Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 149

Systems Group, Department of Computer Science, ETH Zurich

Expressing the Routing Logic of a SDN Controller as a Differential Dataflow

by

Christian Stuecklberger

Supervised by

Dr. Desislava Dimitrova
Dr. Ioannis Liagouris
Prof. Timothy Roscoe

Jan 2016–July 2016

inf | Informatik
Computer Science

Abstract

Programmable networks are quickly gaining popularity in both academic research and industry, as they provide a new way to deal with network management. Several deployments by industry including Microsoft and Google already demonstrate the advantages of such paradigm shifts. Despite the initial momentum in the development of logically centralized control logic, i.e. the network controller, a shift towards proprietary industry-driven solutions is observed. Part of the problem is that open source academic-backed controllers have difficulties to scale at the level required by industry operation, particularly when it comes to the speed of operation. This thesis investigates a new approach towards the controller platform by adopting a dataflow processing framework as the computational foundation. At the same time it introduces the base of a more formal way to reason about network management. Specifically, the thesis builds around a representation of the network as a graph which allows us to specify high-level configuration policies as constraints on top of this graph and use well-understood graph computations in a data-parallel and incremental fashion to calculate the network routing. Our results demonstrate a very competitive performance of the routing module even before potential optimizations are conducted. Furthermore, interaction with the system is intuitive and human-friendly thanks to the higher-level policies we introduce.

Acknowledgments

First I would like to thank my two supervisors, Desislava Dimitrova and Ioannis Liagouris. They were both essential during the process of writing the code and text in the scope of this thesis and were always there to help me when I ran into problems. Desi explained all the parts of network technology I needed for my work and was of immense help with finding errors of grammar or content in this thesis. She also never failed to motivate me when I had the feeling my progress was too slow by providing nice words and assuring me everything is fine. John introduced me to the framework that is fundamental to this thesis, explaining the parts I found hard to understand. He was a huge help in finding bugs and crafting solutions to problems that seemed unsolvable to me at times.
Then I want to thank Zaheer Chothia who was always willing to help with debugging the code and various other things. I also want to thank all friends that took part in discussions about the topic and helped me with the many smaller issues that came up.
Finally I would like to thank Prof. Timothy Roscoe for making this thesis possible. Especially the weekly meetings he organized were helpful to get new input and resolve issues in the scope of this work.

Contents

1 Introduction

In this chapter we provide a high-level overview of the technical challenges in managing today's networks. We then describe recent advances in real-time data processing that motivated our work, allowing us to effectively tackle several open problems in the field of programmable networking. The contributions of our work is summarized at the end of this chapter where we also outline the structure of this thesis.

1.1 Motivation

Computer networks are steadily getting bigger and also the traffic within those networks keeps increasing. This is especially the case in data centers which may include hundreds or even thousands of physical machines. Traditional networking approaches build upon technology that was introduced several decades ago and has already become obsolete. Switches are an example for this, they often rely on incremental network discovery to try to locally determine the best routing within the network. Generally conventional networks tend to be inflexible and slow to respond to changes of e.g. traffic patterns or network topology. The rise of Software Defined Networking (SDN) offers potential to improve on this. SDN tries to solve many problems of traditional networking by giving explicit control over the routing in the network. Although the principles of SDN are promising the solutions currently available face several problems. First they do not offer the performance

and scalability that is needed to handle the fast-pace increase of internal traffic in large-scale data centers. Second they tend to be hard to set up and lack possibilities to easily constrain the flow of packets in the network. Third they are all build upon traditional programming languages and paradigms. In our work we want to improve on all of the above points by applying recent advances in the fields of programming languages, graph databases and real-time data processing. This includes the new programming language Rust and two novel frameworks, Timely Dataflow and Differential Dataflow. They enable real-time data processing with low latency and high throughput, and also support efficient graph computations. Their biggest advantages are the fast processing of incremental updates to the input data and the completely transparent parallelizability of the dataflow program. Both of these features are highly useful when creating a network controller and enable new levels of performance. Additionally the Rust language helps to implement the controller with safe and easy to read code.

1.2 Problem Statement

In the scope of this thesis we want to define a powerful model for various network management tasks. It should allow to capture all important aspects of a big computer network in a principled manner. Next we want to express complex routing tasks as a combination of well-understood dataflow computations. They should be efficiently executable in a scalable fashion and allow incremental updates of the input data. Finally we target a human-friendly configuration of the system through the definition of a policy language. This language should allow both intuitive and formal-based policies which can be translated into restrictions on the dataflow. To summarize, the goals are to provide manageability through the model, simplicity through the policy definitions and efficiency through the dataflow implementation.

1.3 Contributions

Within the scope of this thesis we make the following contributions:

Model We define a model for topologies and policies. A topology consists of hosts, switches and weighted connections. A policy is composed of a source host, a destination host and a constraint defining switches that need to be on the routing path.

We also introduce a concise and simple syntax that allows to describe topologies and policies. The model is extensible and allows to extract different views of the network, depending on the particular management tasks.

System We create the prototype of an SDN Controller implemented as dataflow computation. It is built on top of the differential-dataflow library and written in the Rust programming language. To allow adding constraints to the routing it supports a policy language that corresponds to an expressive subset of widely-used policy languages. It performs the routing tasks in a data-parallel fashion and thus scales well when increasing the number of workers. Furthermore it is able to update routes incrementally with extremely low latency and high throughput. To allow simple instantiation it offers a command-line interface. We provide a thorough evaluation of the controller based on extensive measurements we conducted with Fat-Tree and Jellyfish network topologies to show the strengths of our approach.

Utilities We build a test bench measuring the performance of our SDN controller in detail. With it we show the effectiveness of our approach and how it scales when adding more computing resources or increasing the input sizes. We also create generators for Fat-Tree and Jellyfish topologies and also for policies of arbitrary size. They are used by the test bench to show how our SDN controller, i. e. its underlying dataflow computation, scales when increasing the input size.

1.4 Structure of the Thesis

In the following we explain how this thesis is structured. Chapter 2 gives details on the foundations of our project. It introduces the Rust programming language, the dataflow programming paradigm together with the two libraries implementing it and the concept of Software Defined Networking. In Chapter 3 we present the most well-known network programming languages and other implementations of SDN-controllers. Moreover we show what else has been done with the dataflow framework we use. Chapter 4 describes the model of the data types we used. This encompasses the topology, policy and the syntax we created to allow the creation of input files for both. Next follows Chapter 5 which shows our implementation. It gives an overview of the project structure and then reveals in detail what we implemented. There we also show the interface of our controller and its test bench. Furthermore we justify our choices for data types and algorithms. In

Chapter 6 we provide an evaluation of our work. We define exactly how we conducted our benchmarks and then present the results. This is done with the help of many plots and tables. Additional tables providing all measurements we conducted are shown in the appendix' Chapter A. Finally we provide conclusions of our findings and give directions for future work in Chapter 7.

2

Preliminaries

In this chapter we explain technical details one needs to understand before reading the rest of the thesis. This especially includes the dataflow programming model which is the basis for the two frameworks our work builds upon, timely-dataflow [McSb] and differential-dataflow [McSa]. We introduce Rust, the programming language we used and in which timely and differential are written. Also we give an overview of Software-Defined Networking (SDN).

2.1 The Rust Programming Language

Rust is a new systems programming language designed with the goal to be "a safe, concurrent, practical systems language"[1]. The project was started by Graydon Hoare in 2006 and the first stable version of the compiler – Rust 1.0 – was released 2015 on May 15th[2]. Although sponsored by Mozilla, the development is done by a big community not only consisting of Mozilla employees. The project is released as open source under the MIT [MIT] or the Apache license version 2.0 [Apa].

The language works without the use of a garbage collector and guarantees memory safety and data-race-free concurrency. Memory safety means that it is technically impossible to

[1] `https://www.rust-lang.org/faq.html#what-is-this-projects-goal`
[2] `http://blog.rust-lang.org/2015/05/15/Rust-1.0.html`

dereference a null pointer, to have a dangling pointer to an already deallocated object or to do an out-of-bounds array access. This prevents bad things from happening because all three events listed above can lead to either an abort of the execution or an invalid program state. The second warranty, data-race-free concurrency, guarantees that only one thread can have write access to a variable at any time. This saves the programmer from having to deal with locks in normal circumstances and avoids many common mistakes. In conventional programming languages two threads can potentially write to the same variable simultaneously, which leads to indeterministic program behavior or even invalid program states in the worst case. Rust enables the above properties by introducing the concepts of scoping, non-null types, ownership, borrow checking and lifetimes among others. Please consult our references for more details on those concepts. Apart from array bounds checking which must be done at runtime, most of the other checks enabling the properties mentioned above occur statically at compile time so they do not induce runtime overhead. As a side effect this also allows the compiler to give very detailed error messages in case the programmer does not comply with the language's concepts.

Rust allows memory allocation on the stack and on the heap. All local variables are stored on the stack by default. When something needs to be allocated on the heap, it can be done by using the type `Box`. For example when a primitive type like a number should be stored on the heap, it is wrapped in an object of type `Box`. This object stores the address of the memory location and automatically deallocates its memory on the heap once it goes out of scope.

[RTa, RTb, Bla15]

2.2 Dataflow Programming

The dataflow programming model defines a computation as a directed graph, where the data flows from one processing node to another. Each node can have incoming and outgoing edges where data tuples go in and out respectively. To define a dataflow program, the programmer has to specify the graph's nodes and edges and also how each node modifies and forwards data. In contrast to conventional computer programs the execution order is not explicitly defined. Instead, it is determined by the data present in each nodes' buffer. No data means there is nothing to process and the node is not scheduled for computation. A non-empty buffer on the contrary means the node has work to do.

2.2.1 Timely Dataflow

Timely dataflow is a computational model first introduced in 2013 by Murray et al. [MMI+13]. It was developed to combine high throughput with low latency and allow for incremental, iterative computations with consistency guarantees. Specifically, this model provides the following features (from [MMI+13, p. 1]):

1. loops allowing feedback in the dataflow;
2. stateful dataflow vertices capable of consuming and producing records without global coordination;
3. notifications for vertices once they have received all records for a given round of input or loop iteration.

In Figure 2.1 we show an example graph visualizing a timely computation. It has got an input and output node designating the interface to code outside of the dataflow computation. There also is an embedded loop with a designated ingress (entry), egress (exit) and feedback node (I, E and F respectively). The computation within the loop happens in a separate scope that has its own timestamps. The ingress, egress and feedback nodes handle the conversion from the outer scope's timestamps to the ones used inside. This is necessary to keep track of iterations and allows the loop to iterate until reaching a fixed point. The nodes performing computations are A, B, C and D. They can represent any operators, which are either predefined by the library or are custom-made ones.

The creators of timely dataflow built the prototype implementation *Naiad*. They wrote it in C# and support LINQ-Queries[3] for the differential-dataflow part. Its potential was shown by evaluating it with three different large-scale graph problems and outperforming other systems by orders of magnitude. [MMI+13]
In our work we do not use the Naiad System introduced in the paper cited above. Unfortunately the project was terminated when the Microsoft Research Lab developing it was shut down. Luckily, a modified version was written in Rust by Frank McSherry, a former contributor of the Naiad system. It consists of the library *timely-dataflow* [McSb] released under the MIT License [MIT]. The code is extensively documented and there also exist several tutorials and blog posts on how to use the libraries. Also, Rust is a good choice for a framework like this, for the reasons outlined in Section 2.1.

[3] `https://msdn.microsoft.com/en-us/library/bb397926.aspx`

[MMI+13]

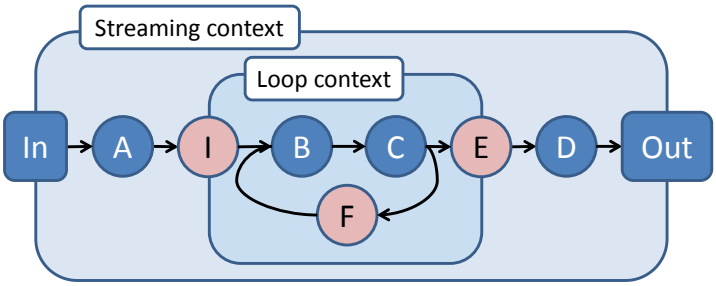


Figure 2.1: Timely dataflow graph example. Contains an input and an output node and a loop with a feedback edge. Source: [MMI+13, Figure 3]

2.2.2 Differential Dataflow

Differential dataflow is an enhancement of timely, extending it with a new data type and additional operators. It introduces the notion of *operators* that are applied to *collections*, which are sets of data tuples. Some of the available operators are *map*, *join*, *group*, *concatenate* or *iterate*. Another one is *iterate*, which contains an inner dataflow computation it repeatedly executes until reaching a fixed point. Many different problems can be expressed with those operators but they are particularly suitable for graph algorithms such as the evaluation of connected components or a breadth-first search.
The defining characteristic of the differential dataflow framework is the fact that it processes its input incrementally. Each time new input arrives only the deltas are trickled through the computation, leading to a low latency response. Another feature of differential dataflow is the fact that computations can be transparently parallelized. This allows to decrease the runtime by just instructing the system to use more threads when running the program on a multi-core machine.
Both properties make the framework perfectly suitable for networking tasks such as routing in a computer network. The low-latency responses to incremental changes allow adapting the forward rules so they fit the current network load very fast. It is also able to support huge network sizes by utilizing all available cores on a powerful multi-core computer. A possibility to implement the routing algorithm is modeling the network as a graph with weighted edges and then calculating the shortest paths with Dijkstra's algorithm which is based on a breadth-first-search. After initially computing those paths between all network nodes the dataflow computation can react very fast to updates of the network. Those

updates could be changes of the network load or connection failures. The former can be modeled as a weight increase of an edge in the graph. The latter manifests in the removal of the respective edge from the input graph.

[MMII13, CLMR16]

2.3 Software-Defined Networking (SDN)

Since the first computer networks were built, network sizes and utilized bandwidth are steadily increasing. Despite this, the technology used to build those networks did not change with the same pace. Most of the hardware utilized today builds upon technology already around for years or even decades. We are now at a point where increasing the hardware specification alone is not enough anymore. New technology like Software-Defined Networking (SDN) can lead to significant performance and cost improvements. In this section we explain how this could be done and also what the problems with traditional network hardware are.

Conventional Networks In conventional networks the routing is generally determined by the switches. None of them has a global picture of the network, they iteratively determine their own approximation of the optimal routing. This procedure is robust but takes time and is especially slow when changes or failures occur in the network, which is its biggest disadvantage. Forwarding rules are also not guaranteed to be consistent or optimal through all switches. Another drawback is the fact that it is very complicated to enforce global policies. Also the high-end switches needed for big networks tend to be very costly and proprietary hardware generally made innovation in the sector slow and expensive. Specifications of the hardware were evolving slower than the increase in data volume. Especially data centers needed alternatives to deliver on their customer's contracts and to cope with the ever-increasing traffic demands.

Switches and Routing in SDN Software-Defined Networking is a new approach in which a central entity – the controller – determines the routing tables. It allows to operate from a global view and respond to changes and failures in the network much faster and more efficiently. Also the traffic management by a central controller entity has potential to be near-optimal in regards of network utilization and path lengths and to simplify the

management of network policies.
In SDN the control plane is separated from the data plane. The former consists of the central controller and all the information it possesses. The latter encompasses the switches and the actual process of forwarding the packets to the correct next network node. Nowadays the data plane is very mature and advanced. The introduction of *OpenFlow* allowed the standardization of the interface across different vendors. The control plane on the other hand is hard configuration work with many different used protocols. There exists no single tool allowing to set up a network easily.
In SDN switches announce their presence by sending keep-alive messages. That way the controller discovers the topology and creates the topology graph. With this information it then runs the routing algorithm and generates flow-rules it deploys on the switches.

3

Related Work

In this chapter we give an overview of efforts in the area of controller logic development for programmable networks. Section 3.1 introduces other SDN controller platforms that are available today. They are all written in conventional programming languages like Python or Java, as we know of no SDN controller that is implemented as a dataflow computation. In Section 3.2 we show network programming languages. Their purpose is to define constraints on the flow of packets in a network.

3.1 SDN Controllers

This section describes and compares five open-source controllers. The paper we cite from written by Khondoker et. al. [KZMB14] suggests those as the most important. They argue that these are the most used controllers that are properly documented, provide a well-engineered implementation and are neither deprecated nor are constructed for special tasks. The following list contains their names, an URL to the project's main website and the language they are written in:

Ryu `https://osrg.github.io/ryu/` (Python)

POX `http://www.noxrepo.org/pox/about-pox/` (Python)

OpenDaylight `https://www.opendaylight.org/` (Java)

Floodlight `http://www.projectfloodlight.org/floodlight/` (Java)

Trema `https://trema.github.io/trema/` (Ruby and C)

All five controller platforms are under active development and two of them are supported by big companies. Ryu is backed by NTT, the biggest Japanese telecommunications provider and Trema by NEC, which is a big network hardware manufacturer. They all support Linux as a platform to run on, POX and FloodLight additionally list Windows and Mac OS as supported systems.

Implementation Language The mentioned platforms outside of Trema are built in Java or Python, Trema is built with C and Ruby. All of those are conventional programming languages that were not specifically created for networking applications. This makes it difficult to reflect the iterative computations in network management. The C language may be the most suited for those applications, as it is very low-level. We think dataflow programming is much better suited for problems found in network management. It is fast, highly parallelizable and based on the same concept as computer networks themselves.

Southbound Interfaces Communication between the controller and the forwarding layers of the network happens through the controller's so called *southbound interfaces*. They are represented by APIs following certain standards, e. g. OpenFlow[1], which is the one most commonly used. Of our controller selection all support OpenFlow version 1.0 and Ryu additionally supports versions 2.0 and 3.0. Through this standard the controller can instruct the switching devices (both in hardware and software) which paths packets in the network should take.

Northbound Interfaces Counterpart to the southbound interface are the *northbound interfaces*. Through it the controller can receive instructions on how the traffic in the network should be routed. It can also allow to set constraints and define restrictions on the routing of some packets. Ryu, POX and Trema provide an ad-hoc API, which means that the controller can be configured by using the implementation language. OpenDaylight and Floodlight on the other hand offer a RESTful API, which means the controller can be programmed through a resource identifier (e.g. for a switch) in combination with an

[1] OpenFlow `https://www.opennetworking.org/sdn-resources/openflow/`

action (e.g. add rule to drop all packets coming from switch X). POX and OpenDaylight additionally provide a graphical user interface. Ryu offers one for the initial phase of the setup and configuration. Floodlight has a web-interface accessible through the browser to execute the RESTful queries. Our work in policies focuses on providing more human-friendly interaction on the northbound interface.

Scaling Another important factor for network controllers is their ability to process big network topologies. When the number of nodes in the network gets higher it is required that the controller platform scales. For this reason Ryu, Floodlight and Trema support multi-threaded execution. OpenDaylight is built with a distributed design concept. This means that several instances of the controller run in parallel and communicate with each other, leading not only to arbitrary scaling but also allows redundancy to provide fail-safeness. Our choice of framework, Timely and Differential Dataflow, was also driven by scalability requirements. Differential Dataflow offers transparent parallelizability and its incremental input processing makes it an excellent candidate for a controller platform.

Virtualization Since the advent of virtual servers residing on physical ones, the capability to handle traffic of virtual networks is a necessity for controller platforms. This is closely related to simulation and emulation of networks, which is used to test network controllers. All controllers except Trema support network emulation with Mininet and virtual networks that use *Open vSwitch*[2]. Open vSwitch is a virtual switch that acts the same as a physical switch in the hardware network, but exists only in software. It connects several virtual machines as if they were connected through a real switch, also embedding them in the physical network the host machine is part of. Trema has its own built-in tools for emulation and virtual switches which we do not describe in more detail. Our prototype has the potential to not only align with software switches but to also integrate with their hardware counterparts.

Modularity The last criterion by which we compare the different controller platforms is modularity. This trait is important because it enables separation of concern and the easy replacement of implementation parts. It leads to a higher reusability of not only the controller's code, but also configurations of the system which can be extensive for big networks. Generally all the controller we present here are more or less modularly

[2]Open vSwitch `http://openvswitch.org/`

built. OpenDaylight and Floodlight are built as a modular system completely where the individual submodules interact through services to form the whole controller platform. [KZMB14, KREV+15]

Table 3.1: Overview of five different SDN-Controller platforms.

	Controller Platform				
	Ryu	POX	OpenDaylight	Floodlight	Trema
Impl. Language	Python	Python	Java	Java	Ruby / C
Supported OS	mostly Linux	Linux, Mac, Windows	Linux	Linux, Mac, Windows	Linux
Open Source	yes	yes	yes	yes	yes
Commerc. backed	by NTT	no	no	no	by NEC
GUI	partial	yes	yes	web	no
Multi-Threaded	yes	no	yes (distr.)	yes	yes
OpenFlow	v3.0	v1.0	v1.0	v1.0	v1.0
Northbound API	ad-hoc	ad-hoc	RESTful	RESTful	ad-hoc
Modularity	medium	medium	high	high	medium
Virtualization	Mininet, Open vSwitch	Mininet, Open vSwitch	Mininet, Open vSwitch	Mininet, Open vSwitch	Built-in

3.2 Network Programming Languages

In this section we give an overview over the most well-known network programming languages. The purpose of network programming languages is to configure the packet forwarding hardware within a network. One simple approach to this is a low-level machine language like OpenFlow that resembles the hardware and provides a unified interface to it. Apart from the complexity the high number of policies that have to be managed an array of other issues for the programmer arise. For example we need additional support to detect overlapping rules or different priority ordering. Also the installation of new rules affecting packets that are still traversing through the network can cause problems. High-

level languages can spare the user of those troubles and also provide abstractions that help becoming more productive to solve complex tasks faster and easier. In the following we present four high-level network programming languages. We took them as inspiration for our system's syntax in regards of what can or should be done. Although we do not want to compare ourselves with them, as we try to offer a base for possible future efforts. It is not directly shown in this thesis but in additional research efforts application-level load balancing was implemented, which introduces higher-level policies hiding the network composition through the ability to automate it.

Frenetic Frenetic was the first attempt to create a high-level language on top of OpenFlow. It is implemented in the C programming language and built on top of the NOX controller platform, a predecessor of POX. The language introduces queries and operators to allow for code with modular design of policies It also aims to have the programmer think only about what he want and not how the network hardware implements it or what problems could arise from his or her chosen constructs. [FHF+11]

Pyretic Pyretic is the successor of Frenetic and written in Python. It also builds upon OpenFlow and its implementation is integrated with POX, which is the Python based version of the NOX platform. In contrast to Frenetic and most other network programming languages it is an imperative language. [RMF+13]

Merlin The Merlin framework is implemented in the OCaml and C programming languages and builds atop the Frenetic controller. It extends its language with bandwidth constraints that it resolves with an optimizer during compilation to generate OpenFlow compatible constraints. [SBM+14]

Fibbing Fibbing is a network architecture that combines the advantages of SDN with the robustness of distributed network routing in conventional networks. Its biggest advantage is the fact that it uses traditional switches rather than OpenFlow compatible network hardware and manipulates the routing by inserting fake nodes in the network. Policies can be defined with help of a high-level language that is part of Fibbing. It allows to set path requirements in the form of network nodes that must or must not be passed on a certain packet stream through the network. [VTVR15]

[KREV+15]

4

Model

In this chapter we describe our model for the necessary data types we use. This encompasses the topology and policy, either of which can be generated in arbitrary sizes directly in the program or parsed from text files. We explain the implementation for both detailed in Section 5.2.

4.1 Topology

A network topology as we understand it is a graph consisting of *nodes* and *links*. In our case nodes can be either *hosts* or *switches* in the network but not both. Links are *connections* among exactly two switches and represent a network cable. Also part of our topology model are *update batches* that depict the removal or update of network connections. In the subsections below we present all of those individual parts in detail and also describe the input file format we accept for topologies.

Host Hosts are the network endpoints, the producers and consumers of information which can be servers, clients or terminals. Source and destination of a packet flow within the network can only be hosts. For routing purposes without bandwidth constraints we deemed this as acceptable. We assume an admission control takes care of capacity checks. Hosts are connected to exactly one switch and no other nodes. We do not consider this

link to its parent as a connection, it is not contained in a topology's list of connections. Each host in our model has a unique name, its own node id and also knows to which parent switch it is connected.

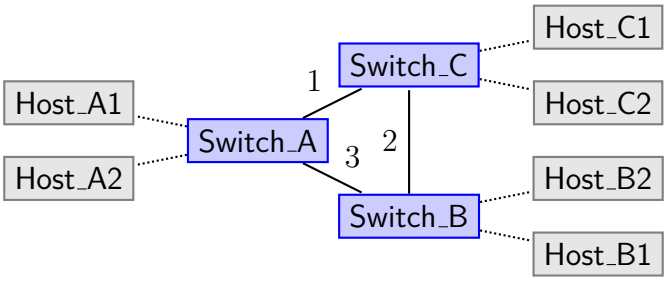
Switch Switches are network intermediates and route the traffic from source to destination host. This route can take several hops over different intermediate switches before reaching the parent switch of the destination host. They can use their available ports for connecting either to other switches or hosts where they cannot connect to the same node twice. Each switch has a unique name, its own node id and knows about all nodes it is connected to.

Connection A connection in our model represents a link between two switches with a certain integer weight. The weight could also be considered a cost where higher numbers are worse than lower numbers. In our model connections are bidirectional so they do not have a designated source and destination. They also do not have an explicit id or name because a link is uniquely identified by the pair of nodes they connect. Each connection knows the ids for two switches and has a weight.

Update Batch We model updates of the topology connections by introducing the *update batch*. Each update batch consists of one list with *removals* and another list with *additions* of connections. This also covers weight update, which can be modeled by first removing the connection to be updated and then adding it again with the new weight.

Input Format For the definition of a topology by an external file we also created a simple syntax. We give an example input file in Listing 4.1 and will refer to it throughout this paragraph. The topology it describes is shown in Figure 4.1. In the syntax a switch is defined by the symbol star "`*`" followed by the switch's unique name. An example is given on lines 4, 7 and 10 of Listing 4.1. A host is defined by a dot "`.`" followed by the name of its parent switch and the new host's unique name preceded by a star "`*`" as can be seen in lines 5 and 6 among others. Connections are defined by providing both switch's node-ids and the weight of the connection enclosed in colons as on lines 17-19. Comments are supported as single line comments indicated by a double slash "`//`" and multi-line comments enclosed within "`/*`" and "`*/`" as in C-like languages.

Figure 4.1: Example topology created from input file in Listing 4.1



Listing 4.1: Example topology input file

```
/* Example Topology File
======================== */
// Switches and Hosts
*Switch_A
.Switch_A*Host_A1
.Switch_A*Host_A2

*Switch_B
.Switch_B*Host_B1
.Switch_B*Host_B2

*Switch_C
.Switch_C*Host_C1
.Switch_C*Host_C2

// Connections
Switch_A :3: Switch_B
Switch_A :1: Switch_C
Switch_B :2: Switch_C
```

4.2 Policy

In our model network policies are rules determining *constraints* on the paths packets in the network take. Packets are identified by the *flow id* attribute described in Section 4.2. In a real network flow ids can be constructed by packet headers, which requires only minor

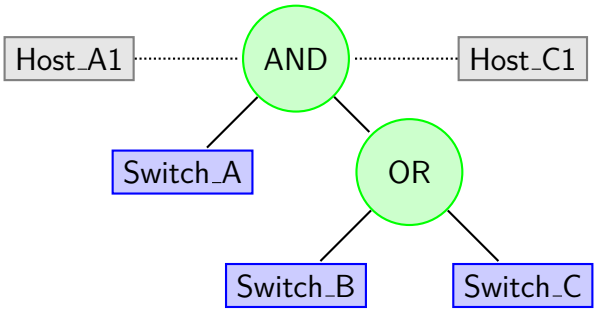
additions to our code. Each policy consists of a flow id and a constraint. Policies can be read from a file or created in arbitrary size as we describe in Subsection 5.6.1. The former has to contain the policy definition string in the syntax defined in the Paragraph *Language Syntax* below.

Flow Id As mentioned before, *flow ids* are classifiers for packets. They depict a certain type of packets being transmitted from one host to another. Each flow id contains packet type, source host and destination host. A flow id going from host a to b is distinct from a flow id going from host b to a. Every transmitted network packet always has exactly one flow id where all fields are properly defined. The packet type or destination field can also be a wildcard, such that a set of flow ids is represented.

Constraint The *constraint* in a policy constrains the path a packet is allowed to take through the network. It can define arbitrary many switches the packet must pass before reaching its destination host. A constraint has logical operations and also nodes on which those operations are executed on. The available operators are *OR* and *AND*. As an example they could be used to load balance certain paths in the network. So one packet stream could be instructed to pass through switches A and B, while another stream could be constrained to always go through switch C and D, which can help to distribute the load over several switches. If the network load is not know a priori, two different paths can be given and concatenated by the OR constraint, so the algorithm picks the one with the lower edge weight. Both operators have exactly two children which can be either a switch or recursively another operator. In Figure 4.2 we show the abstract syntax tree for the first policy from Listing 4.2. In this policy all packets `Host_A1` sends `Host_C1` first have to go through `Switch_A1` and then pass either `Switch_B` or `Switch_C` before reaching their destination `Host_C1`.

Language Syntax Same as with the topology we defined a policy syntax, which allows to import policies from an external file. The content of an example policy file is given in Listing 4.2. Each policy starts with the name of a source host, followed by a colon ":". Then the constraint follows, consisting of arbitrary many switch names concatenated with either dots "`.`" or pipes "`|`" representing *AND* and *OR* concatenation. Operator precedence is higher for the AND than for the OR. This means that `a | b . c` is translated into a constraint equivalent to `a | (b . c)`. Using parentheses to change the precedence is also

Figure 4.2: Abstract Syntax Tree for Policy 1 from Listing 4.2



possible. Another colon ":" follows when the constraint ends. Last part of a policy must be a host name depicting the flow destination. Comments are again possible, where multi-line ones are enclosed by "/*" and "*/", single-line comments start with "//" and ends until the line ending. Packet types are not supported by our syntax, although they are accommodated for in our model.

Listing 4.2: Example policy input file

```
/* Example Policy File
====================== */
Host_A1 : Switch_A . ( Switch_B | Switch_C ) : Host_C1   // Policy 1
Host_B2 : Switch_A | Switch_B . Switch_C : Host_C2       // Policy 2
```

5 Implementation

This chapter contains a detailed description of the code we wrote in the scope of this master thesis. The first section gives an overview of our code structure and explains what parts of the implementation can be found where in the folder structure. Section 5.2 shows the implementation for our model, in specific which data types we used and what functions the structs that implement our model provide. Section 5.3 introduces the policy and topology parsers. Subsequently, Section 5.4 describes the core of our project, how we compute the network paths with help of the timely and differential-dataflow libraries. In Section 5.5 we show how our program can be started from the command line and what the possible arguments are. Section 5.6 demonstrates what kinds of data we can generate to test and benchmark our program with. Finally Section 5.7 shows our measurement tool and its capabilities.

5.1 System-Overview

In this chapter we explain the implementation design, which part of the functionality we put at what place within the code base. This is necessary to understand the logical structure of our code, how components are interconnected and for what purpose. The two subsequent chapters describe the implementation of our controller logic in detail. Sections 5.2, 5.3 and 5.4 are about our project's library part. This is the core of the

thesis, the important control logic happen there. It contains the code responsible for parsing topologies and policies and also computes the routes while regarding the given topology and policies. In Sections 5.5, 5.6 and 5.7 we will then provide details about the interface to execute the code from the command line and all that belongs to it. This part contains tools we implemented, one of which is the benchmark tool we used to produce the measurements presented in Chapter 6.

Code Organization Our project has two code roots. The reason for this is that in Rust, code is either part of a library or an executable. When creating a new project, the programmer has to define which of the two project types he or she wants to build. The new project then consists of a root file which is either `src/lib.rs` for a library or `src/main.rs` for an executable project. In our case we have both: reusable code that belongs to a library as well as code that can be executed directly from the command line as a binary. Fortunately Rust allows to put both into the same project by just defining a `lib.rs` and additionally a `main.rs` leading to two code roots. From each of those roots, the programmer can define functions and modules, which can also be located in separate files and folders lower in the file hierarchy. Rust provides a separate code hierarchy for each of them, encapsulated and separated from each other. If functionality contained in the library part of the code is needed in the main part, it must be imported as if it was taken from an external library. This proved to be useful as it implicitly categorizes the code based on whether it is reusable library code or only needed for the execution.

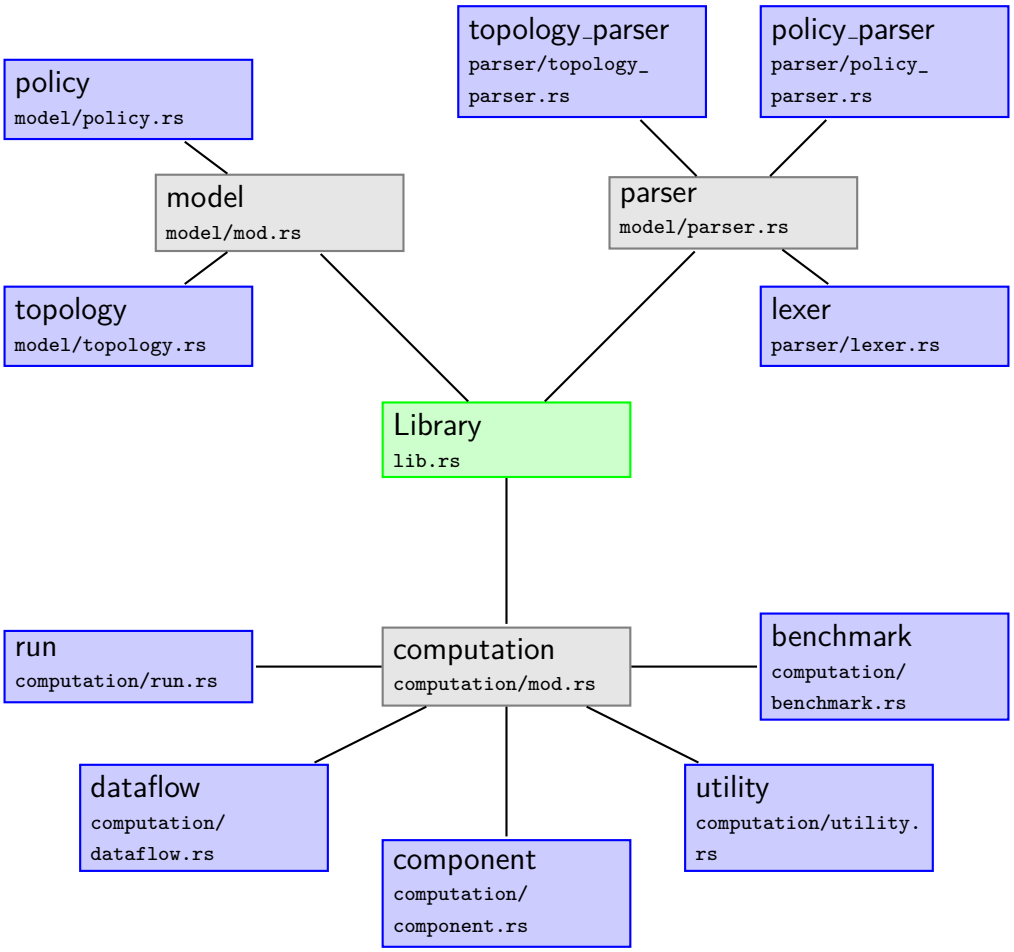
Rust also provides Cargo which acts as a build system and package manager[1]. For it to work properly it needs the file `Cargo.toml` to be present in the root of the project directory. This file contains the projects meta data and allows Cargo to compile the program and resolve all dependencies. The `Cargo.toml` file denotes the external dependencies used in the project, which can be imported from `http://crates.io` or directly from a public Git repository. It is also the place where we define the project's name, the author's name and tell Cargo to generate the documentation for the `lib.rs` and not the `main.rs` as described in Section 5.1.

Library - lib.rs The library part of our project contains all reusable functionality which is generic to multiple applications. It is usable like a library and offers a documentation generated with the `rustdoc` tool as described in Section 5.1. In the root file `src/lib.rs`

[1] `http://doc.crates.io/`

we state the external crates we use and define the first submodules: `model`, `parser` and `computation`. The first contains the code that belongs to the model, namely structs for the topology, policy and their related code. The `parser` module contains the policy parser, the topology parser and the lexer which both parsers use. Finally the `computation` module contains the code that calculates the actual paths and regards the constraints. This module uses timely- and differential-dataflow libraries.

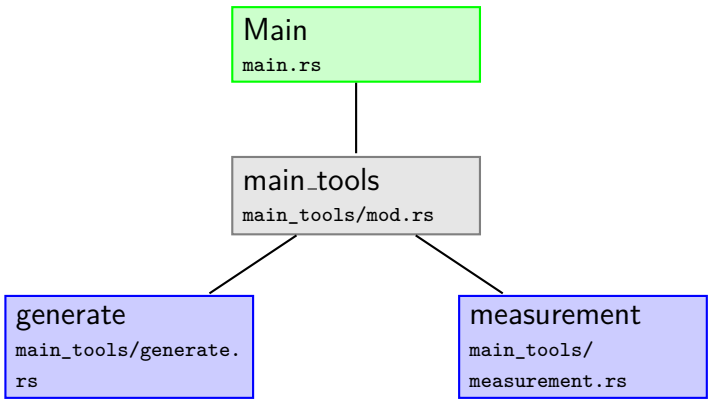
Figure 5.1: Overview Figure of all modules that are part of the code-root *library* with their code location. The root module is the library and colored green. It has three sub modules: model, parser and computation colored gray. Those themselves have submodules colored blue.



Executable - main.rs This part of the code is the entry point for executing our project from the command line. E.g. to run the code with a certain topology and policy or to run measurements. Currently the `main.rs` offers a command-line interface. If in the

future one may need a graphical user interface, it will be also located here. The root file for the executable is `src/main.rs`. Within this file itself the command line argument parsing is done and a normal execution of the SDN controller can be launched. The only module defined here is `main_tools`, which contains all the tools needed for the executable, most notably `generate.rs` and `measurement.rs`. The former offers generator functions for topologies, policies and update batches. The latter contains enhanced execution functions which allow for precise analysis of the library run times on different inputs and allows to pretty-print its results to the terminal.

Figure 5.2: Overview Figure of all modules that are part of the code-root *main* with their code location. The root module is the main and colored green. It has one submodule colored gray which itself has two submodules colored blue.



Code Documentation The Rust language offers two types of code comments: documentation comments and inline code comments. The `rustdoc` tool is able to generate a HTML document from documentation comments. Those comments can be above function definitions, structs, type definitions, module definitions, structs or enums and their respective fields. They cannot be within function bodies or directly before expressions in general. Lines containing documentation comments start with "`///`" compared to the "`//`" for normal comments. The comments can contain Markdown[2] to format the text layout which `rustdoc` then translates to HTML. Rust can only generate a HTML documentation for either the library or executable, so we chose to create one for the former. In our project we were only able to use documentation comments for the library part. We documented the executable directly in the source code with inline comments where necessary.

[2] `http://daringfireball.net/projects/markdown/`

5.2 Model

In the following we describe the implementation of our model as introduced in Chapter 4. This encompasses the topology and policy, which can be found at their locations `src/model/topology.rs` and `src/model/policy.rs`. We often use public fields instead of making them private and using setter and getter functions. The latter would also be reasonable but we opted for simplicity. This can easily be changed later.

5.2.1 Topology

The following Section describes the module `topology` which corresponds to our model for topologies described in Chapter 4. The module contains primitive type definitions, constants, the trait 'Node', the struct 'Host', the struct 'Switch', the struct 'UpdateBatch' and the struct 'Topology'.

Primitive Types To provide an abstraction for the primitive types we use and improve the code readability, we introduced new type names. Switches and hosts have unique node ids of type `NodeId`, which itself is a `u32`. Connections are selected by providing their `LinkId` internally also mapping to a `u32`. Packet types are identified by an `u32` as well. Connections weights or distances are represented by the type `Weight` which maps to a `u64`.

Constants We defined a few constants to represent placeholders if the value of some field is unknown or should represent all possible values. In place of a `NodeId` we also allow `const NODE_UNKNOWN` and `const NODE_WILDCARD` representing an unknown or all nodes respectively. The former is used when doing hops to the next node in the computation. For packet types `const PACKET_WILDCARD` is a possible placeholder. We use this when defining flow identifiers (see Section 4.2) for all packet types. It is also used in combination with `NODE_WILDCARD` to depict all flows going to a certain node without regarding the source. Finally, for weights we allow `const WEIGHT_UNKOWN`, which we use within the distance to a neighbor node is not yet known, e. g. directly after doing a hop.

'Node' Trait The `Node` trait is an abstraction for entities that can either be a host or switch. Every node has a name and unique node-id. So each type implementing the trait

has to offer the function `get_name()` and `get_node_id()` which return the node's name as a `&str` and its node-id as a `NodeId`.

```
pub trait Node {
    fn get_name(&self) -> &str;
    fn get_node_id(&self) -> NodeId;
}
```

'Host' Struct This struct stores data container for host entities. We outline its implementation in the code below. It offers fields for its node-id, name and parent switch's node-id. Also it implements the trait `Node` and additionally offers a constructor-like function `new()`, which makes the creation of new host objects less verbose.

```
pub struct Host {
    pub id: NodeId,
    pub name: String,
    pub switch_id: NodeId
}

impl Host {
    pub fn new(id: NodeId, name: String, switch_id: NodeId) -> Self { ... }
}

impl Node for Host { ... }
```

'Switch' Struct A `Switch` struct stores its node-id, name and also a set containing the node-ids it is connected to. We show the implementation outline in the code below. It contains a constructor-like function `new()` and three procedures to query or modify the switch's neighbors. `connect_to()` adds the given node-id to the list of nodes connected to the switch and panics if this connection already exists. `disconnect_from()` removes the given node-id from the same list and panics if the connection did not exist before calling the function. Finally `is_connected_to()` returns whether the switch is connected to the node with the given id. Because the struct also implements the trait Node, there also exist the functions to get the switch's node-id and name.

```
pub struct Switch {
    pub id: NodeId,
    pub name: String,
    pub connected: HashSet<NodeId>,
```

```
}

impl Switch {
    pub fn new(id: NodeId, name: String) -> Self { ... }
    fn connect_to(& mut self, other: NodeId) { ... }
    fn disconnect_from(& mut self, other: NodeId) { ... }
    pub fn is_connected_to(& self, other: NodeId) -> bool { ... }
}

impl Node for Switch { ... }
```

'Topology' Struct The `Topology` struct is the most extensive one within our project. It consists of three `Vec` objects named `hosts`, `switches` and `connections` which store the topology's nodes and links. We show the code that declares the struct below and in a separate block the implementation. To allow the access of switches and hosts by their names or node-ids, it also contains four maps, e.g. `host_name_i_map`, where `i` is an abbreviation for "index". The topology's implementation offers two constructor-like functions, one to initialize the underlying data-structures with certain sizes and another one if those sizes are not know a priori. To manipulate the topology's hosts, switches and connections, getter and setter functions are defined. Hosts and switches can be obtained by their node id, name or index among all hosts or switches respectively (e.g. the 3rd switch of the 10 available ones, we need this for the Jellyfish generator). The topology also offers query functions, to check if a host or switch with a certain name exists and of course functions to add new or remove existing hosts, switches and connections. To determine the topology size in regard to number of hosts, switches or connections, queries are available, e. g. `get_host_n()`, where emphn means "number".

```
pub struct Topology {
    id_counter: u32,
    hosts: Vec<Host>,
    switches: Vec<Switch>,
    connections: Vec<Connection>,
    host_name_i_map: HashMap<String, usize>,
    host_node_id_i_map: HashMap<NodeId, usize>,
    switch_name_i_map: HashMap<String, usize>,
    switch_node_id_i_map: HashMap<NodeId, usize>
}
```

```
impl Topology {
    pub fn new() -> Topology { ... }
    pub fn with_capacity(n_hosts: u32, n_switches: u32, n_connections: u32)
        -> Topology { ... }

    pub fn generate_node_id(&mut self) -> NodeId { ... }
    pub fn available_node_id(&self) -> NodeId { ... }

    pub fn get_host(& self, id: NodeId) -> &Host { ... }
    pub fn get_ith_host(& self, i: usize) -> &Host { ... }
    pub fn all_hosts(& self) -> Vec<Host> { ... }
    pub fn get_host_n(& self) -> usize { ... }
    pub fn get_host_node_id(& self, name: &str) -> Result<NodeId, String> { ... }
    pub fn has_host_with_name(& self, name: &str) -> bool { ... }
    pub fn add_host(& mut self, host: Host) -> Result<(), String> { ... }

    pub fn get_switch(&self, id: NodeId) -> &Switch { ... }
    pub fn get_ith_switch(&self, i: usize) -> &Switch { ... }
    pub fn all_switches(&self) -> Vec<Switch> { ... }
    pub fn get_switch_n(& self) -> usize { ... }
    pub fn get_switch_node_id(& self, name: &str) -> Result<NodeId, String> { ... }
    pub fn has_switch_with_name(& self, name: &str) -> bool { ... }
    pub fn add_switch(& mut self, switch: Switch) -> Result<(), String> { ... }

    pub fn all_connections(&self) -> &Vec<Connection> { ... }
    pub fn get_connection(&self, id: LinkId) -> &Connection { ... }
    pub fn get_connection_n(&self) -> u32 { ... }
    pub fn is_connected(& self, a: NodeId, b: NodeId) -> bool { ... }
    pub fn add_bidir_connection(& mut self, conn: Connection) { ... }
    pub fn remove_connection(& mut self, connection_id: LinkId)
    -> Connection { ... }
}
```

'UpdateBatch' Struct An `UpdateBatch` struct represents a set of changes to the topology consisting of removals and additions of connections. We define the struct in the file `topology.rs`. It is used to store changes that can be applied to an existing topology. To do this it contains two lists `removals` and `additions`. A creator function is there to allow a less-verbose initialization with two empty lists. Most notable of this struct is that it implements the trait `Display` to create a human-readable output explaining what its content is.

```
pub struct UpdateBatch {
    pub removals: Vec<Connection>,
    pub additions: Vec<Connection>,
}

impl UpdateBatch {
    pub fn new() -> UpdateBatch { ... }
}

impl fmt::Display for UpdateBatch { ... }
```

5.2.2 Policy

This section describes the module `policy` which corresponds to our policy model we describe in Chapter 4. It contains the struct 'FlowId', the struct 'Policy' and the enumeration 'Constraint'.

'FlowId' Struct The struct `FlowId` implements our representation of a flow id presented in Section 4.2. We show the code in the snippet below this paragraph. As described in the model, a flow id provides fields for the source and destination host in the form of node-ids and also a field storing the packet type as `PacketType`. The struct offers three creator-like functions. `some_from_to()` which creates a new flow id with all fields set. `all_from_to()` instantiates a flow id for packets of all types from a certain host to another one. Finally `all_to()` generates a flow id for all packets to a certain destination. We implemented the `fmt::Display` trait, which is used for a human-readable output of `to_string()` among others. All structs that should later be used as part of data-tuples in a differential-dataflow computation need to implement this trait. It automatically implements the trait `Abomonation` for a given struct.

```
pub struct FlowId {
    pub src_id: NodeId,
    pub dest_id: NodeId,
    pub packet_type: PacketType,
}

unsafe_abomonate!(FlowId : src_id, dest_id, packet_type);
```

```
impl FlowId {
    pub fn some_from_to(packet_type: PacketType, src: NodeId, dest: NodeId)
        -> FlowId { ... }
    pub fn all_from_to(src: NodeId, dest: NodeId) -> FlowId { ... }
    pub fn all_to(dest: NodeId) -> FlowId { ... }
}

impl fmt::Display for FlowId { ... }
```

'Policy' Struct The `Policy` struct consists of only two fields, the code implementing it is shown below this paragraph. Its first field `flow` stores which flow id the policy is concerning as type `FlowId`. The second field `constraint` contains the actual constraint and is of enum type `Constraint`.

```
pub struct Policy {
    pub flow: FlowId,
    pub constraint: Constraint
}
```

'Constraint' Enumeration The enumeration type `Constraint` implements our representation from Section 4.2. As can be seen in the code snippet below, a constraint can be one of four possible types. First it can be of type `Id(switch)` where switch denotes an element of type `Switch`. Also it can be one of the binary types `And` or `Or`, which recursively contain two boxed objects of type `Constraint` themselves. Finally it can also be of type `Not`, also wrapping a boxed constraint. The boxing is necessary so the element size is known at compile-time, otherwise the compiler will complain. We explain this detailed in Section 2.1 of the Chapter Preliminaries. To print the constraint in a format defined in the input file syntax we also implemented the `fmt::Display` trait. For pretty-printing the constraint-tree hierarchically, we also implement `fmt::Debug`.

```
pub enum Constraint {
    Id(Switch),
    And(Box<Constraint>, Box<Constraint>),
    Or(Box<Constraint>, Box<Constraint>),
    Not(Box<Constraint>)
}
```

```
impl fmt::Display for Constraint { ... }
impl fmt::Debug for Constraint { ... }
```

5.3 Parsers

We defined a specific syntax for policy input files in Section 4.2 and another one for topologies in Section 4.1. They are used as input to compute the routing tables. We now introduce the parsers we used to create our internal data types from this input. Both parsers are so called *recursive descent parsers*, which are top-down parsers that recursively call member functions until they reach an atom of the grammar. They build upon a lexer we built for this purpose, also described in this Section. We also use this lexer to parse the benchmark input file as described in Section 5.7.2.

'Result' Enumeration Throughout the following subsections we often refer to the enumeration type `Result` Rust provides. We show its declaration in Listing 5.1. This construct is intended to be a type used by functions that perform an operation that can possibly fail. It contains two possible types, `Ok(T)` and `Err(E)`. The types `T` and `E` define what kind of objects are returned in the case of success or an error.

Listing 5.1: Declaration of Rust's enumeration `Result` defined in `std::result`.

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

5.3.1 Lexer

The lexer is a foundation for the parsers. Its code is located in `src/parser/lexer.rs` within the project. We show the struct `lexer` declaration and implementation in a code snipped below. Also we present the token type our lexer is based on.

'Token' Enumeration Tokens are objects of enumeration type `Token`, shown in Listing 5.2. They constitute the smallest logical unit in our topology and policy syntax. A token can represent single symbols, strings acting as identifiers or also numbers. The `Token::Star` for example represents the symbol "*", `Number(U64)` is a number.

Listing 5.2: Declaration of the enumeration `Token`.

```
pub enum Token {
    Invalid(String),
    Id(String),
    Number(u64),
    Colon,
    Pipe,
    Dot,
    Not,
    ParenthesisOpen,
    ParenthesisClose,
    Star,
    Comma,
    End,
}
```

'Lexer' Struct The struct `Lexer` offers two fields as shown in the code snippet below. `chars` stores the string given to the lexer as a `Peekable` of characters. This is an iterator that allows to peek at the next element without consuming it. The second filed is `buffer` containing a Token. We used it to make our lexer peekable. The buffer always contains a Token, which is the next one that would be returned by the lexer, but can also be peeked without being returned. Upon retrieving an item (so not peeking it), our lexer returns the buffered token and puts the next token in its place. The `chars` field is also denoted by the lifetime parameter `'a`. It tells the compiler, that this field's content needs to outlive the lexer. This is part of Rust's lifetime concept and the reason it does not need a garbage collector. Without this remark the program would not compile.

```
pub struct Lexer<'a> {
    chars: Peekable<Chars<'a>>,
    pub buffer: Token,
}
```

Public Interface The public interface offers five functions in total. The first one is a creator-like function `new()`, takes a string as argument and returns a new lexer. `consume()`

moves its internal pointer to the next token and returns the old active one. `peek()` and `peek_clone()` only show the current token, without moving the pointer. When called repeatedly, they always return the same token or a reference to it. Last function of the public interface is `has()`, which returns a boolean depicting whether the input string's end was already reached. All methods returning tokens repeatedly pass `Token::End` after reaching the end. We show the headers of all functions, public and private, in the following code snippet.

```
impl<'a> Lexer<'a> {
    pub fn new(input_string: &'a str) -> Lexer<'a> { ... }

    pub fn has(& self) -> bool { ... }
    pub fn peek_clone(& self) -> Token { ... }
    pub fn peek<'b>(&'b self) -> &Token { ... }
    pub fn consume(& mut self) -> Token { ... }

    fn get_next_token(& mut self) -> Token { ... }
    fn skip_blanks_and_comments(& mut self) -> Option<Token> { ... }
    fn skip_blanks(& mut self) -> bool { ... }
    fn skip_comments(& mut self) -> Option<Token> { ... }
}
```

Internal Functions There are also four private helper functions. The first one is `get_next_token()`, which performs the actual work of converting the string into tokens. For this it utilizes the helper function `skip_blanks_and_comments()`, which skips comments, blanks, linefeeds and tab characters. The function `skip_comments()` tries to skip single-line and multi-line comments. Finally `skip_blanks()` skips all space, linefeed and tab characters and returns a boolean depicting whether any characters were skipped. This is necessary for `skip_comments_and_blanks()` to know, because after each skipped character, a new comment can start. If there was no character skipped, the function can return. All functions related to skipping comments return a `Token::Invalid(str)` in case of an error while parsing the input string, where `s` contains the encountered invalid character sequence. It is not possible to solve this in a different way, because `Peekable`, the trait we use to access the string, allows no lookahead.

5.3.2 Topology Parser

The topology parser allows to transform a valid input file into a `Topology` struct. We introduced the input file syntax in Section 4.1 and show the topology data type structure in Section 5.2.1. The code can be found in the file `src/parser/topo_parser.rs`, which denotes the module `topo_parser`.

Public Interface To use the parser we provide two public functions, both shown in Listing 5.3. They allow to either parse a file or a string containing the topology definition. After the parser finishes successfully, it returns an `Ok(topo)` result, where `topo` is of type `Topology`. When an error occurs during the parsing process, the functions return an `Error(message)` result, where `message` is a string containing an error description.

Listing 5.3: Public interface of module `topo_parser`.

```
pub fn parse(topo_string: &str, verbose: bool) -> Result<Topology, String> { ... }
pub fn parse_file(file: &str, verbose: bool) -> Result<Topology, String> { ... }
```

'TopologyParser' Struct Each struct `TopoParser` consists of two fields. The first one, `lexer`, stores a `Lexer<'a>` as defined in 5.3.1. `topo` contains the topology which is in the process of being created by the parser. The following code declares our topology parser struct:

```
pub struct TopologyParser<'a> {
    lexer: Lexer<'a>,
    topo: Topology,
}
```

Internal Functions To provide the interface we described previously, a set of internal non-public functions are required. We show their headers in the code below this paragraph. The creator-like function `new()` creates a new parser by accepting an object of type `Lexer`, introduced in Section 5.3.1. Then there are many functions parsing certain parts of our topology input syntax as defined in Section 4.1. They are all named after the same scheme, starting with `consume_` and ending with the construct name they parse. So for example the function `consume_host()` parses a host. Upon successful parsing the language construct,

the functions all return a result `Ok()` containing an object of the right type or nothing in case only a single symbol was parsed. When errors occur, an `Error(message)` is returned containing a string describing the error. Root function of all those individual parsers is `parse()`. It triggers the different `consume` functions repeatedly, until the lexer's input is consumed.

```
impl<'a> TopologyParser<'a> {
    pub fn new(lexer: Lexer) -> Parser { ... }

    pub fn parse(mut self) -> Result<Topology, String> { ... }

    fn consume_identifier(& mut self) -> Result<String, String> { ... }
    fn consume_number(& mut self) -> Result<u64, String> { ... }
    fn consume_colon(& mut self) -> Result<(), String> { ... }
    fn consume_star(& mut self) -> Result<(), String> { ... }
    fn consume_dot(& mut self) -> Result<(), String> { ... }
    fn consume_switch(& mut self) -> Result<Switch, String> { ... }
    fn consume_host(& mut self) -> Result<Host, String> { ... }
    fn consume_conn(& mut self) -> Result<Connection, String> { ... }
}
```

5.3.3 Policy Parser

The policy parser is built to parse a valid input file or string and output a list of policies. How a valid syntax is constructed we defined in Section 4.1. All components of the parser we describe here are located in the module `policy_parser` which itself is stored in the file `src/parser/policy_parser.rs`.

Public Interface Same as for the topology parser, the public interface consists of two functions as shown in Listing 5.4. One offers to parse a string and the other takes as argument the name of a file it tries to open and parse. They both return a `Result<Vec<Policy>,String>`, which means they pass back a list of policies after a successful parsing process or an error message in case something went wrong.

'PolicyParser' Struct We outline the struct `PolicyParser` in the code below this paragraph. As the topology parser, it contains a lexer to acquires the tokens from. Second field is a reference to the topology for which the policies should be parsed. It has lifetime

Listing 5.4: Public interface of module `policy_parser`.

```
pub fn parse(policy_string: &str, verbose: bool)
    -> Result<Vec<Policy>, String> { ... }
pub fn parse_file(file: &str, verbose: bool) -> Result<Vec<Policy>, String> { ... }
```

parameter `'b` meaning the referenced topology has to live at least as long as the parser, but is unrelated to the lexer, which has lifetime parameter `'a`.

```
pub struct PolicyParser<'a, 'b> {
    lexer: Lexer<'a>,
    topo: &'b Topology
}
```

Internal Functions The struct `PolicyParser` implements internal functions, comparable to the `TopologyParser` described previously. All function headers of the implementation are shown in the code snippet below. The function `new()` accepts a `Lexer` and reference to a `Topology` and returns a new `TopologyParser`. Top level function is the method `parse()`, which starts the parsing process. It returns a result, either `Ok(policies)`, where policies is a list of `Policy` objects, or `Err(message)` with a string containing an error message telling the user what went wrong while parsing. The other functions all start with `consume_` and each try to parse a certain syntax construct, much like the topology parser does, which we described in paragraph *Internal Functions* of Section 5.3.2.

```
impl<'a, 'b> PolicyParser<'a, 'b> { ... }
    pub fn new(lexer: Lexer<'a>, topo: &'b Topology) -> Parser<'a, 'b> { ... }

    pub fn parse(mut self) -> Result<Vec<Policy>, String> { ... }

    fn consume_node_identifier(& mut self) -> Result<String, String> { ... }
    fn consume_colon(& mut self) -> Result<(), String> { ... }
    fn consume_constraint(& mut self) -> Result<Constraint, String> { ... }
    fn consume_disjunction(& mut self) -> Result<Constraint, String> { ... }
    fn consume_conjunction(& mut self) -> Result<Constraint, String> { ... }
    fn consume_unary(&mut self) -> Result<Constraint, String> { ... }
    fn consume_switch_id(& mut self) -> Result<Constraint, String> { ... }
    fn consume_negation(& mut self) -> Result<Constraint, String> { ... }
    fn consume_enclosed_constraint(& mut self) -> Result<Constraint, String> { ... }
```

```
}
```

5.4 Computation

The following section describes the module `computation`, which is the core of our project because it calculates the routing rules. The root module is located at `src/computation/mod.rs` and contains the submodules `run`, `benchmark`, `dataflow`, `component` and `utility`. The last two contain the dataflow computation definition and are using the timely-dataflow and differential-dataflow Rust libraries. `benchmark` and `run` start those computations and provide them with input. The `utility` module contains helper functionality to configure the computation and translate the policies in actual tuples that are usable by the dataflow. Apart from those submodules, the root module additionally contains three re-exports shown below. They allow to run the computation by calling `computation::run()`, `computation::run_without_policies()` or `computation::benchmark()` which are the main entry-points for starting our routing algorithm.

```
pub use self::run::with_policies as run;
pub use self::run::without_policies as run_without_policies;
pub use self::benchmark::run as benchmark;
```

5.4.1 Run

This module is the entry point to calculate the shortest paths in terms of accumulated connection weights while respecting the given policies. It is contained in the file `src/computation/run.rs`. Both functions it offers are re-exported in the parent module `computation`, its headers are shown below. They both accept as arguments a topology, `usize` parameter `n_proc` defining the number of cores to use for the computation and in case of the first function a list of policies. The flag `verbose` instructs the program to create verbose output and `output_tables` prints the resulting routing tables after the computation is finished.

```
pub fn with_policies(topo: Topology, policies: Vec<Policy>, verbose: bool,
    output_tables: bool, n_proc: usize) { ... }
pub fn without_policies(topo: Topology, verbose: bool, output_tables: bool,
    n_proc: usize) { ... }
```

Implementation We show the implementation of function `with_policies()` in the code below this paragraph. The code in `without_policies()` is similar but does not regard a policy list in its computation. On the uppermost hierarchy it contains a call to `timely::execute()` which invokes a new timely-dataflow computation. This function takes as arguments an object of type `timely::Configuration` and a closure, which is Rust's concept of a lambda function. We generate the configuration object in our `utility` module, introduced in Section 5.4.5. The closure on the other hand is defined directly in the function code, it contains the dataflow definition, handles the data input and finally runs the computation. In the following we explain the code in detail by referring to the line numbers in the code example given under this paragraph. To create the dataflow object, on line 6 we run the function `dataflow::create()` which we explain in Section 5.4.3. Note that it takes as argument also the flag `output_tables` to include an output node to print the resulting forward tables if desired. On line 9-11 we then input the switches, connections and policies into the computation. Next we advance all inputs to timestamp 1 on lines 13-15, which tells the computation that no more inputs with earlier timestamps will arrive. Finally we run the computation by executing `computation.step()` as long as the probe has not processed all tuples of period 0, which is tested for with `probe.le(&RootTimestamp::new(0))`.

```
1  pub fn with_policies(topo: Topology, policies: Vec<Policy>, verbose: bool,
2      output_tables: bool, n_proc: usize)
3  {
4      let _ = timely::execute(utility::get_timely_config(n_proc), move |computation| {
5          let policies = policies.clone();
6          let (mut switch_input, mut edge_input, mut policy_input, probe) =
7              dataflow::create(verbose, output_tables, topo, computation);
8
9          dataflow::input_switches(&mut switch_input, topo, computation);
10         dataflow::input_connections(&mut edge_input, topo, computation);
11         dataflow::input_policies(&mut policy_input, policies, topo, computation);
12
13         switch_input.advance_to(1);
14         edge_input.advance_to(1);
15         policy_input.advance_to(1);
16         while probe.le(&RootTimestamp::new(0)) { computation.step(); }
17     });
18 }
```

5.4.2 Benchmark

This Section presents the module `benchmark`, which runs a computation and returns the time it took to execute the individual sub-computations. The measured steps are the time it takes to convert the policies into `PolicyTuples`, to calculate the shortest-path and generate the forward rules, to compute the additional forward rules for the policies and to apply an update to the topology. The module is located in `src/computation/benchmark.rs` and is similar to the `run` module introduced in the prior Section. We outline its only public function, `run()`, in the code snippet below. As mentioned before, this function is re-exported by the parent module and can be called with `computation::benchmark()`. It resembles the `with_policy()` function from module `run` which we introduced in the prior section. In the following we highlight the new code we introduced to allow the measurement of our dataflow computation's performance.

```
pub fn run(topo: Topology, policies: Vec<Policy>, updates: Option<Vec<UpdateBatch>>,
    verbose: bool, output_tables: bool, n_proc: usize) -> (i64, i64, i64) { ... }
```

Constants and Static Variables We introduced four fields in this module. The first one is the public constant `BENCH_MAX_MS`. It is the default value to which we initialize the fields later containing the actual measurements. The chosen value for this constant is `std::i64::MAX / 3`, because errors are easier to spot with a high number compared to using zero for this purpose. We divided the value by three so the later calculations do not overflow. The remaining three fields are static variables containing the ongoing measurement's results. At the time of writing the code there was no built-in method to return values from within a timely computation.

```
pub const BENCH_MAX_MS: i64 = std::i64::MAX / 3;

static mut RESULT_PARSE_MS: i64 = BENCH_MAX_MS;
static mut RESULT_DIJKSTRA_MS: i64 = BENCH_MAX_MS;
static mut RESULT_CONSTRAINTS_MS: i64 = BENCH_MAX_MS;
```

Private Functions The module also provides two helper functions. `get_total_ms()` calculates the sum of all three static fields and `reset_results()` resets them to the constant `BENCH_MAX_MS`.

```
fn get_total_ms() -> i64 { ... }
fn reset_results() { ... }
```

Storing Measurements We wanted to know the runtimes for individual sub parts of the computation, so we had to conduct them within the timely closure (lines 4-17 in the code of paragraph 'Implementation'). Writing to static variables is not thread-safe, which is why we had to wrap all our accesses to those fields in `unsafe{ ... }` blocks. Additionally we needed to make sure only one thread conducts the writes, so we wrapped it in a `if computation.index() == 0 { ... }`. The following is an example for such a write:

```
if computation.index() == 0 {
    unsafe {
        RESULT_DIJKSTRA_MS = (time::precise_time_ns() - start_d) as i64 / 1000000;
    }
}
```

Topology Updates One of the biggest advantages of the differential-dataflow framework is it's ability to efficiently process updates of the input data. In our case we wanted to measure how long it takes to process update batches as defined in Section 4.1. They contain removals and additions of edges, grouped in batches of arbitrary size. We measure how long it takes the dataflow computation to update the rules so they respect the new topology. Our implementation which we will refer to in the following is outlined in the code snippet below this paragraph. Each batch is input with a new timestamp and all updates within one batch use the same timestamp. After the input is done we notify each input handle that there will be no more tuples with timestamps lower or equal than the one we just used. Then we run the computation until all new tuples are processed on line 9. We omitted the remaining code, which is only responsible for handling the measurements.

```
1  for update in updates {
2      let start_ch = time::precise_time_ns();
3      dataflow::input_connections(&mut edge_input, &update.removals, computation);
4      dataflow::remove_connections(&mut edge_input, &update.additions, computation);
5
6      switch_input.advance_to(current_timestamp + 1);
7      edge_input.advance_to(current_timestamp + 1);
8      policy_input.advance_to(current_timestamp + 1);
9      while probe.le(&RootTimestamp::new(current_timestamp)) { computation.step(); }
10     ...
```

```
11 }
```

5.4.3 Dataflow

In this section we describe the Dataflow module located in `src/computation/dataflow.rs`. It contains functions to create new dataflow computations to compute forwarding rules and input data into them. For this it accepts the data types we created and outputs `Handle` types defined by the timely-dataflow library. In the following we show the function headers and explain what they are used for.

Creator Functions The module offers two different functions to create a new dataflow computation. `create()` accepts as parameters the list of switches, the flag `verbose` for verbose output, another flag `output_tables` and the parameter `computation`. This last parameter is a handle to the timely environment which we need to invoke new computations. The flag `output_tables` sets whether the resulting tables with the forwarding rules should be printed when the computation is done. As a result the function returns a tuple consisting of four fields, a probe and three input. The first one of those is a switch input, the second one is a connection input and the third one is a `ConstraintTuple` input. We introduce the type `ConstraintTuple` in Section 5.4.5.
The other available creator function is `create_without_policies()`, which again accepts the flag `output_tables`, a handle to the computation and a list of switches. The function returns a tuple, containing the same fields as before but without the input for tuples of type `ConstraintTuple`. The difference to the other function is the internal dataflow graph of the computation, which does not contain the additional logic needed to output the rules resulting from the policies.

```
pub fn create(verbose: bool, output_tables: bool, switch_list: Vec<Switch>,
    computation: &mut scopes::Root<timely_communication::allocator::generic::Generic>)
-> (
  input::Handle<Timestamp, (NodeId, i32)>, input::Handle<Timestamp, (Edge, i32)>,
  input::Handle<Timestamp, ((u32, FlowId, u32, u64), i32)>,
  probe::Handle<timely::progress::nested::product::Product<RootTimestamp, Timestamp>>)
{ ... }

pub fn create_without_policies(output_tables: bool,    switch_list: Vec<Switch>,
    computation: &mut scopes::Root<timely_communication::allocator::generic::Generic>)
```

```
-> (
  input::Handle<Timestamp, (NodeId, i32)>, input::Handle<Timestamp, (Edge, i32)>,
  probe::Handle<timely::progress::nested::product::Product<RootTimestamp, Timestamp>>)
{ ... }
```

Input Functions Our module offers three different input functions. They allow the input of connections, policies and switches into our dataflow computation. We list their headers in the code snippet below. They all accept as input parameter the computational scope and a list of the data it should remove. `input_connections()` takes a `std::Vec` of `Connection` objects. `input_switches()` accepts a `std::Vec` of `Switches`. Finally `input_policies()` takes a `std::Vec` of `ConstraintTuple` objects, described in Section 5.4.5.

```
pub fn input_connections(edge_input: &mut input::Handle<Timestamp, (Edge, i32)>,
    connections: & Vec<Connection>,
    computation: &mut scopes::Root<timely_communication::allocator::generic::Generic>)
{ ... }

pub fn input_policies(
    policy_input: & mut input::Handle<Timestamp, (ConstraintTuple, i32)>,
    policies: Vec<Policy>, topo: & Topology,
    computation: &mut scopes::Root<timely_communication::allocator::generic::Generic>)
{ ... }

pub fn input_switches(switch_input: & mut input::Handle<Timestamp, (NodeId, i32)>,
    switches: Vec<Switch>,
    computation: &mut scopes::Root<timely_communication::allocator::generic::Generic>)
{ ... }
```

Removal Function The remaining function is used to remove connections from the dataflow computation. It accepts the computational scope and a list of connections to be removed as parameters. We show its header in the following code:

```
pub fn remove_connections(edge_input: & mut input::Handle<Timestamp, (Edge, i32)>,
    connections: & Vec<Connection>,
    computation: &mut scopes::Root<timely_communication::allocator::generic::Generic>)
{ ... }
```

5.4.4 Component

The following section describes the module `component` located at `src/computation/component.rs`. This is the logical core of the project and a main contribution of this thesis. It contains definitions of differential-dataflow graphs, which we use to generate dataflow computations in module `dataflow`. We also define ten data types of which four are tuple types we use within the dataflow computation.

Type Definitions We grouped the type definitions of this module into tuple and primitive types. Both of them are shown in the code snippet below.
The primitive type `Timestamp` is used to identify input epochs in the timely computation. Each input tuple given to a computation is assigned exactly one timestamp. The three types `SwitchId`, `ForwardId` and `IntmdtId` are aliases of type `NodeId` which we introduced to improve code readability. A `SwitchId` is a node-id that can only belong to a switch. Objects of type `ForwardId` denote the node which comes next on the shortest path to the destination. Finally an `IntmdtId` is also the id of an intermediate switch that must be reached before going to the final destination of the packet. As such they are used in handling the constraints on the path. `Priority` is a type used for the priorities of rules where a higher number means a higher priority. We need priorities so we can partially override other rules. This is necessary for compliance to the given policies. `AccWeight` is a synonym for the type `Weight`. It is also intended to improve code readability and designates an accumulated weight. This means it can be the *distance* between two nodes in the graph that are not necessarily direct neighbors.
The tuple type `Edge` models a connection with certain `Weight` and two `NodeId` fields. A `FwdRule` depicts a routing rule to be deployed on a switch in the network. It contains the switch's node-id, a `Priority`, the packet-stream's `FlowId`, `Weight` and a `ForwardId`. The `IntermediateTuple` extends `FwdRule` with an `IntmdtId`, `AccWeight` and a `VariantId`. They are used within the computation of additional tuples resulting from the policies. If there exists a policy for a certain flow-id that contains at least one `OR` constraint, more than one possible path exist. We need an `IntmdtTuple` for every hop on all possible paths. To assign each tuple to exactly one path we use the `VariantId`. Section 5.4.5 explains how we translate policies to processable input tuples.

```
pub type Timestamp = u32;
pub type SwitchId = NodeId;
pub type ForwardId = NodeId;
pub type IntmdtId = NodeId;
pub type Priority = u32;
pub type AccWeight = Weight;

pub type Edge = (NodeId, NodeId, Weight);
pub type FwdRule = (SwitchId, Priority, FlowId, AccWeight, ForwardId);
pub type IntermediateTuple = (SwitchId, Priority, FlowId, IntmdtId, Weight,
    AccWeight, ForwardId, VariantId);
```

Functions The module defines three public functions, their headers are shown in the code snippet below. The first one is `get_fwd_rules()` which defines a dataflow computation that calculates the forward rules for a network graph given as node-ids and `Edge` tuples. The second function is `get_fwd_rules_for_constraints()` which defines a dataflow computation that calculates the additional forward rules needed so the routing respects the given policies. Finally the function `output_fwd_rules()` is used to output the calculated forward rules with Rust's built-in `println!` macro. All functions accept as parameter and also return as result a differential-dataflow `Collection`.

```
pub fn get_fwd_rules<G: Scope>(edges: &Collection<G, Edge>,
    dests: &Collection<G, NodeId>)
    -> Collection<G, FwdRule>
    where G::Timestamp: LeastUpperBound { ... }

pub fn get_fwd_rules_for_constraints<G: Scope>(fwd_rules: &Collection<G, FwdRule>,
    constraints: &Collection<G, ConstraintTuple>)
    -> Collection<G, FwdRule>
    where G::Timestamp: LeastUpperBound { ... }

pub fn output_fwd_rules<G: Scope>(fwd_tables: &Collection<G, FwdRule>,
    node_list: Vec<Switch>)
    -> Collection<G, FwdRule>
    where G::Timestamp: LeastUpperBound { ... }
```

Differential-Dataflow Computations Dataflow computations with the differential-dataflow library are based on the data type `Collection`. To create a new computation operator functions are applied to an input mapped to a `Collection`. This function itself

returns a `Collection` to which another operator can be applied to. We show the code doing this to create our dataflow programs below.

FwdRules Dataflow Program In the following we describe the function `get_fwd_rules()`. It defines the dataflow we use to compute the forward rules. The parameters it accepts are two `&Collection` objects, one containing `Edge` objects and the other node-ids. We show the code of the function body below and will refer to it in this paragraph.
To decrease the resource consumption we use the intermediate tuple format `(SwitchId, NodeId, Weight, FwdId)` depicting a forward rule. The first field denotes the switch this rule concerns, second is the destination id, third is the distance from the switch denoted in the first field and the last field identifies the next node on the shortest path to the destination. We use it instead of type `FwdRule` which contains fields we do not need for the computation. This saves a lot of memory because the tuple is used to store huge amounts of data in a typical run.
Now we describe the computation itself, which is an implementation of Dijkstra or Breadth-First-Search respectively. At first the function initializes the computation by creating forward rules for each switch saying it can reach itself with zero distance. Then it repeatedly updates the list of reachable switches including distance for each switch to all others. This iteration continues until a fixed point is reached. In detail on lines 13-16 it joins the list of edges with the list of existing forward rules. This propagates for each `Switch_X` all nodes it reaches to all `Switch_Y` to which it is connected. On lines 20-22 it then groups them by switch-id and destination pair and keeps only the tuple with shortest distance for all possible pairs. This sorts them by distance and only keeps the tuples with shortest one for each possible switch and destination. After the iteration it maps the intermediate tuples to our public `FwdRule` format. All forward rules get the priority zero, because they are all unique and may be overwritten by policy-specific rules later. They also concern all types of packets, so we assign `FlowId` objects created with `FlowId::all_to()`.

```
1  pub fn get_fwd_rules<G: Scope>(edges: &Collection<G, Edge>, dests: &Collection<G, NodeId>)
2      -> Collection<G, FwdRule>
3      where G::Timestamp: LeastUpperBound
4  {
5      let nodes = dests.map(|x| (x, x, 0u64 as Weight, x));
6
7      nodes.iterate(|inner| {
8          let edges = edges.enter(&inner.scope());
9          let mapped_edges = edges.map( |(x,y,w2)| (x,(y,w2)) );
10
```

```
11          let nodes = nodes.enter(&inner.scope());
12
13          inner
14          .map( |(x,dest,w1,fwd)| (x, (dest,w1,fwd)))
15          .join_u(&mapped_edges)
16          .map( |(x, (dest,w1,_), (y,w2))| (y,dest,w1+w2,x) )
17
18          .concat(&nodes)
19
20          .map( |(n,dest,w,fwd)| ((n,dest),(w,fwd)) )
21          .group( |_,v,out| out.push( (*v.peek().unwrap().0,1) ) )
22          .map( |((n,dest),(w,fwd))| (n,dest,w,fwd) )
23      })
24      .map(|(x,dest,cost,fwd)| (x,0,FlowId::all_to(dest),cost,fwd))
25  }
```

FwdRules for Constraints Dataflow Program In this paragraph we present the function `get_fwd_rules_for_constraints()`. It defines a dataflow computation to generate additional forward rules needed so the given policies are respected in the routing. To do this it breaks up the path between the source and destination host into sub-paths. For each intermediate switch that has to be passed, additional forward rules with higher priority are inserted to lead the stream along this new path instead of the shortest direct connection between them. The accepted parameters are a `&Collection` of `FwdRule` tuples calculated by `get_fwd_rules()` and another `&Collection` of `ConstraintTuple` entities, which are generated by `utility::get_constraint_tuples()` described in Section 5.4.5. In contrast to the function `fwdRules()` introduced before, we now store intermediate results in local variables and denote their type explicitly to document the code. In the code segment below we removed comments and code that only prints intermediate results to make it shorter and easier to read.

During the computation we use the tuple format `IntermediateTuple`. To join different `Collection` objects we use the tuple type `((SwitchId, SwitchId), *)` where "`*`" is a wildcard. There the first `SwitchId` denotes the switch this tuple concerns and the second one is the next destination to go to. We also use `ConstraintTuple` objects as an input parameter and within the computation. As an example let us consider a policy concerning a flow id from `Host_A` to `Host_B` asking to go through `Switch_1` and then `Switch_2`. To respect it we have to create forward rules instructing to first go from `Host_A` to `Switch_1` and then to `Switch_2` before heading to the final destination `Host_B`.

The computation first initializes the set of additional forwarding rules with the given constraint tuples and assigns them the `ForwardId` specifying the next hop to their respective

`IntermediatId`. It does this by joining the given constraint tuples with the default forwarding rules on lines 9-13. The function then enters a fixed-point iteration, following the `ForwardId` for each tuple to generate new forward rules. For this it first conducts a hop to the tuple's `ForwardId` and then filters out all tuples that reached their destination on lines 19-23. Then on lines 25-29 it again assigns the next `ForwardId` by joining all new tuples with the `FwdRule` tuples given as parameter. At the end of each iteration, we concatenate the new tuples with the previously generated ones on line 31. After the iteration our algorithm selects the best variant for each policy and returns only the tuples belonging to those variants. This is done by first evaluating the best variant for each policy on lines 34-39. Then the result tuples are filtered to only contain the best variants on lines 41-45. Finally the function converts the data to `FwdRule` tuples and returns them on line 47.

```
1  pub fn get_fwd_rules_for_constraints<G: Scope>(fwd_rules: &Collection<G, FwdRule>,
2      constraints: &Collection<G, ConstraintTuple>)
3      -> Collection<G, FwdRule>
4      where G::Timestamp: LeastUpperBound
5  {
6      let mapped_fwd_rules:           Collection<_, ((SwitchId, SwitchId), (ForwardId, Priority, Weight))>
7          = fwd_rules.map( |(n,p,flow,dist,fwd)| ((n,flow.dest_id),(fwd,p,dist)) );
8
9      let mut new_fwd_rules:          Collection<_, IntermediateTuple>
10         = constraints
11         .map( |(n,flow,i,v_id)| ((n,i),(flow,v_id)) )
12         .join(&mapped_fwd_rules)
13         .map( |((n,i), (flow, v_id), (fwd,p,dist))| (n,p+1,flow,i,dist,0,fwd,v_id) );
14
15     let mut new_fwd_rules = new_fwd_rules.iterate(|inner| {
16         let mapped_fwd_rules = mapped_fwd_rules.enter(&inner.scope());
17         let new_fwd_rules = new_fwd_rules.enter(&inner.scope());
18
19         let after_hop:              Collection<_, IntermediateTuple>
20             = inner
21             .map( |(_,p,flow,i,dist,acc_dist,fwd,var)|
22                 (fwd,p,flow,i,WEIGHT_UNKNOWN,acc_dist+dist,NODE_UNKNOWN,var) )
23             .filter(|&(n,_,_,i,_,_,_,_)| n!=i );
24
25         let mut new_new_fwd_rules:  Collection<_, IntermediateTuple>
26             = after_hop
27             .map( |(n,p,flow,i,_,acc_dist,_,var)| ((n,i),(flow,p,acc_dist,var)) )
28             .join(&mapped_fwd_rules)
29             .map( |((n,i), (flow,p,acc_dist,var), (fwd,_,dist))| (n,p,flow,i,dist,acc_dist,fwd,var) );
30
31         new_fwd_rules.concat(&new_new_fwd_rules)
32     });
33
34     let mut best_variants:          Collection<_, ((FlowId, VariantId), AccWeight)>
```

```
35          = new_fwd_rules
36          .filter( |&(_,_,flow,_,_,_,fwd,_)| fwd == flow.dest_id )
37          .map( |(_,_,flow,_,dist,acc_dist,_,var)| (flow, (acc_dist+dist, var)) )
38          .group( |_,v,out| out.push( (*v.next().unwrap().0,1) ) )
39          .map( |(flow, (acc_dist, var))| ((flow, var), acc_dist ) );
40
41      let mut best_fwd_rules:         Collection<_, IntermediateTuple>
42          = new_fwd_rules
43          .map( |(n,p,flow,i,dist,acc_dist,fwd,var)| ((flow, var), (n,p,i,dist,acc_dist,fwd)) )
44          .join(&best_variants)
45          .map( |((flow,var), (n,p,i,dist,acc_dist,fwd), _)| (n,p,flow,i,dist,acc_dist,fwd,var) );
46
47      best_fwd_rules.map( | (n,p,flow,_,dist,_,fwd,_) | (n,p,flow,dist as u64,fwd) )
48  }
```

5.4.5 Utility

The module `utility` is located in the file `src/computation/utility.rs`. Its purpose is to contain common functionality that does not fit into the other modules. This includes the function used to convert policies into tuples our dataflow computation can process. The module contains of two public and an additional private function. Its interface also encompasses two public data type definitions. We explain all entities below and show the code defining them.

Data Types The two new data types introduced in this module are `VariantId` and `ConstraintTuple`. We show both in the code below. Type `VariantId` is an alias for `u64` and depicts a variant introduced by an `OR` operator. The second type's name is `ConstraintTuple` and represents a Tuple defined as `(SwitchId, FlowId, IntmdtId, VariantId)`. Objects of type `ConstraintTuple` are used for computing additional routing rules that result from the given policies. We called them constraint rather than policy tuples, because the function generates an individual tuple for each constraint in the policies.

```
pub type VariantId = u64;
pub type ConstraintTuple = (SwitchId, FlowId, IntmdtId, VariantId);
```

Functions Also part of the same model is one private and two public functions outlined in the code snippet below this paragraph. `get_timely_config()` creates a new `timely::Configuration`, which is required to launch a timely-dataflow computation and tells the library how many threads should be used. The second public function has the name

`get_constraint_tuples()` and outputs a `Vec` containing entities of type `ConstraintTuple` created from a list of `Policy` entities. For this it uses the private function `parse_constraint()`.

```
pub fn get_timely_config(n_proc: usize) -> timely::Configuration { ... }

pub fn get_constraint_tuples(policies: Vec<Policy>, topo: &Topology)
    -> Vec<ConstraintTuple> { ... }

fn parse_constraint(topo: &Topology, flow: FlowId, constraint: Constraint)
    -> Vec<Vec<SwitchId> > { ... }
```

Meaning of 'ConstraintTuple's As we previously described, constraint tuples depict the type `(SwitchId, FlowId, IntmdtId, VariantId)`. The `SwitchId` marks which switch is concerned by the tuple. This switch has to forward packets belonging to the defined `FlowId` not to to its final destination but first to an intermediate switch with the node-id `IntmdtId`. Because each `OR` constraint doubles the total number of possible different paths, we also need `VariantId`, which gives each possible path a unique id.

Creation of 'ConstraintTuple's We create the constraint tuples with help of the function `parse_constraint()` shown below. It gets as input a reference of the topology, flow-id and the constraint. The returned result is a set of lists containing ids. Each contains all ids of a path variant in the order they should be traversed by packets leaving the source before they reach the destination. To create those lists `parse_constraint()` makes a case-distinction on the given constraint. If it is a `Constraint::Id()` the function returns a new set with exactly one list containing a single id. In case it is a `Constraint::Or()` the function recursively evaluates the set of lists for both of its children and then merges them. Finally if it is an `Constraint::And()` the function again recursively evaluates the set of lists for both children and fills a new set with all possible pairs combining one list of the one set with another list from the other set. To show that this is correct, we provide a small example in the following paragraph.

```
fn parse_constraint(topo: &Topology, flow: FlowId, constraint: Constraint)
    -> Vec<Vec<SwitchId> >
{
    match constraint {
        Constraint::Id(node)            => {
            let intermediate_list = vec!(node.id);
```

```
            let table = vec!(intermediate_list);
            table
        },
        Constraint::And(box1, box2)     => {
            let table1 = parse_constraint(topo, flow, *box1);
            let table2 = parse_constraint(topo, flow, *box2);
            let mut new_table = Vec::<Vec<SwitchId> >::new();
            for intermediate_list1 in &table1 {
            for intermediate_list2 in &table2 {
                let mut new_list = intermediate_list1.clone();
                for intermediate_id in intermediate_list2 {
                    new_list.push(*intermediate_id);
                }
                new_table.push(new_list);
            }}
            new_table
        },
        Constraint::Or(box1, box2)  => {
            let mut table1 = parse_constraint(topo, flow, *box1);
            let table2 = parse_constraint(topo, flow, *box2);
            for intermediate_list in table2 {
                table1.push(intermediate_list);
            }
            table1
        },
    }
}
```

Example 'ConstraintTuple' Generation In the following we give an example of generating constraint tuples. For this we came up with a policy shown in Figure 5.3 and the related topology shown in Figure 5.4. The policy instructs packets to go through Switch_A and then either to Switch_B or Switch_C before going to its destination Host_B. When our function `parse_constraint()` evaluates the given policy, it starts with the `AND` constraint. It evaluates both children, one being a set with one list containing a single id "5" and the other being the `OR` constraint. This `OR` constraint evaluates to a set of two lists, one containing id "6" and the other "7". The parent `AND` operator then merges the set containing the list with id "5" with the two new lists resulting in a set with two lists. One of those resulting lists contains the ids "5" and "6" and the other "5" and "7". Those two lists depict the two possible path variants and result in the `ConstraintTuples` shown in

Figure 5.3: Abstract Syntax Tree for `Host_A : Switch_A . (Switch_B | Switch_C) : Host_B`

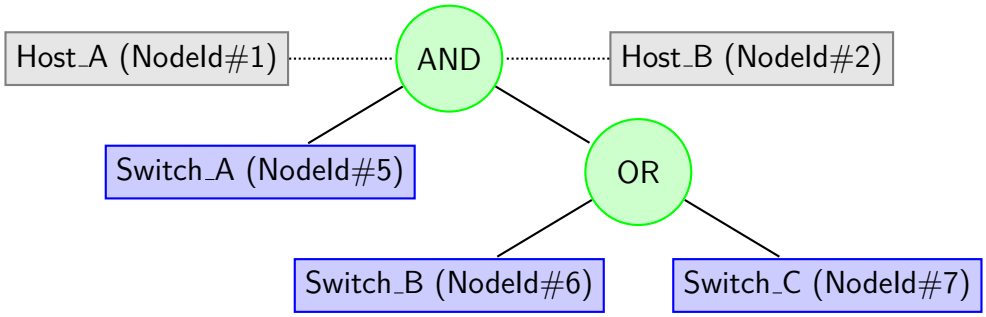


Figure 5.4: Example topology used for policy in 5.3

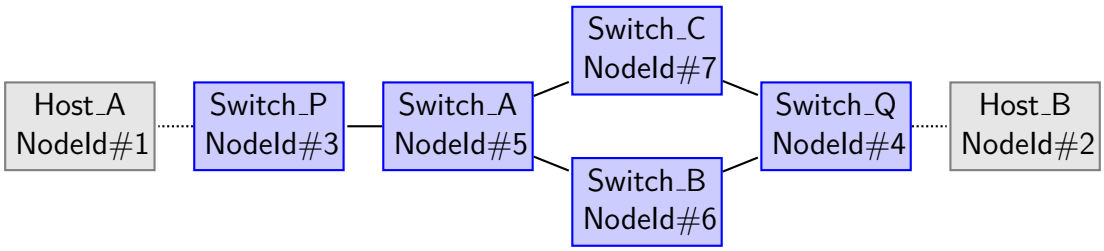


Table 5.1.

SwitchId	FlowId	IntmdtId	VariantId
3	src:1,dest:2,packet_t:*	5	1
5	src:1,dest:2,packet_t:*	6	1
6	src:1,dest:2,packet_t:*	4	1
3	src:1,dest:2,packet_t:*	5	2
5	src:1,dest:2,packet_t:*	7	2
7	src:1,dest:2,packet_t:*	4	2

Table 5.1: All `ConstraintTuples` resulting from policy in Figure 5.3.

5.5 Execution

In Section 5.1 we explain the split into library and execution part. This following Section contains description of the latter in detail. We describe the command line invocation, how the command-line arguments are parsed and what commands with which parameters are

available in Section 5.5.1. Next we give details about the tools we created for generating topologies and policies in Section 5.6 and for conducting the measurements in Section 5.7.

5.5.1 Main

As already mentioned in Section 5.1 the entry point for our project is the file `src/main.rs`. When invoking our program by executing `cargo run` or the compiled executable directly, the `main()` function in this file is called. This invokes the command-line argument parser and then calls the appropriate tool or sub function with the necessary arguments. We give an example invocation in Listing 5.5, where the program is compiled with all optimizations and then run. The parameters are explained in Section 5.5.2.

5.5.2 Command-Line Interface

At the beginning of the project we started out by parsing the command-line arguments directly in the called functions. This proved to be error-prone and hard to maintain. Rather than implementing a command-line parser ourselves we used the crate *docopt*[3] by adding it to our Cargo.toml. This is a Rust implementation of Docopt, a command-line interface description language[4]. The idea behind Docopt is to automatically generate a parser from the interface description, which itself must be formatted in a certain way. This format corresponds to a typical human-readable usage description used by various command line tools. This command-line interface usage description then also serves as the help output shown to the user when requested through the `--help` flag or when the parser encounters errors. We show the complete interface description for our executable in Listing 5.7. The parser created by the *docopt* crate also needs a struct type named `Args` containing fields for all possible options. It then returns an instance of this type containing all the parsed options or their standard values defined in the interface description respectively.

Invocation Example An example invocation is given in Listing 5.5 below. To compile and run the code with all compiler-optimizations, the `--release` flag is used. The remaining specified parameters are `topology.in` as topology input file, `policies.in` as policy input file, `-C8` to run the program with 8 cores and `-o` to output the resulting forwarding rules.

[3] `https://github.com/docopt/docopt.rs`
[4] `http://docopt.org/`

Listing 5.5: Terminal command to run differential-sdn on 8 cores and output the result.

```
$ cargo run --release -- topology.in policies.in -C8 -o
```

Benchmark Example The other example in Listing 5.6 shows a benchmark run. To compile and run the code with all compiler-optimizations, the `--release` flag is used. There other used parameters are a `bench` command, `--fat-tree` flag to measure the performance on a Fat-Tree topology with parameter `-k8` which means 8 ports per switch, `-r100` to create 100 policies and `-C4` to execute the benchmark on 4 CPU cores.

Listing 5.6: Terminal command to benchmark differential-sdn on 4 cores with a fat tree topology, k=8 and 100 policies.

```
$ cargo run --release -- bench --fat-tree -k8 -p100 -C4
```

Listing 5.7: Command-Line Interface Description: Usage and Options.

```
Usage:
    differential-sdn bench INPUTFILE [-R N | --runs=N] [-w N | --max_weight=N]
    differential-sdn bench ((--fat-tree -k N) | (--jellyfish (-n N | --hosts=N)
        (-s N | --switches=N) (-k N | --ports-per-switch=N)))
        [-w N | --max-weight=N] [((-p N | --policies=N)
        [--policy-length=L] )] [-C N | --cores=N] [-R N | --runs=N]
        [-v | --verbose] [-o | --output-result]
        [((--removal-batches=N | --update-batches=N) [--batch-size=N])]
    differential-sdn parse TOPOFILE [POLICYFILE] [-v | --verbose]
        [-o | --output-result]
    differential-sdn generate-topo ((--fat-tree -k N) |
        (--jellyfish (-n N | --hosts=N) (-s N | --switches=N)
        (-p N | --ports-per-switch=N)))
        [(-w N | --max-weight=N)] [(-o | --output-result)] [(-v |--verbose)]
    differential-sdn TOPOFILE [POLICYFILE] [-C N | --cores=N]
        [-v | --verbose] [-o | --output-tables]
    differential-sdn -h | --help

Options:
    -h, --help                  Show this screen.
    -v, --verbose               Enable verbose output.
    -o, --output-result         Enable output of tables / topology / rules.
    -R N, --runs=N              Number of Runs [default: 1].
    -C N, --cores=N             Number of cores to use [default: 1].
    -n N, --hosts=N             Number of hosts.
    -s N, --switches=N          Number of switches.
    -e N, --edges=N             Number of edges.
    -w N, --max-weight=N        Maximum edge weight [default: 100].
    -p N, --policies=N          Number of rules [default: 0].
    -k N, --ports-per-switch=N  Number of ports per switch, k-ary Fat-Tree topology.
    --policy-length=L           Length of rules [default: 4].
    --batch-size=N              Updates per batch [default: 2].
    --removal-batches=N         Number of removal batches.
    --update-batches=N          Number of update batches.
    --jellyfish                 Create a Jellyfish topology.
    --fat-tree                  Create a Fat-Tree topology.
```

5.6 Generate

To benchmark our work extensively we developed several tools to create input for our program. The advantage of dynamically creating the data is that we can choose its size arbitrarily. This especially helps with being flexible when evaluating the computation time against the problem size. Our tools allow the creation of Jellyfish and fat tree topologies, policies with arbitrary constraint lengths and update batches.

5.6.1 Topologies

In the following we explain how we create the two topologies our generator supports, Fat-Tree and Jellyfish. We picked these two because they are based on completely different concepts. Fat-Tree topologies are well adopted in the industry and Jellyfish topologies represent the theoretic optimum in regards to resources used and path-lengths. Details on the topology layouts can be found in Section 6.1.3.

Jellyfish The jellyfish topology generator takes as arguments the number of hosts, switches and ports per switch. It then generates a jellyfish topology as specified by Singla et. al. [SHPG12]. We simplify the topology building process by only supporting switches with the same number of ports within one topology. Also we do not support the later modification of the topology, such as the later insertion of additional switches. To get a functional network it should be built with considerably less hosts than are possible in theory, e. g. 2/3 of the maximum $\#Switches * \#Ports\_per\_Switch$.

Fat Tree The fat tree topology generator takes as argument only the parameter k. K defines the number of ports per switch and is the main defining constant used when constructing a Fat-Tree topology. We then construct the biggest possible topology in regards to k to test the performance boundaries. If our algorithm performs well with this input, a smaller topology would also not be a problem. We list the possible number of switches and hosts in Table 5.2 below.

k / #Ports	4	8	16	32	48	92
Switches	20	80	320	1280	2880	10580
Hosts	16	128	1024	8192	27648	194672

Table 5.2: Topology size in regard to k

5.6.2 Policies

We can also generate policies for any given topology. The policy constraint's length is chosen randomly and lies between the `min_rule_length` and `max_rule_length` which are provided as parameters. In case the length should be the same for all rules, the min and max parameter can be the same number. To keep it simple, we only generate AND constraints. This is enough to conduct all the measurements we desired, because OR constraints also get translated to AND constraints before the computation is actually done. The full reasoning for this is provided in Section 6.1.2.

5.6.3 Update Batches

To simulate connection removals or weight updates in the network topology, we introduced our representation of *update batches* in Section 4.1. An update batch is a set of connection additions and removals used to model updates of a topology. We provide a function to create removal batches which remove connections as well as a function to create update batches which alter edge weights or completely replaces some connections with new ones. Both take as parameters how many batches are desired, how many updates or removals should be done per batch and a topology. The update batch generator additionally needs the desired maximum new edge weight. They both return a list of elements with type `UpdateBatch`, where each consists of a removal and an addition list, as described in Chapter 4. To remove or change data in a differential-dataflow program, one has to provide the exact same data tuple that was given to the computation as input earlier. So we select a random connection in the provided topology and put it in the removals list. In the case of an update batch we put the old connection with a randomly picked new weight in the additions list.

5.7 Measurement

While writing the code and this thesis we conducted many measurements. The tool introduced in the following helped us to do reliable validation and performance testing by conducting benchmarks. It outputs them in tables which saves time and helps to obtain reproducible results. To run quick tests we implemented a function to run a single benchmark by providing the parameters directly on the command line. For more detailed tests we created a specific benchmark input file format and a function executing benchmarks with parameters read from a file with this format. Benchmark results are internally stored and evaluated with the help of two structs, `BenchmarkResult` and `BenchmarkStatistic`.

5.7.1 Run with Command Line Arguments

To quickly run measurements of our program one can use `bench` and provide parameters directly on the command line. We already gave an example invocation in Listing 5.6 of how to run benchmarks for Fat-Tree topologies. In Listing 5.8 we give another example, but this time for a jellyfish topology. The arguments are a `bench` command and the `--jellyfish` flag with parameters `-n100` for 100 hosts, `-s10` for 10 switches and `-k16` for 16 ports per switch. Also we use the flag `-r100` to create 100 policies and `-v` and `-o` to make the output verbose and print the result when finished. We show the full interface in Listing 5.7.

Listing 5.8: Terminal command to benchmark differential-sdn with a jellyfish topology and 100 policies with verbose output and printing results when done.

```
$ cargo run --release -- bench --jellyfish -n100 -s10 -k16 -p100 -v -o
```

5.7.2 Run with File Arguments

The other possibility to run benchmarks is to write an input file containing the parameters. This allows to run a whole set of measurements sequentially without user intervention. We allow to define an arbitrary number of benchmark sets in a single file. To further improve the process we also allow parameter ranges to be used in the input files. This means that instead of only providing a single parameter, a set of several different ones can be given. E.g. the user can specify to run it with 1, 2, 4 and 8 cores instead of having to create a

separate benchmark set for each of those. When several parameter ranges are given, all possible combinations are measured.

Execution example Listing 5.9 shows an invocation example of a benchmark using an input file. The used parameters are a `bench` command, `benchmark.in` as input file for the benchmark-parameters and `-R5` to instruct the tool to run each measurement 5 times and output the minimum, maximum and average time measured.

Listing 5.9: Terminal command to benchmark differential-sdn through 5 runs on the parameters provided in the file `benchmark.in`.

```
$ cargo run --release -- bench benchmark.in -R5
```

Input File Syntax Listing 5.10 shows an example for a possible benchmark input file. We will refer to this example throughout the paragraph. Using comments in the file is very useful to give it structure. Single-line comments start with "`//`" and multi-line comments are enclosed within "`/*`" and "`*/`". Each benchmark input file can contain arbitrary many input parameter sets, which provide the parameters in a certain defined order. In the example we have two such sets, one going from line 1 to 5 and the other from line 7 to 14. Every parameter within a set can either be a range as on line 2 or a single number as on line 3. The former consists of several numbers separated by commas ",". First parameter of each set is always the topology type, which is either `fattree`, `jellyfish` or `j_fixhost`. `j_fixhost` also constitutes the jellyfish topology, but with a fixed number of hosts. For FatTree topology types the following parameter ranges must be the number of cores followed by k, the amount of policies and the policy length. For jellyfish topology types the order is number of cores, switches, ports per switch, policies and the policy length. When using a fixed number of hosts, they are given before the policies, as seen on line 12 in the example.

Listing 5.10: Example benchmark parameter input file.

```
/* Topo     */ fattree
/* Cores    */ 1, 2, 4, 8, 16, 32, 64
/* k        */ 32
/* Policies */ 100000
/* P_len    */ 2, 4, 10

// fat tree topology with k=32 has 1280 switches and 8192 hosts
/* Topo             */ j_fixhost
/* Cores            */ 1, 2, 4, 8, 16, 32, 64
/* Switches         */ 1280
/* Ports per Switch */ 32
/* Hosts            */ 8192
/* Policies         */ 100000
/* P_len            */ 2, 4, 10
```

Parameter Set Parser To support the input file syntax as depicted above, we implemented three small parsing functions, building upon the same lexer as our topology and policy parser introduced in Section 5.3. The functions `consume_number()` and `consume_id()` try to parse a number or an identifier respectively. `consume_range()` also tries to consume a number and if the following character is a comma "," it consumes the comma and then tries to parse another number. It continues to do so until the character after a consumed number is not a comma anymore. Then it returns a vector of all parsed numbers, representing the parameter range. In case any of the above functions fail during parsing they prematurely exit the program with a short error message.

5.7.3 Data Types

In order to model the benchmark results, we created two new structs. `BenchmarkResult` stores the run times of a single benchmark run. To combine several of those, we also created the struct `BenchmarkStatistic`, which combines all benchmark runs with the same parameters. We describe them in detail below.

Benchmark Result As mentioned before, a benchmark result corresponds to a single run with a certain set of parameters. Note that this means a single actual parameter for

each possible one and does not include ranges as described above. It stores the run times of all the individual stages of the computation. This includes parsing the policy, converting it to dataflow tuples, calculating the shortest paths without constraints and generating the additional forwarding rules needed for the constraints. Run times are stored in fields of type `i64` to avoid casting before doing calculations on them, as the numbers can become negative in the process. We implemented a few helper functions for the struct, most notably `apply()` and `merge()`. The former applies a certain function given as parameter to all four stored run times. The latter merges a `BenchmarkResult` with another one by using the provided merging function, which takes two i64 as input and outputs a new i64. Both of these helper functions are used within the struct `BenchmarkStatistic` described in the next Section. Also we provide an implementation of the `std::fmt::Display` trait, so benchmark results can be printed with `println!("{}", benchmark_result)`.

Benchmark Statistic The `BenchmarkStatistic` struct is a collection of arbitrary many `BenchmarkResult` objects. We intend it to represent a benchmark series run with the same parameters, so the minimum, maximum and average can be calculated easily. This can then be used to evaluate those over a e.g. 5 runs. To calculate those efficiently, it utilizes the `merge()` and `apply()` functions offered by `BenchmarkResult`. We also provide the implementations `print_header()` and `print()`. They allow to output the benchmark results in a formatted way such that they later can easily be inspected or exported to programs like excel to produce plots. We provide an example for such an output in Listing 5.11 and 5.12.

Listing 5.11: Example benchmark output part 1.

```
topology  cores hosts    switches #p/s rules    r_len     parse_avg  in p_min  p_max  conv_avg   in cv_min
=============================================================================================================
jellyfish     1    21334     1000   32   100000     4           774 ms    -93    +39        478 ms     -4
jellyfish     2    21334     1000   32   100000     4           795 ms    -66    +26        486 ms     -4
jellyfish     4    21334     1000   32   100000     4           747 ms    -11    +10        461 ms     -4
jellyfish     8    21334     1000   32   100000     4           758 ms    -39    +27        455 ms     -5
jellyfish    12    21334     1000   32   100000     4           763 ms    -53    +42        464 ms     -6
jellyfish    16    21334     1000   32   100000     4           744 ms    -56    +30        566 ms   -112
jellyfish    20    21334     1000   32   100000     4           729 ms    -45    +42        578 ms   -118
jellyfish    24    21334     1000   32   100000     4           739 ms    -67    +35        689 ms   -221
jellyfish    28    21334     1000   32   100000     4           733 ms    -60    +65        491 ms     -5
jellyfish    32    21334     1000   32   100000     4           720 ms    -47    +34        714 ms   -217
jellyfish    36    21334     1000   32   100000     4           695 ms    -25    +33        503 ms     -8
jellyfish    40    21334     1000   32   100000     4           691 ms    -24    +15        628 ms   -131
jellyfish    44    21334     1000   32   100000     4           689 ms    -15    +23        561 ms    -35
jellyfish    48    21334     1000   32   100000     4           695 ms    -26    +24        738 ms   -136
jellyfish    52    21334     1000   32   100000     4           685 ms    -19    +12        911 ms   -122
jellyfish    56    21334     1000   32   100000     4           686 ms    -21    +28        826 ms    -96
jellyfish    60    21334     1000   32   100000     4           700 ms    -27    +27        909 ms   -103
jellyfish    64    21334     1000   32   100000     4           699 ms    -35    +62       1048 ms     -3
```

Listing 5.12: Example benchmark output part 2.

```
cv_max djkstr_avg in d_min  d_max  constr_avg in cs_min cs_max total_avg  in t_min  t_max
=============================================================================================
    +5      39553 ms    -79   +156       3292 ms     -9    +17      43619 ms    -72   +186
    +6      22296 ms   -119    +52       1747 ms     -7     +8      24838 ms    -92    +86
    +5      12814 ms    -63    +69        916 ms    -11     +5      14477 ms    -52    +79
    +4       8674 ms    -57    +88        507 ms     -4     +7       9939 ms    -52   +100
    +4       7603 ms   -582   +542        438 ms    -16    +62       8804 ms   -556   +572
  +419       6313 ms   -255   +115        400 ms    -36    +23       7457 ms   -279   +168
  +411       5801 ms   -148   +147        394 ms     -9    +15       6924 ms   -200   +163
  +322       4891 ms    -87    +93        377 ms     -8    +10       6007 ms   -155   +126
    +9       4605 ms    -85   +213        363 ms     -4     +4       5701 ms   -124   +278
  +336       4333 ms    -54   +134        371 ms     -3     +8       5424 ms    -63    +85
    +8       4321 ms    -82    +72        408 ms    -34   +106       5424 ms   -107    +64
  +415       4067 ms    -56    +53        413 ms    -37   +122       5171 ms    -85    +85
   +22       3637 ms    -29    +36        416 ms    -43   +140       4742 ms    -80   +161
  +213       3313 ms    -50   +134        416 ms    -38   +141       4424 ms    -95   +249
  +132       3245 ms    -66   +116        431 ms    -52   +179       4361 ms    -98   +276
  +210       3178 ms    -75   +121        432 ms    -45   +170       4296 ms   -117   +270
  +133       3116 ms    -91    +75        441 ms    -61   +215       4257 ms   -131   +258
    +4       3252 ms   -404  +1130        435 ms    -50   +196       4386 ms   -456  +1144
```

6 Evaluation

This chapter presents the different experiments we conducted in the scope of this thesis. First we elaborate on the experimental setting in Section 6.1. We introduce the hardware used in the measurements and list the input parameters used, namely the generated policies and topologies. Then we show the actual experiments in the following sections. In all of those we follow the same general structure. First we explain what we measured and why. Then we outline what results we expected for the used input. Finally we show the results in the form of plots, tables or both and discuss the experiments' rationales. All measurements were conducted with 5 repetitions using the same parameters. All results we present are averages over those 5 runs. The complete output of all measurements we conducted are shown in Chapter A of the Appendix.

6.1 Experimental Setting

In this first section of the chapter we introduce the experimental setting. Namely the Hardware we executed the benchmarks on and the data sets we executed. Also we introduce the topologies we used and explain our choice of topologies.

6.1.1 Hardware

We conducted all of the benchmarks presented in this chapter on a Dell PowerEdge R820 rack server[1]. Our specific machine was equipped with four Intel® Xeon® E5-4640 processors. Each of those four processors has eight physical cores running at 2.4 GHz. They all support HyperThreading® , so in total there are 32 physical or 64 virtual cores available respectively. The installed memory was 512 GB of RAM in total.

6.1.2 Policies

To measure the influence of different policies on our computation, we generated various policies. As described in Section 5.6 we generate policies with and-constraints only, because or-constraints are translated into several and-constraints as described in Section 5.4.4. We chose to evaluate policies with short constraint lengths of 2, 4, or 10 where 4 is the default length. The number of policies we varied between 10k, 50k and 100k where 100k is the default to stress test the system. A policy length of n means that every packet from a source to a destination host has to pass exactly n intermediate nodes. Policies are expected to be short in sizes due to the general rule to keep paths short and the particular contained structures of data center network topologies Also as network services become controller applications, they can be aggregated on location, effectively keeping the policy length short. At the same time the number of policies can get quite big, as policies are independent from each other. Also they concern only a single data flow in the network where there can be many different in total.

6.1.3 Topologies

There are several possible topologies for data center networks that have appeared through the years. When we picked the ones to evaluate our work on we tried to cover the different available topology types and concepts as good as possible. One topology we use is the Jellyfish topology, which is randomly generated and especially suited for random-permutation traffic. It is very good to stress test performance with high utilization. Another one is the Fat-Tree topology, which is hierarchical and could be considered a more traditional topology, as it divides networks into subnets. This makes it easy to comprehend and manage

[1] `http://www.dell.com/downloads/global/products/pedge/r820_spec_sheet.pdf`

by human administrators. In Table 6.1 we list the different parameters we used to create the topologies for the measurements. [SHPG12, AFLV08]

	Hosts	Switches	#Ports
Jellyfish	up to 200k	up to 10k	4 - 96 / Switch
Fat-Tree	up to 93k	up to 6480	4 - 72 / Switch

Table 6.1: Parameter range of the conducted measurements

6.1.3.1 Fat-Tree Topology

The Fat-Tree topology is an improved hierarchical network topology. In comparison to purely hierarchical tree topologies, this topology can be built with commodity-switches and still supports a big number of hosts. When constructing a Fat-Tree topology the parameter k is what defines the topology layout, where k is the number of ports per switch. In a k-ary Fat-Tree all hosts are assigned to one of the k pods, and switches are either an *Edge* or *Aggregation* switch within one of the pods, or are assigned to the *Core*. Edge switches use half of their ports to connect to hosts and the other half to connect to Aggregation switches. Aggregation switches themselves use the remaining $k/2$ ports to connect to Core switches. A Fat-Tree topology can support up to $k^3/4$ hosts by using $k^2 + k^2/4$ switches in total, each with k ports. [AFLV08]

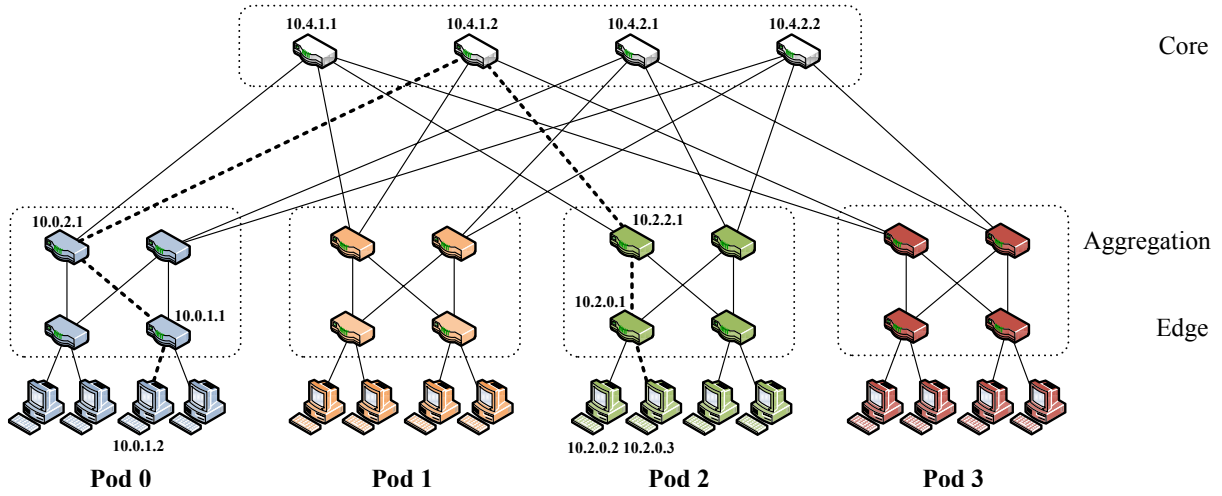


Figure 6.1: Example 4-ary Fat-Tree Topology: 16 hosts and 20 4-port switches. Shows routing from one pod to another. Source: [AFLV08, Figure 3]

6.1.3.2 Jellyfish Topology

A Jellyfish topology is created by connecting exactly k or k+1 hosts to each switch, where k is chosen such that all hosts are assigned to a switch. The remaining ports of the switches are then used to connected the switches randomly but never the same two switches twice. Jellyfish networks generally have a shorter average path length compared to hierarchical networks. As with the Fat-Tree topology, they can be set up with only commodity switches which are much cheaper than the high-end switches needed at aggregation layers of conventional topologies. Another advantage is the fact that they are easily extendable: To insert new hardware just remove random edges and connect the newly free ports with the new hardware. With regards to supported hosts per number of switches and ports per switch the topology is very flexible. In theory nearly all ports of the switches can be used to connect with hosts, but the less of the total available ports are used for hosts, the more connections will be available for traffic routing. [SHPG12]

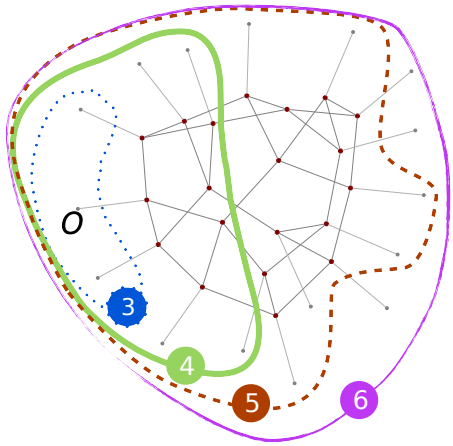


Figure 6.2: Example Jellyfish topology with 16 hosts and 20 4-port switches. Shows the path length from one host to the others. Source: [SHPG12, Figure 1b]

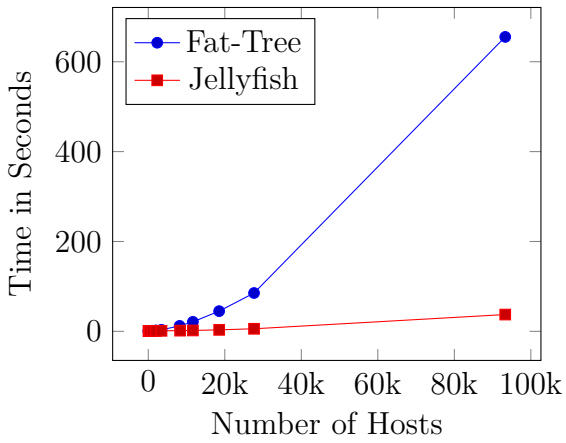
6.2 Network Size Comparison

This first measurement compares Fat-Tree with Jellyfish topologies. The parameter we varied is only the number of supported hosts. We fixed the number of threads to 32, the constraint length to 4 and the number of policies to 100k. The amount of ports per switch is al-

ways the same for all switches within one measurement. In Jellyfish topologies the amount and type of used switches can be chosen freely, so we use $\lfloor 2/3 * ports\ per\ switch \rfloor$ of each switch's available ports to connect to hosts. The remaining $\lceil 1/3 * ports\ per\ switch \rceil$ ports are then used to interconnect the switches.
Our expectation was that the Fat-Tree topology would use more computation time and resources, because it needs more switches to support the same number of hosts compared to a Jellyfish topology. In our algorithms the number of switches influences the computation time most, because it directly increases the complexity and memory consumption. Each switch has to know the next hop for every destination so $n * n$ tuples are needed for topologies with n switches. The results proved us right, Jellyfish seems to be much more efficient in terms of supported host ratio. This also holds true for computation time.

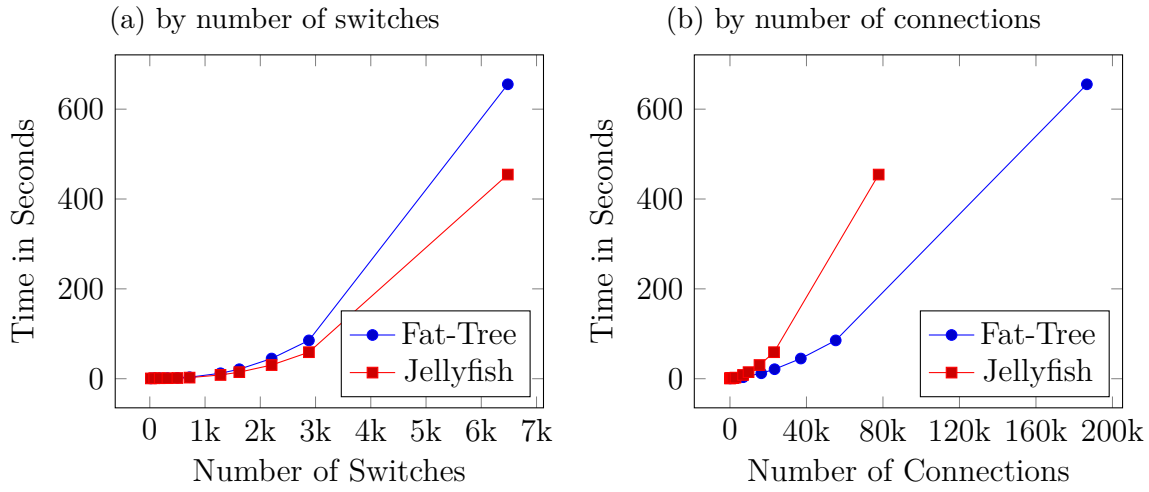
In the first plot shown in Figure 6.3 we compare the two topologies in regards to supported hosts. We picked the parameters used for the Jellyfish topology to match the supported hosts of the Fat-Tree topology. For example one specific Fat-Tree topology uses 2880 switches with 48 ports and supports 27648 hosts, so we ran a measurement with a Jellyfish topology using 864 switches with 48 ports. Each of those switches is connected to 32 hosts, which leads to a topology with 27648 hosts in total. It is clearly visible in the plot that for a Fat-Tree topology our computations need much more time compared to a Jellyfish topology using the same type of switches and supporting the same number of hosts. This stems from the fact that Jellyfish topologies can support more hosts per switch and the complexity of our computation is proportional to the number of switches in the network as mentioned before.

Figure 6.3: Compare runtime by number of supported hosts.



In Figure 6.4 we show two more comparison of the topologies. The Fat-Tree topology is the same as before, but the Jellyfish topology is a different one. There we did not match the number of hosts as before, but kept the number of switches in both topologies the same, namely 2880 switches with 48 ports each. Then the Fat-Tree topology still supports 27648 hosts and the Jellyfish topology supports 92160 hosts, which is more than triple the amount compared to the Fat-Tree topology.
The plot in Figure 6.4a shows that when comparing the topologies in regards of used switch numbers, they are much closer together than when comparing them by number of supported hosts. Although the Jellyfish topology still needs less computation time, which is most certainly due to the longer average path-length in Fat-Tree topologies. Important to notice is the fact, that the Jellyfish topology supports many more hosts when comparing to a Fat-Tree topology with the same number of switches and ports per switch.
In the plot shown in Figure 6.4b we compare the number of connections in the topology. The plot shows that the number of connections does drive the total computation time. In comparison, the Jellyfish topology is slower than a Fat-Tree one with the same number of connections. This is due to the fact that in equally large networks Jellyfish topologies contain far less connections compared to Fat-Tree topologies. For this reason the same number of connections implies a higher number of switches and hosts in the Jellyfish network. This increases runtime because the amount of switches is the main influencing factor for the total runtime, as explained before. As before such comparisons should be carefully interpreted due to the different host count in both topologies.

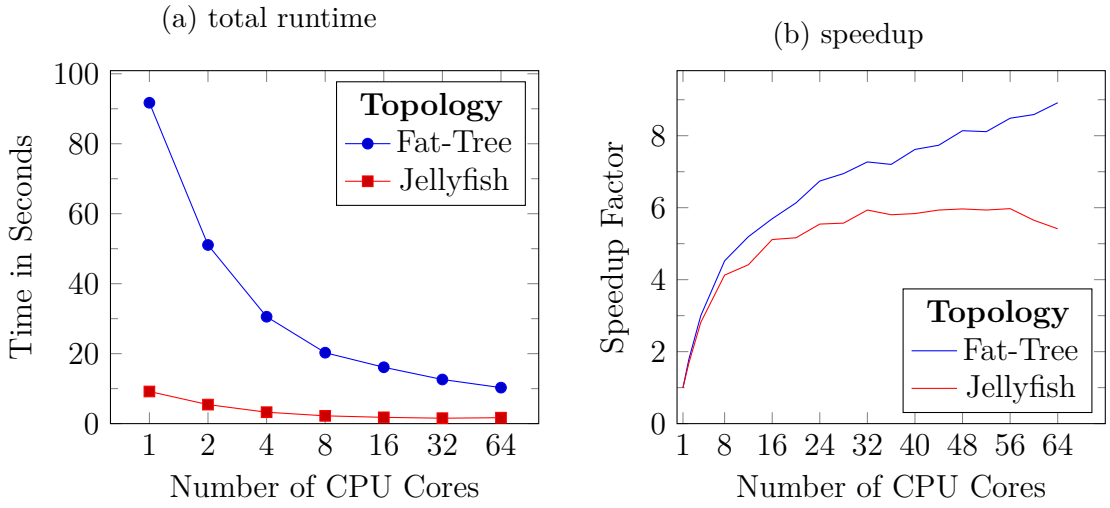
Figure 6.4: Compare runtime for same number of switches in both topologies.



6.3 Scaling with Number of Workers

The next measurements we present focus on how our implementation scales. We conducted benchmarks by running our dataflow computations with 1 to 64 threads. The number of threads was not increased beyond 64, because the machine we used offers 32 physical cores supporting HyperThreading™, which leads to 64 virtual cores. This allows us to determine our project's potential for parallelization and evaluate runtime improvements when adding more processing power. The number of hosts in the topologies was fixed to 8192, the amount of switches was 1280 in the Fat-Tree and 390 in the Jellyfish topology. All switches used in this experiment offer 32 ports. We expected the runtime to decrease when increasing the amount of used threads until reaching a saturation when the overhead for newly added workers exceeds their benefit. This overhead stems from the communication between the different worker threads required to exchange data and synchronize.

Figure 6.5: Different numbers of used threads; topologies with 8192 hosts.



We show plots of our experiment's results in Figure 6.5 and 6.6 As in the previous measurements, the runtime for Fat-Tree topologies is significantly higher for the same number of hosts compared to Jellyfish topologies. For the Fat-Tree topology saturation did not occur but the increase in speedup for each added thread decreased as the total number of threads increased. What went against our expectations was the fact that speedup still occurred when exceeding 32 threads, which is the number of physical processors. This seems

to prove that HyperThreading™ can give a slight performance boost. With the Jellyfish topology on the other hand the speedup stalled at 32 threads and then slowly declined. This is because the number of switches and therefore also the total computational work is less there, so saturation occurs earlier.

Figure 6.6: Different numbers of used threads; topologies with 27648 hosts.

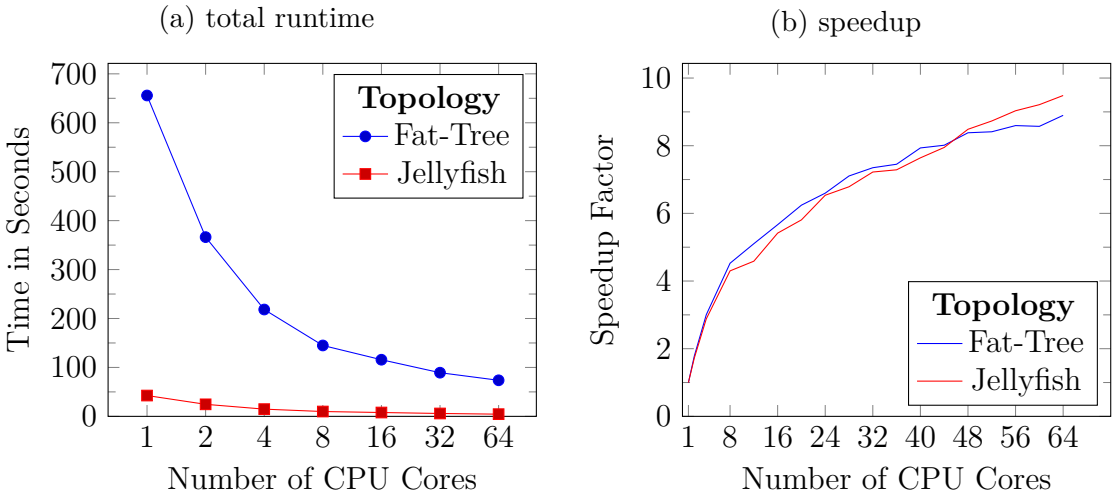


Table 6.2 shows our measurements split up by computational steps. In the first step tagged 'fwdrules' all forward rules for the shortest paths within the network are calculated. The next three steps are all related to policies. We plotted them without the runtime of the first step in Figure 6.7. The step 'policy parse' measures the time needed to parse the policy string, which creates a list of `Policy` objects in the memory. Afterwards in step 'policy convert' the time to convert those objects to tuples of type `ConstraintTuple` is measured. Finally the last step 'policy fwdrules' calculates additional forward rules for the policies. They override the rules generated in the first step and are needed so the policies are respected.

The table shows that the initial computation of the forward rules for the shortest paths is by far the most time-consuming part of the computation. This is also the reason why it scales very well, most of the runtime improvements when adding more threads happen here. As the table shows the runtime decreases monotonically when adding more threads, with one exception. This exception is the computation with a Jellyfish topology and 64 threads, which takes more time than the one with 32. An explanation for this is the small number of switches in comparison with the Fat-Tree topology. This leads to saturation

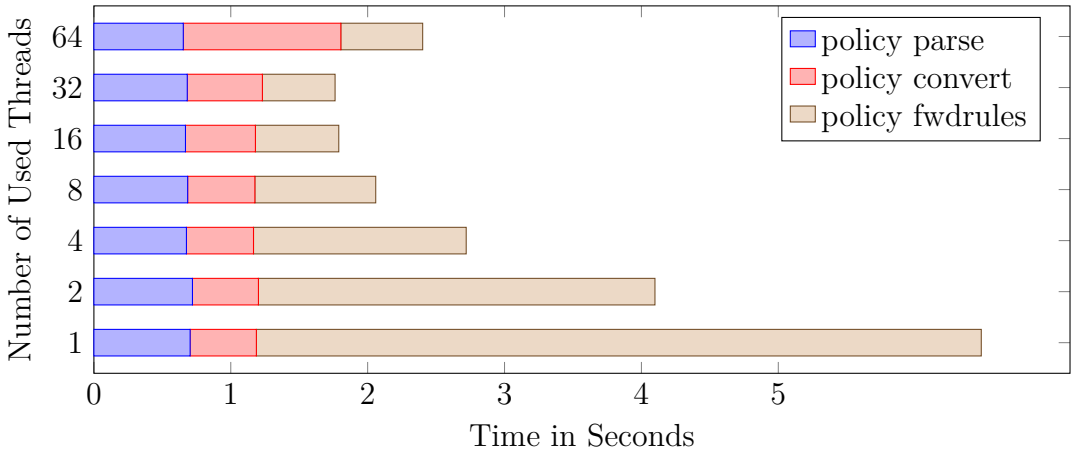
occuring, the total computational effort to distribute the data and perform the work is higher when using 64 workers compared to when using only 32.

Table 6.2: Runtime for different numbers of used threads split up by computation steps.

		Time in Seconds			
			policy		
Topology	**Threads**	fwdrules	parse	convert	fwdrules
Fat-Tree 8192 Hosts, 1280 Switches	**1**	85.72	0.703	0.484	5.296
	2	47.452	0.72	0.482	2.897
	4	28.329	0.676	0.49	1.554
	8	18.708	0.685	0.491	0.883
	16	14.833	0.669	0.511	0.609
	32	11.401	0.682	0.549	0.531
	64	9.036	0.653	1.151	0.598
Jellyfish 8190 Hosts, 390 Switches	**1**	5.207	0.702	0.507	3.278
	2	2.924	0.725	0.508	1.789
	4	1.599	0.671	0.489	0.993
	8	0.991	0.675	0.491	0.56
	16	0.7	0.657	0.727	0.439
	32	0.501	0.643	0.754	0.404
	64	0.627	0.648	1.11	0.422

The two next computational steps do not benefit by increasing the number of cores, as they are always done on one core only. This is because these tasks are not done in a dataflow computation but in plain Rust code. In the future this could also be done in parallel, as the work is consisting of several independent policies that could be parsed and converted in parallel. Finding the reason why the runtimes of those serial tasks go slightly down upon increasing the number of workers would need more detailed investigations.
Finally the last column shows the time it took to calculate the additional forward rules created to respect the policies. The computations of this step are done with dataflow, so it also scales with more used workers. In contrast to the first step, saturation occurs when going above 32 used threads. This is the same for both topologies and also happens with bigger topology sizes. Those results can be found in de appendix.

Figure 6.7: Runtime for different numbers of used threads split up by computation steps for Fat-Tree topologies with 8192 hosts. Does not contain shortest path computation.



6.4 Influence of Policies

In this section we evaluate the influence of policies on the computation's total runtime. As we explained in Section 6.3, each run of our application first parses the policies, then converts them to tuples the dataflow can process and finally generates additional forward rules from those tuples. We present our measurements with different numbers of used policies in Subsection 6.4.1 and with different constraint-lengths of the policies in Subsection 6.4.2.

6.4.1 Number of Policies

In this experiment we varied the number of policies to measure their influence on the total computation time. We evaluated the runtime with 10k, 50k and 100k policies. We reasoned that a higher amount of policies than about 5x the number of hosts is not to be expected. This would increase the complexity beyond the capabilities of human operators. To test the influence on parallel computation we tested with 1 and 32 threads. Different network sizes were also inspected, we picked 8192 or 27648 hosts and used 32-port switches in the former and 48-port switches in the latter.

Table 6.3: Total runtime for different numbers of policies.

Topology Size	# Policies	Time in Seconds			
		Fat-Tree		Jellyfish	
		1 thread	32 threads	1 thread	32 threads
8192 Hosts	**10k**	85.446	11.861	5.845	0.672
	50k	88.146	12.313	7.453	1.095
	100k	89.555	12.726	9.433	1.593
27648 Hosts	**10k**	645.371	88.910	38.296	4.823
	50k	646.478	89.478	40.562	5.269
	100k	659.616	90.489	43.071	5.882

We present the resulting total runtimes of our experiments in Table 6.3. It shows that varying the number of policies' does not have a big impact on the computation. The highest difference in runtime can be observed for the small Jellyfish topology, where the policy itself is far bigger than the topology. For the bigger network increasing the number of policies from 10k to 100k increases the runtime at most by around 17%, which is the case for the smaller Jellyfish topology. In the case of the bigger Fat-Tree topology with 27648 hosts, the runtime even decreased. As can be seen in Table 6.4 this is because the calculation of the forward rules for the shortest paths was faster. We discuss this in more detail below.

Table 6.4: Runtime for different numbers of policies split up by computation steps.

			Time in Seconds			
				policy		
Topology	**Threads**	**Policies**	fwdrules	parse	convert	fwdrules
Fat-Tree 27648 Hosts 2880 Switches	**1**	**10k**	644.585	0.108	0.064	0.678
		50k	642.833	0.424	0.292	3.221
		100k	652.396	0.816	0.57	6.404
	32	**10k**	88.747	0.087	0.066	0.076
		50k	88.828	0.406	0.257	0.244
		100k	89.24	0.771	0.589	0.478
Jellyfish 27648 Hosts 864 Switches	**1**	**10k**	37.702	0.11	0.052	0.484
		50k	37.875	0.402	0.254	2.285
		100k	38.005	0.754	0.505	4.312
	32	**10k**	4.644	0.079	0.064	0.100
		50k	4.647	0.346	0.296	0.276
		100k	4.703	0.703	0.653	0.476

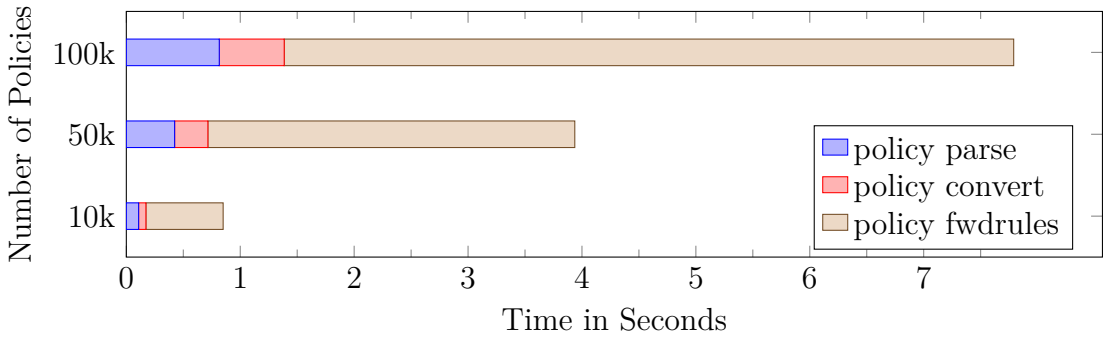
In Table 6.4 we show the runtime split up by computational steps. The first step 'fwdrules' depicts the time needed to calculate all forward rules for the shortest paths within the network. All three remaining steps are related to the policies. They measure the time it takes to parse the topology, convert it to `ConstraintTuples` the dataflow computation can process and to create additional forward rules needed so the network respects the policies. The runtimes of steps related to policies all increase nearly proportionally with the number of used policies. Steps 'parse' and 'convert' need approximately the same time with 1 and 32 threads used for the computation. The final policy step calculating the forward rules decreases its runtime with 32 threads to 36% of its runtime with 1 thread in the worst and 7.5% in the best case. All those results were anticipated. What at first sight may look odd is the runtime fluctuation in some cases for the calculation of initial forward rules in the first step. This should not happen in theory, as the topology is the only parameter relevant for this step which was not changed between the different measurements. We explain this behavior with the measurements variance of several seconds between the different runs. Refer to Chapter A in the appendix for the measurement's maximum deviance of the

average results.

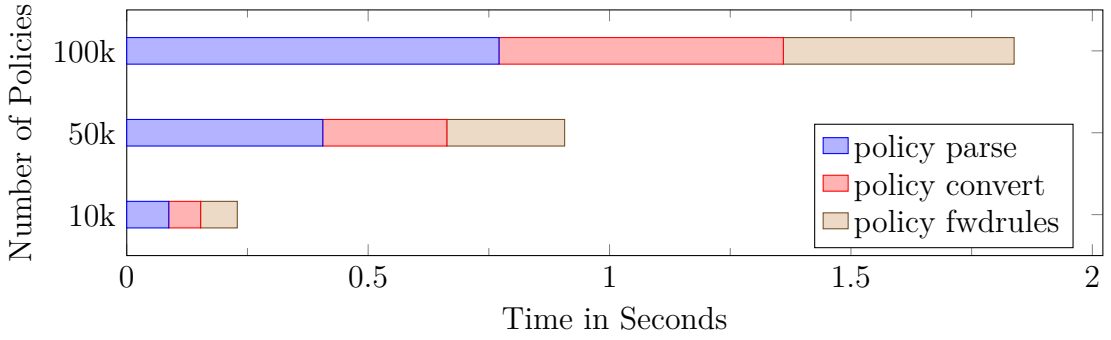
To compare the runtimes of the policy-steps conducted with 1 or 32 threads we created the plots in Figure 6.8. Both show measurements for Fat-Tree topologies with 27648 hosts. They illustrate how the runtimes of the three individual steps increase when adding more policies. When comparing the plots, the speedup of the 'policy fwdrules' step with 32 threads is easily visible. In contrast the other two steps 'parse' and 'convert' do not change their runtime at all, which was expected.

Figure 6.8: Runtime for different numbers of policies split up by computation steps for Fat-Tree topologies with 27648 hosts. Does not contain time for forward rule computation of shortest paths.

(a) with 1 thread



(b) with 32 threads



6.4.2 Policy Length Comparison

To check how the runtimes behave for different lengths of constraints in policies, we made measurements with policy constraints of length 2, 4, and 10. We figured those would be typical lengths, as shorter ones do not make much sense and longer ones are not easily

readable by humans anymore. For our measurements we evaluated Jellyfish topologies with 864 switches and Fat-Tree topologies with 2880 switches having 48 ports each, where both support 27648 hosts. We ran the tests with 1 and 32 threads and 100k policies.

Table 6.5: Runtime for policies with different lengths split up by computation steps.

			Time in Seconds			
				policy		
Topology	**Threads**	**Policy Length**	fwdrules	parse	convert	fwdrules
Fat-Tree 27648 Hosts 2880 Switches	**1**	**2**	647.169	0.477	0.262	4.025
		4	647.825	0.786	0.540	6.425
		10	645.833	1.703	1.361	13.271
	32	**2**	89.060	0.449	0.249	0.297
		4	88.548	0.745	0.603	0.470
		10	89.222	1.569	1.531	1.045
Jellyfish 27648 Hosts 864 Switches	**1**	**2**	37.801	0.472	0.250	2.829
		4	37.827	0.766	0.532	4.297
		10	37.442	1.618	1.430	8.676
	32	**2**	4.700	0.438	0.282	0.321
		4	4.685	0.745	0.675	0.471
		10	4.720	1.503	1.794	0.919

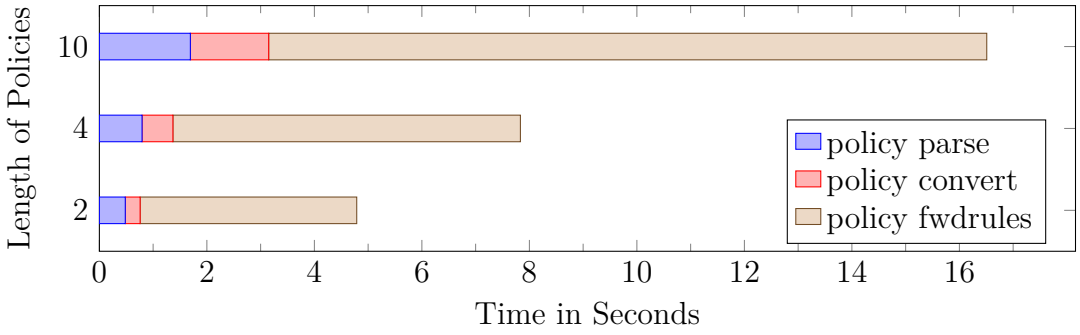
The results are shown in Table 6.5 split up by computation steps as before. First step is again 'fwdrules' which generates all forward rules for the shortest paths in the network. Following are the policy-related steps 'parse' which parses the policy string, 'convert' which converts the policies into `ConstraintTuples` and finally 'fwdrules' calculating the additional forward rules so the policies are respected. We expected the results to be similar as in the prior Section 6.4.1. The measurements proved our expectations right, runtime goes up linearly when increasing the policy-length. As when varying the policy-lengths, runtimes of steps related to policies all increase proportionally with the length of the used policies. Steps 'parse' and 'convert' need about the same time with 1 and 32 threads, as the code is not executed in parallel. On the other hand the runtime for the final step of calculating additional forward rules for the policies decreases significantly when adding more threads to the computation. Compared to the measurements for the number of rules, the runtime for

the calculation of the initial forward rules has a much lower fluctuation. All measurements including their maximum deviation over the 5 conducted runs are again shown in Chapter A of the appendix.

We produced two plots comparing the runtimes of the policy-steps conducted with 1 or 32 threads shown in Figure 6.9. Both show measurements for Fat-Tree topologies with 27648 hosts. As in the previous section, the time used for the parse and convert steps does not decrease when adding more cores. In contrast to this, the forward-rules step shows an impressive speedup. It is about 12 times faster on average than when computing with 1 thread only.

Figure 6.9: Runtime for different lengths of policies split up by computation steps for Fat-Tree topologies with 27648 hosts. Does not contain time for forward rule computation of shortest paths.

(a) with 1 thread



(b) with 32 threads

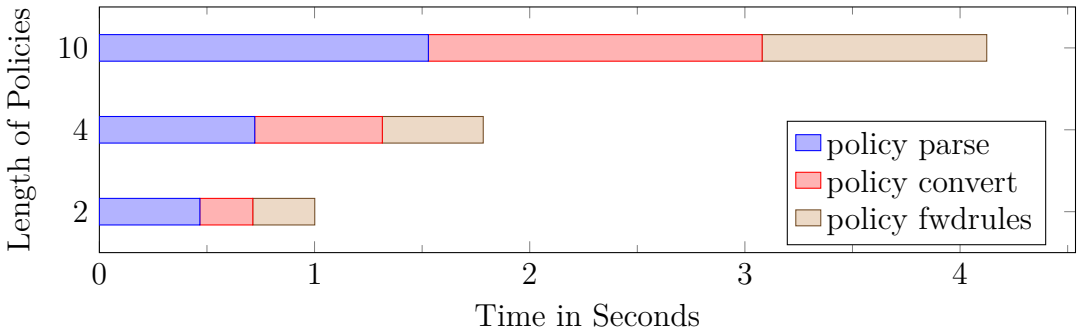
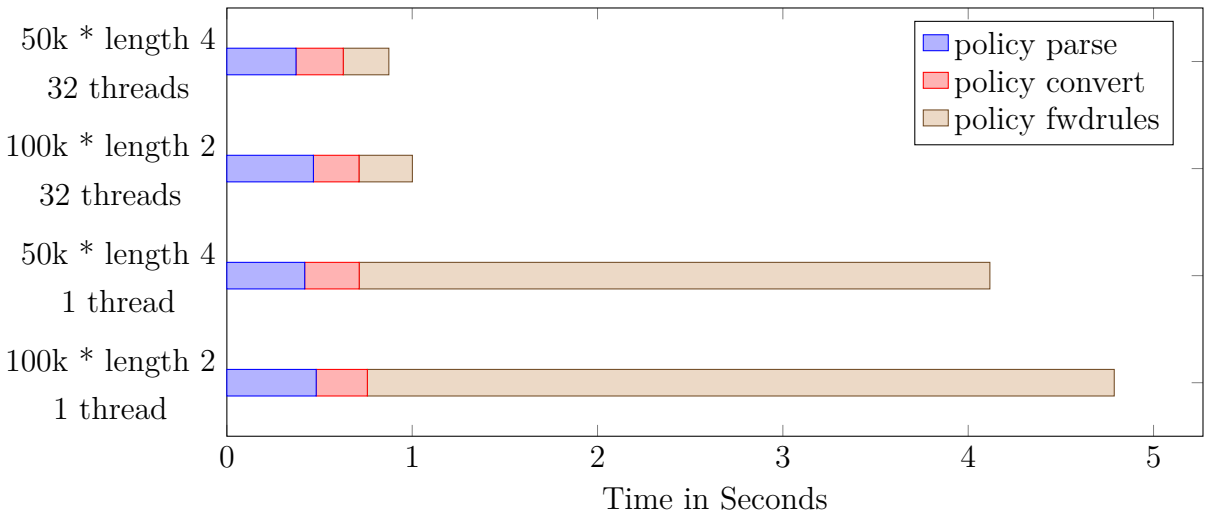


Figure 6.10: Runtime for different lengths of policies split up by computation steps for Fat-Tree topologies with 27648 hosts. Does not contain time for forward rule computation of shortest paths.



Finally we also created a plot comparing the runtimes of 50k policies of length 4 with 100k policies of length 2 for all policy-related steps with 1 or 32 threads. We show this plot in Figure 6.10. It shows that the runtime for those two policy sets is about the same for the individual steps. Intuitively this makes sense, as the performed work is the same for both. The number of parsed tokens is similar, as is the amount of additionally generated forward rules. What can also be noted is the fact that they both approximately scale the same when adding more threads to the computation.

6.5 Topology Updates

One of the biggest advantages of the differential-dataflow framework we use is its ability to process changes of the input data very fast. In this section we showcase this ability by updating the topology and then measuring how much time it takes until all changes are processed and the forward rules are updated. We conducted three different types of changes, removals of connections, updates of connection weights and removal of switches. In the first subsection we show our benchmarks of connection failures. Then we decrease or increase weights of some connections by a certain threshold. Finally we measure the time switch failures take to recover from in the last subsection. For each of them we measured the time until our algorithm has updated all the forward rules after the changes were applied.
In all following measurements we evaluated Fat-Tree and Jellyfish topologies. We use Fat-Tree topologies with 8192 Hosts and 16384 connections or 27648 Hosts and 55296 connections. The used Jellyfish topologies also support 8192 and 27648 hosts but only need 2144 respectively 6912 connections. All smaller topologies use 32-port switches and the bigger ones 48-port switches. Jellyfish topologies need much less connections compared to the Fat-Tree topology because of the conceptual differences between the topologies. The Fat-Tree topology has a hierarchical structure whereas the Jellyfish topology is built randomly. We conducted all benchmarks with 32 threads and 100k policies if not explicitly stated otherwise.

6.5.1 Connection Failures

In this section we present the measurements for connection failures. We removed between 2 and 10 ‰ of the topology's connections and waited until the affected forward rules were recomputed. Important to note is the fact that the bigger Jellyfish topology has less than half the number of connections compared to the smaller Fat-Tree topology although it supports more hosts.

Table 6.6: Runtime to process removals of connections in batches of size between 2‰ to 10‰ of all connections in the topologies.

			Time in Seconds				
			‰ Connections removed per batch				
Topology	**Hosts**	**Connections**	2	4	6	8	10
Fat-Tree	**8192**	**16384**	0.500	0.751	1.146	1.303	1.677
	27648	**55296**	4.000	6.326	7.941	10.053	12.415
Jellyfish	**8192**	**2144**	0.082	0.093	0.108	0.169	0.177
	27648	**6912**	0.301	0.495	0.753	0.906	1.080

We present the results of our experiments in Table 6.6. It shows a near linear increase in runtime when the number of removed connections is increased. This is expected, because each removed connection triggers the generation of new and the removal of now invalid forward rules. The more connections are removed, the more changes are triggered. Jellyfish topologies seem to react much faster to changes but this is related to the fact, that the number of affected connections is much lower, because the topology contains a much lower number of connections in total.

Table 6.7: Runtime to process the removal of a single connection by batch-size.

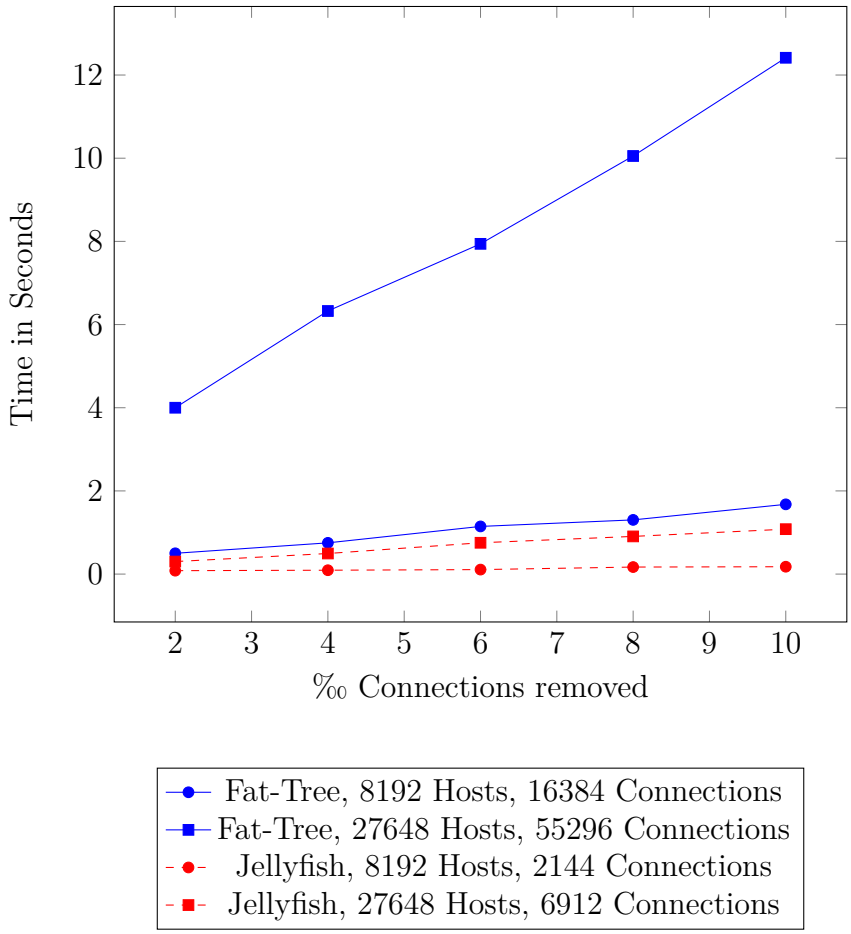
			Time in Milliseconds				
			‰ Connections removed per batch				
Topology	**Hosts**	**Connections**	2	4	6	8	10
Fat-Tree	**8192**	**16384**	15.625	11.554	11.694	9.947	10.288
	27648	**55296**	36.364	28.624	23.991	22.744	22.491
Jellyfish	**8192**	**2144**	20.500	11.625	9.000	9.941	8.429
	27648	**6912**	23.154	18.333	18.366	16.473	15.652

In Table 6.7 we list the times needed per connection removal, grouped by amount of connections removed at once. This table shows that the processing of changes is faster when input in one bigger batch compared to several smaller ones. It also shows a difference between topology sizes, bigger topologies need more time to process a change. The reason

for this is that each removal potentially affects more switches and therefore triggers more updates of forward rules.

We also prepared some plots from the data we collected. The first plot in Figure 6.11 shows the total time until the removal of x ‰ connections was processed. It visualizes the linear increase of time it takes when increasing the number of connections that are removed within a batch. It also clearly shows the advantage of a Jellyfish topology. If a certain percentage of all connections fail, our controller can recover the routing much faster when the network uses a Jellyfish topology layout.

Figure 6.11: Runtime to process connection outages in Fat-Tree and Jellyfish topologies with 8192 or 27648 hosts.



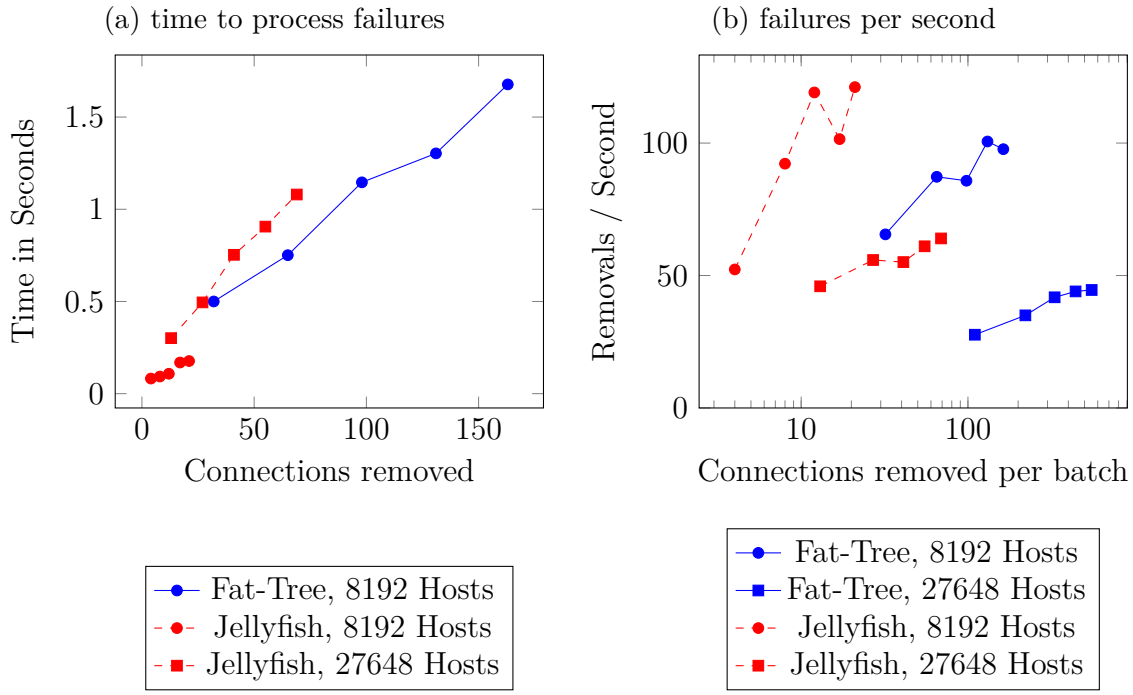
The next two plots in Figure 6.12 examine the runtime for connection removals in terms of absolute numbers.

Figure 6.12a plots the number of connections removed against the time it takes. This

shows that the time to recover after such an update batch is nearly proportional to the number of connections that are removed. Our explanation for this is that a higher number of removed connections leads to more changes in the network so more forward rules have to be replaced. Note that we excluded the bigger Fat-Tree topology from this plot because it removes many more connections and takes much more time than the other topologies. Including it would skew the axes and the other plot lines would not have been clearly visible anymore.

The second plot in Figure 6.12b shows the number of connection removals that can be processed per second. It visualizes the capabilities of our controller to adapt the forwarding rules in real-time for up to more than 100 connection changes per second. The first thing to notice is the capacity difference between the network sizes. Smaller networks allow to process a much higher number of changes per time interval. The reason for this is that less forward rules have to be changed for each removal. If for example after a connection removal a certain switch is only reachable through a different than the prior shortest path, potentially many more switches have to update their forward tables accordingly in a bigger compared to a smaller network.

Figure 6.12: Compare runtime for absolute numbers of failures.



6.5.2 Connection Weight Updates

This section shows the measurements for weight updates of the connections in the topology. Weights can correspond to link utilization. Thus showing how we can handle frequent changes to the topology attributes to reflect real conditions. We randomly increased or decreased the edge weights by either 20% or 60% of their original value. Then we measured the time until the controller processed the changes and updated all forward rules of the network. Important to note is again the difference in absolute numbers of connections contained in the topologies we show. A jellyfish topology is able to support more hosts while still having less connections than a Fat-Tree topology.

Table 6.8: Runtime to process weight update of connections.

	Conn.	**Time in Seconds**				
	Weight	‰ Connections updated				
Topology	**Change**	2	4	6	8	10
Fat-Tree	**20%**	0.498	0.694	0.948	1.247	1.308
8192 hosts, 16384 conn.	**60%**	0.523	0.745	1.094	1.443	1.617
Fat-Tree	**20%**	2.435	3.960	4.971	5.815	8.090
27648 hosts, 55296 conn.	**60%**	2.724	5.127	7.157	8.650	11.319
Jellyfish	**20%**	0.037	0.047	0.168	0.200	0.213
8192 hosts, 2144 conn.	**60%**	0.038	0.071	0.187	0.216	0.226
Jellyfish	**20%**	0.324	0.417	0.602	0.818	1.051
27648 hosts, 6912 conn.	**60%**	0.362	0.456	0.711	0.953	1.092

The results of our experiments are shown in Table 6.8. It shows that the time needed to process the updates grows proportionally with the amount of affected connections. This was expected, because more forward rules need to be updated when more edge weights change. Each weight change possibly introduces new shortest paths or invalidates old ones. Processing runtime also increases when changing the weights by 60% compared to only altering them by 20%. The reason for this is the fact that greater weight-changes increase the likelihood for a change of shortest paths in the network. We also created two plots shown in Figure 6.13 that depict the behavior we just described.

Figure 6.13: Runtime to process weight updates.

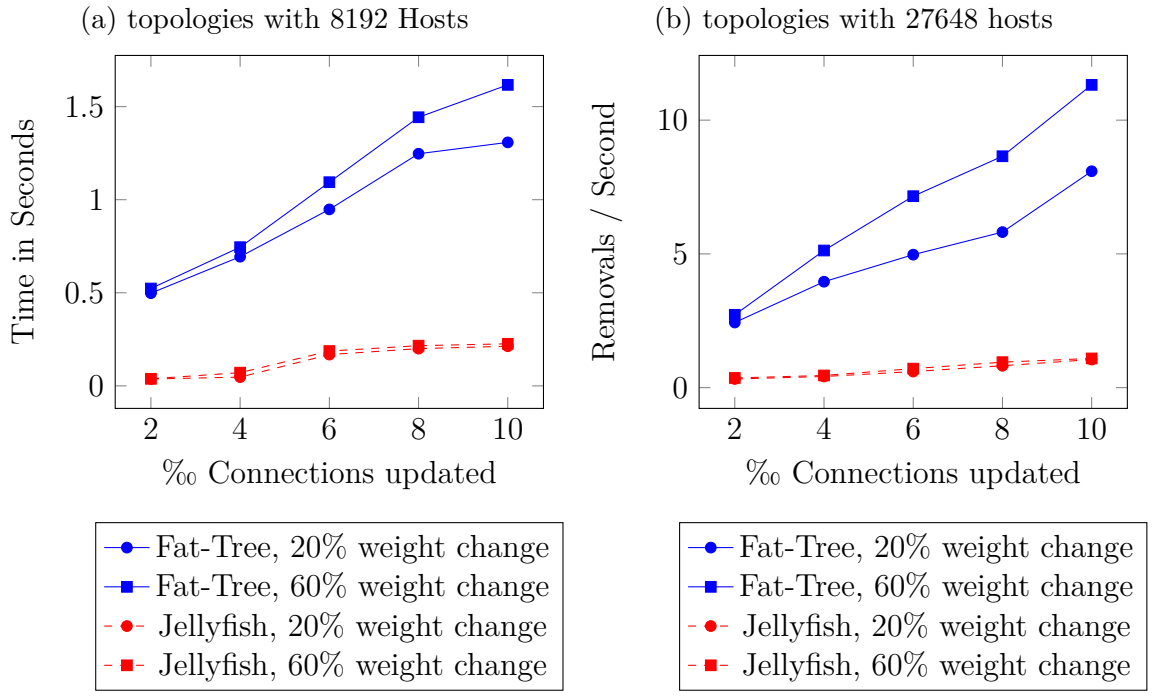
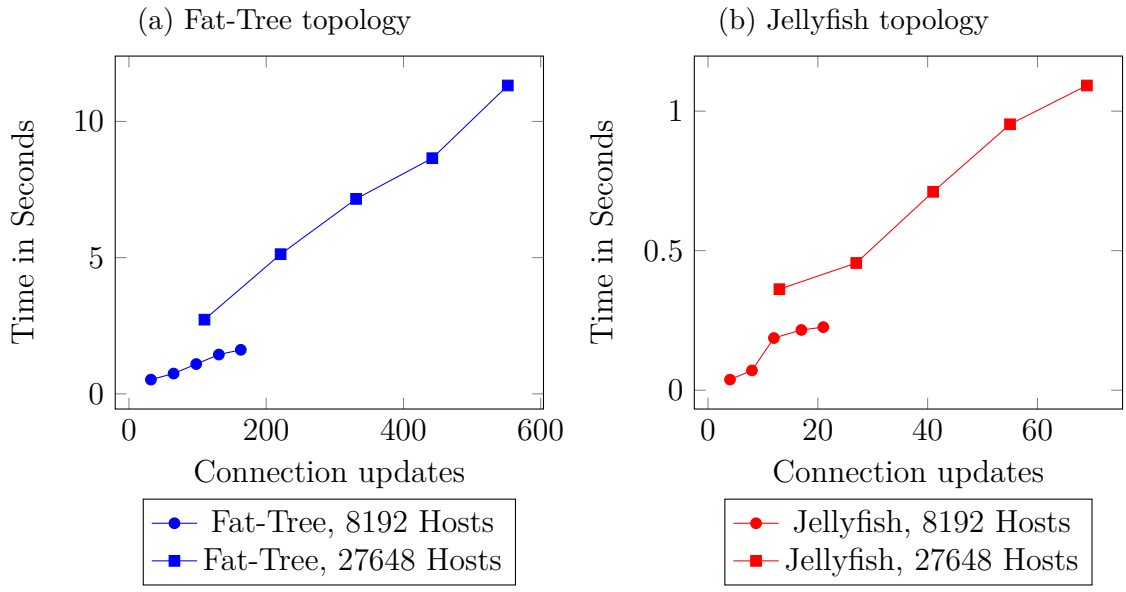


Figure 6.14: Compare runtime for absolute numbers of updates in topologies.



In two more plots contained in Figure 6.14 we show the runtime in regards to absolute numbers of changed connections. The plots show that the runtime evolves similar for

both topologies and is mainly dependent on the number of connections that are updated. Generally Fat-Tree is more affected by the percentage of weight change. We isolated the plots for each topology because the curves of the Jellyfish topologies would be very low on the left side of the plot if we put both topologies in the same. This is because there are less connections in a Jellyfish topology compared to Fat-Tree for the same number of hosts. The computation time generally is less compared to Fat-Tree because there are less links to recompute the shortest paths over leading to a smaller topology graph.

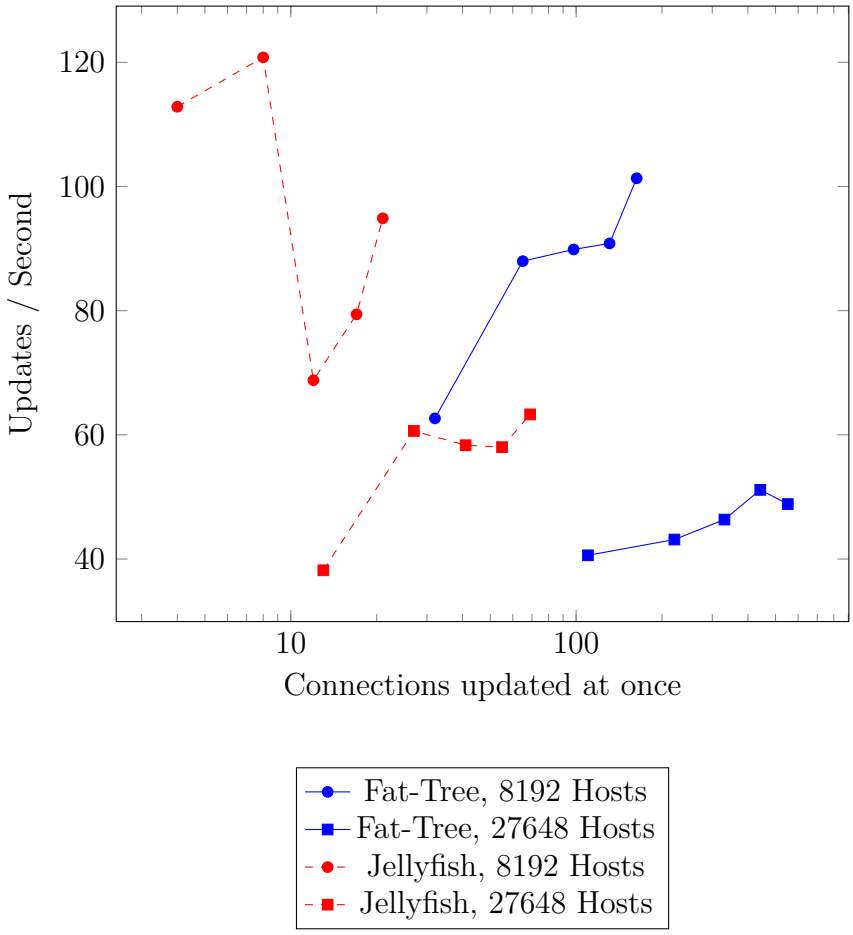
Table 6.9: Runtime to process the update of a single connection by batch-size.

	Conn.	**Time in Milliseconds**				
	Weight	‰ Connections updated at once				
Topology	**Change**	2	4	6	8	10
Fat-Tree	**20%**	15.563	10.677	9.673	9.519	8.025
8192 hosts, 16384 conn.	**60%**	16.344	11.462	11.163	11.015	9.920
Fat-Tree	**20%**	22.136	17.919	15.018	13.156	14.656
27648 hosts, 55296 conn.	**60%**	24.764	23.199	21.622	19.570	20.505
Jellyfish	**20%**	9.250	5.875	14.000	11.765	10.143
8192 hosts, 2144 conn.	**60%**	9.500	8.875	15.583	12.706	10.762
Jellyfish	**20%**	24.923	15.444	14.683	14.873	15.232
27648 hosts, 6912 conn.	**60%**	27.846	16.889	17.341	17.327	15.826

In Table 6.9 we show the times the computation needs for connection updates in batches. Measurements were done on Fat-Tree topologies with either 8192 hosts, 1280 switches and 16384 connections or 27648 hosts, 2880 switches and 55296 connections and Jellyfish topologies with either 8192 hosts, 390 switches and 2144 connections or 27648 hosts, 864 switches and 6912 connections. The table shows that processing updates in bigger batches is faster, as it was the case for the connection removals discussed in the prior section. Also for bigger topologies more time is needed to adapt the network's forward rules. The reason is again that potentially more rules need to be changed because there are more switches in the network that may use this connection on a shortest path to another switch. Figure 6.15 illustrates the maximum number of updates that can be processed per second by our system. The plot lines are mostly consistent with our predictions. Bigger update batches are represented in the graph by more connections updated at once and generally lead to

a higher throughput of possible connection updates per second. Smaller topologies offer a higher throughput, because fewer forward rules have to be adapted for each processed update. This is the case for the reasons outlined above.

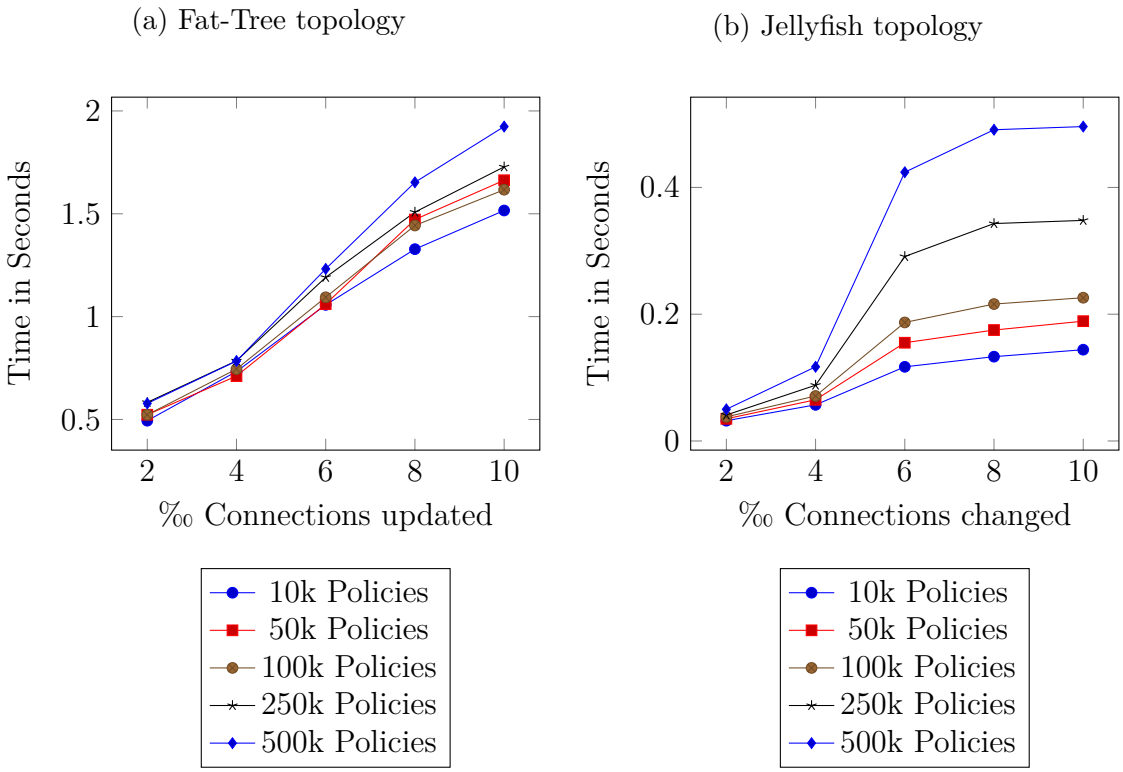
Figure 6.15: Failures per second



The last plots shown in Figure 6.16 compare the number of used policies with the time it took to process the weight updates. It shows that with ordinary numbers of policies below 100k the processing times are nearly the same for any number of policies. For Fat-Tree topologies sometimes even setups with higher policy counts process changes faster than ones with lower counts. Only when increasing the number of policies to very high magnitudes, the runtime is consistently higher than for smaller policies. For Jellyfish topologies the processing time of updates is linearly increasing with more used policies. We explain this differing between the topologies with the fact that it does not make a

difference for Jellyfish topologies which connections are changed, because it is a random topology. For Fat-Tree topologies on the other hand changing a connection higher up in the hierarchy potentially changes many more forward rules compared to changing a connection lower in the hierarchy.

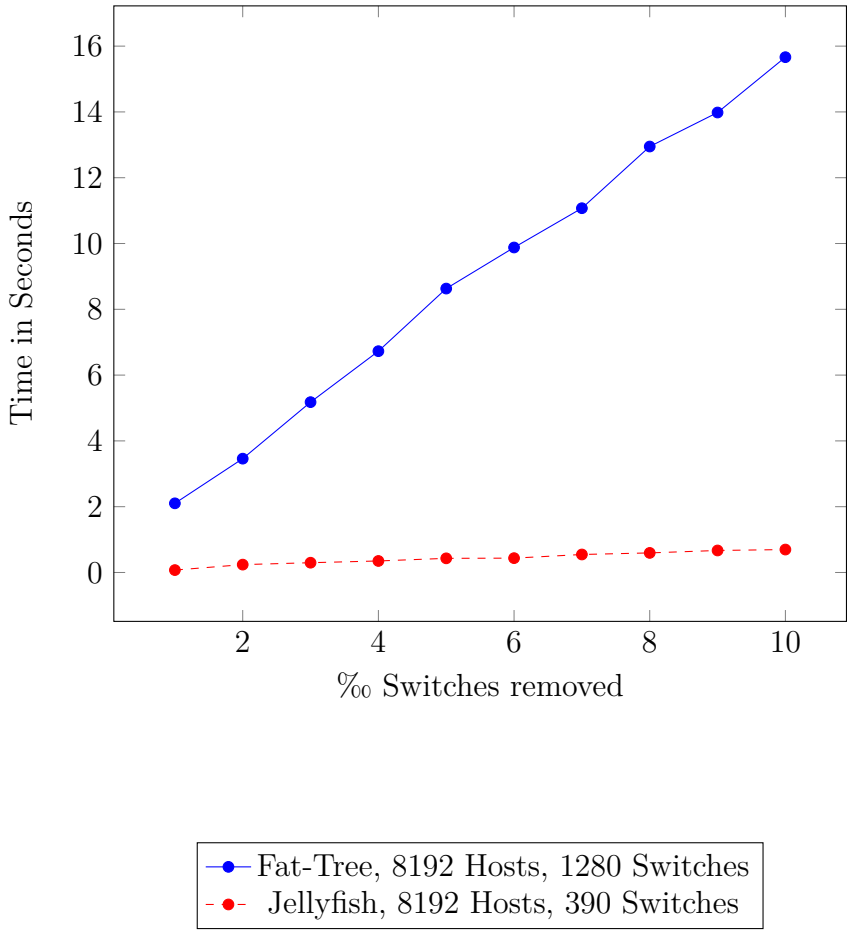
Figure 6.16: Runtime upon changing the weight of x‰ Connections by 60% in a Fat-Tree or Jellyfish topology with 8192 hosts.



6.5.3 Switch Failures

Finally we evaluate how fast our controller can recover from switch failures. To model such events we removed all connections the failing switch is part of. Thus we expected the runtimes for recovering from a switch failure to be comparable with the removal of the equal number of connections. We made our experiments for a topology size of 8192 hosts. The Fat-Tree topology contains 1280 switches and 16384 connections. The Jellyfish topology needs only 390 switches and 2144 connections to support the same number of hosts.

Figure 6.17: Runtime to process switch outages in Fat-Tree and Jellyfish topologies with 8192 hosts.



The plot in Figure 6.17 proves our claims. Runtimes behave exactly like we observed them for removal of connections, only on a bigger scale as many more connections are removed. Therefore we do not further investigate and refer to Section 6.5.1 where we discussed the processing of connection removals.

7 Conclusions

7.1 Summary

In this master thesis we investigated how a SDN controller can be built by using a dataflow computation. The advent of SDN allows to centrally evaluate the best paths to deploy in the network and obtain feedback on the current load in real time. Other SDN controller platforms that have been developed in the past use conventional programming paradigmas to leverage this potential. We think the use of a dataflow computation is a better fit for this problem because it scales to big input sizes and is able to react fast to incremental changes of the input. To prove this we implemented a prototype controller platform in the Rust programming language and evaluates its performance.

The main contributions of this master thesis are threefold. First we implemented a routing module offering competitive performance. Second we created the basis for a more formalized topology model. Third we construct the initial layout of a policy language for programmable networks. Below we discuss these in more detail.

We introduced a model for topologies and policies to define the input for our controller. A topology consists of a list of hosts, switches and connections. Connections have a weight that represents the cost to traverse it. Defining weights allows us to offer input such as link utilization, cost or propagation delay to be captured by our model. To allow the input of a topology and policies from external files, we also created a syntax for both.

We described and evaluated our prototype of an SDN controller implemented as a differential dataflow. It builds on top of the two libraries *differential-dataflow* and *timely-dataflow* written in the Rust programming language. As an approximation of the ideal paths in the network, we calculate the shortest paths regarding the given connection weights.

We also provide a thorough evaluation showing the effectiveness of our approach. To efficiently evaluate our work we also created an extensive testing framework. It includes generators for Fat-Tree as well as Jellyfish topologies and for policies of arbitrary size. We used them to show how the runtime of our controller behaves when increasing input sizes. Fat-Tree represents a widely used hierarchical network topology. Jellyfish is a randomly generated topology with very short average path-lengths that is very efficient in terms of hardware requirements. The evaluation shows that our system's two biggest strengths are its ability to scale and to process incremental changes of the network very fast. It can compute the forwarding rules for networks with 100k policies and a Fat-Tree topology containing 8192 hosts in under 12 seconds and for Jellyfish topologies of the same size in just over 2.5 seconds. When changing the weights of connections in the graph our system needs on average 15ms per changed connection to adapt the forward rules.

7.2 Directions for Future Work

In this section we describe what could be done as extension of the work we have done. During the development of our prototype and the writing of this thesis we had many ideas for improvements which are interesting to follow up.

Extend Policy Syntax The syntax for policies we introduce in this thesis is kept very simple and concise. In the future it could be replaced by a more extensive one. Extensions could allow more complex constraints, e.g. ones that allow the exclusion of certain nodes or paths.

Improve the Topology Model and Syntax The syntax for topologies we defined is also relatively simple. It could be changed to use a standardized input format like RDF[1]. This would additionally allow to easily add more fields describing the network nodes, e.g. to assign names or attach the nodes' hardware specifications. Also the internal topology

[1]RDF `https://www.w3.org/RDF/`

data type could be extended to support queries on its internals. One could for example want to obtain the list of hosts that are connected to the switch with node-id X.

Allow Real-Time Data Input At the moment we only run our computation until it is done and then terminate it. To use our controller in the real world, it would need to keep running and adapt itself to changes of the input. For this it should listen to an input stream where topology or policy changes can be announced. This would also allow to test our controller's ability to react to updates and incorporate feedback into its computations. For this the connection weights within the controller should be updated such that they penalize connections with a high traffic load. As an effect our algorithm would favor connections with free capacity and re-route parts of the packets through those to achieve an evenly distributed network load.

Deploy Forward Rules Our prototype controller calculates forward rules for the underlying network topology and the given policies. To test those forward rules they could be deployed on either an emulated or even a real network. This would require to transform the rules from the internal format into OpenFlow or P4[2]. One could then conduct experiments with varying loads of the network.

Parallelize all Code Our final suggestion for improvement is to parallelize the two code modules that can only be run single-threaded at the moment. This concerns the topology and policy parsers and the function translating the policies to tuples the dataflow computation can process. All of this code can easily be parallelized by launching several threads processing independent chunks of the input data at the same time. We did not do this yet because it is a small part of the computation time only. Though if in the future topology or policy sizes get bigger, this change could pay off.

[2]P4 `http://p4.org/`

A

Detailed Measurements

Table A.1: Average runtime and maximum deviation over five runs for different network sizes split up by computation steps. Number of worker-threads is fixed to 32, number of policies is fixed to 100k and policy length is fixed to 4.

	Hosts	Switches	Time in Seconds			
				policy		
			fwdrules	parse	convert	fwdrules
Fat-Tree	**16**	**20**	0.021 ± 0.022	0.605 ± 0.05	0.584 ± 0.189	0.112 ± 0.044
	128	**80**	0.077 ± 0.07	0.622 ± 0.043	0.548 ± 0.08	0.458 ± 0.405
	432	**180**	0.124 ± 0.006	0.624 ± 0.063	0.608 ± 0.355	0.412 ± 0.11
	1024	**320**	0.287 ± 0.005	0.617 ± 0.017	0.535 ± 0.061	0.42 ± 0.042
	2000	**500**	0.857 ± 0.004	0.647 ± 0.044	0.617 ± 0.365	0.447 ± 0.035
	3456	**720**	2.22 ± 0.051	0.661 ± 0.045	0.655 ± 0.454	0.467 ± 0.027
	8192	**1280**	10.949 ± 0.315	0.705 ± 0.067	0.559 ± 0.01	0.494 ± 0.008
	11664	**1620**	19.935 ± 0.406	0.714 ± 0.058	0.565 ± 0.019	0.496 ± 0.012
	18522	**2205**	43.722 ± 1.311	0.704 ± 0.046	0.583 ± 0.01	0.494 ± 0.02
	27648	**2880**	84.114 ± 1.568	0.723 ± 0.05	0.593 ± 0.005	0.457 ± 0.011
	93312	**6480**	654.054 ± 13.916	0.763 ± 0.051	0.558 ± 0.031	0.528 ± 0.035
Jellyfish match Fat-Tree's # Hosts	**43**	**16**	0.008 ± 0.001	0.622 ± 0.069	0.627 ± 0.323	0.067 ± 0.007
	139	**26**	0.022 ± 0.001	0.613 ± 0.044	0.607 ± 0.216	0.273 ± 0.013
	432	**54**	0.045 ± 0.002	0.613 ± 0.03	0.849 ± 0.316	0.32 ± 0.016
	1099	**103**	0.071 ± 0.004	0.631 ± 0.042	0.526 ± 0.01	0.336 ± 0.005
	2054	**154**	0.106 ± 0.003	0.639 ± 0.04	0.664 ± 0.463	0.36 ± 0.017
	3456	**216**	0.155 ± 0.003	0.65 ± 0.051	0.653 ± 0.432	0.365 ± 0.014
	8320	**390**	0.466 ± 0.01	0.665 ± 0.045	0.877 ± 0.337	0.384 ± 0.007
	11664	**486**	0.864 ± 0.018	0.674 ± 0.048	0.768 ± 0.337	0.394 ± 0.009
	18536	**662**	2.192 ± 0.057	0.676 ± 0.064	0.655 ± 0.463	0.434 ± 0.015
	27648	**864**	4.536 ± 0.091	0.698 ± 0.047	0.564 ± 0.027	0.442 ± 0.014
	93312	**1944**	36.109 ± 1.484	0.763 ± 0.082	0.629 ± 0.005	0.461 ± 0.023
Jellyfish match Fat-Tree's # Switches	**54**	**20**	0.008 ± 0.003	0.601 ± 0.019	0.562 ± 0.035	0.062 ± 0.002
	427	**80**	0.061 ± 0.004	0.62 ± 0.048	0.65 ± 0.436	0.414 ± 0.01
	1440	**180**	0.096 ± 0.003	0.663 ± 0.044	0.586 ± 0.159	0.379 ± 0.017
	3414	**320**	0.19 ± 0.005	0.646 ± 0.061	0.633 ± 0.287	0.391 ± 0.006
	6667	**500**	0.493 ± 0.031	0.656 ± 0.016	0.535 ± 0.004	0.41 ± 0.013
	11520	**720**	1.339 ± 0.071	0.67 ± 0.03	0.752 ± 0.352	0.447 ± 0.009
	27307	**1280**	7.19 ± 0.193	0.71 ± 0.035	0.571 ± 0.004	0.481 ± 0.007
	38880	**1620**	13.39 ± 0.284	0.727 ± 0.042	0.579 ± 0.015	0.475 ± 0.012
	61740	**2205**	29.239 ± 0.589	0.741 ± 0.064	0.611 ± 0.037	0.447 ± 0.018
	92160	**2880**	57.805 ± 1.897	0.755 ± 0.047	0.619 ± 0.058	0.463 ± 0.026
	311040	**6480**	453.158 ± 7.129	0.781 ± 0.048	0.639 ± 0.026	0.492 ± 0.007

Table A.2: Average runtime and maximum deviation over five runs for different numbers of worker counts split up by computation steps. Number of worker-threads is fixed to 32, number of policies is fixed to 100k and policy length is fixed to 4.

	Threads	Time in Seconds			
			policy		
		fwdrules	parse	convert	fwdrules
Fat-Tree 8192 Hosts, 1280 Sw.	**1**	85.72 ± 0.394	0.703 ± 0.084	0.484 ± 0.002	5.296 ± 0.01
	2	47.452 ± 1.382	0.72 ± 0.065	0.482 ± 0.01	2.897 ± 0.007
	4	28.329 ± 0.263	0.676 ± 0.044	0.49 ± 0.005	1.554 ± 0.007
	8	18.708 ± 0.592	0.685 ± 0.038	0.491 ± 0.01	0.883 ± 0.028
	16	14.833 ± 0.959	0.669 ± 0.051	0.511 ± 0.009	0.609 ± 0.068
	32	11.401 ± 0.073	0.682 ± 0.066	0.549 ± 0.016	0.531 ± 0.057
	64	9.036 ± 0.272	0.653 ± 0.045	1.151 ± 0.01	0.598 ± 0.238
Jellyfish 8190 Hosts, 390 Sw.	**1**	5.207 ± 0.041	0.702 ± 0.101	0.507 ± 0.008	3.278 ± 0.015
	2	2.924 ± 0.056	0.725 ± 0.056	0.508 ± 0.007	1.789 ± 0.006
	4	1.599 ± 0.041	0.671 ± 0.006	0.489 ± 0.003	0.993 ± 0.003
	8	0.991 ± 0.037	0.675 ± 0.019	0.491 ± 0.005	0.56 ± 0.008
	16	0.7 ± 0.063	0.657 ± 0.054	0.727 ± 0.335	0.439 ± 0.038
	32	0.501 ± 0.005	0.643 ± 0.034	0.754 ± 0.342	0.404 ± 0.013
	64	0.627 ± 0.09	0.648 ± 0.045	1.11 ± 0.061	0.422 ± 0.007
Fat-Tree 27648 Hosts, 2880 Sw.	**1**	648.531 ± 12.301	0.778 ± 0.07	0.57 ± 0.01	6.464 ± 0.071
	2	362.075 ± 3.63	0.776 ± 0.066	0.556 ± 0.039	3.46 ± 0.026
	4	215.786 ± 2.292	0.749 ± 0.059	0.476 ± 0.009	1.818 ± 0.006
	8	143.187 ± 3.037	0.737 ± 0.043	0.481 ± 0.031	0.974 ± 0.135
	16	114.343 ± 3.163	0.727 ± 0.05	0.598 ± 0.413	0.678 ± 0.112
	32	88.044 ± 0.766	0.713 ± 0.052	0.591 ± 0.009	0.474 ± 0.002
	64	72.447 ± 0.902	0.712 ± 0.056	1.213 ± 0.173	0.546 ± 0.021
Jellyfish 27648 Hosts, 6912 Sw.	**1**	37.499 ± 0.116	0.742 ± 0.093	0.528 ± 0.007	4.304 ± 0.019
	2	21.403 ± 0.169	0.76 ± 0.056	0.537 ± 0.011	2.435 ± 0.004
	4	12.716 ± 0.082	0.74 ± 0.045	0.501 ± 0.009	1.316 ± 0.011
	8	8.471 ± 0.107	0.708 ± 0.023	0.507 ± 0.009	0.715 ± 0.007
	16	6.634 ± 0.303	0.699 ± 0.037	0.636 ± 0.465	0.522 ± 0.039
	32	4.7 ± 0.157	0.725 ± 0.074	0.568 ± 0.009	0.466 ± 0.013
	64	3.323 ± 0.169	0.678 ± 0.032	1.147 ± 0.004	0.485 ± 0.013

Table A.3: Average runtime and maximum deviation over five runs for different numbers of used policies and worker counts split up by computation steps. Policy length is fixed to 10.

		Threads	Policies	Time in Seconds			
					policy		
				fwdrules	parse	convert	fwdrules
Fat-Tree	8192 H., 1280 Sw.	**1**	**10k**	84.744 ± 0.61	0.111 ± 0.03	0.056 ± 0.001	0.591 ± 0.004
			50k	85.000 ± 1.063	0.389 ± 0.069	0.248 ± 0.001	2.757 ± 0.026
			100k	83.528 ± 1.081	0.746 ± 0.102	0.481 ± 0.008	5.281 ± 0.023
		32	**10k**	11.641 ± 0.389	0.079 ± 0.037	0.051 ± 0.004	0.141 ± 0.142
			50k	11.642 ± 0.237	0.36 ± 0.057	0.29 ± 0.005	0.311 ± 0.01
			100k	11.496 ± 0.236	0.703 ± 0.068	0.562 ± 0.004	0.527 ± 0.013
Jellyfish	8192 H., 390 Sw.	**1**	**10k**	5.37 ± 0.099	0.105 ± 0.036	0.050 ± 0.000	0.37 ± 0.002
			50k	5.363 ± 0.07	0.383 ± 0.058	0.245 ± 0.003	1.707 ± 0.011
			100k	5.393 ± 0.033	0.716 ± 0.086	0.491 ± 0.002	3.324 ± 0.018
		32	**10k**	0.514 ± 0.025	0.065 ± 0.005	0.08 ± 0.027	0.093 ± 0.002
			50k	0.505 ± 0.015	0.344 ± 0.057	0.302 ± 0.039	0.246 ± 0.004
			100k	0.504 ± 0.009	0.674 ± 0.074	0.602 ± 0.189	0.415 ± 0.02
Fat-Tree	27648 H., 2880 Sw.	**1**	**10k**	644.585 ± 7.89	0.108 ± 0.042	0.064 ± 0.001	0.678 ± 0.003
			50k	642.833 ± 3.712	0.424 ± 0.079	0.292 ± 0.002	3.221 ± 0.035
			100k	652.396 ± 9.626	0.816 ± 0.066	0.57 ± 0.006	6.404 ± 0.039
		32	**10k**	88.747 ± 1.649	0.087 ± 0.027	0.066 ± 0.003	0.076 ± 0.003
			50k	88.828 ± 1.69	0.406 ± 0.065	0.257 ± 0.01	0.244 ± 0.01
			100k	89.24 ± 1.355	0.771 ± 0.097	0.589 ± 0.022	0.478 ± 0.032
Jellyfish	27648 H., 864 Sw.	**1**	**10k**	37.702 ± 0.335	0.11 ± 0.044	0.052 ± 0.001	0.484 ± 0.002
			50k	37.875 ± 0.269	0.402 ± 0.068	0.254 ± 0.001	2.285 ± 0.008
			100k	38.005 ± 0.327	0.754 ± 0.096	0.505 ± 0.003	4.312 ± 0.006
		32	**10k**	4.644 ± 0.206	0.079 ± 0.034	0.064 ± 0.003	0.1 ± 0.002
			50k	4.647 ± 0.104	0.346 ± 0.016	0.296 ± 0.026	0.276 ± 0.006
			100k	4.703 ± 0.108	0.703 ± 0.042	0.653 ± 0.37	0.476 ± 0.01

Table A.4: Average runtime and maximum deviation over five runs for policies with different constraint lengths and worker counts split up by computation steps. Number of policies is fixed to 100k.

		Threads	Pol.-Len.	Time in Seconds			
					policy		
				fwdrules	parse	convert	fwdrules
Fat-Tree	8192 H., 1280 Sw.	**1**	**2**	83.758 ± 0.727	0.442 ± 0.074	0.222 ± 0.003	3.406 ± 0.014
			4	83.658 ± 0.739	0.727 ± 0.097	0.461 ± 0.011	5.294 ± 0.014
			10	83.158 ± 0.429	1.602 ± 0.165	1.291 ± 0.017	10.552 ± 0.07
		32	**2**	11.553 ± 0.104	0.416 ± 0.061	0.277 ± 0.004	0.352 ± 0.01
			4	11.462 ± 0.253	0.705 ± 0.071	0.674 ± 0.438	0.528 ± 0.011
			10	11.558 ± 0.31	1.522 ± 0.105	1.788 ± 1.102	1.063 ± 0.033
Jellyfish	8192 H., 390 Sw.	**1**	**2**	5.37 ± 0.057	0.436 ± 0.083	0.241 ± 0.004	2.09 ± 0.013
			4	5.387 ± 0.028	0.724 ± 0.105	0.509 ± 0.002	3.322 ± 0.014
			10	5.286 ± 0.064	1.555 ± 0.155	1.409 ± 0.006	7.177 ± 0.045
		32	**2**	0.498 ± 0.01	0.381 ± 0.032	0.283 ± 0.103	0.289 ± 0.012
			4	0.502 ± 0.012	0.693 ± 0.074	0.566 ± 0.084	0.415 ± 0.009
			10	0.518 ± 0.009	1.485 ± 0.102	1.501 ± 0.038	0.783 ± 0.017
Fat-Tree	27648 H., 2880 Sw.	**1**	**2**	647.169 ± 9.082	0.477 ± 0.084	0.262 ± 0.004	4.025 ± 0.017
			4	647.825 ± 4.651	0.786 ± 0.101	0.54 ± 0.006	6.425 ± 0.037
			10	645.833 ± 4.852	1.703 ± 0.055	1.361 ± 0.047	13.271 ± 0.089
		32	**2**	89.06 ± 1.437	0.449 ± 0.049	0.249 ± 0.006	0.297 ± 0.023
			4	88.548 ± 0.463	0.745 ± 0.061	0.603 ± 0.012	0.47 ± 0.011
			10	89.222 ± 0.811	1.569 ± 0.102	1.531 ± 0.108	1.045 ± 0.024
Jellyfish	27648 H., 864 Sw.	**1**	**2**	37.801 ± 0.286	0.472 ± 0.079	0.25 ± 0.003	2.829 ± 0.009
			4	37.827 ± 0.186	0.766 ± 0.11	0.532 ± 0.008	4.297 ± 0.01
			10	37.442 ± 0.355	1.618 ± 0.14	1.43 ± 0.022	8.676 ± 0.085
		32	**2**	4.7 ± 0.109	0.438 ± 0.065	0.282 ± 0.004	0.321 ± 0.005
			4	4.685 ± 0.089	0.745 ± 0.088	0.675 ± 0.439	0.471 ± 0.009
			10	4.72 ± 0.153	1.503 ± 0.067	1.794 ± 1.112	0.919 ± 0.026

Table A.5: Average runtime and maximum deviation over five runs for batch-removals of connections from the topology. Number of worker-threads is fixed to 32, number of policies is fixed to 100k and policy length is fixed to 4.

	Time in Seconds			
‰ Conn. removed	**Fat-Tree** 8192 H., 1280 Sw.	**Jellyfish** 8192 H., 390 Sw.	**Fat-Tree** 27648 H., 2880 Sw.	**Jellyfish** 27648 H., 864 Sw.
2	0.500 ± 0.044	0.082 ± 0.003	4.000 ± 0.430	0.301 ± 0.034
4	0.751 ± 0.062	0.093 ± 0.002	6.326 ± 0.832	0.495 ± 0.078
6	1.146 ± 0.113	0.108 ± 0.006	7.941 ± 0.529	0.753 ± 0.084
8	1.303 ± 0.114	0.169 ± 0.004	10.053 ± 0.798	0.906 ± 0.138
10	1.677 ± 0.085	0.177 ± 0.014	12.415 ± 2.261	1.080 ± 0.101

Table A.6: Average runtime and maximum deviation over five runs for batch-changes of connections in the topology. Number of worker-threads is fixed to 32, number of policies is fixed to 100k and policy length is fixed to 4.

		Time in Seconds			
Weight Change	**Conn. changed**	**Fat-Tree** 8192 H., 1280 Sw.	**Jellyfish** 8192 H., 390 Sw.	**Fat-Tree** 27648 H., 2880 Sw.	**Jellyfish** 27648 H., 864 Sw.
20 %	**2‰**	0.498 ± 0.028	0.037 ± 0.002	2.435 ± 0.309	0.324 ± 0.065
	4‰	0.694 ± 0.068	0.047 ± 0.001	3.960 ± 0.445	0.417 ± 0.022
	6‰	0.948 ± 0.040	0.168 ± 0.011	4.971 ± 0.495	0.602 ± 0.050
	8‰	1.247 ± 0.100	0.200 ± 0.008	5.815 ± 0.792	0.818 ± 0.067
	10‰	1.308 ± 0.095	0.212 ± 0.010	8.090 ± 0.693	1.051 ± 0.058
60 %	**2‰**	0.523 ± 0.039	0.038 ± 0.002	2.724 ± 0.216	0.362 ± 0.054
	4‰	0.745 ± 0.128	0.071 ± 0.004	5.127 ± 0.306	0.456 ± 0.017
	6‰	1.094 ± 0.043	0.187 ± 0.009	7.157 ± 0.562	0.711 ± 0.063
	8‰	1.443 ± 0.077	0.216 ± 0.011	8.650 ± 0.473	0.953 ± 0.110
	10‰	1.617 ± 0.089	0.226 ± 0.012	11.319 ± 1.463	1.092 ± 0.036

Table A.7: Average runtime and maximum deviation over five runs for batch-changes of connections in the topology with different number of policies. Weights are changed by 60 %, number of worker-threads is fixed to 32, topology size is fixed to 8192 hosts and policy length is fixed to 4.

# Policies	Conn. changed	Time in Seconds	
		Fat-Tree 8192 H., 1280 Sw.	**Jellyfish** 8192 H., 390 Sw.
10k	**2‰**	0.494 ± 0.028	0.032 ± 0.001
	4‰	0.733 ± 0.053	0.057 ± 0.006
	6‰	1.057 ± 0.109	0.117 ± 0.009
	8‰	1.328 ± 0.136	0.133 ± 0.006
	10‰	1.516 ± 0.154	0.144 ± 0.011
50k	**2‰**	0.523 ± 0.069	0.035 ± 0.002
	4‰	0.711 ± 0.087	0.065 ± 0.002
	6‰	1.061 ± 0.072	0.155 ± 0.008
	8‰	1.472 ± 0.122	0.175 ± 0.012
	10‰	1.663 ± 0.132	0.189 ± 0.012
100k	**2‰**	0.523 ± 0.039	0.038 ± 0.002
	4‰	0.745 ± 0.128	0.071 ± 0.004
	6‰	1.094 ± 0.043	0.187 ± 0.009
	8‰	1.443 ± 0.077	0.216 ± 0.011
	10‰	1.617 ± 0.089	0.226 ± 0.012
250k	**2‰**	0.583 ± 0.099	0.041 ± 0.001
	4‰	0.785 ± 0.120	0.088 ± 0.004
	6‰	1.191 ± 0.133	0.291 ± 0.014
	8‰	1.508 ± 0.111	0.343 ± 0.019
	10‰	1.728 ± 0.123	0.348 ± 0.016
500k	**2‰**	0.578 ± 0.106	0.050 ± 0.006
	4‰	0.784 ± 0.113	0.117 ± 0.007
	6‰	1.232 ± 0.198	0.424 ± 0.007
	8‰	1.653 ± 0.069	0.491 ± 0.007
	10‰	1.924 ± 0.182	0.496 ± 0.007

Table A.8: Average runtime and maximum deviation over five runs for batch-removals of switches in the topology. Number of worker-threads is fixed to 32, topology size is fixed to 8192 hosts, policy length is fixed to 4 and number of policies is fixed to 100k.

	Time in Seconds	
Switches removed	**Fat-Tree** 8192 H., 1280 Sw.	**Jellyfish** 8192 H., 390 Sw.
1‰	2.102 ± 0.18	0.073 ± 0.004
2‰	3.46 ± 0.248	0.239 ± 0.01
3‰	5.178 ± 0.43	0.297 ± 0.005
4‰	6.727 ± 0.584	0.35 ± 0.014
5‰	8.63 ± 0.939	0.43 ± 0.007
6‰	9.879 ± 0.722	0.436 ± 0.02
7‰	11.073 ± 0.637	0.547 ± 0.019
8‰	12.948 ± 0.908	0.595 ± 0.039
9‰	13.982 ± 0.625	0.669 ± 0.022
10‰	15.664 ± 0.589	0.697 ± 0.024

List of Tables

List of Figures

List of Listings

Bibliography

[AFLV08] M. Al-Fares, A. Loukissas, and A. Vahdat. "A scalable, commodity data center network architecture." *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, 63–74, 2008.

[Apa] S. F. Apache. "Apache License, Version 2.0." `http://www.apache.org/licenses/LICENSE-2.0`.

[Bla15] J. Blandi. *Why Rust?* O'Reilly, 2015.

[CLMR16] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. "Explaining Outputs in Modern Data Analytics." *Tech. rep.*, ETH-Zürich, 2016.

[FHF+11] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. "Frenetic: A network programming language." In *ACM Sigplan Notices*, vol. 46, pp. 279–291. ACM, 2011.

[KREV+15] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-Defined Networking: A Comprehensive Survey." *Proceedings of the IEEE*, vol. 103, no. 1, 14–76, 2015.

[KZMB14] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou. "Feature-based comparison and selection of Software Defined Networking (SDN) controllers." In *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, pp. 1–7. IEEE, 2014.

[McSa] F. McSherry. "Differential Dataflow." `https://github.com/frankmcsherry/differential-dataflow`.

[McSb] F. McSherry. "Timely Dataflow." `https://github.com/frankmcsherry/timely-dataflow`.

[MIT] MIT. "The MIT License (MIT)." `https://opensource.org/licenses/MIT`.

[MMI+13] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. "Naiad: A Timely Dataflow System." In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455. ACM, 2013.

[MMII13] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. "Differential Dataflow." In *Proceedings of the 6th Biennial Converence on Innovative Data Systems Research*. CIDR, 2013.

[RMF+13] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. "Modular sdn programming with pyretic." *Technical Report of USENIX*, 2013.

[RTa] Rust-Team. "The Rust Book." `https://doc.rust-lang.org/book/`. [Online; accessed June 4, 2016].

[RTb] Rust-Team. "The Rust FAQ." `https://www.rust-lang.org/faq.html`. [Online; accessed June 4, 2016].

[SBM+14] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. "Merlin: A language for provisioning network resources." In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 213–226. ACM, 2014.

[SHPG12] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. "Jellyfish: Networking data centers randomly." In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 225–238. 2012.

[VTVR15] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. "Central control over distributed routing." *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 43–56, 2015.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Expressing the Routing Logic of a SDN Controller as a Differential Dataflow

Authored by (in block letters):

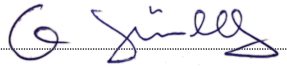
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Stücklberger	Christian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 15.7.2016	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.