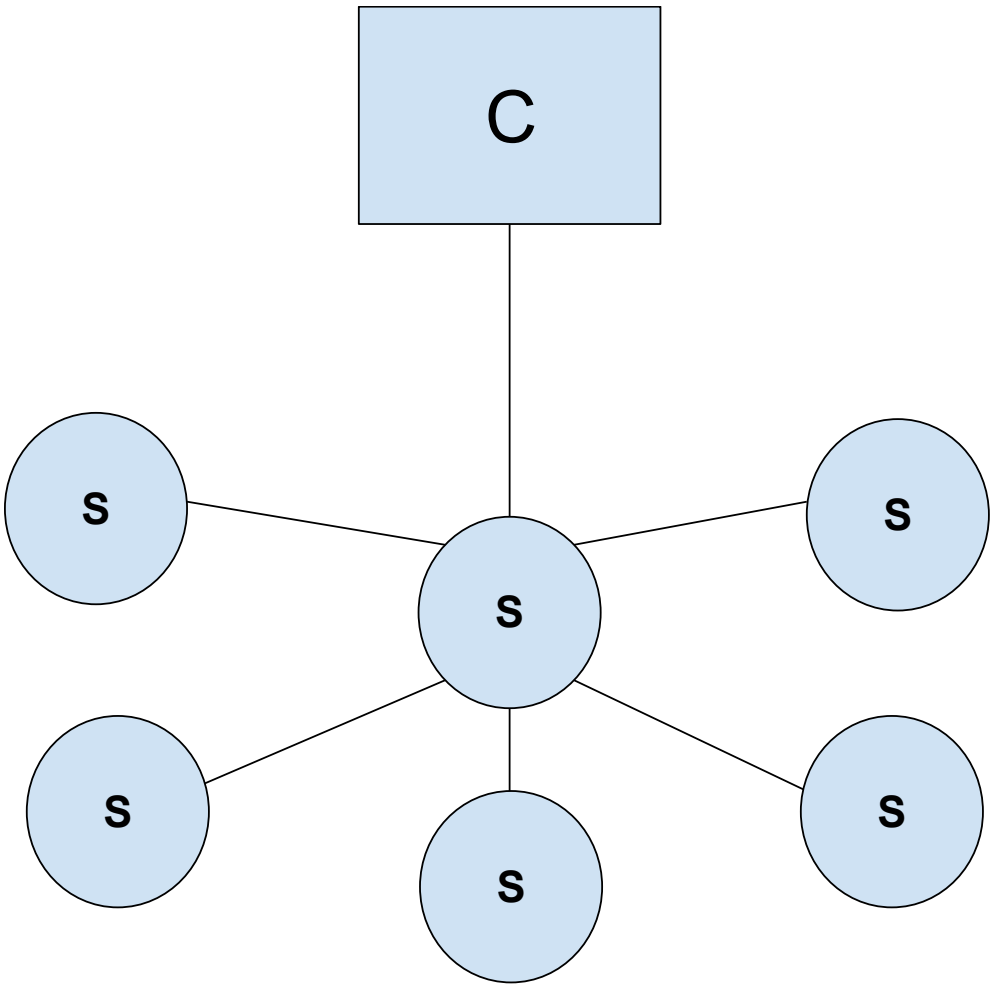
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Implementing and evaluating a Self-Stabilizing Software Defined Network Control Plane

Ivan Tannerud Anton Lundgren

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018

Implementing and evaluating a Self-Stabilizing Control Plane for Software Defined Networks

This report documents the implementation and evaluation of the Renaissance algorithm, which provides a self-stabilizing control plane for Software Defined Networks. A prototype implementation using the Floodlight controller is provided and evaluated, as well as a proof-of-concept implementation in the ONOS controller.

Ivan Tannerud Anton Lundgren

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

This report documents the implementation and evaluation of the Renaissance algorithm, which provides a self-stabilizing control plane for Software Defined Networks. A prototype implementation using the Floodlight controller is provided and evaluated, as well as a proof-of-concept implementation in the ONOS controller.

Supervisor: Elad Michael Schiller, Computer Science and Engineering
Examiner: Pedro Petersen Moura Trancoso, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Software Defined Networking (SDN) provides an attractive alternative to traditional, manual, error prone networks. SDN provides the opportunity to add programmability to networks, and decouple the control plane from the data plane. By doing so, a network administrator may control an entire network from a single SDN controller. While the work in this field is extensive, the question of how to maintain the connection between the control plane and the data plane has not received much attention. There is an algorithm called Renaissance that approaches the problem by utilizing self-stabilization, which is a strong notion of fault-tolerance for distributed systems. However, only a proof-of-concept implementation has been created for Renaissance this far.

This report presents a prototype of the self-stabilizing SDN control plane algorithm that is implemented using the Floodlight controller. In addition we present an evaluation of the prototype. The evaluation shows that the prototype is fault-tolerant and able to recover from transient faults, which means that it is self-stabilizing. The stabilization time is shown to depend mostly on the degree of a network, but also on the number of nodes. Lastly, we present a proof-of-concept implementation using the ONOS controller.

Keywords: Computer, Science, Software, Defined, Networks, SDN, ONOS, Floodlight.

Acknowledgements

We would like to thank our supervisor Elad Schiller for his invaluable help and expertise throughout this project. Also, we give thanks to Stefan Schmid for sharing his extensive knowledge of Software Defined Networking. We also want to thank Pedro Petersen Moura Trancoso, the examiner of this project. Last but not least, we thank Michael Tran and Emelie Ekenstedt for their help with the Floodlight controller.

Ivan Tannerud and Anton Lundgren, Gothenburg, 2018

Contents

List of Figures

1

Introduction

Software Defined Networking (SDN) presents a promising alternative to manually configured, error prone traditional networks. The idea of adding programmability to the networks and as such simplify management dramatically has lead to the concept gaining momentum in recent years [19]. The main idea of an SDN is to decouple the control plane from the data plane allowing for direct programmability of the control plane. Rather than programming each switch individually, the switches connect to a centralized controller, which in turn can be programmed to install rules on the switches and establish network communication. This centralized global view of the network adds flexibility and eases the work of maintaining it.

The benefits of SDN are clear and the work on the field is extensive [15]. However, the work on maintaining a reliable connection between the control plane and the data plane has received less attention [22]. Self-stabilization is a strong notion of fault tolerance for distributed systems, which means it can be applied to software-defined networks. It refers to the ability to recover to a legitimate state after an arbitrary number of faults, including transient faults, within a bounded period of time, provided that no additional faults occur during that time. To have the control plane of an SDN working in a self-stabilizing manner is attractive as it provides high availability and fault tolerance. Canini et al. presents the algorithm Renaissance, which is a self-stabilizing SDN control plan algorithm [11]. Renaissance is proven analytically to be to be self-stabilizing, but a prototype implementation has not been created before. In this report we present a prototype implementation and evaluation of Renaissance using an existing SDN controller called Floodlight. We also present a proof-of-concept implementation in the ONOS SDN controller.

1.1 Challenge description

This project focuses on extending upon the work of Canini et al. [9] [11] by implementing the self-stabilizing algorithm for the control plane of an SDN described in the paper, called Renaissance. The preliminary ideas for [11] appeared in [10]. Canini et al. provides analytic proofs of Renaissance's self-stabilizing property, while

this thesis aims to validate the analysis via our implementation and evaluation. The project builds on earlier work by Tran, who provided an preliminary proof-of-concept implementation of Renaissance [11] (which assumes the use of [13] [14]) in the Floodlight Controller [2]. However, Tran's implementation deviates from the Renaissance algorithm. This project aims to provide a complete prototype implementation loyal to the algorithm in [11] and evaluate it. We also aim to provide a second implementation in the ONOS SDN controller, meant to work as a proof-of-concept.

Problem description

Renaissance provides an answer to the question of how non-faulty controllers can maintain bounded (in-band) communication delays to any switch and non-faulty controller. It is a self-stabilizing algorithm that allows each controller to discover the network and create k-fault-resilient flows to all discovered switches and controllers. The algorithm also enables controllers to remove any stale configuration information in the system. A fresh configuration is provided at all time by monitoring failing and returning nodes and links. All of the above has to be implemented by adding functionality to existing SDN controllers while retaining the existing functionality. By implementing a prototype for the Renaissance algorithm and evaluating it, we can validate the proofs provided by Canini et al. in [11].

Implementation challenges

The Renaissance algorithm considers an abstract SDN architecture, where global controllers query switches for their local neighbourhood. However, the chosen SDN controllers go after fault tolerance in another manner. This leads to multiple implementation challenges described in the following subsections.

Floodlight. Renaissance discovers the network by querying switches about their local neighbourhood. However, since this is not a function that switches support, another SDN controller had to be implemented to work as a proxy for the queries of the global controller. This controller is called a local controller. What this means is that the code divides the Renaissance algorithm in two, namely the global and the local controllers. Implementing this in Floodlight requires extensive knowledge of SDN and OpenFlow (see section 2.3) to make sure the two controllers cooperate in such a way that Renaissance functionality may be gained.

ONOS. ONOS is a modular system, built upon a multitude of applications and services, working together towards a common goal. The global and local controller of Renaissance will have to be implemented as applications that work in harmony

with the existing apps. The functionality of ONOS is to be maintained while new functionality is added, which is a challenging task. A system architecture for how Renaissance may work in ONOS has to be provided. This entails understanding the code of a project that consists of tens of thousands lines of code and figuring out how to maneuver the API in such a way that the desired functionality may be gained. The Renaissance algorithm has to be divided into two SDN controllers in the ONOS system as well.

1.2 Related Work

The implementations described in this report is inspired by two different results within the field of SDN. The first one, Medieval, is a stepping stone towards a self-stabilizing SDN and the second one, Renaissance, is the algorithm that the implementations in this report are based on. It also builds on another attempt at creating a Floodlight implementation of Renaissance. The research is described in more detail below.

Medieval

Schiff et al. presents a plug-and-play in-band solution for a control network [21]. Medieval [21] was implemented using the OpenFlow Protocol and much like Renaissance, aimed to establish a connection between controllers and switches in the network. However, while Medieval is a synchronous system, Renaissance is apart from a failure detector asynchronous. This means that each controller who can move forward with its execution, will move forward. Controllers do not need to wait on the slowest one, making Renaissance a faster algorithm. Also, Medieval does not consider transient faults and thus, it cannot be considered to be self-stabilizing.

Renaissance

The Renaissance algorithm aims to provide k-fault resilient flows from each controller to every other node in the network. This is done in a self-stabilizing manner and in-band. Global controllers continuously query the switches for their local topology via a local controller proxy. Controllers also remove stale information installed by unreachable controllers and maintain the k-fault resilient flows in an asynchronous manner, which leads to a fast self-stabilizing solution.

The algorithm is presented in detail in by Canini et al. in [11], but will also be described more in section 3.2.

Tran's implementation of Renaissance

Tran has provided a preliminary proof-of-concept implementation of the Renaissance algorithm by implementing a local and global controller in Floodlight [2]. After scrutinizing the implementation, we found multiple differences compared to the Renaissance algorithm, see below. The work described in this report aims provide a prototype implementation of the algorithm described in [11]. In this report we are also using slightly different topologies for the evaluation. As such, the results gained from the evaluation differs in terms of performance compared to Tran's results.

Algorithm deviations. This section documents a number of deviations from the Renaissance algorithm [11] found in Tran's preliminary implementation.

- In each round of the Renaissance algorithm, responses are processed and new queries are sent out, dependent on the content of each response. In Trans implementation, the querying task and the task which handles responses run independently, making it hard to ensure the correctness of the queries. Instead, there should be one task for querying which is dependent on the completion of the task which handles responses.

- There are certain tasks in the Renaissance algorithm which are to be carried out only when a new round is initiated. A round in Renaissance refers to when a global controller have received a query reply with the correct label from every node it queried. Tran's implementation does not keep track of new rounds, and thus it does not adhere to only performing certain tasks at the start of new rounds.

- A new round in the Renaissance algorithm also comes with a new label, which is used to make sure all nodes have responded within a single round. Without keeping track of whether a new round has started or not, the labels can also be mixed up.

- In Renaissance, a controller that received too many responses to store provides a C-reset. The concept of C-reset is absent from Tran's implementation.

1.3 Motivation

The motivation for this project a prototype implementation, loyal to the Renaissance algorithm and as such be able to evaluate the algorithm. The correctness of the Renaissance algorithm has been proven analytically [11], and we aim to validate the proofs via evaluation. Simply put, the goal of this project is to prove the

correctness of the algorithm and evaluate it via an implementation in the Floodlight SDN controller.

1.4 Our contribution

We contribute to the Renaissance project with a prototype implementation of a self-stabilizing SDN control plane that is loyal to the algorithm of Renaissance. Our implementation is validated via an evaluation in Mininet [4], done according to the benchmark of earlier tests by Tran. Our implementation is to be used in a lab at Chalmers University of Technology. We also provide a proof of concept implementation in the ONOS SDN controller.

The prototype implementation of Renaissance

The main contribution of this project has been to provide a prototype implementation of Renaissance [11]. The implementation provided is loyal to the Renaissance algorithm, and can therefore be used to evaluate it. Our work is the first time Renaissance has been implemented and as such the first time the analytical proofs provided in [11] have been validated via evaluation.

Evaluation of the Renaissance algorithm

In this project, we provide an evaluation of the implementation in the Floodlight controller, using Mininet as the evaluation platform. We are able to show that the time it takes to bootstrap the network depends mostly on network diameter. This is because the networks with larger amounts of nodes and a smaller diameter are bootstrapped quicker than the opposite. For example, we show that the Clos network (20 nodes, diameter 4) stabilizes in roughly 8 seconds, compared with the smaller network B4 (12 nodes, diameter 5), which stabilizes in roughly 9 seconds, due to its larger diameter. Note that all networks used can be found in Appendix A1.

We show that only when the amount of nodes grow significantly will it have an impact on stabilization time, as query responses from all nodes must be processed before a new round can start. For example, the Ebone network (with 172 nodes, diameter 10), stabilizes in 55 seconds, while the AT&T backbone (96 nodes, diameter 9) only needs roughly 20 seconds. Also, the amount of controllers used when bootstrapping the networks increases the stabilization time linearly. For example, the Telstra backbone stabilizes in 10 seconds when using one global controller, in 15 seconds when using 4, and in 20 seconds when using 7 global controllers. These

stabilization times were measured when quering the network once every 500 milliseconds. We find that the query time has an effect on the stabilization time, an example of this being the B4 network (12 nodes, diameter 5). B4 stabilizes in 15 seconds when our global controllers wait one second between each query round, but only needs 5 seconds when the time in between query rounds is decreased to 100 ms. The stabilization time remains the same until we lower the query time to 10 ms, at which point it is increased dramatically, likely due to network congestion.

Our results show that ping [18] communication is possible using our implementation, even in the presence of link failures. For example, the B4 network maintains a ping round trip time of 180 ms when two hosts at the distance of the diameter communicate. Our findings show that this time increases to 200 ms when a link failure occurs on the path between the two hosts. After three seconds, our system recovers the RTT back to 180 ms after the system recovers by finding a new path with equal distance compared to the original one.

We can also show that the implementation deals with failures, and recovers within seconds from link failures, controller failures and switch failures. For example, the B4 network recovers from a controller failure in under two seconds, and the same goes for switch and link failures. For the largest network Ebone, the recovery time from a controller failure is roughly 10 seconds, while a link failure takes roughly 15 seconds to recover from. The amount of failures occurring simultaneously has no significant effect on recovery time. We note that the Telstra network stabilizes from a controller failure in roughly two seconds, whether it is one controller failure, or six. We also note similar behaviour during during the recovery of 1-6 link failures, all recovered from in three seconds in the Telstra network.

The throughput of the networks is barely affected even in the event of a link failure, which only lowers the throughput by around 50-75 MBits/s. Once the system recovers from the link failure, the throughput also recovers completely. For example, we see that the AT&T backbone network has a throughput of circa 500 MBit/s in the absence of failures. This throughput drops to 430 in after a link failure, but recovers back to 500 in only two seconds. Since we above provide bounded recovery times for all different types of faults, provided that no more faults occur during recovery, we can also conclude that the algorithm is indeed self-stabilizing.

Laying the foundation of a student's lab

The code provided in this project lays the foundation of a student's lab at Chalmers University of Technology. The lab is to be featured in a computer networks course in order to teach students about SDN and provide a clear hands on example of using self-stabilizing algorithms in the context of computer networks. For the lab, parts of the code is removed in order to be re-implemented by the students. In order for the students to be able to grasp and modify the code, we provide a well documented

implementation.

ONOS proof of concept implemetation

We provide a proof of concept implementation using the ONOS SDN Controller. This was done by interpreting the Renaissance algorithm for a system design that fits ONOS. No evaluation of the ONOS implementation is provided in this report, as the implementation lays a foundation for future work which may need to be implemented in a controller other than Floodlight.

2

Background Knowledge

This section describes concepts such as SDN and self-stabilization that are necessary for understanding the implementation detailed in this report. The SDN communication protocol OpenFlow is described, as well as the evaluation environment, Mininet. The SDN controllers Floodlight and NOS are described further as well.

2.1 SDN

The idea of programmable networks gained momentum with the emergence of Software Defined Networking (SDN). Today SDN is an important architecture for the management of large, complex networks. The main idea of SDN is to decouple the data plane from the control plane, and as such easily reconfigure the network when needed. The switches in the network simply forward traffic based on rules installed by controllers in the control plane. This allows for a much simpler, more dynamic and more fault tolerant solution compared to the traditional static, manually configured networks [17] [19].

Controllers and switches in an SDN architecture can interface via the OpenFlow protocol, described in detail in section 2.3.

2.2 Self-stabilization

Fault tolerance is an important property of a computer system, as a continued service in the presence of faults is desired. Systems therefore often employ some form of fault tolerance, although not many reach the level of self-stabilization. This section describes different fault models and highlights the difference between normal fault tolerance and self-stabilization.

Fault Tolerance

Fault tolerance is the guarantee of a continued service even in the presence of faults. However, in order to describe a fault, a fault model is needed. In other words, a way to model what one considers a particular fault to be, which is needed in order to decide how to react to it. Fault tolerant systems or algorithms often consider the following types of fault models.

- **Packet loss, duplication, reordering**
  Packets may not arrive in the order they were sent, arrive more than once, or not arrive at all. This needs to be considered when designing a fault tolerant system.

- **Node crashes and recovery**
  A node in the system may at anytime crash, and at a later point recover. It is important for a fault tolerant system to deal with these crashes and make sure a correct state is maintained. In this project, only fail-stop failures are considered. This means that a failure or a recovery is always noticed by the system.

Transient Faults

Self-Stabilizing algorithms do not only consider the types of faults described above, but also an additional fault type; transient faults. A temporary violation of the assumptions according to which the system is assumed to operate is called a transient fault. In other words, a transient fault is something that cannot be foreseen or predicted, but rather a fault which violates the nature of the system.

An algorithm is said to be self-stabilizing if it can recover after the occurrence of transient faults within a bounded period of time, provided that no more transient faults occur during the time of recovery [12].

Self-stabilization and networks

Self-stabilization is of utter importance in networks, and as such in the internet. If we can not guarantee a recovery from faults within a bounded period of time, the network may remain inoperative until each node in the network is brought down and re-booted. A malfunctioning node may still cause a self-stabilizing network to be dysfunctional, but only until the faulty node is removed.

This means that if a saboteur injects faulty packages into a network, a self-stabilizing network will recover once this injection stops, while a network without self-stabilizing

properties would have to be brought down completely and rebooted, which is both costly and time consuming, especially for a large network. In essence, a self-stabilizing network guarantees easy repairs and quick recovery, rather than costly repairs with long down time [20].

2.3 OpenFlow

OpenFlow is a protocol allowing communication between SDN controllers and network devices. Through OpenFlow, controllers may access the forwarding (data) plane of routers and switches, allowing for manipulation of flow tables, etc. The Open Networking Foundation (ONF) defines OpenFlow as the first communication interface between the control- and data plane of an SDN architecture, and it is today considered the industry standard [7].

Unlike a traditional switch, the data- and control plane of an OpenFlow switch is separated. Rather than residing on the switch, the control plane of an OpenFlow switch resides on a separate SDN controller. The communication between this controller and the OpenFlow switch, is made possible via the OpenFlow protocol. In the event that a switch does not know how to handle a data packet, a *PacketIn* message is sent to the controller via the control plane. This will lead to the controller sending a *PacketOut* to the switch, telling it to drop the packet, or to install a new flow entry, which can be used to direct the data packet and future packets of the same kind.

Flow Entries

The flow entry is appended to the flow table of a switch in order to allow for forwarding decisions. Each entry should contain the following:

- **Match Field**
  Used to match incoming packets, consists of an ingress port and packet headers (Such as TCP, IP and Ethernet headers).
- **Priority**
  Used for matching precedence of the flow entries.
- **Counters**
  Statistics updated each time a packet matches on a flow entry.
- **Instructions**
  Instructions to be applied to the packet upon a packet match, such as forward-

ing to a specific out port.

- **Timeouts**
  A time out for when the flow expires, can be hard or soft (only times out if no packets are matched in a certain time)

- **Cookie**
  An identification of the flow set by the controller, used for flow modification and flow deletion.

Group Tables

Flow entries in OpenFlow also have the ability to point to a group table for more complex forwarding directives. A group table consists of the following:

- **Group ID**
  A 32 bit unsigned integer used to identify the table on the OpenFlow switch.

- **Group Type**
  One out of four group types (indirect, select, all and fast fail-over), each with their own specification of behaviour and action buckets.

- **Counters**
  Statistics updated each time a packet matches on a flow table.

- **Action buckets**
  An ordered list of action buckets. An action bucket is a set of instructions to be applied to the packet.

This means that one flow entry can result in a number of different actions taken by the switch depending on which action buckets are used. [6]

2.4 Open vSwitch

Open vSwitch (OVS), or Open Virtual Switch, is an open source multilayer virtual switch. Open vSwitch supports the OpenFlow protocol and is one of the virtual switches available in Mininet [5].

2.5 Mininet

Mininet is a network emulator which can be used to perform inexpensive testing and research on a network. It can be used to define a network topology and create a virtual network of hosts, switches, controllers and links. Mininet supports custom made switches, OpenFlow switches and Open vSwitch, which is the standard if nothing else is specified when running the emulator. Any type of controller can be used as long as it is a controller that follows the SDN protocol.

Software that is developed and tested for Mininet switches, SDN controllers and OpenFlow switches can be transferred to a real network of the same topology with minor changes. This is due to the fact that Mininet runs standard network stack and applications for Unix/Linux as well as the real Linux Kernel [4].

2.6 Floodlight

Floodlight is a java-based open source SDN control platform. It was implemented using the OpenFlow protocol, and can be used to manage flows in an SDN environment. Also, it is backed by a large open community, which translates to it being maintained and active, allowing for easy access to updated information on how to use it. This is crucial when using a platform for a project as SDN controller projects or open source projects in general may be discontinued or poorly updated and documented. Another pro with using Floodlight is that it is easy to set up, and offers a module loading system, making it possible to inject ones own code and adding more functionality. Floodlight is one of the most mature and used SDN controllers, and makes for a good SDN environment to run tests in [16].

2.7 ONOS

ONOS, or Open Network Operating System, is a java-based open source SDN control platform with focus on scalability, performance and availability [8]. ONOS uses OpenFlow as its SDN protocol and is a promising SDN platform with impressive functionality and support. ONOS achieves SDN functionality through multiple subsystems, working together in a modular fashion. The different subsystems are responsible for different parts of the system, examples being the device, link and host subsystems. This modular nature makes adding functionality possible by creating modules to run together with the already existing ones. This allows developers to make use of all the functionality of ONOS, and add new functionality without disrupting the existing ones, provided the new modules do not disrupt the existing ones, which can be challenging.

2.8 The Renaissance algorithm

Algorithm 1: The Renaissance algorithm, our own interpretation

Local state: *replies* $\subseteq \{m(j) : c_j \in P\}$ contains the responses most recently received;
The current and previous synchronization round are denoted by *currentTag* and *previousTag*;
Interface *myRules(G,j,tag)*: returns the rules of p_i on switch p_j given topology G on round *tag*;
while *true* **do**
    Remove any response from *replies* that comes from unreachable senders and remove any response with a tag other than *previousTag* and *currentTag*. In addition, remove any response from *replies* that comes from p_i and then add a list of the directly connected neighbours, $N_c(i)$;
    if *there is a response in replies (with currentTag) from every node that is considered reachable in relation to the computed local topology, G, in replies* **then**
        Set *previousTag* to *currentTag* and set *currentTag* to a new unique value;
    end
    foreach *switch* $p_j \in P_S$ *(for each switch) and the most recently received response from* p_j **do**
        if *this is the beginning of a new round of synchronization* **then**
            Delete any manager p_k that is not reachable during round *previousTag* and delete any rule p_k from p_j;
        end
        Add p_i to p_j (unless it already has been added) and overwrite the rules of p_i on p_j with *myRules(G,j,tag)*;
    end
    foreach $p_j \in P$ *reachable from* p_i*, based on the responses in replies that was received most recently* **do**
        send an update message (with *currentTag*) to p_j (if $p_j \in P_S$) and query p_j for its configuration;
    end
end
when a query response has arrived m **from** p_j **begin**
    if *replies has no space for m* **then**
        Make a C-Reset (controller reset) and overwrite *replies* with only the immediate neighbourhood, $N_c(i)$;
    end
    if *the tag in m is equal to currentTag* **then**
        delete the previous response from p_j and add m to *replies*;
    end
end
when a query response has arrived (with a *synchronizationTag***) from** p_j **begin**
    send a reply to p_j with the local topology, $N_c(i)$*, and synchronizationTag*;
end

Algorithm 1 shows our interpretation of the Renaissance algorithm, the original is provided in full in [11]. The algorithm description above provides the foundation for the implementation detailed in chapter 4. Renaissance is a self-stabilizing algorithm for the control plane of an SDN. In other words, it returns to a legal state in a bounded period of time after a transient fault, provided that no more transient faults occur in that time period.

Proof of self-stabilizing properties

In order to prove that Renaissance is self-stabilizing, we need to bound the time it takes for the algorithm to return to a legitimate state, after a transient fault occurs. In this section, we bring forward the key ideas of the proof, which can be found in detail in [11]. In order to bound the recovery time, the following three bounds needs to be provided:

- **Memory Requirements**
  Since the switches and controllers in the system have a bounded amount of memory, we may need to delete correct information at times, which will slow down the guaranteed recovery time. Therefore, we need to provide the bounds for the memory requirements.

- **Illegitimate deletions**
  For the next step, we need to bound the amount of illegitimate deletions (see below). This is important since we can only guarantee recovery in the absence of illegitimate deletions.

- **Recovery time in the absence of illegitimate deletions**
  Finally, we need to calculate the time it takes for the system to reach a legitimate state, provided there are no illegitimate deletions during that time period.

Memory requirements. A switch in our system may at most need to store rules for forwarding to every other switch, and every controller in the system. This would give us a memory requirement bounded by $(\#controllers + \#switches)$. However, since every controller may install its rules on every switch, the total memory requirement bound of a switch must be $(\#controllers(\#controllers + \#switches)$. Meanwhile, a controller in our system needs to keep track of every other controller, and every switch in the system. This leads to a total memory requirement bound of $(\#controllers + \#switches)$ for the controller.

Illegitimate deletions. Since a failing controller cannot remove its rules on a switch, the other controllers in the system need to do it for it. However, due to transient faults, a case may occur where a controller removes the rules of another non-failing controller. These "mistakes" are referred to as illegitimate deletions. Using a proof by induction, we can prove that within $c * k + 1$ rounds, there are no illegitimate deletions for at most k-distance neighbours of a controller, where c is a constant, dependant on the link capacity in the system.

The proof by induction uses the following intuition: At the k-th roundtrip, the controller discovers the nodes at distance $k + 1$.

- At $k = 1$, the controller is aware of the 1-distance topology, as only 1 roundtrip is needed for a query round.

- From k to $k + 1$, when the query to a k-distance neighbour returns, the controller is aware of the $k + 1$ distance topology

The maximum number of illegitimate deletions are therefore bounded by $c * \delta + 1$, where δ is the maximum diameter of the topology.

Bounded recovery time in absence of illegitimate deletions. Again, by using a proof by induction, we can bound the amount of time it takes for our algorithm to recover from a transient fault, provided that no illegitimate deletions take place. We prove that our algorithm returns to a correct state for at least k-distance neighbours within $(c' + 2) * k$ rounds, where c' is a constant, dependent on link capacity.

The proof by induction uses the following intuition: One roundtrip is needed for the query of a switch, and another is needed in order to install a rule on a switch.

- At $k = 1$, two roundtrips are needed to guarantee a correct state of distance 1 neighbours.

- From k to $k + 1$, the same rules as in the base case applies.

As such, the algorithm will return to a correct state within $((c' + 2) * \delta$ rounds, where δ is the maximum diameter of the topology, provided there are no illegitimate deletions during this time.

Total recovery time. Since we now know the maximum amount of time needed to recover in the absence of illegitimate deletions, as well as the maximum amount of illegitimate deletions and the memory bounds, we can now calculate the maximum recovery time of our algorithm as such: (Amount of rounds needed in absence of illegitimate deletions)*(maximum amount of illegitimate deletions)*(memory requirement bounds). Since all of these bounds are proven in the above paragraphs, we can conclude that our algorithm needs at most $(c' + 2) * \delta) * (c * \delta + 1)^*(\#switches + \#controllers + 1)$ rounds to guarantee recovery from a transient fault. This is bounded by $O(\delta^2 * \#nodes)$, where δ is the maximum diameter of the topology, and $\#nodes$ are the amount of switches and controllers in the system. Since we can prove that our algorithm returns to a legitimate state after a bounded time following a transient fault, we can call it self-stabilizing.

3

System

This chapter details the architecture and the algorithm that are used in the implementation described by this report.

3.1 Architecture

The architecture consists of a network of switches and controllers, both local and global. Each switch is connected to a local controller and is managed by a number of global controllers. The roles of the global and local controllers are described in detail below.

Local controller

In Renaissance, global controllers query switches about their local neighbourhood in order to discover the entire network topology step by step. However, responding with their local topology is not something the Open vSwitches support. Changing this would require kernel programming, which is outside the scope of the project. Therefore, we implement local controllers working as proxies for the commands of the global controllers. The local controllers functionality is described below.

Local topology discovery. The local topology continuously discovers the local topology of each switch that it is connected to. This is done so that once a global controller queries a switch for its local topology, the local controller stands ready to answer.

Failure detector. A failure detector is implemented by the local controller using the local topology discovery. If the local controller has queried a switch θ times

without getting an answer, it will consider that node crashed and remove it from every local topology it was in.

Installing rules. The local controllers handles actually installing all the rules which the global controller wants to install. Once a global controller has rules for a switch, they are sent to the local controller, which installs them on the switch.

Global Controller

Each global controller runs an instance of the Renaissance algorithm and maintains an overview of the entire network. A breakdown of the Renaissance algorithm is described in section 3.2.

3.2 Renaissance

Renaissance is a self-stabilizing SDN control plane that enables each controller to detect the network and construct a k-fault resilient flow to each other node (switch or controller) in the network. It also removes any stale information on switches, such as rules installed by crashed controllers, and keeps the network in a working state [11].

Renaissance detects the network via an iterative process, that starts with each controller detecting its set of directly connected nodes, and from there detect nodes directly connected to them and so on. This process is combined with bootstrapping communication between each controller and every other node in the system. The algorithm can be divided into tasks which the global controller carry out. Each of these tasks are explained below.

Send queries to each node

Every global controller has an idea of the network topology, hereafter called its global topology, starting with only its local neighbourhood, in which it queries each node for its local neighbourhood.

Respond to queries from other global controllers

When a global controller receives a query from another global controller it should send a response with the current local neighbourhood, which means nodes within one-hop distance.

Handle responses

Whenever a global controller receives a response it should use the local neighbourhood in the response to update its global topology.

Install rules

Each global controller installs rules on all switches in its global topology in such a way that it can communicate with every node.

Remove stale information

Every global controller change their global topologies based on the most recent query response. If a node has crashed or a link has been removed, the global topology is changed accordingly. When a global controller is removed from the global topology it is considered unreachable and all the rules installed by that controller are removed by controllers still in the network.

Synchronization rounds

All the other tasks detailed in the previous sections happen in synchronization rounds, except for the task in section 3.2. When a global controller has received responses from all nodes it queried, it will begin a new synchronization round and thus start the tasks again.

4

Implementation

This chapter delves into the the actual implementations of the system. In the first section we describe how the local and global controller interact with each other. The subsequent sections describe the local and global controller implementations which in practice are the same for both ONOS and Floodlight. However, the way ONOS works compared to Floodlight requires some different approaches when implementing Renaissance in ONOS, which is described in the last section of the chapter.

4.1 Local and Global controller interfacing

Communication between the global and local controller is done via OpenFlow $PacketIn$ and $PacketOut$ messages. For example, a controller $C1$ wishing to communicate with another controller $C2$ sends a $PacketOut$ message to switch $S1$ which in turn sends the message, that was encapsulated in the PacketOut, to switch $S2$. The $S2$ switch will then generate a $PacketIn$ message that it sends to $C2$. $C2$ can then interpret the information in the $PacketIn$ message in order to know how to react. See figure 4.1 for a visual explanation.

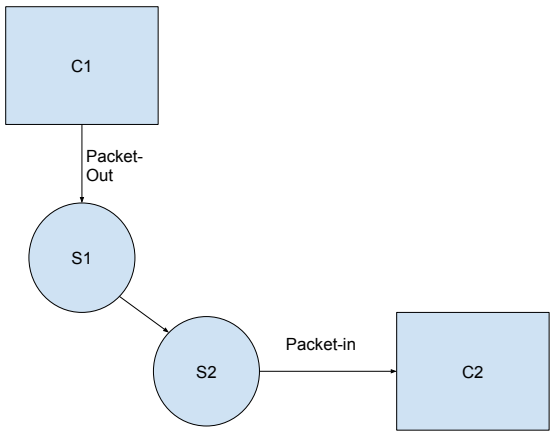


Figure 4.1: Description of how two controllers interface via OpenFlow messages.

The global controller uses three types of $PacketOut$ messages, namely $Install-rules$, $Query$ and $Query-reply$. Intuitively, $Install-rules$ are used as a command to the local controller to install rules on switches, while $Query$ messages are used to query a node about its local topology. $Query-reply$ messages are used to answer $Query$ messages from other global controllers.

The local controller has two types of $PacketOut$ messages, namely $Info$ and $Query-reply$. $Info$ messages are used to discover the local neighbourhood of a switch and also for failure detection. $Query-reply$ messages are replies to $Query$ messages from the global controller. Information on how all of these messages work together to create Renaissance functionality is described below.

4.2 Local controller

The local controller works as a proxy for the commands of the global controller. It discovers the network, detects failures and installs rules. The implementations of these tasks of the local controller are described in more detail below.

Local topology discovery

$Info$ messages are used by the local controller to detect the local neighbourhood of a switch. The local controller spawns a $PacketOut$ message at a switch A with the $Info$ message inside it. This packet is then sent out through all the ports of switch A. If another switch B sends a $PacketIn$ message to the same local controller with the $Info$ message inside it, the local controller will consider these two switches connected and create a link $A <-> B$. Once a $Query$ message is received from a global controller, the local controller replies with a $Query-reply$ containing all the links found regarding the switch.

Failure detector

This task is also carried out via the $Info$ messages, as they contain information on how many times the local controller has queried a node without getting an answer. If an $Info$ message has been sent to a node θ times without resulting in an answer, the local controller considers the node crashed and removes it from its local topology.

Installing rules

$Install - rules$ messages are messages sent from the global controller, containing forwarding rules which the local controller installs on switches in order to establish flows from each controller in the network to every other node.

Forwarding rules that are installed by the local controller make use of an OpenFlow feature called fast fail-over groups. Each group contains an ordered list of buckets where the instructions in the first buckets are used. In a fast fail-over group, each bucket also watches a port that determines if a bucket can be used. For example, if the port of the first bucket in the list goes down, then the second bucket will be used as long as the corresponding port for that bucket is up [6]. The fast fail-over groups are used to implement the concept of backup paths as every group applies to a different destination node and every bucket corresponds to a different path leading to the node.

The local controller installs rules by sending an OpenFlow message called $FlowMod$ to switches. The $FlowMod$ messages contains the current round of the global controller as a cookie, allowing for flow modification and deletion. Two layers of rules are present at the same time, so that even when updating a flow, there is no packet loss. Every time a new layer of rules is installed, the oldest layer is removed. This means that there is always a set of rules present that can forward a packet.

Network updates with tags

In a naive approach to installing rules on the switches, the local controller deletes the old layer of rules as it installs a new one. This can lead to a long recovery time in the presence of failures, a problem which may be solved by using network updates with tags. In this approach, another layer is introduced, making for a total of three layers. Each layer is installed with a unique tag, which packets in the system can match on. The tag is a combination of the synchronisation round of the algorithm, and an index of the manager (the global controller which sent the rule). The index of the manager is dependent on the amount of managers and is used to separate flows installed by separate managers. The local controller uses an $OpenFlow\ OFFlowMod$ message to install the rules of the layer, with the attribute $set - Cookie$ set to the value of the tag. In other words, the tag is stored as a property of the rule in each switch's flow table.

The packets in the system then only match on a certain tag (a new primary path) once it has been fully computed. Until then, the backup paths with the tags of the old layer are used. This is ensured by matching on the tag connected to the rules installed by the local controller. The first switch that a packet arrives at uses the OpenFlow $Set - fields$ action to modify the packet headers, applied via the $Apply - actions$ command, and created via the $Write - actions$ command. Using

network updates with tags allows for a more stable system, compared to the naive approach where the old layer of rules is deleted once a new layer is installed, which leads to a risk of multiple drops in network functionality in the presence of a link failure. The effects of the two different approaches can be seen in section 6.6.

4.3 Global controller

The global controller runs each iteration of the algorithm in [11] at regular intervals. It carries out tasks such as sending queries to nodes, responding to queries from other global controllers and handling responses. Below follows an implementation description of all the tasks of the global controller.

Send queries to each node

Every global controller has an idea of the network topology, hereafter called its global topology, starting with only its local neighbourhood, in which it queries each node for its local neighbourhood. To create this topology, global controllers send out *Query* messages to each node in its *QuerySet*, which is initially only its local topology. *Query* messages contain a label, *CurrentLabel*, which is an integer equal to the number of algorithm iterations in which each node in the *QuerySet* has responded back to the global controller. A $Query - reply$ message from the local controller connected to a queried switch, containing the local neighbourhood of the switch, lets the global controller update its *QuerySet* with new nodes step by step until it finally consists of all the other nodes in the network.

Respond to queries from other global controllers

When a global controller receives a *Query* from another global controller it should reply with the current local neighbourhood, which means nodes within one-hop distance. If a switch connected to a global controller receives a *Query* message, it sends it to the global controller, which replies with the *CurrentLabel* and its local topology in a $Query - reply$.

Handle responses

Whenever a global controller receives a response it should use the local neighbourhood in the response to update its global topology. This is done via $Query - reply$

messages from either a local controller or another global controller. The information in the $Query - reply$ is used both to add new nodes to the $QuerySet$, and to build the global topology. The controller also uses the global topology to calculate primary paths and backup paths, using Breadth-first search, resulting in a set of next hop destinations.

Install rules

Each global controller installs rules on all switches in its global topology in such a way that it can communicate with every node. Once the set of next hops which satisfy all the flows in the network has been computed, they are sent to a switch via $Install - rules$ messages, which will prompt the local controller connected to the switch to install forwarding rules on it, allowing the flows in the network.

Remove stale information

Every global controller change their global topologies based on the most recent query response. If a node has crashed or a link has been removed, the global topology is changed accordingly. When a global controller is removed from the global topology it is considered unreachable and all the rules installed by that controller are removed by controllers still in the network. This is done via a $Install - rules$ message to each switch, telling them to uninstall every rule which was installed by the IP-address of a global controller which is no longer reachable.

Synchronization rounds

All the other tasks detailed in the previous sections happen in synchronization rounds, except for the task in section 4.3. When a global controller has received responses from all nodes it queried, it will begin a new synchronization round and thus start the tasks again. This is when the $CurrentLabel$ is updated.

4.4 ONOS implementation challanges

The implementation idea used in ONOS does not differ significantly from the one used in Floodlight. However, since ONOS is a rather different system, multiple challenges came up when trying to implement the same idea in ONOS, which is why this section focuses on challenges rather than implementation details.

As mentioned in section 2.7, ONOS is a modular system, comprised of multiple modules with different functionality, working together to achieve an SDN controller. These modules are called applications, and can be activated and deactivated at run time. This modular nature allows for a user to add desirable functionality via creating custom applications. These applications can make use of all the other apps in the ONOS system, by calling objects of these other apps. Renaissance was implemented in ONOS by creating two applications, one for the local controller, and one for the global controller.

Local Controller

In ONOS, once an application is installed, it is spread over the whole cluster of controllers. Activating the app in one of the controllers will as such activate it in all of the controllers in the same cluster. This is undesirable behaviour for us since we need some controllers to be local and other to be global, and as a result we looked for ways to get around it. Initially, we sought to place the local and global controllers on different clusters in order to bypass this issue (as every controller would be both local and global otherwise). However this turned out to be a complicated solution as communication between clusters is not effortless in ONOS.

Therefore, a more simple solution was considered, namely to actually run the functionality of the app depending on the IP-address of the cluster node (controller). In essence, a controller would only run its local functionalities if it had a certain IP, and similarly only run its global functionalities if it had a certain IP. This allowed for seamless deployment of local and global controllers and also opened up for the possibility that a controller could be both global and local.

Global Controller

Just like the local controller in ONOS (see section 4.4), a controller would only use its global functionalities if it had a certain IP-address, predetermined to be a global controller IP. Another implementation challenge for the global controller was the master system. In ONOS, only one controller can be the master of a switch at a time and it is this controller that receives all $PacketIn$ messages that are generated when a switch gets a packet it does not have rules for [1]. This had to be considered when implementing the global controller, as it is meant to work as a master for the switches, but could only do that for certain switches. There is functionality in ONOS used to check which switches a controller is the master of, which was used in this case. This also lead to a need to balance out the masters, so that each master would control roughly the same amount of switches.

Mostly, implementing the global controller in ONOS was a challenge due to lack of

information on implementation development, something which was readily available for the Floodlight controller. ONOS comes with an abundance of functionality, even though finding it was challenging. However, it could prove useful for future extensions of the Renaissance implementation in ONOS.

5

Evaluation Environment

This chapter describes the environment that is used to evaluate the self-stabilizing control plane.

5.1 Setup

The Renaissance implementation in Floodlight was tested using five different networks, namely B4, Clos, Telstra, AT&T and Ebone. These topologies were chosen as they differ in both amount of nodes, diameter as well as maximum degree. More information about the topologies can be found in Appendix 1. Mininet was used as the evaluation environment to emulate the networks and traffic, since it is simple to set up and has the features needed to run our tests. OpenFlow 1.3 was the protocol which Mininet was run with, and Open vSwitch 2.5.5 was used to emulate switches. The Maximum Transfer Unit (MTU) for each link in the Mininet networks were set to 65536 bytes. The MTU was set this high due to the fact that large sets of rules needed to be communicated from the controllers to the switches, especially in the larger networks.

A parameter θ is used in the local controller for the failure-detection of nodes, and is set to 10 for B4 and Clos, and 30 for Telstra, AT&T and Ebone. The θ parameter is a threshold denoting the amount of times a local controller will query an unresponsive switch, before it is no longer considered a neighbour. Increasing θ will increase the certainty that the node is in fact down, but will also slow down the recovery time. The values of θ was set to ensure the discovery of the whole topology (as nodes in a larger topology may answer slower depending on their location in relation the the controller querying them) while being kept small for a fast failure recovery.

There is a delay in between each iteration of the algorithm for the global controller, to make sure the system is not overloaded with queries. There is also a delay in between each local topology discovery for the local controller for the same reason. These delays are both set to 500 ms as a default value, but are altered for some experiments. Any alterations are mentioned in relation to the affected graph, and

only occur in figure 6.3, 6.4 and 6.5.

The global controllers compute the paths using Breadth First Search and fast-failover OpenFlow groups are used for the backup paths in the k-resilient flows between controllers. For ping and round trip time experiments, hosts are placed such that the length of the path between them is equal to the diameter of the network.

For all measurements not presented using violin plots, 20 measurements were conducted, after which the minimum as well as the maximum value was dismissed. An average was then computed and used in the graphs. For the violin plots, all 20 measurements (except the maximum and minimum) are shown.

The evaluation was conducted on a PC running Ubuntu 16.04 LTS OS, with the Intel(R) Core(TM) i5-457OS CPU @ 2.9 GHz (4x CPU) processor and 32 GB RAM.

5.2 The test cases

There are a multitude of possible tests which can be performed on a network. The focus here, however, is to run experiments that yield measurements that allow us to evaluate Renaissance. We measure how fast the algorithm perform certain tasks, as well as how well it deals with faults. Below follows a list of test cases designed to measure the aforementioned properties. A more detailed description of each test case can be found in their respective subsections in section 6.

Network Stabilization Time

These measurements show how much time is needed in order to establish a stable network, starting from empty switch configurations. Starting from empty switch tables, each global controller needs to install a k-resilient flow to every other global controller in order for the network to be considered stable.

Network Stabilization Message Cost

These measurements show how many messages are needed in order to establish a stable network, starting from empty switch configurations. Starting from empty switch tables, each global controller needs to install a k-resilient flow to every other global controller in order for the network to be considered stable.

Re-convergence Time after topological changes

After a failure of any sort, it is important that a self stabilizing system converges back to a correct state within a bounded time period. This experiment is focused on finding those bounds for different types of failures.

Pinging in the presence of link failures

A simple way of measuring communication capabilities between hosts in a network is via Ping. This section documents the round trip times for hosts pinging each other in a network using the Renaissance algorithm. A link failure on the primary will be emulated during these measurements in order to evaluate the self-stabilizing qualities of the system.

Query Round Trip Time

As mention in previous sections, the global controller needs to query the switches in order to discover the whole topology. These measurements examines the round trip time (RTT) for a global controller querying a switch at maximal distance.

Throughput

An important measurement in a network is the throughput, in other words how much data it can transfer per second. Here, we measure the throughput of the different networks when managed by the algorithm. Link failures on the primary path will be emulated here as well.

6

Results

This chapter presents the results of the evaluation of the Renaissance algorithm as implemented using the Floodlight SDN Controller. For the evaluation setup, please refer to section 5.1. We examine the time as well as the amount of traffic required to go from an all-empty switch configuration to a stabilized network. We evaluate how well our implementation deals with recovery from failures, such as link failures, switch failures and controller failures. Ping and Iperf [3] are used to generate traffic throughout the network in order to find out what happens with the traffic in the presence of failures.

In short, we find that stabilization time depends mostly on the network diameter, but the amount of nodes is also a factor since messages from each switch needs to be processed before starting a new round. More nodes also leads to a higher network load, because more messages need to be sent, which leads to an increase in stabilization time. The implementation deals with failures and recovers just as quickly from one failure compared to multiple failures of the same kind. Ping RTT increases when a failure occurs, but drops again if the system can find a new primary path as short as the one prior to the failure. We see a drop in throughput after a link failure, but it stabilizes back to normal amounts once a new primary path is computed.

6.1 Network Stabilization Time

Starting from empty switch configurations, it is of interest to find out how fast a stable network can be established using our implementation. In order to evaluate this, we take a look at the time it takes for all global controllers in a network to discover each other as well as all the other nodes. In order to do this, rules from each global controllers have to be installed on each switch in order to create a flow from each controller to every other controller. We use three global controllers and measure the time it takes until every controller has stabilized. The expectation is that the time is linear in correlation with the network diameter.

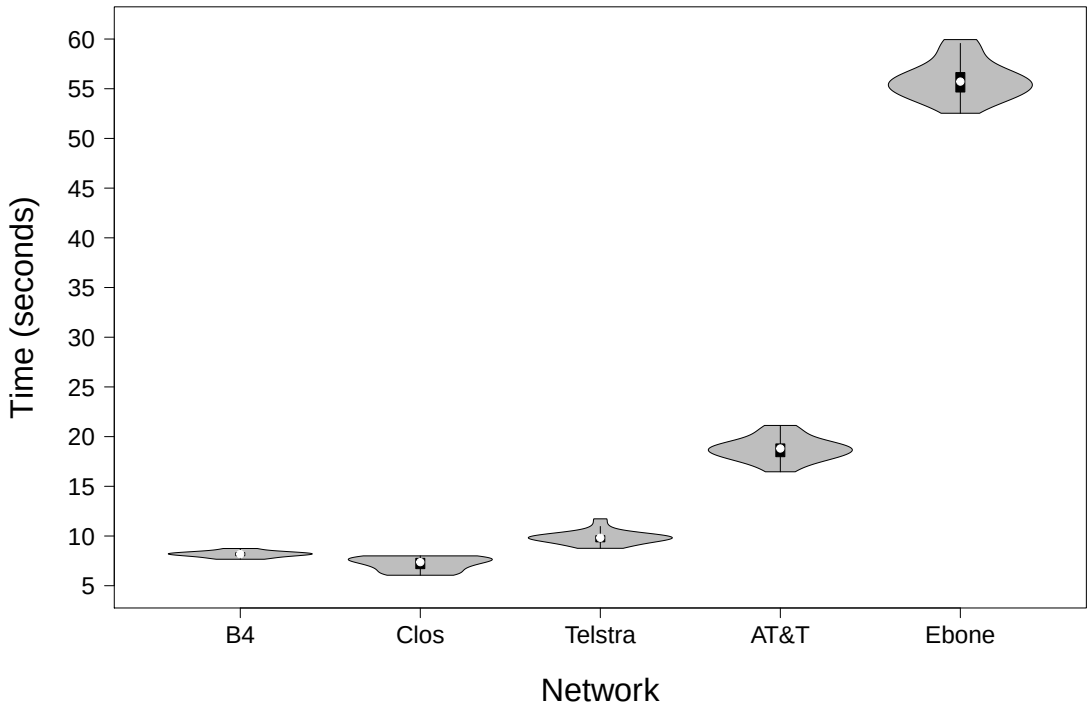


Figure 6.1: Stabilization time for all networks, using three controllers.

As can be seen in figure 6.1, the amount of time it takes to establish a stable network depends mostly on the network diameter. Clos, which is a larger network than B4 as far as switch amount goes, still stabilizes quicker due to the fact that the diameter is smaller. However, we can also see that as complexity increases, the time to stabilize increases noticeably. The EBONE network is only one hop larger than AT&T in diameter, but has more than twice the amount of switches, as well as a more complex topology with a larger amount of back-up paths. The longer time is most likely due to the fact that more local neighbourhoods of switches need to be processed in the global controller before a new query-round can begin.

The amount of time it takes for a network to stabilize may also depend on the amount of controllers used, as flows need to be installed from each controller to every other node. In the following test, we use 1-7 global controllers and measure stabilization time in the same manner as described above. We expect the stabilization time to increase as we use more controllers, as more nodes (the new controllers) needs to be discovered. However that increase should be rather small, what may cause a longer stabilization time is the fact that each new controller has to install rules on all of the nodes, which may cause a larger time difference on a larger network. Figure 6.2 shows the amount of time needed in order to stabilize the different networks, for different amount of controllers.

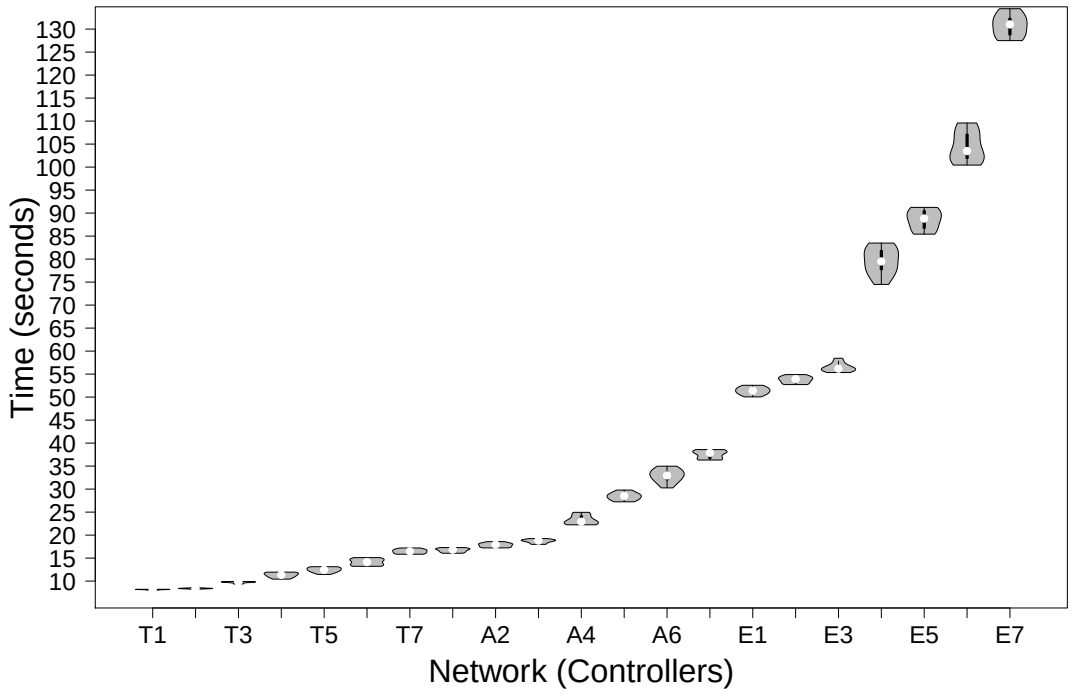


Figure 6.2: Stabilization time using multiple controllers for Telstra (T), AT&T (A) and EBONE (E). Each letter is accompanied by the number of controllers used.

Figure 6.2 denotes the time needed to reach a stable network starting from empty switch configurations. As we can see, the addition of a controller does not extend the time very much in most cases. The extra time is due to the fact that the newly added controller needs to install its rules, and needs to be discovered by the other controllers, which in turn needs to install rules to the newly added controller.

However, when considering the EBONE network, we can see that stabilization time varies quite a bit, especially from 4 controllers and upwards. The reason for this may be due to limitations of Mininet or the computer used, however this is speculative.

As mentioned in section 5.1, there is a task delay between each iteration of the algorithm. In other words, a delay in between each time the global controllers will send queries and process the answers. This delay is present in order not to overload the network, and is set to 500 ms on all above tests. However, network stabilization may be quicker if we shorten the delay. For the following experiments, we measure stabilization times for all the networks using three controllers, and changing the delay. We expect a faster stabilization time with a shorter delay.

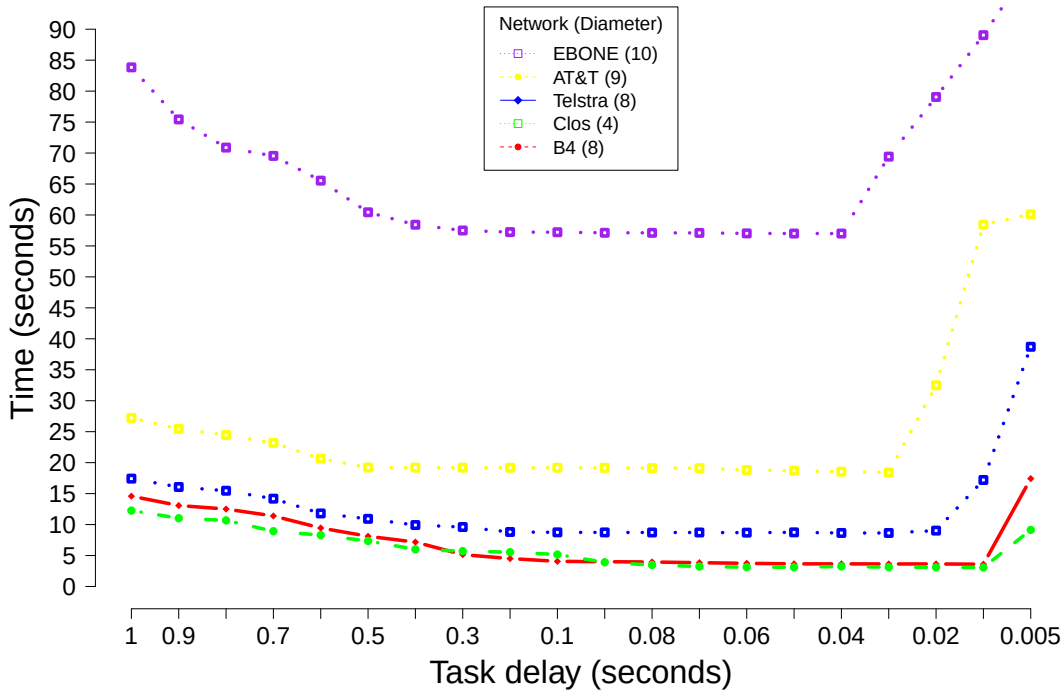


Figure 6.3: Stabilization time for the different networks, with variations in the task delay between each iteration of the algorithm.

Figure 6.3 tells us that the time required to reach network stability differs noticeably depending on the task delay in between each iteration of the algorithm. Notice how each network stabilizes faster depending on the task delay up until it reaches around 0.3 seconds, at which the stabilization time reaches a plateau. There is also a task delay at which the network stabilization time increases dramatically. This is most likely due to network congestion which stems from each global controller querying each node with almost no delay in between queries. As we can see, the larger and more complex network reaches this point at a longer query interval than the smaller ones. For a more detailed figure of the three smaller networks, see figure 6.4.

Figure 6.4 gives a clearer picture of the stabilization times depending on the task delay for the three smaller networks. We can see that the task delay plays a role in how fast a stable network can be established using the algorithm, and that only four seconds is needed in the best case. Decreasing the task delay too much will however result in a dramatic increase in discovery time as the network is congested. Figure 6.5 provides a clear picture of when it is no longer beneficial to shorten the interval, by normalizing the time towards the interval length.

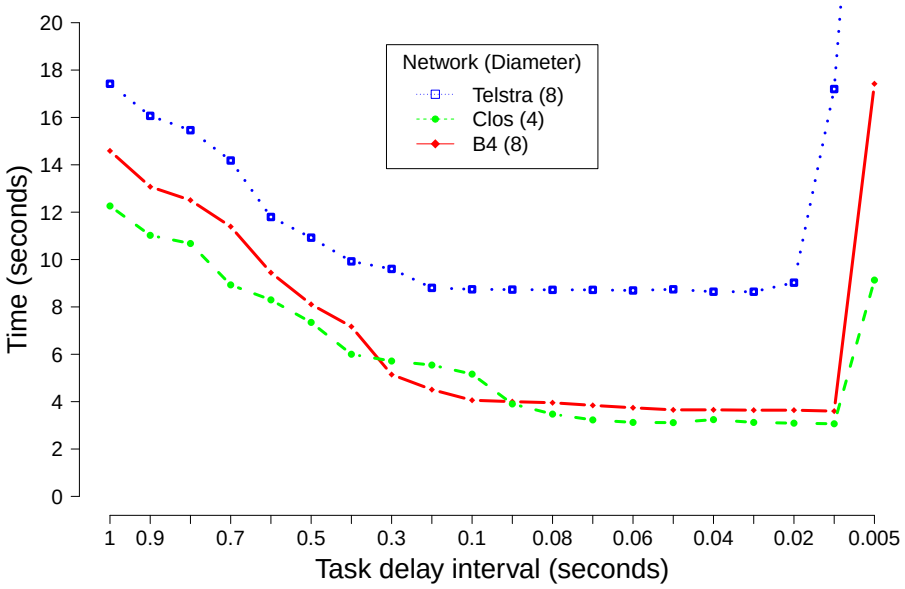


Figure 6.4: Stabilization time for the B4, Clos and Telstra, with variations in the task delay between each iteration of the algorithm.

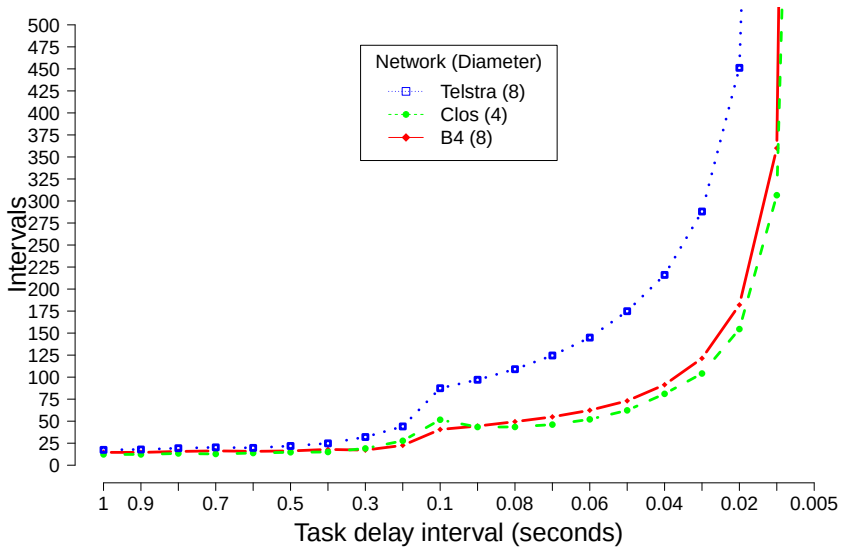


Figure 6.5: Stabilization time for the B4, Clos and Telstra, with variations in the task delay between each iteration of the algorithm. Normalized to show the amount of intervals needed for stabilization.

As we can see in figure 6.5, up until the task delay of 0.3, just as many iterations are needed to stabilize the network. Naturally, running as many iterations but at a higher pace leads to a faster stabilization time. We can see that after 0.3, the amount of iterations needed increases dramatically, making 0.3 the sweet spot and the best interval to use for fast stabilization time.

6.2 Network Stabilization Message Cost

In this section, we present measurements of how many messages are needed in order to establish a stable network, starting from an empty switch configuration. A message is defined as whenever a global controller sends a query. The amount of messages shown was measured after each controller reached a stable state, upon which it reported the amount of queries it had sent. These amounts were then summed up in order to obtain the amounts shown in the graph. We expect the message complexity to depend both on the diameter but also on the amount of nodes, as more nodes are queried.

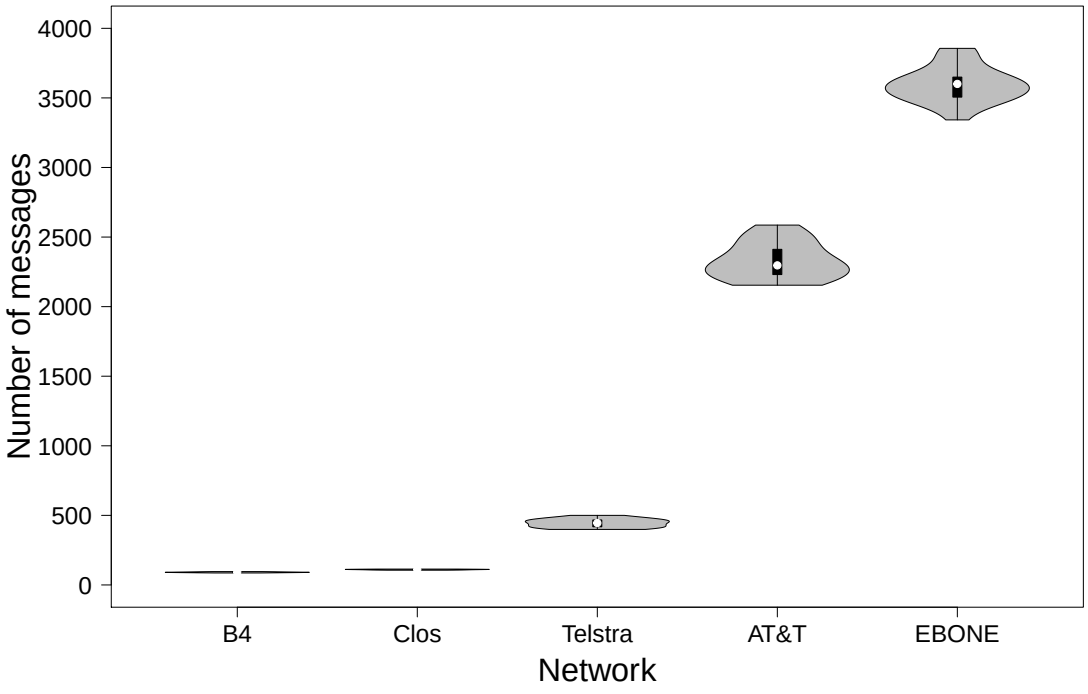


Figure 6.6: Amount of messages needed for each network to reach a stable state.

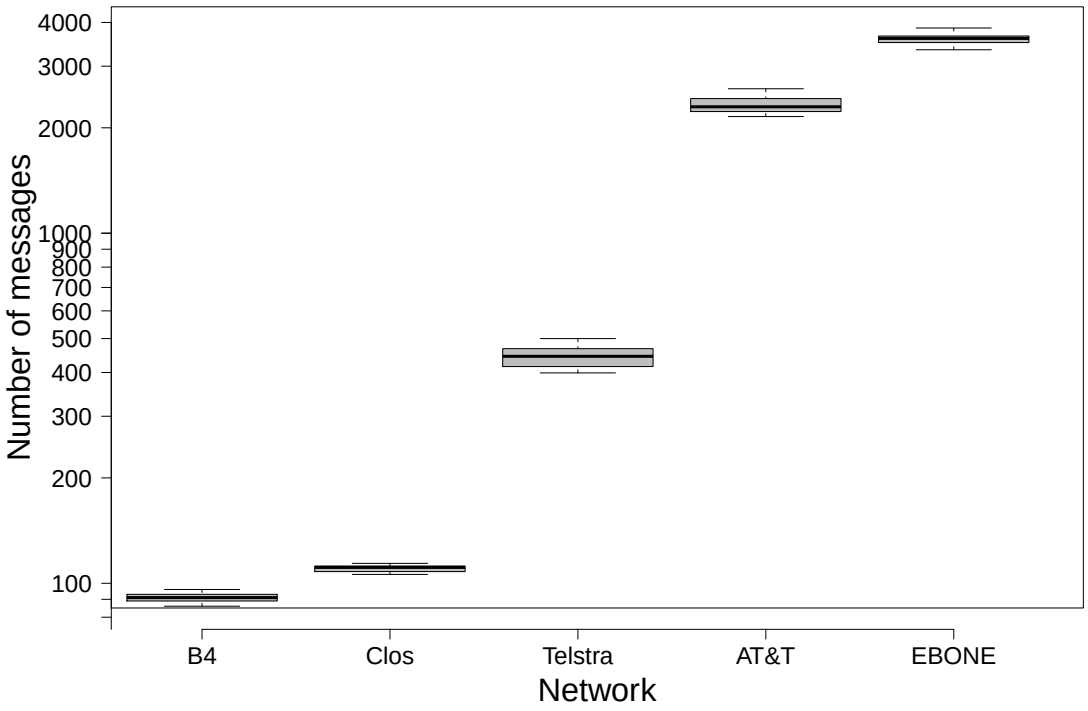


Figure 6.7: Amount of messages needed for each network to reach a stable state (log scale).

In figure 6.6 we can see that B4 and Clos need very few messages in order for each global controller to reach a stable state. B4 and Clos also vary very little in the amount of messages needed from experiment to experiment, which can be due to the simplicity of the topologies. As the amount of nodes grows, we can see a large increase in number of messages needed to stabilize the network. As expected, more nodes require many more messages. But after around 3700 messages, even EBONE, with its 172 switches, is stabilized. For this graph in a logarithmic scale, see figure 6.7.

In order to find out how much the amount of nodes matter in this experiment, we ran the same experiment but scaled the results to the amount of nodes in each graph. We expect to see a much smaller difference in amount of messages, as the amount of nodes are accounted for. The results can be seen in figure 6.8.

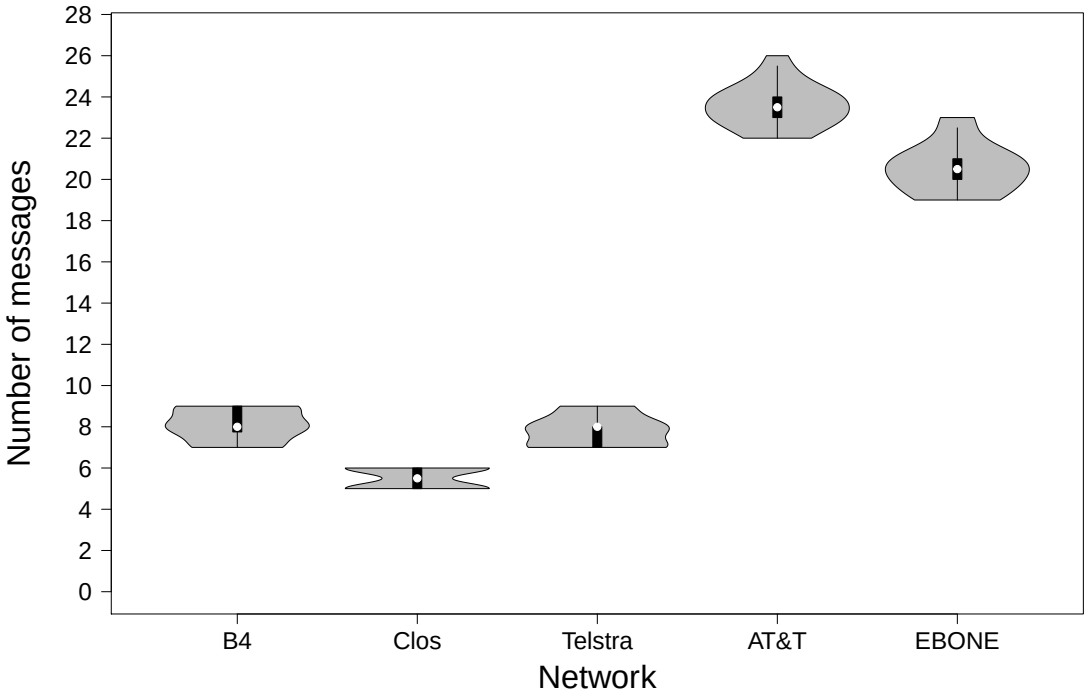


Figure 6.8: Amount of messages needed per node for each network to reach a stable state.

Figure 6.8 tells us that the amount of nodes in a network is important for how many messages is needed in order to reach a stable state. Note that Clos requires more messages to stabilize (see figure 6.6) compared to B4, due to the fact that it has more nodes, but requires less messages per node due to the smaller diameter. However, it is not the only factor. AT&T and EBONE takes almost twice the amount of messages to stabilize compared to the other networks. This could be due to the structural properties of these networks, leading to issues such as congestion.

6.3 Re-convergence Time after topological changes

After a failure of any sort, it is important that a self stabilizing system converges back to a correct state within a bounded time period. This section is focused on finding that time for different types of failures. In order to evaluate each part of a network, we consider fail-stop failures of controllers, permanent switch failures and permanent link failures in these experiments. These experiments are performed by letting the algorithm reach a legitimate stable state, and then applying a failure in order to measure the re-convergence time. In these experiments, B4, Clos and

Telstra are run with 3 global controllers, while AT&T and EBONE are ran with 7.

Fail-stop failures

In order to measure the re-convergence time after the occurrence of a controller failure, we let the network stabilize, after which we disconnect a single controller, chosen at random. We then measure the time it takes in order for the network to re-converge. What this entails is that each other controller should stop considering the failed controller and delete all of its rules and stop querying it. We expect the resulting time to depend on the size of the networks.

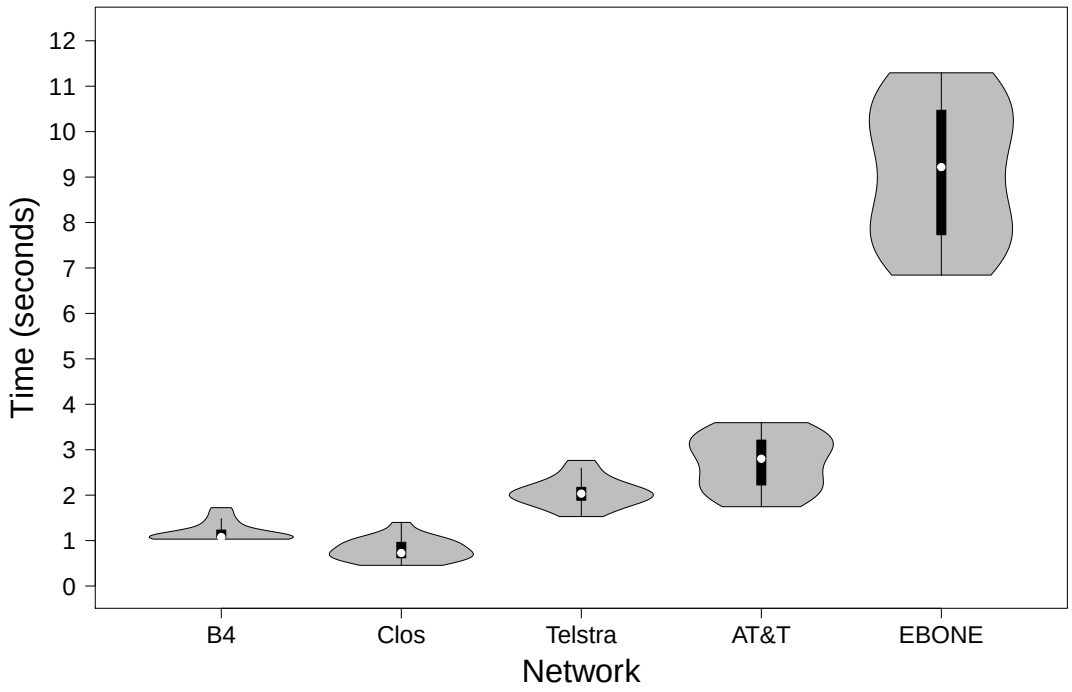


Figure 6.9: Re-convergence time after a fail-stop failure for a global controller.

As we can see in figure 6.9, the algorithm is quick to stabilize each network other than EBONE. This is to be expected as EBONE differs greatly in size compared to all the other networks, but also in structural properties. Other than that, the re-convergence time seems to be rather linear to the diameter of the networks, however the amount of nodes is also a factor.

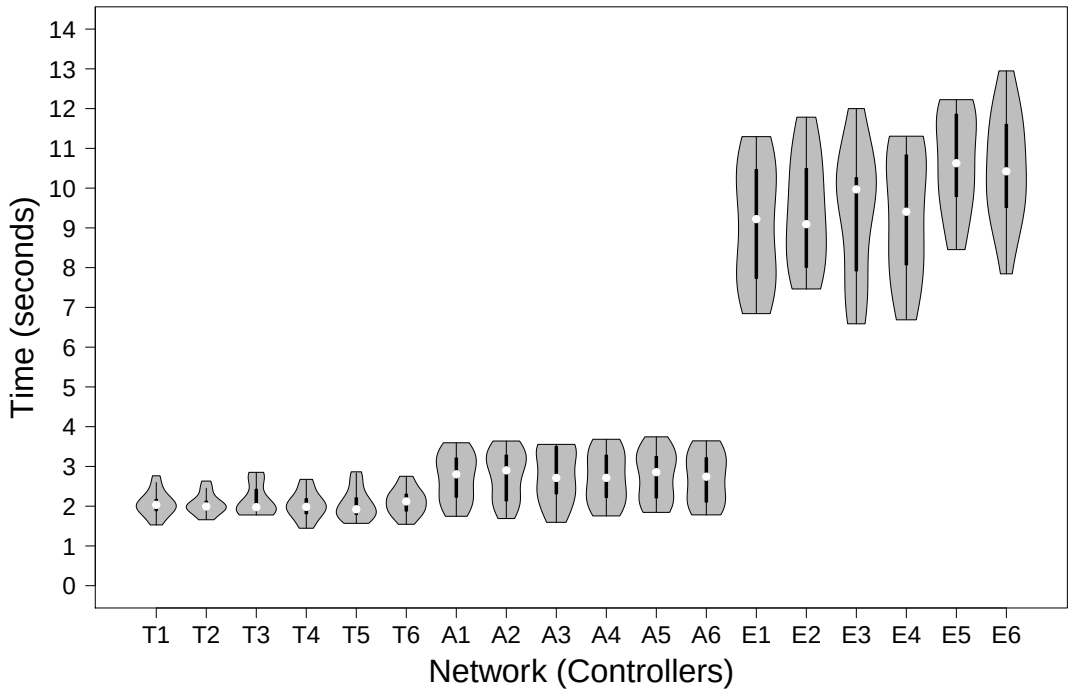


Figure 6.10: Re-convergence time after a fail-stop failure for 1-6 controllers in Telstra (T), AT&T (A) and EBONE (E). Each letter is accompanied by the number of failed controllers.

It is interesting to see if the amount of controllers failing simultaneously is a factor for the re-convergence time. To evaluate this, we let the system reach a legitimate state, and then emulate controller failures. The failing controllers were initially chosen at random, and the same controllers were set to fail for each test on the respective networks. We expect it to not differ much in time, as the controllers fail simultaneously and more controllers result in a mostly insignificant network traffic increase. The results of these measurements can be seen in figure 6.10.

As can be seen in figure 6.10, the amount of controllers failing simultaneously has no significant effect on the re-convergence time. There is a spread in time for the different networks, but this is most likely due to the fact that a test does not guarantee the same result each time on a network. Since all the controllers fail at the same time, the remaining controllers should notice this failure at roughly the same time, and re-converge.

Permanent switch failures

We also want to measure how long it takes to stabilize from permanent switch failures. For this experiment, the network is allowed to reach a stable state, after which a random switch is disconnected. We then measure re-convergence time by looking at the time it takes for each global controller to stop considering the switch a part of the network. We expect this to be proportional to the size of the networks, although with a wide spread of results as we disconnect a random node each time.

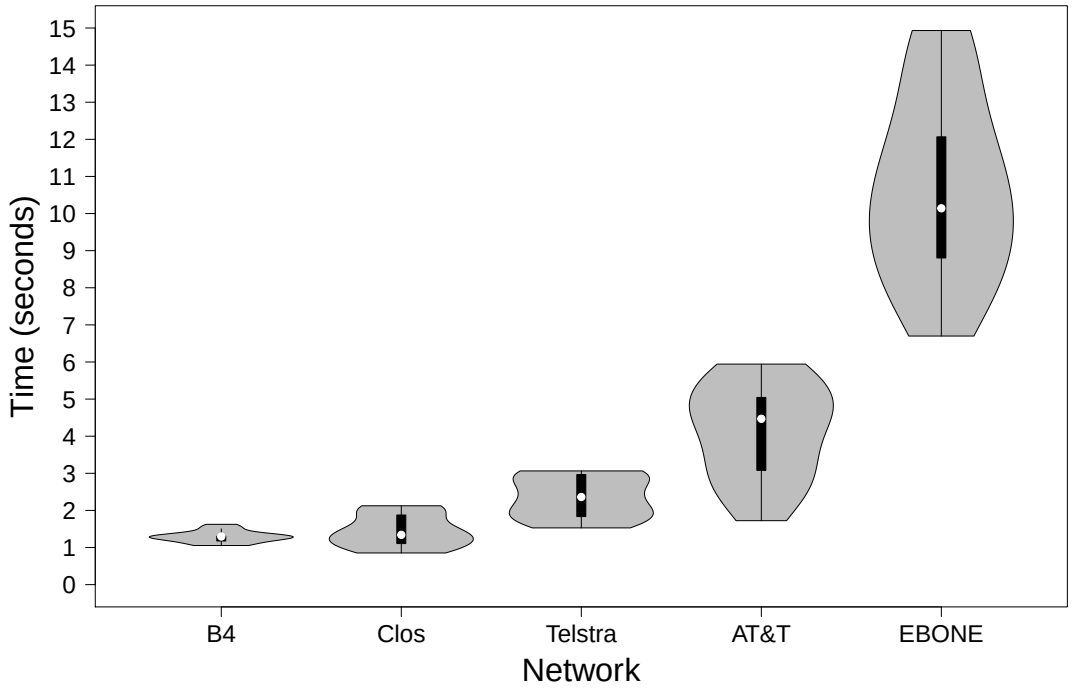


Figure 6.11: Re-convergence time after a permanent switch failure.

Figure 6.11 shows us that the re-convergence after a permanent switch failure increases with the size of the network. The violin plots here show a large spread of the time it takes to stabilize the network, especially in the EBONE network. This is most likely due to the fact that a random switch is disconnected for each measurement, and the relation of the switch and where each global controller is connected to the network may play a part in re-convergence time. Once again we see that EBONE sticks out in that it takes a lot longer to re-converge, while the other networks are rather linear. We can also see that the spread of the re-convergence time increases with the number of nodes in the network.

Permanent link failures

We also want to find out how our implementation deals with permanent link failures. These experiments are conducted after the network has reached a stable state, after which a link is disconnected and re-convergence time is measured. We expect the re-convergence time to increase with the number of nodes in the network since this leads to heavier traffic on the network.

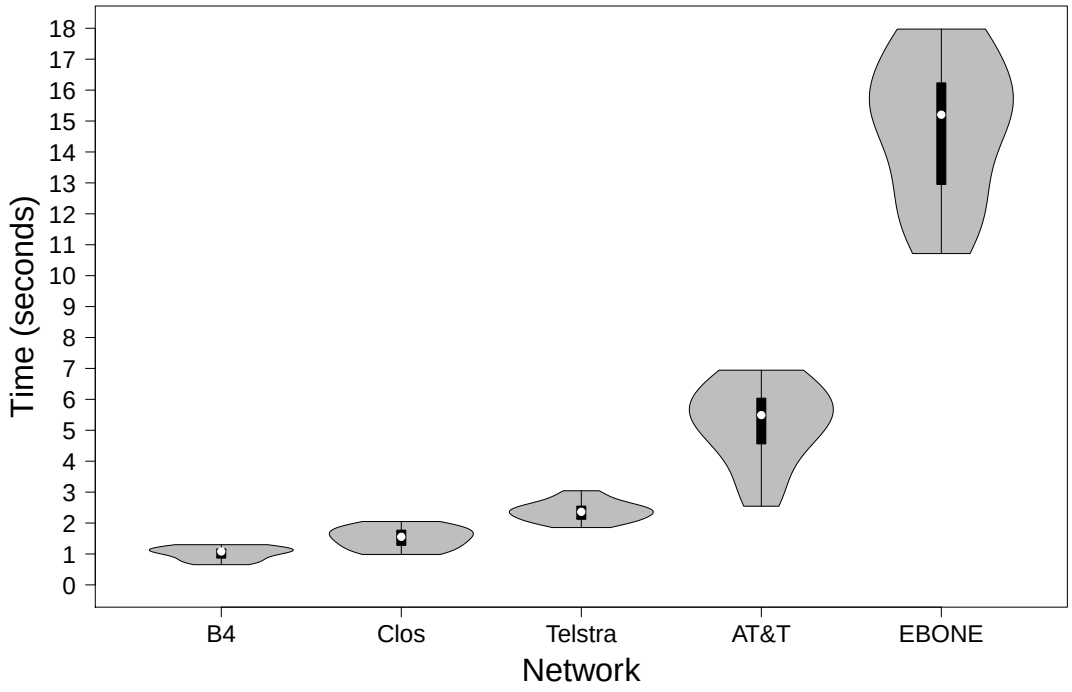


Figure 6.12: Re-convergence time after a permanent link failure.

Figure 6.12 paints a similar picture compared to figure 6.11, with the exception that the larger network takes a longer time to re-converge. This can be due to the fact that once a link has failed, communication with some nodes may take longer as a longer path needs to be chosen. When the amount of nodes in the network increase, so does the chances of a link failure affecting the path from a controller to multiple nodes. Once again, we also see that EBONE takes quite a while longer to re-converge compared to the other networks.

It is also interesting to see what happens if multiple links fail simultaneously and how that can affect the re-convergence time. For the following test, we emulate link failures on multiple links in the networks, and measure stabilization time. We

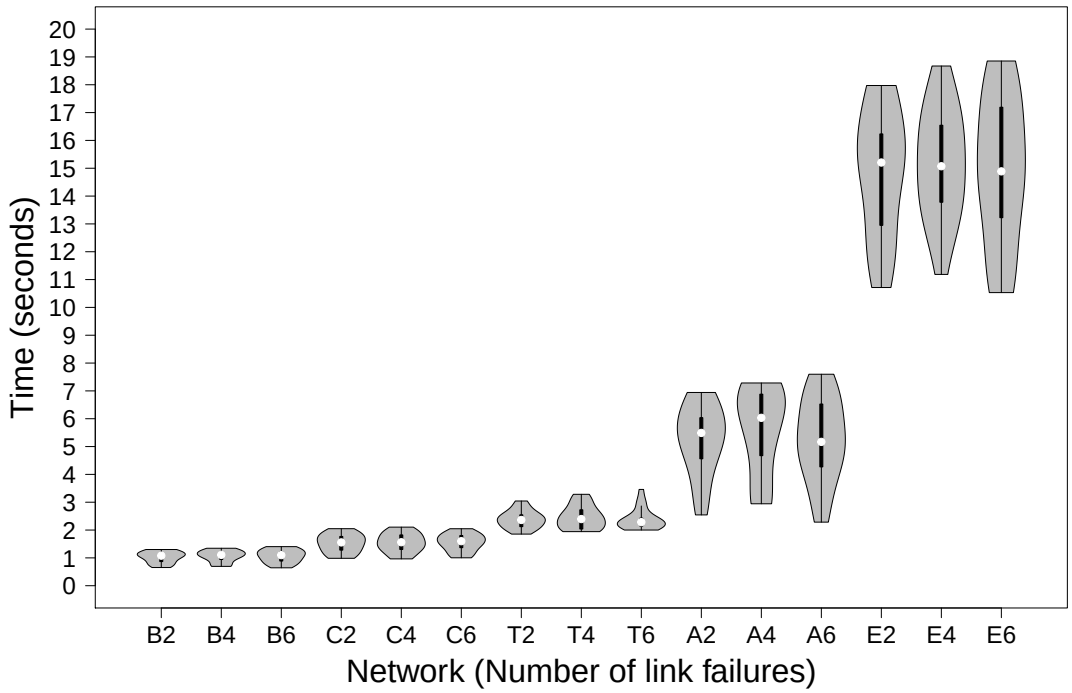


Figure 6.13: Re-convergence time after multiple permanent link failures for B4 (B), Clos (C), Telstra (T), AT&T (A) and Ebone (E). Each letter is accompanied by the number of permanent link failures.

expect no difference in re-convergence time as the links fail at the same time. The results can be seen in figure 6.13.

As expected, and similar to the results of multiple failing controllers seen in figure 6.10, the amount of links failing at the same time does not affect the re-convergence time significantly. There are some variations in the results which possibly is the results of randomness as the variations are rather small. Regardless, it is evident that the algorithm can deal with multiple link failures in all the networks.

6.4 Ping statistics

A simple way of testing communication capabilities between hosts in a network is via Ping. This section documents the round trip times for hosts pinging each other in a network using the Renaissance algorithm. In this experiment, a host in the network will ping another host for a duration of 40 seconds. After 20 seconds, a permanent link failure is injected in order to disrupt the primary path between the

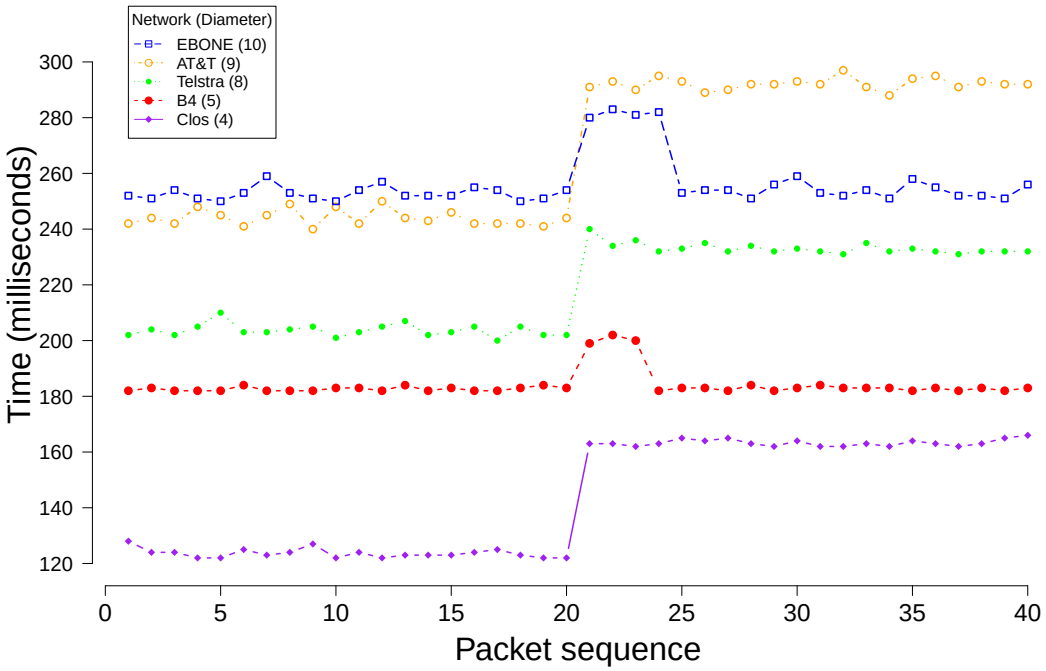


Figure 6.14: Ping statistics for two hosts pinging each other in the different networks. After 20 seconds, the primary path is obstructed by a permanent link failure. A link delay of 10 ms is present.

two hosts. The network then has to re-converge and compute a new primary path. The time for our algorithm to re-converge is quite fast, however it is still too slow not to cause traffic interruption, which is unacceptable. For this reason, OpenFlow fast-failover groups are used for the back-up paths which are used during the re-convergence. A link delay of 10 milliseconds has been added to each link in each network, for a more detailed experiment. The hosts pinging each other are placed in such a manner that the distance between them equals the network diameter, and the link failure is emulated as close to the middle of the primary path as possible. The time noted is the RTT for a ping, measured in milliseconds. We expect to see a higher RTT for a while after the failure, after which it either finds a new primary path which is as long as the previous one, or chooses the back up path as the new primary.

As we can see in figure 6.14, the RTT for a ping is greater depending on the network diameter. This is due to the fact that the two hosts pinging each other are placed so that the distance between them is equal to the diameter. As we can see, pinging is faster in Clos even though B4 is a smaller network, due to the fact that Clos has a smaller diameter. After 20 seconds, when the permanent link failure is emulated,

we notice an increase in RTT, which is due to the fact that a back-up path has to be used during re-convergence. As we can see in Clos, Telstra and AT&T, there is no new primary path with the same distance as the disrupted one, which is why the previous back up path becomes the new primary path, and the RTT is never decreased from this value. However, in B4 and EBONE, a new primary path with the same length as the old one is found and used, resulting in a RTT similar to the one before the link failure (see figure 6.15 - figure 6.18 for an explanation of primary and back-up paths). Notice also how EBONE takes a bit longer to compute the new primary path and re-converge, which is expected as the network is larger.

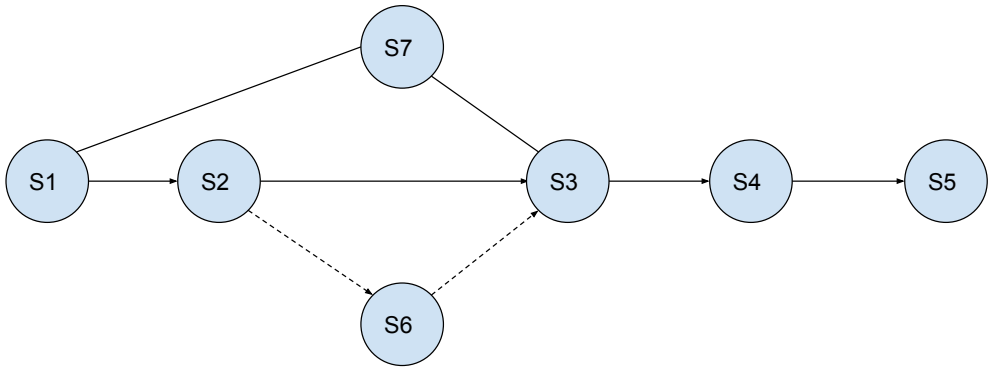


Figure 6.15: A primary path (line-arrowed) that is as long as the back up path (lined). If S2->S3 fails, the back up path S1->S7->S3 will become the new primary path and will be as long as the previous primary path.

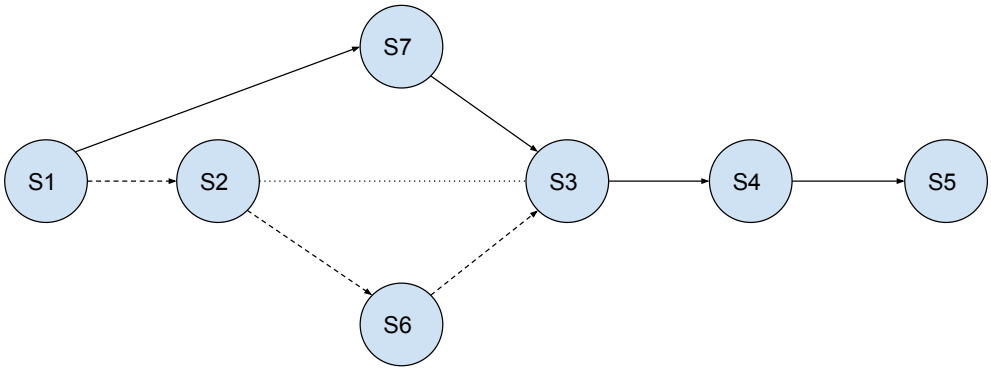


Figure 6.16: S2->S3 (dashed) has failed, and a new, equally long primary path has been computed (line-arrowed). The dash-arrowed path may now be used as a back-up path.

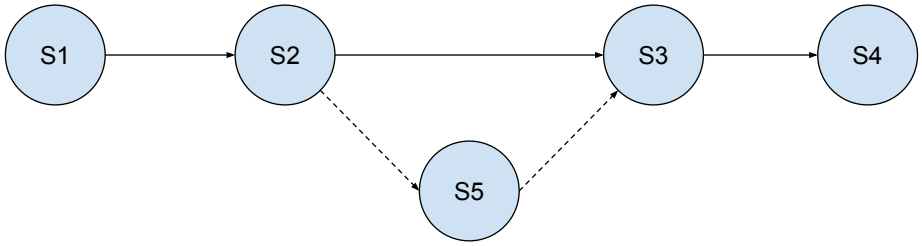


Figure 6.17: A primary path (line-arrowed) which is one hop shorter than the back up path (dash-arrowed)

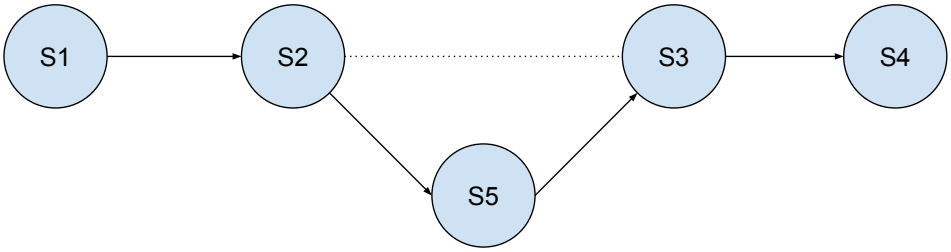


Figure 6.18: An example of a new primary path once S2->S3 fails, that is one hop longer than the previous back up path.

6.5 Query Time Complexity

As mentioned in previous sections, the global controller needs to query the switches in order to discover the whole topology. This section examines the round trip time for a global controller querying a switch at maximum distance. The experiments are conducted by measuring the RTT for a query of a global controller to the switch at maximum distance from the controller. The test is conducted once the system has reached a legitimate state and the time is measured in milliseconds. We expect results to be linear according to the diameter.

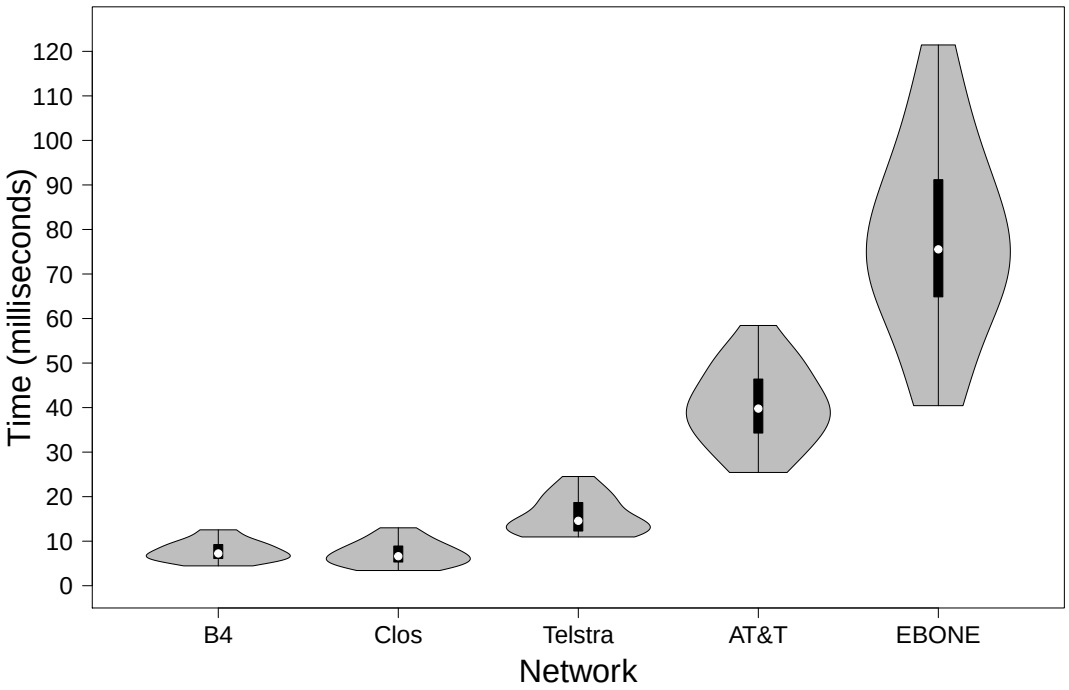


Figure 6.19: RTT for a query to the switch in the network that is at maximal distance from the querying controller.

As expected, the RTT for a query is proportional to the diameter of the network. Figure 6.19 also tells us that the spread of the RTT is rather large, which can be due to the fact that the network load differs from measurement to measurement, depending on where the different controllers are in their algorithm execution. Since the system is asynchronous, this can differ from controller to controller.

6.6 Throughput

This section documents the throughput in the presence of a link failure. We measure the throughput between two controllers in the network, placed at maximum distance from each other, and emulate a permanent link failure on the primary path. Iperf [3] was used to generate the traffic between the hosts. We measure throughput during a 30 second period, where the throughput is reported once each second. After 10 seconds, the permanent link failure disrupts the primary path, after which the algorithm has to recover and compute a new primary path. The maximum throughput per link is set to 1000 MBits/second. We expect there to be a drop in throughput when the link failure happens as a backup path has to be used while a

new primary path is being calculated.

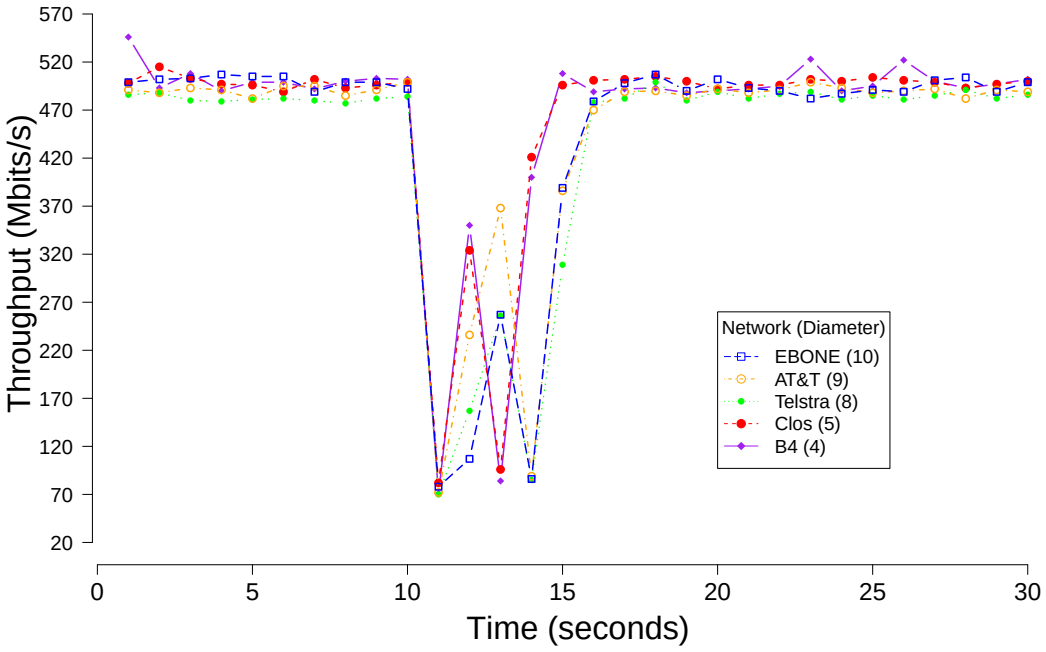


Figure 6.20: Throughput for the different networks over a 30 second period. A permanent link failure is emulated after 10 seconds.

In figure 6.20, we can see that after the link failure, there is a drastic drop in throughput. It picks back up again after a second, when the traffic is re-routed to a back-up path. However there is a second drop, as the network converges and computes a new primary path. After this new primary path is computed and starts being used, the throughput goes back to normal. This is not really desirable behaviour, as the second drop is somewhat unnecessary. With a less naive approach, using network updates with tags (explained below), we can limit the throughput drop to one occurrence, as seen in figure 6.21.

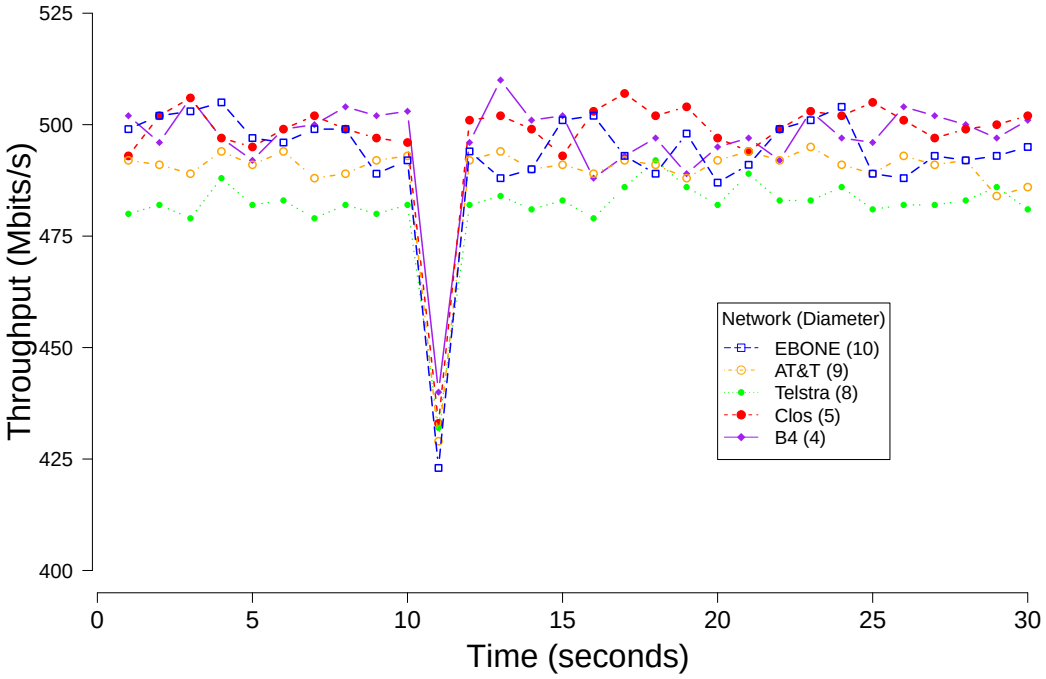


Figure 6.21: Throughput for the different networks over a 30 second period. A permanent link failure is emulated after 10 seconds. Network updates with tags are used.

Figure 6.21 shows us only a small drop in throughput after the link failure, from around 500 to around 430 MBits/s. Also, we can see that only one drop occurs, compared to figure 6.20. This is due to the fact that we use network update with tags when conducting the experiments in 6.21. More information on how the network updates with tags were implemented can be found in section 4.2. This method is to be preferred as only one drop in throughput occurs, leading to a more stable network.

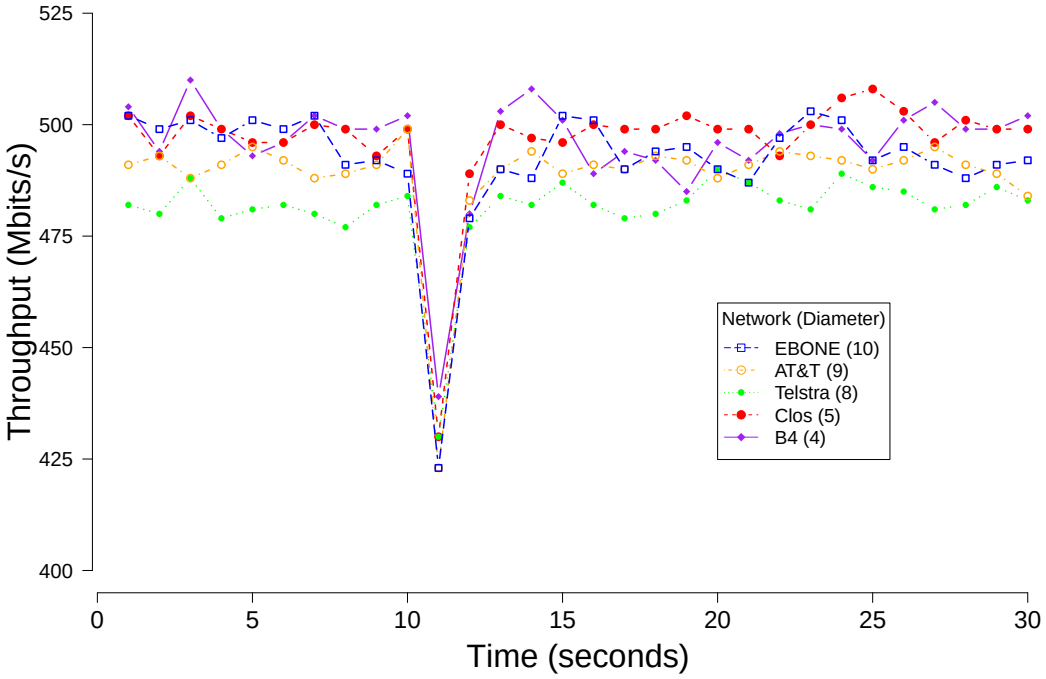


Figure 6.22: Throughput for the different networks over a 30 second period. A permanent link failure is emulated after 10 seconds. No recovery used, back-up paths only.

In figure 6.22, we can see another throughput experiment, similar to the one in figure 6.21, but this time without using any recovery. This means that the traffic will simply be re-routed through the back-up path once a link has been brought down. We expect the throughput post link failure to depend on the quality of the backup path and we still expect a temporary drop and that is what the graph shows. As a result this graph is very similar to figure 6.21.

In order to find out how similar these results actually are, we take a look at the correlation coefficient for each individual network in table 6.1.

Network	Correlation coefficient
B4	0.9354860231
Clos	0.9433182591
Telstra	0.9002880687
AT&T	0.9662039621
EBONE	0.9505491725

Table 6.1: Correlation coefficient of the average throughput for figure 6.21 and figure 6.22.

As can be seen in table 6.1, there is a strong correlation between using network updates with tags, and simply not computing new primary paths. In other words, computing new primary paths has no significant effect on throughput if network updates with tags are used, leaving it to be the preferred method. It is also important to remember that the throughput shown in figure 6.20 - figure 6.22 is an average. For a clearer picture of how much the throughput ranges from one measurement to another, see figure 6.23 to figure 6.27.

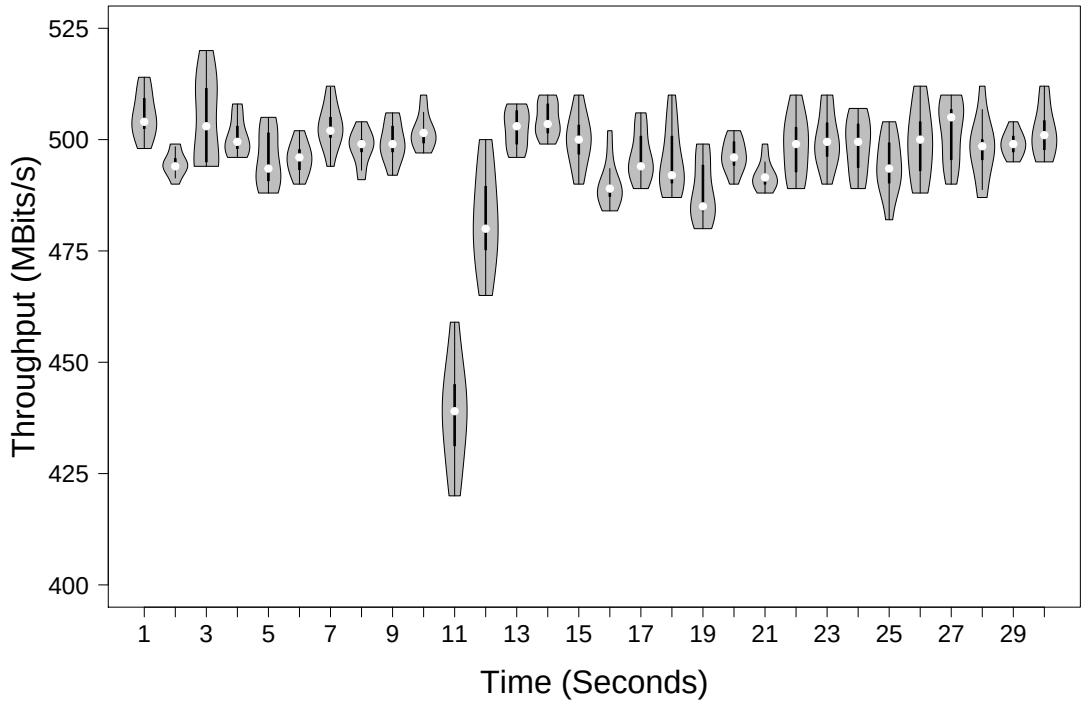


Figure 6.23: Throughput range for each second, B4 network.

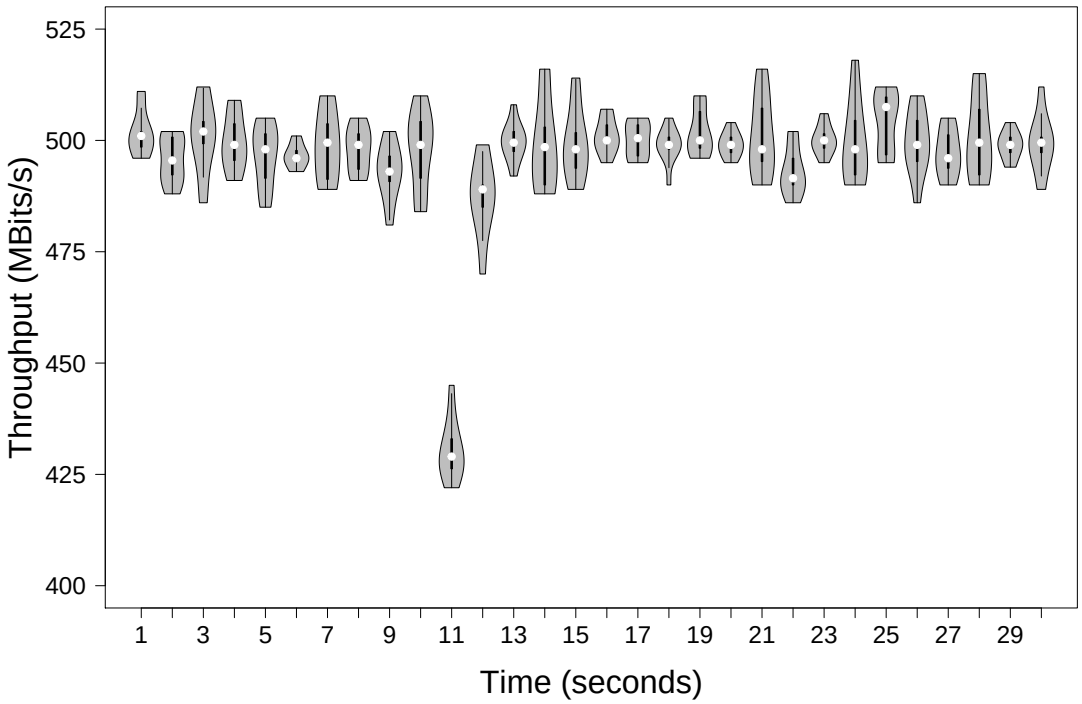


Figure 6.24: Throughput range for each second, Clos network.

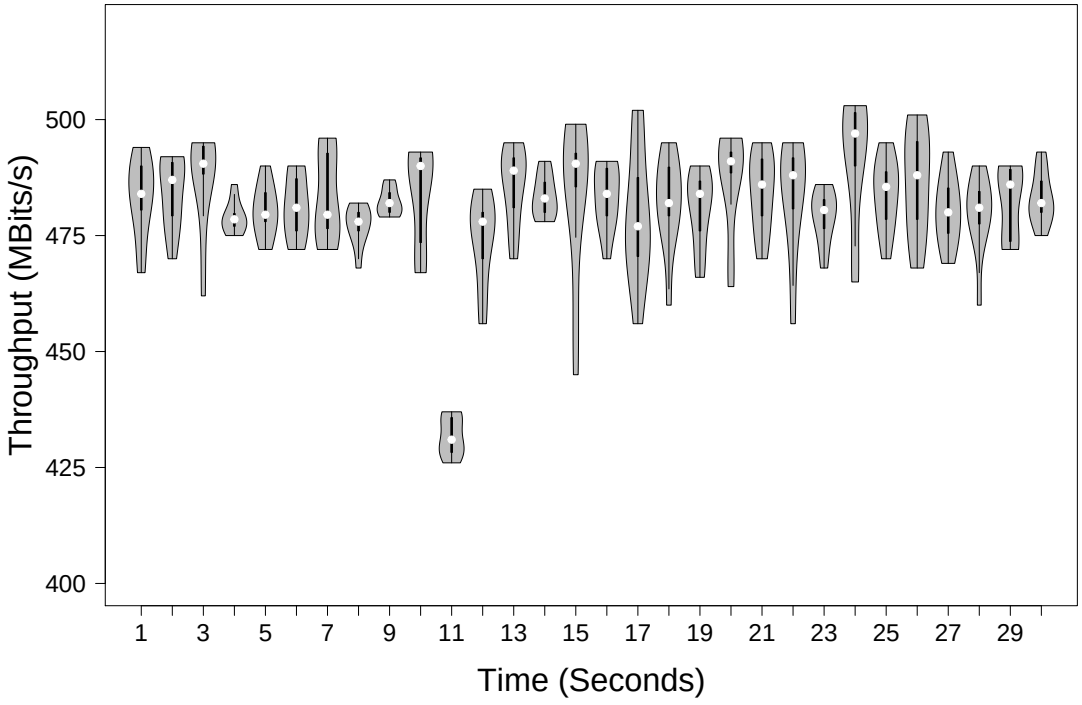


Figure 6.25: Throughput range for each second, Telstra network.

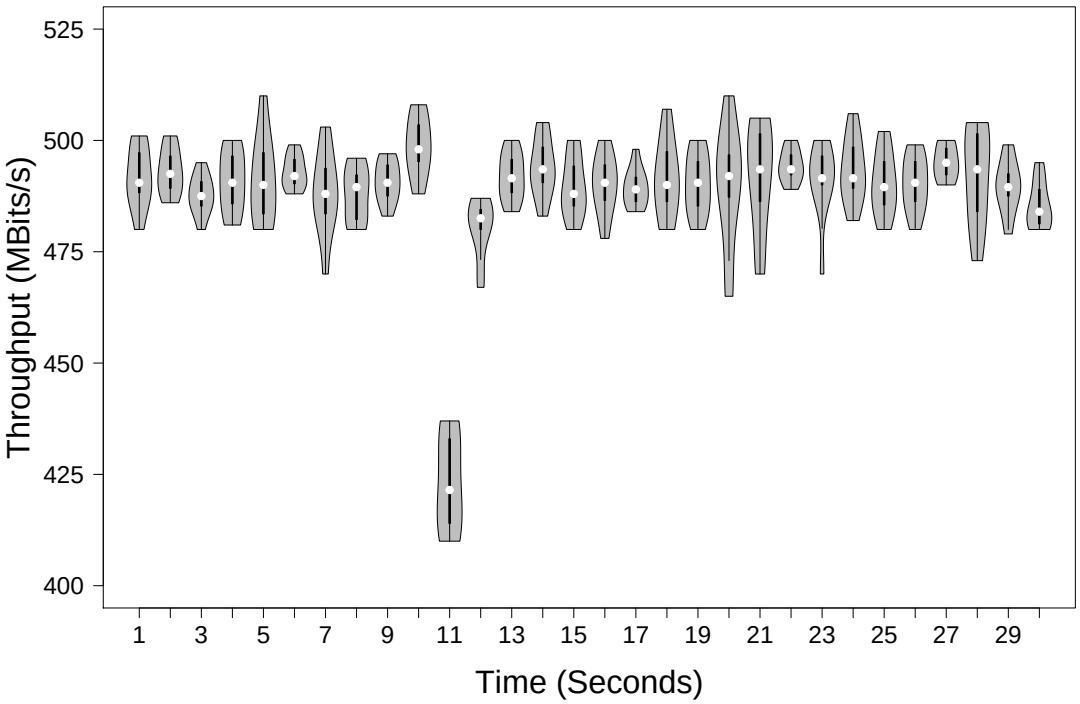


Figure 6.26: Throughput range for each second, AT&T network.

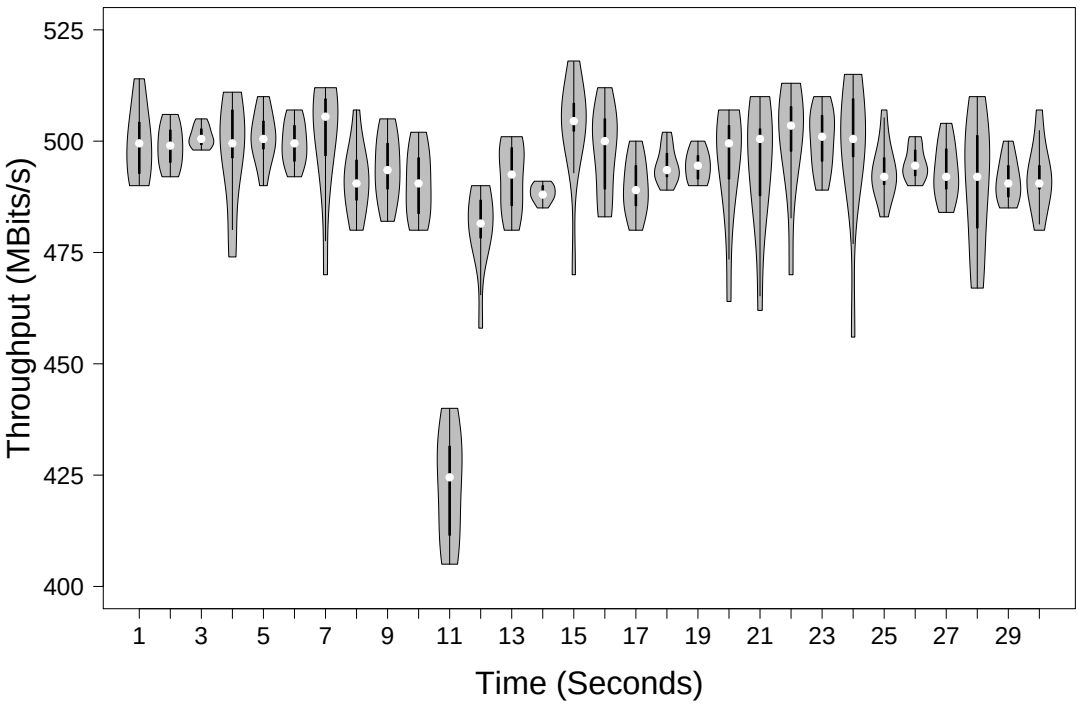


Figure 6.27: Throughput range for each second, EBONE network.

As we can see in figure 6.23 - figure 6.27, the throughput ranges quite a bit from experiment to experiment. This is most likely due to the different loads on the system, depending on where the global controllers are in their execution of the algorithm.

In order to find out the cause of these drops in throughput, we ran the experiments again. This time we looked at the packets captured in Wireshark to find out the cause of the drops, the results of which can be seen in figure 6.28 - figure 6.29

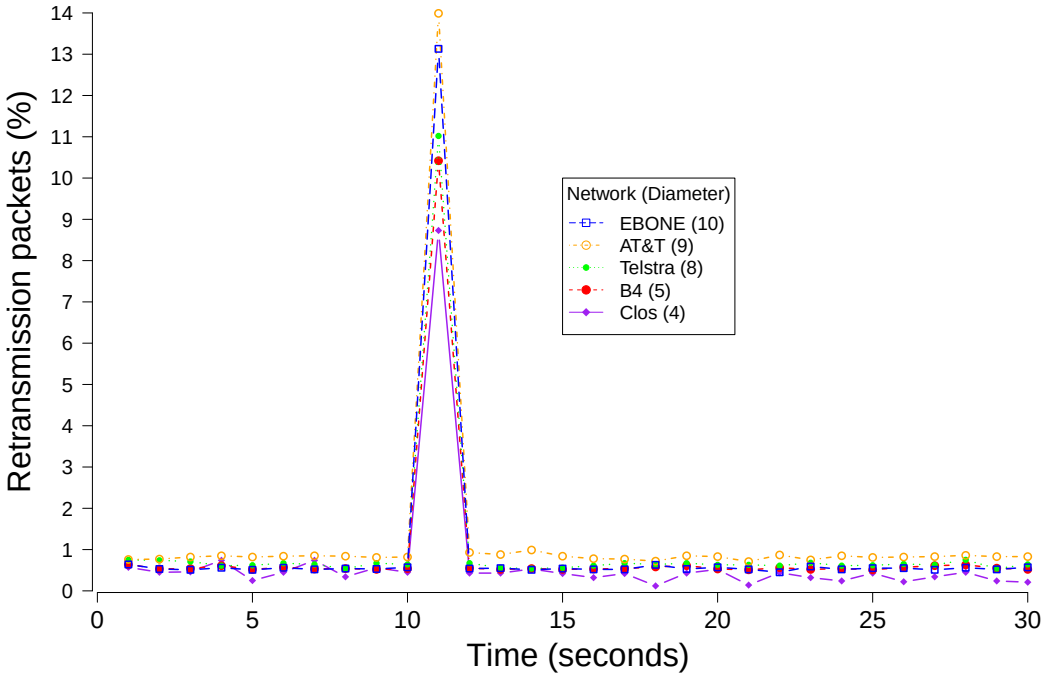


Figure 6.28: Percentage of re-transmissions at the time of a link failure.

As we can see in figure 6.28, the amount of re-transmissions increase heavily at the time of the link failure, which can explain the drop in throughput.

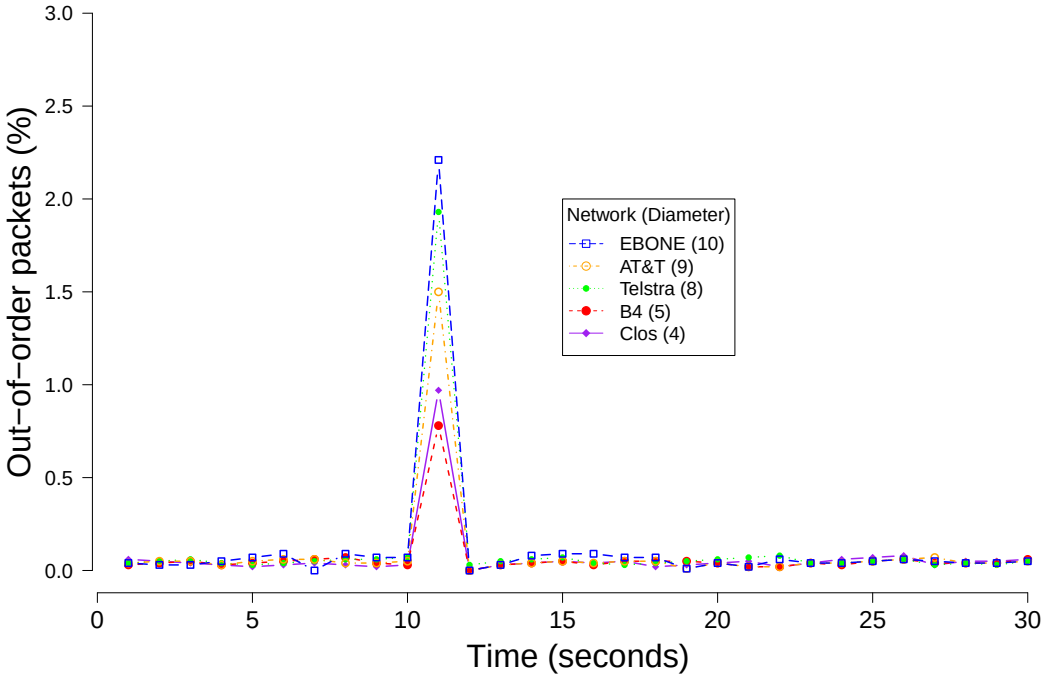


Figure 6.29: Percentage of Out-of-order packets at the time of a link failure.

Figure 6.29 shows us that we also got an increase in out-of-order packets at the time of the link failure. Another explanation for the drops in throughput.

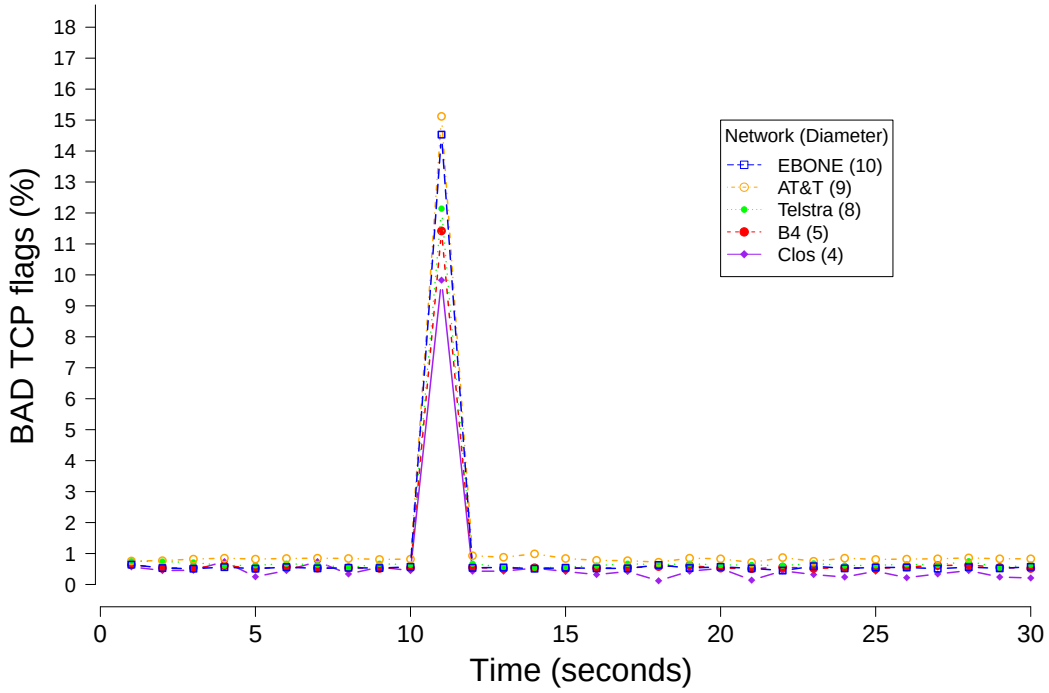


Figure 6.30: Percentage of BAD TCP flags at the time of a link failure.

As is evident in figure 6.30, there is a clear correlation between the throughput drops and the increase in BAD TCP packets. After a link failure is emulated on the tenth second, the percentage of BAD TCP rises from around zero to 10-15% depending on the network. These packets are in the form of TCP re-transmissions as well as Out-of-order packets, which is no surprise since the link failure would cause traffic to be disrupted. Packet loss due to link failure would explain both the drop in throughput and these BAD TCP flags. Also, the TCP congestion protocol (TCP Reno) may explain the drop in throughput, which is out of our hands. We can see that our algorithm stabilizes and goes back to normal after a certain period of time.

7

Conclusion

The benefits of decoupling the data plane and the control plan has received a lot of attention in academia, while the matter of how to do it has received less attention. In this report, we present and evaluate a prototype implementation of the Renaissance algorithm [11], aiming to do just that. A proof-of-concept is also implemented in the ONOS SDN controller.

The self-stabilizing properties of Renaissance is shown in the evaluation as the results display the ability to recover from switch, link and controller failures. The implementation recovers from these faults within seconds, and is not affected by the amount of controllers, links or switches failing simultaneously.

We conclude that the networks stabilization time depends mostly on the diameter of the network, but also increases with a high number of switches. The implementation provides a bootstrapping time of under 25 seconds for networks up to roughly 100 nodes.

Another important conclusion is how the network continues to function in the presence of failures. Pings continue to reach the target hosts and throughput is maintained without significant drops during a failure. Recovery here is also completed in a matter of seconds.

We believe that this first prototype implementation of Renaissance [11] shows promising results and proves the algorithm to be self-stabilizing. We are able to show that the algorithm provides a bootstrapping time dependant on the diameter, even in the presence of any type of faults. The abilities of the implementation is shown in a wide range of networks, differing in complexity, size and diameter. The different networks in combination with multiple types of faults point to a an self-stabilizing algorithm attractive for Software Defined Networks.

Lessons learned

During the course of this masters thesis we have learned a lot about about Software Defined Networking, its perks and challenges. We have also learned to handle the API of a large project such as ONOS, and how many modules may work together to achieve a goal. We learned to use OpenFlow to communicate in a SDN network, and how to build python scripts in Mininet to emulate our topologies.

Final thoughts

In conclusion, we have seen that prototype of Renaissance is self-stabilizing, since it is fault tolerant and can recover from transient faults. We have also seen that the stabilization time depends primarily on the diameter of the network but also increases with a large number of nodes. We believe that our work via evaluation validates the analytic proofs provided in [11].

Bibliography

[1] Cluster coordination. https://wiki.onosproject.org/display/ONOS/Cluster+Coordination#ClusterCoordination-NodeMastershipLifecycle. Accessed: 2018-07-03.

[2] A floodlight extension for supporting a self-stabilizing in-band control plane for software defined networking. Accessed: 2018-07-10.

[3] iperf, the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/. Accessed: 2018-07-07.

[4] Mininet official website. http://mininet.org/overview/. Accessed: 2018-07-10.

[5] Open viritual switch. http://www.openvswitch.org/. Accessed: 2018-06-25.

[6] Openflow switch specification. https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf. Accessed: 2018-06-25.

[7] What is openflow. https://www.sdxcentral.com/sdn/definitions/what-is-openflow/. Accessed: 2018-07-10.

[8] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[9] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid. Renaissance: A self-stabilizing distributed sdn control plane. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 233–243, July 2018.

[10] Marco Canini, Iosif Salem, Liron Schiff, Elad M Schiller, and Stefan Schmid. A self-organizing distributed and in-band sdn control plane. In *2017 IEEE 37th*

International Conference on Distributed Computing Systems (ICDCS), pages 2656–2657. IEEE, 2017.

[11] Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid. Renaissance: Self-stabilizing distributed SDN control plane. *CoRR*, abs/1712.07697, 2017.

[12] Shlomi Dolev. *Self-stabilization*. MIT press, 2000.

[13] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M Schiller. Practically-self-stabilizing virtual synchrony. *Journal of Computer and System Sciences*, 96:50–73, 2018.

[14] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks. In *Symposium on Self-Stabilizing Systems*, pages 133–147. Springer, 2012.

[15] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[16] SDN Floodlight. Openflow controller. *Web: https://github.com/floodlight/floodlight*, 2(1).

[17] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys & Tutorials*, 16(4):2181–2206, 2014.

[18] Linux man page. Ping, man page, 2018.

[19] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[20] Radia Perlman. *Interconnections: bridges, routers, switches, and internetworking protocols*. Pearson Education India, 1999.

[21] Liron Schiff, Stefan Schmid, and Marco Canini. Medieval: Towards a self-stabilizing, plug & play, in-band sdn control network.

[22] Liron Schiff, Stefan Schmid, and Marco Canini. Ground control to major faults: Towards a fault tolerant and adaptive sdn control network. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 90–96. IEEE, 2016.

A

Appendix 1

- B4, Googles SDN network
  https://people.eecs.berkeley.edu/ sylvia/cs268-2014/papers/b4-sigcomm13.pdf
  Amount of nodes: 12
  Diameter: 5

- Clos datacenter network
  http://ccr.sigcomm.org/online/files/p63-alfares.pdf
  Amount of nodes: 20
  Diameter: 4

- Telstra Backbone
  http://www.cs.umd.edu/~nspring/talks/sigcomm-rocketfuel.pdf
  Amount of nodes: 57
  Diameter: 8

- Rocketfuel AT&T
  http://research.cs.washington.edu/networking/rocketfuel/
  Amount of nodes: 96
  Diameter: 9

- Rocketfuel EBONE
  http://research.cs.washington.edu/networking/rocketfuel/
  Amount of nodes: 172
  Diameter: 10