Erweiterung des OpenDaylight SDN-Controllers zur Kommunikation mit VirtualStack

Bachelor-Thesis von Sarah Lettmann aus Darmstadt
Tag der Einreichung:

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Jens Heuschkel, M.Sc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Telecooperation Group
Technische Universität Darmstadt

Erweiterung des OpenDaylight SDN-Controllers zur Kommunikation mit VirtualStack

Vorgelegte Bachelor-Thesis von Sarah Lettmann aus Darmstadt

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Jens Heuschkel, M.Sc.

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 12. Juli 2016

(S. Lettmann)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

SDN Software-defined networking . . . 2
SDNi Software-defined networking interface . . . 12
IANA Internet Assigned Numbers Authority . . . 20
VS VirtualStack . . . 15
QoS Quality of Service . . . 16
VNIC Virtual Network Interface . . . 16
NAT Network Address Translation . . . 17
ToMaTo Topology Management Tool . . . 28
SAL Service Abstraction Layer . . . 2
YANG Yet Another Next Generation . . . 20
NETCONF Network Configuration Protocol . . . 20
RPC Remote Procedure Call . . . 13
REST Representation State Transfer . . . 12
OSGi Open Services Gateway initiative . . . 12
UUID Universally Unique Identifier . . . 19
VM Virtuelle Maschinen . . . 28

Zusammenfassung

Software-defined networking (SDN) ist die Netzwerktechnologie der Zukunft. In Zeiten von immer größer werdenden Netzwerken, die einem immer größeren Datenverkehr ausgesetzt sind, wird eine Technologie wie SDN dringend benötigt, um solche Netzwerke dynamisch aufzubauen und verwalten zu können. Resourcen, die über unzählige Knoten im Netzwerk verteilt sind, können effektiver genutzt und auf ausfallende Knoten kann schnell reagiert werden. Nicht zuletzt steigt die Nachfrage nach großen skalierbaren Netzwerken, die performant und zuverlässig arbeiten, stetig. Das Auflösen von statischen Netzwerkstrukturen durch SDN wirkt sich allerdings nicht auf Anwendungen aus, die das Netzwerk nutzen. Diese verwenden oft spezielle Protokolle, die beispielsweise eine bessere Performanz sichern sollen. Solche Protokolle werden allerdings domainübergreifend in Standardprotokolle umgewandelt. Moderne Netzwerke sollten eine adaptive Lösung für dieses Problem bieten.

In dieser Arbeit wurde gezeigt, dass die adaptive Nutzung von Protokollen auf Endknoten durch einen SDN-Controller gesteuert werden kann. Der SDN-Controller kann sich später beim Setzen von Protokollen an der Auslastung und dem Zustand des Netzwerkes orientieren. Wir verwenden dazu das VirtualStack Framework für Endknoten in Kommunikation mit dem OpenDaylight SDN-Controller. Anstatt spezifische Protokolle durch Standardprotokolle zu ersetzen, können die Endknoten durch VirtualStack adaptiv ein bestimmtes Protokoll verwenden. Anwendungen können so weiterhin spezielle Protokolle nutzen. OpenDaylight wird von den Endknoten über eingehende Anwendungsflows informiert und kann seinerseits die Verwendung eines bestimmten Protokolls für einen Anwendungsflow fordern. Die Evaluation hat gezeigt, dass alle implementierten Befehle beiderseitig korrekt verarbeitet werden.

Abstract

Software-defined networking (SDN) is the network technology of the future. Such a technology is required by the growing amount of networks which have to be robust for the growing amount of traffic. SDN enables networks to be build up and manged dynamically. Resources which are widely spread across the network can be used more efficiently and the network can react dynamically to the loss of nodes. Also the demand for scalable, performant and reliable working networks is increasing constantly. Unfortunately, breaking up the static network structure by SDN does not infect the applications using the network. These applications often use specific protocols, which are converted to standard protocols across different domains. Modern networks should offer an adaptive solution for this problems.
In this work we showed that an SDN controller can control the adaptive use of protocols for endnodes. At a later time, the SDN controller can consider the network throughput as well as the state of the network when setting protocols. We make use of VirtualStack for endnodes and the OpenDaylight SDN controller. Instead of converting specific protocols to standard protocols the VirtualStack framework enables the endnodes to make use of different protocols adaptively. Therefore, applications can still use specific protocols. While the endnodes inform the controller about new application flows, OpenDaylight can enforce the usage of certain protocols for an application flow. The evaluation showed that each of the implemented instructions are processed correctly by the controller and VirtualStack.

1 Einleitung

Netzwerke müssen große Datenmengen sicher und effizient übermitteln können. Wichtige Aspekte hinsichtlich dieser Anforderung sind besonders ein zuverlässiges und effizientes Arbeiten des Netzwerks sowie die Skalierbarkeit, der zugrunde liegenden Netzwerktopologie. Zudem müssen in der Zeit von großen Netzwerkstrukturen und sich schnell ändernder Hardware Netzwerktopologien simpel erweitert werden können. Außerdem sollen sie in der Lage sein, sich dynamische an Änderungen im Netzwerk anzupassen.
Um moderne Netzwerktopologien zu schaffen, die diesen Anforderungen gerecht werden und möglichst dynamisch agieren und anpassbar sind, wurde SDN entwickelt. Hauptaugenmerk liegt dabei auf der Separierung von Datenebene (engl. *data plane*) und Steuerebene (engl. *control plane*) und der damit verbundenen Ernennung eines zentralen oder verteilten Controllers, über den eine einfache Steuerung der Netzwerkgeräte möglich ist. Diese Netzwerkgeräte (Router, Switches) sind indes nicht mehr als einfache Weiterleitungsgeräte (engl. *forwarding devices*) - die Logik ist auf dem Controller hinterlegt. Dadurch sind SDN Topologien skalierbar, da besonders bei einem verteilten Controller Ressourcen effizient genutzt werden können. Zudem werden selbst komplexe Netzwerkstrukturen durch die verschiedenen Abstraktionsschichten handhabbar und einfach steuerbar, was zu einer weniger fehleranfälligen Struktur führt. Geräte im Netzwerk und deren Zustand werden transparent dargestellt und können nun über eine zentrale Steuereinheit konfiguriert werden. Selbst bei Ausfall von Steuerelementen kann im besonderen ein verteilter Controller weiterhin zuverlässig arbeiten.
In dieser Bachelorarbeit wurde ein Protokoll definiert, welches die Kommunikation zwischen einem SDN-Controller und VirtualStack ermöglicht. Dafür wurde der verteilten Open Source SDN-Controller OpenDaylight ausgewählt. Das OpenDaylight Projekt ist eins der meist genutzten und finanziell am meisten geförderten SDN Projekte zur Zeit. Gründe dafür sind die Nutzung des inoffiziellen Standards OpenFlow als Southbound API und die Implementierung des Controllers in Java. VirtualStack ist ein Framework, das auf Endknoten ausgeführt wird. Es erstellt virtuelle Protokollstacks unter der Verwendung von Multipathing. Für jeden Flow werden Stacks gebaut und dann individuell für jedes Paket des Flows entscheidet, welcher Protokollstack zur Übermittlung verwendet werden soll. Dazu ist auch das Decision Module im VirtualStack an das definierte Protokoll angepasst worden. Nach der Implementierung wurden die Ergebnisse über das ToMaTo Testbed evaluiert. Diese Bachelorarbeit ist als Proof of Concept angelegt und soll zeigen, dass eine Kommunikation des VirtualStack Frameworks mit einem SDN-Controller möglich ist und der Controller somit die Möglichkeit hat zu entschieden, welche Protokollstacks gebaut werden. Des Weiteren soll gezeigt werden, dass mittels des Protokolls VirtualStack Informationen über eingehende Flows und verwendete Links an den Controller gesendet werden können. Dadurch erhielten die Endknoten im Netzwerk unabhängig vom Betriebssystem die Möglichkeit die verwendeten Protokolle dynamisch an einzelne Pakete anzupassen. Außerdem kann der SDN-Controller durch sein Wissen über den aktuellen Zustand des Netzwerkes und mithilfe der Informationen, welche er vom VirtualStack erhält, gezielt die Verwendung bestimmter Protokolle für Flows erzwingen und so beispielsweise auf Staus und ausfallende Links reagieren.
Die restliche Arbeit ist wie folgt aufgebaut: Im Kapitel 2 werden zunächst Zusammenhänge aus schon existierenden Arbeiten dargelegt. Außerdem wird dargestellt wie diese Erkenntnisse in Relation zu dieser Arbeit stehen. In den darauf folgenden Abschnitten wird ein Überblick über die Thematik des SDN sowie den OpenDaylight SDN-Controller und VirtualStack gegeben. In Kapitel 3 wird erläutert wie SDN funktioniert und die Grundideen dargelegt. Dazu gehen wir in 3.1 zuerst auf den Aufbau von SDN Topologien ein und erläutern in 3.2 Vorteile sowie Anwendungsfälle. In Kapitel 4 wird genauer auf die Architektur und Besonderheiten von OpenDaylight eingegangen (4.1). Zudem wird ein Überblick über die Entwicklungen auf dem Markt in 4.2 gegeben. Der für die Entwicklung von Plugins besonders wichtige Service Abstraction Layer wird in 4.3 beschrieben. Die Funktionsweise von OpenFlow wird in Abschnitt 5 beleuchtet. VirtualStack wird in Kapitel 6 betrachtet. Dazu werden seine Architektur (Abschnitt 6.2) sowie die Motivation hinter der Entwicklung dieses Frameworks erklärt (Abschnitt 6.1). In Kapitel 7 wird auf Details der Implementierung eingegangen, welche dann evaluiert und im Kapitel 8 erläutert werden.

2 Related Work

Im Folgenden wurden drei Arbeiten ausgewählt, die Protokolle zur Kommunikation zwischen SDN-Controllern und Weiterleitungsgeräten (engl. *forwarding devices*) entwickelt haben. Mit der Entwicklung von Software-defined networking (SDN) wurden solche Protokolle notwendig. Diese sollten es der Kontrollinstanz erlauben forwarding devices zu steuern. Heute sind viele unterschiedliche Ansätze in aktuellen SDN-Controllern verfügbar. Am weitesten verbreitet und am bekanntesten ist OpenFlow [24]. OpenFlow Switches wurden von McKeown et al. [24] an der Stanford University zuerst als Möglichkeit entwickelt, um mit neuen Netzwerkprotokollen zu experimentieren. Dazu sollten Switches jeglicher Art universell ansprechbar sein, ohne dass man eine Vorstellung von der internen Arbeitsweise haben musste. Dafür stellt OpenFlow ein Protokoll bereit, dass Funktionen nutzt, die von vielen Switches und Routern genutzt werden, um die Flusstabellen (engl. *flow tables*) der Switches zu manipulieren [24]. Heute wird OpenFlow in SDN eingesetzt, wodurch der Controller den Switches mitteilen kann, wie Datenströme (engl. *flows*) verarbeitet werden sollen.
Einen weiteren Ansatz präsentiert Song in [27]. Dieser zielt auf die weitere Entkopplung der forwarding devices von spezifischen Protokollen ab. Dazu wird im Rahmen des Protocol-Oblivious Forwarding (POF) ein generisches flow instruction set (FIS) vorgestellt, welches eine sauberer Trennung des data plane vom control plane ermöglicht als OpenFlow es bietet. Das ist möglich, da es plattformunabhängig ist und jedem forwarding device ohne Einfluss des Controllers zur Verfügung steht. OpenFlow deckt zwar mit jeder neuen Version mehr Protokolle ab, jedoch wird diese Kenntnis auch von den forwarding devices benötigt. Zudem sind diese nicht in der Lage unabhängig vom Controller Stati von Flows zu erfassen und deren Verhalten zu beeinflussen [27]. POF hingegen spricht die Geräte des data plane davon frei, spezifische Kenntnis über die verwendeten Protokolle zu haben.
Der nächste Ansatz, den wir hier erwähnen wollen, mehr Flexibilität zu ermöglichen, was die Nutzung verschiedener Protokolle angeht, kommt von Zhou et al. Diese entwickelten im Jahr 2013 PindSwitch. Diese Plattform, die auf den Switches läuft, ermöglicht das Entwerfen eigener Protokollformate und ebenso das Festlegen spezifischer Regeln zur Verarbeitung eingehender Flüsse (engl. *flows*) [28]. Diese werden analysiert und passenden Protokollen zugeordnet (die vom Nutzer definiert werden können). Dadurch kennt der Switch den Aufbau und kann Daten auslesen und diese dann weiter verarbeiten. In jeder Phase dieses Prozesses kann der Nutzer von außen durch Nachrichten Einfluss auf die Verarbeitung nehmen. Auch in dieser Arbeit orientiert man sich an OpenFlow [28].

3 Software-defined networking (SDN)

In diesem Abschnitt wird die Funktionsweise eines Netzwerkes unter der Verwendung von Software-defined networking genauer erläutert. Die Architektur wird beschrieben sowie Funktionen einzelner Interfaces in Kapitel 3.1 dargestellt. Des Weiteren wird in 3.2 kurz auf einige Merkmale von SDN-Controllern eingegangen und Vorteile sowie Anwendungsfälle aufgezeigt.

Software-defined networking (SDN) entwickelte sich aus der Grundidee von aktiven Netzwerken, in denen es jedem Knoten möglich sein sollte mit den Informationen aus den Inhalten der Pakete zu arbeiten und kam zum ersten Mal an der Stanford University auf. Damals beschäftigte man sich noch mit der Arbeit an OpenFlow [22]. Die Hauptidee von SDN besteht in dem Aufbrechen der dezentralisierten Strukturen von Netzwerken. Diese Strukturen sind nÃd'mlich schwer zu managen und zu kontrollieren. Jedes neue Gerät im Netzwerk muss beispielsweise separat konfiguriert werden, wozu nicht selten spezifische Kenntnisse über die Hardware und die proprietären Befehlssätze erforderlich sind. Middleboxes müssen ebenfalls mit bedacht platziert werden und machen die Topology somit noch statischer [22]. Diese Netzwerkgeräte verfügen über keinerlei Routingfunktionalität sondern übernehmen andere Aufgaben im Netzwerk wie die statistische Erfassung des Datenverkehrs, Lastenverteilung (engl. *load balancing*) oder dienen als Firewall [15]. Durch Verwendung von SDN wird hingegen ein konsistenter Blick auf den Zustand des Netzwerkes bereitgestellt sowie Schnittstellen, die eine simple Fernsteuerung der forwarding devices ermöglichen. Außerdem kann viel leichter auf ausfallende Links reagiert werden. Sogar bei einem Ausfall einer Steuereinheit bleibt ein verteilter SDN-Controller weiterhin funktionsfähig und kann auch weiterhin alle Knoten im Netzwerk kontrollieren.

3.1 Architektur eines Netzwerkes unter Verwendung von SDN

Der Kernpunkt von SDN ist die Auftrennung von der steuernden Logik und den einfachen forwarding devices wie Routern und Switches, die sich um das Weiterleiten der ankommenden Pakete kümmern, in control plane und data plane. Die Logik (control plane) kann dann auf einem zentralen oder verteilten Controller hinterlegt werden, der auf einem Server läuft, nach außen eine transparente und globale Sicht auf das Netzwerk ermöglicht und so die Geräte im Netzwerk steuert. Router und Switches haben somit keine eigene Steuerungslogik mehr und sind nur noch für das Weiterleiten von Paketen zuständig. Gesteuert wird ihr Verhalten zentral vom Controller. Dieser verfügt über ein viel breiteres Wissen als ein einfacher forwarding device es haben kann und kann so besser auf Änderungen im Netzwerk reagieren. Bild 1 zeigt die allgemeine Struktur eines solchen Netzwerkes unter Verwendung von SDN. SDN-Controller verfügen über ein Nor-

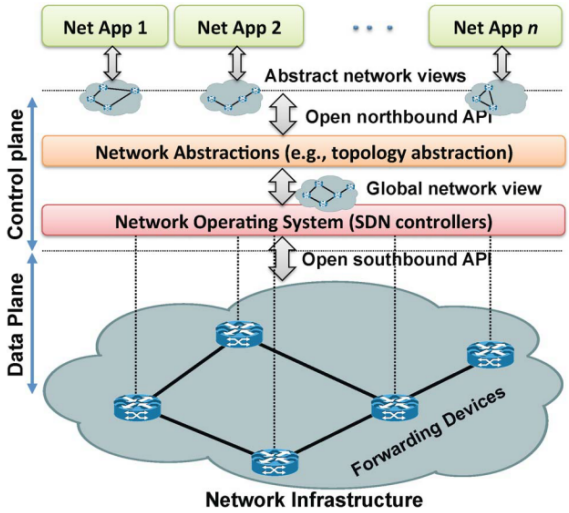

Abbildung 1: SDN Topology [22]

thbound Interface, welches eine Schnittstelle zu Netzwerkanwendungen darstellt. Hier gibt es noch keine Northbound API, die sich als inoffizieller Standard durchgesetzt hat. Oftmals implementieren SDN-Controller eigene Northbound Interfaces. Dabei ist ein allgemeiner Standard für solch ein Software Interface eine wichtige Entwicklung, um ein möglichst hohes Level an Portabilität und Zusammenarbeit zwischen verschiedenen Controllern zu erreichen. Anders sieht das beim Southbound Interface aus. Für uns ist dieses besonders von Bedeutung, da es eine Schnittstelle zwischen der Hardware und dem Controller darstellt - über sie kann gezielt mit Geräten im Netzwerk kommuniziert und ihr aktueller Zustand abgefragt werden. Als Southbound API hat sich OpenFlow etabliert. Viele Controller legen sich auf eine oder zwei Southbound APIs fest. OpenDaylight und einige wenige anderen allerdings bieten ein Interface für viele verschiedene APIs und

Plugins. So stellt OpenDaylight beispielsweise den Service Abstraction Layer (SAL) bereit, der die gleichzeitige Nutzung verschiedener Plugins erlaubt - mehr dazu im Kapitel 4.3. Im Kapitel 5 wird außerdem im Detail auf OpenFlow eingegangen. Zur Koordinierung eines verteilten Controllers ist außerdem ein Eastbound/Westbound Interface notwendig. Da verteilte Controller in der Regel aus mehreren Controllern bestehen, benötigen sie eine Schnittstelle, um Daten austauschen und kommunizieren zu können. Jeder Controller in einer SDN Topology kann über andere Ressource und andere Hardware verfügen, die den anderen Controllern zugänglich gemacht werden müssen. Zudem sind diese Interfaces sehr wichtig für Monitoringzwecke und um die Datenkonsistenz zu wahren - es sind spezielle Mechanismen nötig, um die korrekte Datenverteilung sicherzustellen.

3.2 Vorteile einer SDN Topology und Usecases

SDN wird heute schon in Heimnetzwerken, vor Allem aber in Datenzentren und Internet Exchange Points eingesetzt. SDN Topologien bringen viele Vorteile mit sich. Zum Einen ist eine sehr einfache high-level Programmierung des Netzwerkes möglich und Anwendungen, die in einem Teil des Netzwerkes laufen, können Ressourcen in einem weit entfernten Teil des Netzwerkes nutzen. Diese high-level Programmierung ist auch weniger fehleranfällig als maschinennahe Konfigurationen [22]. Da der Controller einen konsistenten und transparenten Blick auf das Netzwerk bereitstellt, kann auf jede Änderung schnell reagiert werden. Bei Ausfall eines Endknoten wird das Wegfallen dieser Verbindungen sofort erkannt und die Pakete über andere Pfade geleitet. Sogar bei Ausfall eines Controllers werden Endknoten, die vorher durch diesen gesteuert wurden, an andere Controller weitergegeben. Dies bringt eine höhere Robustheit und Zuverlässigkeit mit sich, besonders bei verteilten SDN-Controllern. Alle Geräte im Netzwerk werden zentral gemanagt und verwaltet und können leicht gesteuert und ebenso leicht überwacht werden. Gerade die Konfiguration von Geräten kann nun über diese Steuereinheit einheitlich und zentral stattfinden.
Skalierbarkeit ist ein weiteres Schlüsselwort. Große Firmen mit hunderten oder tausenden von Servern benötigen große skalierbare Netzwerke mit hoher Performanz und preisgünstigen Verbindungen zwischen allen Servern [11]. Klassische Netzwerke werden diesen Anforderungen nicht gerecht. SDN hingegen ist hoch-skalierbar und bietet zudem eine zentrale Verwaltungsmöglichkeit.
Des Weiteren deckt SDN durch seine Eigenschaften viele Use Cases ab, die beispielsweise für Forschungseinrichtungen oder Universitäten von Bedeutung sind. Auch diese Einrichtungen benötigen Netzwerke mit hoher Performanz und Skalierbarkeit. Aber auch für lokale Regierungen kann SDN interessant sein. Es kann in den Bereichen Smart Cities eingesetzt werden, um durch Echtzeitdatenerhebung das Leben in einer Stadt nachhaltiger und angenehmer zu gestalten [8].

4 Der OpenDaylight-Controller

In diesem Kapitel wird ausführlich auf den OpenDaylight SDN-Controller eingegangen. Ziel ist es, die Architektur (Kapitel 4.1) sowie Besonderheiten darzustellen. Des Weiteren soll deutlich gemacht werden, warum wir uns für diesen SDN-Controller entschieden haben und nicht für einen anderen. In Kapitel 4.2 wird eine Übersicht über erwähnenswerte Projekte im Bereich SDN-Controller gegeben. Von besonderer Bedeutung für uns ist der Service Abstraction Layer, auf den in Abschnitt 4.3 genauer eingegangen wird.

Der OpenDaylight Controller ist aktuell der meist genutzte SDN-Controller auf dem Markt. Zudem wartet er mit großen Sponsoren der Branche wie Cisco, IBM und Oracle auf [21]. Durch die Bereitstellung des Projekts als Open Source Projekt ist es jedermann möglich, sich an der Entwicklung zu beteiligen. OpenDaylight lebt also von den Ideen und der Initiative einer Community. Als verteilter Controller bietet er im Gegensatz zu zentralisierten Controllern eine bessere Skalierbarkeit sowie eine höhere Robustheit durch die physikalische Verteilung der Steuereinheiten. Zentralisierte Controller haben zudem nur einen Ausfallpunkt (engl *single point of failure*), was zu einer wesentlich geringeren Zuverlässigkeit und Robustheit führt. Ausfälle an einzelnen Punkten im Netzwerk können durch einen verteilten Controller schnell durch Neuzuteilung der Knoten an andere Server ausgeglichen werden. Der OpenDaylight Controller, aktuell im vierten Release Beryllium, ist in Java implementiert, wodurch er auf jeder Plattform, die Java unterstützt, lauffähig ist. Standardmäßig ist OpenFlow 1.1 vorinstalliert, unterstützt wird aber auch Version 1.3.

4.1 Die Architektur von OpenDaylight

Wie bereits im Kapitel 3.1 über SDN angeschnitten, verfügen SDN-Controller in der Regel über eine ähnliche Struktur. So hat auch OpenDaylight ein Northbound Interface, ein Southbound Interface und ein Eastbound/Westbound Interface. OpenDaylight stellt zwei Northbound APIs zur Verfügung: Representation State Transfer (REST) und das Open Services

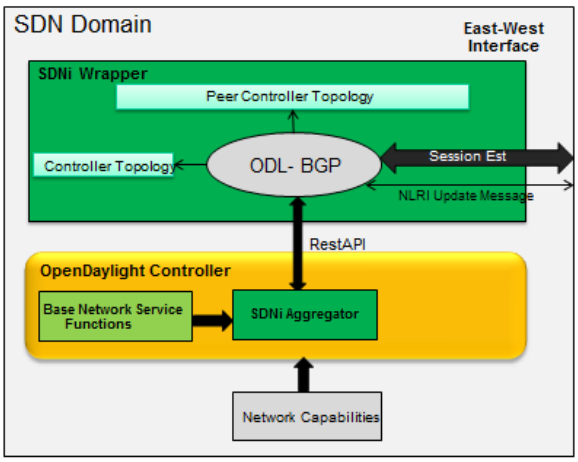

Abbildung 2: OpenDaylight SDNi [9]

Gateway initiative (OSGi) Framework. Während REST von Anwendungen genutzt wird, die einen anderen Adressraum nutzen als der Controller selbst, ist das OSGi Framework für Applikationen gedacht, die im gleichen Adressraum laufen. Somit deckt REST auch Webanwendungen ab [5].
Da es sich bei OpenDaylight um einen verteilten Controller handelt, wird ein Mechanismus zum synchronisieren der Daten zwischen den einzelnen Controllern benötigt. Dazu wird das Eastbound/Westbound Interface genutzt. Das OpenDaylight Projekt verfolgt auch hier einen generischen Ansatz. Das Software-defined networking interface (SDNi) ist als Applikation realisiert und läuft über das Northbound Interface. Als Nourthbound API nutzt diese Anwendung die REST API. Es wird ein sogenannter SDNi Aggregator als Plugin für das Northbound Interface bereit gestellt, der alle nötigen Informationen, die der lokale Controller über das Netzwerk hat, sammelt. Diese gehen dann über die REST API an einen SDNi Wrapper, der sich um den Austausch dieser Daten mit anderen Controllern kümmert [9]. Dazu wird BGP-4 genutzt [26]. Abbildung 2 verdeutlicht die Funktionsweise. Wichtige Daten, die einzelne Controller untereinander austauschen sind beispielsweise bestehende Links, IPs der Hosts oder MAC Adressen von forwarding devices, aber auch Daten, die zu Analysezwecken erhoben werden. Dazu zählen die Anzahl von erhaltenen und übertragenen Paketen sowie die Packet Loss Rate [9].
Als Southbound Interface nutzt OpenDaylight wie bereits erwähnt den sogenannten Service Abstraction Layer (SAL). Dieser Layer unterstützt verschiedene Protokollplugins und Southbound APIs und kann dynamisch Plugins laden. Netzwerkanwendungen nutzen bestimmte Dienste des SDN-Controllers und der SAL ermöglicht es diese unabhängig von der

genutzten Southbound API oder des genutzen Plugins auszuführen. Bild 3 gibt einen Überblick über die Architektur des SDN-Controllers [7]. DLUX ist das Web-User-Interface und kann Eingaben über REST als Northbound API an den Control-

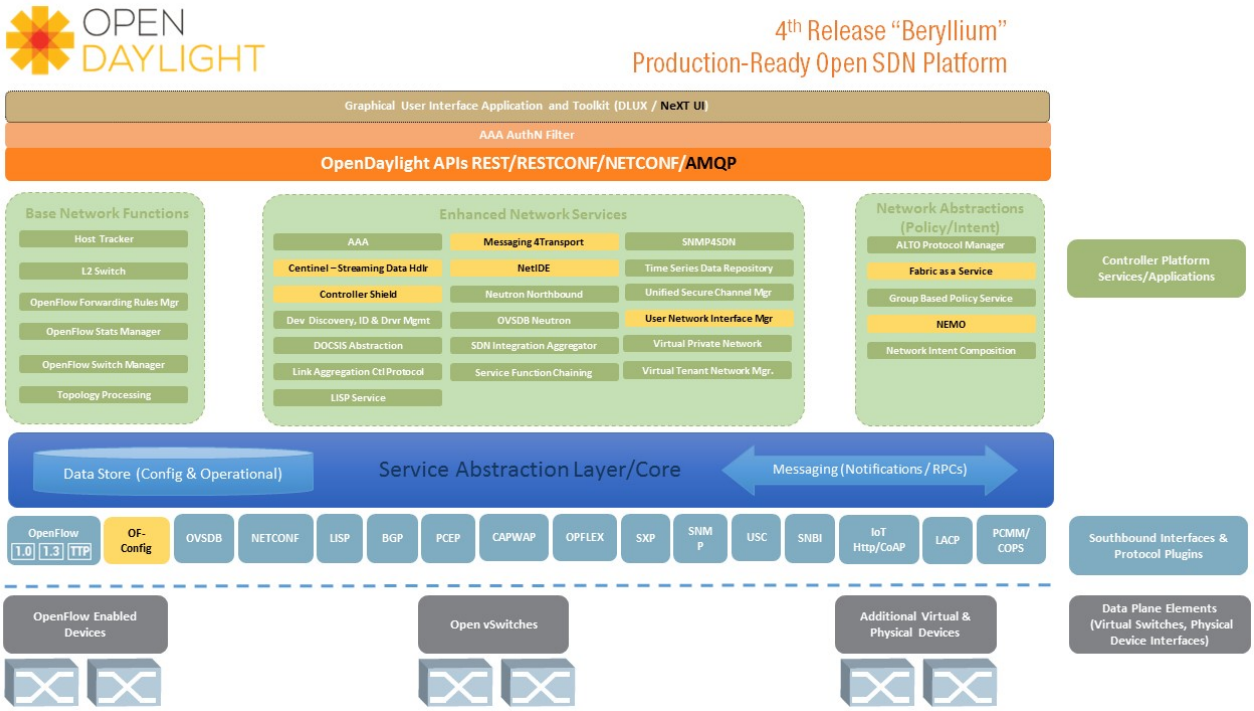

Abbildung 3: Die Architektur des OpenDaylight SDN-Controllers [7]

ler senden. Dieser Remote Procedure Call (RPC) wird dann über den SAL an die passende Southbound API weitergeleitet und wird von dieser wiederum an forwarding devices geschickt. Ebenso ist es möglich Nachrichten von Switches über ein bestimmtes Plugin zu empfangen und dann über den SAL eine Benachrichtigung an ein anderes Plugin zu senden. Dieses kann dann Aktionen auslösen, die Switches betreffen, die wiederum durch dieses Plugin angesprochen werden können.

4.2 OpenDaylight im Vergleich

Um einen kurzen Überblick über andere aktuelle Projekte im Bereich der SDN-Controller zu bekommen, haben wir vier Open Source Controller aus [22] rausgesucht und deren Eigenschaften verglichen. Zum Teil werden hier unterschiedliche Ansätze verfolgt, wie beispielsweise zentralisierte, verteilte oder hierarchische Controller. Es wurden aber auch Controller gewählt, die in der Entwicklungsgeschichte des SDN herausstechen. Tabelle 1 zeigt das Ergebnis.

Als die Entwicklung von SDN-Controllern und NOS anlief, war NOX einer der ersten Controller auf dem Markt. Damals schon nutzte er OpenFlow als Southbound API, hatte als Northbound API aber lediglich ein Programmatic Interface, welches lediglich einen Namespace und den Networkview bereit stellt sowie Eventhandler [18]. Floodlight hingegen ist ein zentralisierter OpenFlow Controller, der mit mehreren Threads arbeitet (engl. *multithreading*) [2]. Er basiert auf dem Beacon OpenFlow Controller entwickelt im Jahre 2010 by Erickson [17]. Hierbei werden *Listener* genutzt, welche in Java über Events informieren. Bei Beacon werden sie zur Erkennung von neuen Switches oder zum Empfangen von Nachrichten verwendet. Bei Floodlight wiederum ist der Umfang an gelieferten Software interessant. Diese liefern nämlich nicht nur den Controller, sondern auch zusätzliche Applikationen, die auf diesem laufen. Dazu zählen beispielsweise Firewall, Load Balancing und Learning Switch [3]. Kandoo, entwickelt im Jahre 2012 von Yeganeh et al., ist hingegen ein hierarchisch verteilter Controller. Das heißt, die Struktur ist wie ein Baum aufgebaut. Es gibt viele lokale Controller, die Switches kontrollieren. Diese lokalen Controller werden wiederum von einem Root Controller gesteuert. Ebenso ist es möglich lokale Anwendungen auf lokalen Controllern laufen zu lassen oder eben eine globale Anwendung auf dem Root Controller [19]. Ryu ist der letzte Controller, der im Vergleich zu OpenDaylight genauer betrachten wird. Auch er nutzt OpenFlow und ist als Framework angelegt. Der Schwerpunkt hierbei ist die Unterstützung und Vereinfachung von SDN Anwendungen, die zum Managen und Steuern des Netzwerkes dienen sollen [25] [1].
Im Vergleich zu diesen Controllern bringt OpenDaylight eindeutig die größte Flexibilität mit sich. Sein SAL erlaubt die Nutzung von aktuell 16 Soutbound APIs und Plugins und ist durch die Implementierung in Java plattformunabhängig.

Tabelle 1: SDN-Controller im Vergleich

Controller	Architektur	Programmier-sprache	Southbound APIs	Northbound APIs	East-/Westbound APIs
Floodlight	zentralisiert, multi-threaded	Java	OpenFlow	REST	-
Kandoo	hierarchisch, verteilt	C, C++, Python	-	-	-
NOX	zentralisiert	C++	OpenFlow	ad-hoc API	-
OpenDaylight	verteilt	Java	OpenFlow 1.0, OpenFlow 1.3, OVSDB, NETCONF, BGP, HTTP, ...	REST, OSGi	SDNi Wrapper
Ryu	zentralisiert, multi-threaded	-	OpenFlow, NETCONF	-	-

4.3 Der Service Abstraction Layer (SAL)

In diesem Abschnitt wird ein genauer Blick auf den Service Abstraction Layer (SAL) geworfen. Er stellt eine Abstraktionsschicht für Southbound APIs dar, kapselt also Funktionalität, die für Funktionen der APIs und Plugins benötigt wird [21]. Dazu gehört beispielsweise der DataBroker, welcher für das Schreiben und Lesen von Daten zuständig ist. In OpenDaylight können viele verschiedene Southbound APIs eingebunden werden. Standardmäßig ist dafür OpenFlow installiert. Zur Entwicklung eines Plugins muss also die Funktionsweise des SAL bekannt sein.

Der SAL bildet eine zentrale Schnittstelle zwischen den Protokollen, die zur Kommunikation mit der data plane verwendet werden, und den Diensten, die dem Netzwerk zur Verfügung gestellt werden. Dienste benötigen in der Regel Zugriff auf bestimmte Daten. Diese Daten werden von Southbound Plugins zur Verfügung gestellt. Im SAL findet dann ein mapping von den Plugins zu den Diensten statt. Des Weiteren werden Dienste auch nur zur Verfügung gestellt, wenn die erforderlichen Plugins installiert sind [5]. Wie Abbildung 4 zeigt, fordert ein Service aus den oberen Schichten des

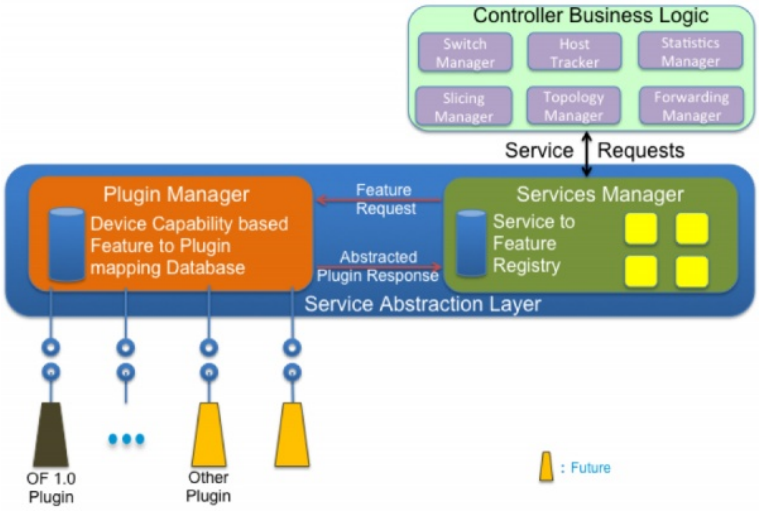

Abbildung 4: SAL des OpenDaylight SDN-Controllers [5]

Controllers Daten zu einem bestimmten Gerät an. Intern schickt der Service Manager eine Anfrage an den Plugin Manager, welcher dann überprüft welche Plugins für dieses Gerät zur Verfügung stehen und ob diese die angeforderten Daten liefern können. Ist dies der Fall, antwortet der Plugin Manager dem Service Manager, welcher wiederum den Dienst benachrichtigt, der die Anfrage gestellt hat [5].
Der Service Abstraction Layer (SAL) ermöglicht es viele unterschiedliche Protokolle zur Kommunikation und Steuerung von forwarding devices zu verwendet. Wie solche Plugins eingebunden und entwickelt werden, wird später in Kapitel 7.2 tiefergehend erklärt.

5 OpenFlow

In diesem Abschnitt wird auf den de facto Standard in Sachen Southbound API eingehen, der zu diesem Zeitpunkt schon einige Male erwähnt wurde. OpenFlow wurde zuerst in Stanford zu Forschungszwecken entwickelt, ist inzwischen aber in vielen SDN-Controllern die standard Southbound API. Die Gründe dafür sowie die Funktionsweise sollen in diesem Kapitel grundlegend dargestellt werden. Für diese Bachelorarbeit wurde ein eigenständiges Protokoll zur Kommunikation zwischen VirtualStack (VS) und OpenDaylight entwickelt, da OpenFlow die benötigten Informationen nicht abdeckt.

OpenFlow sollte eine universelle Schnittstelle zu Switches sein, unabhängig von der verwendeten Software. Um dies zu erreichen, wird die einfache Tatsache ausgenutzt, dass jeder moderne Switch oder Router *flow tables* nutzt, um beispielsweise Statistiken zu erstellen. Diese variieren zwar auch, nutzen aber meist die gleichen Funktionen, die zur Manipulation der flow tables genutzt werden können [24]. OpenFlow teilt sich in ein Protokoll auf, das zur Kommunikation zwischen SDN-Controller und Switches verwendet wird sowie OpenFlow Switches. Dabei kann es sich um handelsübliche Switches handeln, denen ein Interface zur Nutzung von OpenFlow hinzugefügt wurde oder aber dedizierten OpenFlow Switches, die im Gegensatz zu handelsüblichen Switches Daten von der Sicherungsschicht (engl. *data link layer*) und/oder der Vermittlungsschicht (engl. *network layer*) nicht verarbeiten können [24].
Im Folgenden wollen wir einen kurzen Blick auf das Protokoll von OpenFlow werfen. Abbildung 5 zeigt den Header, der in jeder OpenFlow Nachricht verwendet wird. Die Version des Protokolls gibt das *version* Feld an, *type* kann eine

```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version;    /* OFP_VERSION. */
    uint8_t type;       /* One of the OFPT_ constants. */
    uint16_t length;    /* Length including this ofp_header. */
    uint32_t xid;       /* Transaction id associated with this packet.
                           Replies use the same id as was in the request
                           to facilitate pairing. */
};
OFP_ASSERT(sizeof(struct ofp_header) == 8);
```

Abbildung 5: Header einer OpenFlow Nachricht [16]

von vielen verschiedenen vordefinierten Konstanten sein, die den Inhalt der Nachricht genauer spezifizieren, *length* gibt die absolute Länge der Nachricht an und *xid* ist ein eindeutiger Identifizierer für die jeweilige Nachricht [16]. Auch der Header unseres Protokolls enthält beispielsweise ein Feld für den Typ der Instruktion. Mehr dazu wird in Kapitel 7.1.2 beschrieben. Des Weiteren gibt es bestimmte Strukturen, welche häufig in Nachrichten verwendet werden. Dazu zählen Strukturen zur genauen Angabe eines Ports oder zur Angabe einer Übereinstimmung von Flows (engl. *match*).
Nun soll die Arbeitsweise eines OpenFlow Switches genauer beleuchtet werden. Ein dedizierter OpenFlow Switch leitet

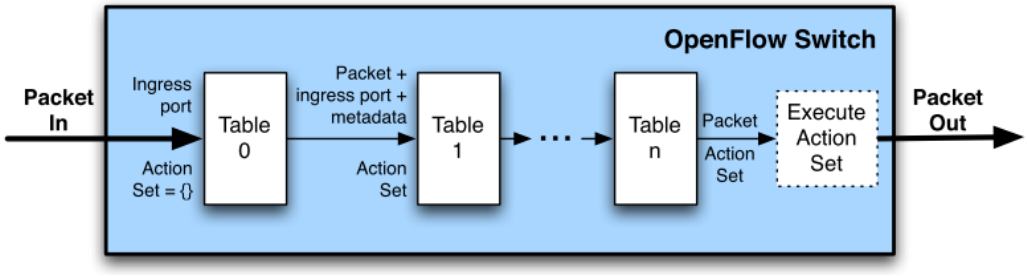

Abbildung 6: Verarbeitung eines eingehenden Flows in OpenFlow [16]

alle eingehenden Flows über die OpenFlow Leitung (engl *pipeline*), da keine andere Verarbeitung möglich ist. Diese pipeline besteht aus flow tables, die aufsteigend nach Kategorie nummeriert sind. Jeder flow table enthält Einträge (engl. *flow entries*), die mit einem eingehenden Flow verglichen werden. Zuerst soll also der Eintrag mit der höchsten Priorität für den eingehenden Flow gefunden werden. Ist das geschehen, werden die im flow entry gespeicherten Daten aktualisiert. Dazu zählen Metadaten, ein Zähler und Aktionen, die die weitere Verarbeitung in der Pipeline beeinflussen können [16]. In unserem Fall wird in der Erweiterung des VirtualStack, falls Aktionen vom Controller eingehen, nach zugehörigen *flowIds* gesucht. Danach werden die Aktionen für diesen Flow ausgeführt. In Abschnitt 7.3 wird genauer darauf eingegangen.

6 Das VirtualStack Framework

In diesem Abschnitt wird der Aufbau des VirtualStack (VS) Frameworks erklärt und die Funktionsweise dargestellt. Genauer wird auch erläutert, warum solch ein Framework notwendig ist und eine logische Erweiterung zu SDN darstellt. Zudem wird auch auf einzelne architektonische Merkmale eingegangen und erklärt wie ein Protokollstack [10] gebaut wird.
VirtualStack läuft auf den Enknoten im Netzwerk und baut dort virtualisierte Protokollstacks. Für jeden Anwendungsflow steht ein Stack von Protokollen vom Physischen- bis zum Transport-Layer bereit. Pro Paket kann nun ausgewählt werden, basierend auf den Informationen im Header und den Zuständen der Ausgangslinks, welches Protokoll verwendet werden soll. Dabei arbeiten verschiedene Module im Framework zusammen, analysieren die Metadaten des Pakets, werten diese aus und wählen dann ein Protokoll. Es wurde an der TU Darmstadt entwickelt.

6.1 Motivation

SDN soll die statischen Strukturen von Netzwerken auflösen und liefert ein zentrales Management, Transparenz und eine adaptive Struktur. Allerdings sind Anwendungen in ihren Anforderung an Netzwerkressourcen immer noch statisch. So nutzen Anwendungen beispielsweise unabhängig vom Zustand des Netzwerkes bestimmt Protokolle. Oft werden sogar sehr spezielle Protokolle für Anwendungen genutzt, um eine höhere Quality of Service (QoS) oder Performanz zu erreichen. Diese Protokolle werden aber domainübergreifend durch Standardprotokolle ersetzt, um Interoperabilität zu sichern [20]. Das resultiert in Performanzeinbußen. Dieser Problematik nimmt sich das VS Framework an. VS erkennt Flows einer Anwendung anhand des Eingangsports. Dann werden verschiedene Stacks gebaut für unterschiedliche Kombinationen aus Protokollen. Diese Protokolle stehen nun jedem Paket des jeweiligen Flows zur Verfügung und können je nach Eigenschaften des Netzwerkes genutzt werden. Die Pakete des Flows werden über den ausgewählten Stack weitergeleitet. So ist eine Transformation des verwendeten Protokolls möglich, die trotzdem einen maximalen Durchsatz erreicht und kaum Einfluss auf die Performanz hat [20].

6.2 Architektur von VirtualStack

Das VirtualStack Framework besteht aus drei Modulen, die verschiedene Aufgaben übernehmen, die nachfolgend näher erläutert werden. Zudem werden Flows über ein Virtual Network Interface (VNIC) empfangen, welches ein standard Socket nutzt. Die empfangenen Daten werden an das Analysis Module weitergereicht.
Das Analysis Module analysiert die eingehenden Flows. Es teilt sich in den Classifier und den ID Extractor auf. Der Classifier ermittelt anhand des IP-Headers welche Protokolle zur Übermittlung genutzt wurden (IPv4 oder IPv6 mit UDP oder TCP). Der ID Extractor ermittelt eine ID für den Flow anhand des Quellports. Wenn der Flow bekannte Protokolle nutzt, kann eine ID ermittelt werden, ansonsten wird die ID des Flows auf *null* gesetzt. Diese Information gehen dann an das Decision Module.
Das Decision Module erhält Informationen über die Flows vom Analysis Module. Es erkennt, ob der Flow bereits be-

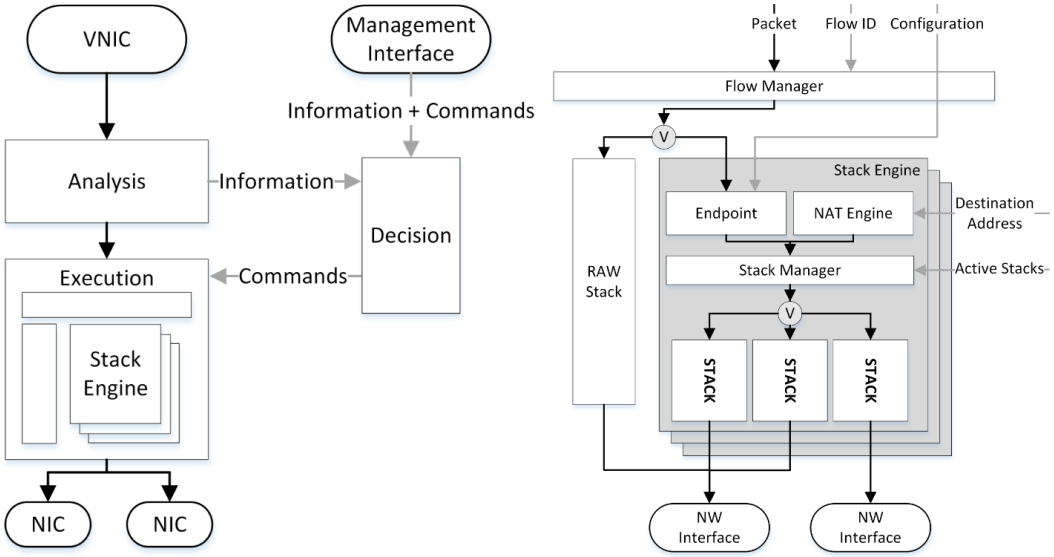

Abbildung 7: Das VirtualStack Framework (links) und das Execution Module im Detail (rechts)

kannt ist und entscheidet anhand der Informationen vom Analysis Module welches Protokoll genutzt werden soll. Das

Decision Module ist für uns zudem von besonderer Bedeutung, denn es besitzt ein Interface über welches es mit einem SDN-Controller kommunizieren kann. Somit können sowohl Informationen an den Controller weitergegeben als auch Befehle von diesem empfangen werden. Abbildung 8 zeigt die Nutzung des ManagementInterface.
Das Execution Module enthält den Flow Manager, den Stack Engine und die physischen Schnittstellen zum Netzwerk. Zuerst verwendet der Flow Manager die Informationen aus dem Decision Module und teilt so jedem Paket eines bekannten Flows anhand der ID die passende Stack Engine zu. Ist der Flow neu, dann werden im nächsten Schritt im Stack Engine die Protokollstacks [10] gebaut. Die Stacks bestehen aus drei Protokollen aus der Sicherungsschicht, Netzwerkschicht und der Transportschicht. Eine Mögliche Kombination ist beispielsweise Ethernet, IPv4, TCP. Über diese Stacks wird der Payload der eingehenden Pakete der Flows weitergesendet. Pakete von Flows mit ID *null* nutzen ein unbekanntes Protokoll und werden unverändert über den RAW Stack geleitet, wodurch das VS Framework mit jeder Anwendung verwendbar ist. Der Stack Engine enthält den Endpoint der Anwendung, der die Verbindung zur Anwendung behandelt, sowie eine Network Address Translation (NAT) Engine, der die Zieladresse des Flows bereithält und die Stacks selbst. Abbildung 7 zeigt den Aufbau des VirtualStack Frameworks.

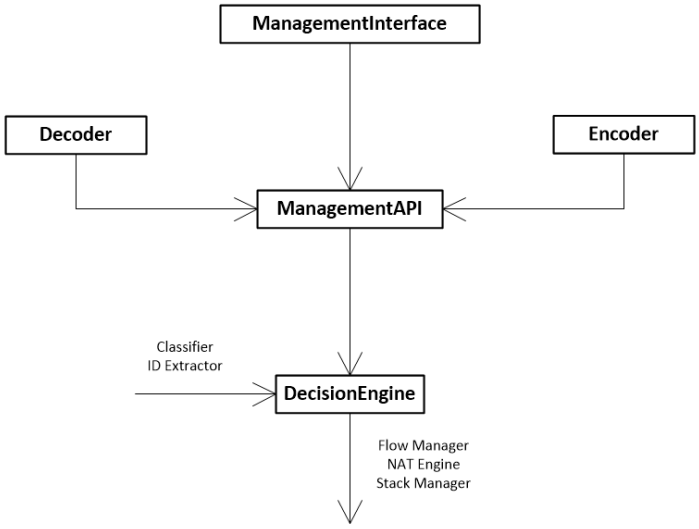

Abbildung 8: Verwendung des Management Interface

7 Implementierung

Dieser Abschnitt beschäftigt sich mit der Implementierung und den damit verbundenen Vorbereitungen. So wird beispielsweise das zur Kommunikation benötigte Protokoll definiert sowie die benötigten Befehle in erklärt. Außerdem werden Designentscheidungen dargelegt und der Code erläutert.

7.1 Vorbereitung

In diesem Abschnitt wird zum Einen eine Übersicht über die zu implementierenden Befehle in Kapitel 7.1.1 gegeben und diese erläutert. Zum Anderen wird zur Kommunikation zwischen dem SDN-Controller und VirtualStack auf den Endknoten ein Protokoll benötigt. Zwar bietet OpenFlow auch Befehle wie *ofp_port_status*, welcher über neue Ports informiert oder *ofp_flow_removed*, welcher über entfernte Flows informiert, aber diese decken nur teilweise benötigte Informationen ab. Beispielsweise bietet *ofp_flow_removed* nur eine *table_id* und keine *flow_id* und *ofp_port_status* enthält leider keine ID für den jeweiligen Port [16]. Das Protokoll wird im Abschnitt 7.1.2 definiert.

7.1.1 Der Enknotenbefehlssatz

Das Protokoll besteht aus insgesamt zehn Befehlen, die in drei Gruppen eingeteilt sind: Reporting, Monitoring und Actions. Während es sich bei der Gruppe Reporting um Benachrichtigungen des VS an den SDN-Controller handelt, sind Actions Befehle vom Controller an das Framework auf den Endknoten, welches gezielt Aktionen durchführen soll. Beim Monitoring handelt es sich um Nachrichten zum Austausch von Daten zur Performanz. Im Detail deckt das Protokoll folgende Befehle zur Kommunikation zwischen Controller und VS ab (die fett gedruckten Befehle werden implementiert).:

1. Reporting
   - **Registrierung eines Hosts (<hostId> [<linkId> <ip>] <stack>)**
     Dient der initialen Registrierung eines neuen Hosts bzw. Endknotens, der VS nutzt. Optional können zudem ein präferierter Link und die IP-Adresse des Knotens angeben werde. Der Controller speichert diese Daten zur weiteren Kommunikation mit dem Host. Der SDN-Controller spricht keine Hosts an, die sich nicht vorher bei ihm registriert haben. Die hostId identifiziert den Host eindeutig.
   - **Neuer Link (<hostId> <linkId> <ip>)**
     Für einen bereits bekannten Host kann ein neuer Link (eine physische Netzwerkschnittstelle, welche nun auch für Flows genutzt werden kann) angegeben werden. Außerdem wird die IP des Hosts, von dem der Report stammt, mit geschickt. Den Flows können später gezielt Ausgangslinks zugewiesen werden, die sie nutzen sollen. Die linkId identifiziert den jeweiligen Link eindeutig.
   - Link verloren (<hostId> <linkId> <ip>)
     Durch diesen Benachrichtigung teilt der VS dem Controller mit, dass ein Link nicht mehr genutzt werden kann, weil beispielsweise die Verbindung abgebrochen ist.
   - **Neuer Flow (<hostId> <linkId> <stack> <ip> <flowId>)**
     Für einen bereits bekannten Host kann ein neuer Flow angeben werden. Hierzu wird außerdem die Ziel-IP des Flows sowie die bereits erstellten Protokollstacks angegeben. Die flowId identifiziert den Flow eindeutig.
   - **Flow abgeschlossen (<hostId> <flowId> <status>)**
     Wenn der Flow einer Anwendung abgeschlossen ist, kann dies über diesen Befehl mitgeteilt werden. Die flowId wird dann im SDN-Controller als geschlossen markiert und es können keine Actions mehr für diesen Flow gesendet werden. Dazu werden die ID des Hosts sowie ein Status angegeben. Dieser kann auch eine Fehlermeldung enthalten.

   Monitoring
   - Leistungsdaten <linkId> [<flowId>] <key> <value>
     Der Controller sendet Leistungsdaten zu einem speziellen Link an VirtualStack.
   - Leistungsdaten <hostId> <linkId> [<flowId>] <key> <value>
     VS sendet Leistungsdaten zu einem bestimmten Link an den Controller. Diese Daten können auch an einen speziellen Flow gebunden sein.

2. Actions
   - **Setzen oder ändern eines Protokolls (<flowId> <stack> [<linkId>])**
     Ermöglicht das Setzen eines neuen Protokolls oder das Ändern eines bereits bestehenden Protokolls für einen Flow. Ist ein Protokollstack bereits vorhanden, wird dieser als aktiv gesetzt. Wahlweise kann das auch an einen spezifischen Link gebunden werden.

- Setzen der Verbindungssrate (engl. *line rate*) für einen Link oder Flow ([<linkId> <flowId>] <linerate>)
  Der Controller teilt VS die *line rate* mit. Diese kann auch jeweils an einen Link und des Weiteren auch an einen Flow gebunden werden.
- Anfragen der *line rate* für einen Link oder einen Flow (<hostId> [<linkId> <flowId>] <linerate>)
  VS fragt eine *line rate* an. Diese Anfrage kann auch für einen speziellen Link oder Flow getätigt werden.

7.1.2 Das Protokoll

Zur Kommunikation zwischen dem SDN-Controller und dem VirtualStack Framework auf den Endknoten wird ein Protokoll zum Austausch von Informationen benötigt. Dieses erlaubt das Senden von Reports des VS an den SDN-Controller und ebenso das Senden von Actions vom SDN-Controller an den VS. Tabelle 2 bietet eine Übersicht des Aufbaus. Die

Tabelle 2: Protokoll

20-40 Bytes	20 Byte	1 Byte	1 Byte	1 Byte	bis zu 297 Bytes
IP	TCP	Timestamp	Type	Checksum	Payload

ersten beiden Felder IP und TCP stellen die entsprechenden Header dar, die zum Versenden der Datenpakete benötigt werden. Danach folgt ein Byte für einen Timestamp, ein Byte für den Befehlstyp und ein Byte für eine Checksumme. Erst danach folgt der Payload, dessen Größe vom Befehlstyp abhängt. Die Verteilung der Bits für die jeweiligen Befehle zeigt Tabelle 3. Der Timestamp wird beim Versenden des Pakets generiert. Bei der Checksumme haben wir uns für eine

Tabelle 3: Bitverteilung der einzelnen Befehle ohne IP/TCP Header

Name	**Type**
Registrierung eines Hosts mit Link und IP	10 01 0001
Registrierung eines Hosts ohne Link und IP	01 01 0001
Neuer Link	01 01 0010
Neuer Flow	01 01 0011
Flow abgeschlossen	01 01 0100
Ändern oder setzen eines Protokolls mit Link	10 11 0001
Ändern oder setzen eines Protokolls ohne Link	01 11 0001

Tabelle 4: Befehlstypen

Version (2 Bit)	Group (2 Bit)	Command (4 Bits)

übliche CRC-Checksumme von 32 Bit entschieden, deren 4 Byte mit XOR verknüpft werden. Der Befehlstyp ist unterteilt wie in Tabelle 4 gezeigt. Die Befehle sind jeweils einer Befehlsgruppen zugeteilt, die über Group (Reporting, Monitoring, Actions) angegeben wird. Der eigentliche Befehl wird von Command angegeben und da die Befehle teilweise mit unterschiedlich viele Daten ausgeführt werden können, gibt das Feld Version an, welche Daten mitgegeben wurden. Im Feld Payload befinden sich alle notwendigen Daten, die entweder vom SDN-Controller gespeichert werden müssen oder die VirtualStack benötigt, um Aktionen ausführen zu können. Tabelle 5 gibt an wie viele Bytes zu welchem Datum gehören. Die Reihenfolge der Daten kann dabei pro Befehl variieren. *hostId*, *linkId* und *flowId* geben jeweils eindeutige Identifier

Tabelle 5: Datengrößen

hostId	16 Byte
linkId	4 Byte
flowId	4 Byte
Status	2 Byte
IP	8 Byte (IPv4), 16 Byte (IPv6)
Stack	1 Byte - 255 Byte

an, die den Host, einen Link dieses Hostes und einen Flow, dessen Datanpakete auf diesem Host empfangen wurde, eindeutig identifizieren. Als *hostId* wurde sich für den Universally Unique Identifier (UUID) entschieden, die sich aus verschiedenen Daten wie der MAC-Adresse der Netzwerkschnittstelle zusammen setzt. Die*linkId* und die *flowId* werden

jeweils vom VS bei Registrierung eines neuen Links bzw bei Eingang eines neuen Flows angelegt. Die Statusbytes geben bei Setzung unterschiedlicher Bits unterschiedliche Fehlercodes an. Wenn alle Bits auf null gesetzt sind, liegt kein Fehler vor. IP gibt die IP-Adresse des jeweiligen Hosts an. Hierbei wird das Feld dynamisch auf IPv4 oder IPv6 angepasst, um unnötigen Overhead zu vermeiden. Dazu gibt es ein zusätzliches Byte vor den IP-Bytes, das die Größe des Feldes angibt. Auch für das Feld Stack muss die Größe angegeben werden, da es sich hierbei um Stacks von Protokoll Identifiern handelt, die für das Protokollfeld im IP-Header definiert sind. Diese Protokollnummern wurden von der Internet Assigned Numbers Authority (IANA) in einer Datenbank definiert [14]. Somit gibt jedes Byte eine ID eines möglichen Protokolls an. So ist beispielsweise TCP über die Nummer 6 identifiziert und UDP über die Nummer 17.

7.2 Entwicklung des Plugins für OpenDaylight

Dieses Kapitel beschreibt die Entwicklung des Plugins für OpenDaylight, welches die Kommunikation mit dem VS Framework ermöglicht. Dazu gehören sowohl das Empfangen von Nachrichten als auch das Versenden. Zuerst wird in Abschnitt 7.2.1 genauer auf die Mechanismen eingegangen, die zur Anbindung der Software, die das Protokoll verarbeitet, an OpenDaylight benötigt werden . Danach, in Abschnitt 7.2.2 wird auf den Teil des Plugins eingegangen, der die Verbindung zum Endknoten aufbaut und die Pakete ausliest und baut.

7.2.1 Entwicklung der Schnittstelle mit YANG

OpenDaylight stellt zur Anbindung unterschiedlicher Plugins den Service Abstraction Layer bereit. Dieser stellt, wie in Kapitel 4.3 bereits erläutert, eine Schnittstelle dar, die Funktionalität für das Plugin kapselt, aber auch eine Abstraktion für die Anwendungen des Northbound-Interfaces bereit stellt. In OpenDaylight wird hier von Verbrauchern (engl. *consumer*) und Anbietern (engl. *provider*) gesprochen. Ein *provider* stellt Dienste und Funktionen durch eine Northbound-API bereit. Ein *consumer* wiederum nutzt Dienste und Funktionen von *providern* [4]. Plugins stellen in der Regel einen *provider* dar, da Funktionalität bereit gestellt werden soll. Um den Java Code für die verwendeten Datenmodelle zu erzeugen, wird Yet Another Next Generation (YANG) verwendet. Dabei handelt es sich um eine Datenmodellierungssprache, die eine Modellierung von Remote Procedure Calls (RPCs) und Benachrichtigungen via Network Configuration Protocol (NETCONF) erlaubt. NETCONF bietet Netzwerkgeräten die Möglichkeit eine API zur Verfügung zu stellen, durch die das Gerät manipuliert und konfiguriert werden kann. Außerdem kann das Gerät Konfigurationsdaten senden [23]. Im folgenden wird die Modellierung der Daten sowie das Einbinden von Benachrichtigungen und RPCs erläutert. Dabei wurde sich nach dem Toaster Tutorial in [6] gerichtet.

Zuerst muss unter der Nutzung des opendaylight-startup-archetypes ein Projekt angelegt werden. In die erhaltene Projektstruktur wird das Plugin dann entwickelt [6]. Diese Struktur besteht aus den folgenden Modulen:

- api: Hier werden die Modelle für die Daten definiert, also welche Daten innerhalb des Plugins benötigt werden. Hierdurch werden dann REST APIs definiert;
- impl: Hier wird die Funktionalität des Plugins beschrieben, also welche Dienste es bereit stellt. Dabei wird die REST API aus dem Projekt *api* genutzt;
- features: Hier wird das Plugin in die Karaf Instanz eingebunden;
- artifacts: Definiert Abhängigkeiten zu api, impl und features;
- karaf: Hier wird eine Karaf Instanz erstellt, die bereits das Plugin enthält. Diese kann dann wie der Controller ausführen werden.

Praktisch hierbei ist, dass für solche Projekte auch ein eigenständiger Controller (Karaf Instanz) gebaut wird, wodurch ein vollständiges laden des Codes nicht notwendig ist. Nun muss nach folgenden Schritten vorgegangen werden:

1. **Modelle in YANG definieren:** Im Projekt *api* findet man eine YANG-Datei vor, in der das Modell definiert werden soll. Alle benötigten Daten wie beispielsweise die *hostId* sind hier aufgeführt. Diese Daten benötigen Zugriff auf den Datenvermittler (hier: *DataBroker*), welcher für das Speichern und Weiterleiten der Daten zuständig ist [6]. Außerdem werden hier auch die RPCs und Benachrichtigungen sowie benötigte Eingabedaten definiert. In Listing 1 sind zwei Definitionen in YANG aufgeführt. Die erste zeigt, wie ein geeignetes Datenmodell aussehen könnte. Das zweite zeigt die Definition eines RPCs. Diese Definition erzeugt Klassen Host und Link mit den jeweiligen Attributen. So hat Host eine Liste von Links und Link Attribute *linkId* und *name*. Für den RPC wird die Klasse ProtocolNumbers angelegt.

Listing 1: Datenmodel und RPC in YANG

```
        list host {
          leaf hostId {
            type uint32;
            config true;
            description "ID of the host running VS.";
          }
          list link {
            key linkId;
            leaf linkId {
              type uint32;
              config true;
              description "ID of the link.";
            }
            leaf name {
              type string;
              config true;
              description "Name of the link.";
            }
            description "Link capabilities of the host.";
          }
        ...
        }


        rpc setProtocol {
          description "Sends an action to VirtualStack to set a
          protocol and a link for a flow.";
          input {
            leaf hostId {
              type uint32;
              default '0';
              description "ID of the host running VS.";
            }
            list protocolNumbers {
              key protocolNumber;
              leaf protocolNumber {
              type uint8;
              default '7';
              description "Number of the protocol.";
            }
            description "Protocol capabilities of the host.";
          }
          ...
          }
        }
```

2. **Anbindung an Databroker und Codegenratoren:** Im Projekt *impl* wiederum findet sich eine YANG-Datei, die eben diese Verbindung zum DataBroker definieren soll und die Dienste zur Registrierung der RPCs und Benachrichtigungen einbindet. Damit die Daten, welche in den YANG-Dateien modelliert werden auch Code generieren, müssen in den pom-Dateien der jeweiligen Projekte noch Codegeneratoren und Abhängigkeiten (engl. *dependencies*) eingebunden werden.

3. **Plugin einbinden:** Jetzt kann in einem zusätzlichen Projekt *config* im Ordner *src/main/resources/initial* eine xml-Datei angelegt werden. Diese beschreibt das Plugin und wird beim Start des Controller geladen. Sie referenziert die zugehörige Instanz des DataBrokers, der RPCRegistry und des NotificationService. Wenn neue Pakete innerhalb des Controllers installiert werden, werden diese Dateien runtergeladen und die entsprechende config-Datei verfügbar. Dieses wird dann bei jedem Start automatisch geladen. Die Datei für das Plugin muss noch unter *con-*

troller/opendaylight/commons/opendaylight/pom.xml und in der *features.xml* im selben Verzeichnis eingebunden werden.

Diese Arbeiten sind zur Erstellung einer Schnittstelle zwischen dem Plugin und OpenDaylight notwendig. In den vorgesehenen Klassen muss noch die Funktionalität implementiert werden. Wichtig dafür ist eine zentrale Klasse, die eigenständig angelegt werden muss. In dieser werden Metadaten gesetzt sowie Methoden implementiert, die Zustandsänderungen an den DataBroker weiterleiten, der diese dann abspeichert. Zudem wird in Methoden die Funktionalität für die RPCs und die Benachrichtigungen beschrieben. Benachrichtigungen werden in der Bespielanwendung in [6] nur zum Ausgeben von Meldungen bezüglich des Zustand genutzt. Hier sollen Daten ausgegeben werden, die von VS empfangen wurden. Die Hauptklasse wird in der Provider-Klasse instanziiert und der DataBroker gesetzt. In dieser Klasse wird dann auch die Instanz der Hauptklasse für den RPC Dienst registriert.

7.2.2 Das Plugin

Zusätzlich zu der Schnittstelle zu OpenDaylight müssen nun noch Socketverbindungen implementiert werden sowie Klassen zum Codieren und Decodieren der Pakete:

- **ConnectionHandler und ConnectionRunner:** Zur Abfertigung von beliebig vielen Clients, wurde die Klasse ConnectionHandler implementiert, die jede akzeptierte Clientverbindung in einem neuen Thread behandelt. Dazu wurde zuerst ein ServerSocket erstellt, welches dann mit *accept()* auf eingehende Clientverbindungen horcht. Verbindet sich ein Client mit dem Server wird ein neuer Thread mit einer neuen Instanz der ConnectionRunner Klasse erstellt, in welcher die Verbindung zum Client behandelt wird. Für jeden Client wurden außerdem das Senden und Empfangen in separate Threads ausgelagert, um diese beiden Aktionen gleichzeitig durchführen zu können. Durch Erstellung der ConnectionRunner-Instanz wird automatisch die Methode *receiveReport* aufgerufen, die kontinuierlich am Socket auf eingehende Pakete horcht. Wenn Daten empfangen werden, gehen zuerst 4 Byte ein, die die Länge des darauf folgenden Pakets enthalten. Dann wird das eigentliche Paket gelesen und an den Decoder weitergeleitet. Um Daten an den Client zu senden, bekommt die Methode *sendAction* des zuständigen ConnectionRunners alle notwendigen Daten, gibt diese an den Encoder und schreibt das fertige Paket auf das Socket. Für das Lesen und Schreiben wurden die Klassen InputStream und OutputStream verwendet. Außerdem enthält die ConnectionHandler Klasse eine *exit()*-Methode, die bei Beendigung der Anwendung alle Sockets und Streams schließt.
- **Decoder:** Der Decoder ist so implementiert, dass er das empfangene Paket als Parameter bekommt und alle Daten daraus extrahiert. Diese werden dann auch gleich gespeichert, um später zur Verfügung zu stehen. Außerdem bekommt er den ConnectionRunner mitgegeben, der sich um die Verbindung mit dem zugehörigen Host kümmert. Dieser wird für den Host abgespeichert und kann später zum Senden der Daten verwendet werden. Wenn der Decoder nun über die Methode *decodePacket* ein Paket zum dekodieren bekommt, prüft er zunächst, ob die Checksumme korrekt ist. Hierzu wird eine CRC32 Checksumme verwendet und diese dann durch XOR auf eine Länge von einem Byte gebracht. Damit wird sichergestellt, dass das Paket korrekt übermittelt wurde. Danach wird der Befehlstyp überprüft und die entsprechende Methode zum weiteren dekodieren der Daten ausgeführt - je nach Befehl *registerHostOnlyHID*, *registerHost*, *newLink*, *newFlow* oder *flowClosed*. Listing 2 zeigt die Methode, welche ausgeführt wird, wenn eine Benachrichtigung bezüglich eines neuen Flows eingeht. Da die Pakete nach einem strickten Schema aufgebaut sind, können die einzelnen Bytes fest einer bestimmten ID zugeordnet werden.

Listing 2: Dekodierung eines Pakets

```
1        /*
2        * Extracts information from the payload to register a new flow
3        * @param payload the payload of the packet
4        */
5        private void newFlow(byte[] payload) {
6          byte[] hostId = Arrays.copyOfRange(payload, 0, 16);
7          byte[] linkId = Arrays.copyOfRange(payload, 16, 20);
8          int maxIndexStackPlusOne = payload[20] + 21;
9          LinkedList<Stack> stacks = new LinkedList<Stack>();
10         for(int i=21; i<maxIndexStackPlusOne; i+=3) {
11           Stack stack = new Stack(payload[i], payload[i+1], payload[i+2]);
12           stacks.add(stack);
13         }
14         int maxIndexIPPlusOne = maxIndexStackPlusOne + 1
```

```
15                  + payload[maxIndexStackPlusOne];
16              byte[] ip = Arrays.copyOfRange(payload, maxIndexStackPlusOne + 1,
17                  maxIndexIPPlusOne);
18              byte[] flowId = Arrays.copyOfRange(payload, maxIndexIPPlusOne,
19                  maxIndexIPPlusOne + 4);
20              db.saveNewFlow(hostId, linkId, stacks, ip, flowId);
21          }
```

- **Encoder:** Im Encoder hingegen werden die Daten, welche zur Übermittlung an VS ausgewählt wurden zu einem Paket gebaut. Da hier nur eine Action implementiert wurde, werden alle Daten an die Methode *changeSetProtocol* gegeben. In dieser Methode werden alle Daten in ein Byte-Array geschrieben. Dazu gehören auch das Setzen eines Timestamps, des Befehlscodes und der CRC32 Checksumme, welche ebenfalls im Encoder berechnet wird. Der Encoder wird bereits aus der Methode *sendAction* aus der ConnectionRunner Instanz aufgerufen, die zu dem entsprechenden Host gehört. Diese erhält das fertige Paket zurück, schreibt dann zuerst die Paketlänge auf das Socket und dann das Paket.

Abbildung 9 zeigt die Kommunikationspfade der Klassen. Es wurde zuerst ein Dummy implementiert, welcher zusätzlich über eine GUI verfügt. Sie verfügt über ein simples Interface, zum Anzeigen durchgeführter Aktionen und empfangener Reports. Außerdem ist es möglich den Host und alle anderen Daten, die an diesen gesendet werden sollen, auszuwählen. Es wurde zudem die Klasse *DataBroker* sowie *Host*, *Link*, *Flow* und *Stack* zur Verwaltung und Speicherung der Daten zur Laufzeit angelegt. Außerdem verfügt der DataBroker über Methoden zur Darstellung der IDs und Stacks in der GUI.

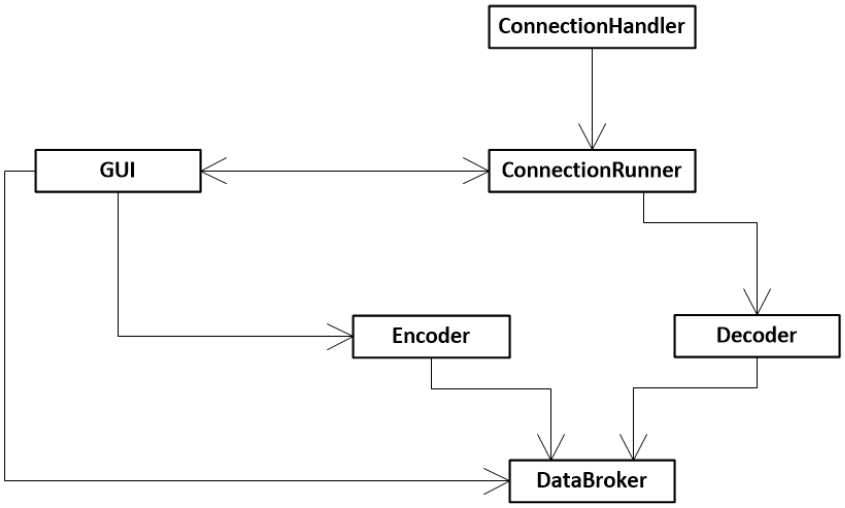

Abbildung 9: Controller-Dummy

7.3 Erweiterung von VirtualStack

In diesem Abschnitt widmen wird die Erweiterung von VirtualStack erläutert. Dazu werden im ersten Abschnitt 7.3.1 Designentscheidungen dargelegt, die zum aktuellen Aufbau der Erweiterung geführt haben. Zudem wird eine Übersicht über die implementierten Klassen gegeben. In 7.3.2 wird dann ausführlich auf die Implementierung eingegangen und die Funktionen einzelner Klassen und Methoden genauer erklärt.

7.3.1 Designentschiedungen

Um eine Kommunikation mit dem SDN-Controller zu ermöglichen, wurde sich für ein modulares Design entschieden. Wie in Abschnitt 6.2 erwähnt, verfügt VS bereits über ein Management Interface, welches zur Kommunikation mit einer Kontroll-Instanz wie einem SDN-Controller verwendet werden kann. Deshalb haben wir uns dazu entschieden, dieses Interface für den Verbindungsaufbau und das Senden und Empfangen von Paketen vom und zum Controller zu verwenden. Des Weiteren sind ein Encoder und ein Decoder implementiert worden. Der Decoder liest die Daten aus den Paketen, die vom SDN-Controller empfangen werden. Der Encoder baut, je nach Befehl, die Pakete, die an den Controller gesendet

werden sollen. Außerdem haben wir uns für eine zentrale Schnittstelle zwischen der Steuereinheit des VS und den Methoden, die zur Übermittlung und dem Empfang der Daten benötigt werden, entschieden. Die Management API kapselt alle diese Funktionalität und setzt die empfangenen Befehle des SDN-Controllers um. Außerdem kann der Manager des VS durch einen simplem Aufruf nach einem bestimmten Ereignis einen Report an den Controller auslösen. Die Management API kapselt außerdem Abläufe wie das generieren eines Host Identifiers, sowie Erkennung der verbundenen Links. Sendet der Controller nun Daten an den Endknoten, gehen diese über das Management Interface ein, werden von der Management API empfangen und dann weiter an den Decoder gegeben, der dekodierte Daten an die Management API zurückgibt. Diese wiederum kann die empfangenen Daten dann im VirtualStack umsetzen und die Aktionen für die Flows ausführt.
Zum Sendern von Paketen ruft der Manager bei bestimmten Aktionen eine Funktion in der Management API auf und löst so aus, dass die Daten von der Management API zum bauen eines Pakets an den Encoder gegeben werden. Die Management API erhält ein Paket zurück und gibt dieses an das Management Interface weiter. Diese wiederum sendet das Paket über ein Socket an den SDN-Controller.
Das Empfangen von Aktionen und das Erfassen der Flows vom VS bzw. das Senden von Mitteilungen an den Controller müssen nebenläufig ablaufen. Es muss kontinuierlich auf das Eingehen von Nachrichten gewartet werden, was die Verwendung von mehreren nebenläufigen Threads (engl. *multithreading*) erforderlich macht.

7.3.2 Implementierung

Im Folgenden wird im Detail auf die Implementierung der Erweiterung für VirtualStack eingegangen.

Die Schnittstelle zwischen dem SDN-Controller und dem VS ist das Management Interface. Dieses richtet ein Socket

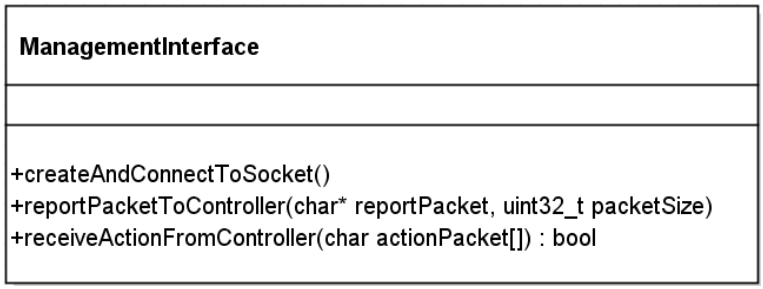

Abbildung 10: Methoden des Management Interface

ein und baut eine Verbindung zu diesem auf. In unserem Fall handelt es sich um ein Socket, welches TCP [13] verwendet, da gewährleistet werden muss, dass die Daten auch wirklich beim Controller eingehen bzw. aus Sicht des Controllers, dass die Daten beim Endknoten eingehen. Würde beispielsweise eine Mitteilung, dass ein bestimmter Link existiert, verloren gehen, könnte der Controller niemals die Verwendung dieses Links verlangen. Das würde den Nutzen der ganzen Kommunikation erheblich einschränken. Abbildung 10 zeigt den Aufbau der Management Interface Klasse.
Wie Abbildung 10 zeigt, gibt es die Methode *createAndConnectToSocket*, die sich um die Erstellung des Sockets kümmert. Das Socket wird dazu zunächst erstellt. Danach wird die IP-Adresse des Servers sowie der zu verwendende Port über *vsSocketAddr.sin_addr.s_addr* und *vsSocketAddr.sin_port* angegeben. Dann wird ein *connect* aufgerufen, welches das Socket mit der *sockaddr_in* verbindet, also einer IP und einem Port zuordnet, zu sehen in Listing 3.

Listing 3: Verbindungsaufbau zum Server

```
1      struct sockaddr_in vsSocketAddr;
2      bzero((char *) &vsSocketAddr, sizeof(vsSocketAddr));
3      vsSocketAddr.sin_family = AF_INET;
4      vsSocketAddr.sin_addr.s_addr = inet_addr("10.0.0.1");
5      vsSocketAddr.sin_port = htons(51234);
6      if(connect(socket_fd, (struct sockaddr*) &vsSocketAddr,
7        sizeof(vsSocketAddr)) < 0)
8      {
9      ..
```

Die Methode *reportPacketToController* realisiert schließlich den Schreibzugriff auf das Socket über ein simples *write*. Hierzu erhält diese ein Character-Array, welches das Paket enthält, das gesendet werden soll und die Länge des Pakets als Integer. Diese wird zuerst übertragen. Das Empfangen der Daten ist in der Methode *receiveActionFromController* implementiert. Diese liest zuerst vier Bytes, die die Paketlänge enthalten und dann jedes Byte einzeln mit einem *read* vom

Socket in einen Puffer. Dieser Puffer wird der Methode schon als Parameter mitgegeben und steht nach dem Lesen der Management API zur Verfügung.

Als nächstes soll ein genauer Blick auf das Bauen und Lesen der Pakete geworfen werden. Der Decoder verfügt zum lesen der Daten aus den Paketen die nötigen Methoden. In der Implementierung ist bisher nur eine Aktion zum Ändern der verwendeten Protokolle und/oder des Links vorgesehen. Die Methode *decodeAction* erhält das Paket als Parameter und schreibt die Daten aus dem Character-Array in die Parameter *flowId*, *protocols* und *linkId*, die als Rückgabewerte verwendet werden. Diese kann die Management API danach weiter verarbeiten. Bei den ID's handelt es sich um ganzzahlige vorzeichenlose Werte mit einer Länge von 4 Byte (*uint32_t*). Der Parameter *protocols* ist als Character-Array deklariert.
Der Encoder befasst sich mit dem Bauen der Pakete für die unterschiedlichen Mitteilungen an den SDN-Controller. Dazu verfügt er über die Methoden *registerHost*, *newLink*, *newFlow* und *flowClosed*. Diese Methoden bekommen jeweils einen Zeiger auf das zu bauende Paket sowie alle Informationen, die in der Nutzlast (engl. *payload*) des Pakets enthalten sein sollen, mit. Abbildung 11 zeigt die Signaturen der Methoden. Die Daten, die an den Encoder gehen, werden von der

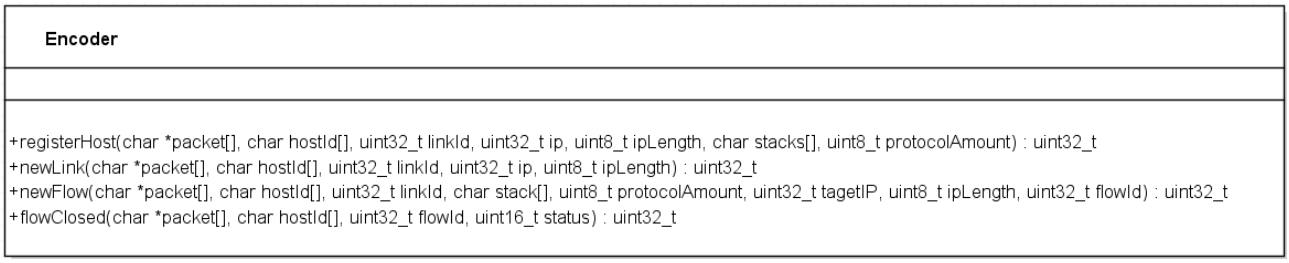

Abbildung 11: Methoden des Encoders

Management API verwaltet. Der Zeiger auf das Paket ist hier notwendig, da in der Management API Daten für Pakets generiert bzw. geschrieben werden. Außerdem wird das Paket selbst mit einer festen Größe erstellt und die eigentliche Größe nach der Generierung des Pakets vom Encoder zurückgegeben. Weiteres dazu später.

Nun soll die Hauptschnittstelle genauer beleuchten werden. Die Management API verwaltet die Daten, kümmert sich um Aufrufe des Managers, löst das Bauen und Lesen der Pakete aus und sendet und empfängt Pakete über das Management Interface. Die Aufgaben der einzelnen Methoden werden im Folgenden anhand der Ereignisketten, die beim Senden von Mitteilungen und beim Empfangen von Aktionen ablaufen, erläutert. Wird der VS gestartet, laufen zunächst folgende Schritte ab:

1. Zuerst wird im Konstruktor des Managers eine neue Management API erstellt, auf deren Methoden der Manager Zugriff hat.

2. Im Konstruktor der Management API wird daraufhin ein neues Management Interface erstellt und eine Verbindung zum Socket hergestellt (sieht oben). Ebenso werden ein neuer Encoder und ein neuer Decoder generiert. Außerdem werden Methoden zur Erstellung der *hostId* und zum Lesen der IP des Hosts aufgerufen. Die gelesenen/generierten Daten werden in der Management API verwaltet und können später in Paketen verwendet werden.

3. Außerdem wird danach die Methode *receiveDataFromController* in einem neuen Thread im Konstruktor des Managers aufgerufen. Diese wartet auf Daten, die über das Socket kommen.

4. Dann werden im Konstruktor des Managers dessen Methoden *init* und *start* ebenfalls in einem neuen Thread aufgerufen. Der Manager kann nun eingehende Flows erkennen und identifizieren.

Bei bestimmten Ereignissen ruft der Manager nun eine entsprechende Methode aus der Management API auf, welche wiederum eine Nachricht an den Controller zur Folge hat. Der genaue Ablauf wird im Folgenden genauer erläutert:

Senden

1. Direkt zu Beginn in der Methode *start* wird *registerNewHost* der Management API aufgerufen, die mitteilen soll, dass auf diesem Host nun VirtualStack läuft und ab diesem Zeitpunkt entsprechende Aktionen vom SDN-Controller empfangen werden können. Ebenso wird bei Identifizierung eines neuen Flows *newFlow* aufgerufen. Zu Anfang der Hauptschleife wird außerdem nach neuen Links gescannt und der Timestamp für

jeden bekannten Flow erhöht. Flows, von denen über einen längeren Zeitraum keine Pakete mehr empfangen wurden, werden über *flowClosed* geschlossen.

2. In der Management API werden in diesen Methoden dann Pakete erstellt. Dazu wird die Länge ermittelt und der Zeitstempel sowie Instruktionstyp festgelegt. Über diese beiden Byte wird die Checksumme generiert. Alle notwendigen Daten sowie das Paket werden an den Encoder gegeben. Hier werden auch die zu Anfang generierten und gelesenen Daten wie die *hostId* verwendet. Der Encoder liefert dann ein fertig gebautes Paket zurück.
3. Dann wird über die Methode *sendDataToController* die Methode *reportPacketToController* des Management Interface aufgerufen, welche die Daten auf das Socket schreibt.

Das Empfangen der Daten findet nebenläufig zum Erfassen der Flows im Manager und zum Senden statt. Im Folgenden der genaue Ablauf und die Methodenaufrufe:

Empfangen

1. Eine vom SDN-Controller an den Endknoten gesendete Aktion geht über das Socket im Management Interface ein. Die Methode *receiveActionFromController* liest die Daten vom Socket und gibt sie an die *receiveDataFromController* Methode der Management API zurück.
2. Diese erkennt den Eingang der Daten und verarbeitet sie weiter. Zuerst wird die Checksumme überprüft und danach der Typ der Instruktion. Dann wird das Paket an den Decoder gegeben.
3. Der Decoder liest die Daten aus dem Paket in die zugehörigen Parameter. Diese erhält die Management API zurück.
4. Nun wird überprüft, ob der Flow vorhanden ist. Wenn ja, werden dessen vorhandene Stacks ebenfalls überprüft. Stacks, die mit den Protokollen aus dem Paket übereinstimmen, werden als aktiv gesetzt. Stacks, die für den Flow noch nicht vorhanden sind, werden gebaut.

Um empfangene Daten auch schreiben zu können, erhält die Management API zu Initialisierung einen Zeiger auf den *flow table* des Managers. Dieser wiederum bietet Zugriff auf alle Daten, die die Flows betreffen.

Zu guter Letzt wollen wir uns noch dem Problem der Nebenläufigkeit widmen. Um dem VS einen normalen Ablauf zu ermöglichen und gleichzeitig Nachrichten empfangen zu können, brauchen wir zwei nebenläufige threads. Listing 4 zeigt das Hervorbringen der threads im Konstruktor der Manager Klasse.

Listing 4: Hervorbringen der threads

```
1   mgAPI = new ManagementAPI();
2   std::thread receiver([&]() {
3     mgAPI->receiveDataFromController();
4   });
5   std::thread managerThread([&]() {
6     this->init();
7     this->start();
8   });
9   receiver.join();
10  managerThread.join();
```

Auf diese Weise kann der Manager Flows überwachen und Nachrichten an den Controller senden und doch gleichzeitig auf eingehende Nachrichten vom Controller lauschen. Dabei muss darauf geachtet werden, dass Daten, auf die beide Threads zugreifen, nicht gleichzeitig geschrieben und gelesen werden. Dazu haben wir Blocking-Mechanismen implementiert unter der Nutzung des Header *mutex.h*. Das *unique_lock* wird jeweils in *receiveDataFromController* und den Methoden *registerHost*, *newLink*, *newFlow* und *flowClosed* erstellt. Danach wird versucht zu sperren und falls es gelingt, der eigentliche Methodenrumpf ausgeführt. Anschließend wird die Sperre (engl. *lock*) wieder aufgehoben. Listing 5 zeigt die Methode *newFlow*, in welcher ein *lock* implementiert ist.

Listing 5: Methode *newFlow* aus der Management API

```
1   /**
2   * @brief ManagementAPI::newFlow reports a new flow with hostId, linkId,
3   * the related protocol stack and the target ip of the flow to the controller.
```

```
4       * @param flowId the id for the reported flow.
5       */
6       void ManagementAPI::newFlow(uint32_t flowId, int flowIndex)
7       {
8         std::unique_lock<std::mutex> lock(mutex, std::defer_lock);
9         while(true)
10        {
11          if(lock.try_lock())
12          {
13            flowIds.push_back(flowId);
14            char* packet = new char[28];
15            char protocols[3];
16            protocols[0] = flow_id_LUT[flowIndex].stacks[0]->sendstack->protocolId;
17            protocols[1] = 4;
18            protocols[2] = 97;
19            uint32_t packetSize = encoder->newFlow(&packet, ownUUID, linkIds.at(0),
20              protocols, 3, flow_id_LUT[flowIndex].endpoint->flow_target.sin_addr.s_addr,
21              4, flowId);
22            packet[0] = (char) std::time(nullptr);
23            packet[1] = (char) 0x53;
24            packet[2] = createChecksum(packet);
25            sendDataToController(packet, packetSize);
26            LOG(INFO) << "ManagementAPI: Registration of new flow initialized!";
27            break;
28          }
29        }
30        lock.unlock();
31      }
```

8 Evaluation mit dem Topology Management Tool (ToMaTo)

In diesem Kapitel wird im Detail auf die Evaluation der implementierten Software eingegangen. Dazu wurde Topology Management Tool (ToMaTo) genutzt, welches eine Oberfläche zum Bau von virtuellen Netzwerken bietet, die auf den Servern von ToMaTo mittels Virtueller Maschinen (VMs) realisiert werden können. Dazu können nicht nur vorgefertigte acpVM verwendet, sondern auch eigene hochgeladen und eingesetzt werden [12]. In diesem Kapitel wird der Versuchsaufbau erläutert sowie die zu evaluierenden Kriterien dargelegt. Danach wird auf die erzielten Ergebnisse eingegangen und Schlussfolgerungen daraus gezogen.

8.1 Evaluationskriterien

Diese Arbeit ist als Proof of Concept angelegt und soll daher zeigen, dass eine Kommunikation zwischen einem Endknoten, auf dem das VirtualStack Framework läuft mit einem SDN-Controller möglich ist und auch vom Controller gesteuert werden kann. Daher haben wir uns dafür entschieden, die korrekte Ausführung der folgenden Befehle zu überprüfen.

1. Melden eines neuen Hosts (*register new host*)
2. Melden eines neuen Flows (*new flow*)
3. Melden eines neuen Links (*new link*)
4. Schließen eines Flows (*flow closed*)
5. Anpassung des verwendeten Protokolls und des Links (*set protocol*)

8.2 Versuchsaufbau

Um die Evaluationskriterien zu überprüfen, wurde eine Topologie erstellt, welche aus einer Virtuellen Maschine besteht, auf welcher der Controller-Dummy ausgeführt wird sowie einem Switch, der diesen mit zwei VMs verbindet, auf denen VirtualStack läuft, zu sehen in Abbildung 12. Um den Controller-Dummy mit GUI testen zu können wurde xfce auf der Maschine installiert. Die GUI des Controller-Dummys hat ein Textfeld zur Ausgabe von Nachrichten. Diese geben beispielsweise an, welche Nachrichten von VS empfangen und welche gesendet wurden. Außerdem verfügt sie über drei Comboboxen in denen man die *hostId* des Hosts an den die Nachricht gesendet werden soll sowie eine *linkId* als auch eine *flowId* auswählen kann. Eine Liste ermöglicht die Auswahl mehrerer Protokollstacks. Somit können alle Daten, die für eine Nachricht *setProtocol* benötigt werden, über die GUI ausgewählt werden. Die GUI ist darüber hinaus so implementiert, dass nur Daten ausgewählt werden können, die zu dem ausgewählen Host gehören. So kann beispielsweise keine *linkId* selektiert werden, welche für den Host der vorher gewählten *hostId* nicht vorhanden ist. Auf den Virtuellen Maschinen, auf denen VS ausgeführt wird, ist keine GUI notwendig. Allerdings musste hier zunächst ein TUN-Gerät hinzugefügt werden und dessen IP-Konfiguration angepasst werden. Da gezeigt werden soll, dass alle implementierten Befehle sowohl vom Controller-Dummy als auch von VS korrekt verarbeitet werden, dienen Ausgaben in Log-Dateien als Grundlage. Zur Erzeugung von Flows für VirtualStack wurden Python-Scripte verwendet.

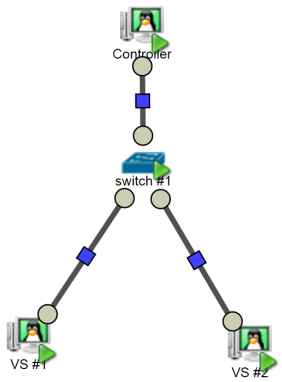

Abbildung 12: Topologie zur Evaluation

8.3 Auswertung der Ergebnisse

In den Log-Dateien sind alle wichtigen Schritte des Controller Dummys und vom VirtualStack aufgelistet. Es wird dokumentiert, wann Pakete fertig gebaut oder dekodiert sind, wann sie empfangen oder gesendet werden und auch einige wichtige Aktionen dazwischen. Im Folgenden sind Ausschnitte aus den Log-Dateien aufgeführt, die das Bauen, Senden und Empfangen der *newFlow* und *flowClosed* Nachrichten zeigen. Die vollständigen Log-Dateien sind im Anhang zu finden.

```
virtualstack_log_1.log:
2016-07-09 10:32:25,495 INFO  [default] Encoder: Register new flow report encoded successfully!
2016-07-09 10:32:25,496 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:25,505 INFO  [default] ManagementAPI: Registration of new flow initialized!
2016-07-09 10:32:25,508 INFO  [default] Encoder: Register flow closed report encoded successfully!
2016-07-09 10:32:25,509 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:25,510 INFO  [default] ManagementAPI: Registration of flow closed initialized!

Controller_LogFile.log:
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Flow saved!
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: Flow closed without error!
```

Direkt nach dem Start senden beide VSs die Nachrichten *registerNewHost* und gleich darauf *newLink* für jeden Link, den sie finden können. Für den ersten VirtualStack wurde ein Flow erstellt, der eine feste Anzahl an Paketen sendet und dann aufhört. Dadurch wird zuerst der Report *newFlow* ausgelöst und nach Abbruch des Flows korrekterweise *flowClosed*. Für den zweiten VS wurde sich für einen kontinuierlichen Datenstrom entschieden. Dadurch können nach der eingehenden *newFlow* Benachrichtigung, mehrere *setProtocol* Aktionen vom Controller gesendet werden. Alle lösen ein korrektes Setzen des jeweilig ausgewählten Stacks aus. Der allgemeine Ablauf einer Kommunikation zwischen dem Controller-Dummy und VS wird in Abbildung 13 gezeigt. Anzumerken ist außerdem, dass VS noch keine Protokollstacks, die IPv6 verwenden,setzen kann. Daher wurden diese durch die äquivalenten IPv4 Protkollstacks ersetzt.

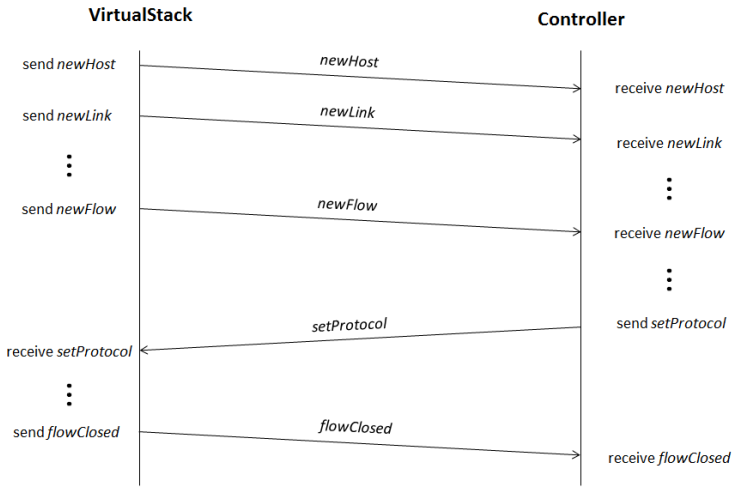

Abbildung 13: Nachrichtensequenzdiagramm des Protokolls

9 Fazit

In dieser Arbeit wurde ein Protokoll zur Kommunikation zwischen VirtualStack und einem SDN-Controller definiert. Außerdem wurden Befehle vorgestellt, die durch das Protokoll zwischen VS und einem SDN-Controller übermittelt werden können. Davon wurden fünf Befehle implementiert und getestet. OpenDaylight wurde als Controller ausgewählt wegen seiner Präsenz auf dem Markt und den umfangreichen Features. VirtualStack hingegen bietet eine Chance darauf, dass Anwendungen im Netzwerk die Möglichkeiten von SDN besser Nutzen können. Spezielle Protokolle werden durch VirtualStack beibehalten, wodurch die Sofware besser genutzt werden kann. VS wurde um Methoden und Klassen zur Verarbeitung des Protokolls erweitert und erhielt die Möglichkeit eine Verbindung zum Controller aufzubauen. Es wurde ein Dummy entwickelt, der die Seite des Controllers übernahm. Dieser stellte einen Server bereit, mit dem sich mehrere Clients verbinden konnten. Die Evaluation hat gezeigt, dass alle Befehle korrekt verarbeitet wurden, sowohl von VirtualStack als auch von der Controller-Seite. In Zukunft können noch weitere Befehle implementiert und neue hinzugefügt werden, um so ein Protokoll zur Steuerung des VirtualStack Frameworks auf den Endknoten im Netzwerk zu ermöglichen.

Literatur

[1] Build sdn agilely - component-based software defined networking framework. https://osrg.github.io/ryu/. Letzter Zugriff: 04.07.2016.

[2] The controller. https://floodlight.atlassian.net/wiki/display/floodlightcontroller/The+Controller. Letzter Zugriff: 16.03.2016.

[3] Floodlight - module applications. https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Module+Applications. Letzter Zugriff: 16.03.2016.

[4] Opendaylight controller: Sal architecture overview. https://wiki.opendaylight.org/view/OpenDaylight_Controller:_SAL_Architecture_Overview. Letzer Zugriff: 21.05.2016.

[5] Opendaylight controller:architectural framework. https://wiki.opendaylight.org/view/OpenDaylight_Controller:Architectural_Framework. Letzter Zugriff: 09.12.2015.

[6] Opendaylight controller:md-sal:toaster step-by-step. https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Toaster_Step-By-Step. Letzter Zugriff: 16.05.2016.

[7] Opendaylight platform architecture. https://www.opendaylight.org/lithium. Letzter Zugriff: 09.12.2015.

[8] Opendaylight: Research, education and government use cases. https://www.opendaylight.org/research-ed-government. Letzter Zugriff: 02.05.2016.

[9] Project proposals:odl-sdni app. https://wiki.opendaylight.org/view/Project_Proposals:ODL-SDNi_App. Letzter Zugriff: 15.03.2016.

[10] Protokollstack. http://www.itwissen.info/definition/lexikon/Protokollstack-protocol-stack.html. Letzter Zugriff: 01.07.2016.

[11] Software-defined networking: The new norm for networks. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf. Letzter Zugriff: 14.12.2015.

[12] Tomato. http://master.tomato-lab.org/. Letzter Zugriff: 09.12.2015.

[13] Transmission control protocol, protocol specification. https://tools.ietf.org/html/rfc793, 09 1981. Letzert Zugriff: 07.05.2016.

[14] Internet Assigned Numbers Authority. Protocol numbers. http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml. Letzter Zugriff: 25.04.2016.

[15] S. Brim B. Carpenter. Middleboxes: Taxonomy and issues. https://www.rfc-editor.org/rfc/pdfrfc/rfc3234.txt.pdf, 2002. Letzter Zugriff: 01.03.2016.

[16] OpenFlow Switch Consortium et al. Openflow switch specification version 1.3.2, 2013.

[17] David Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[18] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[19] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.

[20] Jens Heuschkel, Immanuel Schweizer, and Max Muhlhauser. Virtualstack: A framework for protocol stack virtualization at the edge. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 386–389. IEEE, 2015.

[21] Zuhran Khan Khattak, Muhammad Awais, and Adnan Iqbal. Performance evaluation of opendaylight sdn controller. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 671–676. IEEE, 2014.

[22] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76, 2015.

[23] Ed. M. Bjorklund. Yang - a data modeling language for the network configuration protocol (netconf). `https://tools.ietf.org/html/rfc6020`. Letzter Zugriff: 16.05.2016.

[24] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[25] Ryu project team. *Ryu SDN Framework*. `https://osrg.github.io/ryu-book/en/Ryubook.pdf`.

[26] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). `https://tools.ietf.org/html/rfc4271`, 01 2006. Letzter Zugriff: 15.03.2016.

[27] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013.

[28] Tong Zhou, Gong Xiangyang, Yannan Hu, Xirong Que, and Wang Wendong. Pindswitch: A sdn-based protocol-independent autonomic flow processing platform. In *Globecom Workshops (GC Wkshps), 2013 IEEE*, pages 842–847. IEEE, 2013.

Anhang

Controller_LogFile.log

```
Jul 09, 2016 10:31:35 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionHandler: Server initialized!
Jul 09, 2016 10:31:56 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionHandler: New client accepted!
Jul 09, 2016 10:31:56 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:31:56 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:31:56 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Link saved!
Jul 09, 2016 10:31:56 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:31:56 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Link saved!
Jul 09, 2016 10:32:05 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionHandler: New client accepted!
Jul 09, 2016 10:32:05 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:32:05 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:32:05 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Link saved!
Jul 09, 2016 10:32:05 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:32:05 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Link saved!
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Flow saved!
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:32:25 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: Flow closed without error!
Jul 09, 2016 10:33:00 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:33:00 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Flow saved!
Jul 09, 2016 10:33:13 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Packet written to socket.
Jul 09, 2016 10:33:17 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Packet written to socket.
Jul 09, 2016 10:33:20 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Packet written to socket.
Jul 09, 2016 10:33:26 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Packet written to socket.
Jul 09, 2016 10:33:30 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Packet written to socket.
Jul 09, 2016 10:33:34 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Packet written to socket.
Jul 09, 2016 10:33:49 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:33:50 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: New Flow saved!
Jul 09, 2016 10:33:50 AM GUI.ControllerDummyUI pushTextForTextArea
```

```
INFO: ConnectionRunner: Read packet from socket.
Jul 09, 2016 10:33:50 AM GUI.ControllerDummyUI pushTextForTextArea
INFO: DataBroker: Flow closed without error!
```

virtualstack_log_1.log

```
2016-07-09 10:27:50,684 INFO  [default] ManagementInterface: Connected to server!
2016-07-09 10:27:50,688 INFO  [default] MANAGER: Try to initialize TUN device...
2016-07-09 10:27:50,689 INFO  [default] Tun allocation done.
2016-07-09 10:27:50,689 INFO  [default] MANAGER: Initialization of TUN device done.
2016-07-09 10:27:50,689 INFO  [default] MANAGER: Start Manager Component...
2016-07-09 10:27:50,689 INFO  [default] Encoder: Register new host report encoded successfully!
2016-07-09 10:27:50,689 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:27:50,689 INFO  [default] ManagementAPI: Registration of new host initialized!
2016-07-09 10:27:50,696 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:27:50,698 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:27:50,700 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:27:50,705 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:27:50,707 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:27:50,708 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:27:50,714 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:27:50,715 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:28:21,773 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:28:21,776 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:28:21,782 INFO  [default] Encoder: Register flow closed report encoded successfully!
2016-07-09 10:28:21,783 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:28:21,784 INFO  [default] ManagementInterface: Finished writing packet to socket.
EP-UDP: Got dest address: 10.0.1.2:53488
2016-07-09 10:28:21,788 INFO  [default] EP-UDP: EP init done.
2016-07-09 10:28:21,791 INFO  [default] ManagementAPI: Registration of flow closed initialized!
2016-07-09 10:28:21,796 INFO  [default] Encoder: Register new flow report encoded successfully!
2016-07-09 10:28:21,798 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:28:21,799 INFO  [default] ManagementAPI: Registration of new flow initialized!
2016-07-09 10:31:56,394 INFO  [default] ManagementInterface: Connected to server!
2016-07-09 10:31:56,400 INFO  [default] MANAGER: Try to initialize TUN device...
2016-07-09 10:31:56,401 INFO  [default] Tun allocation done.
2016-07-09 10:31:56,402 INFO  [default] MANAGER: Initialization of TUN device done.
2016-07-09 10:31:56,403 INFO  [default] MANAGER: Start Manager Component...
2016-07-09 10:31:56,403 INFO  [default] Encoder: Register new host report encoded successfully!
2016-07-09 10:31:56,408 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:31:56,409 INFO  [default] ManagementAPI: Registration of new host initialized!
2016-07-09 10:31:56,410 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:31:56,411 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:31:56,412 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:31:56,417 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:31:56,418 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:31:56,419 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:31:56,420 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:31:56,425 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:32:25,490 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:32:25,492 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:32:25,493 INFO  [default] EP-UDP: Got dest address: 10.0.1.2:54444
2016-07-09 10:32:25,494 INFO  [default] EP-UDP: EP init done.
2016-07-09 10:32:25,495 INFO  [default] Encoder: Register new flow report encoded successfully!
2016-07-09 10:32:25,496 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:25,505 INFO  [default] ManagementAPI: Registration of new flow initialized!
2016-07-09 10:32:25,508 INFO  [default] Encoder: Register flow closed report encoded successfully!
2016-07-09 10:32:25,509 INFO  [default] ManagementInterface: Finished writing packet to socket.
```

```
2016-07-09 10:32:25,510 INFO  [default] ManagementAPI: Registration of flow closed initialized!
2016-07-09 10:33:49,745 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:33:49,748 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:33:49,749 INFO  [default] EP-UDP: Got dest address: 10.0.1.2:57646
2016-07-09 10:33:49,750 INFO  [default] EP-UDP: EP init done.
2016-07-09 10:33:49,751 INFO  [default] Encoder: Register new flow report encoded successfully!
2016-07-09 10:33:49,763 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:33:49,773 INFO  [default] ManagementAPI: Registration of new flow initialized!
2016-07-09 10:33:49,776 INFO  [default] Encoder: Register flow closed report encoded successfully!
2016-07-09 10:33:49,778 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:33:49,779 INFO  [default] ManagementAPI: Registration of flow closed initialized!
```

virtualstack_log_2.log

```
2016-07-09 10:27:56,663 INFO  [default] ManagementInterface: Connected to server!
2016-07-09 10:27:56,669 INFO  [default] MANAGER: Try to initialize TUN device...
2016-07-09 10:27:56,669 INFO  [default] Tun allocation done.
2016-07-09 10:27:56,669 INFO  [default] MANAGER: Initialization of TUN device done.
2016-07-09 10:27:56,669 INFO  [default] MANAGER: Start Manager Component...
2016-07-09 10:27:56,669 INFO  [default] Encoder: Register new host report encoded successfully!
2016-07-09 10:27:56,670 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:27:56,670 INFO  [default] ManagementAPI: Registration of new host initialized!
2016-07-09 10:27:56,678 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:27:56,680 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:27:56,686 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:27:56,687 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:27:56,689 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:27:56,694 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:27:56,696 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:27:56,701 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:32:04,631 INFO  [default] ManagementInterface: Connected to server!
2016-07-09 10:32:04,641 INFO  [default] MANAGER: Try to initialize TUN device...
2016-07-09 10:32:04,643 INFO  [default] Tun allocation done.
2016-07-09 10:32:04,645 INFO  [default] MANAGER: Initialization of TUN device done.
2016-07-09 10:32:04,647 INFO  [default] MANAGER: Start Manager Component...
2016-07-09 10:32:04,653 INFO  [default] Encoder: Register new host report encoded successfully!
2016-07-09 10:32:04,661 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:04,662 INFO  [default] ManagementAPI: Registration of new host initialized!
2016-07-09 10:32:04,664 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:32:04,669 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:04,671 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:32:04,672 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:32:04,677 INFO  [default] Encoder: Register new link report encoded successfully!
2016-07-09 10:32:04,679 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:04,680 INFO  [default] ManagementAPI: Registration of new link initialized!
2016-07-09 10:32:04,685 INFO  [default] ManagementAPI: New link found!
2016-07-09 10:32:59,486 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:32:59,497 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:32:59,501 INFO  [default] EP-UDP: Got dest address: 10.0.1.2:35148
2016-07-09 10:32:59,505 INFO  [default] EP-UDP: Got dest address: 10.0.1.2:35148
Encoder: Register flow closed report encoded successfully!
2016-07-09 10:32:59,509 INFO  [default] EP-UDP: EP init done.
2016-07-09 10:32:59,513 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:59,515 INFO  [default] ManagementAPI: Registration of flow closed initialized!
2016-07-09 10:32:59,521 INFO  [default] Encoder: Register new flow report encoded successfully!
2016-07-09 10:32:59,522 INFO  [default] ManagementInterface: Finished writing packet to socket.
2016-07-09 10:32:59,524 INFO  [default] ManagementAPI: Registration of new flow initialized!
2016-07-09 10:33:13,660 INFO  [default] ManagementInterface: Finished reading packet length from socket!
```

```
2016-07-09 10:33:13,988 INFO  [default] ManagementInterface: Finished reading packet from socket!
2016-07-09 10:33:13,991 INFO  [default] ManagementAPI: Valid checksum!
2016-07-09 10:33:13,997 INFO  [default] ManagementAPI: Valid instruction type!
2016-07-09 10:33:13,999 INFO  [default] Decoder: Action decoded successfully!
2016-07-09 10:33:14,000 INFO  [default] ManagementAPI: Action received for flow: 35148
2016-07-09 10:33:14,014 INFO  [default] ManagementAPI: TCP-IPv4 stack set for flow!
2016-07-09 10:33:17,092 INFO  [default] ManagementInterface: Finished reading packet length from socket!
2016-07-09 10:33:17,420 INFO  [default] ManagementInterface: Finished reading packet from socket!
2016-07-09 10:33:17,423 INFO  [default] ManagementAPI: Valid checksum!
2016-07-09 10:33:17,430 INFO  [default] ManagementAPI: Valid instruction type!
2016-07-09 10:33:17,431 INFO  [default] Decoder: Action decoded successfully!
2016-07-09 10:33:17,433 INFO  [default] ManagementAPI: Action received for flow: 35148
2016-07-09 10:33:17,446 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:33:17,448 INFO  [default] ManagementAPI: UDP-IPv4 stack set for flow!
2016-07-09 10:33:20,842 INFO  [default] ManagementInterface: Finished reading packet length from socket!
2016-07-09 10:33:21,171 INFO  [default] ManagementInterface: Finished reading packet from socket!
2016-07-09 10:33:21,175 INFO  [default] ManagementAPI: Valid checksum!
2016-07-09 10:33:21,176 INFO  [default] ManagementAPI: Valid instruction type!
2016-07-09 10:33:21,193 INFO  [default] Decoder: Action decoded successfully!
2016-07-09 10:33:21,194 INFO  [default] ManagementAPI: Action received for flow: 35148
2016-07-09 10:33:21,196 INFO  [default] ManagementAPI: DCCP-IPv4 stack set for flow!
2016-07-09 10:33:26,388 INFO  [default] ManagementInterface: Finished reading packet length from socket!
2016-07-09 10:33:26,724 INFO  [default] ManagementInterface: Finished reading packet from socket!
2016-07-09 10:33:26,728 INFO  [default] ManagementAPI: Valid checksum!
2016-07-09 10:33:26,732 INFO  [default] ManagementAPI: Valid instruction type!
2016-07-09 10:33:26,738 INFO  [default] Decoder: Action decoded successfully!
2016-07-09 10:33:26,739 INFO  [default] ManagementAPI: Action received for flow: 35148
2016-07-09 10:33:26,749 INFO  [default] ManagementAPI: TCP-IPv4 stack set for flow!
2016-07-09 10:33:30,475 INFO  [default] ManagementInterface: Finished reading packet length from socket!
2016-07-09 10:33:30,820 INFO  [default] ManagementInterface: Finished reading packet from socket!
2016-07-09 10:33:30,823 INFO  [default] ManagementAPI: Valid checksum!
2016-07-09 10:33:30,831 INFO  [default] ManagementAPI: Valid instruction type!
2016-07-09 10:33:30,832 INFO  [default] Decoder: Action decoded successfully!
2016-07-09 10:33:30,845 INFO  [default] ManagementAPI: Action received for flow: 35148
2016-07-09 10:33:30,846 INFO  [default] SS-IPv4UDP: init done.
2016-07-09 10:33:30,847 INFO  [default] ManagementAPI: UDP-IPv4 stack set for flow!
2016-07-09 10:33:34,275 INFO  [default] ManagementInterface: Finished reading packet length from socket!
2016-07-09 10:33:34,612 INFO  [default] ManagementInterface: Finished reading packet from socket!
2016-07-09 10:33:34,615 INFO  [default] ManagementAPI: Valid checksum!
2016-07-09 10:33:34,617 INFO  [default] ManagementAPI: Valid instruction type!
2016-07-09 10:33:34,629 INFO  [default] Decoder: Action decoded successfully!
2016-07-09 10:33:34,630 INFO  [default] ManagementAPI: Action received for flow: 35148
2016-07-09 10:33:34,632 INFO  [default] ManagementAPI: DCCP-IPv4 stack set for flow!
```