Advanced Programming Concepts for Software-defined Networks (SDN)

Stefan Schmid
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart
Matrikelnummer: 3102206
Email: st140748@stud.uni-stuttgart.de

Betreuer: Ruben Mayer, Institut für Parallele und Verteilte Systeme (IPVS)

***Zusammenfassung*—Diese Ausarbeitung befasst sich mit dem Vergleich zweier High-Level-Programmiersprachen für Software-defined Networks (SDN).**
Die Ansätze von Frenetic [12], [5], [4], [3] und Maple [14], [7] versprechen durch das hohe Abstraktionslevel eine deutlich einfachere Programmierung für die Entwickler. Dabei verfolgen die zwei genannten Sprachen jedoch verschiedene Grundideen, die ich im folgenden genauer beleuchten möchte.

I. Einleitung

Die Anforderungen an Netzwerke haben sich in den letzten Jahren stark verändert.
Vor allem Cloud-Anwendungen und Virtualisierung erfordern ein hohes Maß an Flexibilität innerhalb der Netzwerkstruktur, um z.B. optimale Lastverteilung zu garantieren.
Ein verbreiteter Ansatz, um sich diesen Herausforderungen anzunehmen, stellen die Software-defined Networks dar.
Der Trend geht dabei weg von statischen Netzwerken hin zu dynamischeren um automatisiert auf geänderte Anforderungen reagieren zu können. [11][S. 36]

II. Software-defined Networks (SDN)

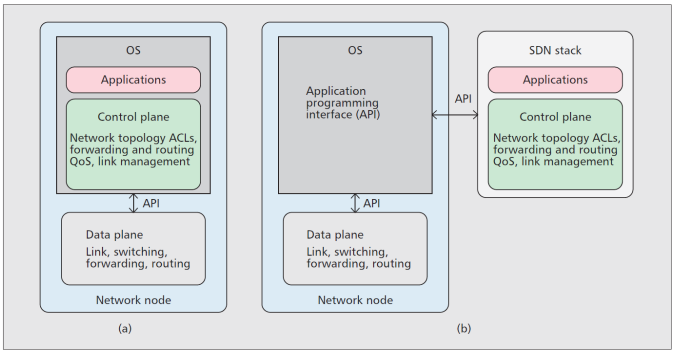

Abbildung 1. Vergleich der herkömmlichen mit der SDN-Netzwerksicht. a) Herkömmlicher Ansatz b) SDN-Ansatz [11][S. 38]

Software Defined Networks (SDN) verfolgen den Ansatz, die Kontrollschicht von der Datenschicht zu abstrahieren.
In Abbildung 1 ist auf der linken Seite schön der traditionelle Ansatz der Netzwerksicht zu sehen. Jeder Knoten beherbergt dabei sein eigenes Betriebssystem, auf welchem Anwendungen und die Routing-Logik läuft. Über eine interne API steuert dieses Betriebssystem die darunterliegende Datenschicht an, welche z.B. für Switching und Forwarding relevant ist. [11][S. 37], [9], [13]

Im Gegensatz wird bei SDNs die Kontrollschicht von den einzelnen Knoten ausgelagert und zentralisiert. Über eine API können auch hier die Knoten angesprochen werden, welche nur noch die untere Datenschicht enhalten, also die Abwicklung zur physischen Übertragung. [11][S. 38]

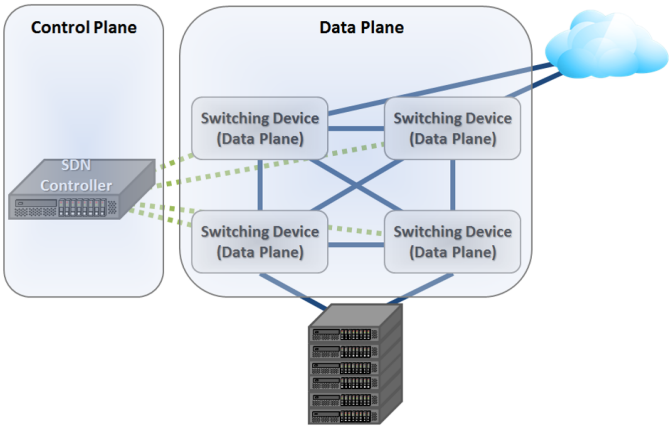

Abbildung 2. Trennung von Kontroll- und Datenschicht in SDN-Netzwerken http://www.definethecloud.net/wp-content/uploads/2012/06/image1.png (aufgerufen am 04.02.2016)

III. OpenFlow Protokoll

OpenFlow ist ein öffentliches Protokoll, um in OpenFlow-fähigen Switches und Routern die Einträge der flow-tables zu manipulieren. [10][S. 70]
In Abbildung 3 wird die grundlegende Spezifikation eines OpenFlow Switches schön dargestellt. Dabei besteht ein OpenFlow Switch aus einem Hardware- als auch einem Software-Teil. Die obere Softwareebene überlagert die Hardware und kommuniziert über das OpenFlow Protokoll

mit dem Controller (z.B. NOX [6]), welcher das SDN verwaltet.
Innerhalb der Switches/Router liegt die sogenannte Flow Table auf der Hardwareebene. Sie enthält Forwarding-Regeln für Pakete, deren Header auf bestimmte Patterns matchen und kann über die Softwareschicht von extern über einen Controller manipuliert werden. [10][S. 70-71], [13]

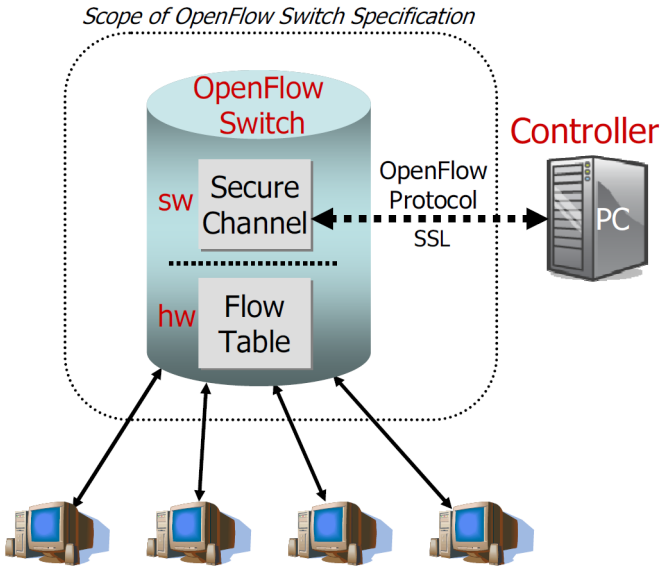

Abbildung 3. Grundlegende Spezifikation eines OpenFlow Switches [10][S. 70]

Die Einträge der Flow Table unterliegen der formalen Definition aus Abbildung 4.

Integers	n
Rules	$r ::= \langle pat, pri, t, [a_1, \ldots, a_n] \rangle$
Patterns	$pat ::= \{h_1 : n_1, \ldots, h_k : n_k\}$
Priorities	$pri ::= n$
Timeouts	$t ::= n \mid \texttt{None}$
Actions	$a ::= \texttt{output}(op) \mid \texttt{modify}(h, n)$
Headers	$h ::= \texttt{in\_port} \mid \texttt{vlan} \mid \texttt{dl\_src} \mid \texttt{dl\_dst} \mid \texttt{dl\_type} \mid \texttt{nw\_src} \mid \texttt{nw\_dst} \mid \texttt{nw\_proto} \mid \texttt{tp\_src} \mid \texttt{tp\_dst}$
Ports	$op ::= n \mid \texttt{flood} \mid \texttt{controller}$

Abbildung 4. Definiton der OpenFlow-Grammatik [4][S. 280]

Tatsächlich in der Tabelle werden dabei die Regeln (Rules) gespeichert. Diese bestehen aus Patterns, welche auf die Paketheader gematched werden, einer Priorität, einem optionalen Timeout und mehreren Aktionen, die für passende Pakete auf das bereits definierte Pattern durchgeführt werden sollen.
Ein Pattern besteht aus mehreren möglichen Header-Informationen, wie z.B. der Quell- oder Ziel-IP-Adresse des Pakets, dem Eingangsport etc.
Die Priorität einer Regel wird über einen simplen Integer-Wert festgelegt, genauso wie auch der Timeout der Regel.
Die durchzuführende Aktion einer Regel kann entweder eine Weiterleitung oder Modifikation des Paket-Headers sein.
Üblicherweise führen OpenFlow Switches auch noch einen Zähler je installierter Regel mit für eventuelle Statistikverwendung. [4][280], [13]

Die Abarbeitung eines Pakets im OpenFLow Switch läuft im Allgemeinen in 3 Stufen ab: [4][S. 280] [11]

1) Anwendung deiner passenden Regel aus der Flow Table. Im Fall, dass diese nicht existiert, wird das Paket direkt an den Controller weitergeleitet und dort weiter verarbeitet.
2) Inkrementierung der Zähler für die entsprechenden Regeln
3) Anwendung der ermittelten Aktionen auf das erhaltene Paket

IV. NOX Controller

Einer der bekanntesten, mehrfach schon erwähnten SDN-Controller ist NOX [6]. Er repräsentiert die Control Plane, wie in Abbildung 2 bereits erwähnt.
NOX versteht sich als eine Art Betriebssystem für Netzwerke. Es abstrahiert die Netzwerksicht auf einem niedrigen Level, sodass nicht mehr mit Maschinencode Vorlieb genommen werden muss.
Darauf aufbauend können Anwendungen für die Netzwerke geschrieben werden. [6][S. 105]
Ein NOX Controller arbeitet eventgesteuert, d.h. es gibt einen Satz an Funktionen, die auf ein bestimmtes Netzwerkverhalten ausgeführt werden: [4],[S. 281] [6][S. 107-108]

- **packet_in(switch,port,packet)**
  Wird getriggert, wenn ein OpenFlow Switch ein Paket an den Controller weiterleitet.
- **stats_in(switch,xid,pattern,packets,bytes)**
  Wird getriggert, sobald der OpenFLow Switch die angeforderten Paket- und Byte-Zähler an den Controller zurück liefert.
- **flow_removed(switch,pattern,packets,bytes)**
  Wird getriggert, wenn eine Regel aufgrund ihres Timeouts entfernt wurde. Die finalen Paket- und Byte-Zähler werden automatisch mit geliefert.
- **switch_join(switch)**
  Wird getriggert, sobald ein neuer Switch dem Netzwerk beitritt.
- **switch_exit(switch)**
  Wird getriggert, sobald ein Switch das Netzwerk verlässt.
- **port_change(switch,port,up)**
  Wird getriggert, sobald an einem Switch ein Port geöffnet oder geschlossen wird.

Um auf die eben genannten Events reagieren zu können, können die OpenFlow Switches über folgende vordefinierte Funktionen manipuliert werden: [4],[S. 281] [6][S. 107-108]

- **install(switch,pattern,priority,timeout,actions)**
  Installiert eine neue Regel in der Flow Table
- **uninstall(switch,pattern)**
  Entfernt eine bestehende Regel aus der Flow Table

- **send(switch,packet,action)** Sendet ein Paket an einen Switch mit einer bestimmten Anweisung zur Weiterleitung.
- **query_stats(switch,pattern)**
  Löst eine Anfrage der Paket- und Byte-Zähler aller Regeln mit einem bestimmten Pattern aus.

V. Probleme bei der NOX-Programmierung von OpenFlow-Netzwerken

Generell erscheint der überschaubare Satz an Events und Send-Methoden eines NOX-Controllers aus dem vorigen Kapitel einleuchtend und relativ simpel in der Implementierung. Das Beispiel eines einfachen Repeaters mit Python und NOX aus Abbildung 5 bestätigt diese These auch zunächst. Eine Repeater-Methode installiert dabei zwei Regeln im Switch. Eine soll von Port 2 auf Port 1 weiterleiten und die andere genau umgekehrt von Port 1 auf Port 2.
Über das in NOX definierte switch_join-Event werden für jeden neu hinzugefügten Switch die repeater-Methode aufgerufen und damit die zwei Regeln in der Flow Table installiert. [4][S. 281]
Dieses einfache Programm stellt durch die Implementierung des passenden Events bereits sicher, das sich das SDN-Netzwerk physisch ändern darf. Dabei spielt es auch keine Rolle mehr, von welchem Hersteller die Switches stammen, denn nur die standardisierte OpenFlow-Schnittstelle muss angesprochen werden.

```
def switch_join(switch):
  repeater(switch)
def repeater(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  install(switch,pat1,DEFAULT,None,[output(2)])
  install(switch,pat2,DEFAULT,None,[output(1)])
```

Abbildung 5. Einfacher Repeater in Python, aufbauend auf NOX [4][S. 281]

Möchte man nun den Repeater aus Abbildung 5 um ein Monitoring erweitern, offenbaren sich schnell versteckte Fehlerursachen, auch wenn die Programmlogik korrekt zu sein scheint.
Wir möchten nun die grundlegende Repeater-Funktionalität um ein Logging der Paketzahlen, welche ausgehend von HTTP-Port 80 an Port 2 eines Switches eintreffen.
Dafür werden in Abbildung 6 ganz primitiv zwei Methoden implementiert, eine zur Installation des gewünschten Monitoring-Patterns, als auch die Event-Funktion, welche die Daten erhalten soll.
Die monitor-Funktion installiert die Regel mit Quellport 80 und Eingangsport 2 im Switch. Danach wird direkt über die NOX-Funktion querystats die Anforderung der Zählerstände angefordert.
In der zugehörigen Event-Methode stats_in werden die Anzahl der bisherigen Bytes ausgegeben und nach 30 Sekunden rekursiv eine neue Anfrage gestartet. Dies hat zur Folge, dass der Zählerstand alle 30 Sekunden ausgegeben wird. [4][S. 281]

```
def monitor(switch):
  pat = {in_port:2,tp_src:80}
  install(switch,pat,DEFAULT,None,[])
  query_stats(switch,pat)
def stats_in(switch,xid,pattern,packets,bytes):
  print bytes
  sleep(30)
  query_stats(switch,pattern)
```

Abbildung 6. Erweiternde Methoden, für das Monitoring des Repeaters [4][S. 281]

Abschließend muss noch die Repeater- und Monitoring-Funktionalität verschmolzen werden. Intuitiv ruft man dazu wie in Abbildung 7 die Repeater und die Monitoring-Funktion nacheinander auf, um die entsprechenden Regeln zu installieren.

```
def repeater_monitor_wrong(switch):
  repeater(switch)
  monitor(switch)
```

Abbildung 7. Vermeintliche Verschmelzung des Repeaters mit der neuen Monitoring-Funktionalität [4][S. 281]

Das Problem mit dieser Implementierung stellen die resultierenden, überlappenden Regeln der Flow Table dar.
Sobald ein Paket an Port 2 ankommt, wird genau eine passende Regel aus der Flow Table gewählt, d.h. es kann nie gesagt werden, ob ein Paket, welches von Port 80 an Port 2 eintrifft gezählt wird oder nicht.
Genau dieses Problem mit überlappenden Regeln ist der Grund dafür, weshalb es keine allgemeingültigen Libraries für bestimmte Monitoring-Funktionalitäten geben kann. [4][S. 182]
Der Repeater-Monitor kann jedoch mittels Prioritäten (siehe Abbildung 8) in den installierten Regeln so umgesetzt werden, dass alle Pakete ausgehend von Port 80 und ankommend an Port 2 sowohl weitergeleitet als auch gezählt werden.

```
def repeater_monitor(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2,tp_src:80}
  install(switch,pat1,[output(2)],DEFAULT)
  install(switch,pat2web,[output(1)],HIGH)
  install(switch,pat2,[output(1)],DEFAULT)
  query_stats(switch,pat2web)
```

Abbildung 8. Korrekte Verschmelzung des Repeaters mit der neuen Monitoring-Funktionalität [4][S. 281]

Unabhängig von den Problemen mit überlappenden Regeln, macht das Low-Level Interface Ärger. Eine einfache logische Differenz zwischen zwei Patterns muss mit überlappenden Regeln und unterschiedlichen Prioritäten gesteuert werden. Dadurch wird zum einen die Komplexität für den Programmierer enorm gesteigert, als auch die spätere Nachvollziehbarkeit

des Codes erheblich erschwert.
Weitere Schwierigkeiten bereiten z.B. asynchrone, parallele Anwendungen, bei denen Pakete nicht in einer festgelegten Reihenfolge eintreffen müssen.
Dies kann zu Race Conditions führen, wenn abhängig vom ersten Paket eines Datenstroms eine Regel zur Abarbeitung der nachfolgenden Pakete erfolgen soll. Trifft z.B. das zweite Datenpaket vor dem ersten ein und der Controller kann dieses nicht handeln, da es nicht das erste Paket des Datenstroms ist, so schlägt das Routing bereits hier fehl. [4][S. 282]

VI. Frenetic

Frenetic versucht sich nun durch ein höheres Abstraktionslevel, aufbauend auf NOX, den eben aufgezeigten Problemen anzunehmen. [4]
Ein hohes Abstraktionslevel verspricht meistens eine einfachere Implementierung von Anwendungen, doch wir möchten auch die technischen Hintergründe näher beleuchten.
Zu Beginn liegt der Fokus aber erst einmal auf den Möglichkeiten, die der deklarative und funktionale Ansatz bietet. Für den Programmierer stehen dazu die Network Query Language und die Network Policy Management Library bereit. [5], [4]

A. Network Query Language

Die Query Language von Frenetic (siehe Abbildung 9) erlaubt es dem Programmierer, den aktuellen Netzwerkzustand auszulesen.

Queries	q ::=	`Select`(a) *
		`Where`(fp) *
		`GroupBy`$([qh_1, \ldots, qh_n])$ *
		`SplitWhen`$([qh_1, \ldots, qh_n])$ *
		`Every`(n) *
		`Limit`(n)
Aggregates	a ::=	`packets` | `sizes` | `counts`
Headers	qh ::=	`inport` | `srcmac` | `dstmac` | `ethtype` |
		`vlan` | `srcip` | `dstip` | `protocol` |
		`srcport` | `dstport` | `switch`
Patterns	fp ::=	`true_fp`() | qh`_fp`(n) |
		`and_fp`$([fp_1, \ldots, fp_n])$ |
		`or_fp`$([fp_1, \ldots, fp_n])$ |
		`diff_fp`(fp_1, fp_2) | `not_fp`(fp)

Abbildung 9. Grammatik der Frenetic Network Query Language [4][S. 283]

Die SQL-ähnliche, deklarative Syntax erlaubt es, aggregierte Werte für Pakete, Paketgrößen als auch Paketzahlen für bestimmte Filter Patterns abzufragen. Diese werden auf die Paketheader gematcht, d.h. z.B. den Eingangsport, das verwendete Protokoll, die Ziel-IP-Adresse usw.
Eine boolsche Verknüpfung dieser Filter Patterns ist analog zu SQL ebenso möglich.
Über das GroupBy-Statement besteht die Möglichkeit, die gefilterten (aggregierten) Werte in kleinere Subsets mit beispielsweise gleicher Ziel-IP-Adresse zu unterteilen. Dies kann im Hauptprogramm die Analyse der Informationen deutlich vereinfachen. [4][S. 283]
Das SplitWhen-Statement ermöglicht ähnlich wie GroupBy eine Unterteilung in Subsets. Die Besonderheit besteht jedoch darin, dass nur aufeinanderfolgende, identische Werte in einem Set zusammengefasst werden. Für jeden Wechsel wird generell ein neues Set angelegt. [4][S. 283]
Bsp.: Folge von Ziel-IP-Adressen (dstip)

(1.1.1.1) (1.1.1.1) (2.2.2.2) (1.1.1.1)

Erzeugte Subsets für ...SplitWhen(dstip)...

1) (1.1.1.1) (1.1.1.1)
2) (2.2.2.2)
3) (1.1.1.1)

Durch Every können die Pakete schlussendlich noch in einem angegebenen Zeitintervall zusammengefasst werden und durch Limit die Anzahl der Pakete je Subset beschränkt werden. [4][S. 283]

Durch den deklarativen Ansatz beschreibt der Programmierer nur noch, welche Werte er auf welche Weise zusammengefasst haben möchte. Wie das darunterliegende System die Werte ermittelt, spielt dabei aus Entwickler-Sicht keine Rolle.
Dafür ist einzig und allein das Frenetic Run-Time System zuständig, welches im weiteren Verlauf genauer analysiert wird. [5][S. 5]

B. Network Policy Management Library

Die Network Policy Management Library (Auszug siehe Abbildung 10) ist eine sogenannte Combinator Library [8][S. 35-36].
Sie ist ein Teil funktionaler reaktiver Programmierung (FRP) [1] und verfolgt die Idee, durch Kombination bestehender Funktionen noch mächtigere Funktionen zu kreieren.
Funktionale reaktive Programmierung ist ein Paradigma, um eventgesteuerte Systeme deklarativ zu programmieren. [8], [1]
Der reine NOX Controller ist (siehe vorherige Kapitel) bereits event-gesteuert, doch Frenetic abstrahiert diese Ebene grob gesagt noch durch funktionale Programmierung, um das Hauptproblem überlappender Regeln aus Anwendersicht zu beheben. [5]
Der Grundgedanke der Architektur ist, dass das System jedes Paket zu jeder Zeit sieht, zumindest aus Programmierersicht. [5][S. 3]

Ein Auszug der Definitionen von Events und Listeners ist in Abbildung 10 zu sehen.

Events repräsentieren bestimmte eintreffende Zustände im Netzwerk, auf die im Programm reagiert werden kann. SwitchJoin trifft ein, sobald ein Switch dem Netzwerk beitritt und liefert den entsprechenden Switch zurück. Analog dazu existiert SwitchExit, sobald ein Switch aus dem Netzwerk ausscheidet.
Das PortChange-Event geht in dieser Hinsicht deutlich tiefer und reagiert auf jede Aktivierung, sowie Deaktivierung von

Switch-Ports. Hier wird der Switch mit einem Boolean für die Aktivierung und der entsprechenden Port-Nummer zurück geliefert. [4][S. 285-286]

Basic Event Functions überführen dagegen normale Funktionen in Eventfunktionen.
Dies bedeutet umgangssprachlich, dass die normale Funktion bei Lift z.B. auf alle Eingabeevents angewendet wird. Bei ApplyFst und ApplySnd würde dies nur für das erste, bzw. zweite Eingabeevent zutreffen. [4][S. 286]
Über die Filter-Funktion können Pakete nach bestimmten Patterns selektiert werden. [5][S. 3]

Events

$$\texttt{Seconds} \in \mathsf{int\ E}$$
$$\texttt{SwitchJoin} \in \mathsf{switch\ E}$$
$$\texttt{SwitchExit} \in \mathsf{switch\ E}$$
$$\texttt{PortChange} \in (\mathsf{switch} \times \mathsf{int} \times \mathsf{bool})\ \mathsf{E}$$
$$\texttt{Once} \in \alpha \to \alpha\ \mathsf{E}$$

Basic Event Functions

$$>> \in \alpha\ \mathsf{E} \to \alpha\ \beta\ \mathsf{EF} \to \beta\ \mathsf{E}$$
$$\texttt{Lift} \in (a \to \beta) \to \alpha\ \beta\ \mathsf{EF}$$
$$>> \in \alpha\ \beta\ \mathsf{EF} \to \beta\ \gamma\ \mathsf{EF} \to \alpha\ \gamma\ \mathsf{EF}$$
$$\texttt{ApplyFst} \in \alpha\ \beta\ \mathsf{EF} \to (\alpha \times \gamma)\ (\beta \times \gamma)\ \mathsf{EF}$$
$$\texttt{ApplySnd} \in \alpha\ \beta\ \mathsf{EF} \to (\gamma \times \alpha)\ (\gamma \times \beta)\ \mathsf{EF}$$
$$\texttt{Merge} \in (\alpha\ \mathsf{E} \times \beta\ \mathsf{E}) \to (\alpha\ \mathsf{option} \times \beta\ \mathsf{option})\ \mathsf{E}$$
$$\texttt{BlendLeft} \in \alpha \times \alpha\ \mathsf{E} \times \beta\ \mathsf{E} \to (\alpha \times \beta)\ \mathsf{E}$$
$$\texttt{BlendRight} \in \beta \times \alpha\ \mathsf{E} \times \beta\ \mathsf{E} \to (\alpha \times \beta)\ \mathsf{E}$$
$$\texttt{Accum} \in (\gamma \times (\alpha \times \gamma \to \gamma) \to \alpha\ \gamma\ \mathsf{EF}$$
$$\texttt{Filter} \in (\alpha \to \mathsf{bool}) \to \alpha\ \alpha\ \mathsf{EF}$$

Listeners

$$>> \in \alpha\ \mathsf{E} \to \alpha\ \mathsf{L} \to \mathsf{unit}$$
$$\texttt{Print} \in \alpha\ \mathsf{L}$$
$$\texttt{Register} \in \mathsf{policy\ L}$$
$$\texttt{Send} \in (\mathsf{switch} \times \mathsf{packet} \times \mathsf{action})\ \mathsf{L}$$

Rules and Policies

$$\texttt{Rule} \in \mathsf{pattern} \times \mathsf{action\ list} \to \mathsf{rule}$$
$$\texttt{MakeForwardRules} \in (\mathsf{switch} \times \mathsf{port} \times \mathsf{packet})\ \mathsf{policy\ EF}$$
$$\texttt{AddRules} \in \mathsf{policy\ policy\ EF}$$

Abbildung 10. Auszug der Frenetic Network Policy Management Library[4][S. 283]

Listeners sind dafür zuständig, Aktionen auf dem Netzwerk oder Logging durchzuführen. Die Print-Funktion schreibt beispielsweise nur den Input auf die Konsole, während Register die übergebenen Strategien in den Switches installiert. Send erhält ein Event, durch welches ein bestimmtes Paket an einen bestimmten Switch gesandt und dort dann die mitgeführte Aktion getätigt wird. [5][S. 285-286]

Rules and Policies stellen den letzten Unterpunkt dar. Über Rule können Regeln mit definierten Patterns und Aktionen erzeugt werden. Diese Regeln können auch automatisiert über MakeForwardRule erzeugt werden. Dabei wird eine Forwarding Regel für alle Pakete mit identischem Header zum übergebenen Paket erzeugt. Diese gilt für einen festgelegten Switch und dortigen Port.
AddRules merged die übergebenen Regeln und gibt die zusammengeführten zurück. [4][S. 285]

C. Beispielprogramm Frenetic

In diesem Kapitel wird ein einfaches Beispielprogramm in Frenetic zum Load Balancing betrachtet.
In Abbildung 11 ist ein Beispielprogramm abgebildet, welches jeder verschiedenen IP-Adresse einen unterschiedlichen Port über die Forwarding-Regeln zuweisen soll. Die Funktion

```
# query returning one packet per source IP
def src_ips() =
  return (Select(packets) *
          Where(inport_fp(1)) *
          GroupBy([srcip]) *
          Limit(1))

# helper to add switch to a port-packet pair
def add_switch(port,packet):
  return (switch(header(packet)),port,packet)

# parameterized load balancer
def balance(balancer):
  return \
    (src_ips()          >>#(IP*packet)E
     ApplyFst(balancer) >>#(port*packet)E
     Lift(add_switch)   >>#(switch*port*packet)E
     MakeForwardRules() >>#policyE
     AddRules())          #policyE
```

Abbildung 11. Einfacher Load Balancer mit Frenetic [4][S. 285]

src_ips() ermittelt über eine Frenetic Query eine Liste aller Pakete mit unterschiedlicher Quell-IP-Adresse. Ein Paket mit bestimmter IP-Adresse taucht in diesem Set dank Limit(1) nur einmal auf.
Die balance-Methode hat den Übergabeparameter balancer, welcher eine Event Function darstellt, die den zugehörigen Port einer IP-Adresse berechnen kann. [4][S. 286]
Folglich wird für jede Quell-IP-Adresse ein Port berechnet, und danach über bekannte Library-Funktionen die passende Regel erzeugt und im Switch gespeichert.

D. Architektur

Nach der Betrachtung und Analyse der Programmierkonstrukte, möchten wir uns die Architektur von Frenetic ansehen.
In Abbildung 12 sind die 3 Schichten der Frenetic Architektur zu sehen. Wie bereits erwähnt, baut Frenetic auf dem bekannten NOX Controller auf. Dieser arbeitet bereits event-gesteuert und kommuniziert direkt mit dem Frenetic Run-Time System. Dieses wiederum kommuniziert event-gesteuert mit dem Frenetic-Programm, dessen Sprachkonstrukte in den vorherigen Kapiteln beschrieben wurden.

Den Hauptteil der Architektur definiert offensichtlich das Run-Time System, welches die Schnittstelle zwischen dem Low-Level-Interface NOX und dem High-Level Frenetic-Programm ist. [4][S. 287]
Das Run-Time System führt in gewisser Weise Buch über das

Installieren und Löschen von Switch-Regeln und verwaltet dazu 3 globale Datenstrukturen: [4][S. 287]

- **policy**
  Ein Wörterbuch, in dem zu jedem Switch der Satz High-Level-Regeln gespeichert wird.
- **flows**
  Ein Set, welches alle aktuell installierten Low-Level-Regeln im Netzwerk speichert.
- **subscribers** Ein Set aus Tupeln mit je: Definition der Paket-Headers, Event, Byte-/Paket-Zähler, Liste offener Statistik-Anfragen

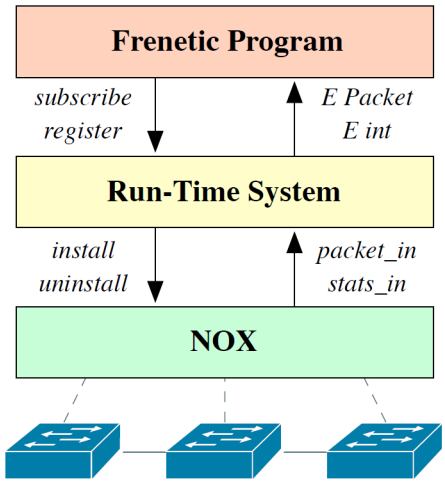

Abbildung 12. Architektur von Frenetic [4][S. 287], [5][S.5]

Zur Umsetzung der High-Level-Regeln in Low-Level-Switch-Regeln reagiert das Run-Time System auf den aktuellen Netzwerkverkehr.
Zu Beginn besteht die Annahme, dass alle FlowTables in den Switches leer sind, da noch keine Pakete im Netzwerk unterwegs sind. Folglich werden alle Pakete zuerst einmal über den packet-in-Handler ans Run-Time System weitergeleitet. Dort werden nacheinander folgende Berechnungen durchgeführt [4][S. 287-288]:

1) Suche in *subscribers* alle Einträge, auf die der Paket-Header matched
2) Verteile das Paket an alle ermittelten Subscribers
3) Ermittle aus *policy* alle Aktionen, die für diesen Paket-Header durchgeführt werden sollen.
4) Ausführung
   Sofern keine Subscribers existieren...
   Installiere Switch-Regeln für identische Pakete, damit der Controller in Zukunft nicht unnötig belastet wird.
   Sofern Subscribers existieren...
   Installiere keine Switch-Regeln, sondern leite das Paket direkt an den Switch mit den ermittelten Aktionen weiter.

Statistikanfragen werden vom Run-Time System grundsätzlich anders abgearbeitet. Für jeden Subscriber wird eine Warteschleife ausgeführt, bis der Timeout abläuft. Dann wird für alle passenden EInträge aus *flows* die Abfrage nach Byte- und Paket-Zähler der Switches gestartet.
Sobald die Werte über den stats_in Handler eintreffen, wird die Anfrage aus den offenen Anfragen des Subscribers entfernt. [4][S. 288]

Die Library der funktionalen reaktiven Programmierung ist nicht wie üblich mit einer pull-basierten Strategie [2], sondern über eine push-basierte Strategie [2] umgesetzt. [4][S. 288]
Pull-basierte Strategien arbeiten demand-driven, d.h. das Programm muss aktiv Daten anfordern. Push-basierte Strategien, wie z.B. Frenetic oder GUIs reagieren dagegen auf Datenströme, sobald sie auftreten. [2][S. 25]

E. Skalierbarkeit und Performance

Da die Frenetic Architektur auf NOX aufbaut und gleichzeitig ein deutlich höheres Abstraktionslevel erreicht, ist es interessant, mehrere Benchmarks von Frenetic und NOX direkt zu vergleichen.
Dabei stellt sich vorrangig die Frage, wie groß der Leistungsverlust durch das hohe Abstraktionslevel wirklich ist und wie gut sich Frenetic im Vergleich zu NOX skalieren lässt.

Der *All-Pais Connectivity* Benchmark aus Abbildung 13 zeigt die Datenmenge an, die je bei NOX und Frenetic über den Controller läuft, wenn alle Hosts im Netzwerk sich gegenseitig Ping-Pakete senden. Dies prüft ob die Kommunikation zwischen allen Netzwerkteilnehmern korrekt funktioniert. [4][S. 288]
Zu sehen ist, dass vor allem bei vielen Hosts, also großen Netzwerken (mehr als 25 Hosts), die Frenetic Architektur den Controller deutlich entlastet.

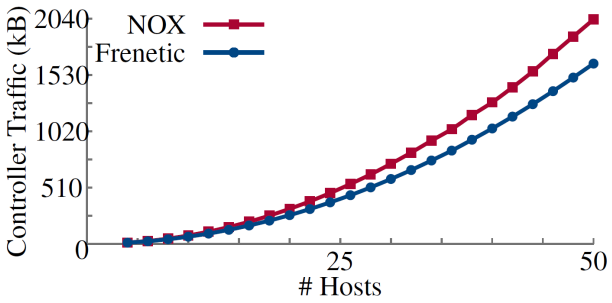

Abbildung 13. Frenetic vs. NOX: All Pairs Connectivity Benchmark [4][S. 289]

Der *Web Statistics* Benchmark aus Abbildung 14 zeigt auch wieder die Datenmenge an, die über den Controller läuft. In diesem Benchmark sendet jeder Host einen einzigen HTTP-Request ab und der Controller misst alle 5 Sekunden den aggregierten HTTP-Netzwerkverkehr. So können die einfachen Monitoring-Kapazitäten ermittelt werden. [4][S. 288]
Bei diesem Benchmark produzierte Frenetic marginal mehr Datenverkehr auf dem Controller, wenn die Anzahl der Hosts

nach oben skaliert wurde. Trotzdem muss gesagt werden, dass der Anstieg absolut konstant und nicht etwas sprunghaft wie bei NOX verläuft.

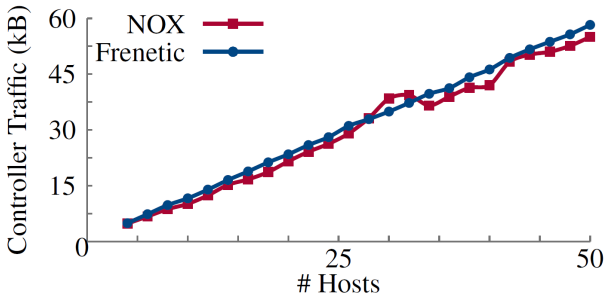

Abbildung 14. Frenetic vs. NOX: Web Statistics Benchmark [4][S. 289]

Der Heavy Hitters Benchmark aus Abbildung 15 stellt in gewisser Weise eine Erweiterung des All-Pais Connectivity Benchmarks aus Abbildung 13 dar. Der Unterschied liegt darin, dass je Host zufällig Pings an andere Hosts im Netzwerk gesendet werden. Der Controller berechnet dazu eine Statistik für jeden einzelnen Host und soll die Hosts mit den meisten gesendeten Paketen auflisten. [4][S. 288]
Auch hier ist wieder zu sehen, dass Frenetic weniger Datenverkehr produziert als NOX, wenn es nach oben skaliert wird.

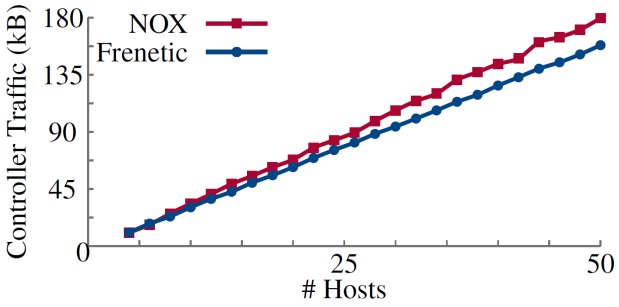

Abbildung 15. Frenetic vs. NOX: Heavy Hitters Benchmark [4][S. 289]

Zusammenfassend kann gesagt werden, dass Frenetic bezüglich der Skalierbarkeit definitiv im Schnitt besser ist als NOX.
Bezüglich der Performance wird behauptet, dass Frenetic-Programme nahezu die gleiche Leistung erreichen als NOX-Programme, jedoch existieren hierzu keine genauen Daten. [4][289]

VII. Maple

Eine weitere High-Level Programmiersprache für Software-defined Networks (SDN) ist Maple. Sie basiert auf dem Konzept der Algorithmic Policies, zu deutsch ungefähr algorithmische Strategien.
Maple verfolgt anders als Frenetik keinen deklarativen oder funktionalen Ansatz, sondern den klassischen Imperativen Programmieransatz. Daher kann in fast jeder bekannten Sprache wie z.B. Java oder Haskell programmiert werden. [14][S. 89]

A. Architektur

In Abbildung 16 ist die Architektur von Maple, aufgeteilt in das User und das System Level, zu sehen.
Ziel ist es, dass der Programmierer nur noch ein einziges Programm f definiert (User Level), welches für alle Pakete abgearbeitet wird. Ein Optimizer und ein Run-Time Scheduler kümmern sich um die Verwaltung der Low-Level-Regeln (System Level). [14][S. 89]

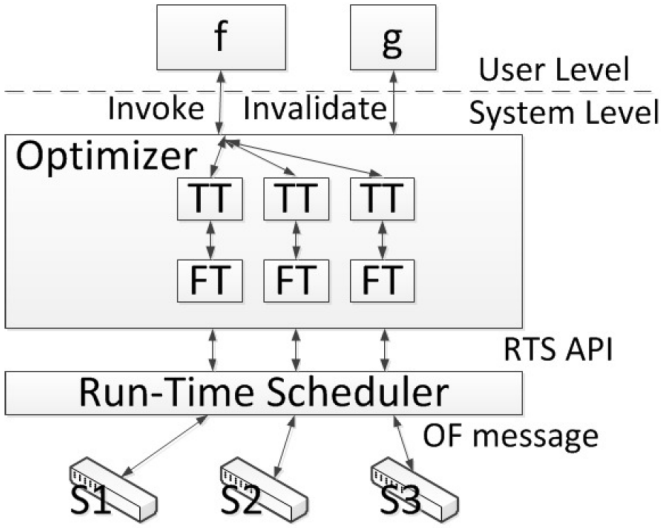

Abbildung 16. Maple Architektur [14][S. 89]
TT = Trace Tree, FT = Flow Table, S.. = Switches

Die Anbindung der Switches S1,S2,S3,... erfolgt parallel auf dem Run-Time Scheduler. Ebenso zwischen dem Run-Time Scheduler und dem Optimizer (RTS API).

B. Optimizer

Der Optimizer stellt die oberste Ebene des Systemlevels dar und gleichzeitig die elementarste. Für jedes Paket erneut das Programm f zu durchlaufen, würde spätestens bei der Skalierung des Netzwerks sehr ineffizient werden. Daher werden sogenannte *Traces* bisheriger Programmdurchläufe in Form von Entscheidungsbäumen gecached, um gleichartige Pakete schnell abarbeiten zu können.
Ebenso kann es möglich sein, dass Pakete, die nur die gleiche Ziel-IP-Adresse besitzen, trotzdem den gleichen Weg im Programm nehmen und damit auf die gleiche Weise behandelt werden könnten. [7][S. 48-49], [14][S. 89-90]
Die sogenannten Trace Trees (TT) (siehe Abbildung 16) werden erzeugt und erweitert, sobald die Algorithmic Policy f auf einem bestimmten Paket durchlaufen wurde. [14][S.90], [7][S. 48-51]
Trace Trees sind folglich immer eine aktuelle Momentaufnahme und repräsentieren nur die bislang durchlaufenen Teile des Programms. Ein Beispiel für solch einen Baum zeigt Abbildung 17.
Pakete mit TCP-Ziel-Port 22 werden zum aktuellen Zeitpunkt generell gedropped, ebenso diejenigen, mit Ethernet-Ziel-Port

2. Hat ein Paket nicht TCP-ZielPort 22, aber Ethernet-Ziel-Port 4 und kommt gleichzeitig von Ethernet-Port 6, so wird es an Port 30 im Switch weitergeleitet. [14][S. 93]

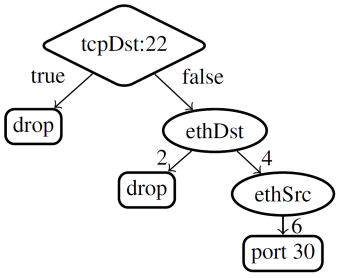

Abbildung 17. Beispiel: Trace Tree in Maple [14][S. 93]

Die Berechnung eines Trace Trees anhand einer Flow Table funktioniert genauso effizient wie die Berechnung einer Flow Table anhand eines Trace Trees. Sobald keine passende Regel in der Flow Table gefunden wird, wird für das entsprechende Paket das Programm im User Level durchlaufen, danach der Trace Tree(s) aktualisiert und daraus wiederum die Flow Table(s) angepasst. [14][S. 90-92]
Maple beinhaltet einige Algorithmen zur Optimierung der erzeugten Trace Trees. So können beispielsweise nicht mehr erreichbare Teilzweige entschlankt oder andere zusammengefasst werden, wenn sie abschnittsweise das gleiche Ergebnis liefern. Daraus folgt, dass auch ständig die Flow Tables automatisch optimiert werden.
Aufgrund der benötigten hohen Rechenleistung des Controllers, gerade für hochskalierte Netzwerke, sieht die Architektur explizit Multi-Prozessor-Server vor. [14][S. 92-95]

C. Skalierbarkeit und Performance

Gleich wie im Falle von Frenetic interessiert uns auch bei Maple der Vorteil der Technologie. Der Fokus liegt dabei vor allem auf der Skalierbarkeit und der Performance.
Die hohe Qualität der Optimierungsalgorithmen für die Flow Table Einträge ist in Abbildung 18 zu sehen.

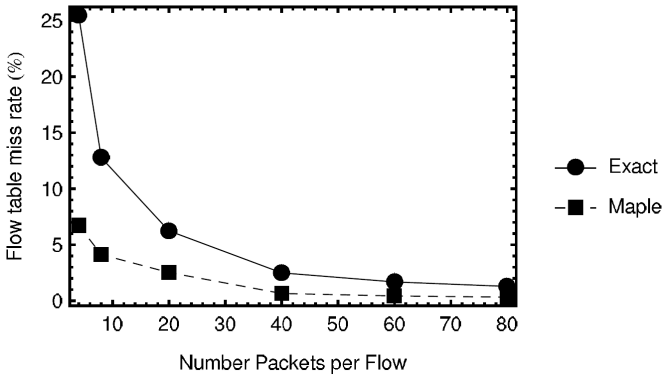

Abbildung 18. Benchmark: Flow Table miss rate für Maple [14][S. 97]

Auf der y-Achse ist dabei der Prozentsatz an Paketen, welche nicht im Switch weitergeleitet werden können, sondern an den Controller übermittelt werden, aufgezeigt. Maple liegt gegenüber dem Controller, welcher die Pakete exakt auf die Flow Table matched deutlich vorne. Maple erreicht dies durch optimierte Regeln, welche nur noch auf die Quell- und Ziel-IP-Adresse gemappt werden. [14][S. 96]
Ein weiterer Benchmark ist in Abbildung 19 dargestellt. Hierfür wurden die Ende-zu-Ende-Zeiten von HTTP-Datenverkehr zwischen den einzelnen Hosts ermittelt. Dadurch kann ein Eindruck der Performance gewonnen werden. [14][S. 96-97]

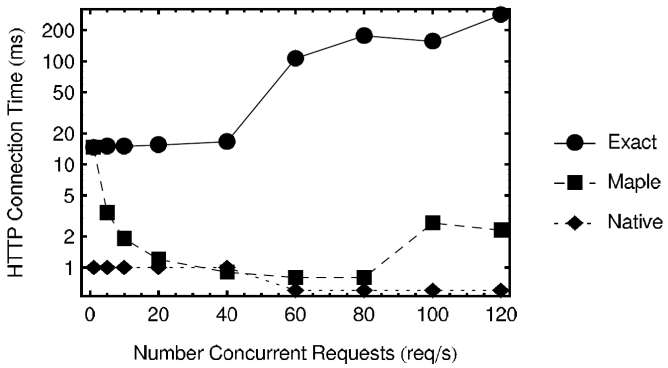

Abbildung 19. Benchmark: HTTP Connection Time für Maple [14][S. 97]

Mit steigender Datenrate sinkt die Verbindungszeit für Maple auf rund 1-2 ms, während sie beim exakten Matching ein Vielfaches darüber liegt. Der Anstieg ab etwa 80 Anfragen pro Sekunde wird mit der Auslastung der Switches begründet. Dennoch fällt das Fazit in diesem Fall phänomenal gut für Maple aus. [14][S. 97]

Am interessantesten ist jedoch der Performance-Vergleich von Maple im Gegensatz zu Frenetic oder NOX.
Bezüglich der Skalierbarkeit mit Steigerung der Prozessorkerne kann Maple in Abbildung 20 den vollen Trumpf gegenüber NOX ausspielen. Wie im Frenetic-Kapitel bereits gesehen, liegt Frenetic in etwa auf Höhe von NOX bezüglich Durchsatz und Performance.

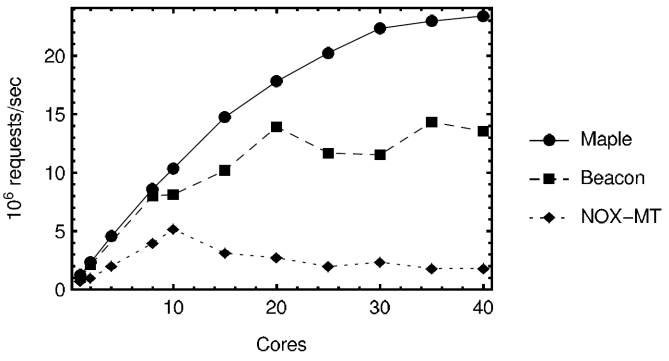

Abbildung 20. Vergleich des Datendurchsatzes [14][S. 98]

NOX und damit auch Frenetic skalieren nur minimal oder garnicht durch die Spendierung mehrerer Prozessoren für

den Controller. Hier kann Maple einen vielfach höheren Durchsatz durch die strikte Parallelarchitektur erzielen.
Dieselbe Beobachtung kann bezüglich der mittleren Antwortzeiten in Abbildung 21 gemacht werden.

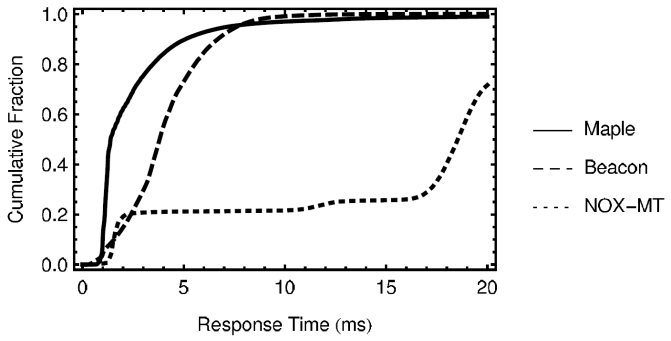

Abbildung 21. Vergleich der mittleren Antwortzeiten [14][S. 98]

Währen Maple im Median bei etwa 1 ms liegt, benötigt NOX und damit auch Frenetic im Median schon etwa 17 ms. Damit ist Maple in diesem Benchmark etwa um den Faktor 17 schneller also NOX oder Frenetic. [14][S. 98]

VIII. Fazit und Zusammenfassung

Zusammenfassend kann gesagt werden, dass Frenetic und Maple auf ganz unterschiedlichen Architekturen und Philosophien basieren.
Während Frenetic auf dem älteren NOX Controller aufbaut und eine Abstraktionsebene für diesen mittels funktionaler Programmierung darstellt, verfolgt Maple den iterativen Ansatz. Maple hat gefühlt das noch höhere Abstraktionslevel, da Regeln aus dem Programmdurchlauf eines iterativen Programms erzeugt werden. Dazu kommt nocht die Parallelarchitektur, welche sich extrem gut skalieren lässt.
Positiv zu nennen ist aber auch, dass Frenetic in etwa gleiche Benchmarkwerte erzielt wie der darunter liegende NOX-Controller. Somit schluckt die Abstraktionsebene für den Benutzer nahezu kaum Systemleistung. Trotzdem fällt es vermutlich vielen iterativen Programmierern schwer, gedanklich dem funktionalen Ansatz zu folgen, der prinzipiell gegen die klassische Softwareentwicklung strebt.
Sobald als Controller ein potenter Server mit vielen Prozessoren zur Verfügung steht und das Netzwerk schnell ziemlich groß wird, für kein Weg mehr an Maple vorbei. Bei Multiprozessorsystemen hat Frenetic oder NOX bezüglich Performance und Skalierbarkeit kaum noch eine Chance. Dazu kommt noch die vermeintliche Hürde der funktionalen Programmierung, was diese Techniken im Allgemeinen uninteressant machen dürfte.

Abbildungsverzeichnis

Literatur

[1] Katrin Ebert. Funktionale reaktive programmierung. *http://www.tu-cottbus.de/fakultaet1/de/programmiersprachen-compilerbau/lehrstuhl/mitarbeiter/ehemalige-mitarbeiter/wolfgang-jeltsch/funktionale-reaktive-programmierung.html (aufgerufen am 09.02.2016)*, December 2011.

[2] Conal Elliott. Push-pull functional reactive programming. *ACM 978-1-60558-508-6/09/09*, September 2009.

[3] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, David Walker, and Major Robert Harrison. Languages for software-defined networks. *IEEE Communications Magazine*, February 2013.

[4] Nate Fostery, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM 978-1-4503-0865-6/11/09*, September 2011.

[5] Nate Fostery, Rob Harrison, Matthew L. Meola, Michael J. Freedman, Jennifer Rexford, and David Walker. Frenetic: A high-level language for openflow networks. *ACM 978-1-4503-0467-2/10/11*, November 2010.

[6] 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), July 2008.

[7] Andreas Richard Voellmy (Dissertation Directors: Paul Hudak and Y. Richard Yang). Programmable and scalable software-defined networking controllers. *A Dissertation Presented to the Faculty of the Graduate School of Yale University in Candidacy for the Degree of Doctor of Philosophy*, May 2014.

[8] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, CA*, April 2007.

[9] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, February 2013.

[10] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Guru Parulkar, Jennifer Rexford, Jennifer Rexford, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2), April 2008.

[11] Sakir Sezer, Sandra Scott-Hayward, Pushpinder Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Marc Miller, and Navneet Rao. Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, July 2013.

[12] Myung-Ki Shin, Ki-Hyuk Nam, Miyoung Kang, and Jin-Young Choi. Formal specification and programming for sdn. *Proposed SDN RG Meeting at IETF 84 (Vancouver, BC, Canada)*, August 2012.

[13] Josias Montag (Betreuer: Daniel Raumer und Florian Wohlfart). Software defined networking mit openflow. *Hauptseminar: Innovative Internettechnologien und Mobilkommunikation, WS 2012/2013, Lehrstuhl Netzarchitekturen und Netzdienste - Fakultät für Informatik, Technische Universität München*, February 2013.

[14] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. *ACM 978-1-4503-2056-6/13/08*, August 2013.