Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2419

Consistent Transformations of Content-based Routing Networks in OpenFlow

Patrick Bosch

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Kurt Rothermel
Betreuer/in:	Dr. Boris Koldehofe
Beginn am:	2013-02-14
Beendet am:	2013-08-14
CR-Nummer:	C.2

Kurzfassung

Content-based Publish/Subscribe-Systeme sind weitverbreitet. Sie bieten die Möglichkeit, Nachrichten schnell und einfach an mehrere Teilnehmer zu verteilen ohne, dass der Sender von den Teilnehmern weiß. Diese Systeme haben allerdings den Nachteil, dass sie auf der Anwendungsebene implementiert sind und damit ein gewisses Maß an Overhead mitnehmen. Mit dem Aufkommen von Software-defined networking hat man nun die Möglichkeit, solche Systeme auf der Netzwerkebene zu implementieren und muss den Overhead der Anwendungsebene nicht mitnehmen. Die Idee zu einem solchen System wurde vorgestellt, allerdings benötigt es dafür unterschiedlichste Komponenten. Eine dieser Komponenten soll in dieser Arbeit vorgestellt werden.

Dabei handelt es sich um einen Update-Mechanismus, der die Veränderungen im Netzwerk ohne Nachrichtenverlust und ohne Duplikate umsetzen kann. Veränderungen kommen durch z. B. neue Subscriber oder veränderte Subscriptions. Dies spiegelt sich in einer Veränderungen des Graphen wieder, der das Netzwerk darstellt. Der Update-Mechanismus erkennt Veränderungen zwischen zwei Graphen und sorgt dafür, dass am Ende der aktuelle Graph im Netzwerk implementiert ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

1. Einleitung

Publish/Subscribe-Systeme sind sehr verbreitet. Sie bieten die Möglichkeit schnell und einfach viele zu erreichen. Die Nachricht muss vom Publisher nur einmal erzeugt werden und danach muss er sich nicht mehr darum kümmern, dass diese Nachricht die Subscriber, die sich für die Nachricht registriert haben, erreicht. Dies übernimmt dann das zu Grunde liegende System. Damit erreicht man eine lose Kopplung zwischen Publishern und Subscribern. Diese haben keine Information darüber von wem sie die Informationen erhalten oder an wen sie die Informationen liefern. Das unterliegende System nennt man ein Brokernetz und dieses ist in der Regel auf der Anwendungsebene realisiert. Das bringt den Vorteil, dass man damit leichter programmieren kann, aber auch den Nachteil, dass man den Overhead der Anwendungsebene mitnimmt. Für hochskalierbare und hochperformante Publish/Subscribe-Systeme kann dieser Overhead problematisch werden und man würde ihn gerne vermeiden. Es bietet sich daher an die Verteilung der Nachrichten nicht mehr über ein Brokernetz zu lösen, sondern direkt auf der Netzwerkebene zu implementieren. Dafür benötigt man aber direkten Zugriff auf diese Ebene und muss das Netzwerk von einer zentralen Stelle aus dirigieren. Mit dem in letzter Zeit aufkommenden Ansatz von Software-defined networking (SDN) ist dies möglich. Es bietet die Möglichkeit einer Abstraktionsebene um die Hardware zu konfigurieren und somit den Fluss der Daten selbst zu bestimmen.

Software-defined networking (SDN) bekam in letzter Zeit immer mehr Aufmerksamkeit. Spätestens seit Google bekannt gab, dass sie in ihren Rechenzentren SDN und OpenFlow nutzen war der öffentliche Durchbruch da. SDN bietet sehr viele Möglichkeiten die Bandbreite in einem Netzwerk möglichst effizient auszunutzen. Durch eine zentrale Einheit, den Controller, der einen globalen Überblick über das Netzwerk hat, kann man den Fluss im Netzwerk so steuern, dass das Netzwerk an sich effizient ausgenutzt ist. Virtualisierung von Netzwerken und ähnliche Vorhaben sind damit auch möglich.

SDN ist der Ansatz um das Netzwerk zentral zu konfigurieren. Dazu ist aber ein Protokoll nötig, das die nötigen Nachrichten übermittelt und damit dann die Hardware konfiguriert. Das Protokoll, welches dies bietet ist OpenFlow. Dieses wird kontinuierlich erweitert und liegt derzeit in Version 1.3.2 vor. Mit diesem Protokoll kann man von einer zentralen Stelle aus die Switches konfigurieren und Informationen von ihnen abrufen. Dies bietet eine große Anzahl an Möglichkeiten, die sich damit realisieren lassen. Man kann damit den kompletten Traffic in einem Netzwerk spezifisch leiten. Dabei kann man verschiedenste Garantien unterstützen. Man muss letztendlich nur festlegen auf welchem Switch welche Pakete wohin weitergeleitet werden. Viele Hardwarehersteller bieten mittlerweile OpenFlow-fähige Switches an womit der Umsetzung von solchen Systemen nur noch wenig entgegensteht.

Mit diesem Ansatz sind allerdings auch Schwierigkeiten verbunden. Es stellt sich zum Beispiel die Frage nach der Skalierbarkeit dieser Netzwerke und dem damit verbundenen Controller. Ein einzelner zentraler Controller ist irgendwann mit der Größe des Netzwerks überlastet bzw. die Verzögerungen zu verschiedenen Switches ist zu unterschiedlich, als dass sich damit noch sinnvoll arbeiten lässt. Andere Probleme werden z. B. in [CMT+11] behandelt. Es gibt allerdings eine große Anzahl an Forschungsarbeiten die sich mit den Schwierigkeiten beschäftigen und diese auch lösen. Gleichzeitig werden aber auch neue Möglichkeiten untersucht. Durch die kontinuierliche Erweiterung des OpenFlow-Protokolls und die damit verbundenen Neuerungen lässt sich immer mehr realisieren. Es werden die unterschiedlichsten Bereiche untersucht. Hierbei handelt sich um sehr allgemeine Themenbereiche als auch spezialisierte. Ein Beispiel hier wären sicherheitsrelevante Themen die sich abdecken lassen [RFRW11]. Es gibt aber auch allgemeine Probleme, zum Beispiel die Platzierung des Controllers, damit die Verzögerung möglichst gleichmäßig ist [HSM12]. Größere Gebiete sind die Virtualisierung von Netzwerken, aber auch Themen wie Publish/Subscribe. Prinzipiell lässt sich wohl ein Großteil schon bekannter Probleme und Lösungen auf SDN und OpenFlow übertragen. In vielen Fällen sind die Lösungen damit sogar einfacher, da man in der Regel eine globale Sicht annehmen kann, die der Controller besitzt. Viele verteile Probleme lassen sich damit zentralisiert lösen.

In dieser Arbeit geht es nun um die dynamische Rekonfiguration eines Pub-/Sub-Systems wie es in [KDTR12] vorgestellt wird. Dabei handelt es sich um ein Content-based Publish/-Subscribe System. Bei dem Problem, das sich ergibt, geht es darum, einen Spannbaum bzw. einen Graphen in einen anderen zu überführen. Dabei dürfen natürlich keine Nachrichten verloren gehen und sollten auch keine Duplikate entstehen. Nach Möglichkeit sollte auch der Controller nicht zu sehr mit Traffic belastet werden, da dieser nicht die Routingkapazitäten eines Switches besitzt. Ein Protokoll, das den Controller stark belastet wurde in [McG12] vorgestellt. Es wird ein Protokoll vorgestellt, dass diese Anforderungen erfüllt und den Controller bei einer Rekonfiguration nicht belastet.

Gliederung

Zu Beginn werden in Kapitel 2 einige grundsätzliche Informationen über Content-based Publish/Subscribe, OpenFlow und SDN allgemein vorgestellt werden. Es soll einen Einblick geben, was genau Publish/Subscribe ist, wie genau SDN funktioniert und was man zusammen mit OpenFlow damit alles machen kann. Zusätzlich wird noch das System Setup vorgestellt. Mit diesem Wissen wird dann in Kapitel 3 die Problemstellung und ihre Schwierigkeiten erklärt und verdeutlicht werden. Gleichzeitig werden noch mehrere verwandte Arbeiten vorgestellt, die sich auch schon in die Richtung von Updatemechanismen und damit verwandten Themen beschäftigt haben. Hier gibt es schon einige Ansätze, die aber alle gewisse Nachteile mit sich bringen bzw. einen Trade-off darstellen. Allerdings mit negativen Punkten in Bereichen, die in unserem System sehr unpraktisch sind. In Kapitel 3 wird zusätzlich noch auf die Umgebung eingegangen, in der sich diese Arbeit und ihre Implementierung befinden.

Im nächsten Kapitel, Kapitel 4 wird das Protokoll mit allen dazugehörigen Information vorgestellt. Die einzelnen Teile des Protokolls, sowie das Zusammenspiel von ihnen, werden erläutert.

In Kapitel 5 folgt die Evaluierung der Implementierung. Es wurden verschiedene Tests durchgeführt und die Resultate werden mit Erwartungen verglichen, die man an den Mechanismus gestellt hat. Dabei wird geschaut wie gut sich der jeweilige Mechanismus schlägt und wo man noch etwas verbessern kann. Zudem werden die Mechanismen miteinander verglichen was ihre Ergebnisse betrifft.

2. Grundlagen

In diesem Kapitel wollen wir uns verschiedene Grundlagen anschauen. Dabei wird das erste Thema Publish/Subscribe bzw. Content-based Publish/Subscribe sein. Hier wollen wir uns den grundlegenden Aufbau eines solchen Systems anschauen. Als nächstes kommt ein Überblick über Software-defined networking, hier wird erklärt wie genau es funktioniert und was es ermöglicht. Um SDN nutzen zu können benötigen wir ein Protokoll, OpenFlow. Dieses wird als nächstes vorgestellt. Danach folgen das System Setup und die Grundlagen von Graphentransformation, auf denen der Mechanismus nachher aufbaut.

2.1. Content-based Publish/Subscribe

Publish/Subscribe ist ein Ansatz, bei dem es darum geht, ein Event von einem Publisher an alle registrierten Subscriber zu schicken. Dabei weiß weder der Publisher an wen das Event geht noch wissen die Subscriber von wem das Event kommt. Es unterstützt damit eine lose Kopplung.

Publisher sind Event- oder Datengeneratoren. Sie veröffentlichen Nachrichten, die zum Beispiel Neuigkeiten enthalten. Subscriber hingegen kann man als Konsumenten beschreiben. Sie sind an gewissen Nachrichten interessiert und konsumieren diese. Die Kopplung dieser verschiedenen Teilnehmer wird dadurch erzielt, dass die Nachrichten der Publisher erst einmal an ein Brokernetz weitergeleitet werden. In diesem System ist auch vermerkt, welcher Subscriber an welchen Nachrichten interessiert ist. Damit hat man zwar Publisher und Subscriber voneinander abgekoppelt, aber die Hauptlast liegt dann auf dem Brokernetz. Dieses muss hochperformant und gut skalierbar sein, ansonsten bricht es schnell zusammen.

Es gibt nun zwei große Unterscheidungen von Publish/Subscribe-Systemen. Zum einen Topic-based und zum anderen Content-based. Beim Topic-based Ansatz werden verschiedene Topics festgelegt, in die Nachrichten einsortiert werden. Dabei kann man diese flach oder auch hierarchisch anordnen. Subscriber registrieren sich dann für verschiedene Topics und bekommen alle Nachrichten, die darunter fallen. Bei der hierarchischen Anordnung heißt das, dass man auch alle Subtopics erhält. Durch den Aufbau in verschiedene Topics hat man allerdings das Problem, dass die Aufteilung der Nachrichten nicht unbedingt sehr feingranular ist. Wenn ein Subscriber zum Beispiel nur an steigenden Preisen eines Produktes interessiert ist, er aber nur die Preise ohne Information, ob sie gestiegen oder gefallen sind bekommt, dann fehlt ihm Information oder er hat die falsche Information. Da wäre es praktischer, wenn die Granularität höher wäre und man seine Subscriptions sehr genau

fassen kann damit man wirklich nur die Informationen bekommt, die man als Subscriber haben will.

Dieser Ansatz wird durch Content-based Publish/Subscribe angegangen. Hierbei werden Nachrichten nach ihrem Inhalt weitergeleitet und Subscriber können sich für sehr speziellen Inhalt registrieren. Das bringt den Vorteil, dass Subscriber nur die Nachrichten erhalten, die sie wirklich wollen. Allerdings bedeutet mehr Granularität auch mehr Last für das Brokernetz. Es muss den Inhalt analysieren und dann die passenden Subscriber auswählen. Beim Topic-based Ansatz genügte ein Topic für die Nachricht, welches in der Regel der Publisher auswählt, und für jedes Topic dann eine Liste an Subscribern, an die die Nachricht weitergeleitet wird.

Um die Leistung solcher Systeme nun noch weiter zu steigern wäre es also wünschenswert auf das Brokernetz verzichten zu können. Damit das aber realisierbar ist, muss etwas anderes die Aufgabe des Netzes, die Nachrichten zu verteilen, übernehmen. Das Ziel ist es, dies in das Netzwerk selber zu integrieren. Mit SDN und einer Art Kodierung der Nachrichten, um den Inhalt darzustellen, ist dies möglich. Allerdings benötigt es dafür verschiedenste Mechanismen, wobei einer hier in dieser Arbeit vorgestellt wird.

2.2. Software-defined networking

SDN-Netzwerke stehen im Kontrast zu herkömmlichen Netzwerken. Aus diesem Grund sollten wir vor der Erklärung, was SDN genau ist, erst einmal die Merkmale herkömmlicher Netzwerke untersuchen. Denn es gibt zwischen den beiden Architekturen einen klaren Unterschied, der heraus sticht. Dieser ist die Berechnung der Forwardingtabellen oder allgemein die Berechnung, welcher Input-Port mit welchem Output-Port verbunden werden soll. Die Berechnung der Forwardingtabellen in herkömmlichen Architekturen ist prinzipiell dezentral. Jeder Router berechnet für sich die kürzesten Wege für die jeweiligen Ziele. Dabei erhält jeder Router Informationen von seinen Nachbarn und verteilt seine Informationen an seine Nachbarn. Wie genau dieser Vorgang und auch die Berechnung der Forwardingtabellen stattfinden, ist vom jeweiligen Routingprotokoll abhängig. Bekannte Protokolle sind zum Beispiel Distance Vector Routing, Link State Routing oder Backward Learning. Jedes dieser Protokolle hat seine Schwächen und Stärken. Sie funktionieren aber zuverlässig und sind auch sehr gut skalierbar. Der Punkt hier ist aber, dass jeder Router für sich seine Forwardingtabellen berechnet. Damit sind die Routen nicht unbedingt optimal für bestimmte Anwendungszwecke, da die globale Sicht fehlt.

SDN geht nun einen anderen Weg. Die Router selber sind nur noch dafür zuständig Pakete weiterzuleiten. Die Berechnung, welches Paket wie weitergeleitet wird, also die Berechnung der Forwardingtabellen, findet ausschließlich in einem Controller, also zentral, statt. Das bringt Vorteile wie auch Nachteile mit sich. Ein Vorteil ist sicher, dass der Controller einen kompletten Überblick über das gesamte Netzwerk besitzt und daher optimale Routen berechnen kann. Das ist in einem dezentralen Netzwerk nur mit relativ viel Aufwand realisierbar. Oder auch Routen, die zwar nicht optimal sind, aber anderen Ansprüchen genügen, wie etwa, dass gewisse Pakete über spezielle Router weitergeleitet werden, sind

möglich. Die Verwaltung der Systeme und Änderung in den Systemen werden einfacher gestaltet, da man sie auf einer höheren Ebene vollziehen kann [RFR+12]. Dies bietet die Möglichkeit einer einfacheren Programmierung der Systeme aber auch der Programme, die das System nutzen. Nachteile könnten bei der Skalierbarkeit entstehen. Je nachdem wie gut sich der Controller skalieren lässt oder auch nicht. Allerdings scheint dies nicht allzu gravierend zu sein bzw. sogar durchaus handhabbar [VW12]. Möglicherweise ist Skalierbarkeit auch mit der Einführung von parallelen und verteilten Controllern erreichbar. Hier kommt jedoch das Problem der Synchronisierung zwischen den Controllern auf. Aber da SDNs erst am Anfang stehen, wird sich hier in Zukunft noch viel ergeben.

Im Grunde bedeutet das nun, dass man ein Netzwerk hat, das zentral gesteuert wird, mit den möglichen Nachteilen eines zentralen Controllers. Aber dieser Aufbau ermöglicht auch, das Netzwerk nach Belieben zu konfigurieren, durchaus auch auf eine Art und Weise, die in einem dezentralen System nur schwer möglich ist. Man hat durch den zentralen Controller einen starken Einfluss auf das System und kann es so ganz nach seinen Wünschen anpassen. Das ist für einige Programme ein sehr wichtiger Punkt. Die Koordinierung von den einzelnen Komponenten kann durch den Controller sehr gezielt erfolgen. Das ist gerade bei dem Mechanismus, der in dieser Arbeit behandelt, wird sehr wichtig. Hier wird SDN voll ausgenutzt und die Zentralisierung des Controllers ist ein starker Vorteil.

2.3. OpenFlow

Zum Konzept von SDN fehlt jetzt noch ein Protokoll, dass diese Architektur umsetzt, bzw. die Einstellungen und Konfigurationen vornehmen kann, die dafür nötig sind. OpenFlow ist dieses Protokoll. Es definiert was alles möglich ist und wie es umgesetzt wird. Es deckt die ganze Kommunikation zwischen den Einheiten im Netzwerk, also Controller und Switch ab, das heißt die Nachrichten die ausgetauscht werden und welche Befehle möglich sind. Auch welche Information Switches speichern und wie der Controller diese dann abrufen kann werden spezifiziert.

Aufgebaut ist das ganze so, dass man für jedes Paket, bzw. jede Gruppe von Paketen, zum Beispiel eine TCP-Verbindung, einen sogenannten Flow kreiert. Durch das erste Paket wird der Flow festgelegt und alle weiteren Pakete der Verbindung nehmen den gleichen Flow. Man kann aber auch schon Flows statisch festlegen, bevor überhaupt ein Paket ankam. Damit lassen sich feste Wege erstellen, die zum Beispiel in einem Pub-/Sub-System notwendig sind. Diese Flows kann man kreieren, löschen und verändern. Über diese Flows kann man Informationen einholen, denn jeder Switch speichert für einen Flow einen Cookie in dem die Anzahl der Pakete, die den Flow genutzt haben stehen und auch wie lange der Flow schon existiert. Wie die einzelnen Nachrichten aussehen, die genauen Mechanismen funktionieren, wie man Informationen die existieren abrufen kann oder was genau alles möglich ist wird in der OpenFlow Spezifikation dargelegt die derzeit in Version 1.3.2 vorhanden ist und weiterentwickelt wird [Fou13].

Die Mechanismen sind teilweise sehr weitreichend und lassen einigen Spielraum für Möglichkeiten, die teilweise auch nicht ganz konventionell sind. Mit jeder weiteren Spezifikation

kommen Neuerungen hinzu die noch mehr ermöglichen. Allerdings hängen die Switchhersteller teilweise deutlich hinter den Spezifikationen hinterher. Man muss also aufpassen, welche Spezifikation man zu Grunde legt. Gerade die neusten Neuerungen werden nicht unbedingt unterstützt. Das kann zu Komplikationen führen, wenn man sich auf die Spezifikation verlässt. Dadurch, dass sich OpenFlow relativ schnell entwickelt, ist es wichtig zu wissen, welche Funktionen von Switches unterstützt werden und zur Not ältere Funktionen zu verwenden um eine Kompatibilität zu erhalten.

In dieser Arbeit wird Floodlight als Controller verwendet. Dieser basiert auf dem OpenFlow 1.0 Standard und bietet sehr viele Möglichkeiten zur Erweiterung. Aufgebaut ist er in Modulen und bietet sowohl eine Java- als auch eine REST-API die es ermöglichen die Module zu nutzen. Das Ganze ist ein Open-Source-Projekt und befindet sich im kontinuierlichem Wandel.

2.4. System Setup

Die Umgebung in der sich diese Arbeit befindet wird in [KDTR12] vorgestellt. Dabei soll die Methode mit Filtern verwendet werden. Filter sind eine Art Kodierung des Inhalts. Es soll ein Teil der IP-Adressen verwendet werden um die Kodierung zu übertragen. Zu dem Filter gehört dann der Inhalt. Weitergeleitet wird aber nur durch den Filter. Die Filter setzen muss der Publisher. Dies geschieht über eine Abstraktion bzw. eine Aufteilung. Für den Inhalt existiert eine Anzahl an Attributen, hier d genannt. Mit diesen Attributen kann man einen d-dimensionalen Raum aufbauen. Diesen Raum kann man in Unterräume unterteilen. Diese Unterräume kann man als binären String darstellen. Sie bieten allerdings nur eine Annäherung an die reale Unterteilung der Nachrichten. Der Binärstring, der dz-Ausdruck genannt wird, kodiert diese Events nun. Diesen Binärstring kann man 1:1 auf IP-Adressen legen und damit IP-Adressen nutzen um die Filter zu implementieren. Geplant sind IPs des IPv6-Protokolls, allerdings unterstützt Floodlight IPv6 noch nicht, womit vorerst IPv4 verwendet wird.

Mit Hilfe von OpenFlow und der damit verbundenen Abstraktion soll eine Virtualisierung geschaffen werden, mit der man das Publish/Subscribe-System in jedem beliebigem Netzwerk installieren und realisieren kann. Dabei kann man sich den Aufbau als verschiedene Pfade von Publishern zu Subscribern vorstellen. Dieser Pfad bedeutet, dass der Subscriber prinzipiell an diesem speziellen Publisher interessiert ist. Die Filter, die auf diesem Pfad angebracht sind, die die Kante bezeichnen und identifizieren, geben das genaue Interesse des Subscribers an. Ereignisse von einem Publisher werden dann gemäß dieser Filter weitergeleitet und alle interessierten Subscriber erhalten die Ereignisse, für die sie sich registriert haben. Aus diesen Pfaden baut sich dann ein Graph auf, der alle Verbindungen repräsentiert.

Dieser Graph wird als eine Art minimaler Spannbaum aufgebaut. Die Anzahl der Kanten wird versucht gering zu halten, indem man verschiedene Pfade vereint und erst nah an den Subscribern die Kanten durch Filter aufteilt. Dadurch lässt sich die Bandbreite senken, da ein geringerer Teil an Nachrichten gesendet werden muss. Die Kanten und ihre Filter lassen sich nahezu 1:1 auf OpenFlow Flows mappen. Daher liegt es nahe, genau dieses Protokoll bei der

Implementierung zu verwenden. Mit den Flows hat der Controller auch die Möglichkeit, wie in den Grundlagen unter OpenFlow erwähnt, Informationen über die Flows abzurufen und diese in weitere Berechnungen über den Aufbau des Graphen mit einfließen zu lassen.

2.5. Graphtransformation

Der Aufbau oder die Repräsentation des Systems, auf das sich der Update-Mechanismus beziehen soll, ist ein Graph. Genauer gesagt zwei Graphen. Dabei soll der eine in den anderen überführt werden. Basierend ist das Ganze auf Graphtransformationen. Prinzipiell geht es dabei darum, einen sogenannten „Mutter"-Graphen in einen „Tochter"-Graphen gemäß einer Regel zu überführen. Zudem gibt es noch einen einbettenden Mechanismus, der das ganze abschließt. Das Ganze nennt sich dann Produktion und wird als ein Triple (M, D, E) dargestellt. Diese Produktion wird dabei auf einen Host-Graphen angewandt. Dabei wird der Teil M aus dem Host-Graphen entfernt und mit dem Teil D ersetzt. D wird dann noch durch den Mechanismus E in den vom Host-Graphen übrig gebliebenen Graphen eingebettet [Roz97]. Oder anders dargestellt, mehr in Form einer Grammatik, gibt es eine linke Seite, die mit einer Produktionsregel p in eine rechte Seite überführt wird, $L \overset{p}{\rightarrow} R$. Dabei ist die linke Seite der Ausgangsgraph und die rechte Seite der Zielgraph.

Für die Graphtransformation gibt es nun zwei Ansätze, einmal den algorithmischen und einmal den algebraischen. Beim algorithmischen besteht M nur aus einem Knoten. Dieser Knoten wird bei der Produktion durch einen Graphen ersetzt. Beim algebraischen werden dafür Kanten umgeschrieben, aber keine Knoten. Beide Ansätze bieten die gleichen Möglichkeiten, kommen aber aus unterschiedlichen Feldern. Für diese Arbeit ist allerdings eher der algorithmische interessant, da Knoten/Switches hinzukommen können oder auch wegfallen und von diesen aus Kanten zu anderen Knoten/Switches existieren. Wir ersetzen also einen Knoten durch einen Graphen, was der algorithmische Ansatz wäre.

3. Problemstellung

In der Aufgabenstellung geht es nun um das Publish/Subscribe-System, das in [KDTR12] vorgestellt wird. Hierbei soll die Methode mit Filtern verwendet werden. Die Aufgabe ist es, den derzeitigen Spannbaum oder auch Graph im Netz in einen neuen, vom Controller durch Veränderungen im System berechneten, zu transformieren. Diese Transformation soll während der Laufzeit stattfinden und während ihr sollen natürlich keine Nachrichten verloren gehen oder Duplikate entstehen. Das Ganze soll mit OpenFlow implementiert werden.

3.1. Aktuell vorhandenes und Related Work

Publish/Subscribe ist ein weit verbreitetes Thema und wir haben sicherlich schon Dienste genutzt, die darauf basieren, ob wir es mitbekommen haben oder auch nicht. Dadurch, dass es so weit verbreitet ist, gibt es eine Menge an Literatur zu diesem Thema. Allerdings bezieht sich diese Literatur auf Publish/Subscribe Systeme auf der Anwendungsebene. Da diese Arbeit und andere Arbeiten in derselben Umgebung jedoch einen Transfer der Funktionalität von Publish/Subscribe-Systemen auf der Anwendungsebene auf die Netzwerkebene anstreben, ist es ratsam Arbeiten zu betrachten, die ähnliche Funktionalität auf der Anwendungsebene bieten.

In [CRW01] wird eine Routingoptimierung zwischen den Brokern vorgestellt. Dabei wird angenommen, dass Subscriptions sich ähneln, bzw. viele Subscriptions gleich sind. Diese ähnlichen Subscriptions werden dann zusammengefasst und können auf einem Broker gespeichert werden. Das gleiche wird mit Publications gemacht. Auch diese werden zusammengefasst gespeichert. Damit entsteht zum einen weniger Traffic zwischen den Brokern, da nicht mehr alle Events an alle anderen Broker weitergeleitet werden muss, zum anderen ist aber auch weniger Speicher notwendig um die Informationen über Subscriptions und Publications zu speichern, da nicht mehr jeder Broker alle Subscriptions und Publications speichern muss, nur noch die relevanten Broker speichern diese ab. Damit erreicht man eine Optimierung dieser Struktur. Ähnliche Optimierungsansätze werden in [Müh02], [Pie04] und [JCL+10] vorgestellt. Spezialisierte Fälle werden zum Beispiel in [KBR09] für bestimmte Netzwerktypen, hier Wireless Mesh Netzwerke, in [TKKR09] für die Garantie verschiedener QoS mit gewisser Wahrscheinlichkeit und in [TKK+11] für QoS, die der Subscriber selber wählen und spezifizieren kann, vorgestellt.

Auch im SDN-Bereich gibt es schon eine recht große Anzahl an Forschungsarbeiten, obwohl das Thema SDN und OpenFlow noch nicht so lange aktuell ist. Darunter befinden sich

natürlich auch Arbeiten, die sich mit dem updaten von SDNs beschäftigen, also das gleiche untersuchen wie diese Arbeit. Eine dieser Arbeiten ist [GC12]. In dieser Arbeit wird beschrieben, wie man virtuelle Knoten konsistent von einem realen Knoten auf einen anderen realen Knoten portieren kann und dabei auch Bandbreitengarantien einhält. Das kommt unserer Problemstellung zwar nicht all zu nahe, ist aber ein Anfang an Updateprotokollen.

Was der Problemstellung dieser Arbeit näher kommt, sind die Ideen aus [RFRW11] und [McG12]. Die erste Arbeit beschreibt ein Updateprotokoll, dass sich vor allem dann eignet, wenn man Pakete von bestimmten Quellen, die als potentiell gefährlich gelten, überprüfen will und andere nicht. Das Beispiel aus der Arbeit ist, dass feste Zugänge in einem Netzwerk als sicher gelten und daher nicht überprüft werden, Gastzugänge prinzipiell als unsicher gelten und auf jeden Fall überprüft werden. Dieser Mechanismus darf beim Update nicht ausgeschaltet werden. Das Protokoll bietet genau diese Garantie.

Die zweite Arbeit bietet ein Prokoll, das dem Ziel dieser Arbeit schon sehr nahe kommt. Allerdings mit einem großen Unterschied. Das Protokoll, das vorgestellt wird bietet ein Update, ohne das Nachrichten verloren gehen, indem es mehrere Phasen hat. Es wird sichergestellt, dass Pakete, wenn sie einmal gestartet sind, entweder den alten oder den neuen Weg gehen, aber nicht gemischte. Wenn das auf einer Route nicht möglich ist, werden die Pakete an den Controller weitergeleitet, der sie in der nächsten Phase dann wieder freigibt.

Prinzipiell kommt das dem sehr nahe, was auch in dieser Arbeit versucht wird. Der große Unterschied besteht aber darin, dass im folgenden Protokoll der Controller nicht belastet werden soll. Im Protokoll, dass in [McG12] vorgestellt wird, besteht die Gefahr, dass der Controller unter der enormen Last, die durch einen Burst entstehen kann, wenn gerade geupdated wird, zusammenbricht und seine Funktion nicht mehr erfüllen kann oder enorme Verzögerungen entstehen. Um das zu verhindern werden im folgenden Protokoll andere Mechanismen greifen um Nachrichtenverlust zu verhindern und auch die Gefahr den Controller außer Gefecht zu setzen zu umgehen.

3.2. Problemstellung

In einem Publish-/Subscribe-System allgemein und auch in dem vorgestellten finden immer wieder Aktualisierungen, also neue, geänderte oder gelöschte Suscriptions oder Publications, statt. Es müssen also immer wieder neue Graphen mit neuen, geänderten oder gelöschten Kanten berechnet werden. Die Änderungen, die in dem Graph auftreten, müssen dann in Echtzeit auf das Netzwerk übertragen werden. OpenFlow bietet zwar die Möglichkeit Switches von einer zentralen Stelle aus zu rekonfigurieren und auch festzulegen was passiert, wenn keine Forwardregel vorhanden ist, allerdings bietet es keine Möglichkeit mehrere Switches mit Garantien wie Bandbreite oder eben Nachrichtenverlust zu rekonfigurieren. Solche Garantien müssen in solch einem System aber herrschen. Vor allem sollte kein Nachrichtenverlust entstehen und die Latenz sollte auch erträglich bleiben.

Das bedeutet, dass ein Protokoll zur Rekonfiguration der Graphen entwickelt werden muss, dass diese Garantien bietet. Am wichtigsten ist dabei die Garantie, dass keine Nachrichten während der Rekonfiguration verloren gehen. Diese Rekonfiguration ist eine Art von Graphentransformation, da wir zwei Graphen haben. Zum einen den Ausgangsgraphen, also den gerade im Netzwerk existierenden Spannbaum, und zum anderen den neuen Spannbaum. Ein Teilgraph hat sich nun geändert und dieser Teilgraph wird in einen anderen Teilgraph überführt, damit sich der neue Graph ergibt. Im Folgenden soll nun ein Protokoll vorgestellt und evaluiert, dass diese Überführung dynamisch während der Laufzeit durchführt und gewisse Garantien hält. Dabei stehen zwei Garantien im Vordergrund. Zum einen, dass kein Nachrichtenverlust existiert und zum anderen, dass keine Duplikate entstehen. Latenz, Bandbreite und Effekte auf andere Applikationen zählen auch mit in die Evaluierung hinein. Diese drei sollten nach Möglichkeit so gering wie möglich gehalten werden um zum einen die Effektivität möglichst hoch zu halten, wie auch Störeffekte möglichst niedrig.

3.3. Graphmodell

Um zu verstehen warum das folgende Modell gewählt wurde, lohnt es sich das umfassende Modell anzuschauen. Der Update-Mechanismus bekommt einen Graphen und muss diesen in das Netzwerk implementieren. Die Frage ist nun woher der Graph kommt, wie oft er sich ändert, etc. Prinzipiell ist die Idee, dass der Controller bei jeder neuen Subscription oder einer entfernten alten, oder aber neuen Publishern, einen neuen Graphen berechnet. Hier gibt es nun mehrere Möglichkeiten. Zum einen kann man den Graphen wirklich bei jedem neuen Ereignis neu berechnen oder aber man hat ein gewisses Intervall, in dem die Veränderungen akkumuliert werden und dann mit mehreren Veränderungen auf einmal ein neuer Graph berechnet wird. Nach Möglichkeit sollten beide Varianten unterstützt werden. Für eine erste Version ist die zweite Variante aber wahrscheinlicher, da sie einfacher zu realisieren ist. Angestrebt wird aber die erste, da diese die Dynamik im System erhöht. Bei großen Systemen ist es allerdings ratsam nicht den ganzen Graphen neu zu berechnen, sondern, wo es möglich ist, nur Teilgraphen. Das erfordert allerdings eine engere Verzahnung von Update-Mechanismus und Graphberechnungs-Mechanismus. Deswegen wird erst einmal angenommen, dass der Graph komplett berechnet wird und ein kompletter Graph für den Update-Mechanismus zur Verfügung steht.

Die Darstellung des ganzen erfolgt als ein gerichteter Graph $G = (V, E)$. Wobei V die Menge der Switches ist, die im Graphen existieren ist. E ist die Menge der Flows bzw. die Aufteilungen und Spezifizierungen der Filter für die Events, die zwischen den Switches existieren. Ein Flow auf einem Switch wird eindeutig durch seinen Filter identifiziert. Der Filter dient dazu nur einen gewissen Teil an Nachrichten weiterzuleiten. Der Filter ergibt sich aus den Subscribern, die auf dem Pfad weiter hinten als dieser Switch liegen. Jeder Switch besitzt nun eine Menge an Flows die eine Teilmenge von E ist. Ein Flow an sich besteht nun wiederum aus einer Menge an Links. Ein Link ist eine Verbindung von zwei Switches. Das heißt ein Flow auf einem Switch kann zu mehreren anderen Switches führen. Aufgrund dieser Spezifizierung ist es nun sehr wahrscheinlich, dass zwischen zwei Switches mehrere Kanten existieren. Diese gehören aber zu unterschiedlichen Flows und

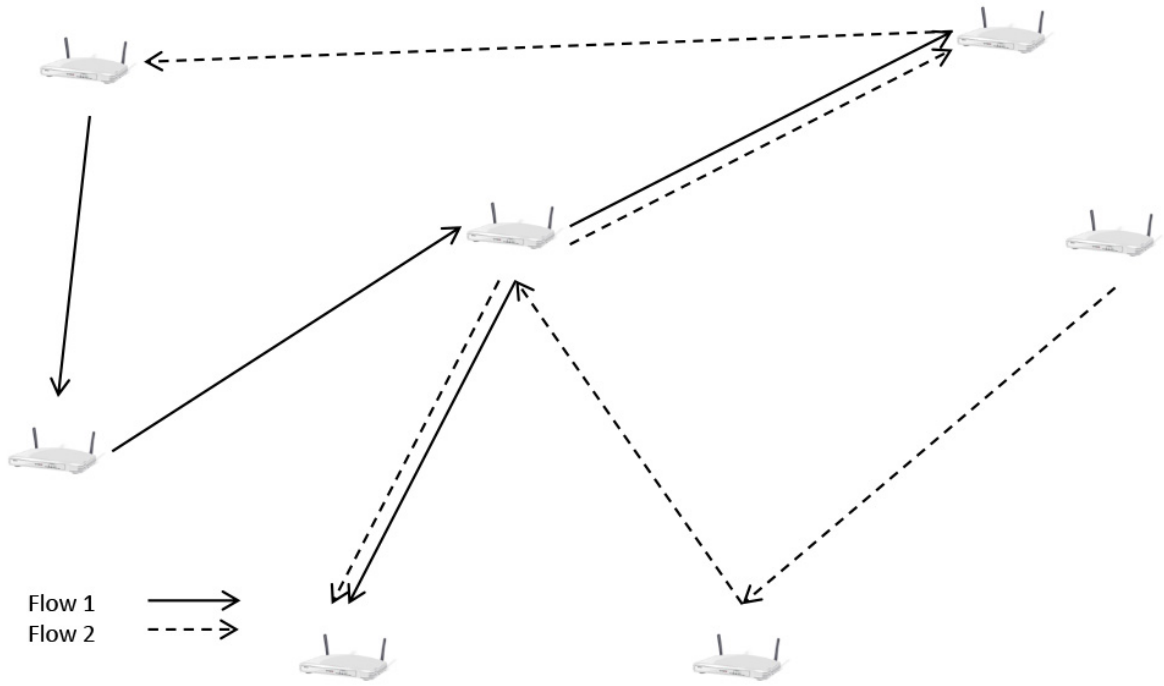

Abbildung 3.1.: Graph mit Teilgraphen und Mehrfachkanten

sind eindeutig voneinander durch ihre Filter unterscheidbar. Das heißt, dass es zwischen zwei Switches mehrere Kanten geben kann, aber keine zwei Kanten mit demselben Filter. Der Graph besitzt also Mehrfachkanten, welche durch ihre Filter spezifiziert werden. Allerdings bedeutet es auch, dass die Kanten in E nur durch die Angabe der dazugehörigen Switches eindeutig ist. Ein Flow kann sozusagen aus mehreren Kanten bestehen, wenn er mehrere Links hat. Durch die Flows ergibt sich nun noch eine weitere Eigenschaft des Graphen. Und zwar ist er gerichtet. Wenn ein Flow auf einem Switch installiert ist, dann bedeutet dies, dass Nachrichten die mit diesem Flow weitergeleitet werden erst einmal nur zum nächsten Switch weitergeleitet werden. Wenn die Nachrichten auch vom nächsten Switch zu diesem Switch gelangen sollen, dann muss auf dem anderen Switch wieder ein Flow existieren, der dies veranlasst. Da die Flows aber die Kanten sind, heißt das, dass der Graph gerichtet ist. Bidirektionalität sollte aber so oder so nicht existieren, da man sonst Zyklen im Graphen erhält, welche in unserer Art von Spannbaum nicht erwünscht sind, da sie zu Duplikaten führen können.

Von dieser Art von Graph existieren nun zwei. Einen alten, der bereits im Netzwerk implementiert ist, und einen neuen, der implementiert werden muss. Was nun geschehen soll ist eine Transformation vom alten Graphen in den neuen. Für die Transformation muss man aber die Unterschiede zwischen den beiden Graphen feststellen. Das erfolgt zum einen durch einen Vereinigungsgraphen und zum anderen durch fünf verschiedene Operationen, die die Veränderungen beschreiben. Die fünf Operationen sind *Add*, *Delete*, *Split*, *Merge* und

Abbildung 3.2.: Die Add-Operation

Abbildung 3.3.: Die Delete-Operation

Migrate. Diese Operationen ergeben sich direkt aus den Veränderungen von Subscribern oder Publishern. Unter diese Veränderungen fallen das Hinzukommen oder Wegfallen von Publishern oder Subscribern, oder aber auch eine Veränderung der Publications oder der Subscriptions. Diese fünf Operationen beschreiben die Veränderungen semantisch. Der Vereinigungsgraph bietet eine Möglichkeit zur Darstellung von allen Veränderungen aber auch von allem was gleich geblieben ist. Auf den Vereinigungsgraphen wird später aber noch weiter eingegangen.

Add

Wie der Name schon sagt bedeutet *Add*, dass eine Kante hinzugefügt wird. Eine Kante hinzufügen bedeutet einen neuen Flow zu installieren, der vorher auf diesem Switch noch nicht vorhanden war. Das kommt in der Regel dann vor, wenn ein neuer Subscriber oder Publisher ins System kommen und dieser dann eingebunden wird. Dabei muss man nicht darauf achten, ob schon eine Kante existiert oder nicht. Diese Operation gehört mit *Delete* zu den einfachen Operationen, da sie wenig Aufwand benötigt.

Delete

Delete ist die zweite einfache Operation. Sie tritt dann auf, wenn ein Publisher oder Subscriber sich aus dem System abmelden und eine Kante damit überflüssig geworden ist. Auch hier muss man nichts weiter beachten, da die Kante inaktiv ist, weil der Publisher oder Subscriber nicht mehr existiert.

Split

Split gehört zu den komplizierteren Operationen. Diese Operation kann zwei verschiedene Ausprägungen haben. Zum einen kann es nötig sein eine Kante mit dem gleichen Filter zu

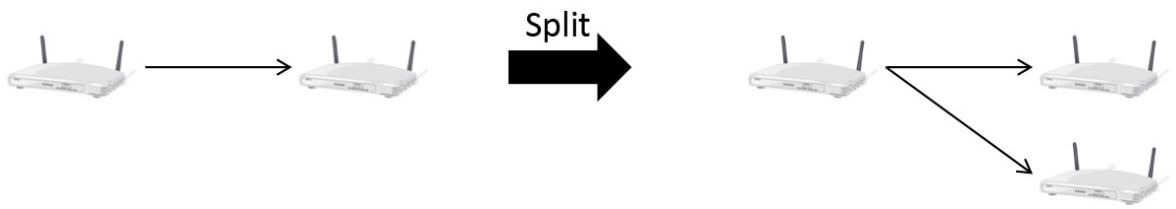

Abbildung 3.4.: Die Split-Operation

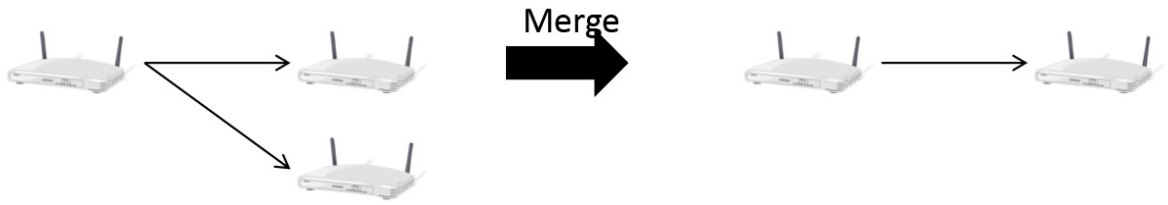

Abbildung 3.5.: Die Merge-Operation

einem oder mehreren zusätzlichen Switches zu leiten und zum anderen kann es nötig sein eine Kante zu spezifizieren und aufzuteilen. Beim ersten muss die Ursprungskante nicht geändert werden, da der Filter gleich bleibt, beim zweiten aber muss auch die Ursprungskante geändert werden, da der Filter sich ändert. Das Resultat ist in beiden Fällen eine Vermehrung von Kanten. Im ersten Fall bedeutet dies konkret, dass nur ein weiterer Link zu dem Flow auf dem Switch hinzugefügt werden muss. Beim zweiten muss sich auf jeden Fall der Filter ändern. Die Links müssen sich nicht zwangsläufig ändern, sondern können sich auch auf die neuen Flows die entstehen aufteilen.

Merge

Merge verhält sich umgekehrt zu Split. Wenn zwei oder mehr Kanten existieren und eine oder mehrere davon nicht mehr benötigt werden, werden sie zusammengeführt. Dabei muss man wieder zwischen einer Veränderung des Filters oder keiner Veränderung unterscheiden. Verändert sich der Filter, also wird er zum Beispiel weniger spezifiziert, dann muss die neue Kante auch geändert werden. Verändert er sich nicht, dann kann die eine Kante bleiben und nur die andere wird gelöscht. Auch hier kann es sein, dass nur Links verändert werden müssen. Dies ist der Fall, wenn der Filter sich nicht ändert.

Migrate

Ein *Migrate* findet statt, wenn der Pfad von einem Publisher zu einem oder mehreren Subscribern sich ändert. Das kann dadurch entstehen, dass sich viele Publisher und Subscriber

Abbildung 3.6.: Die Migrate-Operation

verändert haben oder aber auch dadurch, dass ein neuer Switch ins Netz kam und dieser eine schnellere oder kürzere Verbindung bietet. In beiden Fällen müssen dabei die betroffenen Kanten komplett neu gesetzt werden und alte entfernt. Dabei gibt es aber einen End- und Anfangspunkt, ab dem die Kanten nicht mehr verändert werden. In 3.6 wäre das jeweils der linke und untere Switch.

Diese Operationen können natürlich auch zusammen vorkommen. Wenn zum Beispiel ein *Migrate* stattfindet können sich dabei Flows aufteilen oder auch manche zusammengefasst werden. Gleichzeitig können noch welche hinzukommen oder auch gelöscht werden. Ist dies der Fall, dann ist es teilweise nur noch sehr schwer zu erkennen welche Operation für die Veränderung verantwortlich war. Dies ist bei der eigentlichen Transformation oder Findung der Veränderungen aber nicht relevant, denn die fünf Operationen kann man auf die zwei Operationen *Add* und *Delete* reduzieren wie nachher gezeigt wird.

4. Konzept

4.1. Voraussetzungen

Aufgrund des Aufbaus des Protokolls müssen verschiedene Voraussetzungen erfüllt sein. Diese gehen über die Voraussetzungen hinaus, die man an das OpenFlow-Protokoll stellen kann. Es wird allerdings davon ausgegangen, dass die Funktionen, die von OpenFlow zur Verfügung gestellt werden auch so funktionieren, wie es die Spezifikation darlegt.

Es gibt vor allem zwei wichtige Voraussetzungen für das Protokoll. Zum einen die Aufteilbarkeit der Transformation und zum anderen, dass diese Aufteilung disjunktiv möglich ist. Genauer gesagt haben wir zwei unterschiedliche Graphen bei dem der eine in den anderen überführt werden soll. Bei dieser Überführung werden eher selten alle Kanten neu gesetzt werden müssen, aber selbst das ist möglich. Für das Protokoll ist es nun wichtig, dass man kleine Transformationen nehmen kann, die man unabhängig von anderen ausführen kann. Alle die unabhängig voneinander sind, können gleichzeitig ausgeführt werden.

Man könnte zwar auch alle Transformationen hintereinander ausführen und bräuchte damit keine Unabhängigkeit bzw. Disjunktivität, allerdings würde damit die Leistung drastisch sinken und die Transformationen wären nicht mehr wirklich dynamisch. Damit wäre aber das Ziel die Rekonfiguration zur Laufzeit durchzuführen verfehlt. Deswegen wird versucht so viele Transformationen wie möglich gleichzeitig auszuführen. Mit der parallelen Ausführung wird dann zwar mehr auf einmal rekonfiguriert und damit verbunden haben mehr Nachrichten mehr Verzögerung, allerdings hält sich diese Verzögerung in Grenzen und wenn die Rekonfiguration an sich schneller abgeschlossen ist werden keine Nachrichten mehr verzögert. Bei der Hintereinanderausführung kann es dazu kommen, dass wenige Nachrichten sehr lange verzögert werden. Das ist zum Beispiel der Fall, wenn auf einem Pfad von einem Publisher zu einem Subscriber mehrere Transformationen liegen und diese vom Publisher zum Subscriber hin abgearbeitet werden. Damit kann für einzelne Nachrichten eine sehr hohe Verzögerung entstehen. Bei der parallelen Ausführung tritt dieser Fall zum einen sehr selten auf und zum anderen sehr abgeschwächt da diese Transformationen in der Regel parallel ausgeführt werden können.

Die zweite Voraussetzung ist die Zyklenfreiheit des Graphen. Ist dies nicht gegeben kann es zu Problemen bei der Berechnung von Transformationen kommen bzw. erschwert diese ungemein. Dies sollte allerdings durch den Aufbau des Graphen gegeben sein, da er aus Pfaden von Publishern zu Subscribern besteht. Der Spannbaum an sich ist die optimierte Ansammlung von diesen Pfaden. Optimiert in dem Sinn, dass möglichst wenig Verbindungen bestehen. Zyklenfreiheit können wir durch diesen Aufbau als gegeben annehmen. Ein Graph für ein Publish/Subscribe-System, wie es hier angenommen wird, wäre sicher nicht optimal,

wenn er Zyklen besitzt. Mit Zyklen würden Nachrichten teilweise gar nicht ankommen, womit das System nicht gerade brauchbar wäre.

4.2. Aufbau

Als erstes wollen wir uns den groben Aufbau anschauen, wie das Protokoll mit anderen Teilen interagiert und was sonst noch nötig ist um eine Rekonfiguration durchzuführen. In den nächsten Sektionen werden die einzelnen Teile dann genauer erklärt.

Der grundlegende Aufbau ist gleich wie in jedem SDN mit OpenFlow. Es ist ein Controller vorhanden, der alles handhabt und managt. In diesem Controller ist jetzt auch eine Einheit die neue Spannbäume berechnet, wenn sich im Netz etwas geändert hat. Der alte Spannbaum muss dann in den neuen Spannbaum überführt werden, was die Aufgabe dieser Arbeit ist. Der Mechanismus dazu befindet sich auch im Controller. Die Veränderungen ergeben sich durch neue, veränderte oder gelöschte Subscriptions, Subscriber, Publications oder Pubslisher. Der Controller hat einen Überblick über alle aktiven Publisher und Subscriber und über ihn laufen auch neue Anmeldungen, da Pakete, die einen Table-Miss verursachen automatisch zum Controller geleitet werden. Außerdem soll es laut dem Konzept des Publish/Subscribe-Systems in jedem Switch einen Eintrag geben, der die Anmeldung zu Subscriptions handhabt. Diese laufen über eine statische Adresse und jeder Switch besitzt für diese Adresse eine Weiterleitung an den Controller. Jedes Paket, dass über diese Adresse an den Controller gelangt, wird von diesem bearbeitet und der Controller passt daraufhin den Graphen an, damit die Subscriptions berücksichtigt wird.

Der Controller besitzt noch eine zweite nötige Einheit, den Scheduler. Dieser berechnet mögliche disjunktive und unabhängige Transformationen und gibt diese in Auftrag. Sind die Transformationen voneinander unabhängig, können sie gleichzeitig ausgeführt werden. Sind sie es nicht, müssen sie hintereinander ausgeführt werden. Hier benötigen wir die Voraussetzung, dass es möglich ist die komplette Transformation in kleine, unabhängige Transformationen aufzuteilen. Der Scheduler vergleicht die Graphen und findet Unterschiede heraus. Wie genau dies funktioniert wird später erklärt.

Diese kleinen Transformationen werden nun von dem Updateprotokoll bearbeitet und umgesetzt. Das Protokoll an sich hat allerdings einen Einfluss auf den Scheduler. Dieser muss die Aufteilung so vornehmen, dass sie konform den Mechanismen des Protokolls sind.

4.3. Transformation

Bevor wir uns nun die verschiedenen Mechanismen genauer anschauen, sollten wir festlegen, was genau Transformationen sind, was für sie gilt und auch, was sie beinhalten. Als erstes sollten wir etwas vom Ende aus Kapitel 3 Absatz Graphmodell aufgreifen. Hierbei geht es um die Darstellung der drei komplizierteren Operationen durch die beiden leichteren.

Oder anders gesagt, man kann mit *Add* und mit *Delete* die anderen drei Operationen darstellen. Das ist insofern wichtig, da es in OpenFlow auch nur das Hinzufügen, Löschen und Modifizieren eines Flows gibt. Wobei modifizieren bedeutet, dass man die Aktionen des Flows verändert, also zum Beispiel zu welchem Port die Nachricht weitergeleitet werden soll. Dies kann man zum Beispiel dann nutzen, wenn man bei einem Flow nur einen oder mehrere Links verändern muss. In Floodlight ist das modifizieren allerdings nicht komplett implementiert, weswegen nur hinzufügen und löschen genutzt wird.

Die drei komplizierteren Operationen sind *Split*, *Merge* und *Merge*. Jede von ihnen kann mit einer oder mehreren Anwendungen von *Add* oder *Delete* dargestellt werden. *Split*: Wenn der Filter des Ursprungsflows sich nicht verändert, dann benötigt man ein oder mehrere *Adds*, je nachdem in wie viele Flows sich der Urpsrungsflow aufteilt. Wobei hier Links hinzugefügt werden müssen, mehr nicht. Ändert sich allerdings der Ursprungsflow, dann benötigt man erst ein *Delete* und dann ein oder mehrere *Adds* um die Operation anzuwenden.

Merge: Wenn der Filter des Ursprungsflows sich nicht verändert, dann benötigt man ein oder mehrere *Deletes*, je nachdem wie viele Flows zu einem verschmolzen werden. Verändert sich der Filter aber, dann benötigt man zusätzlich zu den *Deletes* noch ein *Add* um den veränderten Flow zu installieren. Auch hier genügt es nur die Links zu ändern, wenn sich der Filter nicht ändert.

Migrate: Hier benötigt man ein oder mehrere *Deletes* um die alten Flows zu entfernen und einen oder mehrere *Adds* um die neuen Flows auf den neuen Switches zu implementieren, je nachdem wie viel migriert werden muss.

Wenn verschiedene Operationen auf einmal angewendet werden, dann benötigt man je nachdem unterschiedlich viele *Deletes* und *Adds*. Die Anzahl davon können aber geringer sein als die reine Addition der *Deletes* und *Adds* der Einzeloperationen, da sich manche davon möglicherweise überschneiden. Wie viele *Deletes* und *Adds* an sich benötigt werden hängt davon ab, wie groß die jeweilig ersetze Operation ist.

Mit diesem Wissen können wir nun auch erklären, was eine Transformation ist und aus was sie besteht. Eine Transformation ist eine Ansammlung von Switches und darauf befindlichen Flows. Diese Flows werden entweder entfernt oder hinzugefügt. Die Ansammlung der Switches und Flows in einer Transformation machen einen zusammenhängenden Teilgraphen des Graphen aus. Das bedeutet, dass die Switches über die Flows zusammenhängen. Jeder der Switches besitzt einen oder mehrere andere Switches in der Transformation als Nachbar. Zusammenhängend bedeutet hier, dass die Filter der Flows voneinander abhängig sind. Das sind sie dann, wenn sie die den exakt selben Filter haben oder wenn der kürzere Filter vollkommen im längeren enthalten ist. Also wenn der kürzere Filter ein Präfix des längeren Filters ist. Diese Definition von zusammenhängend wird gewählt, da diese Flows ein Pfad für Nachrichten mit bestimmten Adressen sind. Alle Veränderungen die zusammenhängend auf diesem Pfad liegen sollten auf einmal umgesetzt werden. Eine Transformation besitzt zusätzlich noch einen kleinsten Filter. Das ist der kleinste Präfix von allen in der Transformation enthaltenen Flows.

Transformationen sind also zusammenhängende Teilgraphen, die Kanten beinhalten, die gelöscht oder hinzugefügt werden. Die Switches die in der Transformation enthalten sind

Algorithmus 4.1 Scheduler

```
procedure SCHEDULER(neuer Spannbaum)
    if alter Spannbaum = empty then
        INITIALISIERUNG(neuer Spannbaum)
    else
        BERECHNE VEREINIGUNGSGRAPH(alter Spannbaum, neuer Spannbaum)
        BERECHNE TRANSFORMATIONEN(Vereinigungsgraph, GeänderteKnoten)
        MERGE TRANSFORMATIONEN(Vektor von Transformationen)
        BERECHNE DISJUNKTIVE TRANSFORMATIONEN(Vektor von Transformationen)
        BERECHNE ZU BLOCKENDE SWITCHES(Vektor von Vektor von disjunktiven Transformationen)
    end if
    for all Vektoren von disjunktiven Transformationen do
        FÜGE ZUM UPDATEMECHANISMUS HINZU(Vektor von Vektor von disjunktiven Transformationen)
    end for
    alter Spannbaum = neuer Spannbaum
end procedure
```

Algorithmus 4.2 Initialisierung

```
procedure INITIALISIERUNG(neuer Spannbaum)
    Füge alle Knoten und Flows in eine Transformation
    Wende Transformation an
end procedure
```

besitzen Kanten, die sich verändern sollen. Zusätzlich gehören dazu noch Informationen, wie dafür gesorgt wird, dass die Transformation implementiert werden kann, ohne, dass Komplikationen entstehen. Welche Informationen das sind wird im übernächsten Absatz erläutert.

4.4. Scheduler

Als erstes wollen wir auf den Scheduler eingehen. Er berechnet die einzelnen kleinen Transformationen und auch, welche dieser Transformationen parallel ausgeführt werden können. Gleichzeitig berechnet er für jede Transformation, welche Switches kurzzeitig blockiert werden müssen, damit die Flows aktualisiert werden können. In 4.1 sehen wir den prinzipiellen Aufbau und aus welchen Funktionen er besteht. Diese Funktionen werden hintereinander aufgerufen und berechnen alles Nötige.

Falls noch keine Flows und damit kein Graph vorhanden ist, wird das ganze wie in 4.2 initialisiert. Alle Switches und Flows werden in einer großen Transformation zusammengefasst und diese wird dann auf einmal auf den Switches implementiert. Ist schon ein Graph vorhanden

Algorithmus 4.3 Berechne Vereinigungsgraph

```
procedure BERECHNE VEREINIGUNGSGRAPH(alter Spannbaum, neuer Spannbaum)
    Vereinigungsgraph = ∅
    GeänderteKnoten = ∅
    Eltern = ∅
    for all Publisher in altem Spannbaum do
        Markiere Publisher = OLD
        Füge Publisher hinzu
    end for
    for all Publisher in neuem Spannbaum do
        if Publisher schon vorhanden then
            Markiere Publisher = BOTH
        else
            Markiere Publisher = NEW
            Füge Publisher hinzu
        end if
    end for
    for all Switches in altem Spannbaum do
        for all Flows auf Switch do
            Markiere Flow = OLD
            Füge Flow hinzu
            FÜGE ELTERN HINZU(Switch, Flow, Eltern)
            Füge Switch zu GeänderteKnoten hinzu
        end for
    end for
    for all Switches in neuem Spannbaum do
        if Switch vorhanden then
            for all Flows auf Switch do
                if Flow vorhanden und identisch then
                    Markiere Flow = BOTH
                    FÜGE ELTERN HINZU(Switch, Flow, Eltern)
                    Entferne Switch aus GeänderteKnoten
                else if Flow vorhanden und nicht identisch then
                    Markiere Flow = CHANGE
                    FÜGE ELTERN HINZU(Switch, Flow, Eltern)
                    Füge Switch zu GeänderteKnoten hinzu wenn noch nicht vorhanden
                else
                    Markiere Flow = NEW
                    Füge Flow hinzu
                    FÜGE ELTERN HINZU(Switch, Flow, Eltern)
                    Füge Switch zu GeänderteKnoten hinzu wenn noch nicht vorhanden
                end if
            end for
        else
            Markiere alle Flows
            Füge Switch mit allen Flows hinzu
            FÜGE ELTERN HINZU(Switch, Flow, Eltern)
            Füge Switch zu GeänderteKnoten hinzu
        end if
    end for
end procedure
```

wird zuerst ein Vereinigungsgraph aus dem alten und dem neuen Graphen erzeugt. Dies wird in 4.3 veranschaulicht. Hierbei werden alle Informationen aus den beiden Graphen zusammengefügt. Das fängt mit den Publishern an und geht weiter über die Switches und die Flows auf den Switches und dann die Links, die zu den Flows gehören. Dabei wird die Zugehörigkeit der Publisher und Flows markiert. Bei den Publishern ist dies entweder der neue Graph, der alte oder beide. Bei den Flows kommt zu diesen drei Kategorien noch eine weitere hinzu. Und zwar wenn der Flow zwar existiert, allerdings verändert wurde. Das ist dann der Fall, wenn sich seine Links geändert haben. Vorkommen kann das, wenn zum Beispiel ein *Split* durchgeführt wurde, der Filter des Flows sich aber nicht geändert hat. Zusätzlich zu diesen Informationen, die aus dem alten und neuen Graphen kommen, werden noch weitere Informationen generiert. Zum einen werden die Eltern der Switches berechnet und zum anderen werden alle Switches gespeichert, auf denen ein Flow eine Veränderung erfahren hat.

Die Eltern werden nach jedem Hinzufügen eines Switches für die Flows auf dem Switch berechnet. Hier wird vermerkt, von welchen Switches der aktuelle Switch der Eltern-Switch ist und für welche Flows das zutreffend ist. Damit baut sich nach und nach diese Information auf und am Ende sind alle Informationen, welcher Switch wessen Eltern-Switch ist, vorhanden. Die Berechnung für die Eltern ist nötig, da nachher, um eine Transformation aufzubauen, Eltern und Kinder untersucht werden. Direkt aus dem Graphen kann man es nicht ablesen, da Flows Nachrichten nur weiterleiten und es nicht wichtig ist, woher diese kommen. Sie besitzen diese Information also nicht. Da wir es nachher aber benötigen, berechnen wir die Eltern hier. Der Aufwand dafür ist relativ gering.

Das nächste was nebenbei noch berechnet wird sind die Switches, auf denen sich etwas ändert. Dies ist wichtig, da beim Berechnen von Transformationen von ihnen zufällig ein Switch ausgewählt wird und von ihm Transformationen starten. Um diese Switches hinzuzufügen werden erst einmal alle Switches aus dem alten Graphen hinzugefügt und dann die entfernt, die auch im neuen Graphen vorkommen und keine Veränderung besitzen. Damit wird sichergestellt, dass nachher wirklich alle Switches mit Veränderungen darin enthalten sind. Wären manche nicht enthalten, würden sie nachher möglicherweise nicht berücksichtigt und es würden Flows auf ihnen bestehen bleiben. Das kann zu unerwünschten Nebeneffekten führen

Mit dem Vereinigungsgraphen können wir nun die eigentlichen Transformationen berechnen. Dabei wählen wir aus den zuvor berechneten geänderten Switches so lange zufällig welche aus und starten von ihnen Transformationen bis diese Menge leer ist. Vorgegangen wird dabei wie in 4.4 veranschaulicht. Nachdem man einen Switch zufällig gewählt hat werden die Flows auf ihm zusammengefasst. Dabei werden Flows zusammengefasst, bei denen der eine Flow ein Präfix des anderen Flows ist oder beide einen gleichen Präfix besitzen. Das wird gemacht, da diese Flows voneinander abhängig sind oder abhängig sein können. Zwar nicht direkt, aber es kann vorkommen, dass sie einen gemeinsamen Flow haben, der vor diesen beiden Flows auf einem Switch weiter vorne im Pfad kam. Dieser könnte sich aufgespalten haben und auf dem aktuellen Switch wieder zusammengekommen sein. Damit diese Flows nachher in einer Transformation sind, werden Flows mit ähnlichem Präfix

Algorithmus 4.4 Berechne Transformationen

```
procedure BERECHNE TRANSFORMATIONEN(Vereinigungsgraph, GeänderteKnoten)
    Wähle zufälligen Knoten aus GeänderteKnoten aus
    Fasse matchende Flows zusammen
    Starte Transformation für jede matchende Flowmenge
    Markiere Flows
    Setze kleinsten Präfix für Transformation
    for all Transformationen do
        PRÜFEKNOTEN(Kinderknoten)
        PRÜFEKNOTEN(Elternknoten)
    end for
    Entferne abgearbeitete Knoten aus GeänderteKnoten
end procedure
```

Algorithmus 4.5 PrüfeKnoten

```
procedure PRÜFEKNOTEN(Knoten)
    if Wurde noch nicht besucht und hat matchende Flows then
        Fasse matchende Flows zusammen
        Füge passende Flowmenge hinzu
        Markiere Flows
        Update kleinsten Präfix für Transformation wenn nötig
        for all Transformationen do
            PRÜFEKNOTEN(Kinderknoten)
            PRÜFEKNOTEN(Elternknoten)
        end for
    end if
end procedure
```

zusammengefasst. Zudem wird damit dafür gesorgt, dass nachher weniger Transformationen ausgeführt werden müssen und damit die Leistung steigt.

Für jede Menge dieser zusammengefassten Flows wird dann eine Transformation gestartet und die Flows hinzugefügt. Gleichzeitig werden die Flows, die hinzugefügt wurden markiert, dass sie schon besucht und abgearbeitet worden sind. Mit dieser Transformation und ihrem kleinsten Präfix werden dann alle Kinder- und alle Eltern-Switches überprüft, wobei 4.5 aufgerufen wird. Wenn der Switch noch nicht besucht wurde und auf ihm Flows sind, die zu der Transformation passen, bei denen die Filter entweder übereinstimmen oder einen gemeinsamen Präfix haben, werden die Flows zu der Transformation hinzugefügt, markiert und wieder alle Kinder- und Eltern-Switches untersucht. Der kleinste Präfix wird bei Bedarf aktualisiert. Das wird rekursiv so lange gemacht, bis nichts mehr hinzugefügt wird. Sind die Transformationen für den ursprünglich ausgewählten Startswitch dann abgeschlossen, wird die Menge an geänderten Switches untersucht und wenn ein Switch keine Flows mehr

Algorithmus 4.6 Merge Transformationen

```
procedure MERGE TRANSFORMATIONEN(Vektor von Transformationen)
    if Überschneidung zwischen zwei Transformationen then
        Führe Transformationen zusammen
    end if
end procedure
```

Algorithmus 4.7 Berechne disjunktive Transformationen

```
procedure BERECHNE DISJUNKTIVE TRANSFORMATIONEN(Vektor von Transformationen)
    for all Transformationen im Vektor do
        Vergleiche mit schon überprüften Transformationen
        if disjunktiv mit allen Transformationen im Untervektor then
            Füge in den Untervektor ein
        else
            Erstelle neuen Untervektor und füge Transformation ein
        end if
    end for
end procedure
```

besitzt, die verändert werden müssen und noch nicht besucht sind, wird er aus der Menge entfernt.

Nun haben wir eine Menge an Transformationen. Es kann jetzt allerdings vorkommen, dass zwei oder möglicherweise auch mehr Transformationen sich überlappen und einen gleichen Präfix im Filter haben. Das kann dadurch entstehen, dass der Aufbau der Filter theoretisch beliebig sein kann. Auf den Pfaden von Publisher zu Subscriber ist es nicht notwendig, dass die Filter sich immer weiter spezifizieren, sie können auch wieder unspezifischer werden. Durch diese Beliebigkeit kann es nun sein, dass der Algorithmus zur Konstruktion der Transformationen nicht ausreicht und möglicherweise erst zu spät einen ausreichend kleinen Präfix für den Filter hat, dass im Verlauf einige Flows nicht berücksichtigt wurden. Um das zu beheben werden die berechneten Transformationen nun untersucht und bei Bedarf gemergt (4.6). Gemergt werden sie dann, wenn sie einen gleichen oder ähnlichen Präfix haben und sie sich überlappen, also Switches sich in beiden Transformationen befinden.

Die nun bereinigten Transformationen werden danach auf ihre Disjunktivität untersucht. Dabei werden die Transformationen hintereinander durchgegangen und jede Transformation nur mit den schon untersuchten Transformationen verglichen. Dabei wird für jede Menge von disjunktiven Transformationen ein Vektor erzeugt. Ist eine Transformation disjunktiv mit allen bisher in dem Vektor befindlichen Transformationen, wird sie hinzugefügt. Wenn nicht, dann wird der nächste Vektor untersucht, so lange, bis kein Vektor mehr vorhanden ist und dann wird ein neuer Vektor erzeugt. Mit diesem Vorgehen werden sicher nicht die optimalen Ergebnisse erzielt, aber es ist ein einfaches Vorgehen, das für diese Arbeit vorerst ausreicht.

Algorithmus 4.8 Berechne zu blockierende Switches

```
procedure BERECHNE ZU BLOCKIERENDE SWITCHES(Vektor von Vektor von disjunktiven Transformationen)
    for all Transformationen do
        for all Switches in Transformation do
            if Switch besitzt Publisher then
                Füge Switch zu den zu blockierenden Switches mit NAK hinzu
            end if
        end for
        for all Randswitches do
            Suche nächstgelegensten Switch den man blockieren kann
            if Switch mit Publisher then
                Füge Switch zu den zu blockierenden Switches mit NAK hinzu
            else if Switch mit Publisher und Zirkel then
                Füge Switch zu den zu blockierenden Switches mit BOTH hinzu
                Berechne die Switches die für den Zirkel zusätzlich notwendig sind
            else
                Füge Switch zu den zu blockierenden Switches mit CIRCLE hinzu
                Berechne die Switches die für den Zirkel zusätzlich notwendig sind
            end if
        end for
    end for
end procedure
```

Als letzten Schritt müssen jetzt noch die Switches berechnet werden, die blockiert werden müssen um die Transformation nachher durchführen zu können. Warum wir das brauchen und was genau damit gemacht wird, wird im Abschnitt Protokoll erklärt. Um die Switches, die blockiert werden, müssen zu finden, werden erst einmal alle Switches in der Transformation durchgegangen. Dabei wird überprüft, ob einer der Switches mit einem Publisher verbunden ist, der Nachrichten veröffentlicht, die auf den Präfix der Transformation passen. Ist dies der Fall, wird der Switch zu den zu blockierenden Switches hinzugefügt. Danach wird von den Randswitches aus nach Switches gesucht, die möglichst nahe liegen und die man zum Blockieren nutzen kann. Findet man einen Switch mit einem Publisher, dann fügt man ihn hinzu als einen Switch, bei dem der Publisher blockiert werden muss. Findet man einen Switch, auf dem man einen Zirkel installieren kann, fügt man diesen als Zirkelswitch hinzu und berechnet die weiteren Switches, die für den Zirkel notwendig sind. Es kann aber auch vorkommen, dass auf einem Switch sowohl ein Publisher blockiert, als auch ein Zirkel installiert werden muss. Dieser Switch wird entsprechend gekennzeichnet und auch für ihn werden die Switches für den Zirkel berechnet.

Nun haben wir disjunktive Mengen von Transformationen mit allen nötigen Informationen. Diese Transformationen werden als letztes an den Update-Mechanismus übergeben und dieser kümmert sich um alles Weitere.

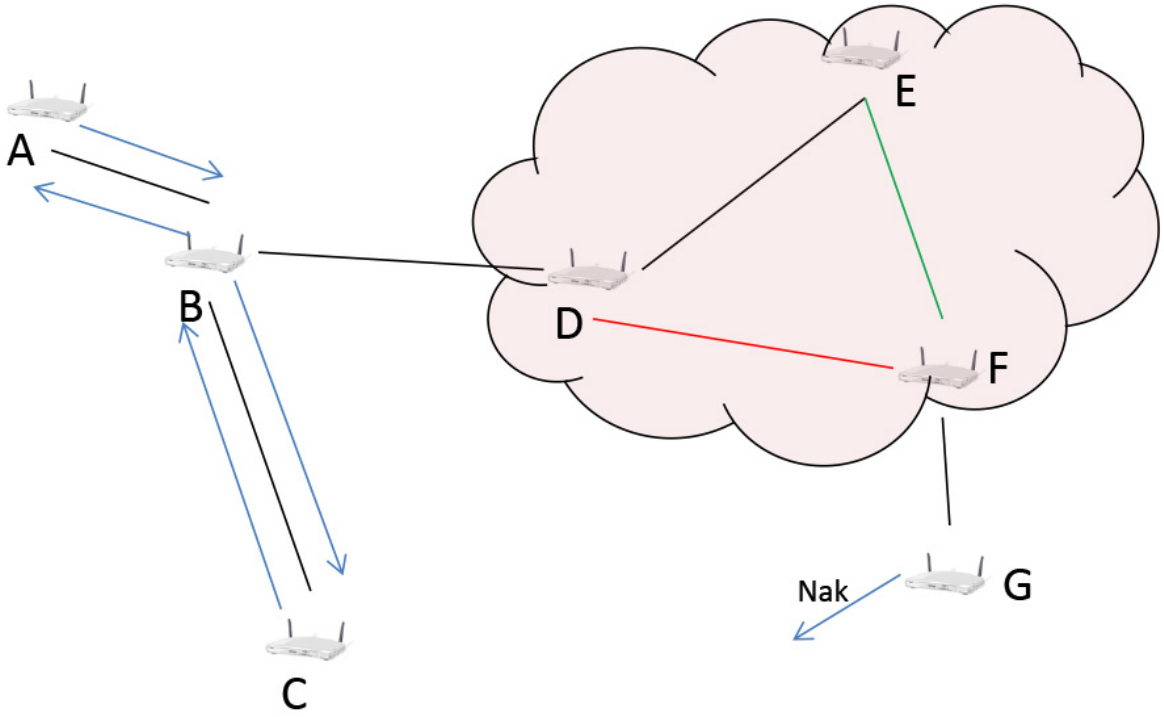

Abbildung 4.1.: Veranschaulichung des Protokolls

Algorithmus 4.9 Starte Update

```
procedure STARTE UPDATE(Vektor von disjunktiven Transformationen)
    for all Transformationen in Vektor do
        Starte Thread mit BEARBEITE TRANSFORMATION(Transformation)
    end for
end procedure
```

4.5. Protokoll

Prinzipiell arbeitet das Protokoll ähnlich wie ein Lockingprotokoll. Die Stelle in der eine Transformation stattfinden soll, wird isoliert, so, dass keine Nachrichten mehr fließen. Wenn in der isolierten Stelle keine Nachrichten mehr vorhanden sind, kann man die Flows beliebig ändern. Damit aber die Nachrichten nicht von den Switches fallengelassen werden, weil kein Flow vorhanden ist, oder diese Nachrichten an den Controller weitergeleitet werden, ist es nötig diese Nachrichten in irgendeiner Art und Weise festzuhalten oder einzufrieren. Wie das funktioniert wird in 4.1 gezeigt. Die Flows auf den Switches D, E und F sollen geändert werden. Der Scheduler hat diese Transformation berechnet und auch gleichzeitig welche Switches wie kurzzeitig geblockt werden. In diesem Beispiel soll die Kante zwischen D und F entfernt und die Kante zwischen E und F eingefügt werden. Um die Blockierung

Algorithmus 4.10 Bearbeite Transformation

```
procedure Bearbeite Transformation(Transformation)
    for all Switches bei denen der Publisher blockiert wird do
        for all Flows auf Switch do
            Füge NAK hinzu(Switch, Flow)
        end for
    end for
    for all Switches mit Zirkel do
        for all Flows auf Switch do
            Füge Zirkel hinzu(Switch, Flow)
        end for
    end for
    Warte 10 ms
    for all Zu entfernende Flows do
        Lösche Flow
    end for
    for all Hinzuzufügende Flows do
        Füge Flow hinzu
    end for
    Warte 5 ms
    for all Switches bei denen der Publisher blockiert wird do
        for all Flows auf Switch do
            Entferne NAK(Switch, Flow)
        end for
    end for
    for all Switches mit Zirkel do
        for all Flows auf Switch do
            Entferne Zirkel(Switch, Flow)
        end for
    end for
    Warte 5 ms
    for all Switches mit Zirkel do
        for all Flows auf Switch do
            Entferne Markierentferner(Switch, Flow)
        end for
    end for
end procedure
```

zu realisieren gibt es nun drei Mechanismen wobei nur zwei davon implementiert sind. Der dritte ist eher eine Möglichkeit für die Zukunft, da er Voraussetzungen hat, die nicht so einfach zu erfüllen sind. Dennoch wird er erwähnt und auch nachher vorgestellt. Die beiden implementierten Mechanismen sind zum einen ein NAK-Mechanismus an Switches, die mit Publishern verbunden sind und ein Zirkel-Mechanismus für Switches mitten im Netz, der zwischen zwei Switches läuft. Die dritte, nicht implementierte Lösung, nutzt einen temporären Speicher.

Diese Mechanismen garantieren nun zum einen, dass keine Nachrichten während der Transformation in der isolierten Stelle sind, zum anderen aber auch, dass keine Nachrichten verloren gehen oder Duplikate entstehen. Da Transformationen in der Regel nicht nur einen Flow betreffen, sondern mehrere, werden auch mehrere der Mechanismen nötig sein um eine isolierte Stelle sicherzustellen. Hierbei ist es dann wichtig, dass man die Mechanismen an Stellen platziert, die möglichst ideal sind. Ideal bedeutet hier, dass die Switches, die blockiert werden sollen, möglichst nahe an der isolierten Stelle liegen sollten. Würde man Switches nehmen, die weiter von der isolierten Stelle entfernt sind, dann bräuchte man zwangsläufig mehr Switches, die man blockieren muss. Es kommen immer mehr Pfade und Abzweigungen hinzu, die man beachten muss. Mit jedem weiteren Switch, der blockiert werden soll, steigt nicht nur der Aufwand für den Controller, er muss mehr Regeln implementieren, sondern auch die Zeit für die Transformation. Auch wenn das setzen eines Flows relativ schnell geht, sollte man die Zeit dennoch nicht unterschätzen, wenn es um sehr viele Regeln geht. Zudem steigt auch die Belastung für das Netzwerk, da länger blockiert werden muss. Alles in allem verschlechtert das Setzen von nicht idealen blockierten Switches die Gesamtleistung des Systems, was man nach Möglichkeit vermeiden will.

Das Ziel ist es jetzt nicht nur keine Nachrichten zu verlieren oder Duplikate entstehen zu lassen, sondern auch eine Transformation möglichst kurz zu halten. Zum einen ist das möglich mit weniger blockierten Switches, zum anderen aber auch mit Timing. Timing zum einen im Sinne von setzen von Regeln, wann und wo diese Regeln gesetzt werden sollen, zum anderen aber auch im Aufteilen der Gesamttransformation in möglichst effektive Einzeltransformationen, von denen man nach Möglichkeit sehr viele gleichzeitig ausführen kann. Hier kann man zum Beispiel Transformationen, die sich noch in der Warteschlange befinden, aber schon wieder obsolet sind, entfernen. Diese können sich beim Update-Mechanismus stauen, wenn sehr viele Transformationen übergeben werden, aber sie nicht schnell genug abgearbeitet werden können. Das ist eine Optimierung und es gibt sicher noch weitere. Allerdings geht es in dieser Arbeit erst einmal um eine Grundfunktionalität. Optimierungen können Teil möglicher weiterer Arbeiten sein.

In 4.9 fängt der Update-Mechanismus nun an. Hier wird ein Vektor von disjunktiven Transformationen gleichzeitig bearbeitet. Dabei wird ein Thread für jede Transformation erzeugt, in dem diese bearbeitet wird. Die Bearbeitung findet dann in 4.10 statt. Dabei werden erst einmal alle Switches blockiert, entweder mit einem NAK oder mit einem Zirkel. Nach einer kurzen Wartezeit werden die alten Flows entfernt und die neuen hinzugefügt. Das Hinzufügen oder Entfernen der Flows läuft dabei über den Controller. Dieser bietet eine Funktion, der die eigentliche Implementierung auf den Switches übernimmt. Nachdem die

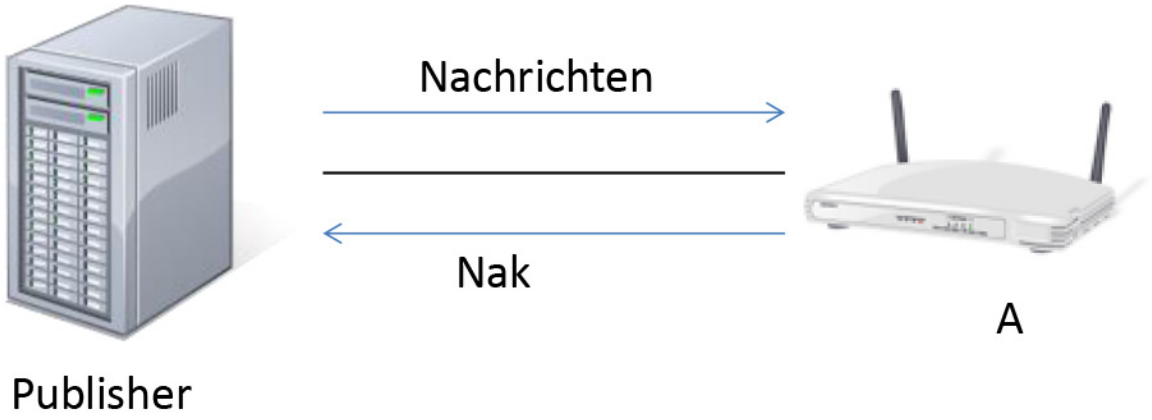

Abbildung 4.2.: Nak-Mechanismus

Flows nun entfernt und hinzugefügt wurden, werden die Blockier-Mechanismen wieder abgebaut und der normale Betrieb kann fortgesetzt werden.

Die Wartezeit zwischen den verschiedenen Schritten wurde der Einfachheit halber fest gewählt. Hier ist sicher noch Platz für Optimierungen. Je größer die Transformation ist, desto mehr Zeit wird benötigt. Bei kleineren wird dementsprechend weniger Zeit benötigt. Das Warten ist allerdings nötig, da Nachrichten, die sich in der Transformation befinden, noch weitergeleitet werden müssen. Außerdem benötigt es auch eine gewisse Zeit, bis die Flows implementiert sind. Diese liegt sicher nicht im Millisekundenbereich, aber für die Tests war diese Zeit in Ordnung, da auf virtuellen Switches getestet wurde, die eine Größenordnung langsamer als echte Switches sind.

Die verschiedenen Mechanismen werden im Folgenden nun vorgestellt. Wo welcher dieser Mechanismen eingesetzt wird, legt der Scheduler fest.

4.5.1. NAK-Mechanismus

Zuerst widmen wir uns dem NAK-Mechanismus. Hier ist es so, dass, wenn derzeit keine Transformation stattfindet, an dem der Switch, mit dem der Publisher verbunden ist, teilnimmt, die Nachrichten normal weitergeleitet werden. Wird der Switch aber zum Blockieren für eine Transformation ausgewählt und der Publisher publiziert Nachrichten, die relevant für die Transformation sind, dann wird eine zusätzliche Regel auf dem Switch implementiert, die eine Art von NAK an den Publisher sendet. Die Regel besitzt dabei eine höhere Priorität als die normale Regel. Somit kann die normale Regel beibehalten werden und muss nicht gelöscht werden. Dieser NAK besteht aus der kompletten Nachricht des Publishers. Der Publisher benötigt dafür einen Mechanismus, der solche zurückgesendeten Nachrichten speichert und versucht erneut zu senden. Das Ganze ist kein klassischer ACK/NAK-Mechanismus und bietet den Vorteil, dass man nicht alle Nachrichten speichern

Algorithmus 4.11 Füge NAK hinzu

```
procedure FÜGE NAK HINZU(Switch, Flow)
    ADDFLOW(Flow-Name, Switch, Flow)          // Funktion vom Controller
end procedure
```

Algorithmus 4.12 Entferne NAK

```
procedure ENTFERNE NAK(Switch, Flow)
    DELETEFLOW(Flow-Name)                     // Funktion vom Controller
end procedure
```

muss, bis ein ACK zurückkommt. Nur wenn einmal der Publisher blockiert wird, müssen Nachrichten gespeichert werden und periodisch versucht werden, diese erneut zu senden. Das ganze bietet auch den Vorteil, dass am Switch, im Fall des nicht blockierten Publishers, weniger Traffic entsteht. Er muss nur eine Nachricht weiterleiten. Dafür erhöht sich der Traffic leicht, wenn der Publisher blockiert wird. Allerdings ist diese Zeitspanne eher kurz.

Um diesen Mechanismus umzusetzen benötigt es sowohl eine Funktion, die die NAK-Regel hinzufügt, als auch eine, die sie wieder entfernt. Diese beiden werden in 4.11 und 4.12 gezeigt. Beide Funktionen sind denkbar einfach, da sofort nachdem die Regel implementiert wurde die Nachrichten zurückgesendet werden oder aber sie wieder normal gesendet werden. Bei diesem Mechanismus gibt es allerdings ein paar Eigenheiten, wenn man Multicast-Adressen verwendet. Diese werden nachher im Anhang erläutert.

Mit der Blockierung der Switches, mit denen Publisher verbunden sind, ist es auch möglich das komplette Netz für eine gewisse Zeit zu blockieren. Das ist allerdings wenig wünschenswert. Deswegen werden zusätzlich noch andere Mechanismen genutzt. Für sehr große Transformationen, die nicht disjunktiv aufgeteilt werden können, oder aber auch solche die wirklich alles verändern, ist es nötig diese Möglichkeit zu haben. Dies sollte aber selten bis gar nicht vorkommen. Bei sehr kleinen Netzen könnte dies aber wahrscheinlicher werden, als bei größeren. Wenn nur wenige Publisher in einem Netz existieren und auch eher wenig Kanten vorhanden sind, kann eine Veränderung größer im Verhältnis zum Gesamtnetz ausfallen, als bei einem großen Gesamtnetz.

4.5.2. Zirkel-Mechanismus

Bei diesem Mechanismus sind mindestens zwei Switches beteiligt, die zusammen einen Zirkel bilden. Nachrichten werden hin und her geschickt, bis die Transformation abgeschlossen ist. Dabei kommt ein Markierbit oder aber auch ein VLan-Tag zum Einsatz. Damit soll garantiert werden, dass keine Duplikate entstehen. Das Markierbit oder der VLan-Tag wird von dem weiter von der Transformation entfernten Switch gesetzt. In 4.3 ist dies Switch A und Switch B soll blockiert werden. Diese Nachrichten mit Markierbit/VLan-Tag werden von diesem Switch auch nur noch an den anderen Switch im Zirkel weitergeleitet, also an Switch B. Der andere Switch macht genau das gleiche, er leitet markierte Nachrichten nur an den anderen

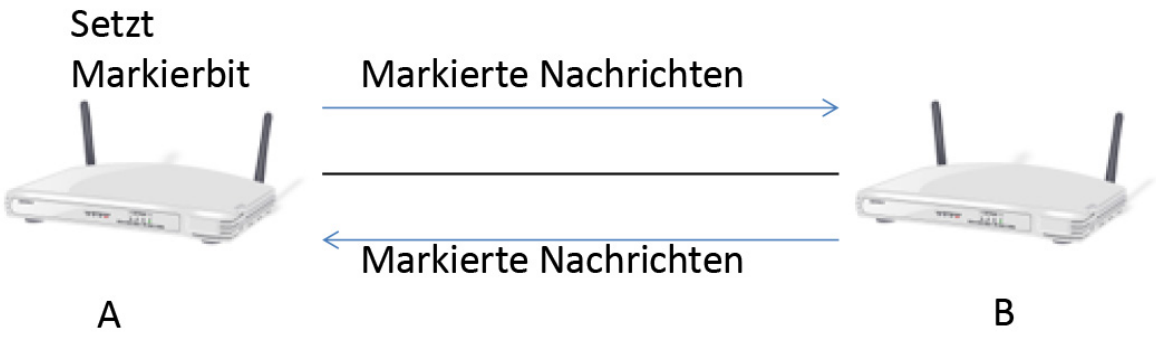

Abbildung 4.3.: Zirkel-Mechanismus mit zwei Switches

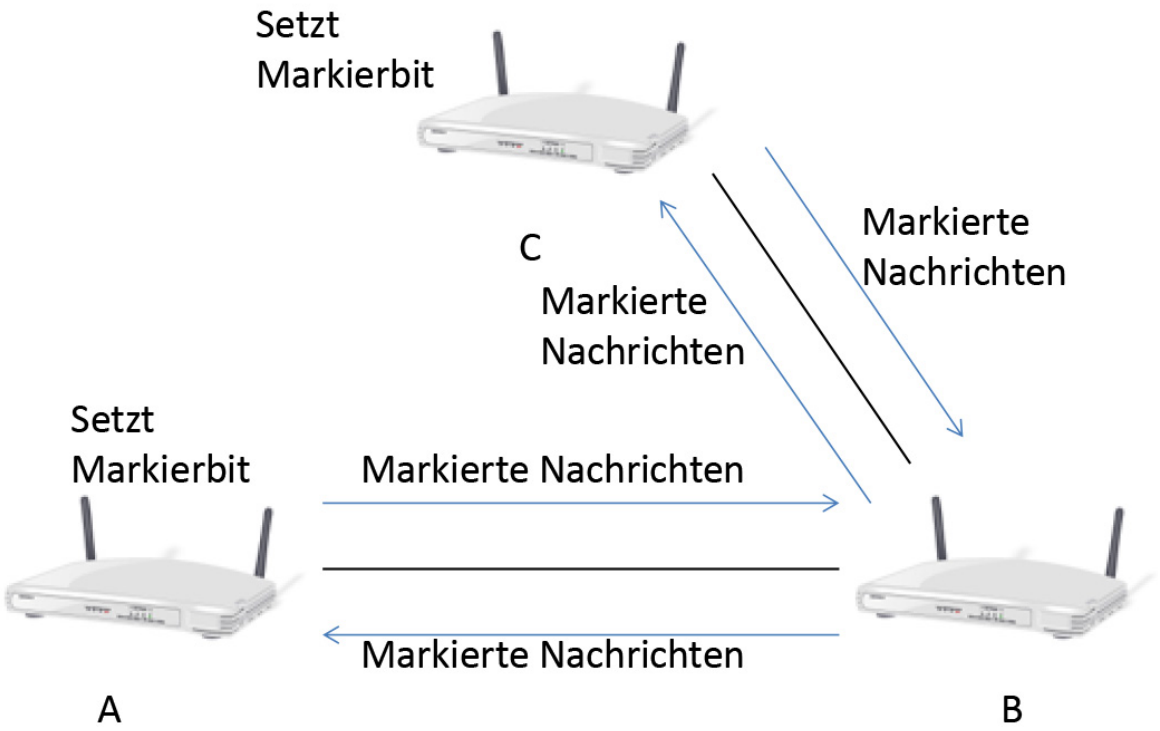

Abbildung 4.4.: Zirkel-Mechanismus mit drei Switches

Switch weiter. B sendet markierte Nachrichten also nur an A. Gleichzeitig leitet er aber noch Nachrichten normal weiter, die von der isolierten Stelle kommen. Damit werden nur die Nachrichten im Zirkel gehalten, die auch über die isolierte Stelle weitergeleitet werden sollen und es wird dafür gesorgt, dass die letzten Nachrichten, die sich noch in der isolierten Stelle befinden, ohne Komplikationen weitergeleitet werden. Dieser Zirkel ist prinzipiell auch mit mehreren Switches möglich und wird auch mehrere Switches benötigen, um eine möglichst kleine isolierte Stelle zu haben. Werden mehrere Switches in den Zirkel miteinbezogen, ist mehr als ein Bit notwendig um die Zugehörigkeit der Nachrichten eindeutig zu halten und Duplikate zu vermeiden. Veranschaulicht wird dies in 4.4. C ist hier der weitere Switch. Bei der Verbindung von C und B wird ein anderes Markierbit eingesetzt, als bei der Verbindung von A und B.

Dieser Zirkel stellt allerdings einen kurzzeitigen Bottleneck dar. Aus dem Grund muss die Transformation schnell ablaufen und es ist sinnvoll nach Switches mit wenig Traffic zu suchen, um den Zirkel an dieser Stelle durchzuführen. Wie groß die Auswirkungen wirklich sind und wie sich das Ganze auf andere Applikationen auswirkt, die das gleiche physikalische Netz nutzen, wird sich in der Evaluation zeigen. Allerdings bietet dieser Mechanismus die Möglichkeit eine lokale Isolation durchzuführen, was den Rest des Netzes nicht beeinflusst. Genauso wird der Controller nicht mit einbezogen und muss sich keinen Bursts aussetzen, die folgen, wenn eine Transformation ausgeführt wird und der Controller als das Ausweichziel für die Nachrichten ausgewählt ist. Der Controller dient aber nach wie vor als Ziel bei Nachrichten, für die keine Regel vorhanden ist. Dies solle aber nie der Fall sein, da zu jedem Zeitpunkt Flows auf den Switches existieren, die Nachrichten weiterleiten.

Beim Setzen der Regeln bei einer Initiierung des Zirkels muss man allerdings aufpassen. Das Setzen der verschiedenen Regeln in der falschen Reihenfolge kann zu Nachrichtenverlust und Duplikaten führen oder auch Traffic in Richtung Controller verursachen. Um dies zu vermeiden wird zuerst die Zirkelregel oder -regeln am zu blockierenden Switch installiert, in 4.3 und 4.4 wäre das B. Danach werden auf A und C die Zirkelregeln und dann die Regeln für das Setzen der Markierbits/VLan-Tags installiert. Die alten Regeln für die Flows bleiben weiterhin bestehen. Allerdings haben sie eine niedrigere Priorität als die Zirkelregeln und werden somit nicht benutzt. Die Regeln müssen weiterbestehen bleiben, damit kein Nachrichtenverlust auftritt, wenn die Zirkelregeln gesetzt oder gelöscht werden. Wären die normalen Regeln zu diesem Zeitpunkt nicht da, kann es sein, dass Nachrichten in genau der Zeitspanne bearbeitet werden, wenn die eine Regel schon gelöscht wurde und die andere noch nicht implementiert wurde. Um dem vorzubeugen, bleiben die normalen Regeln bestehen. Veranschaulicht wird das Ganze in 4.13. Hier werden die drei Schritte für alle Flows und Switches, die blockiert werden sollen, durchgeführt.

Beim Abbau muss man auch eine gewisse Reihenfolge beachten. Dabei wird zuerst auf dem blockierten Switch eine Regel installiert, die das Markierbit oder VLan-Tag entfernt und die Nachrichten in die ehemals isolierte Stelle schickt, also den Zirkel wieder auflöst. Danach wird die Zirkelregel auf diesem Switch gelöscht. Beides ist in 4.14 zu sehen. Im nächsten Schritt wird die Markierregel auf den entfernteren Switches, A und C, gelöscht. Damit werden keine neuen Nachrichten mehr markiert und kommen auch nicht in den Zirkel. Nach

Algorithmus 4.13 Füge Zirkel hinzu

```
procedure FÜGE ZIRKEL HINZU(Switch, Flow)
    for all Zirkelswitches do          // Zirkelswitches sind Switches mit denen der
                                          blockierte Switch einen Zirkel formt
        for all Flows auf Zirkelswitch do
            Füge Match auf Markierung hinzu
            ADDFLOW(Flow-Name, Switch, Flow)          // Funktion vom Controller
                                      // Die Flows werden so spezifiziert, dass sie ihre
                                          Nachrichten auf den Port leiten, von dem sie
                                          die Nachricht bekommen haben
        end for
    end for
    for all Zirkelswitches do
        for all Flows auf Zirkelswitch do
            Füge Match auf Markierung hinzu
            ADDFLOW(Flow-Name, Zirkelswitch, Flow)
        end for
    end for
    for all Zirkelswitches do
        for all Flows auf Zirkelswitch do
            Füge Markiersmechanismus hinzu
                                  // Die Nachrichten werden durch diesen Flow markiert
            ADDFLOW(Flow-Name, Zirkelswitch, Flow)
        end for
    end for
end procedure
```

einer kurzen Wartezeit werden dann sowohl die Zirkelregeln auf den entfernteren Switches entfernt, als auch der Flow, der die Markierentfernung auf dem nahen Switch vornahm. Sichtbar ist dies in 4.15.

4.5.3. Storagelösung

Diese Lösung ähnelt den beiden anderen sehr und sieht auf den ersten Blick auch vielversprechend und einfach aus. Sobald eine Isolation stattfindet, werden die Nachrichten einfach an den Storage weitergeleitet und dort solange gespeichert, bis die Transformation abgeschlossen ist (4.5). Dieser Storage ist speziell für den Ansturm an Nachrichten ausgelegt, was bei einer solchen Möglichkeit stattfinden wird. Nachdem die Transformation abgeschlossen ist, werden die Nachrichten wieder in Umlauf gebracht, indem der Storage als Publisher agiert (4.6). Das ganze kann leicht vom Controller gesteuert werden, da nur eine Regel im Switch geändert werden muss, die besagt was passiert, wenn ein Miss stattfindet. Die anderen Regeln können gelöscht werden oder einfach eine niedrigere Priorität besitzen. Das Verbreiten der Nachrichten wird dann dadurch erreicht, dass die Regeln wieder gesetzt

Algorithmus 4.14 Entferne Zirkel

```
procedure ENTFERNE ZIRKLE(Switch, Flow)
    for all Zirkelswitches do          // Zirkelswitches sind Switches mit denen der
                                          blockierte Switch einen Zirkel formt
        for all Flows auf Zirkelswitch do
            Füge Markierentferner hinzu
            ADDFLOW(Flow-Name, Switch, Flow)          // Funktion vom Controller
        end for
    end for
    for all Zirkelswitches do
        for all Flows auf Zirkelswitch do
                                  // Entfernen des Zirkelflows auf dem blockierten Switch
            DELETEFLOW(Flow-Name)                     // Funktion vom Controller
        end for
    end for
    for all Zirkelswitches do
        for all Flows auf Zirkelswitch do
                                  // Löschen des Flows, der die Markierung übernimmt
            DELETEFLOW(Flow-Name)
        end for
    end for
end procedure
```

Algorithmus 4.15 Entferne Markierentferner

```
procedure ENTFERNE MARKIERENTFERNER(Switch, Flow)
    for all Zirkelswitches do
        for all Flows auf Zirkelswitch do          // Entfernen des Zirkelflows
            DELETEFLOW(Flow-Name)                  // Funktion vom Controller
        end for
    end for
    for all Zirkelswitches do          // Zirkelswitches sind Switches mit denen der
                                          blockierte Switch einen Zirkel formt
        for all Flows auf Zirkelswitch do          // Entferne Markierentferner
            DELETEFLOW(Flow-Name)
        end for
    end for
end procedure
```

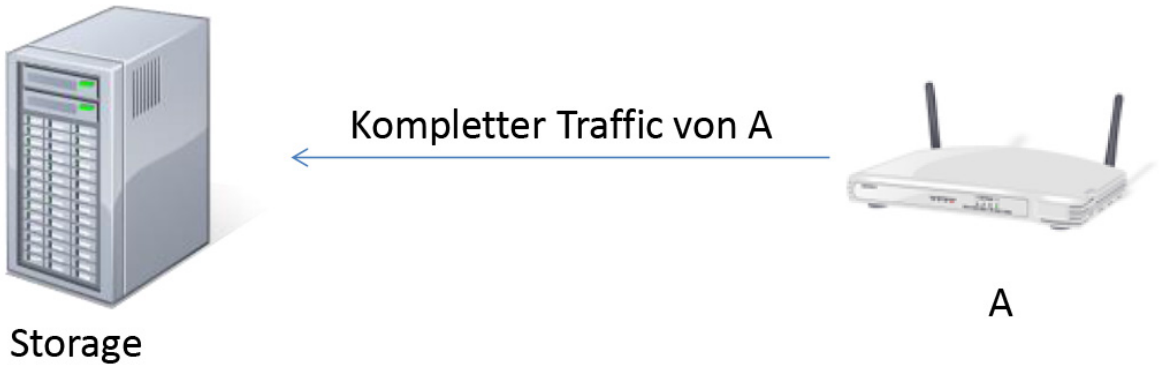

Abbildung 4.5.: Erste Phase des Storage: Alle Nachrichten werden gespeichert

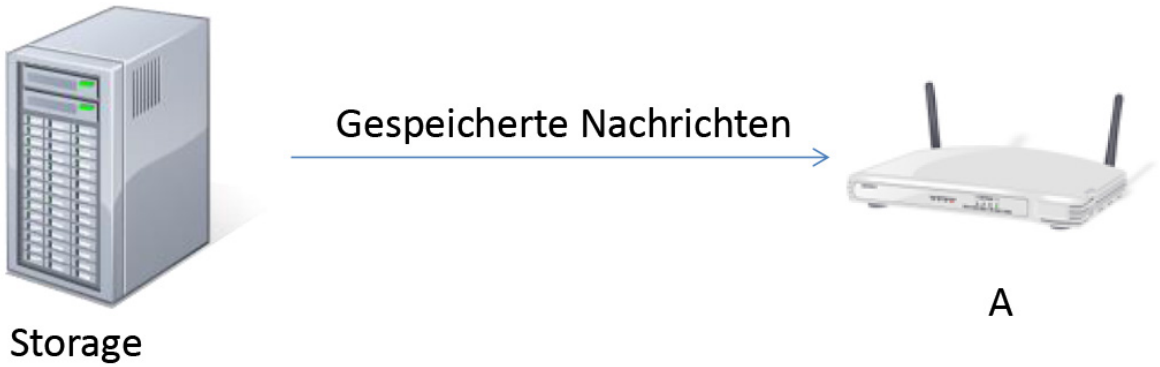

Abbildung 4.6.: Zweite Phase des Storage: Gespeicherte Nachrichten werden wieder verteilt

werden, und der Storage ein Signal bekommt, dass eine Art Publisher-Routine startet und die Nachrichten versendet.

Diese Lösung beinhaltet aber ein Problem. Man muss sehr genau planen, wo man Storage platziert, denn dieser muss sehr nah an Switches sein und am besten direkt erreichbar, ohne über andere Switches zu gehen. Bei einem skalierbaren Netz ist das eine recht starke Einschränkung. Wenn es möglich wäre den Storage dynamisch zu platzieren und auch dynamisch mit Switches zu verbinden, dann wäre dies durchaus eine starke Alternative. Dafür wäre es aber nötig, dass insgesamt sehr viel Storage existiert und man diesen auch beanspruchen kann wenn es nötig ist. Wenn er nicht benötigt wird, kann er für andere Applikationen verwendet werden. Das scheint realisierbar zu sein, geht aber etwas zu weit für diese Arbeit. Deswegen wird diese Lösung zwar mit aufgeführt, aber vorerst nicht berücksichtigt bzw. als eine Art Zusatz gesehen. Es ist aber durchaus denkbar, dass diese Lösung in Zukunft mit implementiert wird und dadurch dem Netz mehr Entlastung gönnt.

5. Evaluierung

Im Folgenden sollen die Mechanismen nun evaluiert werden. Bevor die Resultate vorgestellt werden, wird aber zuerst noch das Testsystem vorgestellt und der Aufbau der Tests. Die Implementierung ist in Java erfolgt, da der Controller schon in Java geschrieben wurde.

5.1. Testsystem

Als Testsystem wurde zum einen Mininet und zum anderen ein Testbed genutzt. Bei Mininet handelt es sich um eine virtuelle Umgebung von Switches, Hosts und Verbindungen, die man frei konfigurieren kann. Dieses System wurde allerdings nur zum Verifizieren genutzt und nicht für die aktuellen Tests, da die Leistung von Mininet dafür nicht ausreichend war. Bei dem Testsbed handelt es sich um virtuelle Switches, die untereinander verbunden sind. Sie laufen jeweils auf einem Rechner und sind alle über einen Switch verbunden. An vier der Switches hängen jeweils zwei Hosts, die sowohl als Publisher, als auch als Subscriber dienen können. Dadurch, dass es virtuelle Switches sind, benötigen sie beim Routen weitaus länger als echte Switches. Die Größenordnung der Zeit für das Routen der virtuellen Switches liegt bei ca. 300 µs. Bei echten Switches liegt diese Zeit im Nanosekundenbereich. Zusätzlich ist anzumerken, dass in dem Testbed das Testen mit VLan-Tags nicht möglich war. Der Aufbau über den echten Switch ermöglicht es zwar leicht die Topologie zu ändern, allerdings nutzt er dafür selber VLan-Tags, womit Nachrichten mit VLan-Tags verloren gehen. Somit wurde auf die MAC-Adresse zum Markieren ausgewichen.

Aufgebaut ist das Testbed wie eine Fat-Tree-Topologie. Der Aufbau wird in 5.1 gezeigt. Dabei sind die oberen beiden Switches mit allen darunter liegenden verbunden. Die anderen acht Switches bilden mit den Hosts als Blättern zwei Unterbäume. Jeweils vier Switches sind dabei miteinander verbunden. An jeweils zwei von diesen Switches befinden sich die Hosts.

5.2. Testaufbau

Getestet wurden der NAK-Mechanismus, der Zirkel-Mechanismus und der Scheduler. Der NAK- und Zirkel-Mechanismus wurden auf dem Testbed getestet. Der Scheduler wurde auf einem normalen Computer und nicht auf dem Testbed getestet. Bei diesen Tests ging es darum, wie viel Zeit für welche Größe von Graphen benötigt wurde. Dabei wurden die Flows nicht gesetzt, sondern nur berechnet.

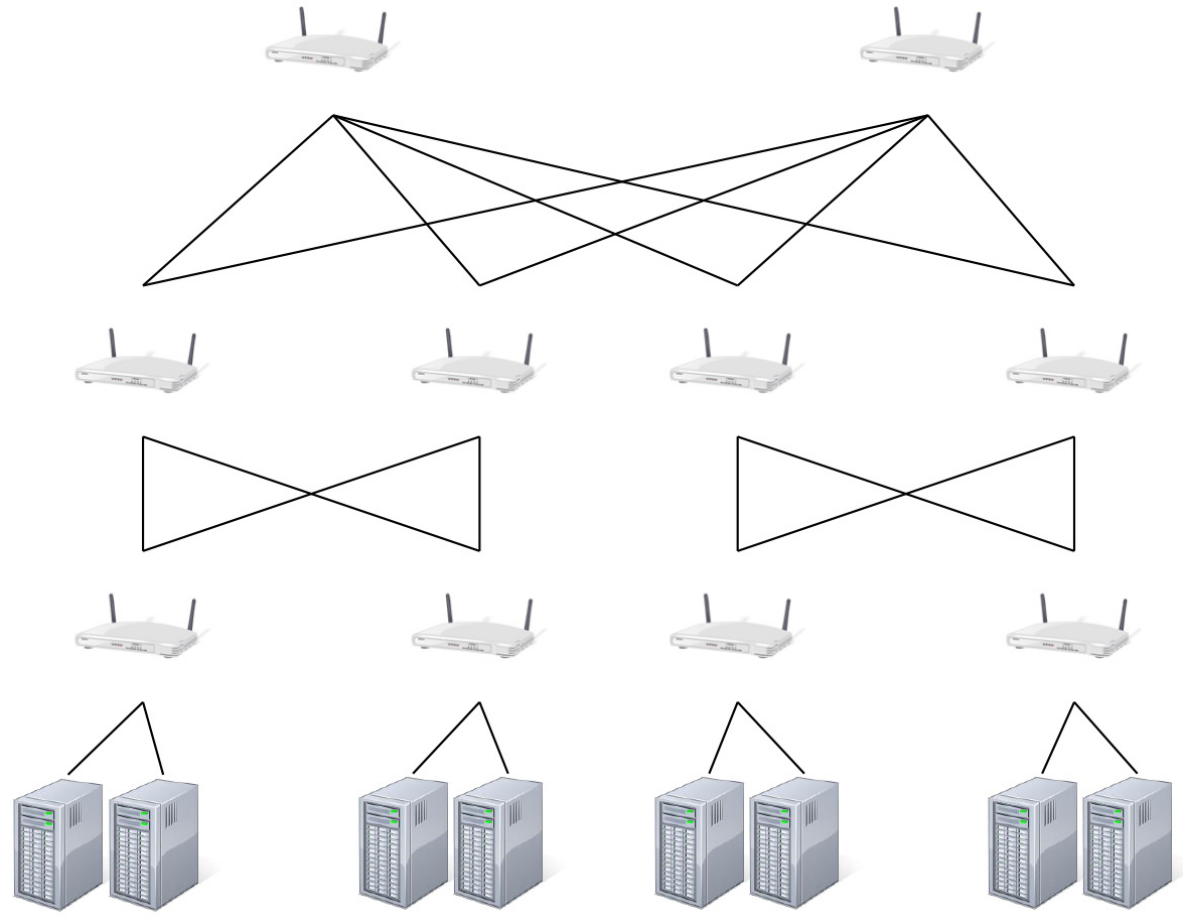

Abbildung 5.1.: Topologie des Testsystems

Beim NAK- und Zirkel-Mechanismus wurden im Testbed zwei Hosts ausgewählt, einer als Publisher und der andere als Subscriber. Zwischen diesen beiden wurde ein Pfad gewählt, der leicht geändert wurde, um den jeweiligen Mechanismus auszuführen. Dieser wurde dann wieder auf den Urpsrungspfad geändert und wieder zurück. Das ganze wurde ca. 60 Sekunden lang gemacht, wobei es zum einen Unterschiede in der Häufigkeit der Änderung gab, als auch in der Sendegeschwindigkeit. Gemessen wurde die Verzögerung zwischen senden und empfangen der Nachrichten. Dabei wurde auf Millisekunden genau getestet, was ausreicht, um die Auswirkung der Mechanismen zu veranschaulichen. In jede Nachricht wurde zusätzlich noch ein Zähler geschrieben, um festzustellen, ob Nachrichten verloren gingen oder mehrfach ankamen. Für den Zirkel-Mechanismus befanden sich auf dem Pfad zwei Switches mehr, als auf dem Pfad des NAK-Mechanismus. Dies wirkt sich allerdings nur unwesentlich auf die Verzögerung aus.

Die Zeiten für die Änderung des Pfades liegen bei 5, 2.5, 1, 0.5 und 0.1 Sekunden. Die Geschwindigkeiten für das Senden der Nachrichten belaufen sich auf 2, 1, 0.5, 0.1 Millise-

kunden. Wobei es recht große Abweichungen gab, wie viele Nachrichten gesendet wurden, je höher die Geschwindigkeit war. Die reale Geschwindigkeit dürfte deshalb etwas niedriger liegen, gerade bei 0.1 Millisekunden. Das liegt hauptsächlich am Aufbau des Senders, der sehr einfach gestaltet ist. Auch der Widersendemechanismus beim Sender, für Nachrichten die zurückkommen, ist sehr einfach gestaltet und kann mit Sicherheit noch optimiert werden. Allerdings hat der einfache Aufbau zum Testen vollkommen ausgereicht.

Wichtig zu erwähnen ist noch, dass für die Tests Multicast-, nicht Unicast-Adressen genutzt wurden. Damit sind zwar auch einige Probleme verbunden, die im Anhang erklärt werden, allerdings hat es auch weniger Restriktionen als Unicast.

5.3. Resultate

Die verschiedenen Settings wurden für beide Mechanismen getestet und jeweils graphisch dargestellt. Das wichtigste für die Tests war die Garantie, dass mit den beiden Mechanismen weder Nachrichten verloren gehen, noch Duplikate entstehen. Beide Garantien konnten gehalten werden. Allerdings kommt dies zu einem gewissen Preis. Wie hoch dieser Preis ist, kann man schon ohne Tests etwas abschätzen. Durch die Wartezeit beim Update-Mechanismus zwischen dem Setzen der verschiedenen Regeln entsteht mindestens eine Verzögerung von 20 ms. Dazu kommt dann noch die normale Verzögerung auf dem Pfad hinzu. Diese lag bei ca. 0-4 ms. Wir haben also während der Mechanismus aktiv ist eine Verzögerung von mindestens 20-24 ms. Wie hoch die Verzögerung in Realität liegt, werden die Ergebnisse zeigen. Anfänglich ist die Verzögerung der Nachrichten recht hoch, das sind allerdings nur die ersten 10-20 Nachrichten. Da dies sowohl in Mininet, als auch im Testbed beobachtet wurde, könnte es ein Problem der Virtualisierung der Switches sein. Das ist aber nicht weiter relevant, sollte aber dennoch erwähnt werden. Die normale Verzögerung wird nach diesen Nachrichten erreicht.

Im Folgenden werden allerdings keine Tests für die Geschwindigkeit mit Senden alle 0.1 ms vorgestellt. Das liegt daran, dass hier die Garantien nicht mehr gehalten werden konnten. Wie vorhin beim Testsystem erwähnt, benötigt der virtuelle Switch ungefähr 300 µs um eine Nachricht zu bearbeiten und weiterzuleiten. Werden die Flows nur einmal gesetzt und danach nie wieder verändert, also keiner der Mechanismen angewandt, dann gehen keine Nachrichten verloren. Das deutet darauf hin, dass die Geschwindigkeit nicht ganz eingehalten wird und die Zeit wohl eher bei 0.3 ms oder etwas höher liegt. Ansonsten müssten schon hier Nachrichten verloren gehen, da die Queues der Switches volllaufen und dann Pakete fallen gelassen werden. Da dies nicht der Fall ist, liegt die Sendegeschwindigkeit niedriger als angegeben. Finden jetzt allerdings Veränderungen statt und wird einer der Mechanismen genutzt, kommt es zu Nachrichtenverlust. Da die Nachrichten allerdings nicht beim Controller ankommen, kann man davon ausgehen, dass sie fallen gelassen worden sind. Das heißt wiederum, dass die Queue vollgelaufen ist. Wenn wir die beiden Mechanismen anschauen, kann das sehr schnell passieren. Wenn man so schon relativ nah am Maximum der Switches ist, dann kommt es sowohl beim Zirkel- als auch beim NAK-Mechanismus zu Verlust, da mehr Nachrichten als normal beim Switch eintreffen. Beim Zirkel sind es

die normalen, die immer weiter ankommen und die Nachrichten im Zirkel, die zusätzlich noch da sind. Beim NAK sind es die normalen Nachrichten und die Nachrichten die erneut gesendet werden, weil sie zurückkamen. In beiden Fällen werden Nachrichten fallen gelassen, sobald man die Kapazität des Switches übersteigt. Da das aber nicht passieren darf, sollte man beim Testbed eine Untergrenze der Geschwindigkeit bei 0.5 ms sehen, damit man noch etwas Spielraum hat.

Zudem kam es manchmal dazu, dass trotzdem Nachrichten verloren gingen, wobei hier speziell beim Zirkel-Mechanismus. Dies scheint aber nicht dem Mechanismus geschuldet, sondern eine Eigenheit des Testbeds zu sein. Wie man auch in 5.9 oder 5.2 sehen kann, scheint der Switch manchmal kurze Verzögerungen beim Weiterleiten zu haben. In 5.9 sieht man dies zwischen den großen Verzögerungen beim Zirkel, in 5.2 zwischen den NAKs. Hier gehen die Verzögerungen dazwischen auf bis zu 3-4 Millisekunden höher als normal. Beim Auftreten des Zirkels kann es nun passieren, dass wir uns in der Phase befinden, in der die Zirkelregel auf dem weiter entfernten Switch noch vorhanden ist und die Demarkierregel auf dem ehemals blockierten Switch. Die anderen Regeln sind nicht mehr vorhanden. Wenn jetzt eine Verzögerung beim ehemals blockierten Switch auftritt, kann die Zirkelregel auf dem entfernteren Switch schon gelöscht sein, bevor die Nachrichten dann ankommen. Da bei den normalen Flows auch auf den Eingangsport gematcht wurde, landen die Nachrichten beim Controller, da es kein Match gibt. Zum einen lässt sich das umgehen in dem man nicht mehr auf den Eingangsport bei normalen Flows matcht, zum anderen kann man es mit längeren Wartezeiten lösen, bevor die Zirkelregeln gelöscht werden. Beides kann Probleme mitbringen. Das erste kann bei der Konfiguration des Graphen Probleme machen, das zweite bei der Leistung. Diese Verzögerungen sollten allerdings bei realen Switches nicht vorkommen, deswegen muss man sich die Frage stellen, ob es überhaupt notwendig ist dies für reale Switches zu ändern.

5.3.1. NAK-Mechanismus

Bevor wir uns die Ergebnisse genauer anschauen, noch einmal kurz die Erinnerung was der Algorithmus macht. Ein Switch wird mit dem NAK-Mechanismus blockiert, indem die Nachrichten an ihn wieder zurückgesendet werden. Der Sender empfängt diese Nachrichten und versucht sie erneut zu senden. Im Sender für die Tests wurde 5 ms gewartet, nachdem die erste Nachricht zurückkam, bevor erneut gesendet wurde. Damit erhöht man die Last auf dem Switch zwar etwas, aber sollte recht gute Ergebnisse in der Verzögerung erreichen. Gewartet wird, da man weiß, dass die Blockierung eine gewisse Zeit besteht.

Es können leider nicht alle Ergebnisse von den verschiedenen Tests vorgestellt werden, deswegen werden exemplarisch die Tests für Änderungen alle 5 und 0.1 Sekunden und Geschwindigkeit von 2 bzw. 0.5 Millisekunden gewählt. In 5.2 sieht man die Auswirkungen des NAK-Mechanismus am besten. Die zwei dicken Linien im unteren Millisekundenbereich spiegeln die normale Verzögerung wieder. Diese liegt bei ca. 0 - 4 ms. Sobald ein NAK in Kraft tritt schießt diese Verzögerung in die Höhe. Interessanterweise weitaus höher als angenommen. Das kann zum einen von der zusätzlichen Last auf dem Switch, an dem der Sender angeschlossen ist, kommen, zum anderen kann man aber auch nicht

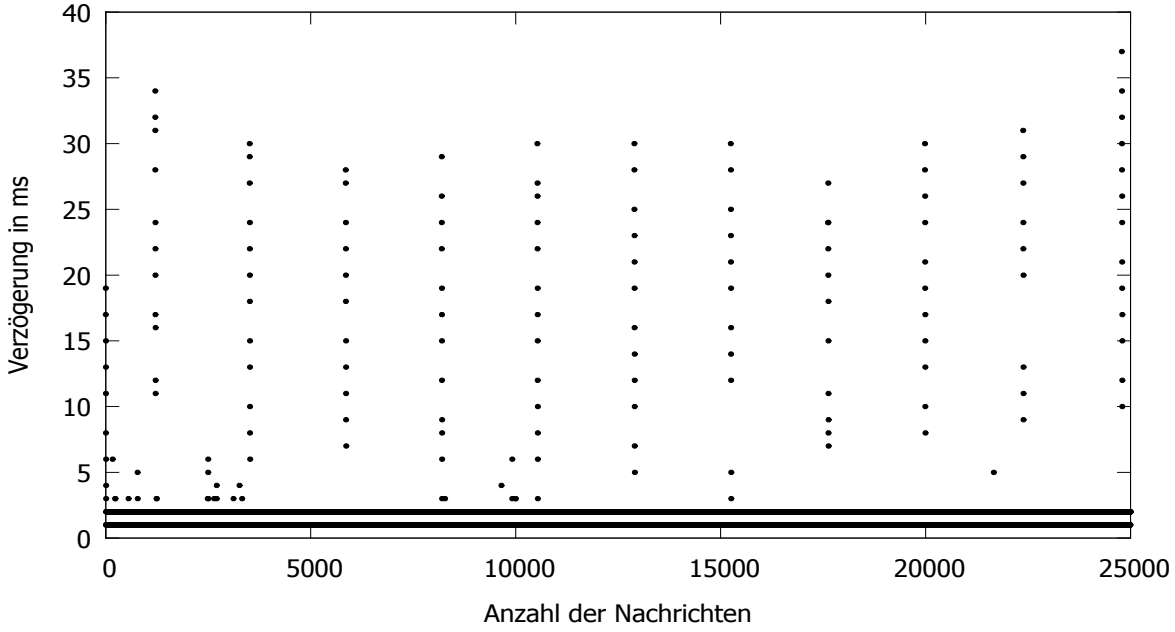

Abbildung 5.2.: NAK-Mechanismus mit Änderungen alle 5 Sekunden und Nachrichten alle 2 Millisekunden

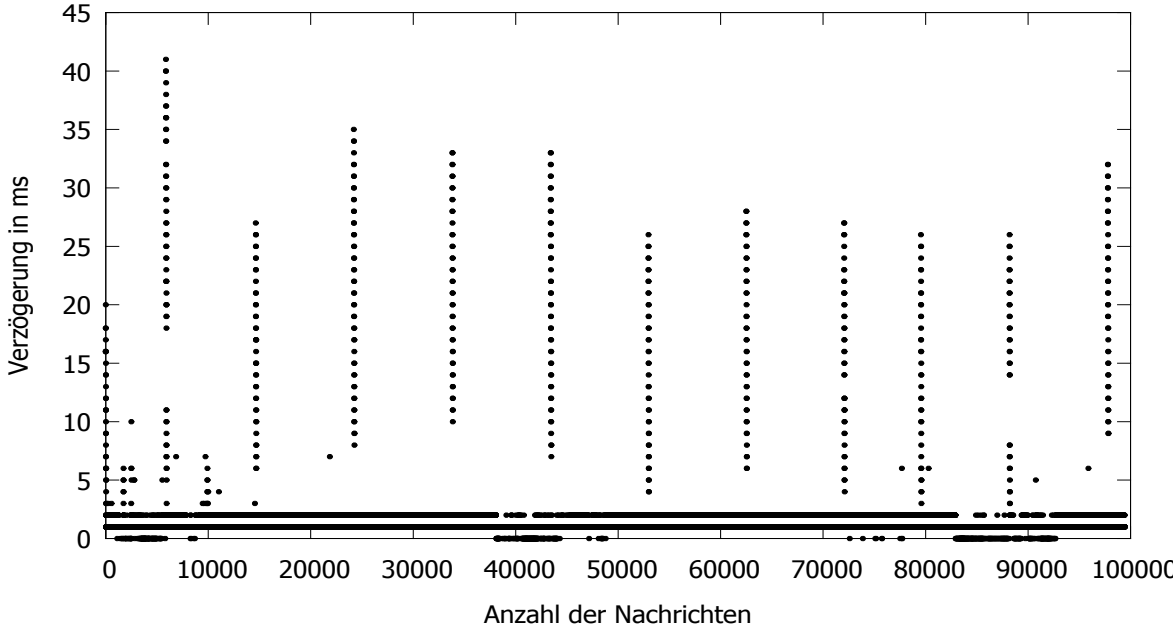

Abbildung 5.3.: NAK-Mechanismus mit Änderungen alle 5 Sekunden und Nachrichten alle 0.5 Millisekunden

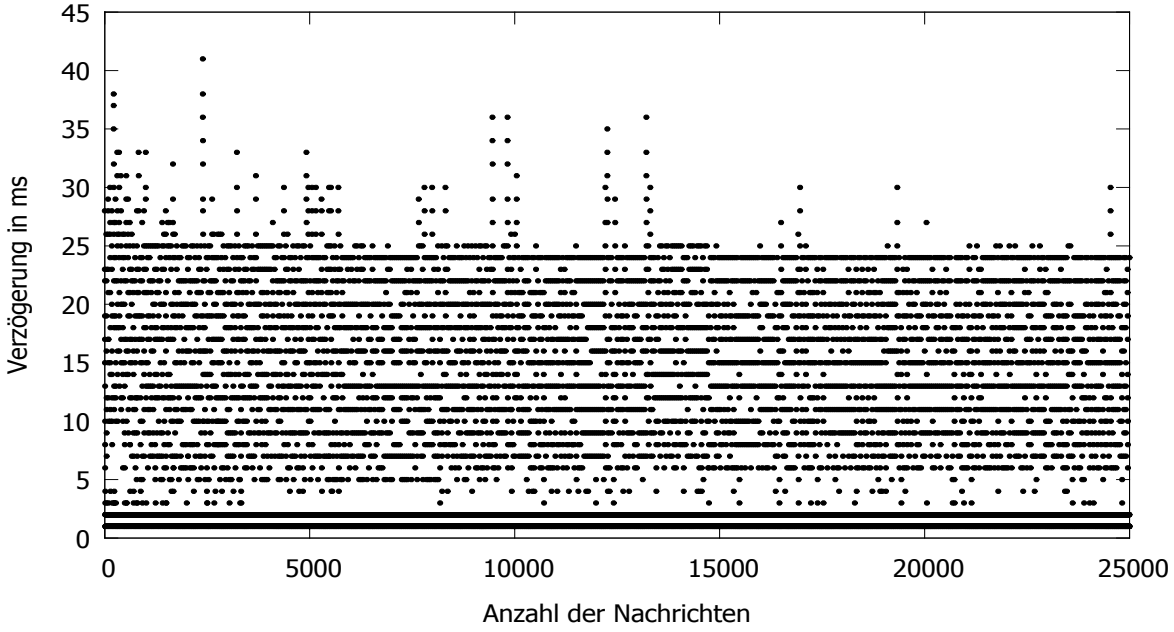

Abbildung 5.4.: NAK-Mechanismus mit Änderungen alle 0.1 Sekunden und Nachrichten alle 2 Millisekunden

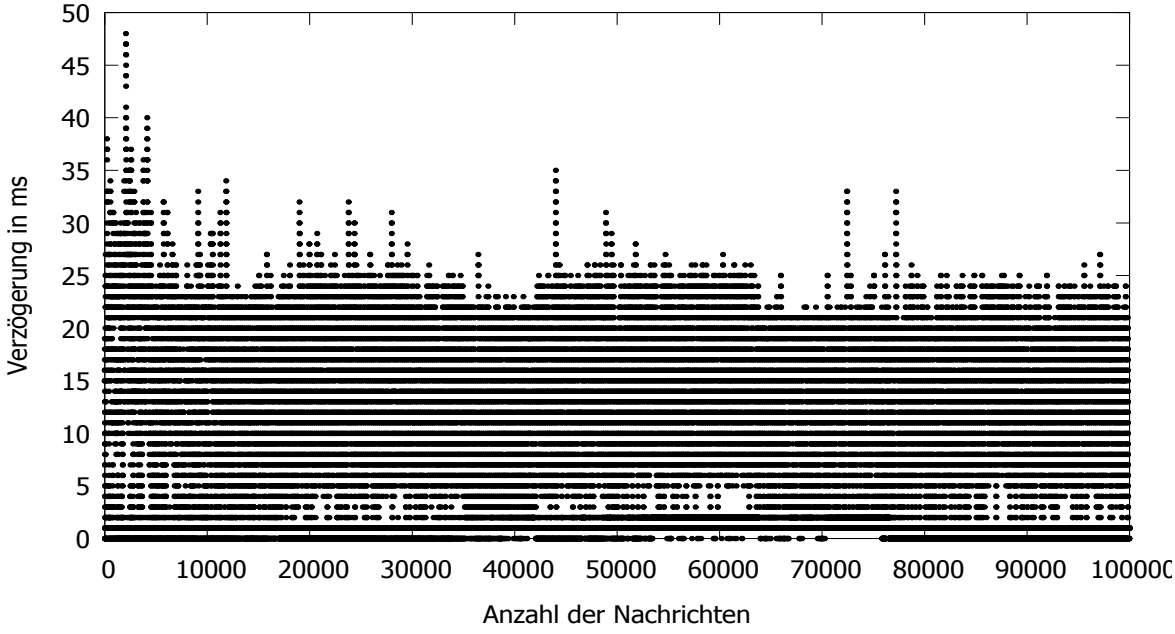

Abbildung 5.5.: NAK-Mechanismus mit Änderungen alle 0.1 Sekunden und Nachrichten alle 0.5 Millisekunden

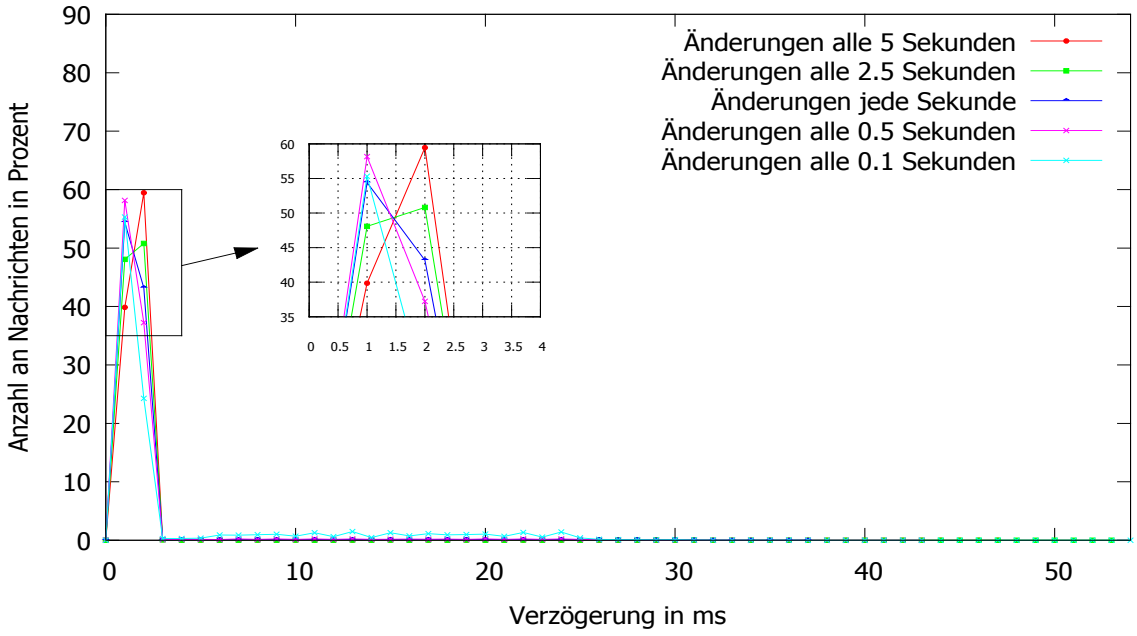

Abbildung 5.6.: NAK: Verteilung der Nachrichten bei Sendegeschwindigkeit von 2 ms

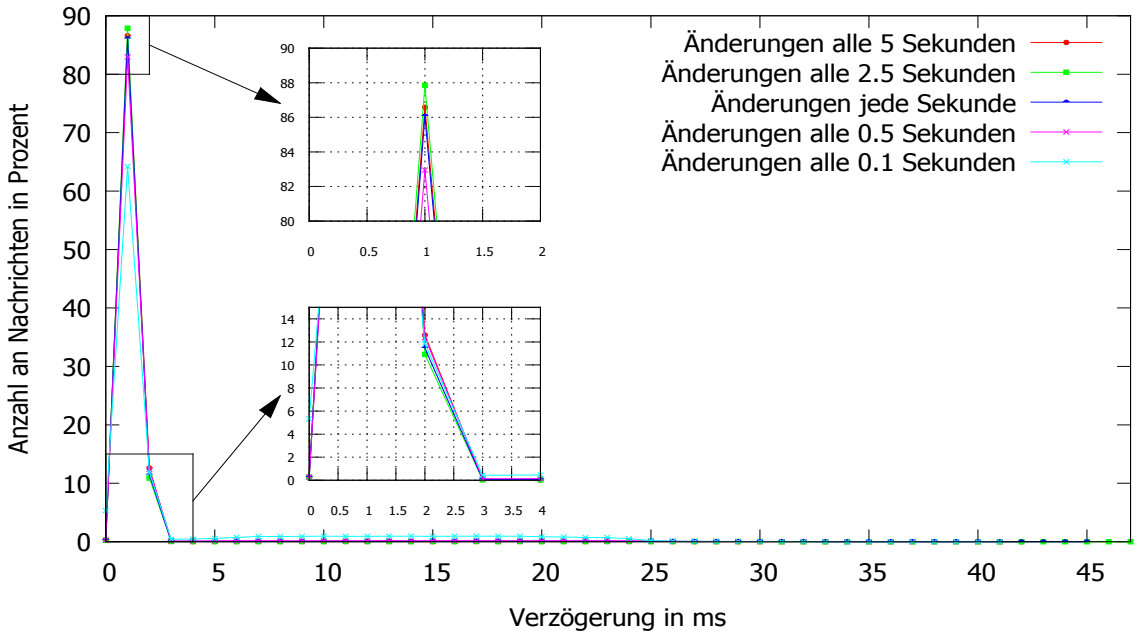

Abbildung 5.7.: NAK: Verteilung der Nachrichten bei Sendegeschwindigkeit von 1 ms

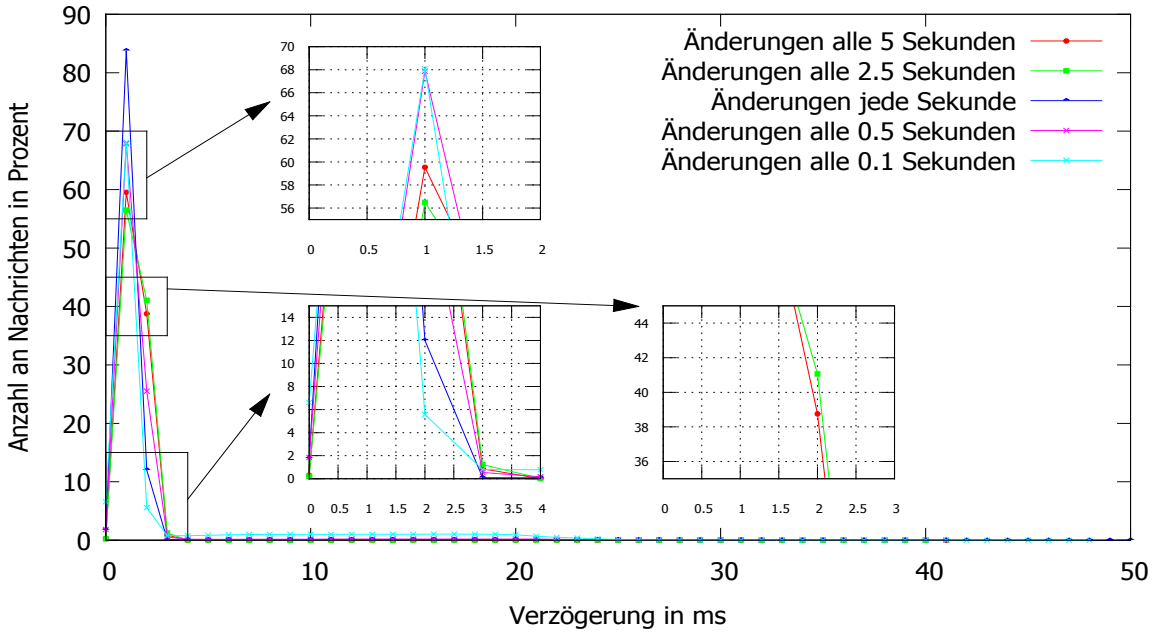

Abbildung 5.8.: NAK: Verteilung der Nachrichten bei Sendegeschwindigkeit von 0.5 ms

ausschließen, dass das Testbed einen Teil der Verzögerung verursacht. Zum Beispiel bei ca. 2500 Nachrichten, kurz bevor der zweite NAK in Kraft tritt, gibt es eine erhöhte Verzögerung der Nachrichten, obwohl kein ersichtlicher Grund vorliegt.

Sowohl in 5.2, als auch in 5.3 sehen wir, dass die Nachrichten bei einem NAK nicht mit konstant erhöhter Verzögerung eintreffen. Das kommt in der Regel von der kurzen Wartezeit beim Wiedersenden. Allerdings gibt es auch Fälle in denen die Verzögerung konstant steigt und im Endeffekt auch im Maximalwert nicht so hoch liegt.

Wenn man sich nun 5.4 und 5.5 anschaut könnte man den Eindruck gewinnen, dass bei schnelleren Änderungen sehr viel mehr Nachrichten länger als die normale Verzögerung brauchen. Dies ist in der Tat der Fall, aber bei weitem nicht so extrem wie es aussieht. Sind bei Änderungen alle 5 Sekunden in 5.6 die Nachrichten mit normaler Verzögerung noch bei fast 100%, sinkt diese Zahl auf ca. 80% ab bei Veränderungen alle 0.1 Sekunden. Allerdings ist selbst bei Veränderungen alle 0.5 Sekunden diese Zahl noch bei ca. 95%. Der recht große Unterschied hier dürfte an dem Verhältnis zwischen blockierter Zeit und nicht blockierter Zeit liegen. Wenn die blockierte Zeit bei ca. 20 ms liegt und die nicht blockierte bei 80 ms, dann ist das relativ wenig um die Verzögerung wieder zu normalisieren. Denn die Nachrichten, die während der 20 ms gesendet wurden, müssen während der 80 ms zusätzlich übertragen werden. Bei 0.5 Sekunden ist das Verhältnis von blockierter Zeit zu nicht blockierter Zeit weitaus höher. Die Nachrichten, die während der 20 ms gesendet wurden, fallen hier nicht so sehr ins Gewicht. Je höher die Zeit der Veränderung liegt, desto

höher ist natürlich das Verhältnis zwischen blockierter und nicht blockierter Zeit. Allerdings nimmt dieses Verhältnis weniger stark zu, als bei 0.1 und 0.5 Sekunden.

Ähnliches ist beim Senden jede Millisekunde zu beobachten. Wobei hier bei jeder Veränderung, egal welcher Zeit, der Prozentsatz der Nachrichten, die während der normalen Verzögerung ankommen, sinkt. Allerdings ist diese Zahl recht gering. Prinzipiell ist das gleiche Verhalten zu sehen, wie schon beim Senden alle 2 Millisekunden. Je häufiger geändert wird, desto weniger Nachrichten kommen während der normalen Verzögerungszeit an. Auch hier ist bei Änderungen alle 0.1 Sekunden wieder der geringste Wert zu sehen.

Selbst beim Senden alle 0.5 Millisekunden ist der Prozentsatz der Nachrichten mit normaler Verzögerung bei Änderungen alle 0.1 Sekunden noch bei ca. 78%. Bei Änderungen alle 5 Sekunden liegt dieser Wert noch bei ca. 99%. Die Geschwindigkeit des Sendens der Nachrichten scheint nicht eine allzu große Rolle zu spielen, wenn es darum geht, wie viele Nachrichten im Bereich der normalen Verzögerung ankommen. Was eine sehr viel größere Rolle spielt ist die Zeit zwischen Änderungen. Wenn man bedenkt, dass man selbst bei Änderungen alle 0.5 Sekunden und bei Nachrichten alle 0.5 Millisekunden noch fast 95% beim Empfangen im normalen Verzögerungsbereich erreicht und dafür keine Nachrichten verliert oder Duplikate hat, klingt das nach einem recht guten Kompromiss. Vorausgesetzt man muss seinen Graphen an derselben Stelle nicht öfter ändern.

5.3.2. Zirkel-Mechanismus

Auch hier, bevor wir uns die Ergebnisse anschauen, eine kleine Auffrischung über die Funktionsweise des Algorithmus. An einem Zirkel sind zwei Switches beteiligt. Auf dem einen wird ein Flow installiert, der Nachrichten markiert, und einer der Nachrichten im Zirkel weiterleitet. Auf dem anderen nur einer, der Nachrichten im Zirkel weiterleitet. Beim Auflösen wird ein Flow installiert, der die Markierung entfernt und normal weiterleitet.

Für den Zirkel wurden vorhin schon Überlegungen angestellt wie hoch ungefähr die Verzögerung beim Transformieren mit einem Zirkel sein muss. Auch hier sehen wir wieder eine höhere maximale Verzögerung als angenommen. Allerdings gibt es keine Lücken wie beim NAK-Mechanismus bei den Verzögerungen wenn ein Zirkel existiert. Die normale Verzögerung kann hier etwas höher liegen, als beim anderen Test, da mehr Switches auf dem Wege lagen. Das macht ungefähr eine Millisekunde aus, welches wir in 5.10 ab einem Viertel und in 5.9 komplett durch sehen. Mögliche Unterschiede sind hier der Messungenauigkeit geschuldet, da nur in Millisekunden gemessen wurde.

Die prinzipielle Struktur ist dem vom NAK-Mechanismus ähnlich. Sobald der Mechanismus greift, erhöht sich die Verzögerung für einen kurzen Moment und fällt danach wieder ab. Die maximale Verzögerung kommt von den Nachrichten, die am Anfang des Mechanismus gesendet wurden. Diese werden am längsten aufgehalten. Je später die Nachricht während des Mechanismus gesendet wurde, desto kürzer braucht sie. Auch hier haben wir wieder ein paar Nachrichten, die eine längere Verzögerung als die normale haben und die nicht zur Zeit einer Veränderung liegt. Wie schon erwähnt, dürfte das durch den Aufbau des Testbeds kommen, da dies virtuelle Switches und keine realen sind.

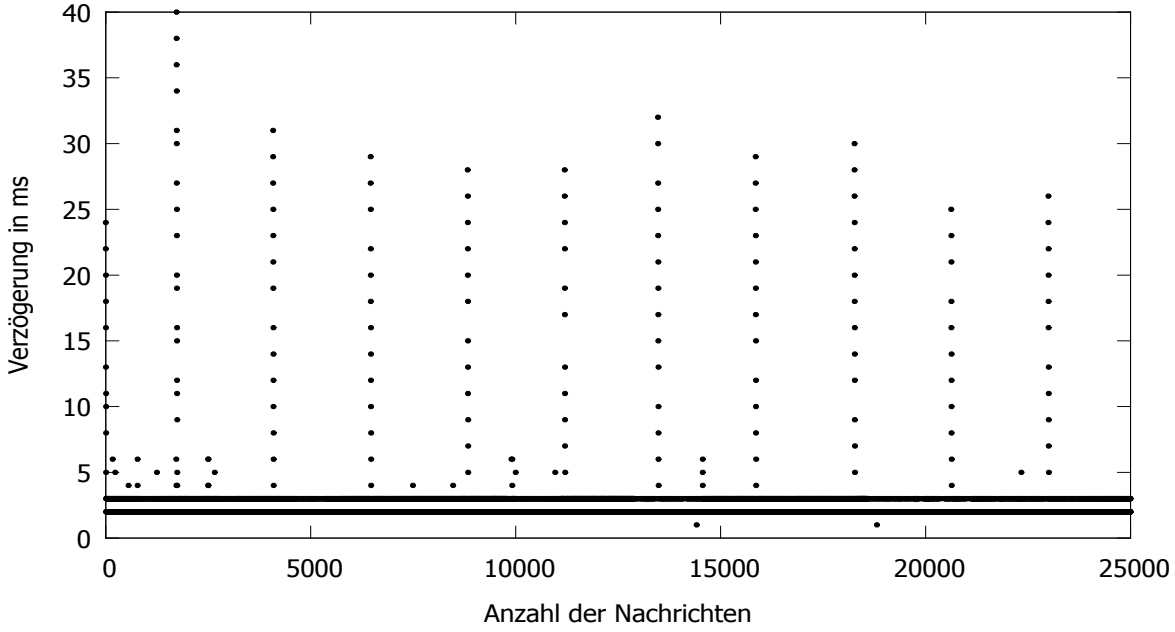

Abbildung 5.9.: Zirkel-Mechanismus mit Änderungen alle 5 Sekunden und Nachrichten alle 2 Millisekunden

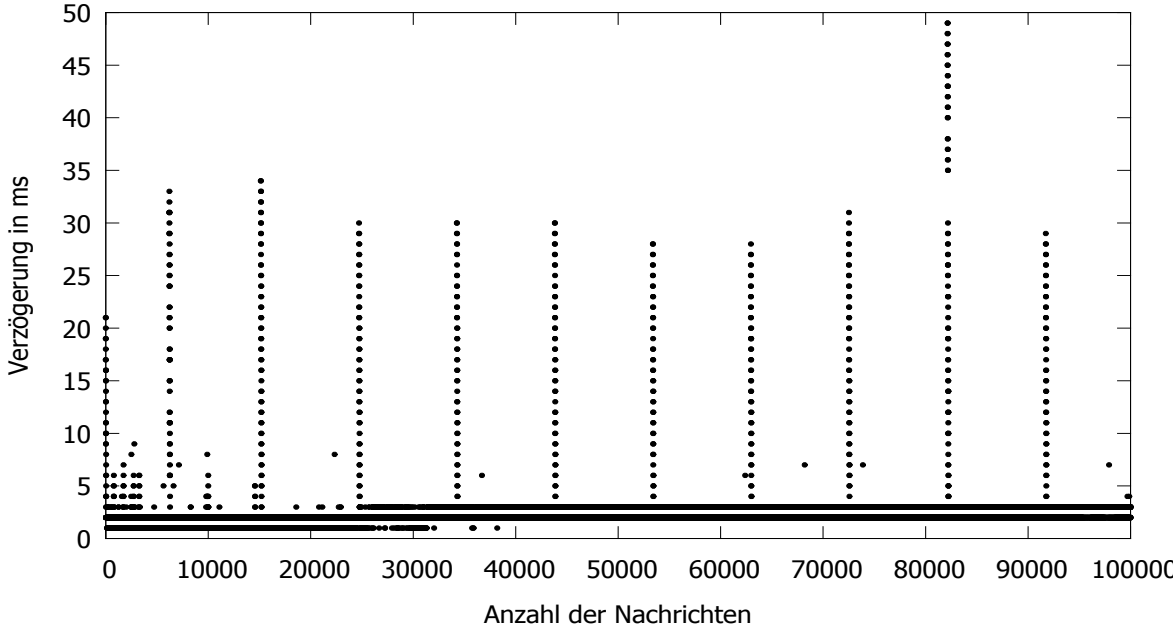

Abbildung 5.10.: Zirkel-Mechanismus mit Änderungen alle 5 Sekunden und Nachrichten alle 0.5 Millisekunden

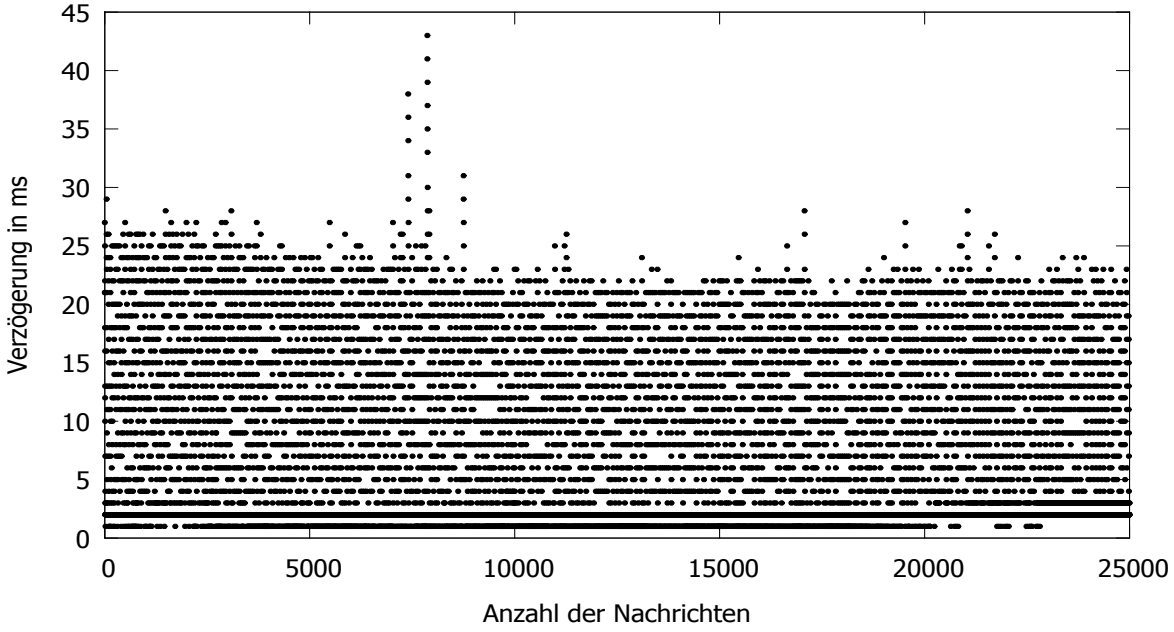

Abbildung 5.11.: Zirkel-Mechanismus mit Änderungen alle 0.1 Sekunden und Nachrichten alle 2 Millisekunden

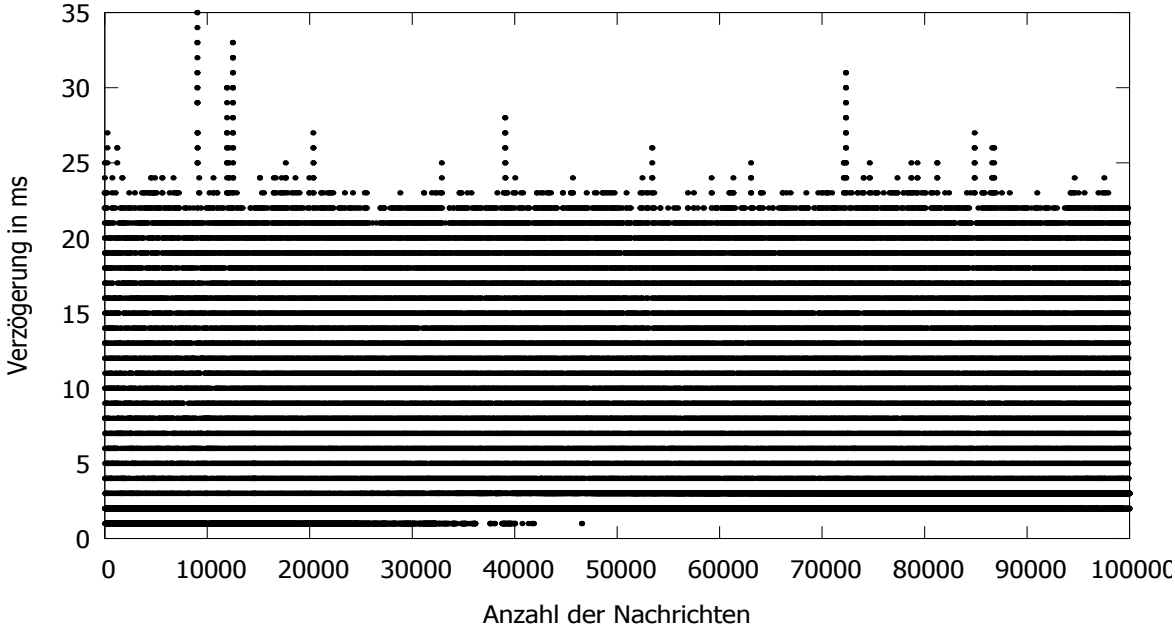

Abbildung 5.12.: Zirkel-Mechanismus mit Änderungen alle 0.1 Sekunden und Nachrichten alle 0.5 Millisekunden

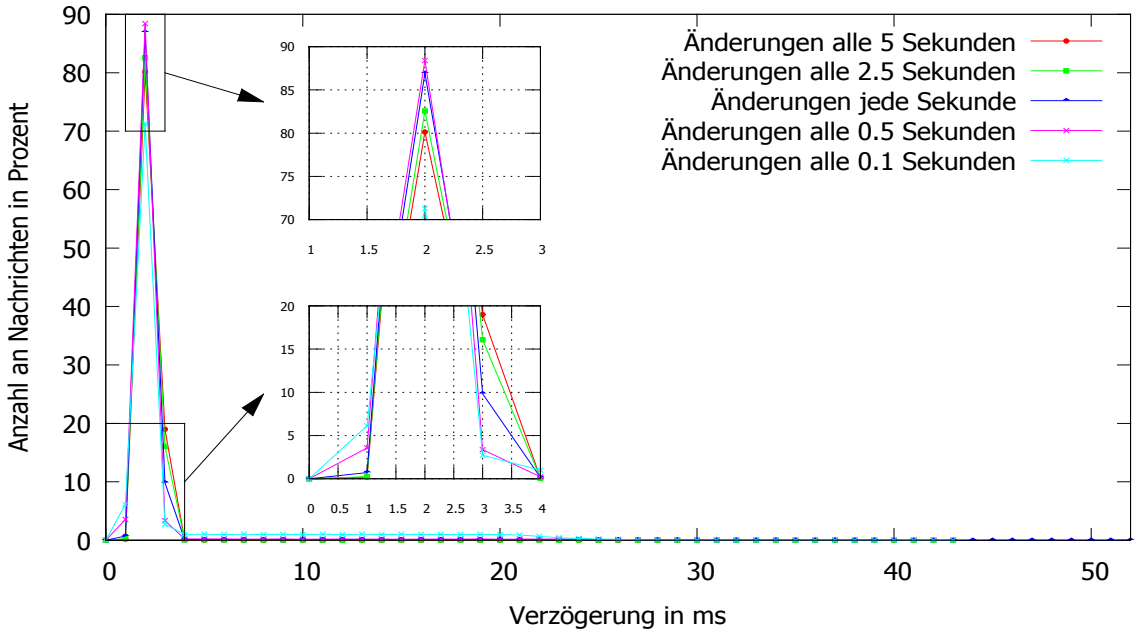

Abbildung 5.13.: Zirkel: Verteilung der Nachrichten bei Sendegeschwindigkeit von 2 ms

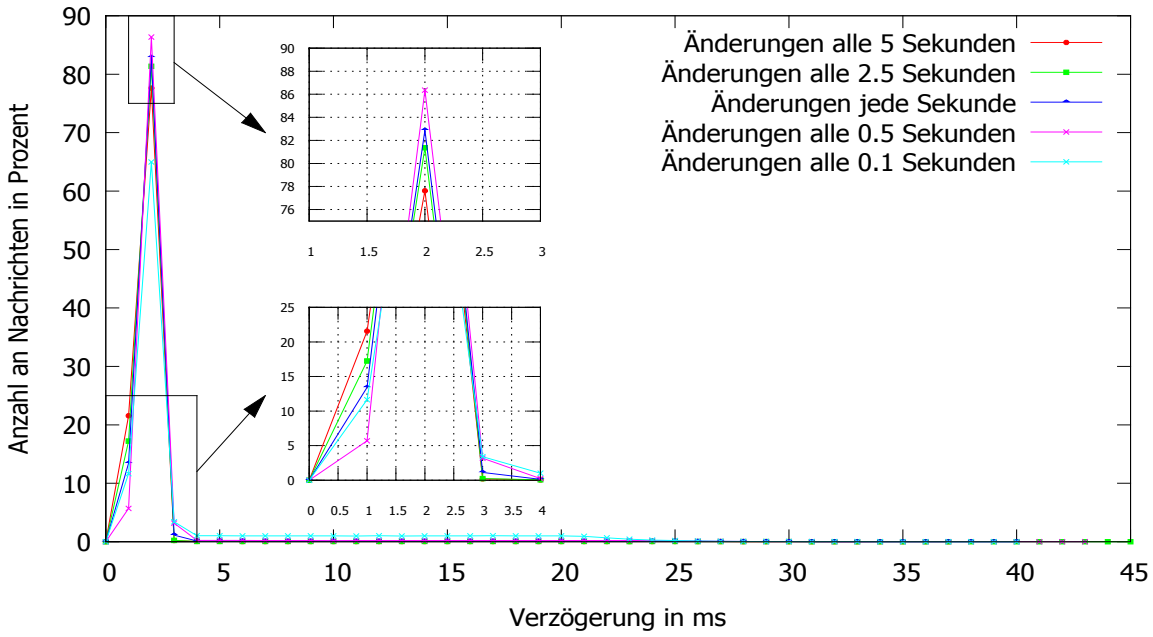

Abbildung 5.14.: Zirkel: Verteilung der Nachrichten bei Sendegeschwindigkeit von 1 ms

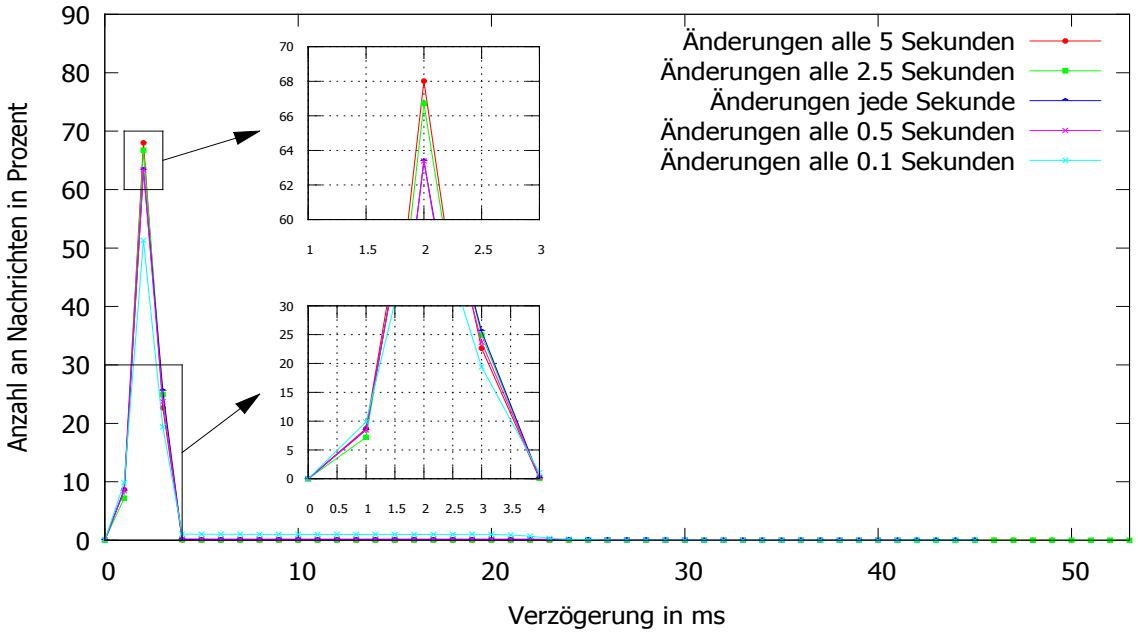

Abbildung 5.15.: Zirkel: Verteilung der Nachrichten bei Sendegeschwindigkeit von 0.5 ms

Auch hier wollen wir uns wieder den Mechanismus mit mehr Änderungen pro Sekunde anschauen. In 5.11 und 5.12 sieht man die Tests mit Änderungen alle 0.1 Sekunden. Was interessant zu beobachten ist, ist die durchschnittlich maximale Verzögerung. Diese scheint beim Zirkel-Mechanismus niedriger zu liegen, als beim NAK-Mechanismus obwohl man vom Aufbau her das umgekehrte annehmen könnte. Möglicherweise hängt das damit zusammen, dass die Last der Nachrichten beim NAK-Mechanismus nur auf einem Switch und dem Sender liegt und beim Zirkel-Mechanismus immerhin auf zwei Switches. Das ist aber durchaus positiv, da der Zirkel-Mechanismus grundsätzlich der praktischere ist, da man mit ihm in der Regel näher an der Transformation blockieren kann. Das liegt daran, dass man mit ihm Switches mehr oder weniger frei wählen kann und nicht auf einen Publisher am Switch angewiesen ist.

Wenn wir uns nun den Vergleich der Änderungen beim Senden von Nachrichten alle zwei Millisekunden anschauen, 5.13, sehen wir wieder ein ähnliches Bild wie schon beim NAK-Mechanismus. Auch hier ist es so, dass je weniger Änderungen vorkommen, desto mehr Nachrichten im normalen Bereich der Verzögerung ankommen. Die Werte bei 5 Sekunden liegen wieder um die 100% und die bei 0.1 Sekunden liegen um die 80%. Bei Änderungen alle 0.5 Sekunden haben wir wieder um die 95%. Für Änderungen alle 0.1 Sekunden liegen die größte Anzahl an Verzögerungen unter 24 Millisekunden. Für Werte darüber verhält es sich ähnlich wie bei Änderungen mit höherer Zeit. Es kommen einzelne vor, aber sie machen keine große Anzahl aus.

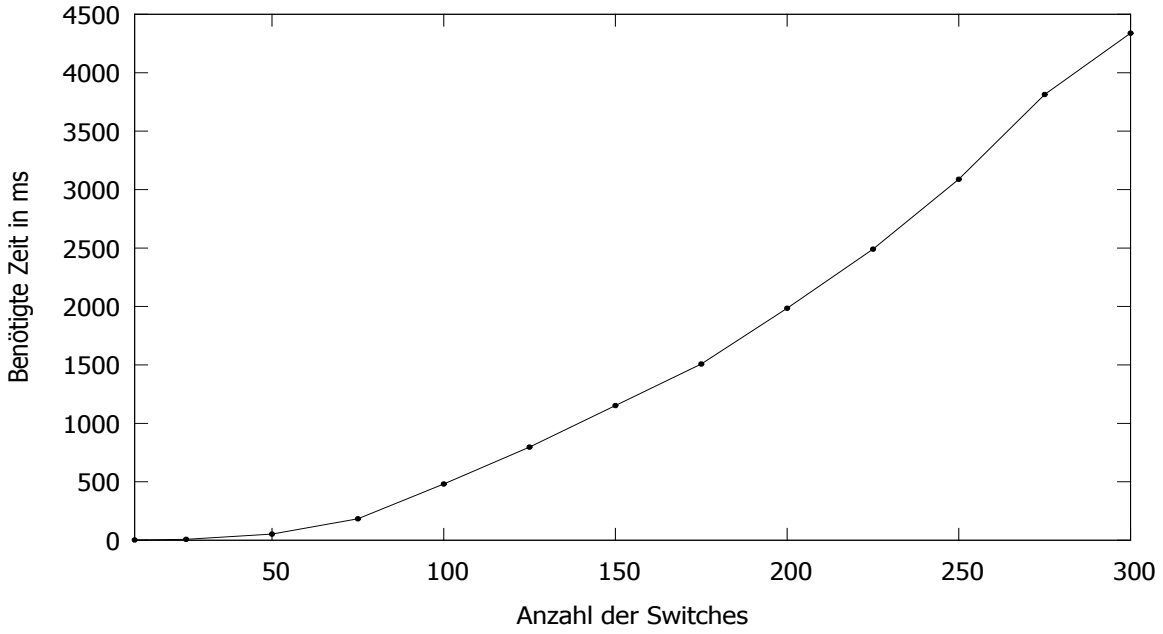

Abbildung 5.16.: Benötigte Zeit des Schedulers

Für die Nachrichten jede Millisekunde ergibt sich ein ähnliches Bild. In 5.14 können wir ablesen, dass die Werte wieder leicht sinken. Wobei auch hier die Abnahme sehr gering ist. Für Änderungen alle 0.1 Sekunden erreichen wir immer noch fast 80% und bei Änderungen alle 5 Sekunden sind wir immer noch bei fast 100%. Auch die anderen Werte liegen ähnlich denen beim Senden alle zwei Millisekunden.

Ein ähnliches Bild zeichnet sich beim Senden alle 0.5 Millisekunden ab. Auch hier sind die Werte im normalen Bereich der Verzögerung von 0 - 4 Millisekunden nahezu identisch mit den vorigen. Auch beim Zirkel-Mechanismus ist es so, dass die Sendegeschwindigkeit eine sehr kleine Rolle bei der Verzögerung der Nachrichten spielt. Eine weitaus größere Rolle spielt wiederum die Häufigkeit der Änderungen. Die Änderungen alle 0.5 Sekunden scheinen auch hier der beste Kompromiss zu sein, da selbst beim Senden alle 0.5 Millisekunden wieder ca. 95% aller Nachrichten im normalen Verzögerungsbereich ankommen. Das Verhältnis von blockierter Zeit zu nicht blockierter Zeit ist bei 0.1 nicht groß genug um einen höheren Prozentsatz zu erreichen. Zu bedenken ist, dass diese Änderungen sich auf denselben Bereich beziehen. Zwischen zwei verschiedenen Bereichen kann man also durchaus schneller Änderungen durchführen. Nur beim gleichen Bereich sollte man nicht zu schnell Änderungen ausführen.

5.3.3. Scheduler

Als letztes wollen wir uns noch die Leistung des Schedulers anschauen. Wir wollen untersuchen für welche Größe von Graphen sich diese Implementierung des Schedulers eignet. Dafür wurden unterschiedlich große Graphen getestet. Diese wurden komplett zufällig aufgebaut, da die Implementierung keine Probleme mit Zirkeln im Graphen hat. Das ist zwar nicht unbedingt nötig bei jeder Implementierung, allerdings hat es das testen einfacher gemacht. Was sich vor allem geändert hat, ist die Anzahl der Switches. Die Anzahl der Flows variiert von Switch zu Switch von eins bis zehn. Dabei ist jede Zahl so gleichberechtigt wie es der Zufallsgenerator hergibt. Die Ziele der Flows wurden auch zufällig ausgewählt, zwischen allen möglichen Switches. Auch hier ist jeder Switch gleichberechtigt. Da auch bei den Filtern jedes einzelne Bit gewürfelt wird ergibt sich im Endeffekt sicher kein besonders effektiver Graph, geschweige denn ein zusammenhängender im Sinne von zusammenhängend durch Filter. Zusätzlich ergeben sich auch sehr viele Änderungen von Graph zu Graph. Allerdings bietet sich diese Konstellation zum Testen des Schedulers an, da sehr viel gemacht werden muss. In realen Umgebungen dürfte er daher sogar etwas besser abschneiden. Um auch Publisher an den Switches zu haben, ist bei jedem Switch eine 33% Chance, dass an ihm ein Publisher mit einem zufälligen Filter ist. Die Tests wurden mit einem AMD Phenom II X6 1090T ausgeführt.

In 5.16 sehen wir die Ergebnisse. Gestartet wurde bei 10 Switches und ab dann in 25er Schritten aufwärts, beginnend bei 25. Wir sehen einen etwa quadratischen Anstieg in der Zeit. Ein exponentielles Wachstum war allerdings zu erwarten. Dass jedoch die Zeit so rasant ansteigt und schon bei 300 Switches bei über vier Sekunden liegt zeigt, dass hier noch viel optimiert werden kann. Der Scheduler, wie er gerade ist, eignet sich nur für recht kleine Netzwerke. Allerdings muss man beachten, dass der Test mehr oder minder das Worst-Case-Szenario darstellt. In einem realen Netzwerk, bzw. wenn wir den Ansatz aus [KDTR12] nehmen, werden diese Werte geringer ausfallen, da sich nicht fast jeder Flow ändern wird. Dies ist in dem Test der Fall, da die Flows zufällig zusammengesetzt werden.

Allerdings weist es auch in die Richtung, das man bei sehr großen Systemen darauf achten sollte nie den kompletten Graphen zu vergleichen. Ein Ansatz mit Teilgraphen ist hier weitaus ratsamer. Das ist allerdings wieder etwas komplexer und man muss abschätzen ob der Aufwand sich lohnt, oder ob es nicht erst einmal sinnvoller ist den Scheduler zu optimieren.

5.4. Vergleich und Bewertung

In diesem Abschnitt wollen wir den NAK- und den Zirkel-Mechanismus miteinander, von den Resultaten her, vergleichen und bewerten.

Prinzipiell eignen sich beide Mechanismen zum Blockieren wie die Tests gezeigt haben. Allerdings gibt es auch bei beiden eine Grenze bei der Zeit der Änderungen, an der es noch effektiv ist. Diese liegt bei Änderungen alle 0.5 Sekunden, wenn es das gleiche Gebiet betrifft.

Für diese ist bei beiden Mechanismen der Prozentsatz der Nachrichten mit normaler Verzögerung bei ca. 95%. Da beide Mechanismen ungefähr die gleiche Verzögerung produzieren ist es insgesamt besser, wenn man den Zirkel-Mechanismus mehr einsetzt, da er auch die isolierte Stelle möglichst gering hält. Der NAK-Mechanismus ist aber auch notwendig, zum Beispiel wenn Publisher in der isolierten Stelle liegen.

Bei der Optimierung scheint es für den NAK-Mechanismus auf den ersten Blick einfacher zu sein. Man benötigt nicht allzu viel Wissen über das Netzwerk um den Sender zu optimieren. Man könnte zum Beispiel eine Art Drosselung beim Wiedersenden einbauen. So lange noch Nachrichten zurückkommen erhöht sich die Zeit, die gewartet wird, bis die Nachrichten wieder gesendet werden.

Beim Zirkel-Mechanismus ist das schon komplizierter. Hier müsste man Wissen darüber haben wie lange die Nachrichten am Switch benötigen um weitergeleitet zu werden, wie lange es braucht um einen neuen Flow zu installieren und weitere Zeiten damit man das Warten zwischen den Schritten anpassen kann. Am besten wäre es, wenn der Controller eine Art ACK zurückbekommt, wenn der Flow gesetzt wurde. Dann könnte man, sobald man alle ACKs hat, den nächsten Schritt einleiten. Allerdings sieht das OpenFlow-Protokoll Nachrichten an den Controller nur vor, wenn ein Fehler auftrat oder der Flow gelöscht wurde. Das heißt man muss die Zeit abschätzen und dafür benötigt man Wissen über das Netzwerk und seine Auslastung.

Wenn man beide Mechanismen weiter optimiert hat, müsste man noch einmal testen ob der NAK-Mechanismus möglicherweise vorne liegt. Dabei müssen allerdings die negativen Effekte, möglicherweise größere isolierte Stellen, minimiert werden. Wahrscheinlich wird sich keiner der beiden Mechanismen alleine durchsetzen können und sie werden zusammen das beste Ergebnis erreichen.

6. Zusammenfassung und Ausblick

Wir haben einen Update-Mechanismus kennen gelernt, der in der speziellen Umgebung von Content-based Publish/Susbcribe, wie sie in [**?**] vorgestellt wurde, Garantien wie Nachrichtenerhalt und Duplikatsvermeidung einhalten kann und dabei, je nachdem wie schnell die Änderungen in einem Gebiet sind, einen recht hohen Prozentsatz an Nachrichten mit normaler Verzögerung bieten kann. Dabei wurde die große Graphtransformation vom Scheduler in kleinere Teile aufgeteilt, die nach Möglichkeit parallel abgearbeitet werden können. Der eigentliche Update-Mechanismus isoliert nun die Stelle, in der die Transformation stattfinden, soll, ähnlich einem Lockingprotokoll.

Die Ergebnisse zeigen, dass dieser Mechanismus gut funktioniert. Allerdings besteht, vor allem beim Scheduler, noch Optimierungsbedarf. Wie viel man allerdings jeweils mit Optimierung herausholen kann, kommt sehr darauf an. Im Falle des Schedulers dürfte es noch sehr viel Spielraum geben, da dieser eigentlich keine weiteren Voraussetzungen benötigt, wie etwa die Optimierung der beiden Update-Mechanismen. Diese würden Wissen über das Netzwerk benötigen um möglichst effektiv zu sein. Dieses Wissen ist allerdings ein Problem, da man es nicht so leicht bekommen kann. Der Controller müsste Abschätzungen durchführen wie lange das Weiterleiten von Nachrichten bei einem Switch benötigen, wie schnell der Switch Flows setzt etc., wenn man die Informationen nicht aus anderen Quellen bekommen kann. Diese Abschätzungen sind für größere Systeme mit unterschiedlichsten Komponenten allerdings nötig, da man nicht alles Wissen vorgeben kann, aber eben auch schwer zu realisieren. OpenFlow bietet hier leider keine Erleichterung. Aber möglicherweise werden noch Punkte in das Protokoll aufgenommen, die solche Abschätzungen leichter machen.

Ausblick

Als ein Punkt für weitere Arbeiten steht sicher die Optimierung der Mechanismen, wie auch des Schedulers. Aber auch der dritte Mechanismus, der kurz vorgestellt wurde, die Storage-Lösung, eignet sich als Thema. Hier ist es sicher interessant, wie man den Speicher konzipiert, dass dieser schnell Daten speichern und diese auch schnell wieder verbreiten kann. Aber auch die Verteilung dieses Speichers ist eine interessante Aufgabe. Er sollte nah an der isolierten Stelle sein, damit nicht zu viel blockiert wird, am besten komplett dynamisch. Möglicherweise sogar als extra Speicher im Switch. Ob das so allerdings zu realisieren ist wird sich zeigen.

Man kann auch den Ansatz weiter verfolgen, dass man nicht mehr nur ganze Graphen übergibt, sondern Teilgraphen, oder aber sogar nur einzelne Ereignisse. Mit beidem ist der Scheduler nicht so sehr belastet wie mit einem ganzen Graphen und könnte schneller die Transformationen berechnen. Das könnte man so weit treiben, dass man den Controller in viele verschiedene aufteilt und sich jeder nur um einen Teilgraphen kümmert. Damit kann man die Last aufteilen, bekommt allerdings die typischen Probleme verteilter Netze wie zum Beispiel Synchronisierung bei Transformationen, die nicht nur in einem Teilgraphen stattfinden.

Alles in allem kann man in dieser Richtung noch sehr viel machen. Der Bedarf an Publish/Subscribe-Systemen und das Aufkommen von SDN zeigen auch, dass hier noch viel Spielraum ist. Wie genau sich dieser Bereich allerdings entwickelt ist schwer zu sagen.

A. Anhang

Hier sollen ein paar Probleme und Schwierigkeiten vorgestellt werden, auf die man bei der Implementierung achten sollte.

Als erstes wollen wir auf die Multicast-Problematik eingehen. Wir verwenden Multicast-Adressen, weil diese keine MAC-Adresse benötigen. Verwendet man Unicast-Adressen, muss man entweder zusätzliche ARP-Flows installieren oder eine Dummy-MAC-Adresse setzen, die dann bei Switches mit Subscribern auf die vom Subscriber geändert wird. Um das zu umgehen wurden Multicast-Adressen genommen. Diese haben allerdings das Problem, dass jede Nachricht auch wieder an den Sender geliefert wird. Das heißt beim NAK-Mechanismus muss man dafür sorgen, dass die Nachrichten sich von diesen Nachrichten unterscheiden. Um das zu erreichen, bzw. es überhaupt zu erreichen, dass die Nachrichten ankommen, muss man die Src-MAC-Adresse und die Src-IP-Adresse ändern, bevor man sie an den Sender zurückschickt. Praktisch wäre hier eine globale MAC und IP damit man das für jeden Publisher einfach implementieren kann.

Aber auch mit Floodlight muss man aufpassen. Es sind noch einige Probleme vorhanden, gerade beim Setzen von Masken für IPs. Dies geht nur über die Wildcards-Klasse korrekt und auch hier sollte man kontrollieren ob sie korrekt gesetzt wurde. Für diese Arbeit hat das nicht geklappt und es wurde eine REST-Funktion im StaticFlowEntryPusher implementiert, um das Setzen von Flows korrekt auszuführen. Floodlight bringt noch andere kleine Probleme, weswegen man immer wieder kontrollieren sollte, ob die Methoden das gewünschte Ergebnis erzielen.

Mininet hat zwar keine Fehler, allerdings muss man für viele Funktionalitäten erst suchen und rumprobieren, da sie nicht direkt erklärt werden. Außerdem kommt es bei der Leistung wohl sehr darauf an wie man Mininet installiert. Bei einer Installation in einer virtuelle Maschine scheint die Leistung niedriger zu sein, als wenn man es direkt unter Linux installiert.

Im Testbed funktionieren VLan-Tags derzeit nicht. In Zukunft ist das vielleicht möglich, aber für diese Arbeit wurden MAC-Adressen verwendet. Diese reichen aus, jedoch wären VLan-Tags besser, da man MAC-Adressen auch noch für die Kodierung der Filter nehmen kann, so lange Floodlight kein IPv6 unterstützt.

Literaturverzeichnis

[Arc09] D. Archambault. Structural differences between two graphs through hierarchies. In *Proceedings of Graphics Interface 2009*, S. 87–94. Canadian Information Processing Society, 2009.

[CMT+11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, S. Banerjee. DevoFlow: scaling flow management for high-performance networks. *SIGCOMM-Computer Communication Review*, 41(4):254, 2011.

[CRW01] A. Carzaniga, D. S. Rosenblum, A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[Fou12a] O. N. Foundation. OpenFlow Management and Configuration Protocol (OF-Config 1.1). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.1.pdf, 2012.

[Fou12b] O. N. Foundation. Software-Defined Networking: The New Norm for Networks. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, 2012.

[Fou13] O. N. Foundation. Openflow switch specification v1.3.2. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf, 2013.

[GC12] S. Ghorbani, M. Caesar. Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks*, S. 67–72. ACM, 2012.

[Hec06] R. Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

[HSM12] B. Heller, R. Sherwood, N. McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, S. 7–12. ACM, 2012.

[JCL+10] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, R. S. Kazemzadeh. The PADRES Publish/Subscribe System., 2010.

[KBR09] B. Koldehofe, J. A. Briones, K. Rothermel. SPINE: Adaptive Publish/Subscribe for Wireless Mesh Networks. *Studia Informatika Universalis*, 7(3):320–353, 2009.

[KDTR12] B. Koldehofe, F. Dürr, M. A. Tariq, K. Rothermel. The power of software-defined networking: line-rate content-based routing using OpenFlow. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, S. 3. ACM, 2012.

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[McG12] R. McGeer. A safe, efficient update protocol for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, S. 61–66. ACM, 2012.

[Müh02] G. Mühl. *Large-scale content-based publish-subscribe systems*. Dissertation, TU Darmstadt, 2002.

[Pie04] P. R. Pietzuch. Hermes: A scalable event-based middleware. *University of Cambridge PhD thesis and TR590*, 2004.

[Plu99] D. Plump. Term graph rewriting. *Handbook of graph grammars and computing by graph transformation*, 2:3–61, 1999.

[RFR+12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, S. 323–334. ACM, 2012.

[RFRW11] M. Reitblatt, N. Foster, J. Rexford, D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, S. 7. ACM, 2011.

[Roz97] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation, vol 1: Foundations*. World Scientific, 1997.

[TKK+11] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, K. Rothermel. Meeting subscriber-defined QoS constraints in publish/subscribe systems. *Concurrency and Computation: Practice and Experience*, 23(17):2140–2153, 2011.

[TKKR09] M. A. Tariq, B. Koldehofe, G. G. Koch, K. Rothermel. Providing probabilistic latency bounds for dynamic publish/subscribe systems. In *Kommunikation in Verteilten Systemen (KiVS)*, S. 155–166. Springer, 2009.

[VW12] A. Voellmy, J. Wang. Scalable software defined network controllers. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, S. 289–290. ACM, 2012.

[wika] Wikipedia - Graph rewriting. https://en.wikipedia.org/wiki/Graph_rewriting.

[wikb] Wikipedia - Publishâ€“subscribe pattern. https://en.wikipedia.org/wiki/Publish-subscribe_pattern.

[wikc] Wikipedia - Software-defined networking. https://de.wikipedia.org/wiki/Software-defined_networking.

[WKB+08] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. In *ACM SIGCOMM Computer Communication Review*, Band 38, S. 231–242. ACM, 2008.

Alle URLs wurden zuletzt am 12. 08. 2013 geprüft.

Erklärung

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

Ort, Datum, Unterschrift