Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 109

Plattformunabhängige lokale SDN-Controller auf offener Weiterleitungshardware durch Anwendung von Containertechnologie

Christian Bäumlisberger

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer/in:	M.Sc. Thomas Kohler
Begonen am:	23. Mai 2016
Beendet am:	22. November 2016
CR-Nummer:	C.2.0, C.2.1, C.2.2

Abstract

Neue und steigende Anforderungen an IT-Netzwerke stellen klassische Netzwerke vor große Herausforderungen. Software-defined Networking (SDN) löst viele Probleme klassischer Netzwerke und gewinnt daher immer mehr an Bedeutung. Dies geschieht unter anderem durch Trennung von Data und Control Plane sowie der Zentralisierung der Logik in einem logisch zentralisierten SDN-Controller. Für einige Anwendungsfälle bietet der klassische dezentrale Ansatz aber weiterhin Vorteile wie z. B. geringere Latenz und weniger Traffic Overhead. Diese Masterarbeit beschäftigst sich deshalb mit der Idee auch beim Software-defined Networking lokale Kontrolllogik mitzuverwenden. Dazu wird ein Konzept für den Einsatz eines virtualisierten und lokalen SDN-Controllers auf einem SDN-Switch zur Umsetzung lokaler Kontrolllogik aufgezeigt. Anschließend folgt die Evaluation verfügbarer SDN-Controller und Virtualisierungstechnologien, ehe über geeignete Anwendungsfälle das Konzept im Ganzen evaluiert wird.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CDPI	Control to Data-Plane Interface
IoT	Internet of Things
KVM	Kernel-based Virtual Machine
LTS	Long Term Support
OF	OpenFlow
OF-DPA	OpenFlow Data Plane Abstraction
ONF	Open Networking Foundation
OVSDB	Open vSwitch Database Management Protocol
QEMU	Quick Emulator
REST	Representational State Transfer
SDN	Software-defined Networking
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtuelle Maschine
WAN	Wide Area Network

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Immer mehr Geräte sind heutzutage miteinander vernetzt und Netzwerke dadurch so gut wie überall zu finden. Der Hauptgrund dafür sind die vielfältigen Möglichkeiten die durch eine Anbindung an das Internet entstehen. Auch in Zukunft werden immer mehr Geräte vernetzt sein, vor allem durch den Trend hin zum Internet of Things (IoT). Der damit verbundene stetige Anstieg an Nutzern und Geräten erfordert immer bessere Netzwerke. Auch die Rechenzentren der Anbieter müssen zum Ausgleich leistungsfähiger und besser vernetzt werden. Sehr häufig fällt in diesem Zusammenhang auch der Begriff des Cloud Computing und das damit verbundene Ziel immer mehr Anwendungen als Service über das Internet anzubieten. Gerade in den letzten Jahren ist Cloud Computing durch moderne Virtualisierungstechniken, die es erlauben Serverhardware effizienter und flexibler zu nutzen, immer erfolgreicher geworden. Die dadurch steigenden Anforderungen an die Rechenzentren, betreffen natürlich auch wieder die Netzwerke.

Netzwerke müssen deshalb nicht nur mit immer mehr Traffic zurechtkommen, sondern sich auch flexibel an veränderte Topologien, ausfallende Geräte, variierende Netzwerklast und weitere neue Anforderungen anpassen. Klassische Netzwerke sind dafür oft nicht ideal geeignet, da sie aufwendig zu konfigurieren und nicht flexibel genug sind [vgl. 18].

Eine Lösung, um diese Einschränkungen klassischer Netzwerke zu überwinden bietet das Software-defined Networking (SDN). Die Grundidee dabei ist die Trennung von Netzwerkmanagement (Control Plane) und Datenweiterleitung (Data Plane bzw. Forwarding Plane) [18]. Das Ziel dabei ist, eine flexible und weniger Hardware zentrierte Architektur, die logisch zentralisiert und mit globaler Sicht auf das Netzwerk über einen programmierbaren logisch zentralisierten SDN-Controller gesteuert wird. Dies bietet den Vorteil, dass das SDN-Netzwerk einfach softwareseitig über Änderungen am logisch zentralisierten SDN-Controller an die aktuellen Anforderungen angepasst werden kann.

In den letzten Jahren hat deshalb viel Forschung im Bereich von Software-defined Networking stattgefunden. Auch in der Praxis wird Software-defined Networking bereits immer öfter eingesetzt. Besonders große Firmen wie z. B. Google und Amazon haben ein großes Interesse daran, weil sie darin gute Möglichkeiten zur Steigerung der Flexibilität, Effizienz und Verwaltbarkeit ihrer Datencenter sehen. Google hat mit B4 z. B. ein SDN-basiertes WAN zwischen ihren Datencentern erfolgreich im Einsatz [25]. Auch AT&T setzt auf Software-defined Networking und plant bei seinen Netzwerken bis 2020 einen SDN Anteil von über 75% [1].

1.1 Zielsetzung

Bei typischen SDN-Architekturen befindest sich die Kontrolllogik im logischen zentralisierten SDN-Controller. In manchen Fällen ist der klassische dezentrale Ansatz allerdings ausreichend und kann wie in Kapitel 3 noch detaillierter beschrieben sogar Vorteile z. B. bei der Latenz, dem Traffic Overhead oder der Fehlertoleranz bieten [17, 28]. Inzwischen beschäftigen sich deswegen auch immer mehr Arbeiten mit der Kombination lokaler und zentraler Kontrollentscheidungen in SDN-Netzwerken [2, 3, 34].

In dieser Arbeit soll deshalb das Thema lokaler Kontrollentscheidungen genauer betrachtet werden. Insbesondere in Bezug auf den Einsatz lokaler SDN-Controller auf offener Weiterleitungshardware. Das Ziel dabei ist die Vorteile der klassischen dezentralen Netzwerkarchitektur mit denen des logisch zentralisierten Ansatzes des SDN zu kombinieren. Für die einfachere Anwendbarkeit, bessere Plattformunabhängigkeit und Isolation von Erweiterungen für bestehende Switches, soll dabei außerdem die Kapselung über Virtualisierungstechniken wie z. B. Containertechnologie sichergestellt werden. Die Gründe für die Virtualisierung werden ebenfalls in Kapitel 3 noch detaillierter beschrieben.

Im Folgenden soll deshalb zuerst ein Überblick über die zugrundeliegenden Technologien gegeben werden ehe dann mögliche Ansätze zur Umsetzung lokaler Kontrolllogik aufgezeigt werden. Anschließend wird ein Konzept für den gewählten virtualisierten lokalen SDN-Controller-Ansatz erstellt. Darauf aufbauend erfolgt die Umsetzung verschiedener Performance-Tests und die Implementierung einiger Anwendungsszenarien. Abschließend folgt die Evaluation und die Auswertung der Ergebnisse.

1.2 Gliederung

Im Folgenden wird zuerst im Kapitel 2 auf die theoretischen Grundlagen wie Software-defined Networking (SDN) und Virtualisierungstechniken eingegangen. Anschließend folgt in Kapitel 3 die Konzeption. Dazu werden verschiedene Alternativen betrachtet und das Konzept für einen virtualisierten lokalen SDN-Controller ausgearbeitet. Darauf aufbauend wird dann die Auswahl geeigneter Technologieren wie Virtualisierungslösungen und SDN-Controller sowie die Umsetzung von Anwendungsfällen in Kapitel 4 beschrieben. In Kapitel 5 folgt eine umfangreiche Evaluation mit Auswertung der Ergebnisse. Abschließend wird in Kapitel 6 eine kurze Zusammenfassung und ein Ausblick gegeben.

2 Grundlagen

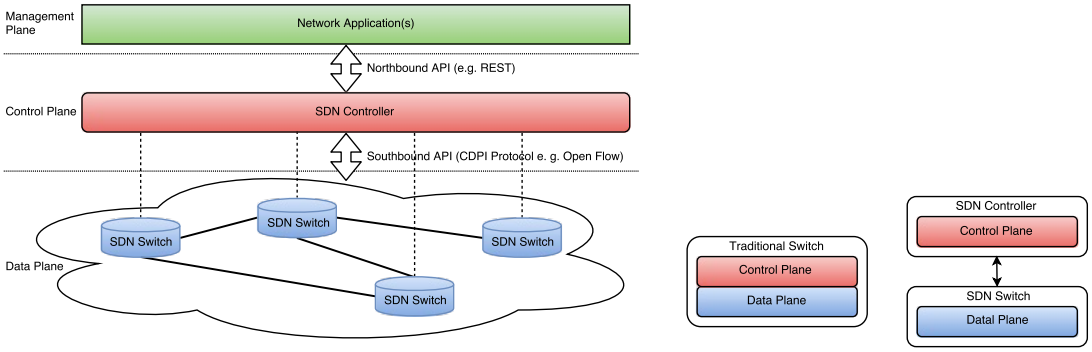
2.1 Software-defined Networking (SDN)

Steigende Anforderungen an Netzwerke haben dazu geführt, dass es für klassische Netzwerkarchitekturen immer schwerer wird diese zu erfüllen [18]. Vor allem mangelnde Flexibilität und aufwendige Konfiguration sind dabei immer wieder genannte Gründe. Im Bereich der Netzwerktechnik ist deswegen Software-defined Networking (SDN), das dynamische, anpassbare, automatisierbare und leicht verwaltbare Netzwerke ermöglichen soll, eines der angesagtesten Themen der letzten Jahre und gewinnt immer mehr an Bedeutung [6].

Der wesentliche Unterschied des Software-defined Networking gegenüber klassischen Netzwerken besteht darin, dass eine strikte Trennung von Control Plane und Data Plane wie in Abbildung 2.1 dargestellt stattfindet was die zentrale Koordination des Netzwerks mit einem logisch zentralisierten SDN-Controller erlaubt [15].

- Die **Data Plane** auch Forwarding Plane genannt ist für das Weiterleiten von Paketen im Netzwerk zuständig. Das Weiterleiten der Pakete geschieht dabei auf Basis von Regeln die von der Control Plane vorgegeben werden.
- Die **Control Plane** entscheidet wohin Pakete weitergeleitet werden und enthält die Logik, Protokolle und Steuerfunktionalität des Netzwerks.

Dies führt zu einen Abstraktion der Data Plane gegenüber höheren Schichten wie der Control Plane oder der noch eine Ebene höher liegenden Management Plane. Dies ermöglicht es wiederum wesentlich einfachere Switches zu bauen, da komplexe Logik und Protokolle nun nicht mehr deren Aufgabe sind. Außerdem erlaubt es, da das Management nun über die Control Plane erfolgt, das Netzwerk leichter zu verwalten und einfacher neue Protokolle zu nutzen. Vor allem da nun eine einfache logische Zentralisierung der Kontrolllogik in einem programmierbaren logisch zentralisierten SDN-Controller möglich wird [15]. Das bietet den Vorteil, dass nun ein logisch zentralisierter SDN-Controller mit globaler Sicht auf das Netzwerk Entscheidungen treffen kann. Entscheidungen auf Basis der globalen Netzwerksicht ermöglichen es z. B. das Netzwerk effizienter auszunutzen als dezentrale Lösungen ohne Sicht auf das komplette Netzwerk. Ein Beispiel dafür ist das Routing mit globaler Netzwerksicht, das es erlaubt bessere Routen zu finden wie der dezentrale Ansatz.



(a) SDN-Architektur (b) Klassischer Switch / SDN-Switch

Abbildung 2.1: Überblick über den Aufbau der SDN-Architektur und die Trennung in Control und Data Plane

Ein klassisches SDN-Netzwerk verfügt also über einen zentralen SDN-Controller der über die Control Plane die SDN-fähigen Switches steuert, die anhand der vom SDN-Controller definierten Regeln Daten auf Ebene der Data Plane weiterleiten.

Die Anfänge des SDN gehen dabei auf Forschungen zum Thema programmierbare Netzwerke, Netzwerkvirtualisierung und Ideen zur Control und Data Plane Separation zurück. Die 2011 gegründete Open Networking Foundation (ONF) [15] nennt als wesentliche Eigenschaften der SDN-Architektur die Hauptmerkmale: direkte Programmierbarkeit, Agilität, zentrales Management, programmatische Konfigurierbarkeit und Herstellerunabhängigkeit durch offene Standards. Damit verbunden ist auch eine verbesserte Automatisierung, schnellere Innovation durch Unabhängigkeit vom Hersteller, verbesserte Zuverlässigkeit und Sicherheit sowie eine besser Kontrolle über das Netzwerk [18, 15].

Die SDN-Idee unterscheidet sich damit deutlich von klassischen Netzwerken in denen viele dezentrale proprietäre Geräte verschiedenste Funktionen implementieren und Weiterentwicklung, Funktionalität, unterstütze Protokolle usw. komplett vom Hersteller abhängen [6]. Klassische Netzwerke haben außerdem das Problem, dass Netzwerkgeräte wie Router, Switch, Middleboxes (z. B. Firewall, Intrusion Detection System, Network Address Translator, Load Balancer), usw. sich dabei nicht nur je nach Aufgabe, sondern oft auch je nach Hersteller und Version deutlich unterscheiden. Das liegt daran, dass die Funktionalität der Geräte meist in proprietärer Software oder Hardware umgesetzt ist. Bei älteren Geräten kommt noch dazu, dass es oft auch keine Updates mehr gibt, so dass man auf einem bestimmten Entwicklungsstand stehen bleibt. Das führt dazu, dass jedes Gerät immer nur bestimmte Protokolle und Konfigurationsschnittstellen in bestimmten Versionen unterstützt. In Bezug auf Flexibilität und einfache Konfigurierbarkeit bietet SDN mit seinem offen und herstellerunabhängigen auf Standards basierendem Ansatz deshalb deutliche Vorteile. Zum Beispiel ist durch den logisch zentralisierten

SDN-Controller, der durch die Trennung von Data und Control Plane ermöglicht wird, es relativ leicht neue Funktionen mit diesem zu implementieren. Eine Anpassung der SDN-Switches ist dabei nicht nötig, da diese einfach nur die dafür nötigen Regeln vom logisch zentralisierten SDN-Controller vorgegeben bekommen.

Abbildung 2.1 (a) zeigt die typische SDN-Architektur, die sich in die Ebenen Data Plane, Control Plane und Management Plane einteilen lässt [28]. Die Control Plane leitet wie bereits beschrieben die Daten weiter und enthält die dafür notwendigen Netzwerkgeräte wie z. B. einen Switch. Darauf aufbauend folgt die Control Plane die über Southbound Protokolle wie z. B. OpenFlow die Geräte der Data Plane steuert und festlegt wie Pakete weitergeleitet werden. Außerdem kommuniziert die Control Plane über ihr Northbound Interface mit den Software Services der Management Plane. Für Netzwerk-Services und deren Anbindung an die Control Plane gibt es bisher keinen Standard. Die Northbound API ist häufig einfach eine Schnittstelle in der Programmiersprache des SDN-Controllers oder eine Representational State Transfer (REST) Schnittstelle. Die Aufgaben der Control Plane werden meist von einem einzelnen zentralen SDN-Controller übernommen auch wenn prinzipiell die Verwendung mehrerer SDN-Controller möglich ist. Mehre SDN-Controller sind vor allem dann interessant, wenn sie nötig sind Ausfallzeiten zu minimieren oder Last auf mehre Rechner zu verteilen. Sie helfen also Skalierbarkeit und Ausfallsicherheit zu erreichen. Wird im Laufe der Arbeit also von einem SDN-Controller bzw. zentralen SDN-Controller gesprochen wäre also auch immer ein nur logisch zentralisierter SDN-Controller denkbar. Die Auswahl an SDN-Controllern die eingesetzt werden können ist groß, eine Übersicht dazu ist in Kapitel 4 zu finden.

Über die Southbound API auch Control to Data-Plane Interface (CDPI) genannt läuft die Kommunikation des SDN-Controllers mit den der Data Plane. Für diesen Einsatzzweck hat sich mit OpenFlow bereits ein sehr gut unterstützter Standard etabliert der große Verbreitung und sehr gute Unterstützung hat. OpenFlow wird dabei oft in Kombination mit einem Managementprotokoll wie OVSDB oder OF-CONFIG eingesetzt. Aber auch andere Ansätze wie ForCES, POF (= Protocol-Oblivious Forwarding) und OpenState kommen als Southbound API in Frage [28]. Auf Ebene der Data Plane befinden sich die SDN-Switches. Dabei gibt es sowohl Hardwarelösungen als auch reine Software Lösungen wie Open vSwitch die zum Beispiel für virtualisierte Umgebungen sehr interessant sind.

2.1.1 OpenFlow (OF)

Als Kommunikationsprotokoll zwischen Control Plane und Data Plane, um das Weiterleiten der Daten zu steuern, hat sich beim SDN der OpenFlow-Standard durchgesetzt. Das OpenFlow-Protokoll erlaubt die direkte Manipulation des Verhaltens der Data Plane, in dem über Regeln genau bestimmt wird was mit eintreffenden Netzwerkpakete auf dem Switch passiert.

Im Dezember 2009 wurde Version 1.0 der OpenFlow-Switch Spezifikation veröffentlicht [15]. Daraufhin folgten weitere Versionen, die verschiedene Neuerungen einführten. Die aktuelle Version ist die durch die Open Networking Foundation veröffentliche OpenFlow Switch Spezifikation Version 1.5 [16]. Die meisten OpenFlow-Switches und SDN-Controller unterstützen bisher aber meist nur ältere Versionen, da sich die neuen Versionen nur langsam durchsetzen.

Zentraler Teil des OpenFlow Standard sind die Flow-Regeln bzw. Einträge für einzelne Paketströme sogenannte Flows. Diese Regeln legen anhand verschiedener Kriterien wie z. B. Quelle oder Ziel eines Pakets fest was mit eingehenden Paketen passiert. Sie werden SDN-Controller über OpenFlow-Nachrichten an den Switch gesendet und von diesem in seine Flow Table aufgenommen. Jeder Eintrag in der Flow Table besteht dabei aus einer Matching-Regel und einer Aktion die ausgeführt wird wenn die Regel zutrifft. Die Matching-Regel spezifiziert also für welche Pakete die dazugehörige Aktion ausgeführt wird. Außerdem sind noch z. B. Counter per-table, per-flow, per-port und per-queue vorgesehen. Für aktuelle OpenFlow-Versionen wurden außerdem noch einige Erweiterungen eingeführt so dass Version 1.5 nun als Felder die Matching-Regel, Priorität, Counter, Action bzw. Instruktionen, Timeout, Cookie und Flags bietet [16].

OpenFlow-Nachrichten starten dabei alle wie in Abbildung 2.2 mit einem gleichen aufgebauten Header (Version, Type, Länge, Transaction Id / xid) auf den der Payload folgt [14]. Es gibt dabei eine Vielzahl an Typen wie z. B. `Hello-`, `FlowMod-`, `PortMod-`, `PacketIn-` `PacketOut-` und `Error`-Nachrichten die alle in der OpenFlow-Spezifikation dokumentiert sind [16]. Die `Hello`-Nachricht z. B. dient dem Verbindungsaufbau und über die `PacketIn`-Nachricht werden Pakete zum SDN-Controller weitergereicht. Über `FlowMod-` und `PortMod`-Nachrichten hingegen konfiguriert der SDN-Controller den SDN-Switch und mit `PacketOut`-Nachrichten kann er Pakete an den SDN-Switch senden die dieser dann weiterleitet. OpenFlow-Nachrichten werden also sowohl vom SDN-Switch zum informieren des SDN-Controllers über relevante Ereignisse als auch vom SDN-Controller zum steuern des Forwarding Verhaltens eines SDN-Switches genutzt.

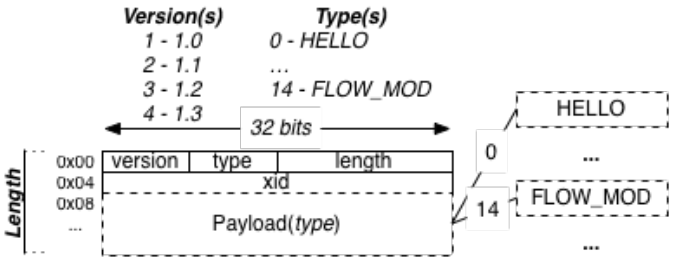


Abbildung 2.2: Struktur eines OpenFlow-Paket [14]

Der Ablauf ist dabei wie folgt. Erreicht ein Paket der Data Plane den OpenFlow-Switch, wird in der Flow Table bzw. meist der ersten von mehreren ein Match mit einer Matching-Regel auch Classifier genannt gesucht, um dann die damit verbundene

Aktion bzw. Instruktionen auszuführen. Am häufigsten wird ein Paket von Switch anhand von Regeln einfach über den passenden Port oder wenn es noch keine Regel gibt an den SDN-Controller weitergeleitet. Das Matching kann dabei in erster Instanz auch, wie im folgenden Abschnitt über SDN-Switches genauer beschrieben, auf einer Anwendungsspezifischen integrierten Schaltung (ASIC) stattfinden ehe dann für größere Flow Tables und komplexere Aktionen das deutlich langsamere Matching mit einer Softwarestatt Hardwareimplementierung stattfindet. Beim Matching auf Softwareebene wird dabei einfach das Paket mit den Regeln der Flow Table bzw. den Flow Tables abgleichen und wenn eine Regel zutrifft die damit verbundene Aktion ausgeführt.

Für die Matching-Regel stehen verschiedene Classifier-Felder zu Verfügung. Bei OpenFlow 1.0 sind dies z. B. Port, Source MAC, Destination MAC, Type, Length, VLAN ID, Priority, Protocol, Source Address, Destination Address, Source Port, Destination Port. Inzwischen werden aber durch neuere OpenFlow-Protokoll-Versionen noch mehr Möglichkeiten geboten die in der jeweiligen zur Version gehörten OpenFlow-Spezifikation definiert sind [14, 16].

Aktionen, die bei neueren OpenFlow-Versionen auch Instruktionen genannt werden, sind zum Beispiel das Verwerfen, Weiterleiten (an SDN-Controller, Port, mehre Ports, Sender, usw.) und Modifizieren von Paketen möglich [16]. Neuere OpenFlow-Versionen bieten auch hier wieder mehr Möglichkeiten wie z. B. eine größere Anzahl Felder von Paketen bzw. Protokollen die modifiziert werden können. Eine weitere Art sind sogenannte Vendor bzw. Experimenter Aktionen und Nachrichten, die für eigene Erweiterungen gedacht sind [14].

Außerdem gibt es auch einige bereits existierende Erweiterungen für OpenFlow wovon die Nicira Extensions die bekannteste und am breitesten unterstützte Erweiterung des OpenFlow Protokolls ist [36]. Die Nicira Extensions erlauben z. B. komplexere Regeln und Aktionen mit mehren Tabellen zu definieren und ermöglichen es auch z. B. eine einfache L2 Switch Umsetzung direkt auf dem SDN-Switch darüber konfigurieren.

Das OpenFlow Protokoll ist somit sehr vielseitig, hat eine gute Dokumentation, ist frei verfügbar und hat viele Unterstützter weshalb es sich inzwischen auch als Standard Southbound API bzw. CDPI-Protokoll durchgesetzt hat. Der Fokus des OpenFlow Protokolls ist dabei rein auf der Konfiguration der Data Plane was dazu führt, dass es häufig in Kombination mit einem zusätzlichen Managementprotokoll eingesetzt wird.

2.1.2 SDN-Switch

SDN-Switches verbinden die einzelnen Geräte im Netzwerk. Neben SDN-fähigen Hardware Switches verschiedener Hersteller gibt es dabei auch reine Softwarelösungen, die vor allem auf virtualisierte Umgebungen ausgerichtet sind. Die SDN-Switches unterscheiden sich dabei wie zu Beginn in Abbildung 2.1 (b) dargestellt deutlich von klassischen Switches da sie sich nur noch um das Weiterleiten anhand von Regeln kümmern die ein SDN-Controller vorgibt. SDN-Switches haben dazu meist einen Port über den sie mit

dem Managementnetzwerk bzw. der Control Plane verbunden sind und ein große Anzahl an Ports auf der Data-Plane-Ebene mit der sie zu anderen Switches und Endgeräten verbunden sind. Das Weiterleiten über die Ports auf der Data-Plane-Ebene wird dabei vom SDN-Controller konfiguriert. Pakete für die keine Regel zur Weiterleitung auf Data-Plane-Ebene existiert werden dabei verworfen oder z. B. über eine Default-Regel, zum Weiterleiten an den SDN-Controller, an den SDN-Controller gesendet, damit dieser den Switch daraufhin passend konfigurieren kann. Für eine genauere Betrachtung ist dabei die Unterteilung in die eigentliche Forwarding Engine, die sich um das Weiterleiten von Paketen kümmert, und den CDPI Agent, der über die Southbound API mit dem SDN-Controller kommuniziert, möglich.

Software Switch / Open vSwitch (OVS)

Als SDN-fähige Software-Switch-Lösung hat sich beim SDN inzwischen Open vSwitch (OVS) [49] durchgesetzt. Open vSwitch ist eine OpenFlow-fähige, für den Produktiveinsatz geeignete Open-Source-Implementierung eines verteilten virtuellen Multilayer Switch. Das Hauptziel des OVS-Projekts ist es einen Switching Stack für Umgebungen mit Virtualisierung anzubieten, der viele Protokolle und Standards unterstützt [49]. Auch SDN-fähige Switch-Hardware, die Open vSwitch als Basis nutzt ist inzwischen verfügbar. Der Code von Open vSwitch steht unter der Apache License 2.0 und neben der aktuellsten Version (derzeit Version 2.6) wird auch immer noch eine Long Term Support (LTS) Version aktiv unterstützt [49].

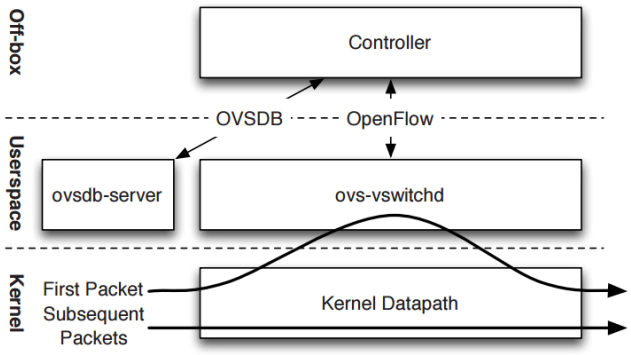


Abbildung 2.3: Open vSwitch Architektur [40]

Die OVS-Architektur wie in Abbildung 2.3 dargestellt besteht aus den Hauptkomponenten `ovs-vswitchd`, `ovsdb-server` und dem Kernel-Datapath-Modul. Das Kernel-Datapath-Modul ist dabei für das sehr schnelle Weiterleiten von Paketen zuständig. Die `ovs-vswitchd`-Instanz hingegen für die Kommunikation mit dem SDN-Controller über OpenFlow und dem `ovsdb-server` sowie das Matching neuer Pakete mit den kompletten Flow Tables, die zu groß bzw. komplex für das Kernel-Datapath-Modul sein können. Der `ovsdb-server` stellt die Datenbank bereit in der alle zum Betrieb nötigen Daten

gespeichert sind. Die Datenbank interagiert dabei intern direkt mit der Switching- und Matching-Logik und kann von außen über das Open vSwitch Database Management Protocol (OVSDB)[39] konfiguriert werden. Außerdem ist das Abrufen von Statusinformationen ebenfalls über das OVSDB-Protokoll möglich. Mit `ovs-vsctl` für die Konfiguration der OVS-Datenbank (`ovsdb` bzw. `ovsdb-server`) und `ovs-ofctl` zur Überwachung und Administration von OpenFlow-Switches sowie `ovs-dpctl` und `ovs-appctl` stehen außerdem komfortable Kommandozeilen Tools zur Verfügung. Gerade für eine schnelle Konfiguration, Fehlersuche oder zu Tests sind diese sehr hilfreich.

Eingehende Pakete werden beim Open vSwitch zuerst in dem Kernel-Datapath-Modul entgegengenommen und falls es dort kein Match gibt an `ovs-vswitchd` zum Matching mit den User Space Flow Tables weitergereicht. Im Fall eines Matchs in einer der Tabellen wird die damit verbundene Aktion ausgeführt bzw. das Paket entsprechend weitergeleitet. Gab es in keiner Flow Table ein Match wird je nach Konfiguration des Switches das Paket verworfen oder an den SDN-Controller weitergeleitet.

Alternative Software-Switches sind z. B. ofsoftswitch13 und LINC-Switch die allerdings lang nicht so bekannt und auch nicht so weitverbreitet sind [28].

Hardware Switch

SDN-fähige Hardware-Switches gibt es von verschiedenen Herstellern. Je nach Anforderungen gibt es sie mit verschieden Anzahlen an Ports und verschieden leistungsfähiger Hardware (CPU, RAM, ASIC, usw.). Dabei ist vor allem auch interessant wie viele Flow Table-Einträge in Hardware also mit einer ASIC verglichen werden können, da dabei die Performance erheblich besser ist wie bei einem Vergleich auf Softwareebene. Hauptgrund für die sehr gute Performance beim Matching ist Ternary Content Addressable Memory (TCAM). Darüber ist es möglich das Matching sehr schnell und für alle in der Hardware vorgehalten Regeln parallel durchzuführen. Der Nachteil ist, dass der TCAM Speicher, da er sehr teuer ist, meist nicht sonderlich groß ist und die Anzahl an Einträgen für die ein Matching auf Hardwareebene möglich ist somit sehr begrenzt ist.

SDN-Switches werden häufig mit einem Linux basierten Betriebssystem betrieben. Inzwischen gibt es deshalb verschiedene proprietäre aber auch einige Open-Source-Linux-Betriebssysteme die speziell für den Einsatz auf einem Switch vorgesehen sind. Bekannte Vertreter dieser Art sind z. B. PicOS (Pica8)[23], Open Network Linux[29] und OpenSwitch (HP) [38]. Der Aufbau dieser Betriebssysteme ist dabei ähnlich. Ein angepasstes Linux, die Ansteuerung der ASIC und der Softwareteil des Switches bilden die Hauptkomponenten. Die Verbindung zum SDN-Controller erfolgt dabei meist mit OpenFlow und wird häufig noch mit einem zusätzlichen Managementprotokoll wie OVSDB oder OF-CONF ergänzt [28]. Manche Hersteller wie PicOS setzten dabei wieder einen modifizieren und auf die Hardware angepassten Software-Switch wie Open vSwitch ein um ihre Hardware SDN-tauglich zu machen [23]. Andere implementieren die SDN-Unterstützung

oder Teile bzw. Komponenten davon komplett selbst. Teilweise wird auch eine Abstraktionsschicht wie z. B. die von Broadcom entwickelte OpenFlow Data Plane Abstraction (OF-DPA) [9] eingesetzt. Diese kommt zum Beispiel beim Indigo OpenFlow Agent auf Open Network Linux zum Einsatz.

Auf einem Hardware Switch läuft also ein speziell angepasstes Betriebssystem und er verfügt auf Hardwareebene über eine große Anzahl Ports und eine ASIC. Auf Softwareebene hingegen läuft eine Forwarding Engine die direkt oder über eine Abstraktionsschicht auf die ASIC zugreift und über einen CDPI Agent mit dem er zur Control Plane verbunden ist.

Für diese Arbeit ist dabei vor allem wichtig, dass der Switch nicht nur auf die offenen SDN-Standards aufbaut sondern auch insgesamt ein sehr offenes System bietet das Modifikationen erlaubt. Die Zielplattform sind also insbesondere sogenannte White Box Switches.

2.2 Virtualisierungstechnologien

Virtualisierung ist ein weit verbreitetes Konzept, das vor allem für den Betrieb von Servern unerlässlich geworden ist. Die Anfänge der Virtualisierung gehen dabei mehr als 50 Jahre zurück. Damals führte IBM Servervirtualisierung auf seinen Großrechnern ein, um durch virtuelle Maschinen mehre Einzelbenutzersysteme zur besseren Auslastung parallel auszuführen [33]. Inzwischen gibt es eine Vielzahl an Virtualisierungslösungen die sich im Wesentlichen in die drei Kategorien Virtuelle Maschinen, Container und Unikernel einteilen lassen. Eine Übersicht über diese im Folgenden noch genauer beschriebenen Typen zeigt Abbildung 2.4.

Die Vorteile von Virtualisierung sind dabei vielfältig. Im Serverumfeld ist das Ziel vor allem Hardware besser auszulasten und so weniger Hardware und damit weniger Strom und Platz zu benötigen. Aber auch eine schnellere Server-Provisionierung und das einfache Einrichten und Nutzen von Testumgebungen ist ein wichtiger Faktor. Außerdem bietet Virtualisierung die Möglichkeit Software, die auf einem neuen System nicht mehr laufen würde, durch die Virtualisierung eines älteren Systems weiter zu nutzten. Da Virtualisierung auch zur Isolation beiträgt besteht hier außerdem der Vorteil, dass auch neue oder experimentelle Software relativ gefahrlos in einer virtualisierten Umgebung ausgeführt werden kann. Auch effektivere Administration und Wartung sowie einfachere Backups und Wiederherstellungen und somit höhere Verfügbarkeit sind ein Vorteil. Vor allem das leichte Hinzufügen, Entfernen und Verschieben virtualisierter Instanzen ist ein großer Vorteil um optimal auf die aktuelle Last reagieren zu können. Es gibt aber nicht nur Vorteile, denn Virtualisierung kostet immer, selbst bei guter Unterstützung durch die Hardware, zumindest ein wenig an Performance.

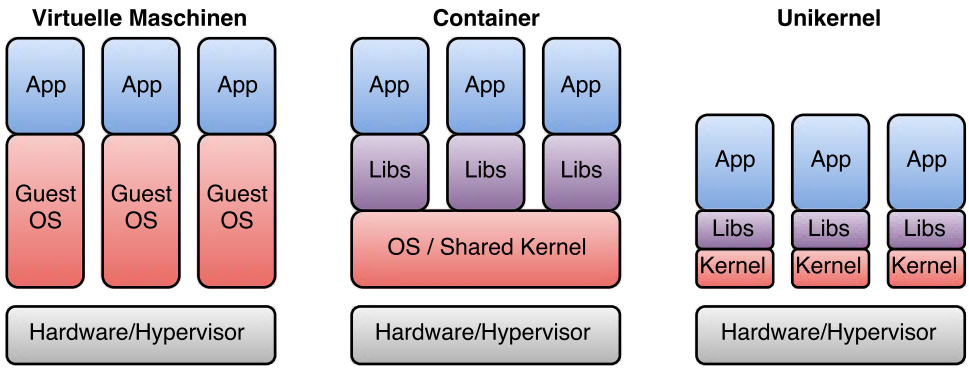


Abbildung 2.4: Vergleich von Virtualisierungstechnologien

2.2.1 Virtuelle Maschinen (VM)

Klassische virtuelle Maschinen (VM) existieren schon am längsten. Sie werden von einem Hypervisor ausgeführt der entweder direkt auf der Hardware läuft oder als Software wie z. B. VirtualBox auf einem Betriebssystem ausgeführt wird. Außerdem besteht die Möglichkeit zur Verwaltung der VMs. Das bedeutet, es stehen Funktionen zum Anlegen, Starten, Stoppen, oder Zuweisen von Ressourcen zur Verfügung. Hardwarevirtualisierung also Software zur Abstraktion der Hardware ist dabei die Grundlage für die eigentliche virtuelle Maschine auf der dann ein Gastbetriebssystem ausgeführt wird. Dies ermöglicht es, dass die reale Hardware von verschiebenden virtuellen Maschinen gleichzeitig genutzt werden kann. Die Gastbetriebssysteme die auf der VM laufen bemerken dabei keinen Unterschied zu exklusiven Ausführung auf realer Hardware, so dass keine Anpassungen der Software nötig sind.

Die Vorteile davon sind eine effizientere Auslastung der Hardware und die Isolation der Betriebssysteme und der darauf ausgeführten Programme gegenüber. Virtuelle Maschinen emulieren dabei immer eine bestimmte Hardware und sind von der Software darauf unabhängig. Es laufen somit alle Betriebssysteme darauf die mit der emulierten Hardware die von der Virtuellen Maschine zur Verfügung gestellt kompatibel sind. Somit läuft auch jegliche Software die auf einem dieser Betriebssysteme läuft in einer VM. Der Nachteil ist allerdings ein großer Overhead der viel Performance kostet, da absolut alles virtualisiert ist. Man hat dafür aber eine sehr gute Isolation die sogar soweit geht, dass eine andere Hardwarearchitektur wie die des Systems auf dem die Virtuelle Maschine läuft emuliert werden kann.

2.2.2 Container

Der Preis für die Vorteile einer Virtualisierung mit VMs ist ein sehr großer Overhead, da immer ein komplettes Betriebsamstem auf virtueller Hardware ausgeführt werden muss.

Dies hat zur Entwicklung von leicht gewichtigeren Lösungen geführt, den sogenannten Containern.

Container basieren auf der Idee Anwendungen auf dem Kernel eines Hostsystems auszuführen aber die Prozesse und Bibliotheken dennoch zu isolieren, indem Kernel-Ressourcen virtualisiert und gegeneinander abgeschottet werden [30]. Sie bauen dazu auf Linux Techniken wie Kernel Namespaces und `cgroups` auf. Für Anwendungen scheint es im Wesentlichen aber so als würden sie, wie bei VMs, auf einer komplett eigenen Betriebssystem Instanz laufen. Bereits 2005 stand Google vor dem Problem, dass VMs ihr Anforderungen nicht zufriedenstellend erfüllten und suchte deshalb nach Alternativen. Dies führte zu der Entwicklung von `cgroups` die dann Anfang 2008 in den Linux Kernel aufgenommen wurden [4]. Darauf aufbauend erschienen dann im August 2008 die Linux Containers (LXC). Diese haben das Ziel eine Umgebung zu schaffen, die so nah wie möglich an einer extra Linux Installation ist, ohne einen separaten Kernel zu benötigen [30]. Den großen Durchbruch haben Container aber erst mit dem Boom des Cloud Computing und dem Erscheinen des sehr einfach verwendbaren Docker im Jahr 2013 geschafft [4].

Ihr Vorteil ist der geringe Overhead und somit gute Performance durch das nutzen eines gemeinsamen Kernels. Die Isolation ist dafür geringer als z. B. bei den komplett getrennten VMs doch es existieren Wege diese durch zusätzliche Mechanismen wieder zu erhöhen. Container haben allerdings auch eine Einschränkung gegenüber VMs denn es können nur Betriebssysteme mit gleichem Kernel virtualisiert werden.

2.2.3 Unikernels

Ein weiterer Ansatz sich von klassischen virtuellen Maschinen zu lösen bieten Unikernels. Diese ermöglichen es Anwendungen zusammen mit den benötigten Systemkomponenten zu kompilieren, so dass sie sich direkt auf einem Hypervisor oder Hardware ausführen lassen [32, 48]. Ziel ist es, ein minimalistisches Image aus Anwendung und Betriebssystemdiensten in Form von Libraries zu erzeugen. Dieses Image braucht dann kein Betriebssystem mehr als Zwischenschicht, da alles enthalten ist um direkt ausgeführt zu werden. Das Betriebssystem liegt dazu in Form von einzelnen Bibliotheken vor, von denen die zum Ausführen der Anwendung Benötigten zusammen mit der Anwendung den Unikernel bilden. Zum Beispiel ein Webserver als Unikernel braucht dann Betriebssystemkomponenten wie ein Netzwerkstack.

Der Vorteil ist, man erhält ein Image mit den benötigten Betriebssystemteilen und der Anwendung, das sehr leichtgewichtig ist und sich direkt auf Hardware oder einem Hypervisor ausführen lässt. Der Nachteil ist, dass die Anwendungen dafür häufig extra für den Unikernel compiliert bzw. sogar speziell angepasst werden müssen. Dies kann in manchen Fällen auch Anpassungen am Code erfordern oder ein Problem darstellen, wenn verwendete Bibliotheken nichts als Sourcecode vorliegen. Auch unterstützt nicht jeder Unikernel jede Programmiersprache, so dass bei der Wahl einer passenden Unikernel Lösung viel mehr zu beachten ist als bei Container oder VMs, die ein vollwertiges Betriebssystem

als Umgebung bereitstellen. Unikernel bieten somit eine gute Isolation und verursachen erheblich weniger Overhead als VMs da kein komplettes Betriebssystem, sondern nur die benötigten Teile ausgeführt werden müssen. Eine bekannte Unikernel-Implementierung ist zum Beispiel Rumprun das in Kapitel 4 genauer vorgestellt wird.

2.3 Literatur / Related Work

In den vorherigen Abschnitten wurden bereits die Grundlagen und der aktuelle Stand auf dem diese Arbeit aufbaut aufgezeigt. Im Folgenden soll nun noch ein Blick auf einige Arbeiten geworfen werden die sich ebenfalls mit dem Thema SDN und lokale Entscheidungen beschäftigen.

Diese Arbeit folgt dabei auf eine vorhergehende Masterbarbeit zu "Local Data Plane Event Handling in Software-defined Networking" [12] die sich ebenfalls mit dem Thema lokale Entscheidungen beschäftigt hat. Der Fokus dabei lag aber auf der direkten Integration in den Code des Switches während in dieser Arbeit nun der Fokus auf dem Einsatz eines virtualisierten lokalen SDN-Controllers liegt.

Ein weit fortgeschrittenes Projekt das sich mit lokalen Aktionen auf SDN-Switches beschäftigt ist das in "OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch" vorgestellte OpenState SDN Projekt [2, 5]. Von Bifulco et al. wurde unter dem Titel "Improving SDN with InSPired Switches" außerdem ein solcher Ansatz vorgestellt der mit der InSP API es erlaubt Operationen im Switch zu definieren, die Pakete senden bzw. Aktionen wie das Verändern oder Weiterleiten eines Pakets auslösen [3]. Ein nochmal etwas anderen Ansatz beschreiben Mekky et al. in "Application-aware data plane processing in SDN" wo Module auf Ebene der User Space Tables des Switches eingeklinkt werden [34].

Diese Arbeiten unterscheiden sich aber alle dahingehend von dieser Arbeit, dass sie eher direkte Anpassungen im Switch umsetzen. Für diese Arbeit aber steht die Idee eines zusätzlichen lokalen und mittels Virtualisierung isolierten SDN-Controllers im Vordergrund.

3 Konzeption

In diesem Kapitel wird das Konzept, das als Grundlage für die Implementierung und Evaluation dient, beschrieben. Dazu wird zuerst auf die Motivation und mögliche Lösungsansätze eingegangen, ehe das eigentliche Konzept für den gewählten Ansatz folgt.

3.1 Motivation

Im Folgenden wird die Motivation zum Entwurf des nachfolgenden Konzepts genauer betrachtet. Dazu werden zuerst die Gründe für den Einsatz lokaler Logik bei Software-defined Networking genauer dargelegt und Beispiele für praktische Anwendungsfälle vorgestellt. Anschließend folgen die Gründe für den Einsatz von Virtualisierung.

3.1.1 Lokale Logik

In herkömmlichen Netzwerken ist jedes Netzwerkgerät wie z. B. ein Switch eine eigenständige Einheit und muss individuell konfiguriert werden. Außerdem können Entscheidungen nur auf Basis einer begrenzten lokalen Sicht auf das Netzwerk getroffen werden.

Software-defined Networking, das auf dem Grundprinzip einer Trennung von Control und Data Plane aufbaut, bietet mit dem Einsatz eines logisch zentralisierten SDN-Controllers daher, wie in Kapitel 2 beschrieben, viele Vorteile. Ein SDN-Switch als Element der Data Plane z. B. ist beim Software-defined Networking nur noch für das Weiterleiten von Paketen nach festen Regeln, die ihm ein logisch zentralisierter SDN-Controller vorgibt, zuständig. Der Vorteil davon ist eine einfache Konfiguration und Steuerung aller Switches über einen logisch zentralisierten SDN-Controller, der außerdem eine globale Sicht auf das Netzwerk hat und von Anwendungen bereitgestellte Informationen nutzen kann um das Netzwerk optimal auszunutzen.

Da nun aber der Switch keine Entscheidungen mehr trifft, müssen alle Ereignisse im folgenden auch Events genannt, die Entscheidungen erfordern, an den logisch zentralisierten SDN-Controller weitergereicht werden. Dieser kann mit seiner Antwort den Switch dann passend konfigurieren. Das bedeutet aber eine zusätzliche Latenz durch die Kommunikation zum logisch zentralisierten SDN-Controller und wieder zurück. Viele Events können aber auch effizient lokal verarbeitet werden, weil z. B. kein globales Wissen nötig ist. Die Delegation dieser einfach lokal durchführbaren Entscheidungen an den Switch

bietet somit die Möglichkeit die Effizienz zu steigern, da die Latenz zum logisch zentralisierten SDN-Controller wegfällt und die lokale Auswertung keine Nachteile mitbringt. Im SDN-Umfeld wird dies meist als „Delegation of control“ bezeichnet, da nach dem SDN Prinzip der SDN-Controller entscheidet welche Aufgaben er an den SDN-Switch delegiert [17]. Lokale Entscheidungen laufen dabei nicht immer vollständig unabhängig vom SDN-Controller, da wichtige Entscheidungen, Statusinformationen oder ähnliches teilweise dennoch zum SDN-Controller geschickt werden müssen, auch wenn bereits eine lokale Reaktion darauf ausgeführt wurde.

Im Folgenden sollen deswegen verschiedene Möglichkeiten für die Umsetzung lokaler Entscheidungen aufgezeigt werden. Anschließend soll eine Lösung entwickelt und evaluiert werden, die die Vorteile eines logisch zentralisierten SDN-Controllers mit denen lokaler Entscheidungen kombiniert.

Mögliche Vorteile einer SDN-Architektur in der lokale Entscheidungen getroffen werden können sind z. B.: [vgl. 28, 17]

- Geringere Latenz, da das Warten auf eine Antwort für die an den logisch zentralisierten SDN-Controller geschickte Anfrage entfällt.
- Weniger Traffic Overhead, da weniger Pakete an den logisch zentralisierten SDN-Controller gesendet bzw. weitergeleitet werden müssen.
- Geringere Last für den logisch zentralisierten SDN-Controller, da weniger Anfragen von den Switches kommen.
- Erprobte Protokolle aus herkömmlichen Netzwerken können auf die gleiche Art verwendet werden.
- Bessere Fehlertoleranz und Ausfallsicherheit, da rein lokale Funktionen auch bei Ausfall des logisch zentralisierten SDN-Controller erhalten bleiben.
- In Switch-Hardware vorhanden Funktionen können besser genutzt werden (Counter, Timer, ...).

3.1.2 Anwendungsfelder

Im folgenden werden einige Beispiele für Anwendungsszenarien vorgestellt die sich besonders gut für den Einsatz lokaler Kontrolllogik eignen: [vgl. 28, 17]

- **MAC Learning / Simple Switch:**
  Grundfunktion klassischer L2-Switches: Lernen der Kombinationen aus MAC-Adresse und Port, um Pakete, die am Switch ankommen, gezielt weiterleiten zu können. Wenn zur Ziel-MAC-Adresse eines ankommenden Pakets der Port bekannt ist wird nur über diesen Port weitergeleitet, ansonsten findet ein Flooding über alle Ports statt. Globales Wissen ist dafür nicht nötig. Die Umsetzung kann somit auch ohne Nachteile lokal erfolgen.

- **Fast Failover:**
  Merken alternativer Wege zu anderen Konten. Falls die Verbindung zu einem Knoten zusammenbricht, kann nun direkt der alternative Weg genommen werden. Somit muss nicht nach einem neuen Weg gesucht werden bzw. der logisch zentralisierte SDN-Controller nach einem gefragt werden. Pakete können somit erheblich schneller wieder weitergeleitet werden was dazu führt, dass die Verbindung nicht so lange unterbrochen ist.

- **Threshold Crossing:**
  Überwachung von Traffic bzw. Performance um den logisch zentralisierten SDN-Controller bei Problemen bzw. dem erreichen bestimmter Werte zu Informieren. Auch hier liegen alle Informationen lokal auf dem Switch vor. Alle Zwischenwerte zum logisch zentralisierten SDN-Controller zusenden würde viel Overhead verursachen, der somit eingespart werden kann.

- **Multicasting:**
  Beim Multicast werden Nachrichten zu einer Gruppe von Teilnehmern geschickt. Die Verwaltung davon kann dabei lokal erfolgen und muss nicht komplett vom logisch zentralisierten SDN-Controller durchgeführt werden. Kommen neue Teilenehmer dazu oder verlassen welche die Gruppe kann somit einfach lokal das nun Weiterleiten oder nicht mehr Weiterleiten von Paketen der Gruppe über die davon betroffen Ausgangsports durch Änderungen der Forwarding-Regeln angepasst werden. Der Weg zum logisch zentralisierten SDN-Controller kann somit eingespart werden.

- **Port Knocking:**
  Beim Port Knocking werden Pakete in bestimmter Reihenfolge an bestimmte Ports geschickt. Über eine State Machine wird dabei auf die korrekte Port-Knocking-Sequenz geprüft, um bei einer korrekten Sequenz eine Aktion, wie z. B. einen Port in der Firewall für den Client zu öffnen, auszuführen. Das Überprüfen der Sequenz kann dabei komplett lokal ablaufen und es muss nicht jedes Paket an den logisch zentralisierten SDN-Controller geschickt werden.

Lokale Entscheidungen sind also immer dann von Vorteil, wenn Aufgaben auszuführen sind für die lokales Wissen ausreichend ist. Also die Anzahl an nötigen Nachrichten zum zentraler SDN-Controller reduziert werden kann bzw. zumindest das Warten auf eine Antwort entfällt.

3.1.3 Virtualisierung

Die Zielplattform für die Integration lokaler Entscheidungen ist ein SDN-fähiger Hardware-Switch, betrieben mit einem dafür optimierten Betriebssystem. Außerdem kommt meist eine ASIC für sehr schnelles Forwarding zum Einsatz. Je nach Hersteller und der Offenheit des Systems ist dabei die Installation anderer Betriebssysteme möglich oder auch nicht. Doch selbst wenn es möglich ist stellt sich noch immer die Frage welche

Betriebssysteme überhaupt in Frage kommen, da für die volle Funktionalität die Kompatibilität mit ASIC sichergestellt sein muss. Es ist somit nicht jeder Switch mit jedem Betriebssystem kompatibel und daher anzunehmen, dass nicht nur ein offenes Betriebssystem, sondern auch zum Teil proprietäre oder alternative offene Betriebssysteme auf den SDN-Switches im Netzwerks im Einsatz sind. Genauso können verschiedene CDPI Agents darauf zum Einsatz kommen. Außerdem ist es möglich, dass verschiedene Versionen eines Betriebssystems im Einsatz sind, weil nicht alle Hardware mit der neusten Version kompatibel ist. Des Weiteren ist es oft so, dass solche Betriebssysteme, um möglichst schlank und effizient zu sein, nicht alle Funktionen besitzen die man erwartet.

Um mit möglichst vielen Zielplattformen zurecht zu kommen wäre es also von großen Vorteil nicht zu sehr vom Betriebssystem eines speziellen Switches abzuhängen. Des Weiteren wäre auch von Vorteil, wenn die Integration lokaler Logik möglichst isoliert und unabhängig vom Rest des Systems ist, um nicht durch unerwünschte Nebeneffekte die eigentliche Switching Funktionalität zu beeinträchtigen.

Zum erreichen dieser Ziele bzw. Vorteile bietet sich der Einsatz geeigneter Virtualisierungstechniken an. Der Grund ist sie ermöglichen die Isolation der lokalen Kontrolllogik von Switch. Das bietet den Vorteil dass die einzige Abhängigkeit vom Switch bzw. dessen Betriebssystem die Verfügbarkeit der Virtualisierungslösung darauf ist. Besonderheiten der Zielplattform und Abhängigkeiten der Kontrolllogik führen so zu keinen Problem mehr. Gleichzeitig hilft Virtualisierung die Sicherheit zu verbessern, da die Kontrolllogik so kein Schaden auf dem Hostsystem anrichten kann. Des weiteren hilft die Virtualisierung die Stabilität des Systems zu verbessern in dem z. B. der Anteil an Ressourcen wie z. B. der CPU für die Kontrolllogik limitiert wird, so dass zu rechenintensive Funktionen oder Bugs die Forwarding Funktionalität des Switches nicht beeinträchtigen. Ein anderer Vorteil sind nützliche Managment Funktionen die Virtualisierungslösungen anbieten und gerade in Kombination mit der durch die Isolation erreichten relativ guten Unabhängigkeit von Hostsystem das Deployment erheblich vereinfachen.

Virtualisierung bietet also viele Vorteile und hilft lokale Kontrolllogik möglichst Plattformunabhängig und ohne unerwünschte Seiteneffekte auf Betriebssystemen für SDN-Switches lauffähig zu bekommen. Da Virtualisierung aber auch immer einen gewissen Overhead bedeutet stellt sich die Frage nach der geeignetsten Virtualisierungstechnologie. Um eine möglichst geeignete Virtualisierungslösung zu finden, werden deshalb die verfügbaren Techniken dazu in den folgenden Kapiteln noch genauer untersucht.

3.2 Klassische SDN-Systemarchitektur

Als Grundlage für die folgenden Teile der Arbeit ist es wichtig die wesentlichen Elemente des Aufbaus eines klassischen SDN-Netzwerks zu definieren. Aus diesen Grund wird im Folgenden noch etwas genauer der Aufbau einer klassischen SDN-Systemarchitektur beschrieben.

Das Wichtigste dabei sind die verschieden Hardwarebausteine bzw. Geräte, die z. B. über Kabel oder Funk verbunden sind, und aus denen das Netzwerk besteht. Im Folgenden werden all diese Geräte einfach abstrakt als Device d einer Menge D von Devices bezeichnet. Da diese irgendwie mit dem Netzwerk verbunden sein müssen, z. B. über ein Kabel, verfügen sie alle über mindestens einen Netzwerkport dp der Menge DP aller Ports auf Ebene der Data Plane. Während die meisten Geräte in einem Netzwerk wie z. B. Server, Desktop-PCs, Smartphones und IoT Devices in der Regel nur einen Port haben verfügen Switches oder Router, die als Verbindungsglieder dienen, in der Regel über eine größere Anzahl an Ports sowieso einen Management Port cp aus der Menge CP. Relevant sind an dieser Stelle die SDN-fähigen Switches s der Menge S über die alle Devices auf Ebene der Data Plane durch eine Netzwerktopologie $T_{DataPlane}$ miteinander verbunden sind. Diese müssen für den korrekten Betrieb außerdem mit einem logisch zentralisierten SDN-Controller verbunden sein. Der Einfachheit halber ist in diesem Fall ein einziger zentraler SDN-Controller (cc) also ein Device mit einer SDN-Controller-Software c angenommen. Der zentrale SDN-Controller und die SDN-Switches sind dazu über eine Netzwerktopologie $T_{ControlPlane}$ auf Ebene der Control Plane verbunden. Auf der Seite des Switches geschieht diese Anbindung über den mit der Forwarding Engine verbunden CDPI Agent a des Switches der über den Management Port eine Verbindung zum zentralen SDN-Controller aufbaut.

Formal lassen sich die wichtigsten Elemente eines klassischen SDN-Netzwerks somit folgendermaßen beschreiben:

Data Plane Ports:	$DP := \{dp_1, ..., dp_n\}$
Control Plane Ports:	$CP := \{cp_0, ..., cp_m\}$
Data Plane topology function:	$T_{DataPlane} \in DP \rightarrow DP$
Control Plane topology function:	$T_{ControlPlane} \in CP \rightarrow CP$
Devices:	$D := \{d_0, d1 = (dp_1, ..., dp_k), ..., d_z = (dp_l, ..., dp_n)\}$
SDN Controller Software:	$C := \{c_0\}$
Forwarding Engine and CDPI Agent:	$A := \{a_1, ..., a_m\}$
SDN-Switches	$S := \{s_1 = (d_1, cp_1, a_1), ..., s_m = (d_m, cp_m, a_m)\}$
Central SDN-Controller:	$cc := (d_0, cp_0, c_0)$

Tabelle 3.1: Formaler Aufbau eines klassischen SDN-Netzwerks

Für das Konzept sind dabei vor allem der Aufbau und die Interaktion von den SDN Switches und dem logisch zentralisierten SDN-Controller relevant. In Abbildung 3.1 ist deshalb der klassische Aufbau eines SDN-Switch und eines zentralen SDN-Controllers einschließlich der Verbindungen grafisch dargestellt. Als logisch zentralisierter SDN-Controller dient dabei meist wie Abgebildet ein Server auf dem eine SDN-Controller Instanz läuft, denkbar wären aber auch mehre. Dieser zentralisierte oder zumindest logisch zentralisierte SDN-Controller kommuniziert mit allen SDN-Switches im Netzwerk über ein CDPI-Protokoll, wie z. B. das OpenFlow-Protokoll, um diese zu steuern und

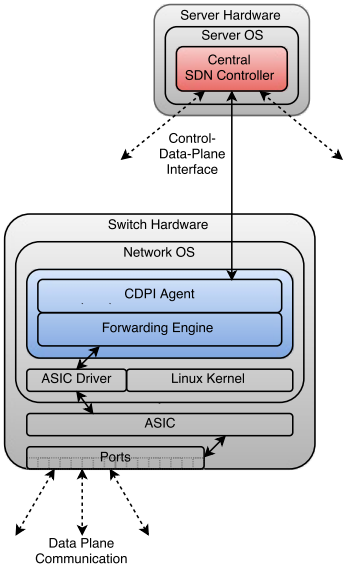


Abbildung 3.1: Klassische SDN-Architektur

Informationen über das Netzwerk zu bekommen. Die SDN-Switches von denen es beliebig viele geben kann, laufen wie schon in Kapitel 2 beschrieben, meist mit einem speziell angepassten Betriebssystem und verfügen über eine große Anzahl schneller Ports. Meist verfügen sie für hohe Performance auch über einen ASIC. Diese wird von der Forwarding Engine, in der Abbildung blau hervorgehoben, gesteuert. Über den auf dem Switch laufen CDPI Agent, meist ein Open Flow Agent, besteht außerdem eine Verbindung der Forwarding Engine zur Control Plane bzw. dem logisch zentralisierten SDN-Controller. Außerdem ist der Switch noch auf Ebene der Data Plane mit anderen Devices wie z. B. Switches oder direkt mit Endgeräten verbunden.

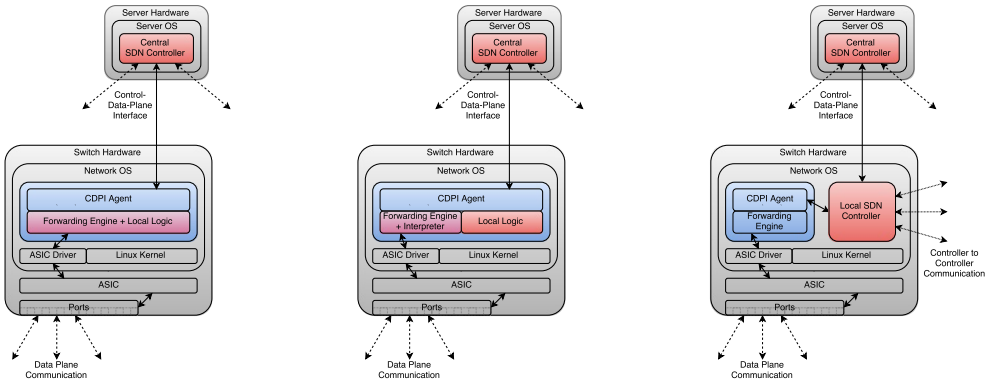
3.3 Umsetzungsvariationen

Für die Integration von lokalen Entscheidungen ist es nötig die klassische SDN-Struktur anzupassen. Dabei sind verschieden Ansätze denkbar.

Ein sehr entscheidender Punkt dabei ist bereits der Punkt, an dem die Integration lokaler Logik eingebaut wird. Die erste Möglichkeit wäre sofern vorhanden direkt im Kernel Modul bzw. der Kernel Flow Table der Forwarding Engine. Dies ist aber nur schwer und kompliziert umzusetzen bietet dafür aber eine sehr hohe Performance. Die zweite Möglichkeit wäre sich im Userspace Modul bzw. in einer User Space Flow Table der Forwarding Engine einzuhängen. Dies ist bereits leichter umzusetzen und bietet immer noch sehr gute Performance. Eine weitere Möglichkeit ist, statt die Forwarding Engine direkt zu verändern beim CDPI Agent anzusetzen. Dies erlaubt einen sehr modularen

Aufbau und erfordert wenig Modifikationen an bestehenden Komponenten was eine einfache Umsetzung erlaubt. Der Nachteil ist eine geringere Performance und zusätzlicher Aufwand für das Packen und Entpacken von CDPI-Nachrichten.

Im folgenden werde nun einige Möglichkeiten, das Ziel lokaler Entscheidungen umzusetzen, genauer beschrieben.



(a) Variante 1: Direkte Modifikation der Forwarding Engine

(b) Variante 2: Integration eines Interpreter in die Forwarding Engine

(c) Variante 3: Lokaler SDN-Controller auf dem Switch

Abbildung 3.2: Überblick SDN-Architekturen

3.3.1 Variante 1: Direkte Modifikation der Forwarding Engine

Die einfachste Möglichkeit wäre es, einfach direkt den Sourcecode der Forwarding Engine um die benötigte Funktionalität zu erweitern. Das liegt daran, dass dort alle Daten direkt vorliegen und somit einfach auf die bestehenden Flow Tables und Forwarding Techniken aufgebaut werden kann. Die nötige Logik kann so direkt an geeigneter Stelle eingebaut werden und bietet den Vorteil, dass sie völlig frei von unnötigen Overhead und daher auch sehr performant ist. Der Nachteil einen solchen Ansatzes ist, dass es praktisch keine Kapselung gibt und Änderungen immer Änderungen am Forwarding Engine Code zur Folge haben. Neue lokale Logik kann außerdem nicht zur Laufzeit des Switches hinzugefügt werden, so dass für jede Änderung der Netzwerkverkehr unterbrochen werden muss. Außerdem besteht das Problem, dass der Code für neue Forwarding Engine Versionen oder andere Forwarding Engines immer neu angepasst werden muss. Die Tiefe der Integration ist dabei variabel und es sind verschiedene Varianten denkbar.

a) Integration auf/nach Ebene der Kernel Flow Table

b) Integration auf/nach Ebene der User Space Flow Tables

Wobei erste Variante wie in der Einleitung dies Abschnitts angemerkt, zwar sehr schnell wäre aber nur schwer umzusetzen, da die Möglichkeiten im Kernel Modul recht eingeschränkt sind. Abbildung 3.2 (a) zeigt diese Variante mit einer direkten Modifikation der Forwarding Engine.

3.3.2 Variante 2: Integration eines Interpreter in die Forwarding Engine

Eine mögliche Optimierung der Variante 1 wäre, wie in Abbildung 3.2 (b) zu sehen, die lokale Logik nicht komplett hart in den Code der Forwarding Engine zu integrieren. Stattdessen kann die Forwarding Engine beispielsweise um einen Interpreter für Module mit lokalen Funktionen erweitert werden. Dazu kann z. B. ein einbaubarer Interpreter wie Lua [21] verwendet werden. Der Vorteil ist dann, dass für neue Funktionen keine Änderungen am Code des Switches mehr nötig sind und im Idealfall auch Änderungen zur Laufzeit möglich sind. Als Nachteil bleibt die mangelnde Kapselung, so dass für jede neue Version oder andere Forwarding Engine die Modifikationen extra wieder eingebaut werden müssen. Außerdem erzeugt der Interpreter zusätzlichen Overhead.

3.3.3 Variante 3: Lokaler SDN-Controller auf dem Switch

Eine weitere Lösung für die Integration lokaler Entscheidungen ist es, die Forwarding Engine selbst gar nicht zu modifizieren, sondern bei der CDPI-Schnittstelle anzusetzen. Der Vorteil davon ist eine sehr saubere Trennung der lokalen Logik von der Forwarding Engine, da wie wenn ein zentraler SDN-Controller eingesetzt wird, der CDPI Agent nun als Schnittstelle dient und die Kommunikation über ein erprobtes CDPI-Protokoll stattfinden kann. Außerdem bietet es den Vorteil, dass die Lösung von der Forwarding Engine unabhängig ist, da nur ein kompatibler CDPI Agent existieren muss. Die Software, die sich mit dem CDPI Agent verbindet, muss deshalb nur einmal geschrieben werden, um eine große Anzahl von Software- und Hardware-Switches abzudecken. Möglich ist so ein Ansatz in dem ein SDN-Controller lokal auf dem Switch ausgeführt wird, zu dem sich der CDPI Agent anstelle des zentralen SDN-Controller verbindet, so dass dieser seinen Platz einnimmt. Der lokale SDN-Controller wiederum hat die Möglichkeit sich dann noch zu einem anderen SDN-Controller wie z. B. dem logisch zentralisierten SDN-Controller zu verbinden, so dass der lokale SDN-Controller einfach dazwischengeschaltet ist. Abbildung 3.2 (c) zeigt dieses Konzept. Der Nachteil dieser Variante ist, dass mehr Overhead entsteht, da nun zwei Programme laufen müssen, die zwar lokal aber dennoch über CDPI-Nachrichten, die erstellt und dann wieder geparst werden müssen, kommunizieren. Da die lokal gesendeten CDPI-Protokoll-Nachrichten also entscheidend für den Overhead sind, sind auch hier Varianten denkbar.

a) Nutzung eines etablierten CDPI-Protokolls wie z. B. OpenFlow

b) Nutzung eines weniger bekannten oder eigenen leicht gewichtigeren CDPI-Protokoll

c) Einbau einer speziellen direkten Schnittstelle in die Forwarding Engine

Auch die Art der Kommunikation spielt dabei eine Rolle, da bestehende CDPI Agents über eine Netzwerkschnittstelle kommunizieren und deshalb meist TCP zum Einsatz kommt während bei Variante c) auch noch schnellere Techniken zur Interprozesskommunikation einsetzbar sind. Dies würde aber wiederum die Unabhängigkeit von Forwarding Engine und CDPI Agent beeinträchtigen und auch dort Anpassungen erfordern.

3.3.4 Variante 4: Lokaler virtualisierter SDN-Controller auf dem Switch

Eine weitere Modifikation der Variante 3 ist wie in der Abbildung 3.3 (b) zu sehen den lokalen SDN-Controller zusätzlich zu virtualisieren also z. B. in einen Container zu verpacken. Das bietet den Vorteil, dass dieser noch besser von Betriebssystem des Switches isoliert ist. Dies hat dann auch Vorteile z. B. beim Deployment und der Interoperabilität. Der Nachteil ist natürlich zusätzlicher Oberhead für die Virtualisierungslösung. Die Wahl geeigneter Virtualisierungstechniken wie z. B. Container oder Unikernels statt klassischen VMs ist dabei dafür entscheidend wie viel Performance verloren geht.

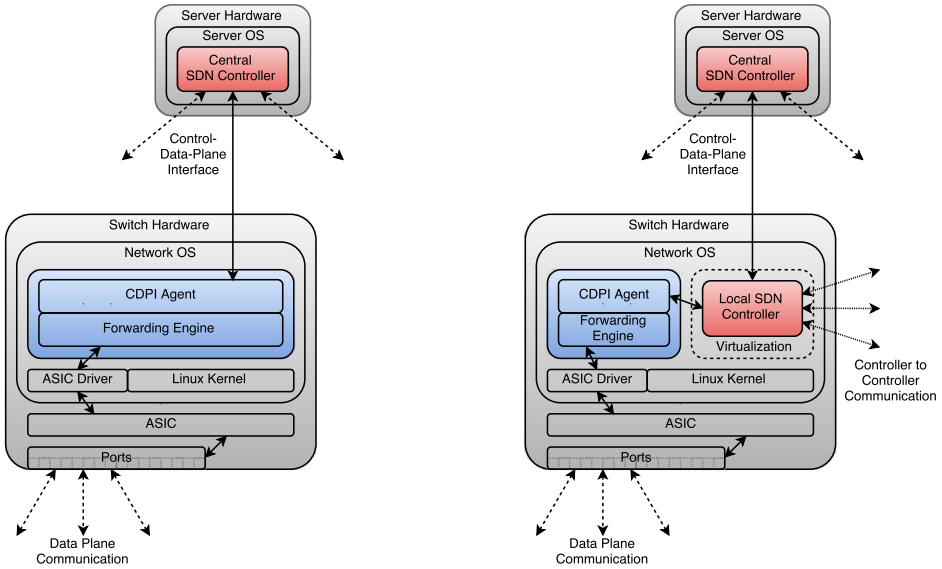
3.3.5 Fazit Umsetzungsvarianten

Die hier beschriebenen Varianten bieten verschiedene Vor- und Nachteile. Für diese Arbeit liegt dabei im Folgenden der Fokus auf Variante 4 mit einem lokalen und virtualisierten SDN-Controller. Der Grund dafür sind die gute Kapselung und Isolation vom Rest des Switches, größere Unabhängigkeit vom OS und der Forwarding Engine des Switches, sowie die klare Architektur bzw. Struktur der Lösung. Für die Integration im Code der Forwarding Engine existiert sowieso bereits eine vorausgegangene Arbeit[12]. Der einzige Nachteil dabei ist der Overhead für die Kommunikation über die CDPI-Schnittstelle und den Container. Hier wird die Evaluation später zeigen wie groß diese Nachteile sind und ob die bessere Trennung und der klarere Aufbau es wert sind diese in Kauf zu nehmen.

Abbildung 3.3 stellt den klassischen Aufbau dem neuen Konzept gegenüber, so dass die Unterschiede zum Konzept des lokalen virtualisierten SDN-Controller klar zu sehen sind. Auf den Switch läuft nun ein lokaler SDN-Controller innerhalb einer Virtualisierungslösung. Abgesehen von der Kommunikation über ein Netzwerkinterface ist dieser somit vom Rest der Software gut isoliert. Ein zentraler SDN-Controller kann dabei dem lokalen SDN-Controller weiterhin übergeordnet sein. Es wäre nun aber auch eine direkte SDN-Controller zu SDN-Controller Kommunikation z. B. in einer hierarchischen Struktur denkbar anstatt oder in Kombination mit einem klassischen zentralen SDN-Controller.

3.4 Systemmodel

Aufbauend auf das zuvor beschriebene klassische SDN Systemmodell wird im Folgenden das eigentliche Konzept zur Umsetzung der Variante mit einem lokalen virtualisierten



(a) Klassische SDN-Architektur

(b) Variante 4: Lokaler virtualisierter SDN-Controller auf dem Switch

Abbildung 3.3: Vergleich SDN-Architekturen

SDN-Controller für lokale Entscheidungen aus dem vorherigen Abschnitt vorgestellt. Die Idee dabei ist zur Isolation eine, z. B. mit einem Container, virtualisierte lokale SDN-Controller-Instanz auf dem Switch auszuführen, wie in Abbildung 3.3 (b) zu sehen ist. Das Ziel davon ist es die Vorteile einer bereits lokalen Verarbeitung bestimmter Events zu nutzen, um eine höhere Effizienz zu erreichen.

3.4.1 Systemarchitektur mit lokalem SDN-Controller

Die klassische SDN Variante wurde bereits im Abschnitt klassische SDN-Systemarchitektur beschrieben. Mit der Integration von lokalen SDN-Controllern ergeben sich aber einige Änderungen. Im Folgenden ist deswegen die Struktur nochmal grob unter Einbeziehung des lokalen SDN-Controller-Ansatzes formal definiert.

Der Unterschied zur klassischen SDN-Systemarchitektur ist also, dass auf jedem SDN-Switch s nun auch ein SDN-Controller c existiert. Dadurch ist nicht mehr zwingend nötig, dass der zentrale SDN-Controller cc als eigenständige Einheit existiert. Es wäre genauso denkbar, dass die lokalen SDN-Controller direkt miteinander kommunizieren. Dazu könnte z. B. eine hierarchische Struktur, wie ein Baum, mit den SDN-Controllern der Switches erschaffen werden.

Für die Umsetzung der virtualisierten lokalen SDN-Controller Variante ist es also nötig einen SDN-Controller auf den SDN-Switches auszuführen. Die Hauptaufgabe eines SDN-

Data Plane Ports:	$DP := \{dp_1, ..., dp_n\}$
Control Plane Ports:	$CP := \{cp_0, ..., cp_m\}$
Data Plane topology function:	$T_{DataPlane} \in DP \rightarrow DP$
Control Plane topology function:	$T_{ControlPlane} \in CP \rightarrow CP$
Devices:	$D := \{d_0, d1 = (dp_1, ..., dp_k), ..., d_z = (dp_l, ..., dp_n)\}$
(Virtualized) SDN Controller Software:	$C := \{c_0, ..., c_m\}$
Forwarding Engine and CDPI Agent:	$A := \{a_1, ..., a_m\}$
SDN-Switches	$S := \{s_1 = (d_1, cp_1, a_1, c_1), ..., s_m = (d_m, cp_m, a_m, c_m)\}$
Optional central SDN-Controller:	$cc := (d_0, cp_0, c_0)$

Tabelle 3.2: Formaler Aufbau eines SDN-Netzwerks mit lokalem SDN-Controller-Ansatz

Switches ist es dabei weiterhin Pakete, die über die Ports auf Data-Plane-Ebene eingehen, über andere Ports wieder weiterzuleiten. Klassische Switches machen das lokal nach einfachen Regeln. SDN-Switches hingegen leiten nach vom logisch zentralisierten SDN-Controller vorgegeben Regeln weiter. Dies führt aber zu höheren Latenzen, weshalb für diesen Ansatz ein lokaler SDN-Controller, neben Forwarding Engine und CDPI Agent a, auf dem Switch dazu dient Entscheidungen, die lokal durchführbar sind, auch lokal und ohne die Latenz zum logisch zentralisierten SDN-Controller zu verarbeiten.

Der SDN-Switch ist dabei weiterhin über den Managementnetzwerkport cp mit der Control Plane verbunden und verfügt über eine größere Anzahl an Ports dp auf Data-Plane-Ebene. Die Forwarding Engine mit dazugehörigem CDPI Agent a ist dabei aber nicht mehr wie bei klassischen Ansatz über den CDPI Agent direkt mit dem logisch zentralisierten SDN-Controller verbunden sondern mit dem lokalen SDN-Controller. Der Grund ist, gut lokal entscheidbare Probleme können so nun sofort vom lokalen SDN-Controller gelöst werden. Nicht lokal durchführbare Kontrollentscheidungen hingegen werden einfach vom lokalen SDN-Controller, der nun die Aufgabe der Verbindung zum logisch zentralisierten SDN-Controller übernimmt, zum logisch zentralisierten SDN-Controller weitergeleitet.

Auf dem SDN-Switch muss also ein lokaler SDN-Controller laufen. Dazu muss als Grundlage auf dem Switch natürlich erst einmal ein Betriebssystem laufen. Da darauf auch der lokale SDN-Controller läuft, ist dabei ein halbwegs offenes System nötig, das es erlaubt einen SDN-Controller darauf auszuführen. Hier kommt der Vorteil des Einsatzes einer Virtualisierungslösung zum Tragen, da diese den SDN-Controller isoliert und vom Rest des Systems weitgehend unabhängig macht. Für das Weiterleiten der Daten existiert dabei auch meist eine ASIC, die über eine dazu gehörende API angesprochen werden kann. Je nach Switch bestehen auch große Unterschiede bei der Leistungsfähigkeit des Switches in Bezug auf Speicher, Rechenpower und ASIC. Alle auf dem Switch ausgeführten Anwendungen sollten deswegen möglichst leichtgewichtig sein. Das ist auch vor allem für den lokalen SDN-Controller relevant, da Switch-Hardware meist nicht so leistungsfähig

wie ein Server ist, auf dem sonst ein klassischer SDN-Controller läuft.

Da nun auf jedem Switch ein SDN-Controller läuft wäre an dieser Stelle wie schon zuvor angemerkt auch eine Abweichung vom Einsatz eines extra zentralen SDN-Controllers denkbar, weshalb dieser in der formalen Definition auch als optional gekennzeichnet ist. Es wäre z. B. genauso möglich all diese SDN-Controller in einer Hierarchie anzuordnen und nicht lokal verarbeitete Probleme dann immer zum nächst höheren Switch zu eskalieren. Dies ergibt zwar eine komplexere Struktur und manche Nachrichten müssen eventuell mehre Stufen durchlaufen ehe sie verarbeitet werden, verteilt aber dafür die Last im Netzwerk dafür besser. Das kann gerade in großen Netzwerken, wo eine größere Anzahl an Switches an einem einzigen zentraleren SDN-Controller hängen, von Vorteil sein.

3.4.2 Kommunikation

Für die Umsetzung relevant ist dabei vor allem auch der Ansatz, den eigentlichen Switch bzw. die Forwarding Engine und den CDPI Agent mit dem lokalen SDN-Controller zu verbinden bzw. allgemein den lokalen SDN-Controller zu integrieren.

Kommunikation zwischen Data und Control Plane

Für die Kommunikation zwischen dem klassischen SDN-Switch-Teil und dem neuen lokalen SDN-Controller stehen verschiedene Möglichkeiten zur Verfügung.

Die nötige Kommunikation zwischen CDPI Agent und lokalem SDN-Controller kann z. B. klassisch über CDPI-Protokolle wie OpenFlow stattfinden. Im Idealfall ist dabei dann für den CDPI Agent kein wesentlicher Unterschied zum herkömmlichen Software-defined Networking mit nur einem zentralen SDN-Controller zu erkennen. Dies ist möglich, da nur die Verbindung zum logisch zentralisierten SDN-Controller zu einer Verbindung zum lokalen SDN-Controller, geändert wird. Somit sind auch keine Anpassungen am bestehenden Code notwendig.

Änderungen können aber nötig werden, sollte der Overhead beim Einsatz von erprobten und komplexeren CDPI-Protokollen zu groß sein und wie bei Variante 3 b) und c) im vorherigen Abschnitt beschrieben, ein eigenes Protokoll oder sogar eine direkte Interprozesskommunikation gewünscht sein. Letzteres ist mit dem Einsatz von Virtualisierungstechnologie aber vermutlich nur schwer bzw. nicht kombinierbar und würde auch wieder die Idee der Isolation zum Teil verletzen.

Es bietet sich somit der Einsatz eines klassischen CDPI-Protokolls als Schnittstelle an. Vor allem da es Anpassungen an erprobten Komponenten einspart und es ermöglicht auf bewährte Protokolle zu setzen. Alternativen wie komplett eigene Protokolle oder gar eine direkte Schnittstelle machen daher nur Sinn, wenn sie für die Performance nötig

sind. Im Kapitel 5 findet deswegen eine Analyse der Performance im Zusammenhang mit OpenFlow statt, um zu sehen wie viel der Einsatz davon an Performance kostet.

SDN-Controller zu SDN-Controller Kommunikation

Die Kommunikation auf Control-Plane-Ebene zu einem übergeordneten SDN-Controller kann ebenfalls weiter über ein CDPI-Protokoll wie OpenFlow stattfinden. Das hat den Vorteil, dass ein Switch mit lokalem SDN-Controller nach außen hin zu einem klassischen SDN-Netzwerk kompatibel bleibt und z. B. lokale Logik eben vom zentralen SDN-Controller über spezielle Nachrichten aktiviert und deaktiviert werden kann. Ohne Aktivierung der Features würde der lokale SDN-Controller dann z. B. die Pakete nur durchreichen. Aktiviert der übergeordnete SDN-Controller über spezielle Nachrichten aber lokale Features kann der lokale SDN-Controller seine Vorteile nutzen. Es wäre aber genauso denkbar für solche Switches ein völlig neues Protokoll zu nutzen, das genau auf diesen Anwendungsfall zugeschnitten ist.

3.4.3 Funktionsweise

Im Folgenden wird für beide Kommunikationswege ein klassisches CDPI-Protokoll wie OpenFlow angenommen. Für ein besseres Verständnis der Funktionsweise und Abläufe, vor allem in Bezug auf den lokalen SDN-Controller, wird diese nun genauer vorgestellt. Abbildung 3.4 zeigt dazu wie das Zusammenspiel zwischen lokalem SDN-Controller und Switch dann aussieht.

Die Forwarding Engine mit dem CDPI Agent über den sie gesteuert wird ist dabei wie bei klassischen SDN-Switches für alle Aktivitäten auf Ebene der Data Plane zuständig. Neu ist aber der lokale SDN-Controller der ebenfalls auf dem Betriebssystem des Switches läuft. Jegliche Kommunikation zwischen Control Plane und CDPI Agent läuft nun über den lokalen SDN-Controller der für den CDPI Agent den einzigen Zugangspunkt zur Control Plane darstellt. Die große Änderung gegenüber dem klassischen SDN ist also, dass es einen SDN-Controller (untere Hälfte in Abbildung 3.4) gibt der auf dem Switch läuft und der CDPI Agent sich zu diesem, statt dem logisch zentralisierten SDN-Controller, verbindet.

Der Ablauf, wenn ein Paket auf Ebene der Data Plane an einem Port ankommt ist dann wie folgt und auch gut auf Abbildung 3.4 zu sehen. Als erstes findet ein Matching auf bestehende Regeln mit der ASIC statt, die genau für diese Aufgabe optimiert, am leichtesten mit großen Traffic Mengen umgehen kann. In der ASIC findet dazu ein sehr schneller und effizienter Abgleich auf Hardwarebasis mit den Flow Regeln der ASIC statt, wie bereits in Kapitel 2 beschrieben wurde. Ist bereits hier ein Weiterleiten möglich ist dies ideal. Die Anzahl an Einträgen in der Flow Table der ASIC ist aber begrenzt und manche komplexere Aktionen können dort auch nicht direkt verarbeitet werden.

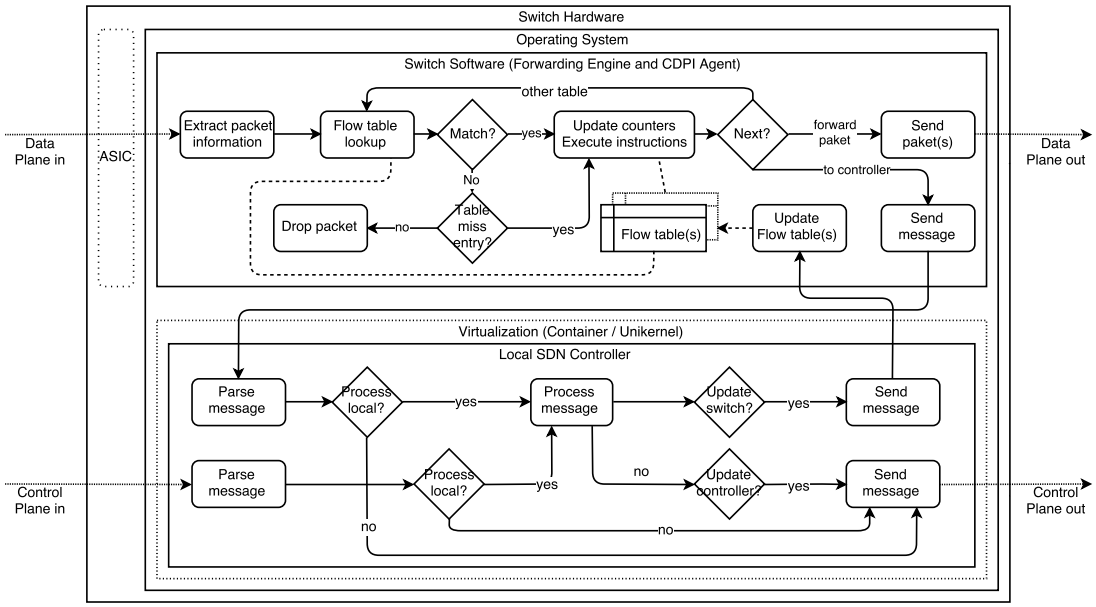


Abbildung 3.4: Abläufe und Interaktionen auf dem SDN-Switch

Ist kein direktes Forwarding mit der ASIC möglich, weil für den Flow keine Regel in der ASIC existiert, findet das weitere Matching auf Softwareebene der Forwarding Engine statt. Das Matching mit Kernel und User Space Flow Tables ist zwar langsamer als mit der ASIC doch dafür sind größere Flow Tables möglich. Im nächsten Schritt werden dafür dann zuerst die fürs Flow Table Matching benötigte Informationen ausgelesen. Dann findet ein Lookup in der ersten Flow Table statt. Bei einem Match werden die damit verbundenen Aktionen ausführt. Gibt es kein Match wird überprüft, ob es als Table Miss Entry eine auszuführende Aktion gibt. Falls auch das nicht der Fall ist wird das Paket verworfen. Falls das Paket nicht verworfen wurde werden die damit verbunden Aktionen ausgeführt. Das kann ein Aktualisieren von Statusdaten wie z. B. ein Paket Counter sein und/oder das Ausführen einer damit verbundenen Aktion wie z. B. das Verändern der Header des Pakets. Anschließend kann abhängig von der Action ein Match in einer weiteren Tabelle gesucht werden, das Paket über einen oder mehrere Ports weitergeleitet werden oder in eine CDPI-Nachricht verpackt zum SDN-Controller geschickt werden. In diesem neuen Modell wird diese CDPI-Nachricht dann innerhalb des Switch verschickt was sehr schnell und unabhängig von der Latenz zum zentralen SDN-Controller ist.

Kommt ein vom CDPI Agent geschicktes Paket am lokalen SDN-Controller an, wird dieses entpackt und geprüft ob es lokal verarbeitet werden kann. Ist das der Fall wird es lokal verarbeitet oder andernfalls an den logisch zentralisierten bzw. übergeordneten SDN-Controller weitergeleitet. Verarbeitet der SDN-Controller ein Paket lokal muss er sofern nötig anschließend den Switch also die Forwading Engine, in dem er eine Nachricht an den CDPI Agent sendet, aktualisieren. War die Nachricht vom CDPI Agent und wurde

lokal verarbeitet kann es außerdem auch noch nötig sein den logisch zentralisierteren SDN-Controller mit einer Nachricht zu informieren. In diesem Fall würde also auch noch eine Nachricht an den logisch zentralisierten bzw. überordneten SDN-Controller geschickt.

Kommt ein Paket auf Ebene der Control Plane an wird es entpackt und verarbeitet. Sofern nötig kann dann der Switch bzw. die Forwading Engine wieder über den CDPI Agent informiert werden und/oder eine Antwortnachricht zum SDN-Controller geschickt werden. Die nötigen Aktionen hängen dabei davon ab ob der übergeordnete SDN-Controller nur Informationen abruft, von sich aus mit Anweisungen ins Weiterleiteverhalten eingreift oder es eine Antwort auf eine eigene Anfrage ist. Interaktion mit dem zentralen oder übergeordneten SDN-Controller ist dabei vor allem dann nötig wenn eine Entscheidung nicht lokal verarbeitet werden kann, Statusinformation verarbeitet werden sollen oder mit globaler Sicht das Netzwerk optimiert wird.

3.4.4 Modifikation, Konfiguration und Erweiterbarkeit

Ein wichtiger Punkt ist auch die Modifikation, Konfiguration und Erweiterbarkeit. Der gewählte Ansatz mit einem lokalen virtualisierten SDN-Controller bietet dafür verschiedene Möglichkeiten. Ein großer Vorteil ist dabei, dass die Kommunikation über eine klare Schnittstelle wie z. B. OpenFlow als CDPI-Protokoll stattfindet. Auf dem Switch wird also nur eine Virtualisierungslösung ausgeführt und über das CDPI-Protokoll mit dem darin laufenden SDN-Controller kommuniziert. Das führt zu einer nur losen und klar definierten Verbindung mit dem SDN-Controller.

Änderungen am SDN-Controller oder der Umstieg auf einen anderen SDN-Controller können also leicht erfolgen. Es muss nur der Container oder Unikernel auf dem Switch ersetzt und neu gestartet werden, was im Prinzip auch automatisiert werden kann. Gegenüber einer Lösung ohne den Einsatz von Virtualisierung besteht aber der Overhead des Starten der Virtualisierungslösung. Von Vorteil ist dafür, dass sich aber nichts an den Abhängigkeiten gegenüber dem Hostsystem ändert, da der SDN-Controller gut isoliert ist. Dieser Vorteil überwiegt vor allem dann, wenn nicht alle Switches im Netzwerk mit dem gleichen Betriebssystem laufen und der SDN-Controller sonst keine einheitliche Umgebung in der er läuft hätte. Außerdem wäre es denkbar, da die alte und neue Version ja durch die Virtualisierungslösung isoliert sind, die neue Version erst zu starten an diese zu übergeben und anschließend die alte zu beenden. Der Nachteil eines langsameren neustarten kann somit vermieden werden und sogar ein Vorteil daraus werden.

Es sind aber nicht nur Updates die den ganzen SDN-Controller ersetzen denkbar. Wie auch bei einem nicht virtualisierten lokalen SDN-Controller kann dieser Befehle von außen erhalten, so dass dieser Punkt unabhängig von Einsatz von Virtualisierungstechnik ist. Dazu sind spezielle Nachrichten nötig die der SDN-Controller versteht. Bei den meisten CDPI-Protokollen sind eigene Protokollerweiterungen aber sogar schon vorgesehen z. B. kann wenn OpenFlow eingesetzt wird über Vendor- bzw. Experimenter-Nachrichten

der Austausch von Informationen erfolgen für die im Protokoll noch keine eigenen Nachrichten Typen vorgesehen sind. Es wäre somit ohne Probleme möglich, wenn die SDN-Controller hierarchisch strukturiert sind oder mit einem logisch zentralisierten SDN-Controller kommunizieren auf diese Art Informationen auszutauschen. So kann z. B. von zentraler Stelle bestimmt werden was lokal verarbeitet werden soll und was nicht oder die Möglichkeit gegeben werden von zentraler Stelle Einfluss darauf zu nehmen wie der lokale SDN-Controller seine Entscheidungen fällt. Die Möglichkeiten dabei sind vielfältig und vermutlich stark vom geplanten Anwendungsfall abhängig.

Es ist alles ohne Weiteres möglich auch im Betrieb über spezielle Nachrichten Module mit lokaler Logik zu aktivieren oder zu deaktivieren. Außerdem ist auch der Aufwand für das Verteilen neuer SDN-Controller Versionen aufgrund der Virtualisierung nur sehr gering. Es muss nur sichergestellt werden, dass bei Veränderungen zurück in einen Standard Status gesprungen wird oder der aktuelle korrekt übergeben wird um nicht in die Gefahr inkonsistenter Zustände zu laufen.

3.4.5 Life Cycle und Konsistenz

Um den SDN-Controller problemlos zu integrieren sollte die Ausführung des virtualisierten SDN-Controllers automatisch nach dem Starten des Betriebssystems auf dem Switch erfolgen. Das kann z. B. über ein Skript geschehen das den SDN-Controller bzw. dessen Container oder Unikernel startet. Voraussetzung ist natürlich, dass jeder Switch initial einmal für den Einsatz der Virtualisierungslösung konfiguriert wurde also z. B. LXC, Docker oder QEMU/KVM drauf eingereicht wurde.

Sobald der lokale SDN-Controller sich dann mit dem CDPI Agent und seinem übergeordneten SDN-Controller verbunden hat ist der Switch einsatzbereit. Der einzige Unterschied gegenüber dem klassischen Ansatz ist der dazwischen geschaltete lokale SDN-Controller. Für die lokalen Features bietet sich dabei an, um damit verbundene Probleme und Inkonsistenzen zu vermeiden, dass diese standardmäßig deaktiviert sind. Um diese zu nutzen muss somit der lokale SDN-Controller sich erst zum übergeordneten SDN-Controller verbinden, damit dieser die lokalen Features im lokalen SDN-Controller aktivieren kann.

Neue Versionen könnten dabei immer leicht vor dem Start abgerufen und aktiviert werden. Es wäre aber auch wie zuvor beschrieben theoretisch möglich neue SDN-Controller Versionen zur Laufzeit einzuspielen. Dabei müsste aber eine korrekte Übergabe des aktuellen Status oder ein Zurücksetzen in einen konsistenten Ausgangsstatus erfolgen.

Beim Beenden sollte der CDPI Agent die Verbindung als erster beenden, so dass der SDN-Controller noch in der Lage ist diese Information an einen übergeordneten SDN-Controller weiterzugeben. Die Abhängigkeiten sind aber durch die, Protokoll basierte, Trennung an dieser Stelle sehr gering.

Probleme mit verschiedenen Versionen des lokalen SDN-Controllers im Netzwerk die durchaus auftreten könnten, wenn man die Switches nach und nach auf eine neue lokale SDN-Controller Version aktualisiert, können dabei z. B. damit umgangen werden in dem der logisch zentralisierte oder in der Hierarchie höchste SDN-Controller neue Funktionen erst dann über Nachrichten aktiviert, wenn alle lokalen SDN-Controller damit zurechtkommen. Somit arbeitet alles wie bereits erprobt bis alle lokale SDN-Controller bereit sind.

Fällt der übergeordnete zentrale SDN-Controller aus steht erstmal, wie im klassischen SDN-Netzwerk, die korrekte Funktionalität nicht mehr zu Verfügung. Da nun aber auf jeden Switch ein SDN-Controller läuft der Events lokal verarbeiten kann, kann man zur Verbesserung der Fehlertoleranz bzw. Ausfallsicherheit in einen eingeschränkten Betriebsmodus wechseln. Dieser würde zumindest die Grundfunktionalität mit Hilfe lokaler Logik erhalten was die Verfügbarkeit des Netzwerks verbessert. Sind die SDN-Controller was bei diesem Ansatz ja möglich ist, aber untereinander z. B. hierarchisch verbunden, kann auch versucht werden nun zu einem anderen SDN-Controller zu verbinden um den Ausgefallenen bis er wieder funktioniert zu umgehen.

3.5 Anforderungen an Umsetzung und Evaluation

Für eine bestmögliche Umsetzung des nun detailliert beschriebenen Ansatzes eines lokalen und virtualisierten SDN-Controllers ist neben der eigentlichen Umsetzung auch eine Evaluation und passende Auswahl eingesetzter Technologien nötig, weshalb im Folgenden die Anforderungen an eine Umsetzung zusammengefasst werden.

Offene Weiterleitungshardware (Open Switching Hardware) bildet die Grundlage für die Umsetzung des lokalen SDN-Controller-Konzepts dieser Arbeit, da ein möglichst offenes System das erweiterbar ist für diesen Ansatz nötig ist. Dazu gehört natürlich auch ein Netzwerkbetriebssystem das darauf läuft und offen genug ist um neben seiner eigentlichen Paket Switching Aufgabe einen virtualisierten SDN-Controller auszuführen. Da die Rechenleistung und der Speicher auf der Hardware des Switch begrenzt sind ist es dabei wichtig, dass sowohl Virtualisierungstechnologie als auch SDN-Controller möglichst leichtgewichtig sind. Auch die Kommunikation zwischen lokalem SDN-Controller und Switch sollte möglichst effizient sein. OpenFlow ist dabei der Standard, doch das Verpacken und Entpacken der OpenFlow Nachrichten erzeugt Overhead, der ebenfalls evaluiert werden sollte um sicherzustellen, dass eine Umsetzung damit sinnvoll möglich ist oder ob andere Lösungen zu bevorzugen sind. Damit ergeben sich die folgenden Arbeitsschritte:

- Umsetzung des Konzepts mit geeigneten Technologien
  - Evaluation der/des OpenFlow Message Kosten/Overhead, um Eignung für den geplanten Anwendungsfall sicherzustellen bzw. Alternativen suchen zu können.

  - Evaluation geeigneter SDN-Controller Software um eine leichtgewichtige Lösung die alle nötigen Funktionen bietet und auf dem Switch eingesetzt werden kann zu finden.
  - Evaluation geeigneter Virtualisierungstechnologie um einen leichtgewichtigen Container oder Unikernel zu finden der sich zum virtualisieren des lokalen SDN-Controller eignet.
  - Wahl eines geeigneten Netzwerkbetriebssystem
- Implementierung von Anwendungsfällen
- Evaluation des gesamten Konzepts bzw. der Anwendungsfälle

Zur Umsetzung ist also ein genaues Betrachten der eingesetzten Technologien nötig ehe darauf aufbauend die eigentliche Umsetzung und Evaluation des Konzepts im Ganzen stattfinden kann. Während die einzelnen Evaluationen der verschiedenen Technologien dabei vor allem Teilaspekte untersuchen, ist über die Anwendungsszenarien die Evaluation des gesamten Aufbaus geplant. Dazu wird der Ansatz eines lokalen SDN-Controllers zusammen mit einigen, der im Abschnitt Anwendungsfelder beschrieben, Szenarien im Folgenden umgesetzt, um Anwendungsszenarios für die dann folgende Evaluation zu haben.

4 Umsetzung

Im vorangegangen Teil dieser Arbeit wurde die Idee für eine SDN-Architektur, in der auch lokale Entscheidungen auf Switches stattfinden, vorgestellt und ein Konzept für die Umsetzung erstellt. Die wesentliche Idee des dabei entstandenen Konzepts ist der Einsatz eines lokalen SDN-Controllers auf dem Switch der es ermöglicht lokale Entscheidungen zu treffen. Außerdem wurde zur Isolation des SDN-Controllers vom restlichen System der Einsatz von Virtualisierungstechniken festgelegt. Im Folgenden wird nun zuerst die vorgegebene Hardware- und Softwareumgebung beschrieben auf der die Umsetzung stattfindet. Anschließend wird die eigentliche Umsetzung dieses Konzepts die als Grundlage für die Evaluation im darauffolgenden Kapitel dient beschrieben. Dazu wird auf die verschiedenen dafür in Frage kommenden existierenden SDN-Controller eingegangen ehe die verschiedenen Virtualisierungsmöglichkeiten beschrieben werden. Anschließend folgt der Teil zur eigentlichen Umsetzung und Implementierung der Anwendungsszenarien für die Evaluation.

4.1 Hardware- und Softwareumgebung

Im Folgenden wird die bei der Umsetzung und Evaluation genutzte Hardware- und Softwareumgebung kurz vorgestellt. Damit ist die Evaluationen leichter nachzuvollziehen und Einschränkungen und Besonderheiten von Anfang an klar definiert.

Das SDN Hardware Testbed besteht aus einem Switch (`vssdn2-sw`) und zwei Endsystemen (`vssdn2-1` und `vssdn2-2`). Der Switch vom Modell "Edgecore AS5712-54X" [10] ist dabei der wichtigste Teil und wurde mit dem proprietären PicOS und den freien Open Network Linux (ONL) genutzt. Dieser verfügt über 48x 10GbE Ports für die Verbindungen auf Data-Plane-Ebene und ist über einen 1GbE Port mit dem Managementnetzwerk auf Control-Plane-Ebene verbunden. Als CPU ist eine Intel Atom C2538 CPU mit 4x 2.40 GHz verbaut und es stehen 8 GB DDR3 RAM zur Verfügung. Als Switching Hardware dient ein Broadcom BCM56854 Trident II Chip mit 720Gbps. Sowohl das eingesetzte PicOS als auch ONL kommen mit einsatzbereiter Forwarding Engine und OpenFlow Agent. Zur Ausführung von Programmen zur Evaluation wurde außerdem auf allen Betriebssystemen Python3 installiert. Zur Virtualisierung standen je nach Betriebssystem verscheide Virtualisierungslösungen zur Verfügung, weitere Informationen folgen auch noch im Abschnitt 4.3 dieses Kapitels.

- PicOS[23]:
  Das prioritäre PicOS wird von Pica8 entwickelt und basiert auf einer um viele speziellen Netzwerkfunktionen erweiterte Debian Version und nutzt XORP und Open vSwitch zur Realisierung seiner Funktionen. Außerdem verfügt es über eine Hardwareabstraktionsschicht, die es erlaubt ASICs verschiedener Hersteller zu nutzen. Auf der Testhardware kam Version 2.7.2 von PicOS zum Einsatz. Containertechnologie wird von PicOS bisher nicht unterstützt, dafür steht KVM-Unterstützung zur Verfügung. Zur Virtualisierung stand daher auf PicOS nur QEMU/KVM das zur Ausführung des Rumprun Unikernel verwendet wurde zur Verfügung. Als OpenFlow Agent wird bei PicOS wird eine angepasste Open vSwitch Installation genutzt die genau wie das Open vSwitch Projekt selbst gut dokumentiert ist und sich schnell und einfach verwenden lässt [49, 23]. Diese nutzt anstelle eines klassischen Open vSwitch Kernelmoduls eine spezielle Version von Pica8 die über die Hardwareabstraktionsschicht die ASIC ansteuert.

- Open Network Linux (ONL)[29]:
  Das freie unter der Eclipse Public License Version 1.0 veröffentliche Open Network Linux basiert auf Debian und erweitert diese um Switching Features für den Betrieb auf offener Weiterleitungshardware. Open Network Linux kam dabei in zwei Versionen zum Einsatz. Einmal in seiner Debian 7 basierten Version mit Kernel 3.2 im Folgenden als ONL V1 bezeichnet, da dies die einzige ONL Version mit funktionierender ASIC auf der Testhardware ist. Außerdem wurde ONL in seiner Debian 8 basierten Version mit Kernel 3.18 genutzt im Folgenden als ONL V2 bezeichnet, da nur auf dieser neueren Version Docker als Containerlösung eingesetzt werden kann. Zur Virtualisierung standen auf ONL V1 QEMU und LXC zur Verfügung und mit ONL V2 wurde Docker genutzt. ONL verwendet den Indigo OpenFlow Agent und basiert auf der OpenFlow Data Plane Abstraction (OFD-PA) [9] von Broadcom die etwas eingeschränkter ist und auch nicht ganz so gut dokumentiert wie z. B. Open vSwitch.

Abbildung 4.1 zeigt eine Skizze des SDN Hardware Testbed. Vom Switch wurden dabei die Ports 1-4, die mit dem ersten Endsystem verbunden sind, die Ports 5-8, die mit dem zweiten Endsystem verbunden sind, und die Ports 18-21, die für zwei Schleifen genutzt sind, verwendet. Der Zugriff auf die Geräte läuft dabei über das Frontend (`vssdn2-fe`), das wie alle anderen Geräte am Managementnetzwerk hängt. Die Endsysteme (`vssdn2-1` und `vssdn2-2`) laufen mit Cent OS. Unter CentOS standen QEMU/KVM und LXC zur Virtualisierung zur Verfügung.

Außerdem wurde für Vorabtests und Entwicklung Mininet [45] eingesetzt, das es einfach und komfortabel ermöglicht lokal in einem simulierten SDN-Netzwerk zu experimentieren. Das spart in vielen Fällen den erheblich größeren Aufwand für das Deployment auf der reale Hardware. Außerdem konnten so Versuche in einer sehr flexiblen Umgebung mit allen Features gemacht werden ehe für Einschränkungen auf den Zielplattformen spezielle Anpassungen gemacht wurden.

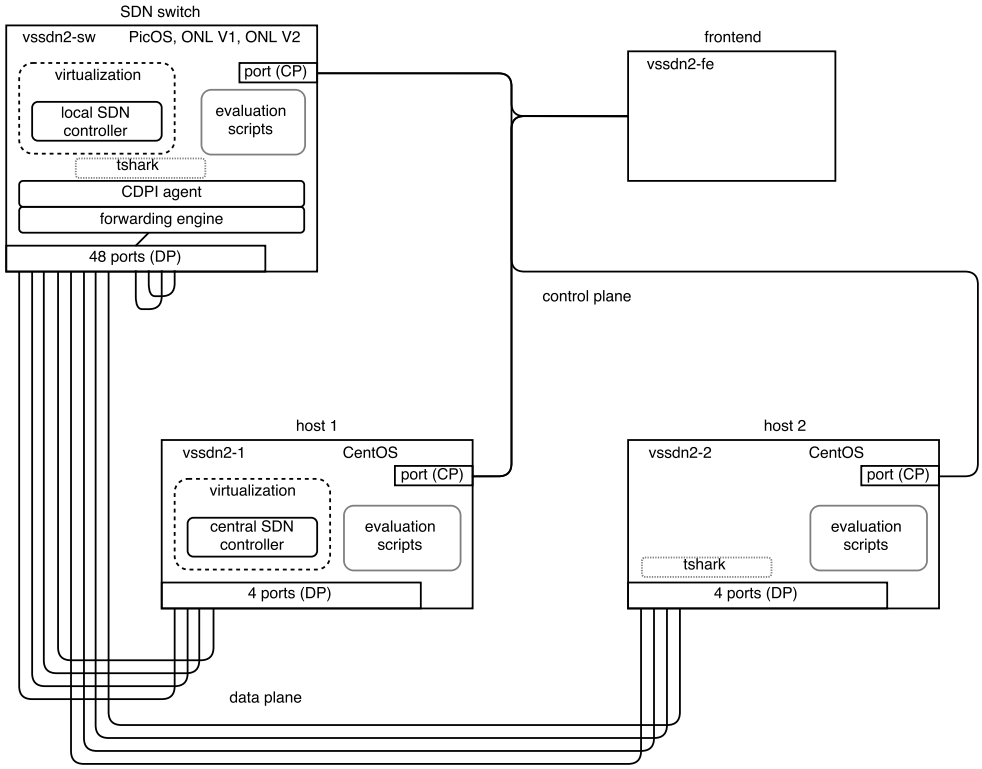


Abbildung 4.1: SDN Hardware Testbed

4.2 Vergleich und Auswahl von SDN-Controllern

SDN-Controller oder oft auch SDN-Controller-Plattformen wurden bereits im Kapitel 2 zu den Grundlagen vorgestellt und steuern die Router und Switches die zu ihrem Netzwerk gehören. Sie sind somit für den Fluss der Daten im Netzwerk zuständig. Sie sind meist leicht um Funktionen bzw. Module erweiterbar und bieten Schnittstellen zur Anwendungsebene. Am häufigsten ist dabei eine API in der Implementierungssprache in Kombination mit einer REST Schnittelle vorzufinden.

In den letzten Jahren ist eine sehr große Anzahl an verschieden SDN-Controllern entwickelt worden. Inzwischen gibt es deswegen eine gewaltige Auswahl an SDN-Controllern für die verschiedensten Einsatzzwecke. Einige Beispiele dafür sind:

Beacon, DISCO, Fleet, Floodlight, Flowvisor, Helios, HP VAN SDN, HyperFlow, IRIS, Jaxon, Kandoo, Maestro, Meridian, MobileFlow, MUL, NodeFlow, NOX, NOX-MT, NVP SDN-Controller, OESS, Onix, ONOS, OpenContrail, OpenDaylight, ovs-SDN-Controller, PANE, POX, ProgrammableFlow, Rosemary, RouteFlow, Ryu, SMaRtLight, SNAC, Trema, usw. [vgl. 28]

Einige dieser SDN-Controller sind dabei allgemein als SDN-Controller geeignet, andere hingegen spezialisieren sich auf bestimme Einsatzgebiete. Sie sind in einer Vielzahl von

Programmiersprachen geschrieben. Am häufigsten aber in C/C++, Python oder Java. Für die Nutzung stehen aber oft mehrere Programmiersprachen zur Verfügung, da einige SDN-Controller auch Schnittstellen zu anderen Programmiersprachen mitliefern. Auch im Bereich aktiver Weiterentwicklung unterscheiden sich die SDN-Controller. Während einige SDN-Controller stetig weiterentwickelt werden gibt es auch viele die einmal entwickelt wurden und anschließend keine Updates mehr erhalten haben.

Für den produktiven Einsatz oder im speziellen für die Umsetzung dieses Konzepts, sind viele der existierenden SDN-Controller allerdings nur bedingt geeignet. Der Grund dafür ist, dass viele der SDN-Controller nicht mehr aktiv weiterentwickelt werden, kaum dokumentiert sind, noch viel zu unausgereift sind oder auf sehr spezielle Anwendungsfälle zugeschnitten sind [27, 44].

Welcher SDN-Controller am besten geeignet ist hängt dabei sehr stark von den individuellen Anforderungen ab. Auch mit welcher Programmiersprache der SDN-Controller gesteuert wird kann dabei entscheidend sein wenn eine Anbindung oder Nutzung bestehenden Codes nötig ist. Die große Anzahl von existierenden SDN-Controller sorgt dabei für gute Chancen einen zu den Anforderungen passenden zu finden.

Eine Übersicht über bekannte SDN-Controller die für die verschiedensten Einsatzgebiete geeignet sind zeigt Tabelle 4.1. Große und bekannte SDN-Controller die auch noch aktiv weiterentwickelt werden und über eine umfangreiche Dokumentation verfügen sind z. B. Ryu (Python) [8], Floodlight (Java) [13] und Trema (Ruby/C) [47]. Alle drei bieten außerdem einen einfachen Weg um eine REST Schnittstelle als Northbound Interface zu nutzen. Sehr bekannt ist auch der OpenDaylight-SDN-Controller (Java) [37] als Teil der sehr umfangreichen OpenDaylight-Plattform und der rudimentäre ovs-controller (C) als Teil des Open vSwitch Projects.

Auch die aktuell nicht mehr aktiv weiterentwickelten SDN-Controller POX (Python) und dessen Vorgänger NOX (C++) sowie Beacon (Java) der Vorgänger von Floodlight haben noch immer eine hohe Bekanntheit.

SDN-Controller	**Programmiersprache(n)**	**Aktive Weiterentwicklung**
Ryu	Python	Ja
Floodlight	Java	Ja
Trema	Ruby, C	Ja
OpenDaylight	Java	Ja
ovs-controller	C	Ja
Beacon	Java	Nein
POX	Python	Nein
NOX	C++, Python	Nein

Tabelle 4.1: Übersicht bekannter SDN-Controller

Neben fertigen SDN-Controllern gibt es außerdem auch eine Reihe von OpenFlow Bibliotheken, für die verschiedensten Programmiersprachen, die es erleichtern eigene zu erstellen. Beispiele dafür sind loxigen[31] das Schnittstellen für verscheide Programmiersprachen bietet, twink [26] für Python sowie libfluid [11] und rofl-common [19] für C++.

Es gibt also eine große Auswahl an SDN-Controllern für die verschiedensten Einsatzgebiete und die durchdachte Wahl eines geeigneten SDN-Controller kann den Einsatz erheblich vereinfachen.

Als interessante Kandidaten für den geplanten Einsatz bieten sich dabei vor allem Ryu, Floodlight und NOX an. Floodlight und Ryu vor allem wegen ihrer sehr guten Dokumentation, vielen Features und sehr aktiven Weiterentwicklung, NOX hingegen als schlanke und native Alternative.

4.2.1 Floodlight

Der Floodlight-SDN-Controller ist ein Java basierter unter der Apache Lizenz stehender OpenFlow-SDN-Controller [13]. Das erweitern der Funktionalität von Floodlight ist über ein einfaches Modulsystem möglich. Hilfreich dabei ist auch die sehr gute Dokumentation von Floodlight. Außerdem ist Floodlight einfach auf einem Zielsystem zu deployen da im Prinzip eine einzige .jar-Datei erzeugt wird das dort von der Java Runtime ausgeführt wird. Die einzige Abhängigkeit ist somit im Prinzip eine existierende Java Runtime. Floodlight ist für den Einsatz in Kombination mit einer Vielzahl an verschieden virtuellen und physischen Switches geeignet und kommt auch mit einer Mischung aus OpenFlow und nicht OpenFlow Netzwerken zurecht. Floodlight ist des Weiteren auf hohe Performance ausgelegt und dient auch als Grundlage für Big Switch Networks kommerzielle SDN-Lösung.

Floodlight ist damit ein im Prinzip von den Features her sehr gut geeigneter SDN-Controller weshalb er für diese Arbeit auch in die engere Wahl kam. Aufgrund der im Abschnitten 4.3 beschrieben Performance Probleme in Verbindung mit Rumprun zeigte sich im Laufe der Arbeit allerdings dass Ryu für diese Arbeit besser geeignet ist.

4.2.2 NOX

NOX ist eine in C++ geschriebener unter der GNU General Public Lizenz stehende SDN-Controller-Plattform [42]. NOX wurde ursprünglich von Nicira Networks entwickelt und 2008 dann als Open Source der Allgemeinheit zur Verfügung gestellt. NOX gilt als sehr schneller SDN-Controller und diente für viele Forschungsarbeiten als Grundlage. Inzwischen wird NOX aber nicht mehr aktiv weiterentwickelt.

Dennoch ist NOX als in C++ geschriebener SDN-Controller im Prinzip gut geeignet da er im Gegensatz zu Floodlight und Ryu keine große Laufzeitumgebung wie Python

oder Java braucht, sondern nativ ausgeführt wird. Das macht ihn potenziell schneller und leichtgewichtiger was für diesen Einsatzzweck ja gewünschte und sehr vorteilhafte Eigenschaften sind. Allerdings ist NOX wie in Abschnitt 4.3 beschrieben nicht ohne Anpassungen mit Rumprun kompatibel weshalb der Einsatz von NOX für diese Arbeit dann doch nicht möglich war. Im Prinzip könnte eine Anpassung von NOX sich aber lohnen da bei NOX Potential besteht aufgrund seiner Implementierung in C++ Overhead gegenüber anderen SDN-Controllern mit aufwendiger Laufzeitumgebung einzusparen.

4.2.3 Ryu

Im Folgenden wird nun der Ryu-SDN-Controller, der zur Umsetzung dieser Arbeit ausgewählt wurde, noch etwas genauer vorgestellt. Die Wahl viel dabei vor allem wegen der guten Dokumentation, der sehr modernen Python3.5 kompatiblen Umsetzung mit vielen nützlichen Features und der sehr aktiven stetigen Weiterentwicklung auf Ryu.

Ryu ist ein Komponenten basiertes unter der Apache 2.0 Lizenz stehendes SDN Framework mit dem Ziel es Entwicklern einfacher zu machen Netzwerkmanagement- und Control-Anwendungen zu schreiben. Ryu unterstützt dazu verschiedene Protokolle wie OpenFlow, Netconf, OF-CONFIG, usw. wobei der Support von OpenFlow dabei von Version 1.0 bis 1.5 einschließlich Nicira Extensions reicht [8]. Ryu kommt außerdem mit einer großen Anzahl von Beispiel-SDN-Controller-Anwendungen und bietet über eine REST API die einfache Möglichkeit Aktionen auszuführen bzw. Informationen abzurufen. Der Sourcecode von Ryu ist auf GitHub verfügbar, wen der Sourcecode nicht verändert wird kann Ryu aber auch einfach über pip für Python3 installiert werden.

Anwendungen für den Ryu-SDN-Controller sind dabei von `RyuApp` abgeleitete ”Applications” und Implementieren für die Events die von Interesse sind passende Event-Handler-Methoden. Abbildung 4.2 zeigt die wesentlichen Elemente der Ryu Architektur. Eine genaue Beschreibung dieser ist in der Dokumentation von Ryu zu finden. Das Starten von Ryu Anwendungen geschieht dann in dem der `ryu-manager` mit den genutzten Applications als Parameter aufgerufen wird. Dabei kann auch ein passendes Log Level gewählt werden was die Fehlersuche mit Ryu erheblich vereinfacht.

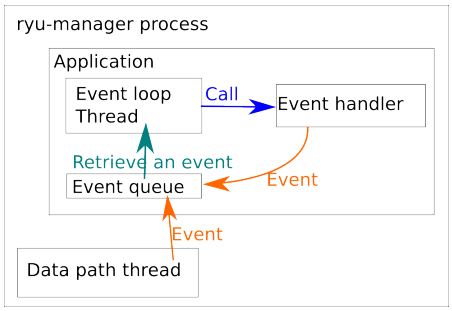


Abbildung 4.2: Ryu Architektur [8]

Die im Folgenden implementierten SDN-Controller sind aufgrund der Vorteile von Ryu und der Kompatibilität mit allen genutzten Virtualisierungstechnologien alle auf Basis von Ryu umgesetzt.

4.3 Vergleich und Auswahl von Virtualisierungslösungen

Ein wichtiges Element im Konzept stellt auch die Virtualisierung des SDN-Controllers dar, um diesen möglichst portabel und vom System auf dem er läuft isoliert zu halten. Dazu ist die Kapselung des SDN-Controllers z. B. in einer VM, einem Container oder Unikernel nötig. Der Vorteil davon ist, dass der SDN-Controller dann vom Rest des Systems isoliert ist was die Lösung unabhängiger von Hostsystem macht und Konflikte mit anderen laufenden Programmen verhindert. Für die Kapselung des lokalen SDN-Controllers auf dem Switch ist deswegen eine geeignete Virtualisierungstechnologie nötig. Die Leistung der Hardware eines Switches ist aber meist eher begrenzt, so dass eine leichtgewichtige Lösung nötig ist um nicht zu viele Ressourcen für die Visualisierung zu verschwenden. Virtuelle Maschinen eignen sich deshalb nicht da sie einen relativ großen Overhead haben und sowohl in Bezug auf Rechenleistung als auch Speicherbedarf für diesen Einsatzzweck eher ungeeignet sind. Im Folgenden soll deswegen ein Blick auf existierenden Containertechnologien und Unikernel Lösungen geworfen werden die wie im Kapitel 2 beschrieben deutlich leichtgewichtigere und daher gut geeignete Alternativen sind.

4.3.1 Container

Im Umfeld der Containertechnologie gibt es vor allem die zwei großen und bekannten Vertreter LXC und Docker.

LXC

Linu**X** **C**ontainer (LXC) [30] basieren auch dem Prinzip Prozesse zu isolieren indem Kernel-Ressourcen virtualisiert und gegeneinander abgeschottet werden. Sie nutzen dazu Technologien wie Kernel Namespaces and `cgroups`. LXC ermöglichen es so Linux-Systeme auf einem Host-Linux auszuführen die statt einem eigenen Kernel den Kernel des Host-Linux gemeinsam nutzen. Der Vorteil davon ist, dass man gegenüber Virtuellen Maschinen deutlich weniger Overhead hat, da keine Hardware emuliert werden muss und auch nur ein einziger Kernel ausgeführt werden muss. Linux Containers (LXC) gibt es bereits seit dem Jahr 2008 und seit Linux-Kernel Version 2.6 sind sie fester Bestandteil von Linux [30].

Zur Installation muss nur das `lxc` Paket installiert werden und anschließend kann über die Konsole mit `lxc-checkconfig` überprüft werden ob LXC voll funktionsfähig ist.

Auf den Switch-Betriebssystemen waren z. B. `cgroups` die von LXC benötigt werden standardmäßig nicht aktiv, so dass diese für die aktuell Session gemountet oder über einen Eintrag in `/etc/fstab` dauerhaft aktiviert werden mussten. Unter PicOS war dies aber nicht möglich und Container werden auch offiziell nicht unterstützt da bereits eine gute KVM Unterstützung gepflegt wird. Anschließend können Container z. B. mit `lxc-create` erstellt werden. Über `lxc-start` können diese dann gestartet werden und mit `lxc-attach` oder `lxc-console` kann man sich mit dem laufenden Container dann verbinden. Stoppen und löschen erfolgt über `lxc-stop` bzw. `lxc-destroy`.

Docker

Docker[22] hat mit seinem Erscheinen 2013 den Container vollends zum Durchbruch verholfen und dabei stark vom Boom des Cloud Computing profitiert. Das Ziel von Docker war es, es so einfach wie möglich zu machen Container zu erzeugen und auszuführen. Ein wichtiger Bestandteil dabei ist das Dockerfile, das eine einfache Konfiguration von Container erlaubt. Anwendungen die in einen Docker Container verpackt sind laufen in jeder Docker-Umgebung, da alle benötigte Pakete im Container Image enthalten sind. Von der Funktionsweise unterscheidet sich Docker dabei kaum von LXC. Bis Version 0.9 war die LXC-Bibliothek sogar die Grundlage von Docker ehe dann die Eigenentwicklung libcontainer diese ablöste [22]. Großer Vorteil von Docker ist die große Auswahl an Docker Images, die über den Docker Hub verfügbar sind und die sehr einfache Erstellung und Konfiguration eigener Container über das Dockerfile, was den Umgang mit Docker erheblich einfacher und effektiver macht wie die Arbeit mit LXC.

Für die Installation stehen Pakete bereit, die über das Hinzufügen der passenden URL als Paketquelle genutzt werden können, sofern Docker nicht schon standardmäßig über die Paketquellen der Linux Distributionen verfügbar ist. Auch Docker benötigt `cgroups`. Sind diese wie unter PicOS nicht verfügbar funktioniert Docker nicht. Die komplette Konfiguration eines Containers erfolgt dann komfortabel über ein Dockerfile, wobei die gute Dokumentation von Docker sehr hilfreich ist ehe der Container mit dem `docker build` Befehl erstellt wird. Ein Dockerfile enthält dabei als erstes eine Zeile mit der Instruktion `FROM` wie z. B. `FROM ubuntu:16.04`, die das als Basis dienende Image angibt. Weitere Befehle wie z. B. `RUN, COPY, ADD, WORKDIR, CMD` sind optional und dienen z. B. dazu Befehle auszuführen oder Dateien hinzuzufügen. Eine Liste aller mögliche Befehle einschließlich ausführlicher Beschreibung der Funktionalität ist in der Dokumentation von Docker zu finden. Ausgeführt wird ein Docker Container dann mit `docker run`. Dabei bietet Docker per Parameter die Möglichkeit direkt beim Ausführen noch schnell gemeinsame Ordner einzubinden, Ports (UDP oder TCP) weiterzuleiten oder einen bestimmten Befehl auszuführen. Stoppen und löschen von Container erfolgt über `docker stop` bzw. `docker rm`.

4.3.2 Fazit Container

LXC und Docker sind somit zwei potenziell gut geeignete Kandidaten zur Virtualisierung eines SDN-Controllers. Wechselt man aber von der theoretischen Sicht in den praktischen Einsatz zeigen sich aber schnell einige Probleme. Das ist zu einem, dass `cgroups` unterstützt werden müssen. Diese sind zwar seit Kernel 2.6 im prinzipiell unterstützt funktionieren aber dennoch nicht auf jeder Linux Distribution die einen neueren Kernel hat. So gibt es für PicOS von Pica8 z. B. offiziell keine Unterstützung für `cgroups` und LXC/Docker. Stattdessen wird auf die gute KVM-Unterstützung verwiesen. Die zweite Plattform auf der Testhardware ONL funktioniert ebenfalls nicht ohne Weiteres. ONL gibt es in zwei Versionen einmal ONL V1 mit Debian 7 und einmal ONL V2 mit Debian 8. Debian 7 erschien mit Linux Kernel 3.2 und der LXC Support weist eine Reihe von Problemen, die erst mit der Zeit weitgehend behoben wurden, auf. Mit der Debian 7 Kernel 3.2 basierten ONL Version die als einzige die ASIC des Switch ansteuern kann läuft also kein Docker, da Docker einen Kernel Version 3.10 oder neuer braucht. Außerdem unterliegt der LXC Support einigen Problemen, wie z. B. dass nicht alle Container Templates funktionieren, Probleme mit systemd existieren, kein `lxc-attach` möglich ist und die Default-LXC-Netzwerkverbindung nicht funktioniert. Die Probleme mit LXC lassen sich aber weitgehend umgehen, so dass LXC dennoch eingesetzt werden kann. Mit der Debian 8 basierten ONL-Version laufen sowohl LXC als auch Docker aber diese ONL-Version ist nicht mit der ASIC der Testhardware kompatibel. Zusammenfassend lässt sich also festhalten, dass PicOS keine Container unterstützt und ONL V1 nur LXC während ONL V2 sowohl Docker als auch LXC unterstützt vorausgesetzt die Hardware ist mit ONL V2 kompatibel. Von Vorteil ist auch, dass sowohl LXC als auch Docker sich per Shell-Skript ansteuern lassen, was dabei hilft das Erzeugen und Starten von Container zu automatisieren. Nur die mangelnde `lxc-attach`-Unterstützung unter ONL V1 erwies sich dabei als störend, da nur damit Befehle mit root Rechten ohne manuellen Login im Container ausgeführt werden können. Tabelle 4.2 fast die wesentlichen Punkte der Container Evaluation nocheinmal übersichtlich zusammen.

Für den Einsatz von Container ist also wichtig, dass diese vom Zielsystem unterstützt werden und es sollte eine möglichst aktuelle Linux-Kernel-Version auf den Switch laufen. Während LXC vermutlich auf mehr Systemen lauffähig ist, da es geringere Anforderungen an das System hat, ist Docker dafür aber die erheblich komfortablere Wahl. Im Prinzip sind aber beide Lösungen für den geplanten Einsatz geeignet.

4.3.3 Unikernel

Die Unikernel Idee ist keine wirklich neue Entwicklung und schon relativ alt, dennoch gewinnen Unikernel als Alternative zu VMs und Containern aber erst durch die Entwicklungen der letzten Jahre an Bedeutung. Für das geplante Einsatzszenario wäre ein Unikernel somit eine Alternative gegenüber Containern und VMs. Ihr Vorteil ist eine hohe Sicherheit bzw. Isolation durch die Trennung in einzelne Betriebssysteminstanzen

	LXC	**Docker**
Min. Kernel Version	2.6	3.10
Konfiguration	teils etwas umständlich	sehr einfach
Gastbetriebssysteme über	LXC Templates	Docker Images (Docker Hub)
Portweiterleitung	manuell	über Docker
Isolation	gut	sehr gut
Mit Shell Skript steuerbar	Ja	Ja
Läuft auf PicOS	Nein	Nein
Läuft auf ONL V1	Ja	Nein
Läuft uaf ONL V2	Ja	Ja

Tabelle 4.2: Vergleich von Docker und LXC

und die starke Reduktion von Code der deployed wird gegenüber VMs. Außerdem sind sie klein und belegen nur wenig Speicherplatz während sie gleichzeitig sehr schnell booten, da nur wirklich benötigte Dienste geladen werden müssen und das Betriebssystem für diesen Einsatz optimiert ist.

Im Umfeld der Unikernel gibt es inzwischen eine Vielzahl an Lösungen. Unikernel.org[48] listet einige verfügbare Unikernel-Projekte auf seiner Webseite auf. Für den geplanten Einsatz geeignete Lösungen sind dabei aber eher Wenige zu finden. Einer der bekanntesten Vertreter ist z. B. MirageOS[35] für Software in der Programmiersprache OCaml. MirageOS ist aber genau wie HaLVM für Haskell oder LING für Erlang und auch viele weitere Unikernel-Lösungen für den geplanten Einsatzzweck eher ungeeignet da ein speziell für diese Plattformen geschriebener SDN-Controller nötig wäre was den Einsatz bestehender und erprobter SDN-Controller-Lösungen ausschließen würde. Andere Unikernel wie z. B. runtimejs.org oder includeos.org stehen sowieso noch am Anfang der Entwicklung und bezeichnen sich selbst als noch nicht ”production ready” [48].

Weitergehende Lösungen sind vor allem Rumprun[43] und OSv[7] für die es mit dem noch recht jungen Unik-Projekt[41] auch ein Tool gibt, das anstrebt Unikernel-Lösungen einfacher zu erstellen und ausführen zu können. Unik steht aber noch in den Anfängen und man gelangt schnell in Situationen, für die es noch keine Dokumentation gibt, vor allem wenn Probleme beim noch nicht ganz ausgereiften Tool auftreten. Dennoch könnte Unik durch das Ziel Unikernel einfacher verwendbar zu machen, ähnlich wie Docker einst für Container, in Zukunft noch eine für das Unikernel-Umfeld sehr wichtige Entwicklung sein. OSv das Linux-Anwendungen, die bestimmte Bedingungen erfüllen ausführen kann und außerdem Support für C, C++, JVM, Ruby and Node.js bietet zielt vor allem auf Cloud und Server als Zielgruppe ab. Als geeignetste Lösung bietet sich deswegen vor allem ein Rumpkernel bzw. Rumprun an. Das liegt daran, dass Rumprun nicht nur für ein spezielles Einsatzgebiet gedacht ist und außerdem viele POSIX-Anwendungen ohne große Modifikationen damit als Unikernel ausgeführt werden können. In der Praxis zeigen sich aber auch hier noch einige Schwierigkeiten. Diese hängen vor allem damit

zusammen, dass ein Rumprun Unikernel immer genau ein statisches Binary ausführt und Programme die aus verschieden Prozessen aufgebaut sind oder dynamische Libraries zur Laufzeit laden somit nicht einfach so lauffähig sind.

Rumprun

Das Rumprun-Projekt[43] bietet die passenden Tools um einfach Unikernel auf Basis der sehr bekannten Rump Kernels zu erzeugen. Das Rump-Kernel-Projekt ist aus dem NetBSD-Projekt entstanden und orientiert sich am Anykernel-Konzept. Das Ziel dabei war ursprünglich Treiber-Entwicklung im User Space zu ermöglichen. Der Vorteil von Rum Kernels ist, dass sie es erlauben UNIX-Applikationen ohne große Änderungen unabhängig von einem kompletten Betriebssystem laufen zu lassen. Sie sind deshalb mit einer erheblich größeren Anzahl an Anwendungen und Programmiersprachen kompatibel als andere Unikernel-Projekte. Vor allem da mit dem rumprun-packages Repository auf GitHub für eine große Anzahl an Programmiersprachen passende Build Unterstützung bereitsteht. Um ein Unikernel mit Rumprun zu erzeugen wird die Anwendung mit dem Corsscompieler des Rumprun-Projekts für die gewünschte Zielplattform des Unikernel kompiliert. Die Unterstützten Plattformen sind dabei alle großen Hardwarearchitekturen und Virtualisierungssysteme. Anschließend werden dem entstandene Binary beim sogenannten 'baking' die benötigten Betriebssystem Libarys hinzugefügt. Als weiteren Schritt unterstützt Rumprun außerdem dieses Binary direkt lokal auf einem unterstützten Hypervisor auszuführen oder ein Bootable ISO daraus zu erzeugen.

In der Praxis zeigt sich, dass Rumprun für viele Einsatzzwecke sehr gut funktioniert. Allerdings stößt man auch schnell auf Probleme, weil etwas nicht sofort funktioniert und Anpassungen nötig sind. Im Prinzip sollte eine Anpassung für Rumprun fast immer möglich sein und meist auch nur geringe Änderungen erfordern, dennoch ist der Aufwand schnell erheblich größer wie bei Containern, wo keinerlei Modifikationen nötig sind. Außerdem ist zu beachten, dass immer nur ein Binary existiert bzw. alles als ein Prozess ausgeführt wird. Anwendungen die aus verschiedenen zusammenarbeitenden Programmen bestehen lassen sich also nicht direkt für Rumprun portieren. Auch ist es somit nicht einfach möglich dynamisch Libraries zur Laufzeit zur laden, stattdessen müssen alle Libraries statisch eingebunden werden.

Hypervisor: QEMU/KVM

Rumprun basiert auf dem Unikernel-Prinzip und läuft somit im Gegensatz zu den Containerlösungen nicht direkt auf einem Betriebssystem. Der Unikernel ist die Kombination der Anwendung mit den benötigten Teilen des Betriebssystems und läuft somit wie ein vollwertiges Betriebssystem direkt auf realer Hardware oder einem Hypervisor. Im Rahmen dieser Arbeit kam Rumprun dazu in Kombination mit, QEMU kurz für Quick Emulator, zum Einsatz. QEMU ist ein Open Source Hypervisor bzw. eine Hardwarevirtualisierungslösung die einen Rechner emuliert. Die doch relative teure Emulation

von Hardware erzeugt aber einen gewissen Overhead. QEMU kann deswegen auch zusammen mit der Kernel-based Virtual Machine (KVM) Funktionalität des Linux Kernels verwendet werden. Dies ermöglicht die Ausführung mit beinahe nativer Geschwindigkeit, erfordert aber Hardware mit Unterstürzung für Hardwarevirtualisierung, die dazu verwendet wird. Auch wenn in dieser Arbeit ausschließlich QEMU/KVM eingesetzt werden wäre im Prinzip die Ausführung aller Rumprun Unikernels mit jedem anderen Hypervisor oder direkt auf Hardware möglich. Für den Ansatz eines virtualisierten lokalen SDN-Controllers ist dabei aber zu beachten, dass der Unikernel auf dem Betriebssystem des Switch neben Forwarding Engine und OpenFlow Agent laufen muss, so dass eine Ausführung direkt auf Hardware nicht möglich ist und nur einf für das Zielbetriebssystem geeignete Hypervisor in Frage kommt.

4.3.4 Fazit Rumprun

Um den lokalen SDN-Controller als Unikernel auszuführen bietet sich also Rumprun an. Für viele Einsatzzwecke funktioniert Rumprun auch sehr gut. Will man aber bestehende Software oder speziell bestehende SDN-Controller damit verwenden stößt man schnell auf Schwierigkeiten. Das liegt daran, dass Rumprun immer genau eine statische Binary braucht die passend kompiliert und um Systembibliotheken ergänzt ausgeführt wird.

Der Test verschiedener SDN-Controller führte deswegen schnell z. B. zu folgenden Problemen:

- Der POX-SDN-Controller läuft, wie einige andere Python basierte SDN-Controller nur mit Python2.X aber Rumprun bietet nur Support für Python3.5.
- Floodlight läuft zwar, aber der noch ganz neue Java 8 Support für Rumprun scheint in Kombination mit Floodlight auf schwacher Hardware mit gewaltigen Performance Problemen zu kämpfen, die Floodlight unbrauchbar langsam machen. Allgemein scheint der Java Support für diesen Einsatzzweck nicht ideal zu sein.
- Ryu als in Python3.5 geschriebene Anwendung wäre im Prinzip geeignet, aber eine der Abhängigkeiten, die greenlet Library [20], ist nicht in Python sondern in C geschrieben. Hier sind also Anpassungen nötig.
- NOX lief aus nicht ganz klarem Grund auch nicht. Das Problem ist vermutlich ebenfalls auf die Einschränkung, dass der Cross Compiler ein statisches Binary das als ein Prozess ausgeführt wird erzeugen muss zurückzuführen.

Python2.X nach Rumprun zu portieren damit POX läuft wäre im Prinzip möglich, da es ähnlich wie die Python3.5 Portierung gelöst werden könnte. Dies stellt aber natürlich sehr viel Aufwand dar. Es gibt bereits einer Feature Request dazu auf GitHub. Bisher haben aber noch keine Entwickler die Zeit dafür investiert, da in den meisten Fällen wohl eine Portierung der Anwendung nach Python3.5 weniger aufwendig, sinnvoller und zukunftssicherer ist. Eine Portierung von POX nach Python3.5 wäre natürlich auch eine Lösung, ist für diese Arbeit aber ebenfalls etwas zu umfangreich. Die Performance

Probleme von Floodlight auf der neuen erst seit August 2016 existierendem Java 8 Portierung sind vermutlich eher schwer zu lösen. Floodlight und andere Java-SDN-Controller sind meist nicht gerade leichtgewichtig und die Java Runtime genauso wenig, was für hohe Anforderungen sorgt. Dazu kommt auch, dass Probleme mit der sehr neuen noch unausgereiften Portierung hier ein entscheidender Faktor sein könnten.

Ryu lauffähig zu machen erscheint auf den ersten Blick schon deutlich einfacher, ist das Problem doch nur die in C geschriebene greenlet.so-Bibliothek [20], die normal in Binärform beiliegend zur Laufzeit geladen wird. Ein Rumprun Unikernel hat aber nur eine Binary und kann zur Laufzeit keine Bibliotheken nachladen. Der greenlet Support müsste also direkt in die Python Runtime integriert werden. Die Python-Unterstützung für Rumprun die auf GitHub mit vielen anderen Erweiterungen für Rumprun im rumprun-packages Repository liegt müsste also nur geringfügig angepasst werden. Das bedeutet CPython, das als Runtime dient und über einige Patches an Rumprun angepasst wird, muss zusammen mit greenlet manuell in ein Binary (zur Python Runtime mit greenlet Modul) kompiliert werden, das dann den Python-Code ausführt. Erste Versuche damit waren aber nicht sehr erfolgreich und Kontakt mit dem Rumprun-Python-Paket-Projekt zeigte schnell, dass es zwar möglich aber doch recht umständlich ist nur eine einzelne Library zusätzlich mit zu compilieren. Das liegt daran, dass es nicht vorgesehen war und so noch realtiv tief in den Build-Prozess von Python3.5 für Rumprun integriert werden musste. Für die Zukunft ist deswegen wohl geplant diesen Prozess deutlich zu vereinfachen, um es Nutzern vom Rumprun-Python-Paket leichter zu machen auch Python Module zu nutzen, die nicht rein in Python geschrieben sind. Die Chancen sind somit gut das Python3-Code zukünftig auch mit binären Abhängigkeiten ohne zu großen Aufwand nutzbar ist.

Es ist also nicht immer ganz leicht Software mit Rumprun lauffähig zu bekommen weswegen für diese Arbeit nur der Ryu-SDN-Controller auf Rumprun eingesetzt wird. Im Prinzip ist aber vermutlich mit ein wenig Aufwand der großteil existierender SDN-Controller portierbar. Dennoch bietet es sich beim geplanten Einsatz von Rumprun an vorher die Anwendung, die damit ausgeführt werden soll, genau anzuschauen, da es eben gewisse Einschränkungen gibt und nicht jede Anwendung sich ohne Weiteres portieren lässt. Außerdem ist natürlich auch die Performance ein entscheidender Faktor, wenn ein Anwendung wie Floodlight zu langsam läuft, weil zu viel Performance verloren geht, ist der Nutzen nicht sehr groß.

Rumprun bietet aber auch einen großen Vorteil. Hat man erstmal einen Unikernel erzeugt läuft dieser überall wo auch eine Linux Distribution laufen würde. Egal ob auf echter Hardware oder einer Virtualisierungslösung wie z.B: QEMU/KVN oder XEN. Auf der für die Evaluation vorhanden Hardware wo PicOS und ONL in Debian 7 und 8 Version laufen ist somit der Einsatz von Rumprun problemlos möglich, da alle Betriebsysteme QEMU/KVM-Unterstützung haben.

Zusammenfassend lässt sich also sagen, dass Rumprun eigentlich überall problemlos läuft dafür aber mehr Aufwand nötig ist die Software mit Rumprun kompatibel zu bekommen. Auch wie bei den Containern kann das Starten des Unikernel dann per Skript erfolgen

was ein einfaches Deployment erlaubt. In Tabelle 4.3 sind die wichtigsten Punkte von Rumprun noch einmal übersichtlich aufgelistet.

	Rumprun
Konfiguration	Aufwendig (App extra compilieren)
Sonstiges	Nicht mit jeder Anwendung kompatibel
Netzwerkinterface mit Portweiterleitung	über QEMU oder manuell
Isolation	sehr gut
Mit Shell Skript steuerbar	Ja, aber eingeschränkter wie Container
Läuft auf PicOS	Ja mit z. B. QEMU
Läuft auf ONL V1	Ja mit z. B. QEMU
Läuft uaf ONL V2	Ja mit z. B. QEMU

Tabelle 4.3: Übersicht Rumprun

4.3.5 Fazit Container und Unikernel

Sowohl Container wie LXC und Docker als auch Unikernel wie Rumprun sind gut zur Virtualisierung eines SDN-Controller geeignet, eine Übersicht über die Unterschiede bietet Tabelle 4.4. Container bieten dabei den Vorteil, dass eigentlich jeder SDN-Controller darauf problemlos läuft. Im Gegenzug aber haben Container dafür gewisse Anforderungen an das System auf dem sie laufen. Ein SDN-Controller auf einem Unikernel zum Laufen zu bringen ist aufwendiger und kann Anpassungen erfordern, da Einschränkungen existieren. Dafür hat ein Unikernel aber keine Abhängigkeiten und läuft sowohl direkt auf Hardware als auch auf sämtlichen Virtualisierungslösungen, wie z. B. dem unter Linux weit verbreiten QEMU/KVM.

Sollen möglichst viele Switches und vor allem auch welche mit älteren Linux-Kernel-Versionen unterstützt werden ist ein Unikernel wie Rumprun definitiv die beste Wahl. Der Preis dafür sind aber eventuelle Sourcode-Anpassungen bzw. Einschränkungen bei der Wahl des SDN-Controller. Container bieten hier den Vorteil, dass eigentlich alle SDN-Controller darauf ohne Anpassung laufen. Unterstützen alle Zielgeräte bereits LXC ist dies somit eine gute Alternative die den Aufwand für eine Anpassung des SDN-Controller für Rumprun spart. Muss der SDN-Controller nur auf Geräten laufen, die modern genug sind um Docker zu unterstützen spielt Docker seine Vorteile aus und ist dank vielen Features und erheblich einfacher bzw. vor allem komfortableren Bedienbarkeit und Konfiguration gegenüber LXC deutlich im Vorteil.

Für die Ziele aus dem Konzept wie Isolation und Unabhängigkeit vom Switch bietet sich deswegen Rumprun an. Container sind zwar auch gut geeignet stellen aber bereits Anforderungen an den Switch die mangels älterer Kernel oder mangelndem Support des Herstellers unter Umständen nicht überall erfüllt werden. Container können deshalb in Bezug auf Plattformunabhängigkeit nicht mit Rumprun mithalten.

Neben diesem allgemeinen Vergleich von LXC, Docker und Rumprun auf Basis der Funktionalität folgt in Kapitel 5 noch ein Vergleich in Bezug auf die Performance, die ebenfalls ein wichtiger Faktor bei der Wahl einer geeigneten Lösung ist.

	LXC	**Docker**	**Rumprun**
Läuft ab/auf	Linux Kernel 2.6 mit `cgroups`, usw	Linux Kernel 3.10 mit `cgroups`, usw	Hardware oder Hypervisor
Anforderungen an Anwendung	Linux kompatibel	Linux kompatibel	Extra compiliert
Netzwerkinterface mit Portweiterleitung	manuell	über Docker	über QEMU oder manuell
Isolation	gut	seht gut	sehr gut
Mit Shell Skript steuerbar	Ja	Ja	Ja, aber nicht so umfangreich
Läuft auf PicOS	Nein	Nein	Ja mit QEMU
Läuft auf ONL V1	Ja	Nein	Ja mit QEMU
Läuft uaf ONL V2	Ja	Nein	Ja mit QEMU

Tabelle 4.4: Vergleich zwischen LXC, Docker und Rumprun

4.4 Anwendungsszenarien

Im Folgenden wird die Implementierung der Anwendungsszenarien beschrieben die als Grundlage für die Evaluation im folgenden Kapitel dienen. Dazu wurden die folgenden drei Anwendungsszenarien ausgewählt:

- Simple Switch
- Port Knocking
- Fast Failover

Die Umsetzung dieser Anwendungsszenarien erfolgte in Python3 für den Ryu-SDN-Controller, da dieser sich zuvor beim Betrachten der verschieden SDN-Controller als ein sehr moderner, vielseitiger und gut dokumentierter SDN-Controller herausgestellt hatte und auch für den Einsatz mit allen Virtualisierungsstechnicken geeignet ist.

4.4.1 Simple Switch

Die Simple-Switch-SDN-Controller-Implementierung setzt die Grundfunktionalität klassischer L2-Switches um. Dazu setzt die mit Ryu umgesetzte SDN-Controller-Implementierung

ein einfaches MAC lernen und damit verbundenes Flooding bzw. Installieren von Flows, wenn das Ziel eines Pakets schon bekannt ist um.

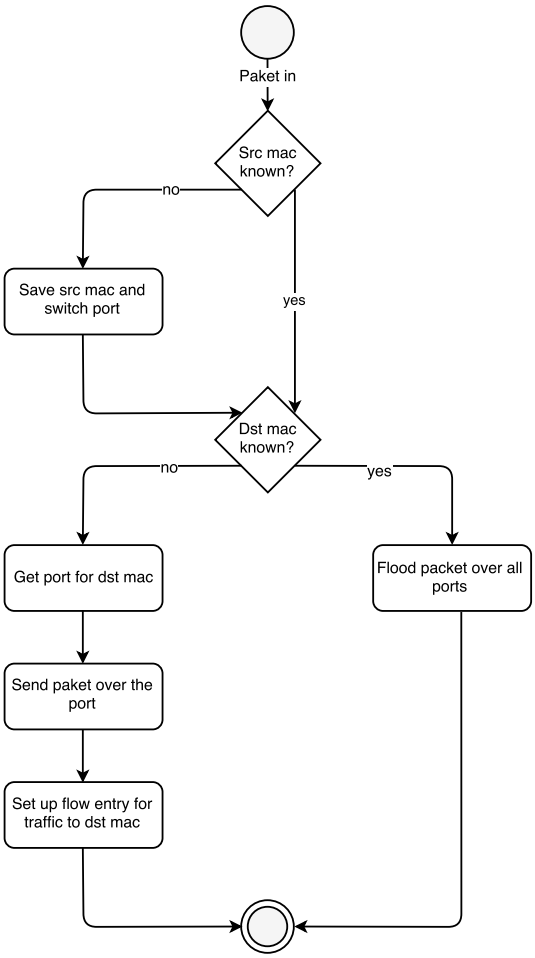


Abbildung 4.3: Simple-Switch-Implementierung

Der Ablauf dabei ist wie in Abbildung 4.3 dargestellt folgendermaßen: Der SDN-Controller installiert einen Standard-Flow, so dass Pakete für die noch kein Flow Entry existiert zum SDN-Controller weitergeleitet werden. Kommt ein Paket, das zum SDN-Controller weitergeleitet wurde, bei diesem an speichert dieser sich, sofern die Source-MAC-Adresse noch unbekannt ist diese ab. Dazu prüft der SDN-Controller ob er schon einen Eintrag zu der Source-MAC-Adresse hat und Speichert falls nicht unter der Source-MAC-Adresse den Port, auf dem die Nachricht den Switch erreicht hat, ab. Anschließend schaut der SDN-Controller ob ein Eintrag für die Ziel-MAC-Adresse existiert. Gibt es noch keinen Eintrag kennt der SDN-Controller die Ziel-MAC-Adresse noch nicht und lässt den Switch über eine Flooding Aufforderung das Paket über alle Ports weiterleiten. Hat der SDN-Controller hingegen einen Eintrag für die Ziel-MAC-Adresse dann lässt er das Paket

nur über den zu dieser Ziel-MAC-Adresse gespeicherten Port weiterleiten und installiert einen Flow-Table-Eintrag für zukünftige Pakete. Auf Zukünftige Pakete mit gleichem Absender und Ziel passt dann der Flow-Table-Eintrag, so dass ein direktes Weiterleiten ohne Interaktion mit dem SDN-Controller möglich ist.

Als Grundlage und Ausgangsbasis zur Implementation des Simple-Switch-Anwendungsszenarios diente dabei das `simple_switch_13.py` Beispiel von Ryu und die auf GitHub zu findende OF-DPA-Variante davon [24]. Am Ryu Beispiel das mit Open vSwitch unter PicOS Problemlos funktioniert waren dabei kaum Änderungen nötig. Für die OFP-DPA Version, die mit ONL und Indigo funktioniert, musste das schon etwas ältere Sample neben einigen kleinen Anpassungen vor allem auch noch Python3.5 kompatibel gemacht werden. Die Unterschiede zwischen PicOS mit Open vSwitch und ONL mit Indigo Agent der OF-DPA nutzt sind dabei der Grund, dass zwei verschiedene Beispiele als Basis dienten. Dabei zeigte sich außerdem, dass OF-DPA doch sehr viele Einschränkungen hat, für die die Implementierung einzeln angepasst werden muss und dass auch nicht alle Probleme ohne weiteres mit OF-DPA gelöst werden können. Ein Problem mit dem Indigo Agent und OF-DPA auf ONL ist dabei vor allem, dass es nicht funktionierte Pakete an den SDN-Controller zu senden und nur wenn nötig ein Flooding zu machen. Es funktionierte nur die Variante aus dem Sample, die ein Flooding vor dem Weiterleiten an den SDN-Controller macht. Somit war es nicht möglich alle Anwendungsfälle auch für eine Evaluation unter ONL umzusetzen. Mangels Informationen zu diesem spezifischen Problem ist nicht ganz klar ob es eine bestimmtes durch andere Konfiguration theoretisch lösbares Problem ist oder die auf dem Switch zum Einsatz kommenden Indigo- / OF-DPA-Version dies einfach nicht unterstützt.

4.4.2 Port Knocking

Port Knocking ist ein Verfahren um über ”Anklopfen” an verschieden Ports eine bestimmte Aktion auszulösen. Dazu wird eine Sequenz von Paketen mit verschiedenen Zielports gesendet. Eine State Maschine die auf ankommende Pakete achtet löst dann bei korrekte Port-Knocking-Sequenz die damit verbundene Aktion aus. Der Hauptanwendungsfall dabei ist Ports in einer Firewall, die Server bzw. Serverdienste absichert, freizuschalten. Der Client klopft dabei über das Senden von z. B. TCP-SYN- oder UDP-Paketen an in dem er die Pakete in einer zuvor definierten Sequenz an verschiedene Ports schickt. Erkennt die Firewall, dass der Client eine korrekte Port-Knocking-Sequenz geschickt hat öffnet sie für den Client den damit verbunden Port, so dass dieser auf die über diesen Port angebotene Dienste zugreifen kann. Ohne korrekte Port-Knocking-Sequenz hingeben bleibt der Port geschlossen und ankommende Pakete werden verworfen. Der Vorteil des Verfahrens ist, dass von außen z. B. mit einem Portscanner nun nicht zu erkennen ist welche Dienste angeboten werden und nur bei bekannter Port-Knocking-Sequenz darauf zugegriffen werden kann. Port Knocking kann so zum Beispiel dazu verwendet werden um Fernzugriffsmöglichkeiten wie SSH zu verschleiern aber ersetzt keine Authentifizierung und Verschlüsselung zur Absicherung von Diensten.

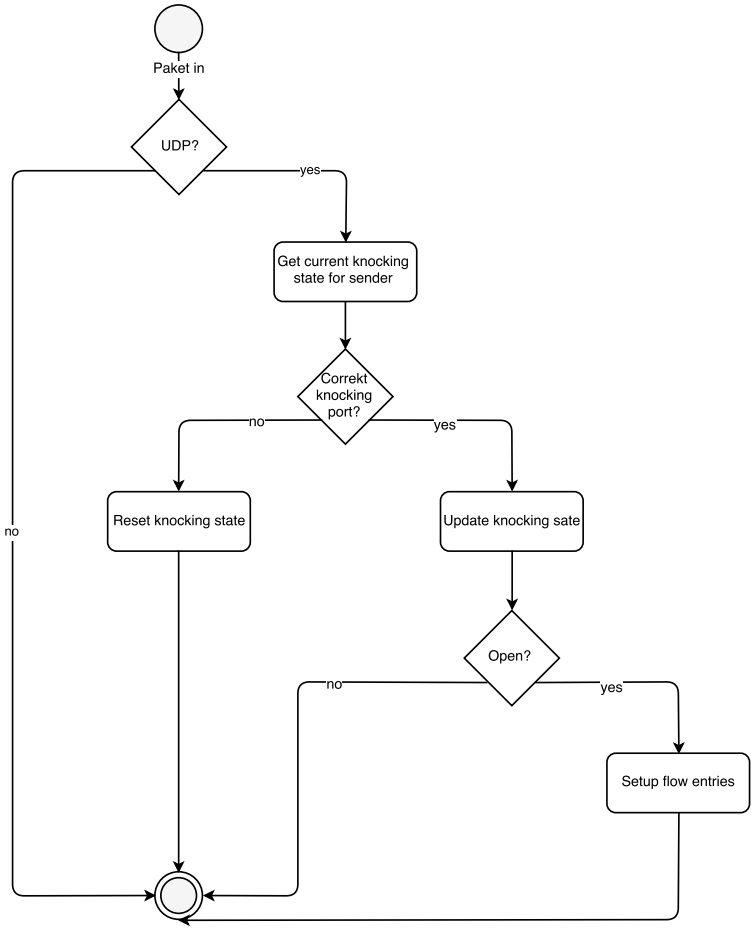


Abbildung 4.4: Port-Knocking-Implementierung

Für das Port-Knocking-Anwendungsszenario wurde der SDN-Controller so angepasst, dass er auf eingehende UDP-Pakete achtet. Abbildung 4.4 zeigt die Funktionsweise dieser Implementierung. Erreicht ein UDP-Paket den SDN-Controller schaut dieser auf welchem Port das nächste Paket vom Sender erwartet wurde. Liegt noch kein Statuseintrag für den Absender vor ist dies der Anfang der Port-Knocking-Sequenz, ansonsten der gespeicherte Status. Kam das Paket auf dem erwarteten Port an geht es in der Port-Knocking-Sequenz weiter und der neue Status wird gespeichert bis das nächste Port-Knocking-Paket des Senders ankommt. Stimmt der Port nicht wird der Status verworfen und als nächstes eine neu beginnende Port-Knocking-Sequenz erwartet. War der Port korrekt und das Ende der Port-Knocking-Sequenz erreicht wird ein Flow-Eintrag installiert, der dem Client die Kommunikation über den vereinbarten Port erlaubt.

4.4.3 Fast Failover

Dem Fast-Failover-Ansatz liegen redundante Verbindungen im Netzwerk zugrunde. Ziel dabei ist es durch Ausfälle von Verbindungen verursachte Unterbrechungen des Netzwerks schneller zu beheben bzw. die Ausfallzeit zu minimieren. Statt sich nur die besten Verbindungen zu merken, wie es im Normalfall ausreichen würde, liegt nun der Fokus auf redundanten Verbindungen. Das bedeutet, wenn es eine alternative Verbindung gibt wird diese ebenfalls gespeichert, für den Fall, dass die aktuell genutzte Verbindung ausfällt. Fällt eine Verbindung aus kann so, wenn verfügbar sofort auf eine alternative Verbindung gesetzt werden. Die Suche nach einem neuen Weg für die Daten entfällt somit, sofern eine alternative Verbindung auf die ausgewichen werden kann bekannt ist. Dadurch wird die Ausfallzeit reduziert.

Zur Umsetzung des Fast-Failover-Ansatzes wurde der SDN-Controller so erweitert, dass er über Statusänderungen von Ports informiert wird. Fällt ein aktiv genutzter Port aus, kann der SDN-Controller nun schauen ob er für die Flows über diesen Port Alternativen kennt. Ist dies der Fall, kann der SDN-Controller dann sofort neue Flows, die den Ausweichweg nutzen, installieren. Damit ist die Verbindung dann repariert und die Ausfallzeit geringer als wenn erst beim Bekanntwerden des Verbindungsausfalls eine neue Verbindung gesucht werden muss. Abbildung 4.5 zeigt die Funktionsweise der Fast-Failover-Implementierung.

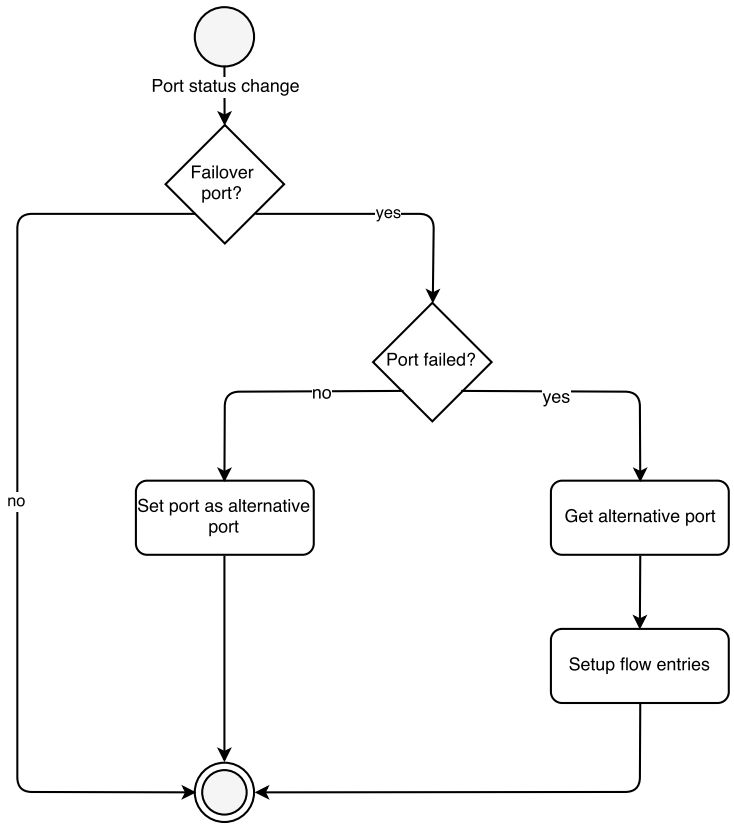


Abbildung 4.5: Fast-Failover-Implementierung

5 Evaluation

Im Folgenden wird das Konzept und dessen Umsetzung evaluiert. Dazu folgt als erstes eine Evaluation zur Performance der verschiedenen Virtualisierungstechniken wozu deren Performance in Form der Round Trip Time (RTT) bei der Kommunikation über TCP betrachtet wird. Anschließend wird die Performance beim Einsatz von OpenFlow untersucht, um zu sehen was das Erzeugen und Parsen von OpenFlow-Nachrichten an Zeit kostet. Als letztes findet dann die Evaluation von drei Anwendungsszenarien satt. Dies soll helfen den Nutzen des virtualisierten lokalen SDN-Controller-Ansatzes in praktischen Einsatz aufzuzeigen.

5.1 TCP Performance verschiedener Virtualisierungslösungen

Bereits bei der Betrachtung der verschiedenen Virtualisierungslösungen im vorausgegangen Kapitel zeigte sich, dass Container und Unikernel für den geplanten Einsatz in Frage kommen. In diesem Abschnitt folgt nun die Evaluation der Performance dieser verschiedenen Virtualisierungstechniken auf der zur Verfügung stehenden Hardware. Dazu wird die Performance der Netzwerkanbindung in Form der RTT von den in Kapitel 4 vorgestellten Containerlösungen LXC und Docker sowie des dort als am geeignetsten bestimmten und näher beschriebenen Rumprun Unikernel untersucht.

Für den geplanten Einsatz, einen lokalen SDN-Controller zu virtualisieren, ist dabei vor allem die Performance der Netzwerkanbindung relevant. Das liegt daran, dass diese die entscheidende Verbindung sowohl zur Data Plane als auch zur Control Plane darstellt und jegliche Interaktion darüber stattfindet. Als Protokoll kommt dabei vor allem TCP zum Einsatz auf das z. B. auch OpenFlow aufbaut, weshalb der Fokus auf der Performance in Verbindung mit TCP liegt.

Um die Performance einer TCP-Verbindung über die Netzwerkschnittstelle der Virtualisierungslösungen zu evaluieren wurde deshalb eine kleine ab Python3.3 laufende Testanwendung entwickelt. Diese besteht aus einem TCP Responder (Server), der virtualisiert wird, und einem TCP Sender (Client), der sich mit diesem verbindet. Der TCP Responder übernimmt die Rolle des z. B. im Container oder als Unikernel ausgeführten SDN-Controllers. Der TCP Sender hingegen übernimmt die Rolle des CDPI Agent und läuft direkt auf dem Betriebssystem, das auf der Switch-Hardware ausgeführt wird. Dieses Setup ermöglicht es die Performance einer TCP-Verbindung über die Netzwerkschnittstelle in einer Testumgebung zu evaluieren, die sehr ähnlich mit der tatsächlichen Nutzung ist.

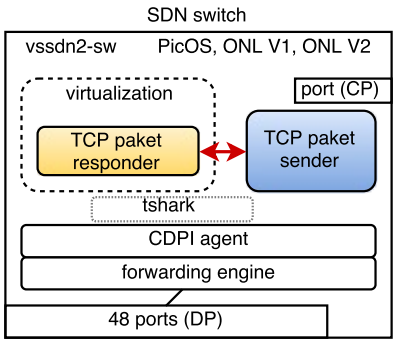


Abbildung 5.1: Versuchsaufbau für TCP-RTT-Messung

In Abbildung 5.1 ist dazu der Testaufbau skizziert. Der TCP Sender ist dabei in blau markiert und der TCP Responder innerhalb der Virtualisierungslösung in gelb hervorgehoben. Die Kommunikation über TCP zwischen TCP Sender und TCP Responder für die, die RTT gemessen wurde ist rot markiert.

Der in Python3 geschriebene TCP Responder hat dabei die Aufgabe Verbindungen entgegen zu nehmen und alle ankommenden Pakete wieder zurück zum Absender zu schicken. Hauptbestandteil ist dazu eine Schleife die alle ankommende Pakete wieder zurück zum Absender schickt.

Listing 5.1: Funktionsweise TCP Responder

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((ip, port))
sock.listen(1)
while True:
connection, address = sock.accept()
while True:
  data = connection.recv(1024)
  if not data: break
  connection.send(data)
connection.close()
```

Der TCP Sender ist das Gegenstück zum TCP Responder und ebenfalls mit Python3 umgesetzt. Seine Aufgabe ist der Aufbau einer Verbindung zum TCP Responder und das senden von Paketen. Außerdem führt der TCP Sender auch das Messen der benötigten Zeit bzw. der RTT durch. Dazu stoppt der TCP Sender die Zeit, die nötig ist für eine vorgegebene Anzahl an Wiederholungen ein Paket bestimmter Größe zum TCP Responder zu schicken und auf die Antwort darauf zu warten. Die Zeit für alle Wiederholungen wird dabei für eine möglichst genaue Zeitmessung mit time.perf_counter() gemessen und am Ende durch die Anzahl an Wiederholungen geteilt um eine möglichst genaue Round Trip Time (RTT) für ein einzelnes TCP-Paket zu ermitteln.

Listing 5.2: Funktionsweise TCP Sender

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((ip, port))
timestamp = time.perf_counter()
  for i in range(1, n+1):
  sock.send(data)
  data = sock.recv(1024)
timespan = (time.perf_counter() - timestamp) / n
```

Die Durchführung der Messungen fand sowohl auf PicOS als auch unter ONL V1 und ONL V2 statt. Neben der Evaluation der Performance des virtualisierten TCP Responder mit den je nach Plattform verschiedenen verfügbaren Virtualisierungstechniken erfolgte dabei auch immer ein Test ohne Virtualisierungslösung, um einen Vergleichswert, für die TCP Performance ohne den Overhead einer Virtualisierungslösung, zu haben.

Unter PicOS fand die Messung nativ und mit Rumprun auf QEMU/KVM statt. Unter ONL V1 nativ, mit LXC und mit Rumprun auf QEMU ohne KVM Support. Auf ONL V2 fand der Test nativ und mit Docker statt. Bei Rumprun wurden dabei immer zwei Messungen gemacht. Einmal mit User Networking (SLIRP), das im Wiki[46] zu QEMU als einfaches Standard Interface mit relativ viel Overhead und nicht so guter Performance beschrieben wird. Dann ein zweites Mal mit einem Tap Interface, das ein auf dem Host angelegte Tap-Netzwerkschnittstelle nutzt und im QEMU Wiki als schnellere Alternative beschrieben wird. Wie die folgenden Performancetests zeigen sind die Unterschiede zwischen diesen beiden Netzwerkinterfaces von QEMU aber zumindest in diesem Szenario sehr gering.

Zur Festlegung der Anzahl an Wiederholungen wurden diese schrittweise erhöht bis verschiedene Ausführungen des Programms zu keinen nennenswerten Schwankungen mehr führten. Dabei zeigte sich, dass 50000 Wiederholungen genügen um ein zuverlässiges Ergebnis zu erzielen. Da dennoch vereinzelt Abweichungen entstanden sind, die Vermutlich durch Prozesse, die im Hintergrund auf dem Switch laufen, entstehen, wurden allen Tests mehrfach durchgeführt. Mit ausreichend Wiederholungen konnte verhindert werden, dass einzelne Ausreißer, bei denen das Ergebnis durch Hintergrundprozesse auf dem Host ein wenig abweicht, das Gesamtergebnis verfälschen. Ohne in die Quere kommende Hintergrundprozesse blieben die Abweichungen zwischen verschieden Ausführungen dann bei unter 0,001 ms. Für die Unterschiede zwischen den verschiedenen Virtualisierungslösungen hingegen zeigte sich, dass diese größer als 0,01 ms sind. Für die für die Evaluation interessanten Unterschiede zwischen den Virtualisierungslösungen sind somit die um Faktor 10 geringen Abweichungen nicht störend.

Um verschiedene Paketgrößen zu simulieren wurden 50 Byte, 500 Byte und 1000 Byte große Pakete verwendet. Wie die Tabelle 5.1 und Abbildung 5.2 zeigt spielt die Paketgröße wenn genug Bandbreite vorhanden ist, jedoch keine sehr große Rolle. Direkt auf dem Betriebssystem oder in einem Container ausgeführt war deshalb kein Unterschied festzustellen. Für QEMU/KVM in Verbindung mit SLIRP-/TAP-Netzwerkinterface hin-

gegen zeigte sich, dass die Paketgröße durchaus eine Rolle spielt. Die dabei entstehenden Abweichungen liegen aber im Bereich von unter 0,1 ms.

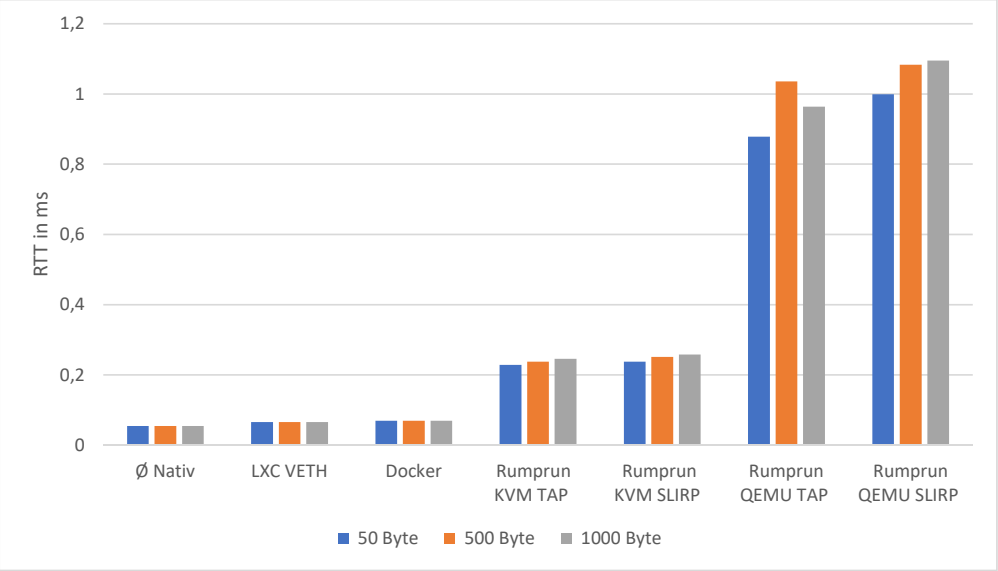


Abbildung 5.2: Ergebnisse der TCP-RTT-Messung der Virtualisierungslösungen: LXC, Docker und native Ausführung in etwa gleichauf. Rumprun mit KVM bereits etwas langsamer und ohne KVM erheblich langsamer.

OS	**Virtualisierung**	**50 Byte**	**500 Byte**	**1000 Byte**
PicOS				
	Nativ	0,055	0,055	0,055
	Rumprun KVM SLIRP	0,238	0,251	0,258
	Rumprun KVM TAP	0,229	0,238	0,246
ONL V1 (3.2)				
	Nativ	0,056	0,056	0,056
	LXC VETH	0,066	0,066	0,066
	Rumprun QEMU SLIRP	0,999	1,083	1,095
	Rumprun QEMU TAP	0,878	1,036	0,964
ONL V2 (3.18)				
	Nativ	0,055	0,055	0,055
	Docker	0,070	0,070	0,070

Tabelle 5.1: Messergebnisse für die durchschnittliche TCP RTT (in Millisekunden) der verschiedenen Virtualisierungslösungen

Wie in Tabelle 5.1 und Abbildung 5.2 ebenfalls zu sehen, ist LXC insgesamt mit einer RTT von 0,066 ms die schnellste Virtualisierungslösung aber Docker ist mit 0,07 ms in

etwa gleichauf. Die Container-Lösungen führen damit gegenüber dem Unikernel-Ansatz deutlich. Rumprun mit KVM hat eine ca. dreimal so große RTT von im Schnitt 0,249 ms. Ausgeführt mit QEMU (ohne KVM Support) bricht, wie durch die Virtualisierung ohne Hardwareunterstützung zu erwarten, durch die deutlich langsamere Virtualisierung die Performance sogar noch weiter ein und die RTT beträgt ca. 1 ms.

5.2 OpenFlow Performance Evaluation

Im Konzeptteil zeigte sich, dass eine Verwendung des OpenFlow-Protokolls zur Kommunikation zwischen OpenFlow Agent und lokalem SDN-Controller auf dem Switch genauso wie zwischen lokalen SDN-Controller und übergeordnetem bzw. zentralem SDN-Controller sehr praktisch ist, da dann nur wenige Anpassungen nötig sind. Dennoch stellt sich die Frage ob OpenFlow dazu geeignet ist, da es doch einiges an Overhead mitbringt und jeder der beiden Kommunikationswege eventuell durch einen anderen leicht gewichtigeren Ansatz noch effizienter werden könnte. Aus diesem Grund wird im Folgenden ein Blick auf die Kosten, die das OpenFlow-Protokoll verursacht, geworfen. Dazu werden die Kosten für das Encodieren und Decodieren von OpenFlow-Nachrichten in verschiedenen Szenarien evaluiert.

5.2.1 twink

Eine erste grobe Evaluation dazu wurde mit der twink Library für Python gemacht, die ausschließlich Funktionen zum Encodieren bzw. Decodieren von OpenFlow-Nachrichten bietet. Das hat den Vorteil, dass verschiedene dieser Funktionen einfach wiederholt aufgerufen werden können während die Zeit mit `time.perf_counter()` gestoppt wird, ehe am Ende die Zeit wieder auf einen Durchschnittswert für einen Aufruf herunter gebrochen wird. Ab 100000 Wiederholungen waren dabei die Abweichungen für das Encoding wieder kleiner als 0,001 ms, so dass stabile Ergebnisse abgelesen werden konnten. Beim Decodieren, das erheblich länger dauert, wurden 10000 Wiederholungen gemacht womit die Abweichungen bei unter 0,02 ms lagen. Gemessenen wurden dabei die durchschnittlichen Kosten für eine OpenFlow-Nachricht, weswegen zur Messung verschiedene OpenFlow-Nachrichtentypen verwendet wurden, um sowohl kleine Nachrichten wie z. B. eine `Hello`-Nachricht aber auch größere wie z. B. eine `PaketOut`-Nachricht im Ergebnis repräsentiert zu haben.

Leider zeigte sich, dass mit Rumprun keine Tests unter zu starker Last möglich sind, da alle Funktionen in Bezug auf Uhrzeit oder Performance Counter unter Last versagen und keine sinnvollen Werte mehr zurückgeben. Rumprun hat dazu auf GitHub auch ein offenes Ticket mit der Bezeichnung "Time jumps backwards when user code is cpu-bound for long enough #71". Die Lösung dort ist, unter C mit `sched_yield()` sicherzustellen, dass das Zeitmanagement oft genug durchgeführt werden kann. Python kennt diese Funktion aber nicht und dynamisch mit ctypes die Library (libc.so.6) zu laden geht aufgrund der

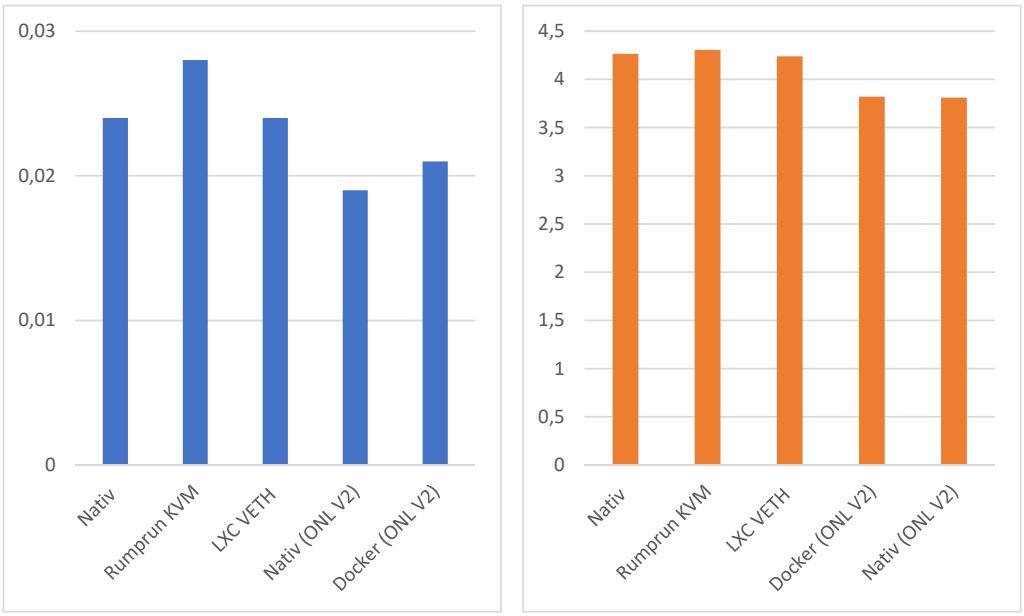
Unikernel-Architektur nicht. Die einzige Lösung war somit nur sehr kurze Schleifen zu messen, die die CPU nicht zu lange blocken. Dabei treten aber die erheblich größeren Schwankungen auf die bereits zuvor der Grund für mehr Wiederholungen waren. Aus diesem Grund wurden nun viele Messungen mit 10 Wiederholungen gemacht und dazwischen mit einem `time.sleep()` Aufruf dem System Zeit gegeben damit keine Probleme mit der Zeitberechnung auftreten. Bei vorher z. B. 100000 Wiederholungen wurde nun stattdessen 10000 mal eine einzelne Messung mit 10 Wiederholungen und darauf folgender Pause ausgeführt. Der Durchschnitt der vielen kleinen Messungen ergab so eine stabile und recht genaue Zeitmessung auch für Rumprun.

OS	**Virtualisierung**	**Encode (ms)**	**Decode (ms)**
PicOS:			
	Nativ	0,024	4,284
	Rumprun KVM	0,028	4,206
ONL V1 (3.2):			
	Nativ	0,024	4,247
	LXC VETH	0,024	4,239
	Rumprun QEMU	0,508	16,645
ONL V2 (3.18):			
	Nativ	0,019	3,811
	Docker	0,021	3,819

Tabelle 5.2: Messergebnisse für die durchschnittliche Zeit (in Millisekunden) die twink für das Decodieren bzw. Encodieren einer OpenFlow-Nachricht benötigt

Dabei zeigte sich wie in Tabelle 5.2 und Abbildung 5.3 zu sehen, sehr schnell, dass die Kosten bei der twink Library vor allem beim Parsen des OpenFlow-Pakets liegen. Die Pakete zum Senden werden meist Stück für Stück befüllt indem die Daten an die passende Stelle geschrieben werden, was sowieso auch bei anderen Protokollen unvermeidbar ist und keinen nennenswerten Overhead verursacht. Hier schlägt sich twink auch sehr gut. Ein empfangenes Paket hingegen muss passend interpretiert werden was deutlich aufwendiger ist. Pakettypen, Keys und Längenangaben helfen dabei das Paket entsprechend der Definitionen in der OpenFlow-Spezifikation, die viele verschiedene Pakettypen und Inhalte dieser spezifiziert, zu parsen. Das Parsen von OpenFlow ist also relativ aufwendig, aber auch hier ist zu beachten, dass dieses Problem natürlich auch alternative Protokolle haben. Alternativen wären also vor allem dann im Vorteil, wenn es sich dabei um ein einfacheres und schlankeres Protokoll handelt das leichter zu parsen ist. Ein großer Teil der hohen Kosten für das Parsen von OpenFlow-Nachrichten entfällt dabei sowieso auf die scheinbar nicht sehr gut optimierte Implementierung der twink Library, da wie der hierauf folgende Test mit Ryu zeigt Pakte mit Python auch deutlich schneller geparst werden können. In Abbildung 5.3 sind auch die wichtigsten Zeiten im Diagramm gegenüber gestellt. ONL V2 und Docker sind dabei vermutlich aufgrund des neueren Linux Kernel schneller. Rumprun mit QEMU ohne KVM Support ist durch die Virtualisierung ohne Hardwareunterstützung sehr langsam wenn die Performance wichtig ist würde man

Rumprun mit QEMU ohne KVM Support nicht nutzen. Aus diesem Grund wurde es in den Diagrammen auch weggelassen, da so die Unterschiede der anderen Lösungen besser sichtbar bleiben.



(a) Encode Zeit in ms

(b) Decode zeit in ms

Abbildung 5.3: Ergebnisse der Encode- und Decode-Zeitmessung mit twink: Unterschiede zwischen den Virtualisierungslösungen sind von QEMU ohne KVM abgesehen gering. Die Decodierung ist allerdings deutlich langsamer wie das Encodieren.

5.2.2 Ryu

Wichtig beim Parsen ist vor allem die Effizienz des Codes, der diese Aufgabe übernimmt. Die recht simple twink Library zum Beispiel ist scheinbar nicht auf Performance optimiert und für den produktiven Einsatz somit nicht so gut geeignet wie Ryu.

Tabelle 5.3 zeigt die Performance von Ryu beim Parsen von OpenFlow-Nachrichten. Dabei wurde die Zeit zum Parsen einfach während des Betriebs des SDN-Controller unter realen Bedingungen mit gemessen. Damit der Ryu-SDN-Controller aber nicht nur `Echo`-Nachrichten empfängt wurden Pakete durchs Netzwerkgeschick für die keine Regel im Switch existierte, so dass `PaketIn`-Nachrichten erzeugt und an den Ryu-SDN-Controller

geschickt wurden. Auf der Ryu-SDN-Controller Seite wurde zu Evaluation dann das Parsen aller am Ryu-SDN-Controller ankommenden Nachrichten gemessen und die durchschnittliche Zeit für eine Nachricht berechnet. Die gemessenen Zeiten repräsentieren also die durchschnittliche Zeit für das Parsen einer OpenFlow-Nachricht im normalen Betrieb.

Da dabei die QEMU und Rumprun Kombination nicht bis ans Limit ausgelastet war funktionierte auch dort die Messung problemlos. Dafür war die Messung mit Docker in diesem Fall nicht möglich, da Docker nur auf ONL V2 läuft und dort der OpenFlow Agent nicht korrekt funktioniert. Ryu parsed die OpenFlow-Nachrichten dabei recht schnell nativ auf dem Betriebssystem des Switches und mit einem LXC Container beträgt die Zeit im Schnitt 0,08 ms. Auch Rumprun mit KVM ist mit 0,10 ms kaum langsamer QEMU ohne KVM Support ist wie erwartet mit ca. 1 ms aber wieder deutlich langsamer, da hier viel Performance beim virtualisieren des SDN-Controllers verloren geht.

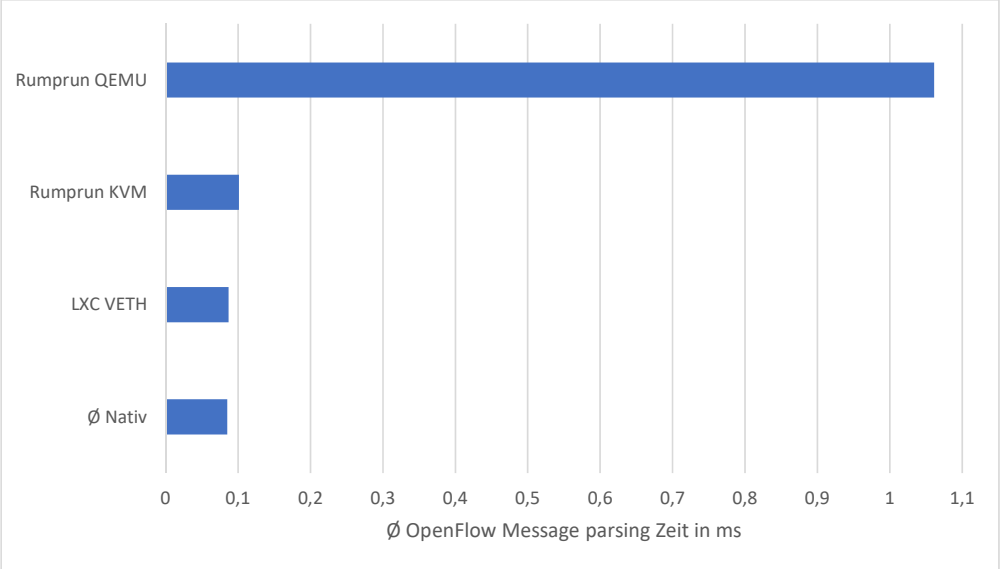


Abbildung 5.4: Ergebnisse der OpenFlow-Decode-Zeitmessung mit Ryu: LXC und native Ausführung sind ähnlich schnell. Ruprum ist selbst mit KVM minimal langsamer und bricht ohne KVM weiter ein.

5.2.3 Open vSwitch

Als letzter Teil der OpenFlow Performance Evaluation wurde dann noch die Zeit für das Encodieren und Decodieren von OpenFlow-Nachrichten Switchseitig beim Open vSwitch analysiert. Dazu wurde das Open vSwitch 2.5.1 LTS Relase mit den für die Evaluation nötigen Modifikationen zur Zeitmessung erweitert und nativ auf ONL gebaut und ausgeführt. Das erlaubte es ohne zu große Modifikationen am Code von Open

OS	Virtualisierung	Zeit in ms
PicOS:		
	Nativ	0,085
	Rumprun KVM	0,101
ONL V1 (3.2):		
	Nativ	0,085
	LXC VETH	0,087
	Rumprun QEMU	1,061
ONL V2 (3.18):		
	Nativ	-
	Docker	-

Tabelle 5.3: Messergebnisse für die durchschnittliche Zeit (in Millisekunden) die Ryu für das Decodieren einer OpenFlow-Nachricht benötigt

vSwitch über einen Ryu basierten SDN-Controller das Encodieren und Decodieren von OpenFlow-Nachrichten anzustoßen. Dabei zeigte sich, dass die Implementierung des in C geschriebenen Open vSwitch sehr schnell ist.

Die Messung eines einzigen Aufrufs einer Decode oder Encode Funktion lies sich dabei nicht stoppen, da sämtliche Zeitmessfunktionen Probleme haben Aktionen <1 ms genau zu erfassen und durch unterschiedliche Auslastung der CPU in diesem feinen Bereich auch sehr viele Schwankungen auftreten. Um möglichst exakte Messergebnisse zu erhalten wurde deswegen der Code des Open vSwitch so modifiziert, dass wenn von außen der Aufruf einer Decode oder Encode Funktion ausgelöst wurde, dieser viele tausend Mal ausgeführt wurde. Dabei wurde und die Zeit dabei gestoppt und durch das Teilen der Gesamtzeit für z. B. 10000 Ausführungen durch 10000 konnte dann die durchschnittliche Zeit, für die Ausführung von einem Aufruf ermittelt werden. Zum Messen der Zeit wurde dabei `clock_gettime()` verwendet das mit einer Auflösung im Nanosekundenbereich genauere Ergebnisse liefert als `clock()` das mit Millisekunden arbeitet. Im Prinzip ist der Unterschied zwischen diesen aber unbedeutend, da sowieso sehr viele Wiederholungen nötig waren.

Die einfache OpenFlow `Hello` Message die nur aus den die beim Verbindungsaufbau ausgetauscht wird wurde dabei als erstes gemessen, da der Erwartung nach es die am schnellsten zu verarbeitende Nachricht sein sollte. Hierbei war das Encodieren der Nachricht mit 0,407 μs sogar deutlich teuer als das Decodieren mit 0,015 μs. Das liegt vermutlich daran, dass für das Decodieren einer so einfachen Nachricht fast keine Zeit gebraucht wird und alle Daten schon im Speicher existieren. Das Encodieren einer so einfachen Zeit sollte zwar ebenfalls fast keine Zeit kosten aber das unvermeidbare Anlegen des Speicherbereichs für die Nachricht kostet zusätzlich Zeit.

Das Nächste war die Messung einer `FlowMod` Message, da dies eine relativ komplexe Nachricht ist die außerdem sehr häufig empfangen wird. Dazu wurde die Decode-Methode im

Open vSwitch die eine eingehende `FlowMod` Message entpackt und den Inhalt zur Weiterverarbeitung aufbereitet analysiert. Mit 0,554 μs geschieht auch das sehr schnell.

Anschließend wurde auch noch die Zeit für das Erstellen einer `PaketIn` Message gemessen. Diese dient zum Weiterleiten eines Paktes an den SDN-Controller. Mit 1,606 μs dauerte das relativ lange was vermutlich daran liegt, dass zu einem eine Nachricht erstellt werden muss und zu anderen eine komplette andere Nachricht in diese eingefügt werden muss, was es erfordert relativ viel Speicher anzulegen und viele Bytes zu kopieren.

Abschließend lässt sich festhalten, dass der OpenFlow Overhead im Open vSwitch auf jeden Fall nur sehr gering ist. Für viele Nachrichten scheint er bei unter 1 μs zu liegen und nur aufwendige Funktionen wie das Weiterleiten eines Pakets scheinen über 1 μs zu kommen.

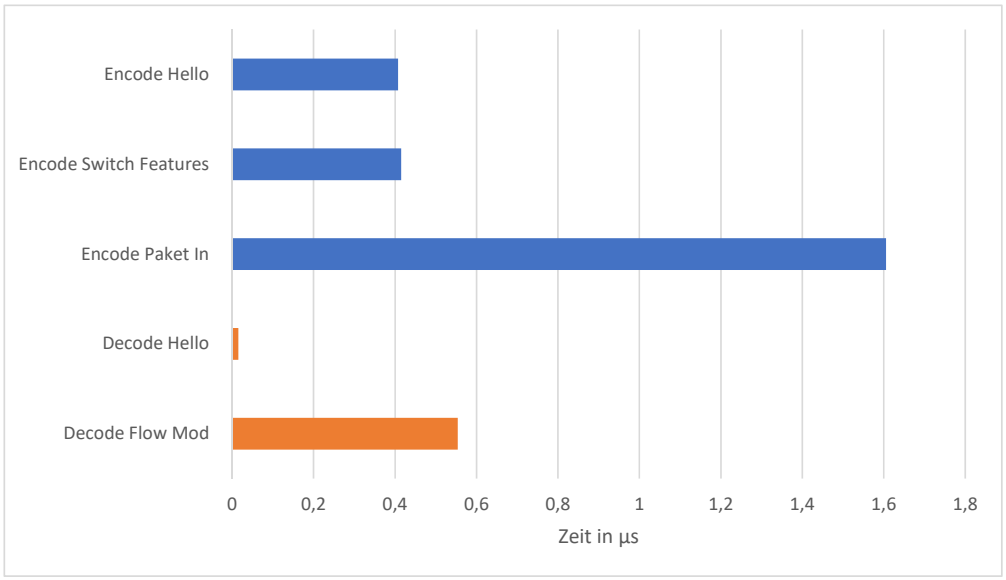


Abbildung 5.5: Ergebnisse der Decode- und Encode-Zeitmessung mit Open vSwitch

Aktion	**Zeit in ms**
Encode `Hello`	0,407
Encode `SwitchFeatures`	0,415
Encode `PaketIn`	1,606
Decode `Hello`	0,015
Decode `FlowMod`	0,554

Tabelle 5.4: Messergebnisse für die durchschnittliche Zeit (in Mikrosekunden) die Open vSwitch für das Decodieren bzw. Encodieren einer OpenFlow-Nachricht benötigt

Damit ist die Geschwindigkeit der C-Implementierung den Python basierten Lösungen weit voraus. Die Gründe dafür sind die Ausführung des Python-Codes über die Python-Runtime und ein vermutlich weniger effizientes Memory Management beim Durchreichen und Verarbeiten der Nachrichten. Kommt es also auf maximale Effizienz an ist eine C basierte Lösung deutlich im Vorteil.

5.2.4 Fazit OpenFlow Performance Evaluation

Für die Evaluation der OpenFlow-Performance wurde die twink Library, Ryu und Open vSwitch betrachtet. Dabei zeigte sich ganz klar, dass die Performance beim Encodieren und Decodieren von OpenFlow-Nachrichten sehr stark von der Effizienz der Implementierung abhängt. Die recht simple und auch nicht mehr wirklich aktiv weiterentwickelte in Python geschriebene twink Library würde beim Einsatz von OpenFlow deswegen unnötig viel Overhead erzeugen. Die ebenfalls in Python geschriebene Ryu-Implementierung zum parsen von OpenFlow ist bereits deutlich effizienter. Dennoch besteht auch hier ein deutlicher Overhead. Open vSwitch das in C implementiert ist kann hier noch einmal deutlich mehr Performance erreichen und ist deutlich schneller als die beiden Python-Lösungen. Dafür ist der Einstieg mit Ryu und Python aber auch viel einfacher und komfortabler. Für möglichst wenig Overhead und somit möglichst hohe Effizienz ist es also wichtig eine schnelle OpenFlow-Implementierung zu nutzen. Eine in C oder einer vergleichbaren Sprache geschriebene und gut optimierte OpenFlow-Implementierung bzw. SDN-Controller-Plattform ist also vorzuziehen. Bei einer guten Implementierung ist der Overhead für den Einsatz von OpenFlow überschaubar. Bei einer Schlechten kann die Performance aber deutlich leiden. Die Implementierung eines eigenen Protokolls muss also sehr gut sein und in einer schlanken Programmiersprache geschrieben sein, da sonst vermutlich keine Vorteile erreicht werden können. In den meisten Fällen sollte deswegen ohne Probleme eine gute OpenFlow-Implementierung genutzt werden können, ohne dass dadurch zu viel Overhead entsteht. Die Vorteile den lokalen Controller über ein etabliertes Protokoll anzubinden sollten dabei dann gegenüber des OpenFlow Overhead überwiegen.

5.3 Evaluation von Anwendungsszenarien

Nach der Evaluation einzelner Teile folgt nun noch die Evaluation der drei bereits im Kapitel 4 beschriebenen Anwendungsfällen. Als erstes kommt die Evaluation eines einfachen Simple-Switch-Szenarios anschließend werden noch Port Knocking und Fast Failover evaluiert. Als Grundlage dafür findet an dieser Stelle die Evaluation der Zeit, die für die Kommunikation mit dem SDN-Controller nötig ist, statt. Diese hat einen wesentlichen Einfluss auf die folgenden Anwendungsszenarien, da sie in allen Messungen direkt oder indirekt enthalten ist.

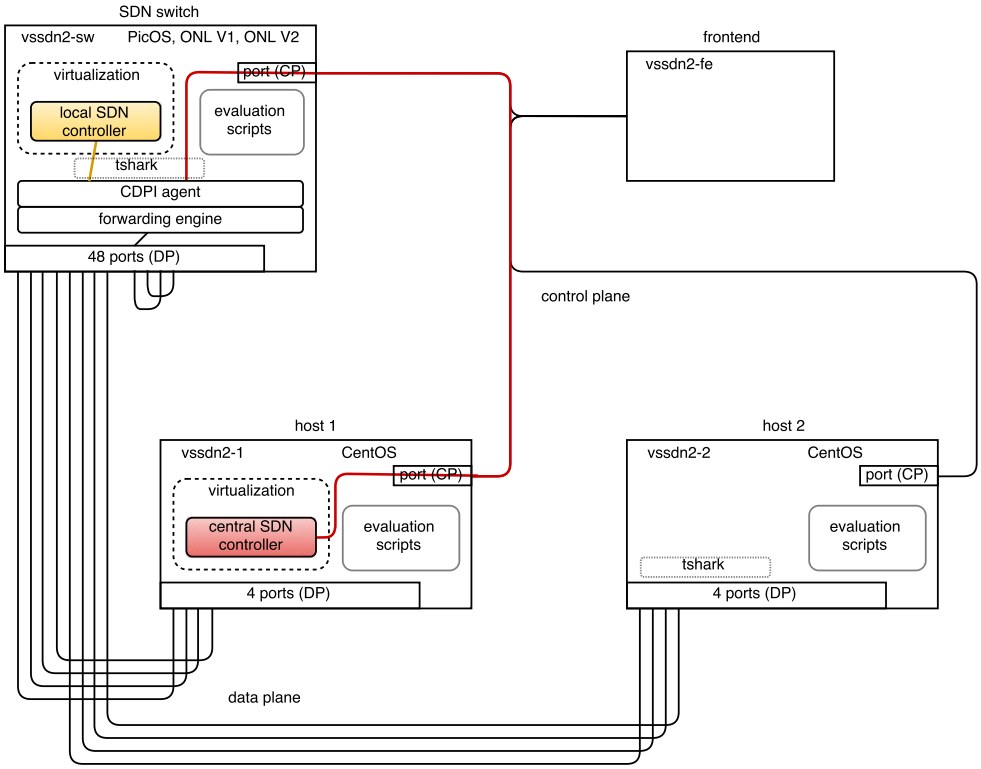


Abbildung 5.6: Versuchsaufbau zum Messen der Zeit für die Kommunikation mit dem lokalen bzw. zentralen SDN-Controller

Zur Messung der Zeitspanne die zur Kommunikation mit dem SDN-Controller nötig ist wird die Zeit gemessen, die vergeht bis auf eine Nachricht vom OpenFlow Agent an den SDN-Controller dessen Antwort beim OpenFlow Agent ankommt. Dazu wurde mit `tshark` der Paketverkehr mitgeschnitten und die Zeit zwischen einer Nachricht die vom OpenFlow-Agent ausgeht bis zum Eingehen einer `Flow Mod`-Nachricht als Reaktion darauf gemessen.

Da die Latenz zum zentralen SDN-Controller dabei erheblichen Einfluss hat werden verschiedene RTT zum zentralen SDN-Controller emuliert. Dazu wird das Shell Tool `tc` verwendet, das es erlaubt das Traffic Shaping des Linux-Kernels zu konfigurieren. Somit konnte nicht nur mir der von der reellen Hardware Latenz vorgegeben Verzögerung, sondern auch mit auf 2, 4, 6 und 8 ms erhöhter Verzögerung zum zentralen SDN-Controller gemessen werden. Abbildung 5.6 zeigt den Versuchsaufbau als Skizze. In gelb ist der lokale SDN-Controller markiert, in rot der zentrale SDN Controller.

Tabelle 5.5 zeigt die dabei gemessen Zeiten bis auf eine gesendete Nachricht die Antwort vom Controller zurück kommt. Dabei ist gut zu sehen, dass die Virtualisierung mit Rumprun in diesem Fall knapp 2 ms kostet. Vermutlich aufgrund der begrenzten Leistung des Switches ist außerdem der zentrale SDN-Controller bei einer nur sehr geringen Latenz

schneller. Mit steigender Latenz zum zentralen SDN-Controller wird der lokale SDN-Controller dessen Zeit konstant bleibt aber deutlich schneller. Sehr deutlich sieht man dies im Diagramm in Abbildung 5.7.

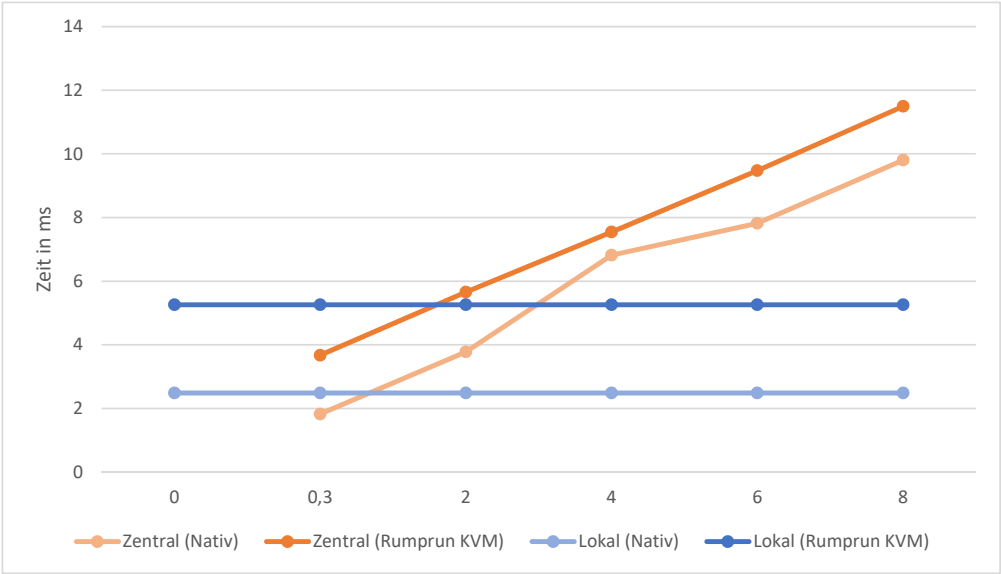


Abbildung 5.7: Ergebnisse der Zeitmessung für die Kommunikation mit dem lokalen bzw. zentralen SDN-Controller: Die Zeit für den zentralen SDN-Controller steigt mit der Latenz zu diesem, die des lokalen SDN-Controllers bleibt konstant. Die Virtualisierung sorgt in beiden Fällen für einen geringfügigen Overhead.

Ansatz | RTT zum SDN-Controller	**Lokal**	**0,3 ms**	**2 ms**	**4 ms**	**6 ms**	**8 ms**
Zentral (Nativ)		1,826	3,779	6,816	7,818	9,805
Zentral (Rumprun KVM)		3,675	5,660	7,541	9,475	11,493
Lokal (Nativ)	2,486					
Lokal (Rumprun KVM)	5,259					

Tabelle 5.5: Messergebnisse für die Zeit (in Millisekunden) zwischen dem Absenden einer OpenFlow-Nachricht an den SDN-Controller und einer Antwort darauf.

5.3.1 Simple Switch

Die Simple-Switch-Implementierung wurde bereits in Kapitel 4 vorgestellt. Darauf aufbauend folgt nun die Evaluation des Simple-Switch-Anwendungsszenarios. Dazu wird die RTT gemessen, die das erste TCP-Paket hat, das von einem Sender zu einem Empfänger gesendet wird.

Dazu wird auf `vssdn2-2` ein Python3-Skript ausgeführt, das als TCP Responder dient, und auf `vssdn2-1` ein Skript, das ein TCP-Paket sendet und die RTT misst. Als SDN-Controller wird die im Implementierungsteil erstelle Simple-Switch-Implementierung verwendet. Gemessen wird dann die Performance mit einem nativ ausgeführten lokalen SDN-Controller auf dem Switch (`vssdn2-sw`) und wenn dieser mit verschiedenen Technologien virtualisiert ist. Zum Vergleich mit dem klassischen SDN-Ansatz folgt dann außerdem die Evaluation mit einem zentralen statt lokalem SDN-Controller wozu der SDN-Controller auf `vssdn2-1` ausgeführt wird. Da für die Simple-Switch-Evaluation nur die erste RTT, also die wo noch kein Flow installiert ist, interessant ist werden zwischen den einzelnen Tests immer alle Flows zurückgesetzt. Da für den zentralen SDN-Controller-Ansatz außerdem die Latenz zum SDN-Controller relevant ist wird neben der Verzögerung durch die reale Hardware von 0,3ms außerdem mit einer künstlich erhöhten Verzögerung von 2, 4, 6 und 8 ms gemessen.

Abbildung 5.8 skizziert die Testumgebung. In grün die ausgeführten Python-Skripte für die Evaluation und in blau die Leitungen über die diese auf Ebene der Data Plane kommunizieren. Außerdem ist der lokale SDN-Controller in gelb hervorgehoben und der zentrale SDN-Controller so wie der Weg über die Control Plane zu diesem in rot.

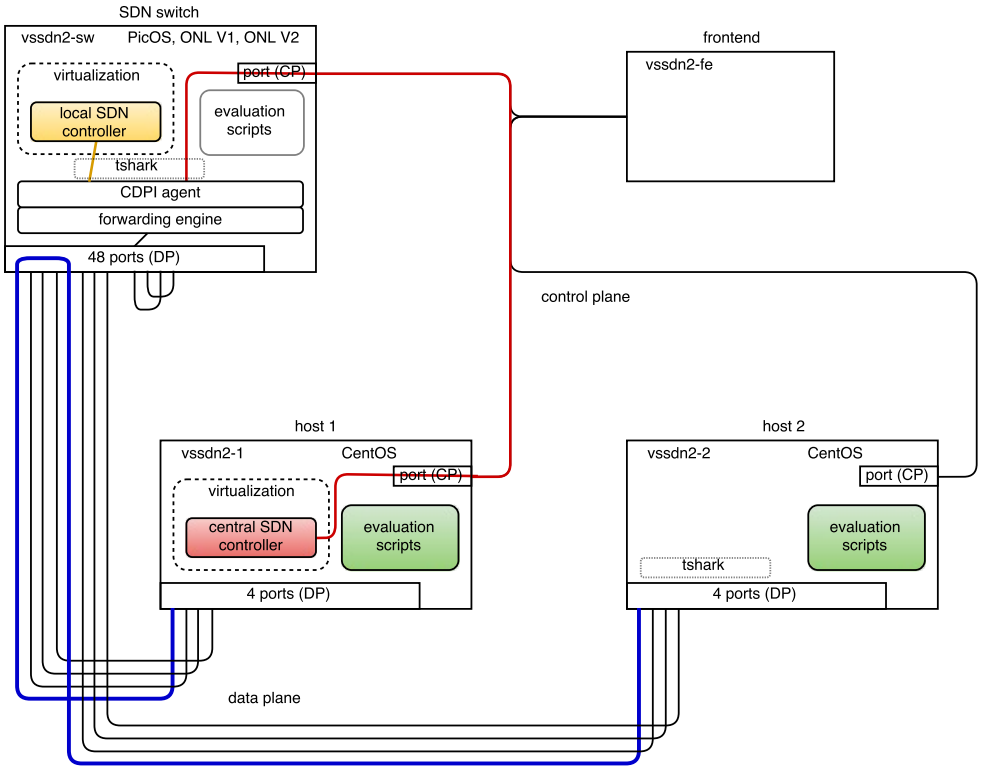


Abbildung 5.8: Versuchsaufbau für das Simple-Switch-Anwendungsszenario

Alle Testkonfigurationen wurden dabei mehrfach durchgeführt um ein möglichst exaktes

Ergebnis zu erzielen. Dabei zeigte sich, dass bereits relativ wenige Ausführungen ein gutes Ergebnis liegen und Abweichungen meist deutlich unter 1ms liegen.

Tabelle 5.8 zeigt die dabei gemessen Durchschnittswerte unter PicOS. Der lokale Ansatz ist zwar sobald er virtualisiert wird langsamer, allerdings wird er schnell wieder deutlich performanter wenn die Verzögerung zum zentralen SDN-Controller steigt. Vergleicht man mit einem ebenfalls Virtualisierten zentralen SDN-Controller ist der lokale Ansatz immer schneller.

Wie in Tabelle 5.8 zu sehen ist kostet die Virtualisierung des lokalen SDN-Controller auf `vssdn2-sw` ca. 1 Millisekunde. Dies ist zwar deutlich messbar sollte in der Praxis aber nicht zu sehr in Gewicht fallen, vor allem da eine steigende Latenz zum SDN-Controller diesen Nachteil sehr schnell aufwiegt und den lokalen Ansatz deutlich schneller macht wie in Abbildung 5.9 zu sehen ist.

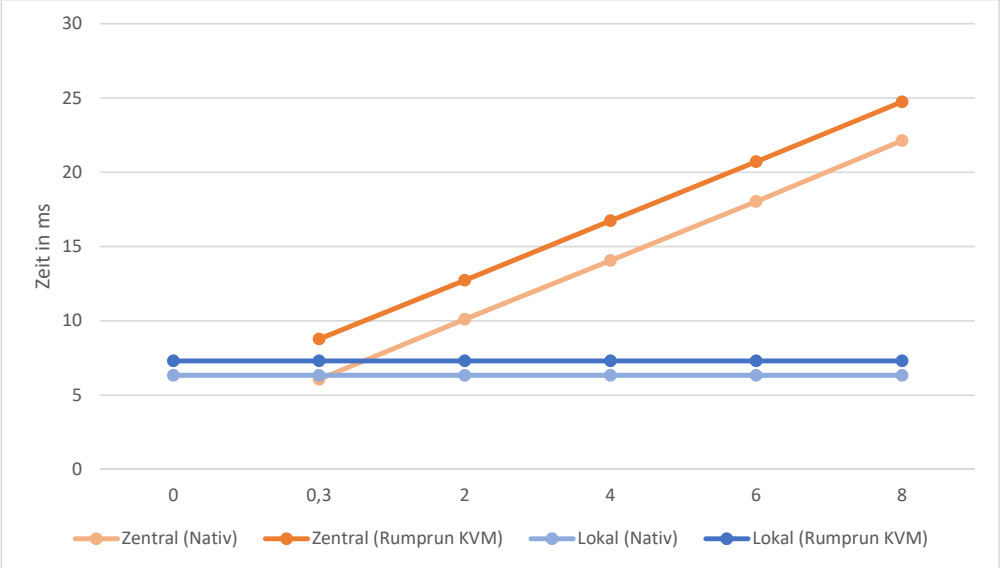


Abbildung 5.9: Ergebnisse der RTT-Messung des Simple-Switch-Anwendungsszenarios: Nativ sind lokale und zentrale Variante fast gleich schnell ehe mit steigender Latenz der zentrale Ansatz langsamer wird. Mit Rumprum ist der lokale Ansatz von Anfang an schneller.

Abbildung 5.9 zeigt die Ergebnisse noch als Diagramm. Dabei sieht man wie mit steigender Verzögerung zum zentralen SDN-Controller die zentrale SDN-Controller-Variante immer langsamer wird während der lokale SDN-Controller-Ansatz davon natürlich nicht betroffen ist.

Ansatz | RTT zum SDN-Controller	Lokal	0,3 ms	2 ms	4 ms	6 ms	8 ms
Zentral (Nativ)		6,058	10,102	14,060	18,029	22,128
Zentral (Rumprun KVM)		8,772	12,727	16,731	20,718	24,744
Lokal (Nativ)	6,331					
Lokal (Rumprun KVM)	7,296					

Tabelle 5.6: Messergebnisse für die RTT (in Millisekunden) für das erste Paket beim Simple-Swtich-Anwendungsszenario

5.3.2 Port Knocking

Die Port-Knocking-Implementierung läuft auf der gleichen Testumgebung wie das Simple-Switch-Szenario dessen Aufbau in Abbildung 5.8 dargestellt ist. Gemessen wurde in diesem Fall die Zeit die benötigt wird um mit Hilfe der Port-Knocking-Sequenz einen Port zu öffnen und eine Antwort auf ein über den Port gesendetes Paket zu bekommen.

Dazu wurde wieder ein Python3-Skript auf `vssdn2-1` ausgeführt, das als Sender diente. Dieses sendet bei Ausführung als erstes die Port-Knocking-Sequenz wobei zwischen jedem UDP-Paket der Port-Knocking-Sequenz je 2 ms gewartet wurde ehe es dann auf die Antwort eines gesendeten Pakets wartet. Auf `vssdn2-2` kam dabei wie auch im Simple-Switch-Beispiel ein einfaches Responder-Skript dieses Mal aber UDP basiert zum Einsatz, das ankommende Pakete direkt wieder zum Empfänger zurückschickt.

Tabelle 5.8 zeigt die dabei gemessen Durchschnittswerte unter PicOS. Dabei wurde mit 5 Paketen gearbeitet, wovon das 5. wenn die 4 vorherigen Pakete die korrekte Port-Knocking-Sequenz repräsentierten auf dem freigeschalteten Port weitergeleitet wurde. Die Abweichungen bei den einzelnen Messungen waren dabei kleiner 0,1 ms.

Ansatz | RTT zum SDN-Controller	Lokal	0,3 ms	2 ms	4 ms	6 ms	8 ms
Zentral (Nativ)		14,620	18,689	22,669	26,644	30,823
Zentral (Rumprun KVM)		15,074	19,250	23,268	27,325	31,766
Lokal (Nativ)	15,687					
Lokal (Rumprun KVM)	17,061					

Tabelle 5.7: Messergebnisse für die Zeit (in Millisekunden) beim Port-Knocking-Anwendungsszenario mit 5 Nachrichten

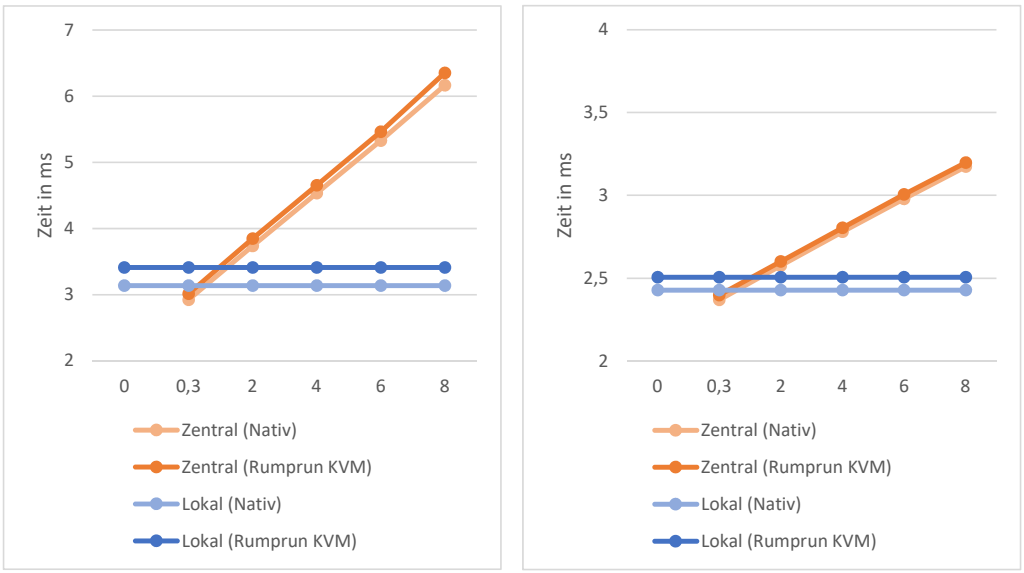
Da die Pakete dabei in einem festen Intervall gesendet werden dauert das Senden der Port-Knocking-Sequenz mit länger werdender Sequenz natürlich länger. Zum Vergleich wurde deshalb noch mit einer längeren Port-Knocking-Sequenz derselbe Test durchgeführt. Die Ergebnisse des Tests sind in Tabelle 5.8 abgebildet. Die Abweichungen waren auch dabei wieder kleiner 0,1 ms und es wurde mit 20 Paketen gearbeitet, wovon das 20. wenn die 19 vorherigen Pakete die korrekte Port-Knocking-Sequenz repräsentierten

weitergeleitet wurde. Wie zuvor entsprach der Abstand zwischen den Paketen wieder 2 ms.

Ansatz | RTT zum SDN-Controller	Lokal	0,3 ms	2 ms	4 ms	6 ms	8 ms
Zentral (Nativ)		47,369	51,508	55,590	59,586	63,464
Zentral (Rumprun KVM)		47,954	52,012	56,082	60,096	63,953
Lokal (Nativ)	48,555					
Lokal (Rumprun KVM)	50,115					

Tabelle 5.8: Messergebnisse für die Zeit (in Millisekunden) beim Port-Knocking-Anwendungsszenario mit 20 Nachrichten

Abbildung 5.10 zeigt die Kosten je Paket, also die gemessene Zeit geteilt durch die Anzahl gesendeter Pakete. Da der Overhead für die Kommunikation zum SDN-Controller unabhängig von der Länge der Port-Knocking-Sequenz ist, da die Pakete ja in einem festen Intervall gesendet werden, schneidet wie erwartet die Variante mit der längeren Sequenz dabei besser ab. Je länger die Port-Knocking-Sequenz desto weniger fällt die Latenz zum SDN-Controller also ins Gewichtt.



(a) Zeit je Nachricht bei bei 5 Nachrichten (b) Zeit je Nachricht bei bei 20 Nachrichten

Abbildung 5.10: Vergleich des Port Knocking Overhead

Interessant ist hier eher der Vorteil, dass beim lokalen Ansatz keine Pakete über die Con-

trol Plane zum SDN-Controller geschickt werden müssen. Damit spart man auf dem Weg zum zentralen SDN-Controller so viele Pakete ein wie die Port-Knocking-Sequenz lang ist. Dies verteilt die Last deutlich besser als wenn wie bei einem zentralen SDN-Controller die weitergeleiteten Pakete verschiedener Switches, an denen Clients anklopfen, verarbeitet werden müssen. Besonders dann, wenn lange Port-Knocking-Sequenzen verwendet werden.

5.3.3 Fast Failover

Der letzte Anwendungsfall der evaluiert wurde war das Fast-Failover-Szenario. Im Gegensatz zu den anderen Anwendungsfällen wurde dabei aber nicht die Zeit gestoppt, sondern gemessen wie viele UDP-Pakete das Ziel erreichen bzw. wie viele verloren gehen.

Für den Fast-Failover-Anwendungsfalls war eine Anpassung der Testumgebung nötig, da nun zwei redundante Verbindungen nötig waren. Abbildung 5.12 zeigt die neue Testumgebung als Skizze. In grün wieder die ausgeführten Python-Skripte für die Evaluation und in blau die Leitungen über die diese auf Ebene der Data Plane kommunizieren. Außerdem ist der lokale SDN-Controller in gelb hervorgehoben und der zentrale SDN-Controller so wie der Weg über die Control Plane zu diesem in rot.

Das zusätzliche Skript für die Evaluation auf dem Switch das gegenüber den vorherigen Testfällen hinzu kommt dient zur Simulation von Ausfällen einer der beiden redundanten Verbindungen. Der Switch lief dazu mit zwei über Open vSwitch konfigurierten Bridges und simulierte so zwei Switches die über eine redundante Verbindung verfügen. Dazu waren von Port 18 zu 20 und Port 19 zu 21 zwei Loops eingerichtet die in der Abbildung 4.5 lila dargestellt sind. Bridge `br0`, die den ersten Switch simuliert, hatte als Ports die Verbindung zu `vssdn2-1` mit Port 1 und mit den redundanten Ports 18 und 19 eine Verbindung zum anderen Simulierten Switch. Die Bridge `br1` hatte sich daraus ergebend die Ports 20 und 21 als redundante Verbindung zu `br0` sowie eine Verbindung zu `vssdn2-2` über Port 5. Aufgabe des Skripts auf dem Switch war es nun die Verbindung von Port 18 zu 20 bzw. die Verbindung von Port 19 zu 21 im Wechsel zu unterbrechen und wiederherzustellen. Die Pausen dazwischen wurden so gewählt, dass alle 6 Sekunden eine der Verbindungen unterbrochen wurde.

Gemessen wurde dabei wie groß die Paketverluste bei einer UDP-Übertragung von `vssdn2-1` zu `vssdn2-2` ist. Dazu lief auf `vssdn2-1` ein in Python3 geschriebenes Skript, das pro Millisekunde 10 Pakete mir einer Größe von 500 Byte an `vssdn2-2` sendete bis in Summe 1000000 Pakete verschickt wurden. Auf `vssdn2-2` lief ein Python3-Skript das als Empfänger diente und die Anzahl eingehender Pakete zählte. Die Anzahl gesendeter Pakte abzüglich der Anzahl empfangener Pakete ergab dann die Anzahl verlorengegangener Pakete. Geteilt durch die Anzahl gesendeter Pakete ergab sich dann der Prozentsatz verlorener Pakete.

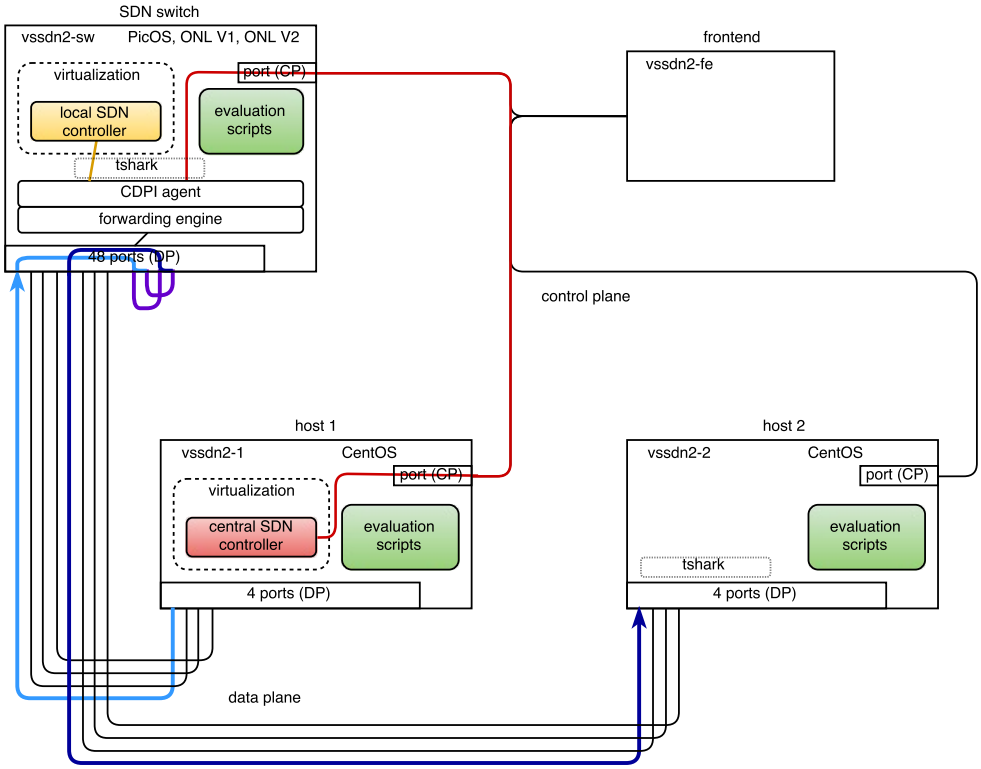


Abbildung 5.11: Versuchsaufbau für das Fast-Failover-Anwendungsszenario

Tabelle 5.9 zeigt die dabei gemessen Paketverlustraten unter PicOS. Dabei zeigt sich, dass auch hier der lokale Ansatz Vorteile bietet, vor allem dann, wenn die Verzögerung zum zentralen SDN-Controller steigt. Abbildung 5.12 zeigt dazu die Unterschiede in Prozent zur lokalen nativen Variante.

Ansatz | RTT zum SDN-Controller	Lokal	0,3 ms	2 ms	4 ms	6 ms	8 ms
Zentral (Nativ)		21,00%	21,63%	21,91%	22,14%	22,44%
Zentral (Rumprun KVM)		21,41%	21,84%	22,07%	22,38%	23,14%
Lokal (Nativ)	20,70%					
Lokal (Rumprun KVM)	21,12%					

Tabelle 5.9: Messergebnisse der Paketverlustrate für UDP-Kommunikation zwischen Sender und Empfänger beim Fast-Failover-Anwendungsszenario

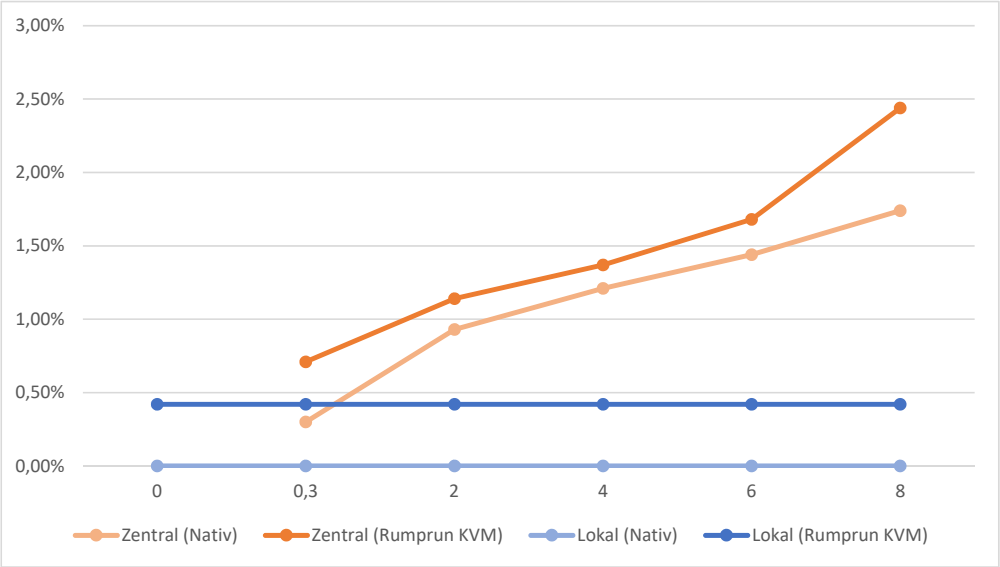


Abbildung 5.12: Erhöhung der Paketverlustrate im Vergleich

5.3.4 Fazit

Bei der Evaluation der Anwendungsszenarien zeigte sich schnell, dass die lokale Ausführung manchmal minimal langsamer ist als ein sehr gut angebundener zentraler SDN-Controller. Das liegt daran, dass die Latenz zum zentralen SDN-Controller, wenn diese sehr weit unter 1 ms liegt nicht wirklich in Gewicht fällt. Ein weiter Grund ist, dass die rein lokale Verarbeitung mehr Last auf der CPU des Switches verursacht. Forwarding Engine und lokaler SDN-Controller laufen ja nun beide auf dem Switch und teilen sich dessen Leistung. Steigt die Latenz zum zentralen SDN-Controller ist aber die lokale Variante viel vorteilhafter und ist im Vergleich schnell erheblich performanter.

Für den Overhead des OpenFlow-Protokolls zeigte sich, dass dieser stark von der Implementierung abhängt. Es lässt sich deshalb festhalten, dass bei einer guten Implementierung wie dem in C geschriebenen Open vSwitch der OpenFlow Overhead nicht störend auffällt. Bei der Wahl einer OpenFlow Library sollte also auf eine schnelle und schlanke Implementierung geachtet werden.

Der Overhead durch die Virtualisierung liegt meist bei ca. 1 ms. Das ist aber kein großer Nachteil und verschiebt den Punkt ab welchem die höhere Latenz zum zentralen SDN-Controller diesen langsamer macht nur minimal. Die Vorteile von Virtualisierung dürften daher in den meisten Fällen überwiegen.

In einem kleinen Netzwerk mit einem sehr schnell und ohne große Latenz angebundenem zentralen SDN-Controller bietet der Ansatz eines virtualisierten lokalen SDN-Controllers daher nur wenige Vorteile. Je größer die Latenz zum zentralen SDN-Controller wird desto

größer wird aber der Performance Gewinn durch den lokalen SDN-Controller-Ansatz. Für große Netze wo auch die Latenz zum zentralen SDN-Controller steigt bietet der Ansatz eines virtualisierten lokalen SDN-Controllers deshalb deutliche Vorteile.

6 Zusammenfassung und Ausblick

Im Folgenden werden die wesentlichen Punkte und Erkenntnisse dieser Arbeit noch einmal zusammengefasst und anschließend noch ein kurzer Ausblick gegeben.

Wie bereits in der Einleitung angemerkt steigen die Anforderungen an klassische IT-Netzwerke immer weiter. Das Software-defined Networking, das eine Trennung in Data und Control Plane einführt, auf offene Standards setzt und einen logisch zentralisierten SDN-Controller ermöglicht, gewinnt deswegen immer mehr an Bedeutung. Ein Vorteil davon sind relativ einfache SDN-Switches, die Daten anhand vom logisch zentralisierten SDN-Controller vorgegeben Regeln weiterleiten.

Wie in Kapitel 3 beschrieben gibt es aber auch viele Situationen in denen weiterhin eine lokale Verarbeitung wie bei klassischen Switches möglich wäre. Eine lokale Verarbeitung bietet dabei die Chance die Latenz zum logisch zentralisierten SDN-Controller zu vermeiden und gleichzeitig dessen Last zu reduzieren.

Um die Vorteile lokaler Logik beim Software-defined Networking nutzen zu können wurde in Kapitel 3 ein Konzept für lokale Entscheidungen auf SDN-Switches entworfen. Dazu wurde der Ansatz eines virtualisierten lokalen SDN-Controllers auf dem SDN-Switch gewählt. Dabei wird mit Hilfe von Virtualisierungstechnologie, wie z. B. Containern oder Unikernels, ein vom Rest des SDN-Switch weitgehend isolierter SDN-Controller wie z. B. Ryu ausgeführt. Der CDPI Agent des SDN-Switches verbindet sich dann statt mit dem logisch zentralisierten SDN-Controller mit dem virtualisierten lokalen SDN-Controller. Dieser wiederum kann sich mit dem logisch zentralisierten SDN-Controller verbinden, so dass der virtualisierte lokale SDN-Controller im Prinzip einfach zwischen den CDPI Agent des SDN-Switches und den logisch zentralisierten SDN-Controller geschaltet ist. Der virtualisierte lokale SDN-Controller-Ansatz bietet dadurch viele Vorteile:

- Geringere bzw. von der Verzögerung zum logisch zentralisierten SDN-Controller unabhängige Latenz für Aktionen, die vom virtualisierten lokalen SDN-Controller verarbeitet werden können.
- Weniger Traffic vom Switch zum logisch zentralisierten SDN-Controller, da der virtualisierte lokale SDN-Controller viele der Anfragen vom OpenFlow Agent direkt lokal verarbeiten kann, so dass keine Nachricht zu diesem gesendet werden muss.
- Geringere Last für den logisch zentralisierten SDN-Controller, da ihm der virtualisierte lokale SDN-Controller lokal durchführbare Arbeit abnimmt. Es wäre sogar eine hierarchische Organisation der virtualisierten lokalen SDN-Controller denkbar.

- Erprobte dezentrale Protokolle aus klassischen Netzwerken können relativ leicht portiert und auf dem virtualisierten lokalen SDN-Controller genutzt werden.
- Bessere Fehlertoleranz, da rein lokale Funktionen auch bei Ausfall des logisch zentralisierten SDN-Controllers erhalten bleiben.

Es stellen sich dabei aber auch viele Fragen z. B. in Bezug auf die Effizienz der Virtualisierung, den Overhead der Kommunikation, den geeignetsten SDN-Controller und den praktischen Nutzen, weshalb eine umfangreiche Evaluation folgte.

Beim Vergleich und der Analyse der Virtualisierungstechnologien in Kapitel 4 und 5 zeigte sich, dass sowohl Container als auch Unikernel geeignet sind. Bei den Containern kommen die zwei großen und bekannten Vertreter LXC und Docker für den Einsatz in Frage. Bei den Unikernels ist die Auswahl theoretisch größer, sucht man aber nach einer möglichst allgemeinen Lösung, die keine großen Einschränkungen hat, bleibt im Prinzip nur Rumprun. Die anderen Unikernel-Projekte sind noch zu unausgereift oder speziell für eine Programmiersprache oder ein Anwendungsgebiet gedacht.

Die Container können wie die Evaluation zeigte vor allem mit einer hohen Performance punkten, die sich kaum von der nativen Ausführung unterscheidet. Der Grund ist der gemeinsame Kernel mit dem Hostbetriebssystem, wodurch kaum Overhead anfällt. Der Nachteil davon ist, dass Container Anforderungen, wie eine bestimmte minimale Linux-Kernel-Version oder aktive `cgroups`, an das Betriebssystem stellen. Dies führt dazu, dass Container nicht überall eingesetzt werden können, gerade in Bezug auf Betriebssysteme für SDN-Switches, die unter Umständen noch ältere Linux Kernel Versionen nutzen oder um möglichst leichtgewichtig zu sein nicht alle nötigen Features bieten.

Unikernel wie Rumprun punkten wie sich zeigte vor allem bei der Plattformunabhängigkeit, da sie sowohl auf echter als auch virtueller Hardware, z. B. mit einem Hypervisor wie QEMU, ausgeführt werden können. Die Unikernel bauen dabei auf der Idee auf, anstatt einem ganzen Betriebssystem wie bei klassischen virtuellen Maschinen nur die für Anwendung benötigten Teile eines Betriebssystems, das in Form vom Libraries vorliegt, auszuführen. Der Nachteil dabei besteht darin, dass die Anwendung für den Unikernel extra kompiliert und um Betriebssystemkomponenten erweitert werden muss. Dazu muss die Anwendung komplett als Sourcecode vorliegen und mit den durch den Unikernel gegebenen Einschränkungen kompatibel sein. Die Performance hängt dabei stark vom Hypervisor bzw. der Unterstützung durch die darunterliegende Hardware ab. Mit QEMU/KVM ist Rumprun fast so schnell wie die Containerlösungen, bei fehlendem KVM Support bricht die Performance hingegen deutlich ein.

Bei der Betrachtung der vielen verschiedenen verfügbaren SDN-Controller in Kapitel 4 stellte sich außerdem heraus, dass die Auswahl zwar sehr groß ist aber nicht jeder SDN-Controller für jeden Anwendungsfall gleich gut geeignet ist und viele nicht aktiv weiterentwickelt werden. Die großen und bekannten SDN-Controller wie z. B. Ryu, Floodlight und NOX sind aber allgemein genug aufgebaut um theoretisch für jeden Anwendungsfall verwendbar zu sein.

Bei der darauffolgenden Evaluation des OpenFlow-Protokolls in Kapitel 5 zeigte sich, dass bei einer guten Implementierung wie z. B. beim in C geschriebenen Open vSwitch nur wenig Overhead entsteht. Wird eine gute OpenFlow-Implementierung genutzt sollte der Einsatz von OpenFlow als CDPI-Protokoll kein Problem darstellen. Somit sind auch keine Anpassungen am CDPI Agent nötig, was die Anbindung vereinfacht.

Im Anschluss folgte dann die Evaluation von Anwendungsszenarien, die am Ende von Kapitel 5 dokumentiert ist. Dabei zeigte sich, dass der Ansatz eines virtualisierten lokalen SDN-Controllers in allen untersuchten Anwendungsszenarien auch Vorteile bringt. Ist der logisch zentralisierte SDN-Controller sehr gut und mit einer Latenz von deutlich unter 1 ms angebunden ist der Ansatz eines virtualisierten lokalen SDN-Controllers zwar teilweise noch minimal langsamer aber mit steigender Latenz zum logisch zentralisierten SDN-Controller ist dieser schnell deutlich langsamer. Das kommt daher, dass sich der virtualisierte lokale SDN-Controller die Ressourcen des SDN-Switches mit der anderen Software darauf teilt und auch die Virtualisierung, deren Vorteile aber deutlich überwiegen, einen kleinen Overhead erzeugt. In sehr kleinen und schnellen Netzwerken dürfte der Ansatz eines virtualisierten lokalen SDN-Controllers deswegen nur geringe Vorteile bringen. In großen Netzwerken hingegen, wo es auch eine große Latenz zum zentralen SDN-Controller gibt und dieser durch eine größere Anzahl SDN-Switches stärker ausgelastet ist, kann der Ansatz eines virtualisierten lokalen SDN-Controllers aber durchaus große Vorteile bringen.

Der Einsatz virtualisierter lokaler SDN-Controller für lokale Kontrolllogik bietet somit, wie diese Arbeit zeigt, verschiedene Vorteile. Die Reduzierung der Last des logisch zentralisierten SDN-Controllers, das Einsparen der Latenz zu diesem bei lokalen Entscheidungen, die gute Isolation der dafür nötigen Anpassungen und die erhöhte Ausfallsicherheit sind dabei vermutlich die entschiedensten Punkte.

Dennoch sind für die Zukunft einige Fragen offen. Aufgrund der noch relativ einfachen Anwendungsszenarien für diese Arbeit wurde zum Beispiel noch nicht untersucht wie groß die Vorteile bei komplexeren Anwendungsfällen oder nur teilweise lokal verarbeiteten Anwendungsfällen sind. Auch sind weitere Untersuchungen in größeren Netzwerken als einem kleinen Testbed Setup denkbar, um noch bessere Aussagen über den Nutzen in der Praxis machen zu können. Des Weiteren wäre auch ein Vergleich mit Alternativen für lokale Kontrolllogik interessant, um zu sehen welche Ansätze die meisten Vorteile bieten.

Literatur

[1] AT&T. *AT&T SDN Network Design Challenge*. 2016. URL: http://about.att.com/innovation/labs/SDNChallenge.

[2] Giuseppe Bianchi et al. „OpenState: programming platform-independent stateful openflow applications inside the switch“. In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), S. 44–51.

[3] Roberto Bifulco et al. „Improving SDN with InSPired Switches“. In: *Proceedings of the Symposium on SDN Research*. SOSR '16. Santa Clara, CA, USA: ACM, 2016, 11:1–11:12. ISBN: 978-1-4503-4211-7. DOI: 10.1145/2890955.2890962. URL: http://doi.acm.org/10.1145/2890955.2890962.

[4] Dr. James Bottomley. *What is All the Container Hype?* 2014. URL: http://www.odin.com/fileadmin/media/hcap/pcs/documents/ParCloudStorage_Mini_WP_EN_042014.pdf.

[5] Carmelo Cascone et al. *OpenState SDN*. 2016. URL: http://openstate-sdn.org/.

[6] Citrix. *SDN 101: Eine Einführung in softwaredefiniertes Networking*. 2012. URL: https://www.citrix.com/content/dam/citrix/en_us/documents/oth/sdn-101-an-introduction-to-software-defined-networking-de.pdf.

[7] Cloudius. *OSv*. 2016. URL: http://osv.io/.

[8] Ryu SDN Framework Community. *Ryu*. 2016. URL: https://osrg.github.io/ryu/.

[9] Broadcom Corporation. *OpenFlow Data Plane Abstraction (OF-DPA)*. 2014. URL: https://www.broadcom.com/docs/support/OF-DPA-Specs_v2.pdf.

[10] Edgecore Networks Corporation. *AS5712-54X*. 2016. URL: http://www.edge-core.com/productsInfo.php?cls=1&cls2=8&cls3=44&id=15.

[11] CPqD. *libfluid*. 2016. URL: http://opennetworkingfoundation.github.io/libfluid/.

[12] Matthias Fetzer. „Local Data Plane Event Handling in Software-defined Networking“. Magisterarb. University of Stuttgart, 2016.

[13] Project Floodlight. *Floodlight*. 2016. URL: http://www.projectfloodlight.org/floodlight/.

[14] Flowgrammable. *Flowgrammable*. 2016. URL: http://flowgrammable.org/.

[15] Open Networking Foundation. *Open Networking Foundation*. 2016. URL: https://www.opennetworking.org.

[16] Open Networking Foundation. *OpenFlow Switch Specification*. 2015. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[17] Open Networking Foundation. *SDN architecture*. 2014. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf.

[18] Open Networking Fundation. „Software-defined Networking: The new norm for networks“. In: *ONF White Paper* (2012).

[19] BISDN GmbH. *libfluid*. 2016. URL: https://bisdn.github.io/rofl-core/rofl-common/index.html.

[20] python greenlet. *greenlet*. 2016. URL: https://github.com/python-greenlet/greenlet.

[21] Roberto Ierusalimschy, Waldemar Celes und Luiz Henrique de Figueiredo. *Lua*. 2016. URL: http://www.lua.org/about.html.

[22] Docker Inc. *Docker*. 2016. URL: https://www.docker.com/.

[23] Pica8 Inc. *PicOS*. 2016. URL: http://www.pica8.com/products/picos.

[24] InterfaceMasters. *OF-DPA Simple Switch*. 2014. URL: https://github.com/InterfaceMasters/ryu/blob/imt_ofdpa_simple_switch_13/ryu/app/simple_switch_13.py.

[25] Sushant Jain et al. „B4: Experience with a globally-deployed software defined WAN“. In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), S. 3–14.

[26] Hiroaki KAWAI. *twink*. 2016. URL: https://github.com/hkwi/twink/.

[27] Rahamatullah Khondoker et al. „Feature-based comparison and selection of Software Defined Networking (SDN) SDN-Controllers“. In: *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*. IEEE. 2014, S. 1–7.

[28] Diego Kreutz et al. „Software-defined Networking: A comprehensive survey“. In: *Proceedings of the IEEE* 103.1 (2015), S. 14–76.

[29] Open Network Linux. *Open Network Linux*. 2016. URL: https://opennetlinux.org/.

[30] linuxcontainers.org. *LXC*. 2016. URL: https://linuxcontainers.org/.

[31] loxigen. *loxigen*. 2016. URL: https://github.com/floodlight/loxigen.

[32] Anil Madhavapeddy und David J Scott. „Unikernels: the rise of the virtual library operating system“. In: *Communications of the ACM* 57.1 (2014), S. 61–69.

[33] C. Meinel et al. *Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht*. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Univ.-Verlag, 2011. ISBN: 9783869561134.

[34] Hesham Mekky et al. „Application-aware Data Plane Processing in SDN“. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, S. 13–18. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620735. URL: http://doi.acm.org/10.1145/2620728.2620735.

[35] Mirage. *MirageOS*. 2016. URL: https://mirage.io/.

[36] Nicira. *Nicira Extensions*. 2016. URL: https://github.com/openvswitch/ovs/blob/master/include/openflow/nicira-ext.h.

[37] opendaylight.org. *OpenDaylight*. 2016. URL: https://www.opendaylight.org/platform-overview/.

[38] Hewlett Packard. *OpenSwitch*. 2016. URL: http://www.openswitch.net/.

[39] Ben Pfaff und Bruce Davie. *RFC 7047 The Open vSwitch Database Management Protocol*. 2013. URL: http://www.ietf.org/rfc/rfc7047.txthttps://tools.ietf.org/html/rfc7047.

[40] Ben Pfaff et al. „The design and implementation of open vswitch“. In: *12th USENIX symposium on networked systems design and implementation (NSDI 15)*. 2015, S. 117–130.

[41] UniK Project. *UniK*. 2016. URL: https://github.com/emc-advanced-dev/unik.

[42] NOX Repo. *NOX*. 2016. URL: https://github.com/noxrepo/nox.

[43] Rumprun. *Rumprun*. 2016. URL: https://github.com/rumpkernel/rumprun.

[44] Alexander Shalimov et al. „Advanced study of SDN/OpenFlow SDN-Controllers“. In: *Proceedings of the 9th central & eastern european software engineering conference in russia*. ACM. 2013, S. 1.

[45] Mininet Team. *Mininet*. 2016. URL: http://mininet.org/.

[46] QEMU Team. *QEMU*. 2016. URL: http://wiki.qemu.org/Main_Page.

[47] Trema. *Trema*. 2016. URL: https://trema.github.io/trema/.

[48] unikernel.org. *Unikernels*. 2016. URL: http://unikernel.org/.

[49] Open vSwitch. *Open vSwitch*. URL: http://openvswitch.org/.

Erklärung

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

Ort, Datum, Unterschrift