()`-Funktion definieren, die den auf ein Paket anzuwendenden Algorithmus beschreibt. Diese Funktion greift auf die Ressourcen des Kontrollblocks zu. Das P4-Programm führt diesen Algorithmus aus, wenn die `apply()`-Funktion eines Kontrollblocks aufgerufen wird. In der `apply()`-Funktion sind Programmkonstrukte wie Verzweigungen durch `if` und `switch`-Ausdrücke, Auswertungen von logischen oder arithmetischen Ausdrücken (beschränkt auf Addition und Subtraktion) und Aufrufe von anderen Kontrollblöcken, oder MATs mit Überprüfung auf Übereinstimmung möglich.

Besonders zu beachten ist, dass ein Algorithmus als gerichteter azyklischer Graph darstellbar sein muss. Dies bedeutet, dass Schleifenkonstrukte wie `for` oder `while` in einem P4-Programm nicht existieren.

Listing 2.3 zeigt einen beispielhaften Kontrollblock, der ein einfaches IPv4 Forwarding implementiert. Er definiert zwei Actions zum Weiterleiten und Verwerfen von Paketen sowie eine LPM-Tabelle für IPv4-Adressen. Der Kontrollblock wendet zunächst auf ein eingehendes Paket in der `apply()`-Funktion die MAT an, die dann bei einer Übereinstimmung die `ipv4_forward`-Action aufruft und die Header-, sowie Metadatenfelder entsprechend manipuliert. Wird kein passender Eintrag in der Tabelle gefunden, wird das Paket verworfen.

```
control IngressIPv4(inout headers hdr,
                    inout metadata meta,
                    inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

Listing 2.3: Beispielhafter IPv4-Kontrollblock in P4 [23].

2.2.1.4. Deparser

Der *Deparser* ist ein spezieller Kontrollblock, der die Paket-Header in der spezifizierten Reihenfolge serialisiert und in einen Bytestream umwandelt. Dabei gibt er nur die Header, bei denen das implizite *valid*-Flag gesetzt ist, mit einem `emit()`-Aufruf aus.

Listing 2.4 zeigt einen beispielhaften Deparser für das Eingangsbeispiel des Parsers in Abschnitt 2.2.1.2, der einen optionalen 802.1Q-Header parst. Wenn ein Paket diesen nicht enthält, ist das *valid*-Flag nicht gesetzt und der Deparser gibt nur den Ethernet- und IPv4-Header aus.

```
control IngressDeparser(packet_out pkt, in headers hdr){
    apply {
        pkt.emit(hdr.ethernet);
        pkt.emit(hdr.eth_802_1q);
        pkt.emit(hdr.ipv4);
    }
}
```

Listing 2.4: Beispielhafter Deparser in P4.

2.2.1.5. Externs

Extern-Objekte erweitern die Funktionalität eines P4-Programms durch die Definition zusätzlicher, target-spezifischer Methoden. Externs in P4-Programmen sind vergleichbar mit abstrakten Klassendefinitionen in der objektorientierten Programmierung, wobei die tatsächliche Implementierung der zusätzlichen Funktionalität vom Target abhängt. In der PSA sind mehrere Externs definiert, darunter die Folgenden [27]:

- *Counter* können Statistiken über verarbeitete Bytes und Pakete erheben.
- *Meter* implementieren eine 3-Farben Markierung nach RFC2698 [13] und können komplexere Statistiken sammeln.
- *Register* sind zustandsorientierte Speicher, deren Werte während der Paketverarbeitung gelesen und geschrieben werden.

2.2.2. Tofino Native Architecture (TNA)

Der Tofino Switch-ASIC ist ein von Intel® produziertes Hardware-Target, das die *Tofino Native Architecture (TNA)* implementiert. Die Architektur des Targets ist der in Abschnitt 2.2.1 beschriebenen PSA sehr ähnlich, unterscheidet sich jedoch in einigen wichtigen Punkten, die in diesem Abschnitt aufgezeigt werden [15].

Der schematische Aufbau der *TNA-Pipeline* ist in Abbildung 2.5 beschrieben. Je nach Hardwareausführung besteht die Pipeline aus zwei oder vier identisch aufgebauten Pipes. Diese Pipes sind individuell oder simultan programmierbar. Neben zugeordneten Input- und Output Ports, besteht jede Pipe aus P4-programmierbaren

In- bzw. Egressblöcken. Im Gegensatz zur PSA findet das Queueing von Frames hier jedoch zwischen Ingress und Egress und nicht am Ende der Pipe statt (vgl. Abbildung 2.3: PSA) [15].

Zusätzlich besitzt jede Pipe einen *Paketgenerator*, der Pakete, ausgehend vom Tofino Switch, generiert und an die eigenen Ports der jeweiligen Pipe sendet. Die Control Plane muss den Generator aktivieren und konfigurieren. Für die Paketgenerierung kann der Paketgenerator verschiedene Trigger verwenden, die z.B. einmalig, oder periodisch Pakete senden. Der Paketgenerator stellt einem generierten Paket einen `Timer-Header` voran, der das Paket identifiziert und der vom Parser extrahiert werden muss.

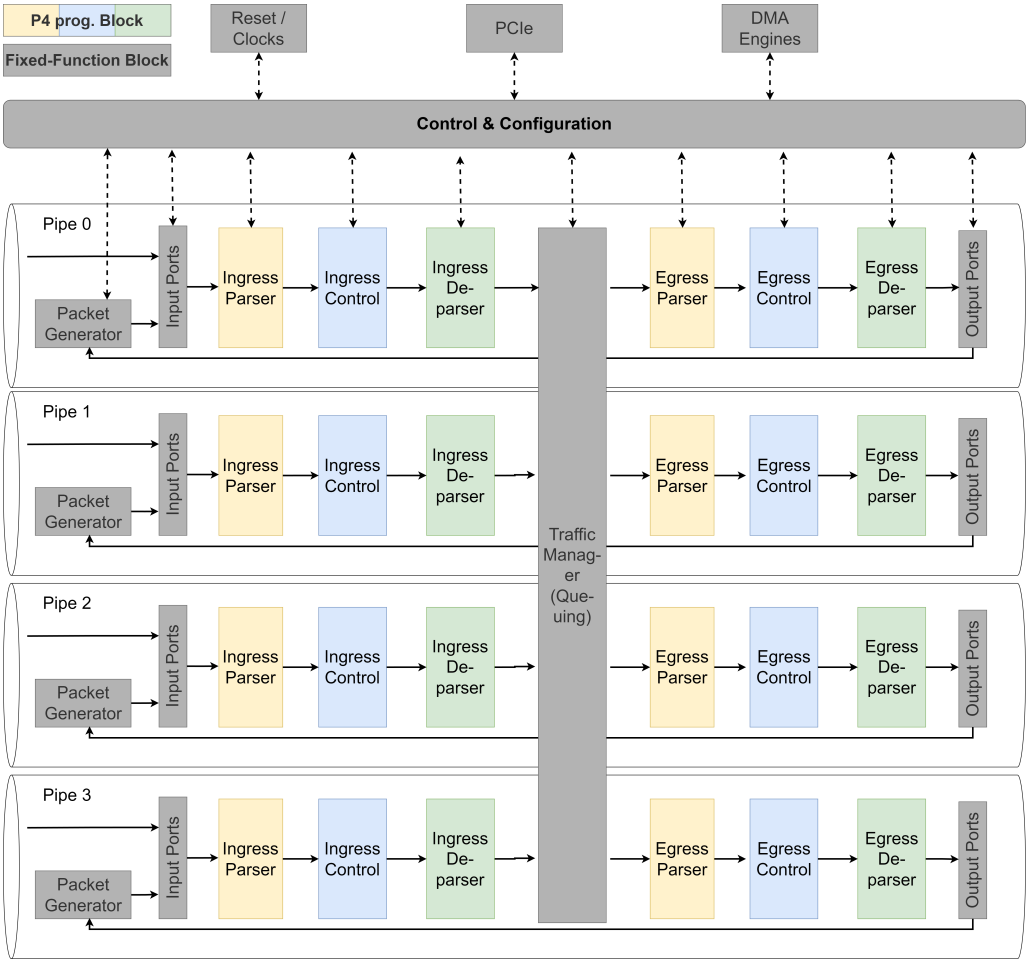


Abbildung 2.5.: Pipeline der TNA [15] mit vier Pipes.

Des Weiteren führt die TNA neben `exact`, `ternary` und `LPM` eine weitere Möglichkeit zur Prüfung auf Übereinstimmung in MATs ein: `range`. Der Typ `range` als Schlüsselfeld ordnet Header- und Metadatenfelder einem Intervall zu. Die Intervalle müssen von der Control Plane spezifiziert werden und die Zuordnung gilt inklusiv für beide Intervallgrenzen. Neben den in der PSA definierten Metadaten, bietet die TNA zusätzliche intrinsische Metadaten. Diese sind als eigene `struct`-Typen refe-

renzierbar und werden vom Target jeweils nach Abschluss des Parsers, des Deparsers und des In- bzw. Egress-Kontrollblocks generiert. Sie beinhalten zusätzliche Flags, wie beispielsweise die Zuweisung zu einer Traffic-Queue.

Die TNA implementiert, aufbauend auf der PSA, einige weitere *Externs*, wie z.B. *Digests*. Digests sind Nachrichten, die von der Data Plane mit Informationen angereichert und an die Control Plane gesendet werden. Die Generierung erfolgt durch Setzen des `ig_dprsr_md.digest_type` Feldes auf einen Wert zwischen null und sieben in den intrinsischen Metadaten nach Abschluss des Ingress-Kontrollblocks. Deshalb können ausschließlich Ingress-Kontrollblöcke Digests generieren. Da das Feld drei Bit groß ist, kann ein P4-Programm insgesamt acht verschiedene Digests mit unterschiedlichen Informationen generieren. Der Deparser-Kontrollblock überprüft den Wert des Feldes und weist dem Digest durch Aufruf der Funktion `pack()` die entsprechenden Informationen zu. Im Anschluss führt er den restlichen Code aus und sendet den Digest zusätzlich zum eigentlichen Paket.

2.3. Time-Sensitive Networking (TSN)

Time-Sensitive Networking (TSN) ist eine Sammlung von Standards, die Mechanismen für die Datenübertragung auf der Basis von Best-Effort Services in Ethernet-Netzwerken definieren. Dabei festigt TSN die vier Säulen der Netzwerkarchitektur: Zeitsynchronisierung, begrenzte, niedrige Latenz, hohe Zuverlässigkeit und Ressourcenmanagement [11]. In diesem Abschnitt werden die für diese Arbeit relevanten Bereiche von TSN skizziert.

Eine *Best-Effort* Übertragung betrachtet jedes Frame als gleichwertig und leitet es unter Berücksichtigung der noch zur Verfügung stehenden Ressourcen schnellstmöglich weiter. Überschreitet ein Frame die Kapazität des Netzes, kann die Übertragung nicht mehr gewährleistet werden. So kann es bei Überlastung leicht zu Paketverlust kommen, der dann von einer der höheren Schichten des OSI-Modells ausgeglichen werden muss. Die Standards 802.1D [1] und 802.1Q [2] adressieren dieses Problem zwar, indem sie Ethernet um Prioritätsklassen und eine logische Trennung mittels VLANs erweitern, fügen aber insbesondere bei der Verwendung mehrerer, gleichzeitiger Flows eine zusätzliche Latenz hinzu [19].

Best-Effort Services und insbesondere TSN setzen ein per Ethernet kommunizierendes Netzwerk voraus. In der Vergangenheit war Ethernet für viele Anwendungen aufgrund der damals vergleichsweise hohen Kosten und der mangelnden Zuverlässigkeit nicht geeignet, weshalb dedizierte Systeme entwickelt wurden. Heute sind die Kosten wesentlich geringer und es gibt Mechanismen, wie TSN sie bietet, die Zuverlässigkeit in solchen Systemen ermöglichen [19].

TSN bietet gleich mehrere Versprechen bezüglich der Zustellgarantie in Ethernet-basierten Netzwerken. Um diese zu erreichen schließt TSN einen Vertrag zwischen jedem TSN-Teilnehmer und dem Netzwerk. Der Teilnehmer verpflichtet sich dabei, nur eine bestimmte Datenmenge in einem bestimmten Zeitintervall zu versenden,

während das Netzwerk im Gegenzug die Bandbreite inklusive Buffering und Scheduling explizit für diesen Teilnehmer reserviert [10]. Wird ein Netzwerk für alle zeitkritischen Teilnehmer auf diese Weise geplant, ist sichergestellt, dass kein Paketverlust aufgrund von Stau durch Netzwerküberlastung auftreten wird [19]. Darüber hinaus existieren TSN-Standards, die eine deterministische Aussage über eine garantierte Obergrenze der Latenz ermöglichen. Durch weitere Mechanismen, die im Standard *802.1CB „Frame Replication and Elimination for Reliability“* [5] definiert sind, macht TSN zudem ein Versprechen hinsichtlich eines geringen Paketverlusts bei Hardwareausfällen durch redundante Datenpfade.

Außerdem bietet TSN eine *Zeitsynchronisierung* aller Netzwerkteilnehmer mit einer Genauigkeit zwischen $1\mu s$ und $10ns$ [10]. Dies ist eine Voraussetzung für viele, auf TSN basierende Traffic-Shaping-Algorithmen, bei denen Zeitkritikalität eine Rolle spielt. Die hohe Genauigkeit erreicht TSN durch das Precision Time Protocol (PTP), welches im Standard *IEEE 1588* [6] definiert ist. Mithilfe des in Standard *802.1Qat* [3] definierten „*reservation protocols*“ ist es zudem möglich, neue TSN-Verträge dynamisch zur Laufzeit zu erzeugen, was eine hohe Flexibilität ermöglicht. Netzwerkteilnehmer, die keinen zeitkritischen Restriktionen unterliegen, können zusammen mit TSN-Teilnehmern in einem Netzwerk *koexistieren*. Für sie steht die Bandbreite zur Verfügung, die TSN keinem TSN-Host fest zugewiesen hat. Das Netzwerk leitet solche Pakete nach dem Best-Effort-Prinzip weiter.

Anwendungsfälle für TSN sind z.B. Echtzeit-Audio/Video-Streams, Kommunikation zwischen Maschinen in der Industrie, Mobilfunk, oder Kommunikation im Automobil. Als konkretes Beispiel für die Kommunikation im Automobil kann ein automatischer Notbremsassistent genannt werden. Hier ist ein Paketverlust eines Signals, das ein Hindernis anzeigt und eine sofortige Notbremsung auslösen soll, fatal. Insbesondere dann, wenn der Paketverlust durch die Überlastung einer weniger wichtigen Funktionalität verursacht wird [10].

2.4. Per-Stream Filtering and Policing (PSFP)

Die im vorigen Abschnitt beschriebenen Garantien von TSN funktionieren nur, solange sich alle Teilnehmer an ihren Vertrag halten und nicht mehr Daten als vereinbart, oder außerhalb ihres vereinbarten Zeitintervalls, senden. Bricht ein Host seinen Vertrag, sei es durch Fehlkonfiguration oder durch einen absichtlichen *Denial of Service* Angriff [8], und beansprucht dadurch mehr Bandbreite als ihm zusteht, kann das Netzwerk seinen Vertrag mit den anderen Teilnehmern nicht mehr erfüllen. Die Garantien gegenüber allen anderen Netzteilnehmern sind nicht mehr gewährleistet, da die eingeplanten Ressourcen nicht mehr zur Verfügung stehen.

Per-Stream Filtering and Policing (PSFP) adressiert dieses Problem, indem es fehlverhaltende Hosts oder Streams von der Netzwerkkommunikation ausschließt.

2.4.1. Aufbau

Dieser Abschnitt zeigt zunächst die Struktur von PSFP auf, die einen Mechanismus zur Paketverarbeitung innerhalb eines Switches beschreibt und erläutert anschließend die einzelnen Komponenten Stream-Filter, Stream-Gate und Flow-Meter, die für einen korrekten Betrieb notwendig sind. Abschließend werden die verschiedenen Counter-Instanzen erklärt, die die Funktionalität von PSFP protokollieren.

Abbildung 2.6 zeigt den Forwarding-Prozess, den jedes eingehende Paket in einer Bridge (= Switch) nach IEEE 802.1Qci [4] durchläuft. Dabei passiert das Paket mehrere Kontrollblöcke (Ingress und Egress), in denen die Bridge anhand verschiedener Kriterien Paketfelder verändert, analysiert oder hinzufügt. Am Ende kann der Switch auf dieser Grundlage eine Entscheidung über die Zuweisung zu einer Prioritätsqueue, sowie über die Weiterleitung treffen.

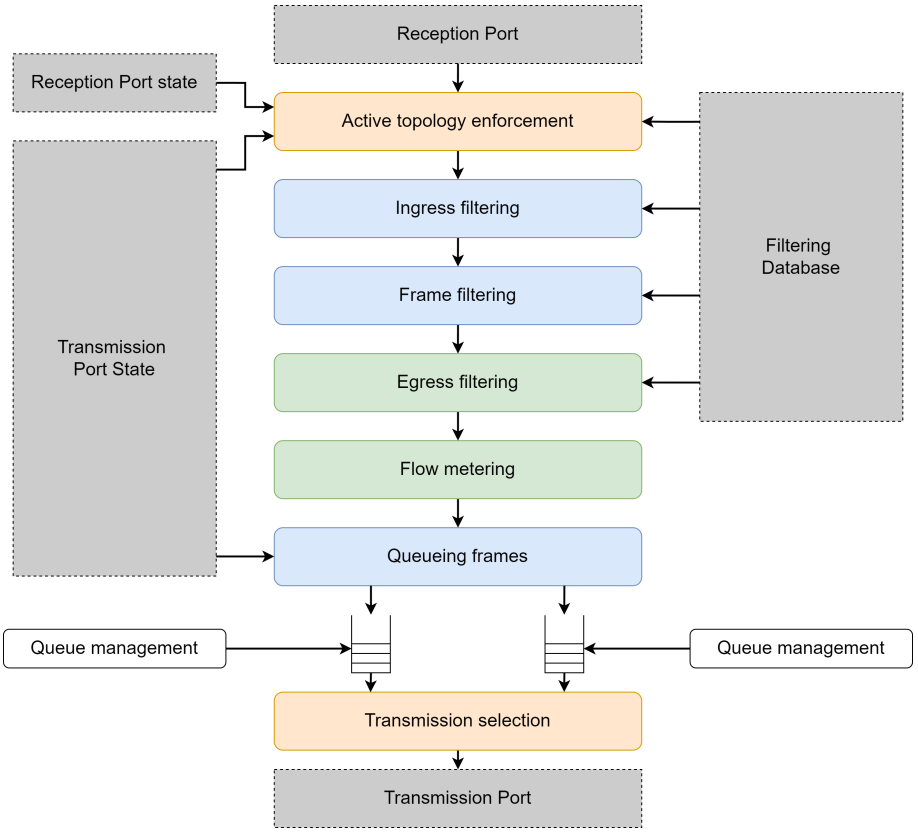


Abbildung 2.6.: Forwarding-Prozess nach 802.1Qci [4].

Gemäß *802.1Qci* befindet sich der PSFP-Mechanismus im *Egress Filtering* und *Flow Metering* einer Bridge, wie in Abbildung 2.7 dargestellt. Die Abbildung zeigt die drei wesentlichen Komponenten von PSFP: *Stream-Filter*, *Stream-Gates* und *Flow-Meter*, die im Folgenden erläutert werden. Zu erkennen ist auch eine *Gate Control List*, die als Datenquelle für die Filterung von Streams und Paketen dient, jedoch von einem externen Programm (siehe 2.1.1 *Control Plane*) mit Inhalt versorgt

wird. Es existieren mehrere Stream-Gate, Stream-Filter und Flow-Meter Instanzen innerhalb einer Bridge. PSFP ordnet jedem Stream-Filter genau ein Stream-Gate und eine oder mehrere Flow-Meter Instanzen zu. Mehrere Stream-Filter können zudem dieselbe Stream-Gate Instanz verwenden.

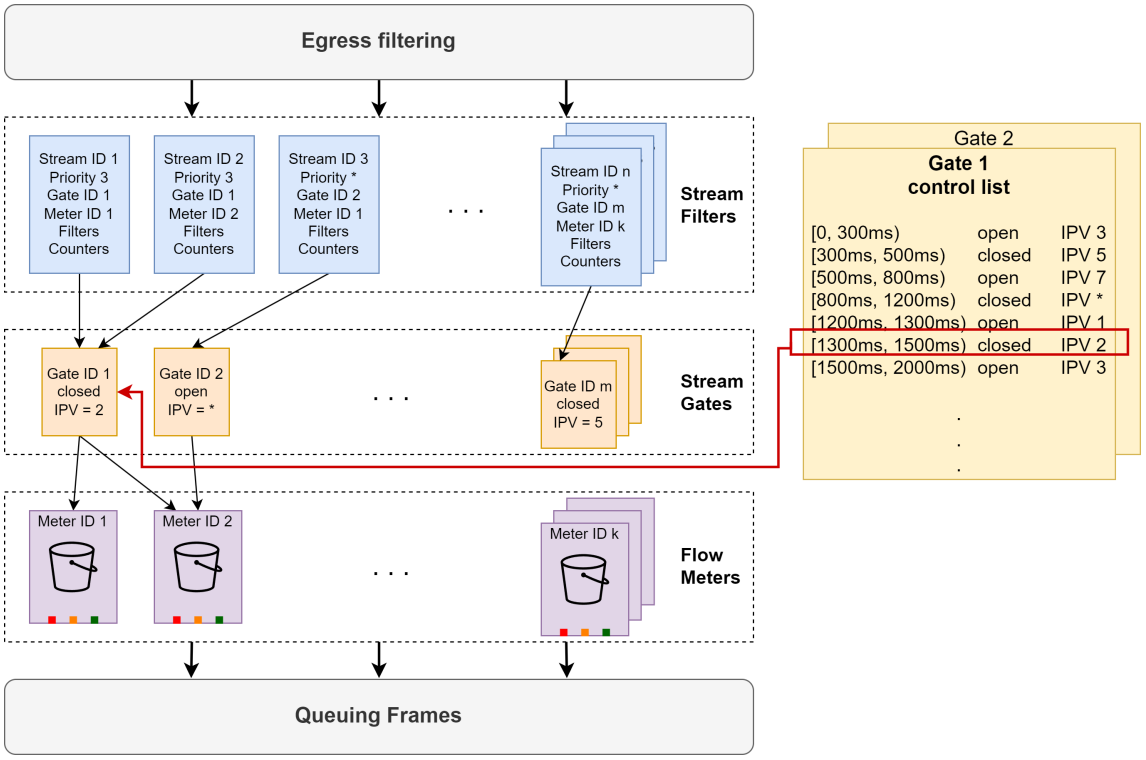


Abbildung 2.7.: PSFP nach 802.1Qci [4].

2.4.1.1. Stream-Filter

Ein *Stream-Filter* hat im Wesentlichen die Aufgabe, einen Stream auf Basis mehrerer Kriterien zu identifizieren und ihn anhand dessen einem bestimmten Stream-Gate und einem oder mehreren Flow-Metern zuzuordnen [4].

Die Stream-Filter Instanz organisiert mehrere Filterregeln in einer geordneten Liste, die sie auf jedes Frame anwendet. Trifft eine Regel zu, weist sie dem Frame den Parameter *stream_handle* zu, der von nun an das Frame oder den Stream identifiziert. Wenn nach Anwendung der Regeln keine Übereinstimmung gefunden wurde, weist sie dem Frame kein *stream_handle* zu und der Switch behandelt es wie ein normales Paket ohne PSFP-Mechanismus. Er leitet es nach dem Best-Effort-Prinzip weiter.

Wenn das Frame den *stream_handle* Parameter erhalten hat, ordnet die Stream-Filter Instanz es einem Stream-Gate und einer oder mehreren Flow-Meter Instanzen zu und leitet es anschließend zur Stream-Gate Instanz weiter.

Der Standard 802.1CB [5] definiert mehrere Funktionen mit verschiedenen Kriterien zur Identifizierung von Streams, die in Tabelle 2.1 dargestellt sind. Eine *passive* Identifizierung überprüft nur auf übereinstimmende Header-Felder, während eine *aktive* Identifizierung zusätzlich weitere Header-Felder des Frames überschreibt. Die Funktionen können zudem auch miteinander kombiniert werden, um feingranularere Filter zu definieren.

Stream-identifizierungs-funktion	**Aktiv / Passiv**	**Untersuchte Header Felder**	**Überschreibt**
Null stream identification	Passiv	destination_address, vlan_identifier	-
Source MAC and VLAN stream identification	Passiv	source_address, vlan_identifier	-
Active Destination MAC and VLAN stream identification	Aktiv	destination_address, vlan_identifier	destination_address, vlan_identifier, priority
IP stream identification	Passiv	destination_address, vlan_identifier, IP source address, IP dest. address, DSCP, IP next protocol, source port, dest. port	-

Tabelle 2.1.: Stream identification functions nach 802.1CB [5].

Zusätzlich filtert ein Stream-Filter auf die maximale Größe der *Service Data Unit (SDU)*. Die SDU repräsentiert die noch nicht eingekapselten Nutzdaten, abhängig von der betrachteten Schicht des OSI-Modells. Da TSN und PSFP Ethernet-Netze betrachten, entspricht die SDU der Anzahl der Bytes ab OSI-Schicht 3. Überschreitet ein Frame diese Grenze, verwirft die Bridge das Frame. Optional kann ein Stream-Filter mit dem Parameter *StreamBlockedDueToOversizeFrame* auch so konfiguriert werden, dass ein Stream permanent blockiert wird, sobald er die Schwelle zum ersten Mal überschreitet.

2.4.1.2. Stream-Gate

Ein *Stream-Gate* ist ein periodisch zeitgesteuertes Gate, das einerseits Streams kontrolliert, indem es Frames nur innerhalb definierter Zeitintervalle passieren lässt und andererseits die Priorität und damit die Queue eines Frames verändern kann [4].

Die *Stream-Gate* Instanz prüft ein vom Stream-Filter erhaltenes Frame zunächst anhand eines Zeitstempels gegen einen periodisch geplanten Zeitplan. Dieser Zeitplan enthält mehrere Intervalle und speichert je einen Zustand (offen / geschlossen) und einen optionalen *Internal Priority Value (IPV)* pro Intervall. Fällt ein Frame in ein Intervall im geschlossenen Zustand, verwirft die Stream-Gate Instanz das Frame, bzw. leitet es im offenen Zustand an die Flow-Meter Instanz(en) weiter. Der optionale IPV legt die Prioritätsqueue des Frames fest und hat nur innerhalb der Bridge eine Gültigkeit. Er wird nicht in Form eines Header-Feldes dem Paket hinzugefügt. Sollte der IPV nicht gesetzt sein, wählt der Switch die Queue anhand des sich im 802.1Q-Header befindenden *Priority Code Point (PCP)*-Feldes aus [4]. Besonders wichtig für die korrekte Funktion der Stream-Gates ist eine Zeitsynchronität mit sehr hoher Präzision, wie sie TSN mithilfe von PTP bietet [10], damit die Zuordnung zu den Zeitintervallen exakt erfolgen kann. Dies ist insbesondere wichtig, wenn mehrere PSFP Bridges auf dem Pfad liegen, da die Bridges die Zeitpläne synchron zur Netzwerkzeit halten müssen.

Wichtig bei der Übertragung ist, dass die Bridge in der Lage sein muss, ein Frame vollständig zu senden, auch wenn ein Gate während der Übertragung in einen geschlossenen Zustand wechselt. Dafür werden sogenannte *Guard Bands* eingesetzt, die die Öffnungsphase um die Dauer der Übertragung eines maximal großen Frames verkürzen und während der die Bridge kein neues Frame durch das Stream-Gate senden darf [19]. Dieses Verhalten kann auch implizit von der Control Plane durch entsprechende Einplanung in die Intervalle erreicht werden.

Darüber hinaus besitzt ein Stream-Gate die zusätzlichen Parameter *GateClosedDueToInvalidRX* und *GateClosedDueToOctectsExceeded*, mit denen PSFP Fehlerzustände in der Kommunikation, bzw. eine Verletzung der TSN-Verträge, erkennen kann [19]. Der Parameter `GateClosedDueToInvalidRX` schließt dabei Teilnehmer aus, die ein Frame senden während sich das Gate im geschlossenen Zustand befindet. Der Parameter `GateClosedDueToOctectsExceeded` schließt Teilnehmer aus, die innerhalb des aktuellen Zeitintervalls bereits zu viele Daten gesendet haben. Beide Funktionen sind nützlich, um die Einhaltung von TSN-Verträgen für andere Teilnehmer gemäß Abschnitt 2.3 zu gewährleisten, in dem ein fehlverhaltender, oder fehlkonfigurierter Host permanent blockiert wird [4].

2.4.1.3. Flow-Meter

Nachdem ein Frame mithilfe der Stream-Filter Instanz einem Stream zugeordnet wurde und das Stream-Gate den erlaubten Sendezeitraum verifiziert hat, überprüfen nun die *Flow-Meter* Instanzen die Einhaltung der im TSN-Vertrag spezifizierten Bandbreite [4].

Eine Flow-Meter Instanz implementiert einen *Token Bucket Policer* nach RFC2698 [13]. Das Flow-Meter bezahlt dabei für ein Frame, abhängig von seiner Größe, mit Token aus einem Bucket. Sind nicht mehr genügend Token im Bucket vorhanden, kann das Flow-Meter das Frame nicht bezahlen und verwirft es. Der Policer generiert neue Token mit einer Rate, der *Committed Information Rate (CIR)*, die der im TSN-Vertrag festgelegten Bandbreite entspricht. Analog zur CIR ist die *Excess Information Rate (EIR)* die Rate, mit der Token vom Policer für Frames bezahlt werden, die die erlaubte Bandbreite zwar überschreiten, aufgrund von vorhandener Kapazität aber noch übertragen werden können. Die Parameter *Committed Burst Size (CBS)* bzw. *Excess Burst Size (EBS)* legen die maximale Kapazität an Token in einem Bucket fest [19]. Die *Peak Information Rate (PIR)* beschreibt die Summe aus CIR und EIR und analog dazu beschreibt die *Peak Burst Size (PBS)* die Summe aus CBS und EBS.

Abhängig davon aus welchem Bucket das Flow-Meter ein Frame bezahlt, färbt der Token Bucket Algorithmus das Frame in einer von drei Farben ein. Ein Frame, das Token der CIR konsumiert, wird *grün* markiert und somit weitergeleitet. Kann ein Frame nicht mehr mit Token der CIR, jedoch der EIR bezahlt werden, so wird es *gelb* markiert. Zusätzlich setzt die Bridge das *DropEligibleIndicator (DEI)*-Flag im 802.1Q-Header des Ethernet Frames, das nachfolgenden Bridges indiziert, dass dieses Frame die Bandbreite auf dem bisherigen Pfad bereits überschritten hat. Nachfolgende Bridges können dann, je nach Konfiguration, das Frame direkt verwerfen, oder neu färben. Kann ein Frame weder mit CIR- noch mit EIR-Token bezahlt werden, markiert der Policer es *rot* und die Flow-Meter Instanz verwirft es anschließend.

In PSFP erhält eine Flow-Meter Instanz zusätzlich die Parameter *DropOnYellow*, *MarkAllFramesRed* und *ColorMode*, der die Werte *color-blind* und *color-aware* annehmen kann. Der ColorMode gibt an, ob eine Bridge die bereits gesetzte Farbe eines Frames respektieren soll. Wurde ein Paket durch Setzen des DEI-Flags im 802.1Q-Header auf dem bisherigen Pfad durch das Netzwerk bereits gelb markiert, so kann das Flow-Meter es im *color-blind* Modus wieder grün markieren, wenn es innerhalb der erlaubten Bandbreite an der aktuellen Bridge liegt. Im *color-aware* Modus kann das Flow-Meter ein zuvor gelb markiertes Frame nicht mehr grün färben. Da das PSFP Flow-Meter grün und rot markierte Frames nicht durch zusätzliche Header-Felder kennzeichnet, trifft der ColorMode nur auf gelb markierte Frames zu. In Kombination mit dem `DropOnYellow`-Flag kann eine Bridge so bereits gelb markierte Frames später verwerfen.

Ist das Flag `MarkAllFramesRed` gesetzt, markiert das Flow-Meter alle nachfolgenden Pakete rot, sobald ein Frame die PIR überschreitet und somit rot markiert wird [4]. Wie beim Stream-Gate und Stream-Filter können so Teilnehmer, die gegen die im TSN-Vertrag vereinbarten Parameter verstoßen, dauerhaft ausgeschlossen werden.

2.4.1.4. Counter

Um die Funktionalität und Metriken für die Nutzung im PSFP-Netzwerk zu erfassen, definiert eine Stream-Filter Instanz zusätzliche *Counter*-Instanzen. Die verschiedenen Counter und ihre Bedeutung sind in Tabelle 2.2 aufgelistet. Sie messen jeweils die Anzahl der Frames, die die Stream-Filter, Stream-Gate und Flow-Meter Instanzen passieren, bzw. verworfen werden.

Counter	**Bedeutung**
MatchingFramesCount	#Frames, die mit dieser Stream-Filter Instanz identifiziert wurden.
PassingFramesCount	#Frames, die innerhalb eines offenen Zeitintervalls des Stream-Gates lagen.
NotPassingFramesCount	#Frames, die innerhalb eines geschlossenen Zeitintervalls des StreamGates lagen.
PassingSDUCount	#Frames, die die maximale SDU nicht überschritten haben.
NotPassingSDUCount	#Frames, die die maximale SDU überschritten haben.
RedFramesCount	#Frames, die vom FlowMeter rot markiert wurden.

Tabelle 2.2.: *Counter*-Instanzen und deren Bedeutungen.

3. Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten im Bereich von PSFP und SDN vorgestellt. Dies beinhaltet eine Simulation von PSFP im OMNeT++ Framework, sowie eine Implementierung des PTP Protokolls in der Data Plane zur präzisen Zeitsynchronisierung.

3.1. PSFP Simulation im Simulationsframework OMNeT++

In einer Bachelorarbeit [14] des Lehrstuhls für Kommunikationsnetze an der Eberhard Karls Universität Tübingen wurde der Nutzen des PSFP-Mechanismus bereits im Jahr 2021 von Benedikt Hopf in einer simulierten Umgebung mithilfe des OMNeT++ Frameworks aufgezeigt. OMNeT++ ermöglicht unter Verwendung der Network Description Language (NED) die Konfiguration, Simulation und Evaluierung von Netzwerken [22]. Durch die Erweiterungen INET und NeSTiNg [9] werden weitere Netzwerkprotokolle, insbesondere TSN, unterstützt. Eigene Funktionalitäten zur Manipulation von Paketen können darin in der Sprache C++ definiert werden.

In der Arbeit wurde der PSFP-Mechanismus, sowie Cyclic Queueing and Forwarding (CQF) als möglicher Anwendungsfall von PSFP, in einem simulierten Netzwerk implementiert, das einen fehlerhaft konfigurierten Host enthält. Das Netzwerk wird durch diesen gestört, indem der Host gegen die TSN-Verträge verstößt. Das Ergebnis der Arbeit war, dass der PSFP-Mechanismus die Latenz und den Jitter der zeitsensitiven Frames signifikant reduzieren konnte, obwohl die Qualität der Übertragung unter der Überlastung durch den fehlerhaft konfigurierten Host vor Einbindung von PSFP litt [14].

3.2. Data Plane Precision Time Protocol

Kannan et al. haben in ihrer Arbeit [16] einen PTP-Mechanismus in der Data Plane implementiert, um die Zeitsynchronisierung von Netzwerkgeräten weiter zu präzisieren. Bisherige Implementierungen setzten das PTP Protokoll in der Control Plane um, was die Verarbeitung verlangsamt. Das *Data Plane Precision Time Protocol (DPTP)* hingegen speichert die Zeit und beantwortet DPTP-Anfragen vollständig in der Data Plane, um bei Datenraten von bis zu 100 Gbps eine Genauigkeit im Nanosekundenbereich zu unterstützen.

DPTP wurde im Rahmen der Arbeit von Kannan et al. in der Programmiersprache P4 auf einem Intel® Tofino Switch-ASIC implementiert. Die Implementierung unterstützt die Zeitsynchronisierung zwischen Switches, sowie zwischen Switches und

Hosts. Das P4-Programm spezifiziert eigene DPTP-Header, die hardwareunterstützte Zeitstempel verschiedener Zeitpunkte in der Paketverarbeitung innerhalb eines Switches enthalten. Aufgrund der Implementierung in der Data Plane sind diese hochpräzise. Aus diesen Zeitstempeln berechnet der Switch die Referenzzeit und synchronisiert die angeschlossenen Hosts, indem er auf DPTP-Anfragen antwortet.

Eine hochpräzise Zeitsynchronisierung ist eine wesentliche Voraussetzung für die korrekte Funktionsweise verschiedener TSN-Mechanismen. Die Arbeit von Kannan et al. bietet eine Möglichkeit, die für TSN und insbesondere PSFP notwendige Zeitsynchronisierung auf der Basis eines Intel® Tofino Switch-ASIC in P4, wie er auch in dieser Arbeit verwendet wird, bereitzustellen.

Der Beispielcode der Implementierung [17] enthält den Quellcode des Programms der Data Plane in P4, den C++-Code der implementierten Control Plane, sowie Anweisungen zum Einrichten der Umgebung, einschließlich der zu synchronisierenden Hosts. Beim Start der Implementierung muss einer der Switches als Master definiert werden. Die anderen Switches kommunizieren im Anschluss mit dem Master-Switch, um die globale Zeit zu erhalten. Sobald alle Switches miteinander synchronisiert sind, können diese Zeitsynchronisierungsanfragen von Hosts annehmen und beantworten. Unter großer Netzwerkauslastung erreicht die DPTP-Implementierung einen Synchronisierungsfehler von $\sim 50ns$ zwischen Switches und Hosts, die bis zu 6 Hops voneinander entfernt sind.

4. Implementierung

Im folgenden Kapitel wird die Implementierung von Per-Stream Filtering and Policing (PSFP) zur Einhaltung und Überwachung von TSN-Verträgen auf einem 100G Intel® Tofino Switch-ASIC in der P4-Programmiersprache beschrieben. Dabei werden zunächst die allgemeinen Strukturen und Konzepte der Implementierung aufgezeigt und im Anschluss die einzelnen PSFP-Komponenten hinsichtlich ihrer Funktionalität und P4-spezifischen Probleme bzw. deren Lösungen erläutert. Abschließend wird die Funktionalität der implementierten Control Plane erläutert. Um dieses Kapitel verständlich zu halten, wird auf eine detaillierte Codeanalyse verzichtet.

4.1. Besonderheiten und Überblick

Dieser Abschnitt beschreibt im Allgemeinen die Implementierung von PSFP in der Data Plane, die die Packet-Forwarding-Logik definiert. Als Erstes werden einige P4- bzw. Intel® Tofino-spezifische Besonderheiten der Implementierung erläutert. Im Anschluss wird der Kontrollfluss innerhalb eines mit PSFP-programmierten Switches auf abstrakter Ebene dargestellt.

Eine Besonderheit der Tofino Native Architecture (TNA), die in Abbildung 2.5 veranschaulicht wird, besteht darin, dass das Queueing von Frames bereits nach dem Ingress-Block erfolgt. Da PSFP die Priorität und damit die Queue-Zuordnung eines Frames durch den IPV verändert, muss der PSFP-Mechanismus folglich im *Ingress*-Block des Switches implementiert werden. Im Kontrast dazu sind bestimmte intrinsische Metadaten in der TNA, die PSFP benötigt, nur im Egress-Block verfügbar. Aus diesem Grund müssen Frames eine *Recirculation* durchlaufen. Bei einer Recirculation setzt der Switch den Ausgangsport für ein Frame auf einen seiner eigenen Eingangsports, der im Loopback-Modus konfiguriert ist. Ein Frame, dessen Ausgangsport auf einen Recirculation-Port gesetzt ist, durchläuft die gesamte TNA-Pipeline und wird anschließend wieder als neues, eingehendes Frame im Ingress behandelt. In diesem Prozess kann der Switch zusätzliche Informationen, verpackt in `Bridge`-Header, an das Paket anhängen, die der Ingress-Parser auslesen kann. In den Abschnitten 4.2.2 und 4.2.3.2 wird die Notwendigkeit der Recirculation näher erläutert.

Abbildung 4.1 zeigt den abstrakten Kontrollfluss, den ein eingehendes Frame in einem Switch der PSFP-Implementierung durchläuft. Wenn ein Frame keinen `Ethernet 802.1Q`-Header besitzt oder keinem generierten Paket entspricht, das durch einen `Timer`-Header identifiziert wird, behandelt der Ingress-Kontrollblock das Frame als TSN-unaware Verkehr und leitet es per Best-Effort Übertragung weiter.

Andernfalls unterscheidet er anhand des Eingangsports zwischen einem generierten Paket, wenn es an dem für den Paketgenerator konfigurierten Port eingeht, und einem Frame auf das PSFP angewendet wird, wenn es an einem anderen Port eintrifft. Abhängig davon wird der Zeitstempel des Frames in ein Register geschrieben oder aus einem Register gelesen (siehe Abschnitt 4.2.3.1 *Periodizität*). Handelt es sich um ein generiertes Paket, wird es anschließend zum Verwerfen markiert. Sichtet der Switch ein Frame, auf das PSFP angewendet werden soll, zum ersten Mal, berechnet er anhand des ausgelesenen Registerwertes die aktuelle Position im Zeitplan des Stream-Gates und sendet das Frame per Recirculation wieder zurück an den Anfang. Da das Frame nun auf dem für die Recirculation vorgesehenen Port eintrifft, erkennt der Switch die zweite Sichtung des Frames und wendet folgend die drei PSFP-Komponenten Stream-Filter, Stream-Gate und Flow-Meter auf das Frame an. In jeder dieser Komponenten besteht die Möglichkeit, das Frame zu verwerfen. Passiert es jedoch alle Komponenten, leitet der Switch es per IPv4 weiter. Kontrollstrukturen der Periodenberechnung sind in Abbildung 4.1 violett, PSFP-Komponenten blau und Abläufe der Recirculation gelb dargestellt.

Es ist zu beachten, dass PSFP für beliebige Layer-3-Protokolle verwendet werden kann. Der Einfachheit halber wird in der hier beschriebenen Implementierung IPv4 als Layer-3-Protokoll verwendet, dieses kann jedoch beliebig ausgetauscht werden.

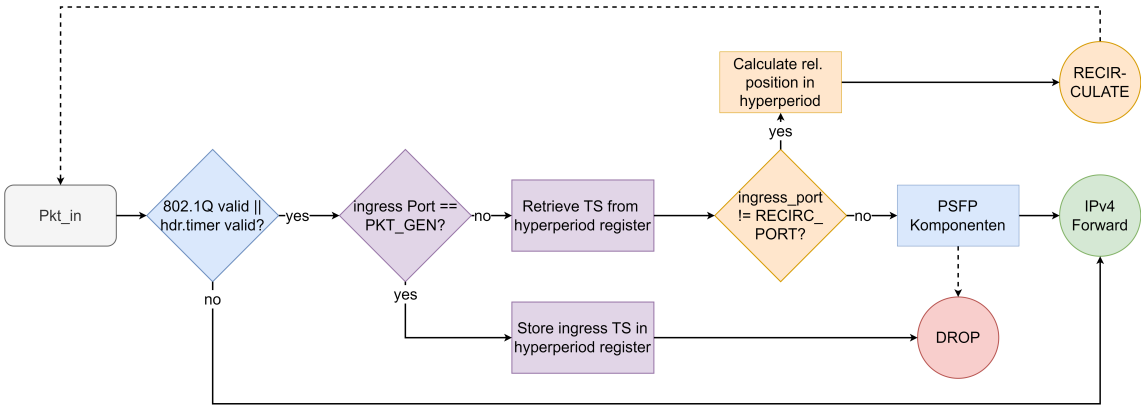


Abbildung 4.1.: Kontrollflussgraph des Ingress-Kontrollblocks.

4.2. Komponenten

Dieser Abschnitt beschreibt die Implementierung der für PSFP notwendigen Komponenten im Detail. Zunächst wird der Ablauf des Ingress-Parsers erläutert, der die verschiedenen Paket-Header extrahiert. Im Anschluss wird der Kontrollfluss der einzelnen Komponenten, Stream-Filter, Stream-Gate und Flow-Meter auf detaillierter Ebene erläutert. Abschließend wird eine Lösung zur Synchronisierung mit der Netzwerkzeit vorgestellt und die Funktionalität der implementierten Control Plane gezeigt. Die Komponenten in den Abschnitten 4.2.2 bis 4.2.4 verwenden zur Darstellung des Kontrollflusses die in Abbildung 4.2 gezeigte Legende.

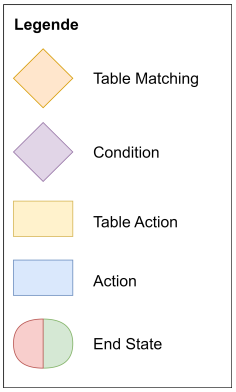


Abbildung 4.2.: Legende der Kontrollflussstrukturen der PSFP-Komponenten.

4.2.1. Implementierter Parser

Abbildung 4.3 zeigt den endlichen Automaten des implementierten Ingress-Parsers. Der Tofino-Parser fügt einem eingehenden Frame zunächst die intrinsischen Metadaten der TNA hinzu. Anschließend entscheidet der Parser anhand des eingehenden Ports über den nächsten Parserzustand. Für ein generiertes Paket extrahiert er den `Timer`-Header, dessen Signifikanz in Abschnitt 4.2.3.2 erläutert wird. Für ein Frame, das auf einem Recirculation-Port eintrifft, extrahiert der Parser den `Bridge`-Header und geht über in den Ethernet-Zustand. Trifft ein Frame auf einem der anderen Ports ein, muss es einen `Ethernet`-Header besitzen, gefolgt von einem `IPv4`-Header, oder einem optionalen `Ethernet 802.1Q`-Header. Andernfalls wird das Paket bereits im Parser verworfen. Zuletzt extrahiert der Parser die Header-Felder der Transportschicht und akzeptiert schließlich das Frame. Da der Egress-Kontrollblock in der PSFP-Implementierung keine Routing-Entscheidungen treffen muss, ist hier ein Parsen der intrinsischen Metadaten, oder zusätzlich der `Bridge`-Header, ausreichend.

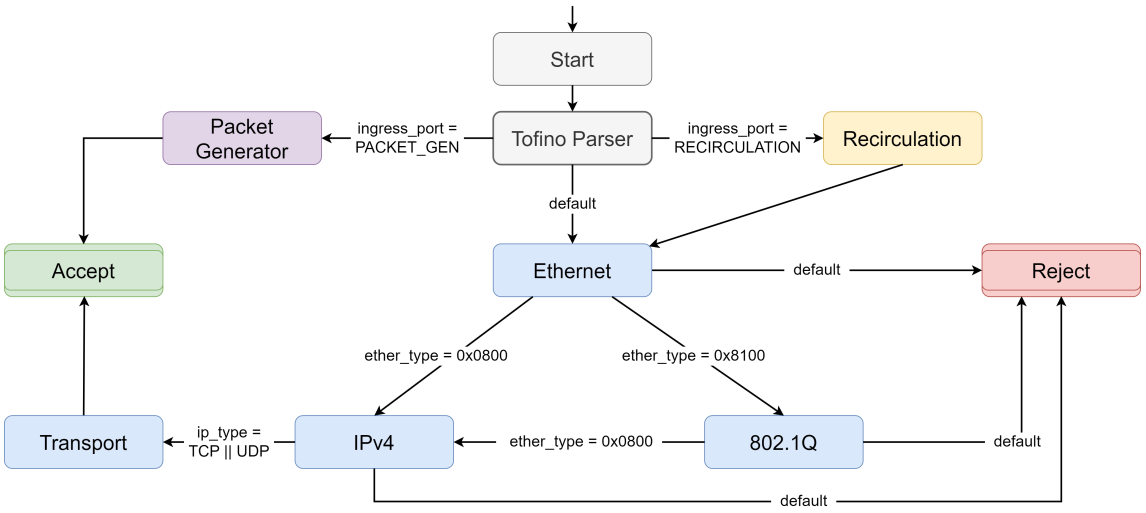


Abbildung 4.3.: Endlicher Automat des Ingress-Parsers.

4.2.2. Stream-Filter Implementierung

Die Stream-Filter Instanz ist als separater Kontrollblock im Ingress implementiert. Sie entspricht der ersten PSFP-Komponente, die ein Frame betritt, auf das PSFP angewendet werden soll, nachdem die Recirculation abgeschlossen ist. Der Kontrollfluss der Komponente ist in Abbildung 4.4 dargestellt. Im Folgenden wird der abstrahierte Ablauf innerhalb der Stream-Filter Instanz erläutert und die Notwendigkeit der Recirculation in Bezug auf die Komponente verdeutlicht.

Als erstes prüft der Stream-Filter anhand der `stream_id`-MAT, ob er das Frame einem von der Control Plane definierten Stream zuordnen kann. Die Tabelle enthält alle in Tabelle 2.1 aufgeführten Header-Felder als ternäres Schlüsselfeld. So können die Stream-Identifikationsfunktionen beliebig miteinander kombiniert werden, indem die Control Plane Header-Felder, die nicht relevant für die Identifizierung sind, in der MAT mit einer konstanten Nullmaske der ternären Übereinstimmung belegt. Diese Einträge entsprechen einer Wildcard. Findet ein Frame keine Übereinstimmung, führt der Switch kein PSFP für das Frame durch, sondern behandelt es als TSN-unaware Traffic. Andernfalls weist er ihm den `stream_handle`-Parameter und ein Flag zu, das bestimmt, ob die in der Tabelle 2.1 aktive Streamidentifizierung durchgeführt werden soll. Nachdem der `stream_handle` Parameter verfügbar ist, ordnet der Stream-Filter auf dessen Basis mithilfe einer weiteren MAT dem Frame eine Stream-Gate und eine Flow-Meter Instanz zu. Zusätzlich schreibt die Action die Werte der Parameter, z.B. `stream_blocked_due_to_oversize_enable`, in die Metadaten des Frames.

Im Anschluss führt der Stream-Filter die Prüfung der maximalen SDU durch. Wurde dieser Stream bereits durch ein zuvor empfangenes Frame aufgrund dessen Größe blockiert, wird das Frame zum Verwerfen markiert. In der TNA ist die voll-

ständige Paketgröße und damit die SDU erst in den intrinsischen Metadaten des Egress-Blocks verfügbar. Da PSFP jedoch aufgrund des Queueings im Ingress implementiert werden muss, muss ein Frame eine *Recirculation* durchlaufen. Vor der Recirculation hängt der Egress die volle Paketgröße in einem zusätzlichen `Bridge`-Header dem Frame an. Nach der Recirculation wird dieser Header extrahiert und kann im Ingress zur Überprüfung verwendet werden. Dies geschieht mit einer zusätzlichen MAT, die ein Schlüsselfeld des Typs `range` enthält. Die Tabelle prüft, ob die SDU des Frames im Intervall [0, max_SDU] liegt. Ein Frame, das nicht in das Intervall passt, wird im Anschluss zum Verwerfen markiert. Da das Paket um die Größe des `Bridge`-Headers erweitert wurde, addiert die Control Plane den Wert zur konfigurierten oberen Grenze der SDU, damit diese nicht in die Überprüfung einfließt. Ist der Parameter zum permanenten Blockieren des Streams gesetzt, generiert der Deparser eine Digest-Nachricht an die Control Plane. Bei Erhalt einer solchen Nachricht, ändert die Control Plane den korrespondierenden MAT-Eintrag ab, so dass der `stream_blocked_due_to_oversize`-Parameter gesetzt ist. Folglich markiert der Stream-Filter die durch diesen Filter identifizierten Frames zum Verwerfen.

Hat ein Frame den Max-SDU-Filter passiert, können optional einige von der Control Plane spezifizierte Header-Felder überschrieben werden. Schließlich wird das Frame an die Stream-Gate Instanz weitergeleitet. Entgegen dem Standard 802.1Qci [4], sind die Counter-Instanzen in der P4-Implementierung nicht zentral im Stream-Filter verankert, sondern auf die jeweils zutreffenden Komponenten verteilt.

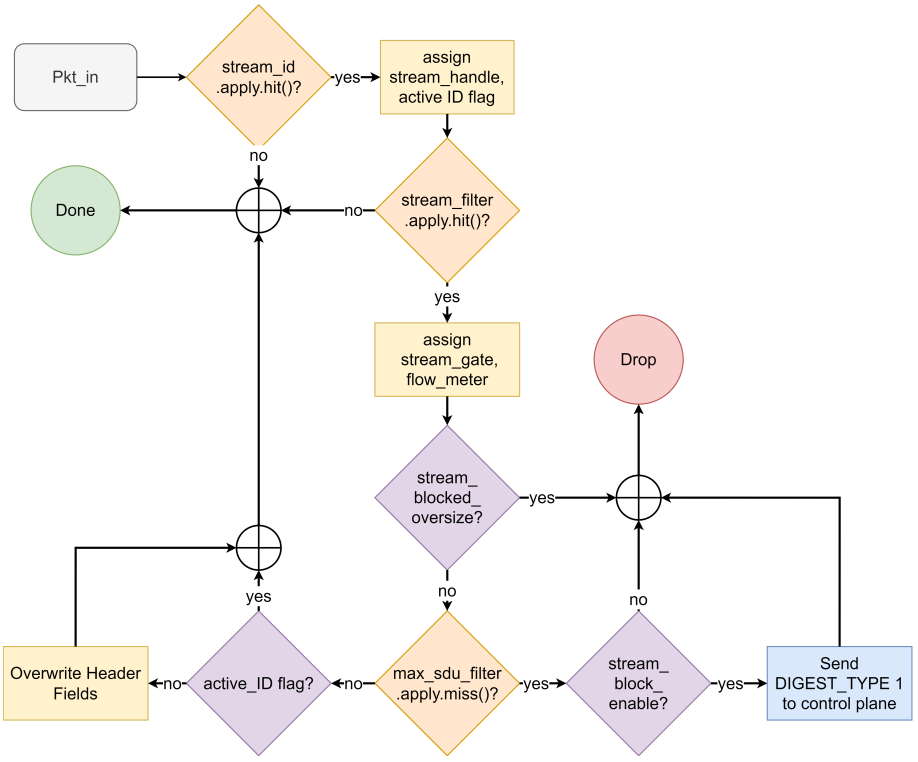


Abbildung 4.4.: Kontrollflussgraph des Stream-Filters.

4.2.3. Stream-Gate Implementierung

Nachdem der Stream-Filter ein Frame identifiziert und mithilfe des `stream_handle`-Parameters einer Stream-Gate Instanz zugeordnet hat, gelangt das Frame in den Kontrollblock des Stream-Gates. Der Ablauf ist in Abbildung 4.5 dargestellt und wird in diesem Abschnitt erläutert. Zusätzlich wird auf die Umsetzung der Periodizität mittels Hyperperioden, sowie auf ein Problem der TNA beim Intervallmatching eingegangen.

Zuerst lädt der Kontrollblock mit einer separaten MAT, die die ID des Stream-Gates als Schlüsselfeld enthält, alle Konfigurationsparameter des Stream-Gates in die Metadaten. Anschließend prüft die Instanz, analog zum Stream-Filter, ob das Gate bereits blockiert ist und markiert das Frame gegebenenfalls zum Verwerfen. Im Anschluss überprüft das Stream-Gate, ob das Frame in einem offenen Intervall angekommen ist. Es ist wichtig, dass diese Überprüfung nur für Frames durchgeführt wird, die noch nicht vom Stream-Filter zum Verwerfen markiert wurden. Andernfalls könnte ein bereits durch den Stream-Filter blockierter Stream ein permanentes Schließen des Gates für andere Teilnehmer auslösen. Der Zeitplan des Stream-Gates ist als MAT repräsentiert, die die Stream-Gate ID, sowie alle Zeitintervalle als `range`-matching Typ mit den korrespondierenden Zuständen und optionalen Internal Priority Values (IPVs) enthält. Ein eingehendes Frame wird anhand seines Ingress-Zeitstempels der ersten Sichtung, d.h. vor Durchlaufen der Recirculation, einem Intervall zugeordnet. Befindet sich der Zeitplan gerade in einem geschlossenen Zustand, markiert das Stream-Gate das Frame zum Verwerfen und sendet optional eine Digest-Nachricht, die die Control Plane anweist, die Stream-Gate Instanz permanent zu schließen. Da der IPV ein optionaler Parameter ist, wird ihm von der Control Plane der Wert „8“ zugewiesen, wenn stattdessen der PCP aus dem 802.1Q Header zur Queue-Zuordnung verwendet werden soll. Dieser Wert entspricht einer von acht Verkehrsklassen, wobei der Wert „7“ die höchste Priorität besitzt. Abhängig davon setzt der Kontrollblock die Queue-ID in den intrinsischen Metadaten des Traffic Managers auf den Wert des PCP. Die *GateClosedDueToOctectsExceeded*-Funkionalität lässt sich leider nicht ohne Weiteres in P4 implementieren, da keine Möglichkeit besteht, Oktette pro Zeitintervall zu zählen und auch entsprechend in einem neuen Intervall wieder rechtzeitig zurückzusetzen.

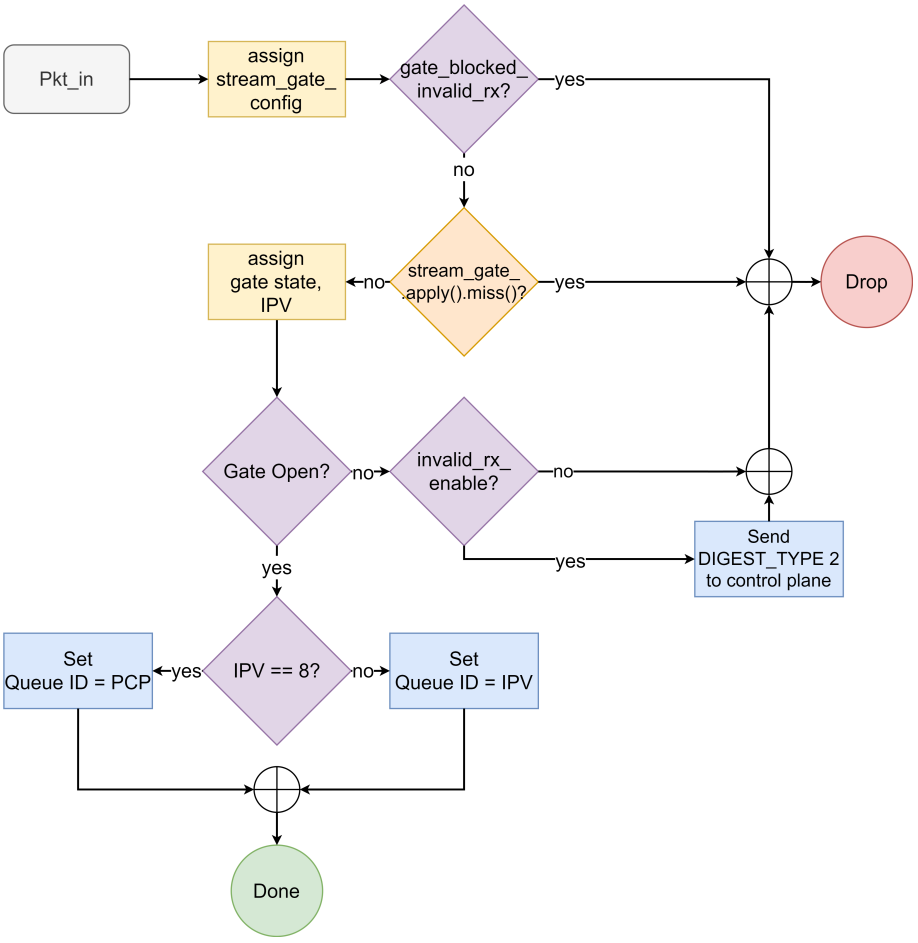


Abbildung 4.5.: Kontrollflussgraph des Stream-Gates.

4.2.3.1. Periodizität

Um einen fortwährenden Betrieb des PSFP-Switches auf unbestimmte Zeit zu ermöglichen, muss der Zeitplan einer Stream-Gate Instanz periodisch geplant werden. Eine Herausforderung bei der Implementierung ist die Abbildung des absoluten Zeitstempels eines eingehenden Frames auf die relative Position innerhalb einer geplanten Periode des Stream-Gates. Dafür wird der Zeitpunkt der letzten, vollständigen Periode gespeichert und für die Berechnung verwendet.

Als erster Schritt wird hierfür der in der TNA spezifizierte *Paketgenerator* (vgl. Abb. 2.5) so konfiguriert, dass er nach Ablauf einer vollständigen Periode ein Paket generiert. Es können insgesamt acht verschiedene zu generierende Pakete mit unterschiedlichen Perioden konfiguriert werden. Ein so generiertes Paket enthält einen `Ethernet`-Header, sowie einen `Timer`-Header, der das Paket hinsichtlich der gesendeten Pipe und der Periode identifiziert. Beim Eingang eines Frames auf einem für den Paketgenerator konfigurierten Port, das einen `Timer`-Header enthält, wird der Zeitstempel des generierten Pakets in ein portspezifisches Register geschrieben. Dieser gespeicherte Wert entspricht der letzten, vollständig abgeschlossenen Periode.

Für ein eingehendes Frame, das die Stream-Gate Instanz durchläuft, wird nun die relative Position aus der Differenz zwischen dem Eingangs-Zeitstempel und der letzten vollständig abgeschlossenen Periode berechnet.

Der obige Ansatz hat die Einschränkung, dass ein PSFP-Switch nur acht verschiedene Stream-Gate Zeitpläne verwalten kann. Aus diesem Grund berechnet die Control Plane eine Hyperperiode aus allen für das Stream-Gate definierten Perioden. Die Dauer der Hyperperiode entspricht dem kleinsten, gemeinsamen Vielfachen (kgV) aller Einzelperioden und wird als Intervall für den Paketgenerator konfiguriert. Jede einzelne Periode muss für die Dauer einer vollständigen Hyperperiode geplant werden. Somit lassen sich insgesamt acht Hyperperioden, bestehend aus theoretisch beliebig vielen einzelnen Perioden, für einen PSFP-Switch konfigurieren. Abbildung 4.6 beschreibt eine beispielhafte Planung einer Hyperperiode, bestehend aus drei Einzelperioden mit mehreren Intervallen im offenen bzw. geschlossenen Zustand und einer Periodenlänge von zwei, drei bzw. vier Zeitschritten. Daraus ergibt sich ein kgV von $h = 12$ als Dauer der Hyperperiode, auf die jede Einzelperiode verlängert wird. Sei i die Anzahl der bereits vollständig abgeschlossenen Hyperperioden und t_i der Zeitpunkt eines eingehenden Frames, so berechnet sich die relative Position innerhalb der Hyperperiode mit $t_{rel_i} = t_i - (i * h)$.

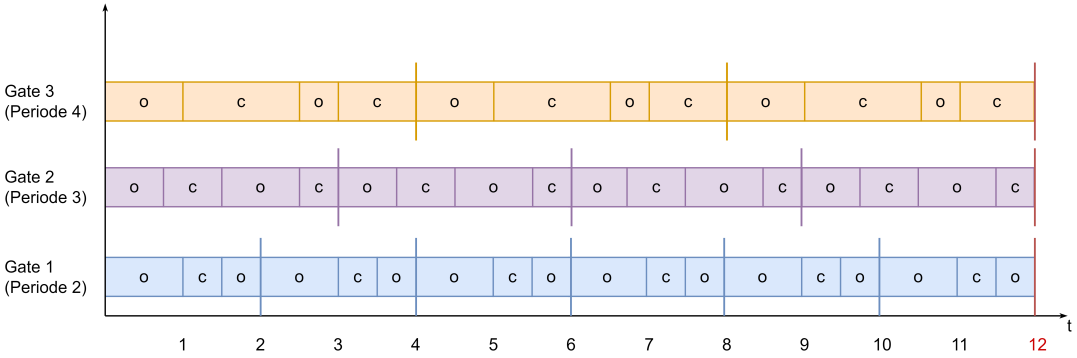


Abbildung 4.6.: Beispielhafte Planung einer Hyperperiode mit offenen (o) und geschlossenen (c) Zuständen.

Ein weiteres Problem des Paketgenerators ist, dass der Datentyp der Periode für die Paketgenerierung einem Integer entspricht. Somit wären maximal $2^{32}ns \sim 4s$ lange Perioden möglich. Zeitpläne in TSN können jedoch länger sein, insbesondere im Hinblick auf Hyperperioden. Aus diesem Grund beinhaltet die P4 Implementierung ein zusätzliches Register, das die generierten Pakete zählt und erst bei einer bestimmten, erreichten Anzahl den Zeitstempel der Hyperperiode speichert. Bei einer zu konfigurierenden Periode von mehr als $2^{32}ns$ berechnet die Control Plane die minimale Anzahl an Paketen, die die Hyperperiode mit Perioden geringer als $2^{32}ns$, teilt. Die berechneten Werte werden dynamisch über eine MAT an die Data Plane übermittelt. Beim Beispiel in Abbildung 4.6 werden drei Pakete in einem Intervall von $4s$ generiert. Beim Empfang jedes dritten Pakets wird der Zeitstempel als Abschluss der Hyperperiode mit einer Dauer von $12s$ im Register gespeichert.

4.2.3.2. Matching

Die Implementierung wählt 20 Bit aus einem 48 Bit breiten Zeitstempel aus, wodurch die zeitliche Auflösung des Schlüsselfeldes der MAT verringert wird, indem der entsprechende Bereich aus dem Zeitstempel herausgeschnitten wird. Dies ist aufgrund der Komplexität der PSFP-Implementierung erforderlich.

Abbildung 4.7 zeigt den Bereich der zeitlichen Auflösung in Abhängigkeit davon, welche der 20 Bits ausgeschnitten werden. In dieser Arbeit wurden Bit 15 bis Bit 34 ausgewählt. Dies ermöglicht eine Auflösung von $2^{15} \approx 32\mu s$ bis $2^{34} \approx 17s$. Ein Problem dieser Methode ist, dass durch das Ausschneiden zeitliche Genauigkeit verloren geht. Bei der oberen Grenze ist dies weniger problematisch, da TSN-Perioden in der Regel nicht länger geplant werden und diese durch das periodische Verhalten der Hyperperiode dargestellt werden können. Da jedoch mit PTP eine Genauigkeit von bis zu $50ns$ erreicht werden kann, leidet die P4-Implementierung unter dieser Einschränkung.

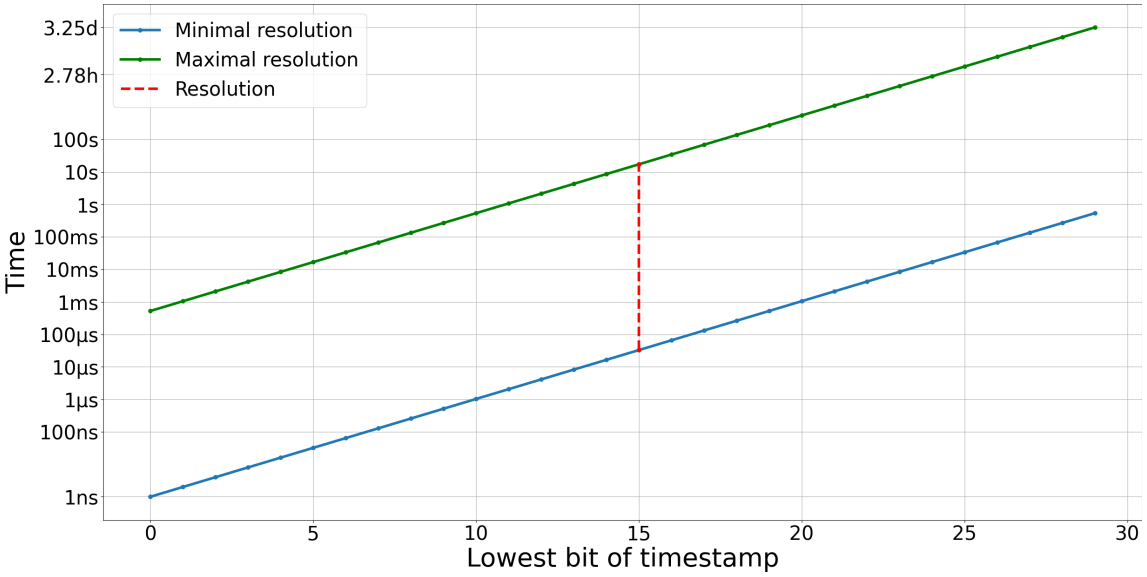


Abbildung 4.7.: Zeitliche Auflösung der Intervalle, abhängig welche der 20 Bit ausgeschnitten wurden.

4.2.4. Flow-Meter Implementierung

Als letzte Komponente von PSFP überwacht die Flow-Meter Instanz die Bandbreite von Streams, deren Frames bis zu diesem Punkt noch nicht von einer der anderen PSFP-Komponenten verworfen wurden. Dieser Abschnitt erläutert, anhand von Abbildung 4.8, wie der Switch die Einfärbung von Frames in der P4-Implementierung, die Einhaltung der Bandbreite, sowie die Parameter der Flow-Meter Instanz `markAllFramesRed`, `colorAware` und `dropOnYellow` implementiert.

Analog zu den vorhergehenden PSFP-Komponenten lädt die Flow-Meter Instanz die Konfigurationsparameter, in Abhängigkeit von der zugewiesenen Flow-Meter ID, mittels einer MAT in die Metadaten des Frames. Anschließend wird der Parameter `pre-color`, welcher der initialen Farbe des Frames entspricht, in Abhängigkeit des `color`-Modus gesetzt. Ist die Instanz bereits permanent blockiert, weil ein vorhergehendes Frame rot markiert wurde, wird im Gegensatz zu den vorigen Komponenten das Frame nicht direkt zum Verwerfen markiert, sondern zunächst nur rot eingefärbt. Andernfalls folgt die Einfärbung des Frames durch ein *Meter*-Extern. Dazu verwendet die Instanz die `flow_meter`-MAT, die als Schlüsselfeld die ID der zugewiesenen Flow-Meter Instanz und als Action die Ausführung des Meter-Externs enthält. Wichtig ist, dass diese Action nur für Frames ausgeführt wird, die noch nicht zum Verwerfen markiert wurden. Ansonsten würden diese die reservierte Bandbreite aufbrauchen, obwohl sie bspw. das Stream-Gate nicht passiert haben und dort bereits zum Verwerfen markiert wurden. Das Meter-Extern implementiert einen Token-Bucket Policer, dessen Parameterwerte PIR, CIR, CBS und PBS von der Control Plane gesetzt werden. Abhängig vom Zustand des Token Buckets, schreibt die Action des Meter-Externs die Farbe grün, gelb, oder rot in Form eines numerischen Werts in die intrinsischen Metadaten des Traffic Managers. Es ist zu beachten, dass der `Bridge`-Header der Recirculation, die in Abschnitt 4.2.2, sowie Abschnitt 4.2.3.2 beschrieben wird, einem Frame zusätzlich 35 Byte hinzufügt. Damit diese nicht in die Berechnung des Token Bucket Algorithmus einfließen, kann die Control Plane eine Konstante definieren, die bei jeder Token Bucket Operation von der Größe des betrachteten Frames subtrahiert wird.

Befindet sich die Flow-Meter Instanz im `colorAware`-Modus, darf ein zuvor gelb markiertes Frame, indiziert durch das DEI-Flag im 802.1Q Header, nicht wieder grün gefärbt werden. Das Meter-Extern in P4 implementiert bereits den `colorAware`-Modus nach RFC2698 [13]. Er wird aktiviert, indem der `execute()` Funktion des Meters der optionale Parameter `pre-color` übergeben wird. Dieser muss zuvor in die intrinsischen Metadaten geschrieben werden und wird im Fall von PSFP anhand des DEI-Flags entschieden. Befindet sich das Flow-Meter im `colorBlind`-Modus wird zwar trotzdem der optionale `color`-Parameter an das Meter übergeben, dieser ist jedoch immer auf den Wert grün gesetzt.

Da das Meter-Extern der TNA nur die Farbe des Pakets festlegt, damit jedoch keine Action assoziiert, muss diese pro Farbe selbst implementiert werden. Dies wird in der unteren Hälfte von Abbildung 4.8 beschrieben. Dort markiert das Flow-Meter rote Pakete zum Verwerfen und sendet optional eine Digest-Nachricht an die Control Plane. Bei Erhalt dieser Nachricht schreibt die Control Plane entsprechende Einträge in die MAT, so dass zukünftig alle Frames rot markiert werden, sobald ein Frame die PIR überschritten hat und der Parameter `markAllFramesRed` gesetzt ist. Ist das Frame gelb markiert setzt der Kontrollblock, abhängig vom `dropOnYellow`-Parameter das DEI-Flag, oder markiert es zum Verwerfen. Ein grün gefärbtes Frame wird ohne weitere Action weitergeleitet.

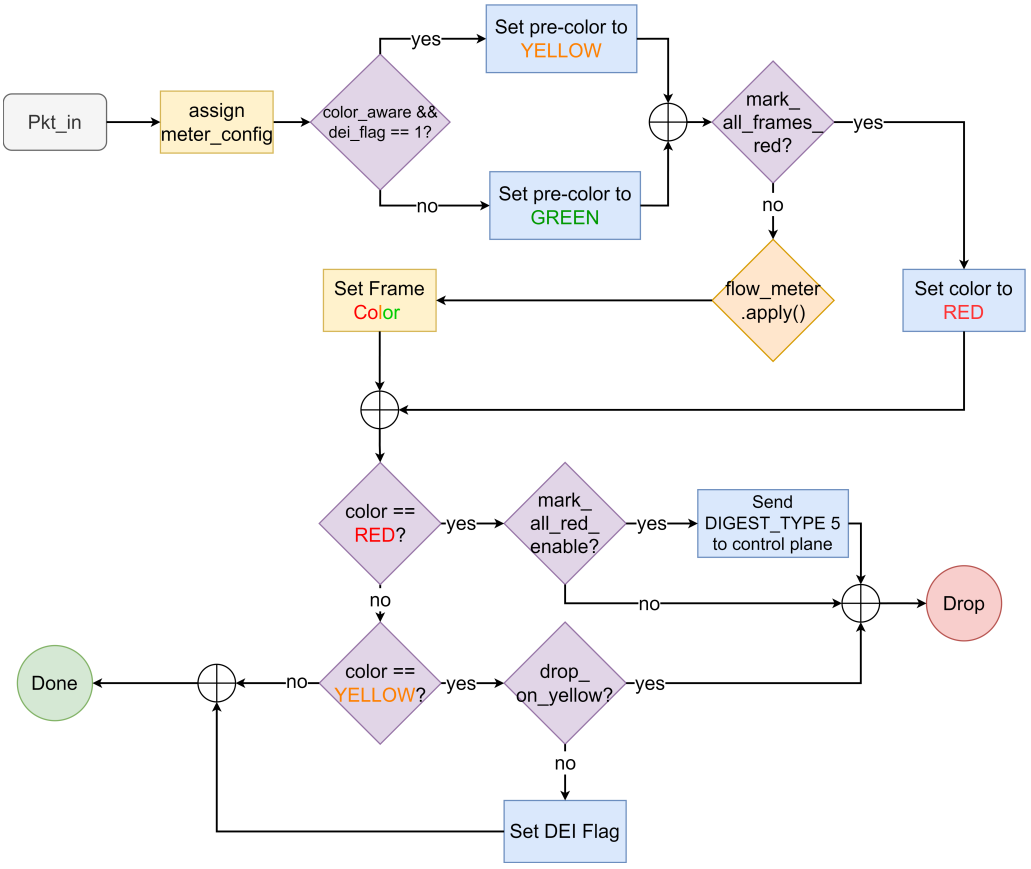


Abbildung 4.8.: Kontrollflussgraph des Flow-Meters.

Zusätzlich zu dem in Tabelle 2.2 definierten Counter, der die Anzahl der rot markierten Frames im Flow-Meter zählt, wurden weitere Counter-Instanzen für empfangene und grün bzw. gelb markierte Frames implementiert. Dies ermöglicht eine genauere Verifizierung der Funktionalität.

Da ein Frame nur einmal gegen eine spezifische MAT geprüft und eine entsprechende Action nur einmal ausgeführt werden kann, ergibt sich die Einschränkung, dass einem identifizierten Frame in der P4-Implementierung lediglich eine einzige und nicht mehrere Flow-Meter Instanzen zugeordnet werden können.

4.2.5. Synchronisierung mit der Netzwerkzeit

Eine Voraussetzung für die Funktion des PSFP-Mechanismus ist die Zeitsynchronisierung aller Netzwerkteilnehmer, insbesondere der Hosts und Switches. Nur so kann die exakte Zuordnung eines Frames zu einem PSFP-Zeitintervall korrekt erfolgen.

In dieser Arbeit wird davon ausgegangen, dass die Control Plane und die Hosts bereits eine funktionierende Zeitsynchronisierung, z.B. mittels PTP, konfiguriert haben. Die Control Plane synchronisiert die Zeitintervalle des PSFP-Zeitplans in der Data Plane mit ihrer eigenen Netzwerkzeit, indem sie Funktionen bereitstellt, die alle Intervalle um einen konstanten Faktor verschieben. Nach Abschluss der ersten

Hyperperiode eines Zeitplans generiert die Data Plane eine Digest-Nachricht, die den Zeitstempel enthält, zu dem die Hyperperiode abgeschlossen wurde. Beim Empfang durch die Control Plane berechnet diese zunächst den Faktor um den die Zeit der Data Plane gegenüber der Netzwerkzeit verschoben ist aus der Differenz des Zeitstempels und der aktuellen Netzwerkzeit. Auf dieser Grundlage berechnet sie dann die daraus resultierenden, neuen Intervallgrenzen. Der Switch leitet Frames erst weiter, nachdem der initiale Abschluss der ersten Hyperperiode vollendet ist. Dies wird dem Anwender über eine entsprechende Ausgabe über die Konsole der Control Plane mitgeteilt.

Mit zusätzlichem Konfigurationsaufwand könnte der Intel® Tofino Switch-ASIC auch direkt als PTP-Teilnehmer in ein Netzwerk integriert werden, um eine hochpräzise Zeitsynchronisierung zu erreichen. Alternativ kann ein Intel® Tofino Switch-ASIC selbst als globaler Zeitgeber für ein Netzwerk fungieren, indem er das Data Plane Precision Time Protocol (DPTP) verwendet, wie in Kapitel 3.2 beschrieben. Um die Möglichkeit einer P4-Implementierung von PSFP zu demonstrieren, ist es jedoch ausreichend, von einer bereits synchronisierten Netzwerkzeit auszugehen.

4.2.6. Implementierte Control Plane

Die Funktionalität der Algorithmen der PSFP-Komponenten wird in der Data Plane mithilfe der P4-Sprache spezifiziert. Zusätzlich müssen die Match-Action-Tables (MATs) der im Abschnitt 4.2 beschriebenen Komponenten mit Informationen über die Konfiguration und Definition der einzelnen Stream-Filter, Stream-Gate und Flow-Meter Instanzen versorgt werden. Diese Aufgabe übernimmt die in diesem Abschnitt beschriebene Control Plane.

Das Programm der Control Plane ist in `Python3` geschrieben und kommuniziert mit der Data Plane über das gRPC-Protokoll unter Verwendung der proprietären `bfruntime`-Bibliothek. Es bildet den Switch als Objekt mit den PSFP-Komponenten als Unterklassen ab, die jeweils die Konfigurationsparameter und Tabellenstrukturen der verschiedenen Komponenten von PSFP abbilden. Beim Start der Control Plane durchläuft der Controller einen Initialisierungsprozess, der die Ports des Switches konfiguriert und die Konfigurationsparameter validiert. Beispielsweise wird überprüft, ob alle `stream_handle`-Parameter, die in Flow-Meter- und Stream-Gate-Instanzen verwendet werden, in einer Stream-Filter-Instanz definiert sind. Im Anschluss schreibt der Controller die Konfigurationsparameter der PSFP-Komponenten, sowie die Identifikationsregeln des Stream-Filters in die korrespondierenden MATs der Data Plane. Die Parameter können in einer separaten Konfigurationsdatei angegeben werden, die in Listing 4.1 dargestellt ist. Als nächstes berechnet der Controller die für die Hyperperiode benötigten Intervalle und die Anzahl der Pakete des Paketgenerators und startet diesen. Zuletzt startet der Controller einen Thread, der kontinuierlich auf Digest-Nachrichten lauscht, sowie einen Thread, der die Counter-Externs zur Berechnung der Bandbreite in einem Intervall von $250ms$ ausliest und visualisiert. Nach Ablauf der ersten Hyperperiode berechnet die Control Plane den zeitsynchronisierten Zeitplan des Stream-Gates unter Verwendung

des Verschiebungsfaktors zur Netzwerkzeit gemäß Abschnitt 4.2.5 und schreibt die Intervalle in die MAT des Stream-Gates. Der Switch ist nun bereit, PSFP auf eingehende Frames anzuwenden. Der Anwender wird über den Initialisierungsprozess, sowie über Ereignisse, wie z.B. das Blockieren von Stream-Gates, über die Konsole der Control Plane informiert. Ist der Parameter `simulation` in der Konfigurationsdatei gesetzt, beendet die Control Plane die Ausführung nach 20 Sekunden und protokolliert die gesammelten Daten der Counter-Externs in der angegebenen `json`-Datei.

```
{
    "parameters": {
        "max_sdu": 2000,
        "filter_sdu": true ,
        "schedule": "FIFTYFIFTY",
        "gate_time_shift": 0,
        "close_gate": false ,
        "mark_red": false ,
        "drop_yellow": false ,
        "color_aware": false ,
        "cir_kbps": 700000,
        "pir_kbps": 800000,
        "cbs": 1000,
        "pbs": 1000,
        "simulation": true ,
        "simulation_data_file": "result.json"
    }
}
```

Listing 4.1: Konfigurationsdatei der Control Plane.

5. Ergebnisse und Simulation

In diesem Kapitel wird die korrekte Funktionsweise der P4-Implementierung von PSFP auf einem Intel® Tofino Switch-ASIC verifiziert. Dazu wird zunächst die Umgebung, in der die Implementierung getestet wurde, und die allgemeine Vorgehensweise beschrieben. Anschließend werden einige Testfälle zur Verifikation der einzelnen PSFP-Komponenten definiert und deren Ergebnisse diskutiert.

5.1. Modalitäten der Testumgebung

Die Testumgebung der PSFP-Implementierung beinhaltet einen Host, dem eine IPv4-Adresse zugewiesen wurde und der über Ethernet mit dem Intel® Tofino Switch verbunden ist. Der Host startet ein `Python3`-Skript, das mithilfe der `scapy`-Bibliothek Pakete generiert, die einen Ethernet-, einen 802.1Q- und einen IPv4-Header, sowie Nutzdaten mit variabler Länge enthalten. Die gesendeten Pakete sind an den Host selbst adressiert und werden über das mit dem Switch verbundene Netzwerkinterface gesendet, wo sie den PSFP-Mechanismus durchlaufen. Dem Skript können Parameter, wie z.B. die zu sendende Bandbreite und die Paketgröße, übergeben werden. Zusätzlich kann ein weiteres `Python3`-Skript auf dem Host gestartet werden, das die empfangenen Pakete und deren Größe protokolliert. Außerdem liest die Control Plane die Werte der Counter-Externs der P4-Implementierung des Switches periodisch in einem Intervall von $250ms$ aus und erfasst so die Raten der jeweiligen Komponenten.

Zur Verifizierung der PSFP-Implementierung werden die einzelnen Funktionen isoliert getestet. Parameter, die für den Test einer bestimmten Funktion nicht benötigt werden, werden so gewählt, dass sie keinen Einfluss auf die zu testende Funktion nehmen können. Beispielsweise werden die Parameter der Flow-Meter-Instanz (CIR und PIR) für Tests der Funktionalität der Stream-Filter oder Stream-Gates auf einen größeren Wert, als den der Senderate gesetzt, so dass die Flow-Meter Instanz diese nicht beeinflussen kann. Tabelle 5.1 gibt einen Überblick über die gewählten Parameter, mit denen die isolierten Tests durchgeführt wurden. Diese können in einer Konfigurationsdatei der Control Plane spezifiziert werden.

Die Control Plane implementiert drei beispielhafte Zeitpläne der Stream-Gates, die in Abbildung 5.1 dargestellt sind. Jeder der Zeitpläne beschreibt eine Periode von acht Sekunden mit unterschiedlichen Intervallen. Der *50:50*-Zeitplan beinhaltet zwei Intervalle, die jeweils für die Hälfte der Periode offen bzw. geschlossen sind. Im Zeitplan *1-4-2-1* ist eine zeitliche Abfolge von Intervallen unterschiedlicher Länge definiert, in denen der Zustand alterniert. Er beginnt immer mit dem Zustand „offen“. Der *Open*-Zeitplan besteht aus einem einzigen Intervall, das sich permanent

im offenen Zustand befindet. Sofern nicht anders angegeben, sendet der angeschlossene Host mit einer Datenrate von 1 Gbps. Die Durchführung der Tests und ihre Ergebnisse werden im Folgenden beschrieben.

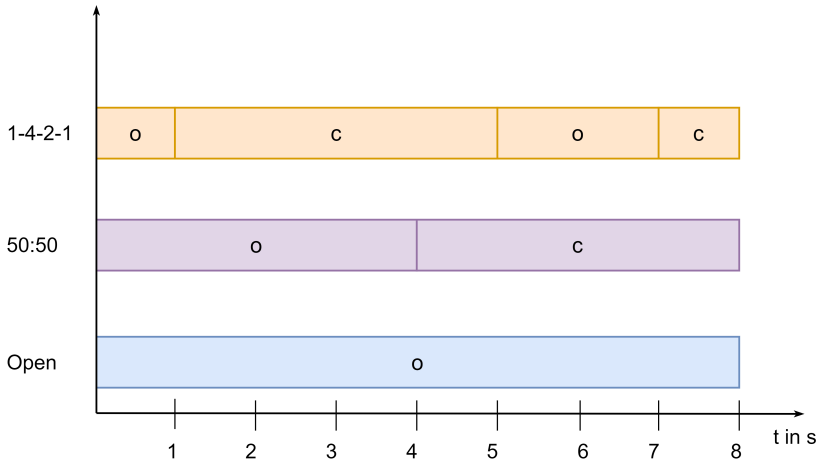


Abbildung 5.1.: Intervalle der implementierten Stream-Gate Zeitpläne mit offenen (o) und geschlossenen (c) Zuständen.

Getestete Funktion	**SDU** (bytes)	**MaxSDU** (bytes)	**Gate Schedule**	**CIR** (mbps)	**PIR** (mbps)	**CBS** (mbit)	**PBS** (mbit)
MaxSDU	1200, 1500	1300	Open	1200	1300	1	1
50:50 schedule	1500	2000	50:50	1200	1300	1	1
1-4-2-1 schedule	1500	2000	1-4-2-1	1200	1300	1	1
Invalid RX	1500	2000	50:50	1200	1300	1	1
Bandwidth Small CBS	1500	2000	Open	700	800	1	1
Bandwidth Large CBS	1500	2000	Open	700	800	1400	1500
MarkAllFramesRed	1500	2000	Open	700	800	1	1
DropOnYellow	1500	2000	Open	700	800	1	1
ColorAware	1500	2000	Open	1100	1200	1	1

Tabelle 5.1.: Gewählte Parameter der Tests zur Verifizierung der PSFP-Implementierung.

5.1.1. Statistische Aussagekraft

Jeder der folgenden Tests wurde mit einer Stichprobengröße von $N = 100$ Durchläufen für je 20 Sekunden simuliert. Zu diesem Zweck wurde die Simulation, bestehend aus dem Start der Control Plane und dem Start des Sende- und Empfängerskripts auf dem Host mit den entsprechenden Parametern, sowie die Protokollierung der Ergebnisse automatisiert. Listing 5.1 zeigt beispielhaft ein Skript zur Automatisierung der Simulation des Stream-Filters in Abschnitt 5.2.1. Auf den gesammelten Daten wurden Konfidenzintervalle mit einem Signifikanzniveau von $\alpha = 5\%$ berechnet. Der Stichprobenumfang von $N = 100$ war mindestens erforderlich, um einen relativen Fehler von $\gamma = \frac{\delta(n,\alpha)}{|\bar{X}(n)|-\delta(n,\alpha)} \leq 5\%$ bei einem absoluten Fehler von $\delta(n, \alpha)$ einhalten zu können [20]. Auf eine grafische Darstellung der Konfidenzintervalle in den Ergebnissen wurde aus Gründen der Übersichtlichkeit verzichtet.

```
#!/bin/bash

N=100

for i in $(seq $N); do
    echo "Starting simulation $i"
    # Log incoming packets on host
    ssh h1 "sudo python3 receive.py -l max_sdu.json" &
    # Send with a SDU of 1200 Bytes
    ssh h1 "sudo python3 send.py -l 1200 -s 800 -c -1" &
    # Increase SDU to 1500 Bytes
    ssh h1 "sleep 14 && sudo python3 send.py -l 1500 -s 200 -c -1" &
    # Start controller
    python3 ./controller.py
    sleep 2 && cleanUp && sleep 1
    echo "Simulation $i/$N complete."
done

cleanUp() {
    ssh h1 "sudo pkill python3; sudo pkill tcpreplay"
    sudo pkill python3
}

trap cleanUp SIGINT
```

Listing 5.1: Automatisierte Simulation der Stream-Filter Funktion.

5.2. Durchführung der Tests

Nachfolgend werden die Ergebnisse der einzelnen durchgeführten Tests und deren Aussagen dargestellt. Für die Verifikation des Stream-Filters und des Stream-Gates wurden die empfangenen Pakete am Host gemessen. Die Graphen des Flow-Meters entsprechen den Werten aus den Counter-Externs. Die blaue Senderate entspricht dabei der am Switch gemessenen Senderate des Hosts.

5.2.1. Verifizierung der Stream-Filter Funktion

Der Funktionsumfang einer Stream-Filter Instanz umfasst die korrekte Identifikation und Zuordnung eines Frames zu einem Stream, die aktive Stream-Identifikation, bei der bestimmte Felder überschrieben werden, sowie den Parameter *StreamBlockedDueToOversizeFrame*. Die korrekte Funktionsweise wird im Folgenden gezeigt.

Die Funktionen zur *Streamidentifizierung* wurden nicht explizit getestet, da der Switch diese implizit beim Betrachten jedes Frames durchführt. Die korrekte Identifizierung kann daher angenommen werden, da der Switch die PSFP-Mechanismen auf die identifizierten Frames anwendet. Der Stream des Hosts in den Tests wird durch die zugewiesene VLAN-ID, den Zielport sowie die Ziel-MAC-Adresse identifiziert. Zusätzlich wird für jeden Test eine *aktive* Identifizierung verwendet, die die VLAN-ID überschreibt. Dieses Verhalten kann anhand der empfangenen Pakete im `Python3`-Skript verifiziert werden.

Abbildung 5.2 zeigt die Funktionsweise des *StreamBlockedDueToOversizeFrame*-Parameters. Die Grafik zeigt die Anzahl der empfangenen Frames beim Host, der 20 Sekunden lang konstant Daten mit einer SDU von 1200 Byte an sich selbst sendet. Nach fünf Sekunden wird die SDU auf 1500 Byte erhöht und überschreitet damit die konfigurierte SDU von 1300 Byte. Es ist zu erkennen, dass der Host ab diesem Zeitpunkt keine Frames mehr empfängt, da der Stream-Filter den Teilnehmer korrekterweise permanent von der Kommunikation im Netzwerk ausgeschlossen hat.

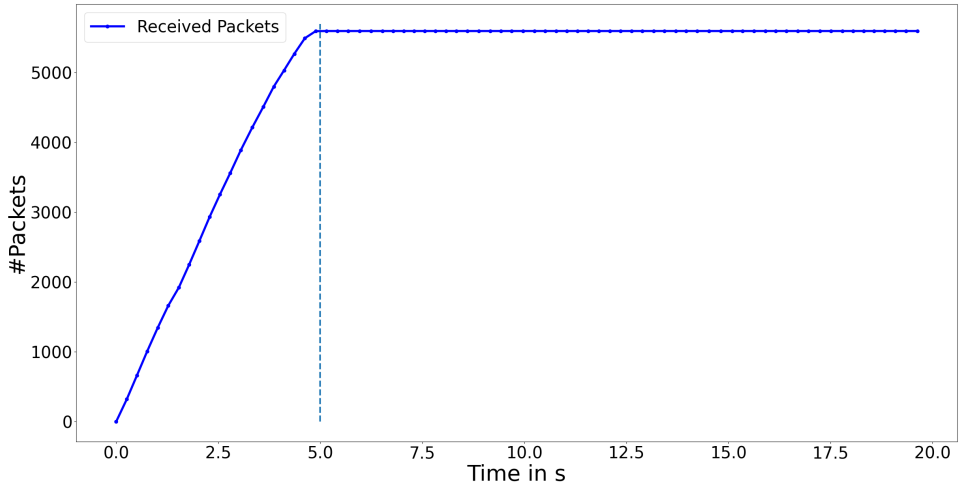


Abbildung 5.2.: Funktion des *StreamBlockedDueToOversizeFrame*-Parameters.

5.2.2. Verifizierung der Stream-Gate Funktion

Eine Stream-Gate Instanz muss eingehende Frames einem Intervall des zugewiesenen Zeitplans zuordnen und dessen Periodizität sicherstellen. Hierfür werden im Folgenden die beiden implementierten Zeitpläne *50:50* und *1-4-2-1* isoliert betrachtet. Anschließend wird die Funktionalität des Parameters *GateClosedDueToInvalidRX* überprüft.

Während des Tests der jeweiligen Zeitpläne sendet der angeschlossene Host kontinuierlich Frames über den Switch an sich selbst. Abbildung 5.3 und Abbildung 5.4 zeigen jeweils die Anzahl der empfangenen Pakete beim Host unter Anwendung des zugewiesenen Zeitplans. Als Zeitpunkt 0 wird der Zeitpunkt des ersten empfangenen Frames verwendet, der gleichzeitig der Zeitpunkt des Endes der ersten Hyperperiode nach acht Sekunden ist. Diese Zeit muss abgewartet werden, um sich mit der Netzwerkzeit gemäß Abschnitt 4.2.5 zu synchronisieren. Die Grafiken zeigen die ersten 20 Sekunden des Empfangsfensters. Es wird deutlich, dass exakt 2.5 Perioden mit einer Dauer von acht Sekunden auftreten. Dies verifiziert die korrekte Funktion des periodischen Ablaufs. Zusätzlich zeigen die Tabellen 5.2 und 5.3 die Werte der Counter-Externs der einzelnen Intervalle im Vergleich zum erwarteten, relativen zeitlichen Anteil der Intervalle an der gemessenen Gesamtdauer. Es ergibt sich eine maximale Abweichung von 0.9% bei der Zuweisung eines Frames zum entsprechenden Intervall. Dies könnte jedoch auf die Ungenauigkeiten beim Auslesen der Counter-Externs, oder auf die nur angenommene Zeitsynchronität des Hosts mit dem Netzwerk zurückzuführen sein.

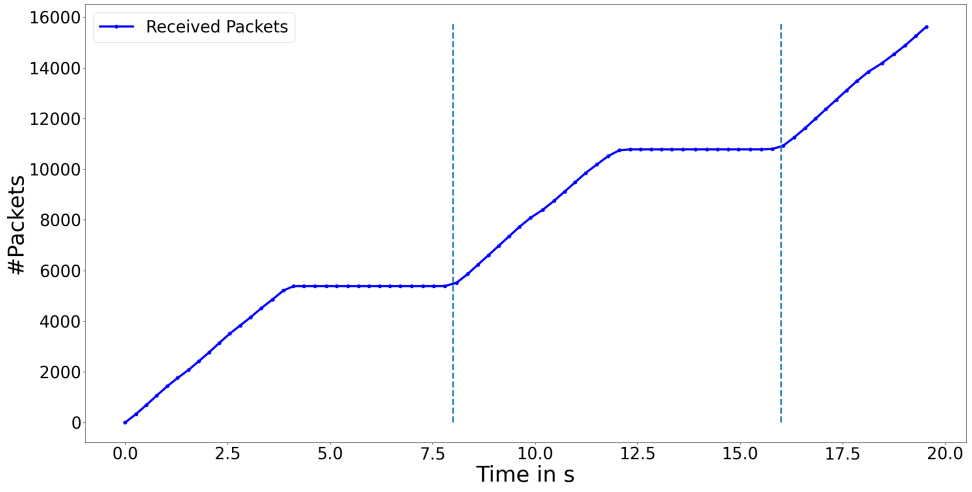


Abbildung 5.3.: Funktion des 50:50 Zeitplans.

Intervall	#Pakete	Zustand	Dauer pro Periode	Gemessener Anteil	Relativer Anteil an der Gesamtdauer	Abweichung
1	999611.5	1	4s	0.584	0.593	0.009
2	711137.3	0	4s	0.416	0.407	0.009
Summe	**1710748.8**		**8s**	**1**	**1**	

Tabelle 5.2.: Paket-Counter Werte im P4-Programm der Intervalle des 50:50s Zeitplans nach $N = 100$ Durchläufen bei einer Dauer von $\approx 20.220s$.

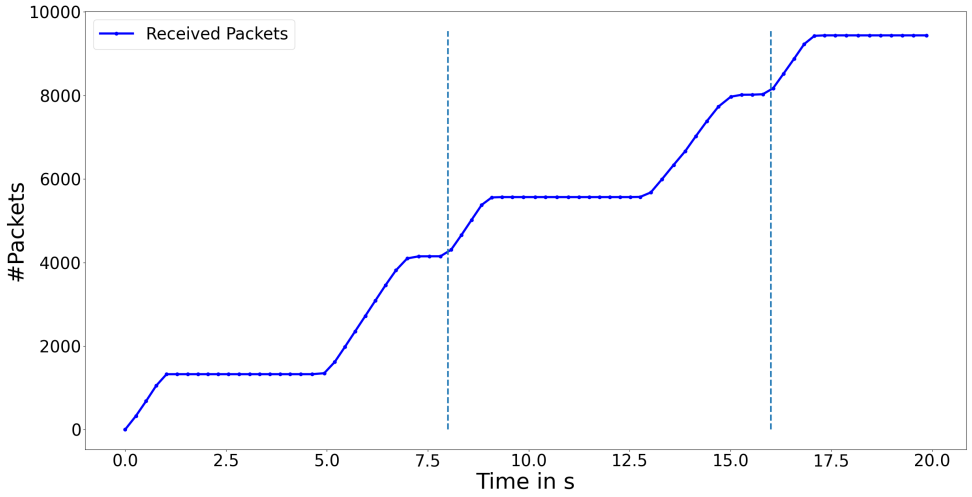


Abbildung 5.4.: Funktion des 1-4-2-1 Zeitplans.

Intervall	#Pakete	Zustand	Dauer pro Periode	Gemessener Anteil	Relativer Anteil an der Gesamtdauer	Abweichung
1	249400.8	1	1s	0.146	0.148	0.002
2	961724.9	0	4s	0.562	0.555	0.007
3	333351.8	1	2s	0.195	0.198	0.003
4	166628.8	0	1s	0.097	0.099	0.001
Summe	**1711106.3**		**8s**	**1**	**1**	

Tabelle 5.3.: Paket-Counter Werte im P4-Programm der Intervalle des 1-4-2-1 Zeitplans nach $N = 100$ Durchläufen bei einer Dauer von $\approx 20.223s$.

Abbildung 5.5 zeigt das Verhalten des Parameters *GateClosedDueToInvalidRX* unter Verwendung des 50:50 Zeitplans. Das erste Intervall des 50:50 Zeitplans dauert vier Sekunden an und das Stream-Gate befindet sich währenddessen im geöffneten Zustand. Der Host sendet kontinuierlich mit einer konstanten Rate Frames über den Switch. Im zweiten Intervall wechselt das Stream-Gate in den geschlossenen Zustand. In Kombination mit dem Parameter *GateClosedDueToInvalidRX* wird das

Stream-Gate permanent blockiert, sobald ein Frame in einem geschlossenen Zustand am Switch eintrifft. Dies ist in Abbildung 5.5 nach vier Sekunden, beim Wechsel in das Intervall mit geschlossenem Zustand, zu sehen. Im Gegensatz zu Abbildung 5.3 wird das Stream-Gate nach Ablauf der ersten Periode nicht wieder geöffnet, sondern bleibt permanent blockiert.

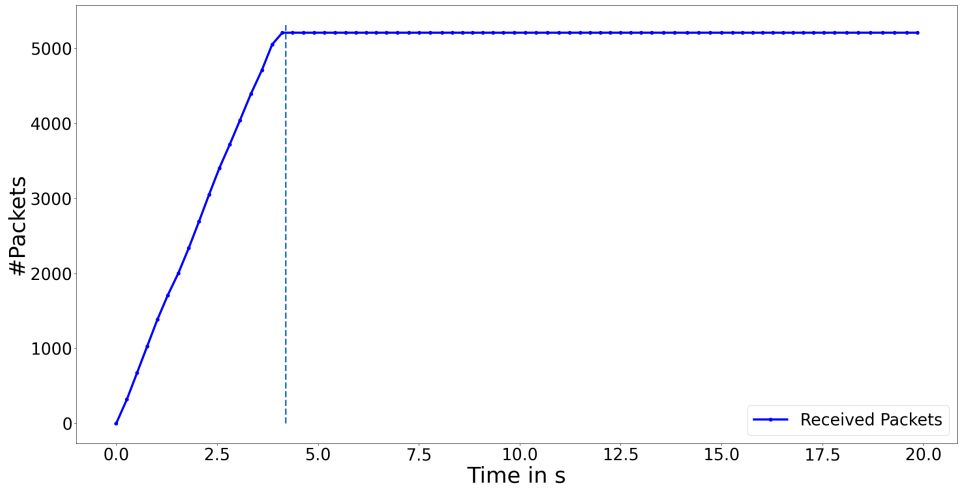


Abbildung 5.5.: Funktion des *GateClosedDueToInvalidRX*-Parameters.

5.2.3. Verifizierung der Flow-Meter Funktion

Die Flow-Meter Instanz muss die Einhaltung der zugesicherten Bandbreite überwachen und die betrachteten Frames entsprechend einer von drei Farben einfärben. Zusätzlich implementiert sie die Parameter *MarkAllFramesRed*, *DropOnYellow*, sowie *ColorMode*. Für die folgenden Auswertungen der Funktionen wurden die Werte der Counter-Externs im P4-Programm periodisch von der Control Plane ausgelesen. Für jede Farbe, sowie für die am Flow-Meter eingehenden Frames existiert hierfür ein eigener Counter aus dessen Wert die Control Plane die Bandbreite der jeweils farbig markierten, sowie der empfangenen Frames berechnet. Nachfolgend wird die korrekte Funktion der Farbmarkierung für einen Bucket mit kleiner bzw. großer CBS und der zusätzlichen Parameter verifiziert.

Zunächst zeigt Abbildung 5.6 die Einfärbung von Frames mit einer sehr geringen Committed Burst Size (CBS) bzw. Peak Burst Size (PBS) von 1 Mbit. Der Host sendet mit einer konstanten Rate von 1 Gbps. Gemäß Tabelle 5.1 ist die Committed Information Rate (CIR) auf 700 Mbps eingestellt und durch die gestrichelte grüne Linie dargestellt. Sie entspricht der maximalen, sicher zugesicherten Rate. Analog dazu ergibt sich die Excess Information Rate (EIR) aus der Differenz von Peak Information Rate (PIR) und CIR. Sie ist auf 100 Mbps gesetzt und entspricht der Obergrenze der gelb markierten Frames. Es ist gut zu erkennen, dass sich die Rate

der grün und gelb markierten Frames exakt der konfigurierten EIR bzw. CIR angleicht. Die Differenz zwischen Senderate und PIR beträgt in der Durchführung 200 Mbps und entspricht genau der Rate der rot markierten Frames. Die Summe der grün, gelb und rot markierten Frames entspricht somit exakt der Senderate.

Abbildung 5.7 beschreibt den selben Versuchsaufbau, jedoch mit einer CBS, die der doppelten CIR entspricht. Die PBS beträgt 1500 Mbit. Da der Token Bucket zu Beginn bis zur maximalen CBS bzw. PBS gefüllt ist, kann der Switch Burst-Übertragungen abfangen. Erkennbar ist dies am Anstieg der grünen Rate auf die Senderate in den ersten vier Sekunden bzw. am Anstieg der gelben Rate von vier bis sieben Sekunden auf 300 Mbps. In diesem Zeitraum müssen keine Frames verworfen werden. Nach dem Leeren beider Token Buckets gleicht sich der Verlauf dem in Abbildung 5.6 dargestellten an und die Rate der rot markierten Frames steigt auf 200 Mbps.

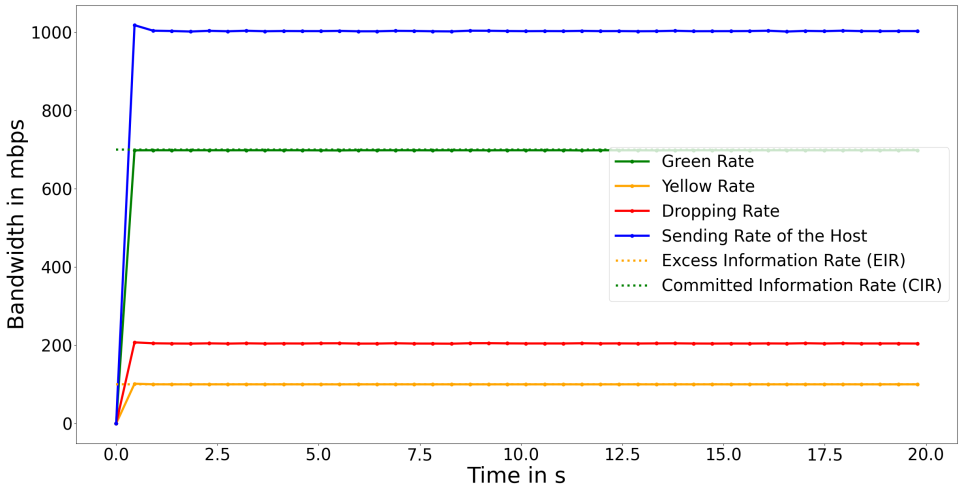


Abbildung 5.6.: Funktion des Flow-Meters mit kleinem CBS.

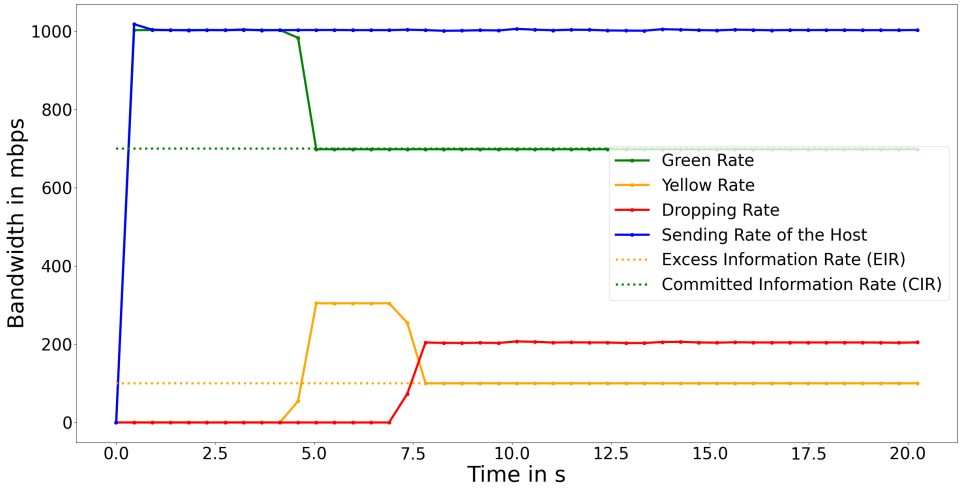


Abbildung 5.7.: Funktion des Flow-Meters mit großem CBS.

Die Funktionsweise des Parameters *DropOnYellow* ist in Abbildung 5.8 dargestellt. Ist dieser Parameter aktiviert, werden zuvor gelb markierte Frames rot umgefärbt und in Folge dessen verworfen. Die Abbildung zeigt, dass unter den gleichen Bedingungen wie in Abbildung 5.6 die Rate der rot markierten Frames nun 300 Mbps entspricht, während keine Frames mehr gelb markiert sind. Die 100 Mbps der EIR werden also zusätzlich verworfen.

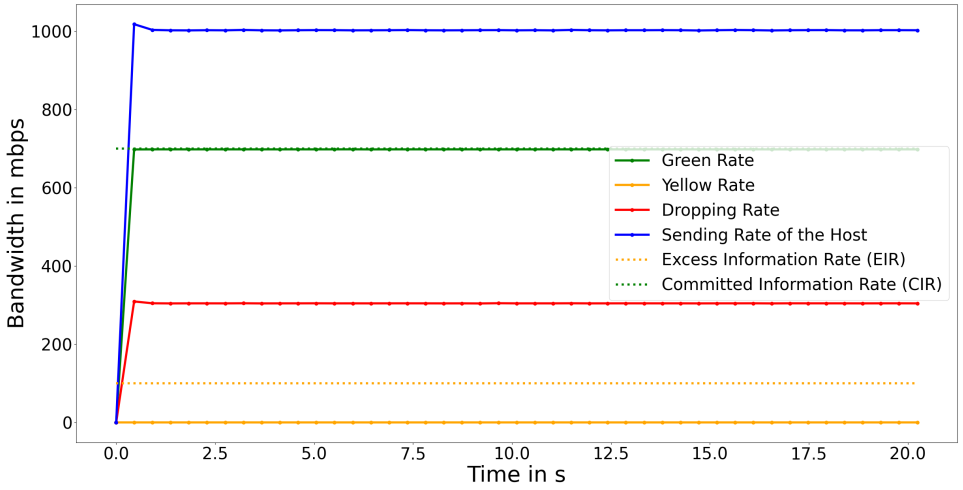


Abbildung 5.8.: Funktion des *DropOnYellow*-Parameters.

Um die Funktionalität des Parameters *MarkAllFramesRed* zu überprüfen, sendet der Host zunächst mit einer reduzierten Bandbreite von 750 Mbps, wobei 700 Mbps grün und 50 Mbps gelb markiert und keine Frames verworfen werden. Nach vier Sekunden wird die Senderate des Hosts auf 1 Gbps erhöht. Zu diesem Zeitpunkt überschreitet die Senderate die PIR und Frames werden rot markiert. Durch Setzen des Parameters *MarkAllFramesRed* werden ab der ersten, roten Markierung alle folgenden Frames verworfen. Dies ist in Abbildung 5.9 gut zu erkennen. Zum markierten Zeitpunkt steigt die Senderate auf 1 Gbps an und die gelbe bzw. grüne Rate fällt auf 0 Mbps ab, während die rote Rate sich der Senderate angleicht.

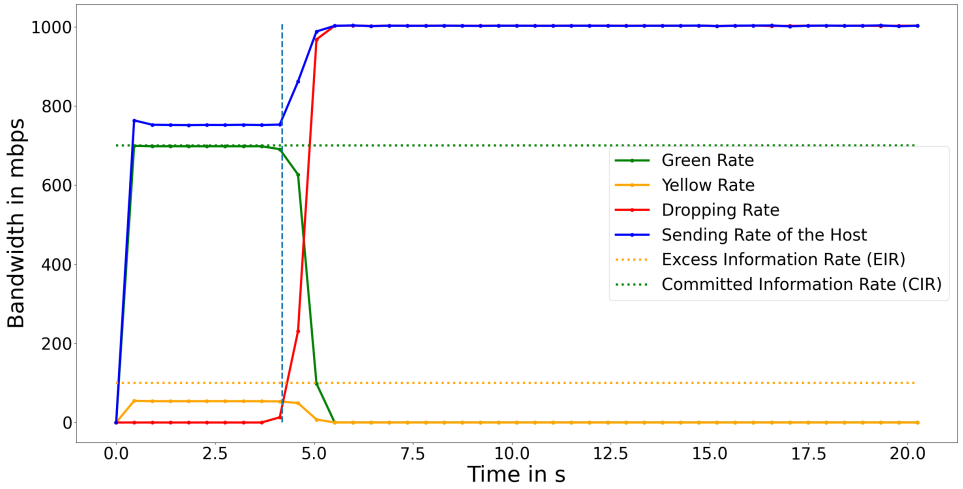


Abbildung 5.9.: Funktion des *MarkAllFramesRed*-Parameters.

Abschließend wird die Funktionalität des Parameters *ColorAware* verifiziert. Dazu wird die CIR auf einen Wert gesetzt, der größer als die Senderate ist. Der Host generiert nun Pakete, die mit einer Wahrscheinlichkeit von 60% grün und mit 40% bereits gelb markiert den Switch erreichen. Obwohl die Senderate von 1 Gbps unterhalb der konfigurierten CIR von 1.1 Gbps liegt, bleiben die 400 Mbps der bereits gelb markierten Frames des in Abbildung 5.10 dargestellten *ColorAware* Modus gelb markiert. Unter den gleichen Umständen werden die vollen 1 Gbps im *ColorBlind* Modus grün gefärbt. Dies ist in Abbildung 5.11 zu sehen.

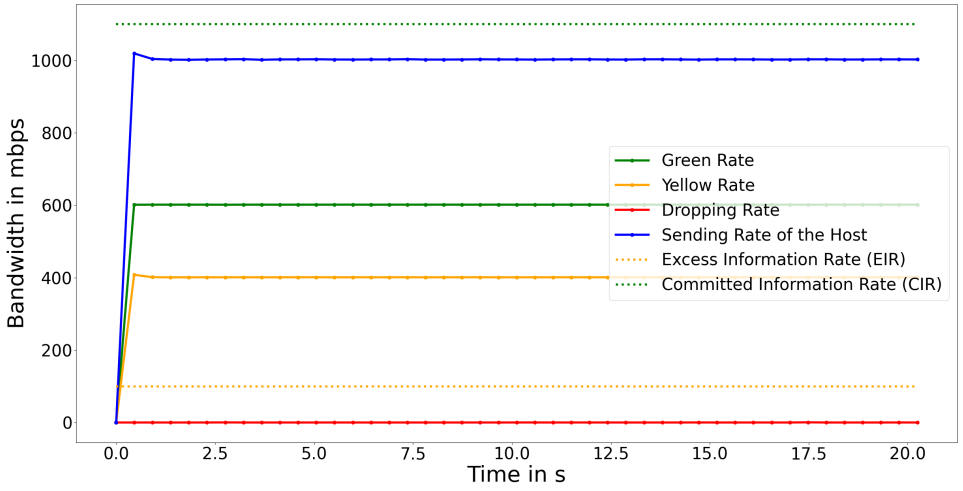


Abbildung 5.10.: Funktion des *ColorAware*-Parameters.

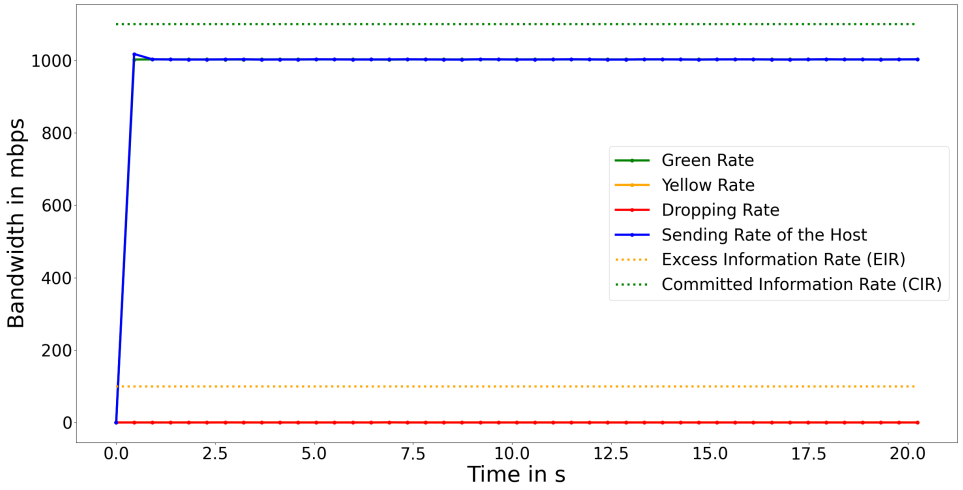


Abbildung 5.11.: *ColorBlind* Modus.

Bei allen Graphen des Flow-Meters ist kurz nach dem Erreichen der maximalen Rate von 1 Gbps ein Sprung in der Senderate zu erkennen. Dies könnte auf eine transiente Phase hindeuten, in der der Host kurzzeitig bursthaft sendet, bevor sich die Senderate stabilisiert. Ein weiterer Grund könnte die Zeitverzögerung beim ersten Auslesen des Counters sein.

6. Schlussfolgerung

Dieses Kapitel fasst die Zielsetzung und die Ergebnisse der Arbeit kurz zusammen. Anschließend wird kurz auf die Einschränkungen der PSFP-Implementierung in P4 eingegangen, die sich aus der Verwendung der TNA ergeben haben. Zuletzt wird ein Ausblick auf mögliche zukünftige Verbesserungen der Implementierung gegeben, mit denen diese Einschränkungen teilweise aufgehoben werden können.

6.1. Zusammenfassung

Das Ziel dieser Arbeit war es, den PSFP-Mechanismus zur Überwachung und zur Einhaltung von TSN-Verträgen im Kontext von SDN in der P4-Programmiersprache auf einem Intel® Tofino Switch-ASIC zu implementieren. Das implementierte Programm der Data Plane beinhaltet die drei Komponenten Stream-Filter, Stream-Gate und Flow-Meter von PSFP und bietet damit Möglichkeiten, Streams anhand verschiedener Kriterien zu identifizieren und die Einhaltung der TSN-Verträge durch die Teilnehmer zu überwachen. Dabei wurden verschiedene P4- sowie TNA-spezifische Probleme erkannt und gelöst. Die implementierte Control Plane setzt eine Schnittstelle zur Konfiguration der Komponenten und deren Parameter um, um die PSFP-Implementierung auf lokale Gegebenheiten der Netzwerkumgebung anpassen zu können. Die korrekte Funktionsweise der einzelnen PSFP-Komponenten, sowie deren Parameter konnte durch isolierte Testfälle erfolgreich in einer Simulation verifiziert werden.

6.1.1. Einschränkungen

Durch die Verwendung der P4-Programmiersprache und eines Switches, der die TNA implementiert, ergaben sich im Rahmen der Arbeit Einschränkungen gegenüber dem PSFP-Standard 802.11Qci hinsichtlich der Anzahl der Flow-Meter Instanzen pro Stream, des Parameters *GateClosedDueToOctetsExceeded* sowie der zeitlichen Genauigkeit bei der Abbildung von Zeitintervallen. Diese werden im Folgenden zusammengefasst.

Da die grundsätzliche Struktur eines P4-Programms auf MATs basiert, kann ein einzelnes Frame nur einmalig gegen eine bestimmte MAT geprüft werden und infolge dessen kann auch nur eine Action dieser MAT ausgeführt werden. Dies führt zu der Einschränkung, dass einem durch den Stream-Filter identifizierten Frame nur eine einzige Flow-Meter Instanz zugeordnet werden kann.

Die TNA führt, zusätzlich zum P4 Standard, zwar die Zuordnung von Daten in Intervalle ein, jedoch können Oktette nicht ohne Weiteres pro Intervall gezählt und der Zähler bei Übergang in ein neues Intervall wieder zurückgesetzt werden.

Aus diesem Grund wurde der Parameter *GateClosedDueToOctetsExceeded* in dieser Arbeit nicht implementiert.

Aufgrund der Komplexität der PSFP-Implementierung musste die Genauigkeit der zeitlichen Zuordnung von einem Intervall auf 20 Bit reduziert werden. Mit den gewählten Bits in der Implementierung wird somit eine Auflösung von $\approx 32\mu s$ bis $\approx 17s$ für die Intervalle erreicht.

6.1.2. Ausblick

Die in dieser Arbeit vorgestellte Implementierung beschreibt eine funktionsfähige Umsetzung des PSFP-Standards in P4. Darüber hinaus können die in Abschnitt 6.1.1 genannten Einschränkungen teilweise durch eine Verbesserung der Implementierung gelöst und die Robustheit durch zusätzliche Funktionalität verbessert werden.

Die Implementierung verwendet den in der TNA eingeführten `range`-Match Typen für die Zuordnung eines Frames zu einem Intervall. Eine alternative Möglichkeit wäre es, die Intervalle durch mehrere `LPM`-Match Typen abzubilden. Dies hätte den Vorteil, dass die zeitliche Genauigkeit nicht reduziert würde, sondern die vollen 48 Bit verwendet werden könnten. Der Nachteil ist, dass pro Intervall bis zu 48 Einträge benötigt werden. Die maximale Anzahl an Einträgen in MATs ist jedoch durch die Komplexität der PSFP-Implementierung begrenzt.

Eine mögliche Verbesserung der Implementierung wäre die Synchronisierung der Hosts und des Netzwerks, beispielsweise mit der DPTP-Implementierung, die in Abschnitt 3.2 vorgestellt wurde. In dieser Arbeit wurde die Zeitsynchronisierung als bereits gegeben betrachtet. Unter Verwendung eines tatsächlich vollständig synchronisierten Netzwerks könnte das Ergebnis der Simulation erneut verifiziert werden.

Eine Grundfunktionalität der Implementierung besteht aus der Kommunikation der Data Plane mit der Control Plane über Digest Nachrichten. Ein böswilliger Angreifer könnte diese Tatsache ausnutzen und einen Denial of Service Angriff provozieren, indem er die Control Plane mit Digest Nachrichten überflutet. Dieser Angriffsvektor könnte verhindert werden, indem die Data Plane zusätzlich einen Status über bereits gesendete Digest Nachrichten enthält, so dass diese nur einmal übermittelt werden.

A. Inhalt der beigelegten CD

Die beigelegte CD enthält den Quellcode der in dieser Arbeit entwickelten PSFP-Implementierung in P4, sowie den Quellcode der Control Plane und der Skripte zur Durchführung der Simulationen in den folgenden Unterordnern:

- **Implementation/P4-Implementation** enthält den Quellcode der einzelnen PSFP-Komponenten des P4-Programms, sowie das `Makefile`, das zum Start benötigt wird.
- **Implementation/Local-Controller** enthält den Quellcode der implementierten Control Plane in Python3, sowie die Konfigurationsdatei nach Listing 4.1.
- **Implementation/util** enthält das Sende- und Empfängerskript des Hosts, sowie die Skripte zur automatischen Durchführung der Simulationen.

A.1. Setup

Voraussetzung der implementierten Data Plane ist die Installation und Konfiguration der `bf-sde`, die den Compiler und die Architektur des Intel® Tofino Switch-ASIC liefert. In dieser Arbeit wurde die Version `9.9.0` der `bf-sde` verwendet.

Die Control Plane benötigt die mit der `bf-sde` mitgelieferten Python3 Bibliotheken zur Verbindung mit der Data Plane. Zusätzliche Abhängigkeiten sind in der Datei `requirements.txt` spezifiziert. Für die Control Plane wurde Python3 Version 3.8.10 verwendet.

Die Abhängigkeiten der Host-Skripte sind in der Datei `requirements.txt` im Unterordner *util* enthalten. Hierfür wurde Python3 Version 3.10.6 verwendet. Um mithilfe der `scapy`-Bibliothek Datenraten von 1 Gbps erreichen zu können, ist zusätzlich das Paket `tcpreplay` erforderlich. Dieses kann z.B. mit dem Paketmanager `apt` und dem Befehl `apt install tcpreplay` installiert werden. Zum Ausführen der Skripte sind Root Berechtigungen erforderlich.

Für die Durchführung einer Simulation ist außerdem eine Verbindung per SSH ohne Passworteingabe, ausgehend vom Tofino Switch zum Host, notwendig.

A.2. Start

Um die P4-Implementierung zu starten, muss das Makefile mit dem Befehl `make all` ausgeführt werden. Gegebenenfalls müssen die Pfade in der Datei auf das aktuelle Verzeichnis angepasst werden. Dieser Befehl führt die beiden Schritte `compile` und `start` aus, die zunächst das Programm in target-spezifischen Code kompilieren und anschließend auf dem Intel® Tofino Switch-ASIC starten. Wird das Programm beendet, ist ein Ausführen des Befehls `make start` zum erneuten Starten des Programms beim nächsten Mal ausreichend.

Die Control Plane kann in einem weiteren Terminalfenster mit dem folgenden Befehl gestartet werden:

- `./controller.py`

Nach dem Start des Controllers muss auf den Abschluss der ersten Hyperperiode gewartet werden. Im Anschluss ist der PSFP-Mechanismus einsatzbereit.

Auf dem Host kann das Senderskript beispielsweise mit den folgenden Parametern gestartet werden, um dauerhaft mit einer SDU von 1500 Bytes und einer Bandbreite von 1 Gbps zu senden:

- `python3 send.py -l 1500 -s 1000 -c -1`.

Das Empfängerskript kann optional zusätzlich mit den folgenden Parametern in einem weiteren Terminalfenster auf dem Host gestartet werden, um die gesammelten Daten in einer Datei namens `results.json` zu speichern:

- `python3 receive.py -l result.json`.

Eine Liste aller verfügbaren Parameter der beiden Skripte befindet sich in der beigelegten `README`-Datei.

A.3. Durchführung einer Simulation

Um eine Simulation zur Verifizierung eines der Parameter zu starten, können die Skripte im Unterordner `Implementation/util` verwendet werden. Der Stichprobenumfang kann im Skript festgelegt werden. Je nach durchgeführter Simulation werden die Control Plane, das Senderskript und ggf. das Empfängerskript mit den erforderlichen Parametern gestartet. Die Parameter der Control Plane müssen zu Beginn in der Konfigurationsdatei festgelegt werden. Die Ergebnisse der Simulation werden kumuliert in der angegebenen `json`-Datei auf dem Host bzw. in der Control Plane protokolliert.

Glossar

TSN Time-Sensitive Networking

PTP Precision Time Protocol

PSFP Per-Stream Filtering and Policing

SDU Service Data Unit

IPV Internal Priority Value

PCP Priority Code Point

CIR Committed Information Rate

EIR Excess Information Rate

PIR Peak Information Rate

CBS Committed Burst Size

EBS Excess Burst Size

PBS Peak Burst Size

DEI DropEligibleIndicator

SNMP Simple Network Management Protocol

SDN Software-Defined Networking

OF OpenFlow

API Application Programming Interface

P4 Programming protocol-independent packet processors

ARP Address Resolution Protocol

ASIC Application-specific integrated circuit

FPGA Field Programmable Gate Array

MAT Match-Action-Table

LPM Longest-Prefix-Match

PSA Portable Switch Architecture

TNA Tofino Native Architecture

DPTP Data Plane Precision Time Protocol

NED Network Description Language

CQF Cyclic Queueing and Forwarding

Literaturverzeichnis

[1] IEEE Standard for Local Area Network MAC (Media Access Control) Bridges. *IEEE Std. 802.1D-1998*, June 1998.

[2] IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks. *IEEE Std. 802.1Q-1998*, June 1998.

[3] IEEE Standard for Local and metropolitan area networks–Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP). *IEEE 802.1Qat-2010*, Sept. 2010.

[4] IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks— Amendment 28: Per-Stream Filtering and Policing. *IEEE Std 802.1Qci-2017*, Jan. 2017.

[5] IEEE Standard for Local and metropolitan area networks–Frame Replication and Elimination for Reliability. *IEEE Std 802.1CB-2017*, pages 1–102, Oct. 2017.

[6] IEC/IEEE International Standard - Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588*, June 2021.

[7] Ethernet ring protection switching. *ITU-T G.8032/Y.1344 Corrigendum 1*, Feb. 2022.

[8] S. Cheruvu, A. Kumar, N. Smith, and D. M. Wheeler. *Demystifying Internet of Things Security*. Apress, 1 edition, Aug. 2019.

[9] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel. NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++. In *Proceedings of the 2019 International Conference on Networked Systems (NetSys)*, Garching b. München, Germany, Mar. 2019.

[10] N. Finn. Introduction to Time-Sensitive Networking. IEEE Communications Standards Magazine, June 2018.

[11] D. Hagarty, S. Ajmeri, and A. Tanwar. *Synchronizing 5G Mobile Networks*. Cisco Press, 1 edition, June 2021.

[12] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561, 2023.

[13] J. Heinanen and R. Guerin. A Two Rate Three Color Marker. RFC 2698, Sept. 1999.

[14] B. Hopf. Analyse von „Per-Stream Filtering and Policing“ in Time-Sensitive Networking und Implementierung im Simulations-Framework OMNeT++. Bachelor's thesis, Eberhard Karls Universität Tübingen, July 2021.

[15] Intel®. $P4_{16}$ Intel® Tofino™ Native Architecture – Public Version. `https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf`, Apr. 2021. *Zugriff am 07.12.2022.*

[16] P. G. Kannan, R. Joshi, and M. C. Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 8–20, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Kannan, Pravein Govindan and Joshi, Raj and Chan, Mun Choon. DPTP (Data-Plane Time synchronization Protocol). `https://github.com/praveingk/DPTP`, Jan. 2021. *Zugriff am 30.01.2023.*

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *Computer Communication Review*, 38:69–74, 04 2008.

[19] J. L. Messenger. Time-Sensitive Networking: An Introduction. IEEE Communications Standards Magazine, June 2018.

[20] Michael Menth. Lecture notes in Modellierung und Simulation 1. Eberhard Karls Universität Tübingen, May 2022.

[21] R. S. Oliver, S. S. Craciunas, and W. Steiner. IEEE 802.1Qbv Gate Control List Synthesis Using Array Theory Encoding. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–24, Apr. 2018.

[22] OpenSim Ltd. A Quick Overview of the OMNeT++ IDE. `https://omnetpp.org/documentation/ide-overview/`, Jan. 2023. *Zugriff am 30.01.2023.*

[23] p4lang. p4lang/tutorials. `https://github.com/p4lang/tutorials`, Nov. 2022. *Zugriff am 03.11.2022.*

[24] P. Schnabel. IPv6-Tunneling mit 6in4 / 6to4 / 6over4 / 4in6. `https://www.elektronik-kompendium.de/sites/net/1904031.htm`, Oct. 2022. *Zugriff am 07.11.2022.*

[25] The P4 Language Consortium. $P4_{16}$ Language Specification. `https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf`, May 2021. *Zugriff am 14.11.2022.*

[26] The P4 Language Consortium. P4 Open Source Programming Language. `https://p4.org/`, Oct. 2022. *Zugriff am 03.11.2022.*

[27] The P4.org Architecture Working Group. $P4_{16}$ Portable Switch Architecture (PSA). `https://p4.org/p4-spec/docs/PSA.html` (working draft), Apr. 2021. *Zugriff am 20.11.2022.*

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis